UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA
Departamento de Arquitectura de Computadores y Automática

TESIS DOCTORAL

Emplazamiento de objetos de datos dinámicos sobre organizaciones de memoria heterogéneas en sistemas empotrados

Placement of dynamic data objects over heterogeneous memory organizations in embedded systems

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Miguel Peón Quirós

Directores

José Manuel Mendías Cuadros
Francky Catthoor

Madrid, 2016

UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA

Departamento de Arquitectura de Computadores y Automática

TESIS DOCTORAL

Emplazamiento de Objetos de Datos Dinámicos sobre Organizaciones de Memoria Heterogéneas en Sistemas Empotrados

Placement of Dynamic Data Objects over Heterogeneous Memory Organizations in Embedded Systems

Miguel Peón Quirós

Directores

José Manuel Mendías Cuadros
Francky Catthoor

Madrid, Septiembre de 2015

UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA

Departamento de Arquitectura de Computadores y Automática

TESIS DOCTORAL

Emplazamiento de Objetos de Datos Dinámicos sobre Organizaciones de Memoria Heterogéneas en Sistemas Empotrados

Placement of Dynamic Data Objects over Heterogeneous Memory Organizations in Embedded Systems

MEMORIA PARA OPTAR AL GRADO DE DOCTOR
PRESENTADA POR

Miguel Peón Quirós

Directores

José Manuel Mendías Cuadros
Francky Catthoor

Madrid, Septiembre de 2015

Typeset by the author with PDFLaTeX 2ε

Emplazamiento de Estructuras de Datos Dinámicas sobre Organizaciones de Memoria Heterogéneas en Sistemas Empotrados

Memoria presentada por Miguel Peón Quirós para optar al grado de Doctor por la Universidad Complutense de Madrid, realizada bajo la dirección de José Manuel Mendías Cuadros (Universidad Complutense de Madrid) y Francky Catthoor (IMEC, Bélgica):

Placement of Linked Dynamic Data Structures over Heterogeneous Memory Organizations in Embedded Systems

Dissertation submitted by Miguel Peón Quirós to the Complutense University of Madrid in partial fulfilment of the requirements for the degree of doctor of philosophy, work supervised by José Manuel Mendías Cuadros (Universidad Complutense de Madrid) and Francky Catthoor (IMEC, Belgium):

Madrid, 2015.

Acknowledgments

As Francky used to say: A PhD was meant to be the work of an individual, but this is probably not a good model anymore. I can say he was right because this work would not have been possible without the selfless and sincere help of many other people. I would like to thank all of them here. First, Αλέξανδρος Μπάρτζας for tirelessly working with me for unending hours... And then, unending years. Stylianos Mamagkakis (Stelios) helped us a lot at the beginning of our PhDs, and then some more. He shared with us many valuable ideas, but mainly he became a great friend. Dimitrios Soudris, who was Alexis' advisor, always supported us both during our joint work. He continued supporting me even after it was reasonable to expect. David Atienza deserves special mention as he granted me invaluable support and counsel during the whole of my PhD, especially at its beginnings.

My PhD advisors deserve also special acknowledgment. They supported me through many more years than was reasonable for a PhD to be completed. Francky Catthoor has never failed to give me valuable and prompt advice on any matters, always willing to counsel on anything. José Manuel Mendías ("Mendi") has supported me unconditionally through all these years. In fact, he is the person who made with his support this PhD thesis possible. He is probably one of the most patient people I have ever known – as the saying goes in Spanish: "He is more patient than Job." I could not count the hours he has spent discussing with me or reviewing my work, never putting it down even if it was just a bunch of silly ideas.

I have known a lot of nice people along the years. From the people I met while working as a teaching assistant at UCM (Nacho, Javi, Marcos, Christian, Sara, Pablo, Fran, Alberto, Carlos, Guillermo, José Luis, Sonia, Guadalupe,...), to the nice people I met at IMEC (Anne! a.k.a. "German girl," Willy, Ruth, Lidia, Christophe,...) or later at IMDEA (too many to name all of you! Kirill, George, Evgenia, José Félix, Dejan, Antonio,...). They all offered me valuable support, ideas and (most dear) friendship.

I would also like to thank the people who have worked on my education along the years, from elementary school to the grad school. All of them contributed with their effort and special interest to my education.

Finally, I want to thank those who have been my friends along the years: JaBa, Antonio, Noe, Catu, Rafa, Eva, Iván (a.k.a. "Eso es fácil"), Nadia,... And to my (big!) family for their continuing support. To my too-soon deceased uncle Javi, who was one of the most interesting people you could ever hope to talk with. To my brother; may we remain together for long years. Specially Nati, who has lovingly supported me through most of my PhD years.

To all of you: Thanks!

A mis padres

Abstract

The most extended computational model nowadays is based on processing elements that read both their instructions and data from storage units known as "memories." However, it was soon clear that the amount of data that could be stored in the memories was increasing at a much faster pace than their access speed. Worse, processor speeds were increasing even faster, causing them to stall and wait for the arrival of new instructions and data from the memories. This problem was named as the "memory wall" and has been one of the main worries of computer architects for several decades. A fundamental observation is that, given a particular silicon technology, access speed decreases as memory size increases. Thus, the idea of using several memories of varying sizes and access speeds appeared soon, giving birth to the concept of memory hierarchy. The presence of memories with different sizes and characteristics leads inevitably to the problem of placing the correct data into each memory element.

The use of a cache memory, that is, a small and fast memory that holds a copy of the most commonly used data set from a larger memory, has been the preferred option to implement the concept of memory hierarchy because it offers a plain view of the memory space to the programmer or compiler. The key observation that enabled the cache memory is access locality, both temporal and spatial. In general, cache memories have introduced a significant improvement on data access speeds, but they represent an additional cost in terms of hardware area (both for the storage needed to hold replicated data and for the logic to manage them) and energy consumption. These are substantial issues, especially in the realm of embedded systems with constrained energy, temperature or area budgets.

Several techniques have been proposed over the years to improve the performance of the memory subsystem using application-specific knowledge, in contrast to the generic, albeit transparent, work of the cache memory. In general, the main idea of these approaches is that the software controls the data movements across the elements in the memory hierarchy, often with the help of a Direct Memory Access (DMA) engine. These movements can be introduced implicitly by the compiler, or the programmer can include explicit commands for the DMAs. While these techniques eliminate the overheads introduced by the cache controller logic, they still require a high access locality to amortize the energy cost of those movements.

Another important milestone, now in the software realm, was the introduction of the dynamic memory because it enabled the software to adapt to changing conditions such as the number and size of the inputs. However, its utilization – particularly in the case of linked structures – can reduce data locality significantly, hence establishing a potentially adverse in-

teraction between the hardware mechanism of cache memory and the software mechanism of dynamic memory that requires specific solutions.

In this work, I present and argue the thesis that for applications that use dynamic memory and have a low data access locality, a specifically tailored data placement can report significant energy savings and performance improvements in comparison with traditional cache memories or other caching techniques based on data movements. This approach introduces two main questions. First, what is the mechanism that will implement the placement – I propose using the dynamic memory manager itself. Second, what information will be available to make decisions. The interface offered by the usual programming languages does not include enough information to allow the dynamic memory manager to produce an efficient data placement. Therefore, I propose to extend that interface with the data type of the objects under creation. In this way, the dynamic memory manager will be able to use prior knowledge about the typical behavior of the instances of the corresponding data type.

However, if a resource (or part thereof) is reserved exclusively for the instances of one data type, chances are that it will undergo periods of underutilization when only a small number of instances of that data type are alive. To limit the impact of that possibility, I propose to make a preliminary analysis that identifies which data types can be grouped together in the same memory resources, without significant detriment to system performance. Data types can be placed together if their instances are accessed with similar frequency and pattern – so that, for all practical purposes, which instances are placed in a given resource becomes indifferent – or created during different phases of the application execution – hence, they do not compete for space.

I support the previous claims with a functional tool, $\mathcal{DynAsT}$, to improve the placement of dynamic data objects over the memory subsystem of embedded systems. Despite its simplicity – it currently implements a set of simple algorithms and heuristics – $\mathcal{DynAsT}$ shows the promising results that can be attained with a data placement that takes into account the characteristics of the application data types. Conveniently, this tool can be used to improve the placement of dynamic data structures on the memory subsystem of an existing platform, or to steer the definition of the platform itself according to the particular needs of future applications.

As an additional contribution, I present a method for the systematic characterization of the applications' dynamic-data access behavior. The generated metadata can serve as a central repository that consecutive optimization techniques will use to operate on an application in a structured way. I also explain how this concept was used to link several optimization tools in a practical case.

Resumen

El modelo computacional más extendido hoy en día se basa en unidades de procesamiento que leen tanto sus instrucciones como sus datos desde elementos de almacenamiento conocidos como «memorias». Sin embargo, pronto se hizo evidente que la cantidad de datos que podían almacenarse en las memorias se incrementaba a un ritmo mayor que la velocidad de acceso a los mismos. Peor aún, la velocidad de los procesadores se incrementaba incluso más rápido, por lo que éstos debían pararse y esperar a que llegasen nuevas instrucciones y datos desde las memorias. Este problema, conocido como «la barrera de la memoria», ha sido una de las mayores preocupaciones de los arquitectos de computadores durante décadas. Una observación fundamental es que, dada una tecnología de silicio concreta, la velocidad de acceso a los datos se reduce según el tamaño de la memoria aumenta. De este modo, la idea de utilizar varias memorias de distintos tamaños (y, por tanto, velocidades de acceso) no tardó en aparecer, dando lugar al nacimiento del concepto de jerarquía de memoria. La presencia de memorias de diversos tamaños y características lleva inevitablemente al problema de emplazar los datos correctos en cada elemento de memoria.

La «memoria caché», es decir, una memoria pequeña y rápida que almacena una copia del subconjunto más accedido de los datos contenidos en otra memoria mayor, ha sido la opción preferida para implementar el concepto de jerarquía de memoria porque ofrece al programador o compilador una visión uniforme del espacio de memoria. La observación clave que posibilitó el diseño de la memoria caché es la localidad en el acceso a los datos, tanto temporal como espacial. Aunque las memorias caché han mejorado significativamente la velocidad de acceso a los datos, suponen un coste adicional debido al área ocupada por el hardware (réplica de los datos y lógica de gestión) y a su consumo de energía. Ambos son problemas relevantes, especialmente en el ámbito de los sistemas empotrados con restricciones de consumo energético, temperatura o área.

A lo largo de los años se han propuesto diversas técnicas para mejorar el rendimiento del subsistema de memoria mediante la utilización de conocimiento específico sobre las aplicaciones, en contraste con el funcionamiento genérico, aunque transparente, de la memoria caché. La idea general es que el software controle los movimientos de datos entre elementos de la jerarquía de memoria, a menudo con la ayuda de un controlador de acceso directo a memoria. Si bien estas técnicas eliminan los sobrecostes introducidos por el controlador de la memoria caché, también requieren una alta localidad de accesos para compensar el coste energético de tales movimientos.

Otro hito relevante, esta vez en el ámbito del software, fue la introducción de la memoria dinámica porque permitió a éste adaptarse a condiciones cambiantes, por ejemplo, en el núme-

ro y tamaño de las entradas. Sin embargo, su uso, especialmente en el caso de las estructuras dinámicas de datos (EDDs), puede reducir la localidad de los accesos a datos en la que se basan las memorias caché.

En este trabajo defiendo la tesis de que, para aplicaciones que usen memoria dinámica y presenten una baja localidad en sus accesos a datos, un emplazamiento de datos a medida (pero no necesariamente manual) puede comportar ahorros de energía y aumentos de rendimiento significativos en comparación con las memorias caché tradicionales u otras técnicas basadas en movimientos de datos. Existe, sin embargo, una condición: Debe preservarse una alta explotación de recursos.

Si se asigna un recurso (o fracción del mismo) en exclusiva a los ejemplares de una EDD, puede suceder que durante ciertos periodos no haya suficientes ejemplares activos de la misma y el recurso se desaproveche. Para limitar el impacto de esta situación, un análisis previo de las EDDs determina, en base a una caracterización («profiling») en tiempo de diseño, cuáles se pueden agrupar en un mismo recurso de memoria sin grandes perjuicios para el rendimiento del sistema. Así, se emplazarán juntas las EDDs que, o bien sean muy parecidas, con lo que resulte indiferente que se asigne el espacio a ejemplares de una o de la otra, o bien no presenten grandes conflictos entre ellas porque la mayoría de sus ejemplares se creen en fases distintas de la ejecución. Este agrupamiento es una solución de compromiso que permite proveer espacio dedicado para los ejemplares de las EDDs más accedidas de la aplicación, pero limitando el desaprovechamiento de recursos. El rendimiento del sistema y su consumo energético mejoran porque los recursos más eficientes son utilizados para los datos más críticos.

Este planteamiento introduce dos interrogantes. El primero es cuál será el mecanismo mediante el que se realice el emplazamiento. Para ello, propongo emplear el propio gestor de memoria dinámica. El segundo es qué información estará disponible para tomar las decisiones. La interfaz que ofrecen los lenguajes de programación más habituales no proporciona información suficiente para implementar el emplazamiento. Por tanto, propongo extender esta interfaz, mediante la instrumentación ya utilizada durante la caracterización inicial, para que incluya el tipo de los objetos que se vayan a crear. Así, el gestor de memoria dinámica conocerá el comportamiento típico de los ejemplares de la EDD correspondiente al elegir el recurso de memoria en el que deba alojarse cada nuevo objeto.

Las propuestas anteriores se sustentan mediante la implementación de una herramienta, *DynAsT*, que incluye un completo simulador de organizaciones de memoria y es utilizada en varios casos de estudio con claras mejoras en comparación con las tradicionales memorias caché. Esta herramienta puede utilizarse tanto para mejorar el emplazamiento de las EDDs en una plataforma existente, como para dirigir el proceso de diseño de la propia plataforma según las necesidades particulares de las futuras aplicaciones.

Como contribución adicional, presento un método para la caracterización sistemática de los accesos a datos dinámicos de las aplicaciones. Los metadatos generados en este proceso podrían actuar como el repositorio central que sucesivas herramientas de optimización utilizasen para trabajar sobre una aplicación coordinadamente. Esta información es explotada para enlazar diferentes herramientas de optimización en un caso práctico.

Short table of contents

Detailed table of contents

List of Figures

List of Tables

List of Acronyms

API	Application Program Interface
ARM	ARM Holdings plc
ASIC	Application-Specific Integrated Circuit
AVL	Self-balancing binary search tree invented by Adelson-Velskii and Landis
CRC	Cyclic Redundancy Check
DDR	Double Data Rate
DDT	Dynamic Data Type
DM	Dynamic Memory
DMA	Direct Memory Access
DMM	Dynamic Memory Manager
DRAM	Dynamic Random Access Memory
DRR	Deficit Round Robin
DSP	Digital Signal Processor
FPB	Frequency of accesses Per Byte
FPGA	Field Programmable Gate Array
HW	Hardware
ID	Identifier assigned to a dynamic data type during instrumentation
IP	Internet Protocol
JEDEC	JEDEC Solid State Technology Association
LRU	Least Recently Used
MMU	Memory Management Unit
NUMA	Non-Uniform Memory Access
OS	Operating System
QoS	Quality of Service
SCM	Storage-Class Memory
SDRAM	Synchronous Dynamic Random Access Memory
SoC	System on a Chip
SPM	Scratchpad Memory
SRAM	Static Random Access Memory
SW	Software
TCP	Transmission Control Protocol
WSC	Warehouse-Scale Computer

Conventions

The International Electrotechnical Commission (IEC) encourages the use of the binary prefixes (e.g., KiB) to differentiate between the SI ones used in fields such as Physics ($k = 10^3$) and those traditionally used in Computer Engineering ($K = 2^{10} = 1024$). These recommendations have been followed in fields such as Communication Technology, where the standard use is $1\,kB = 10^3\,B$. However, the old prefixes are still widely used when referring to capacities of silicon memories or processor addressing. Therefore, in this text I shall use the prefixes KB (uppercase 'K'), MB and GB with the sense of 2^{10} B, 2^{20} B and 2^{30} B, respectively. The following table summarizes these conventions:

SI	**Binary (IEC/SI)**	**This text**
$k = 10^3$	$Ki = 2^{10}$	$K = 2^{10}$
$M = 10^6$	$Mi = 2^{20}$	$M = 2^{20}$
$G = 10^9$	$Gi = 2^{30}$	$G = 2^{30}$

Chapter 1

Introduction

— Challenges to the improvement of computer performance

The performance of computing systems has experienced an exponential increase over the last decades. However, it seems clear that today computer performance is at a crossroads. The technological and architectural advances that have sustained those performance improvements up to now seem to be nearly exhausted. This situation can be traced to three main issues [ABC+06]:

- The power wall.
- The ILP wall.
- The memory wall.

The power wall owes to the growing difficulties in reducing the energy consumption of transistors on par with their physical dimensions. The power dissipation per area unit is not (almost) constant anymore. Therefore, we can now put more transistors in a chip than we can afford to power at once. Either some of them are turned down at different times, or their operating frequency has to be kept under a limit. Techniques such as voltage-frequency scaling or aggressive clock gating have been applied to mitigate its effects. The ILP (Instruction-Level Parallelism) wall means that adding more hardware to extract more parallelism from single-threaded code has diminishing results. Therefore, during the last years the focus has turned to extract the parallelism present at higher levels of abstraction: Thread, process or request-level parallelism. Finally, the memory wall is due to the fact that processor speeds have increased at a much higher rate than memory access speeds. As a consequence, the execution speed of many applications is dominated by the memory access time. This work focuses on the memory wall for data accesses.

— The memory wall

The most common computational model nowadays is based on processing elements that read both their instructions and data from storage units known as "memories." Soon it was observed that the amount of data that could be stored in the memories was increasing at a faster pace than their access speed. But the real problem was that processor speeds were increasing even faster: Bigger problems could now be solved and hence, more data needed to be accessed.

As the discrepancy between processor and data access speeds widened, processors had to stall and wait for the arrival of new instructions and data from the memories more often. This problem, known as the "memory wall," has been one of the main worries of computer architects for several decades. Its importance was outlined by Wulf and McKee [WM95] in an effort to draw attention and foster innovative approaches.

— Memory hierarchy and data placement

A fundamental observation is that, given a particular silicon technology, data access speed reduces as memory size increases: Bigger memories have decoders with more logic levels and a larger area means higher propagation delays through longer wires. Additionally, SRAM technology, which has been traditionally employed to build faster memories, requires more area per bit than DRAM. This has the effect that integrating bigger SRAMs increases significantly the chip area, and therefore also its final price.

Those factors led to the idea of combining several memories of varying sizes and access speeds, hence giving birth to the concept of memory hierarchy: A collection of memory modules where the fastest ones are placed close to the processing element and the biggest (and slowest) are placed logically – and commonly also physically – further from it. As suggested by Burks et al. [BGN46], an ideal memory hierarchy would approximate the higher speed of expensive memory technologies that can only be used to build small memories while retaining the lower cost per bit of other technologies that can be used to build larger, although slower, memories.

The presence of memories with different sizes and access characteristics leads inevitably to the following question: *Which data should be placed into each memory element to obtain the best possible performance?*

— Cache memory for HW-controlled data placement

The use of a cache memory, that is, a small and fast memory that holds the most commonly used subset of data or instructions from a larger memory, has been the preferred option to implement the concept of memory hierarchy.[1] With its introduction, computer architecture provided a transparent and immediate mechanism to reduce memory access time by delegating to the hardware the choice of the data objects that should be stored at each level in the memory hierarchy at any given time.

The key observation that led to the design of the cache memory is access locality, both temporal and spatial: Data (or instructions) that have been accessed recently have a high probability of being accessed again quite soon and addresses close to that being accessed at a given moment have a high chance of being accessed next.

In general, cache memories have introduced a significant improvement on data access speeds, but they represent an additional cost in terms of hardware area (both for the storage needed to hold replicated data and for the logic to manage them) and energy consumption. These are substantial issues, especially for embedded systems with constrained energy, temperature or area budgets.

[1] A preliminary design for a cache memory was presented by Wilkes [Wil65] under the name of "slave memory." He explained the use of address tags to match the contents of the "lines" in the slave memory with those in the main one and of validity/modification bits to characterize the data held in each line.

— SW-controlled data placement

The general-purpose logic of cache memories may not be suitable for all types of applications. To overcome that limitation, and/or to avoid their additional hardware costs, several software-controlled mechanisms have been proposed over the years. Those approaches exploit specific knowledge about the applications, in contrast to the generic, albeit transparent, work of the cache memory. They rely (usually) on a Direct Memory Access (DMA) engine or prefetching instructions in the processor to perform data movements across the elements in the memory hierarchy, which can be introduced implicitly by the compiler or explicitly by the programmer in the form of commands for the DMAs or prefetching instructions for the processor. The goal is to bring data closer to the processor before they are going to be needed, according to the intrinsic properties of each algorithm.

While those techniques may eliminate the overheads introduced by the cache controller logic, they still require a high locality in data accesses, especially to achieve significant energy savings. Bringing data that will be accessed just once to a closer memory may reduce the latency of bigger memories. However, each of these movements – a read from a far memory, a write to the closer cache and a final read from the cache when the processor actually needs the data – consumes some energy. For data elements that are not required by the processor, but that are nevertheless transferred as part of a bigger transaction (a cache line), this means a certain waste of energy – and some maybe harmless waste of bandwidth. These considerations give rise to a new question: *Can we reduce data access latency without sacrificing energy consumption?*

— Dynamic memory for greater flexibility

On a parallel track of events, the introduction of the dynamic memory (DM, originally known as "free storage" [MPS71, BBDT84]) in the software realm brought better adaptability to changes in the operating conditions, mainly in the number and size of the inputs: Instead of assigning a fixed size and location in the memory space (i.e., a memory address) for the variables and arrays of the application at design time, DM allows resolving these values at run-time. The binding between memory resources and data structures may thus be recalculated as the needs of the application evolve.

The new capabilities introduced by DM enabled the development of systems with more features, capable of more complex interactions with unknown orderings of events and, in general, more dynamic. In lieu of resorting to a worst-case allocation of resources, applications can rely on DM to handle such changing conditions.

— DM challenges: Allocation, placement and locality

Despite its many advantages, dynamic memory – particularly in the case of linked structures – introduces three new challenges. The most relevant is a reduction of data locality that establishes a potentially adverse interaction between the hardware mechanism of cache memory and the software mechanism of dynamic memory.

1. Allocation of data objects. The dynamic memory manager (DMM) is the software mechanism that handles the (de)allocation of blocks of varying sizes at run-time. Its services are

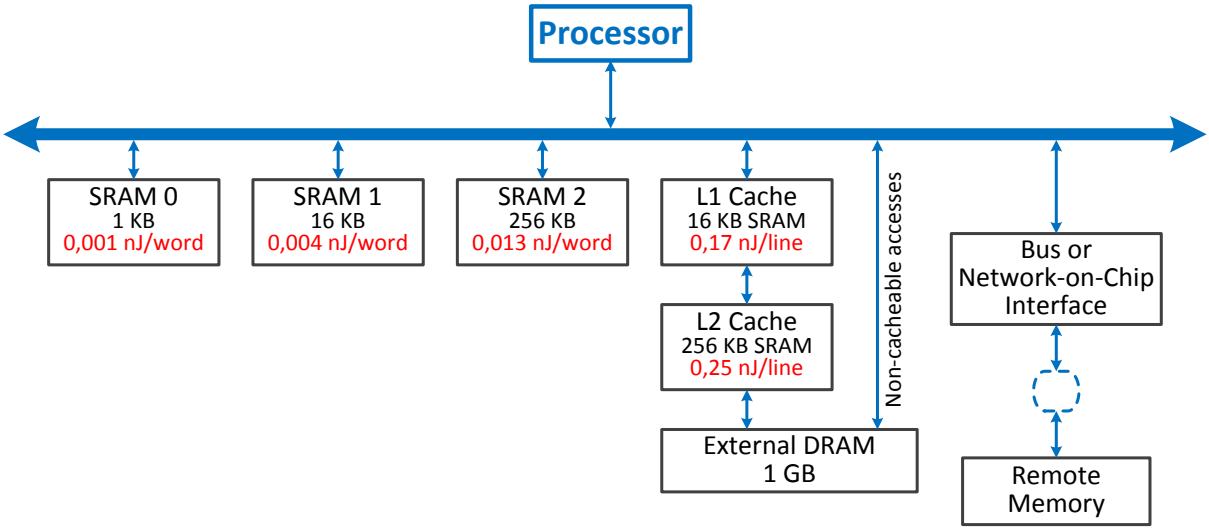


Figure 1.1.: Memory subsystem with several SRAM modules of varying sizes, external DRAM with a two-level cache hierarchy and Network-on-Chip connection to distant resources.

accessed through functions such as `malloc()` and `free()` in C or the `new` and `delete` operators in C++ or other object-oriented languages. Behind, physical memory is claimed from platform resources, usually with page-granularity, using standardized interfaces provided by the operating system (OS) such as `sbrk` and `mmap`, `VirtualAlloc` or similar ones. These solutions have evolved over decades to deal pretty efficiently with the allocation problem. A good overview of classic techniques is presented by Wilson et al. [WJNB95].

2. Placement of data objects. The use of complex memory organizations, such as the one presented in Figure 1.1, is common practice in the design of embedded systems. In contrast with the plain-view offered to programmers in general purpose systems, where hardware-controlled cache memories are transparent, the memory subsystem is in many cases part of the platform's programmer model. This is mainly because of the potential area and energy savings that can be achieved [BSL+02]: Caches use additional area to store address tags and power-hungry control circuitry, especially for higher degrees of associativity.

Even if due to other reasons, multicore desktop or server systems increasingly present a Non-Uniform Memory Architecture (NUMA) as well. Hence, system programmers have enough slack to make big improvements in this area (see the excellent report presented by Drepper [Dre07] or the documentation for GNU's libNUMA).

Those explicit memory organizations facilitate the utilization of an exclusive memory model, where the memories closer to the processor do not necessarily duplicate data from the bigger and slower ones; in comparison, cache hierarchies tend to favor an inclusive memory model where the smaller levels keep a subset of the data contained in the bigger ones. A consequence of a non-transparent memory subsystem is the need to decide the most appropriate physical location for each new allocation in such a way that the most accessed data reside in the most efficient memories.

3. Lower access locality. Dynamic memory objects are frequently associated in linked data structures where each object is related to one or several others through pointers (or references). A very frequent access pattern is then the traversal of the structure, either to access all the

nodes or to locate a particular one. However, *logically-adjacent linked nodes are not necessarily stored in consecutive memory addresses* (nodes may be added and deleted at any position in a dynamically linked structure), breaking the property of spatial locality. Moreover, in some structures such as trees, *the path taken can be very different from one traversal to the next one*, thus hindering also temporal locality. Therefore, the traversal of these dynamic data structures can have a devastating effect for those mechanisms that rely on the exploitation of data access locality by means of prefetching (soon to be used) and storing (recently used) data: Whole blocks of data are moved between memory elements even if they are used only to fetch a pointer (or at most for a brief update), with the consequent waste of energy. Even worst, many lines in a cache may be replaced during a traversal, leaving it in a "cold state" that will hurt the access time to the data previously stored in those lines.

An interesting consideration is that the repeated creation and destruction of objects with the same type does not necessarily imply the reuse of the same memory positions: The DMM is free to choose any available free block to satisfy a request – consider, for instance, a DMM using a FIFO policy to assign free blocks. When the application accesses these objects, many different cache lines are evicted as if the effective size of the application's working set were much bigger.

At this point the nature of the problem becomes clear: Classic dynamic memory solutions, originally devised to deal with all sorts of applications on desktop and server computers, take good care of finding a free memory block – but *any* memory block – for the requests as they arrive; as long as they are adequate for the requested allocation size, all blocks are considered the same regardless of their physical position in system memory. They are only concerned with the performance of the dynamic memory manager itself (the time it needs to find a suitable memory block), the memory overhead of its internal data structures and the problem of fragmentation – in its internal and external flavors. However, more complex memory organizations require that the characteristics of the physical memory chosen for an allocation match the pattern of accesses to the object that will be placed in that block. Previous techniques for static-data placement cannot help either because it is generally not possible to determine the size, position nor even the number of instances of a dynamic data type that will be created at run-time; hence, a fixed assignment is not possible. To put the icing on top of the cake, linked dynamic data structures can reduce data access locality and void the improvements of hardware and software mechanisms based on data movements.

— Energy consumption as a first-class citizen

The previous paragraphs illustrate how computer architects have focused traditionally on improving the performance of computers. Energy consumption was a secondary concern solved mainly by technological advances. Quite conveniently, Dennard scaling complemented Moore's law so that, as the number of transistors integrated in a microprocessor increased, their energy efficiency increased as well at an (almost) equivalent rate. I present a more elaborate discussion on this topic in Appendix B.

However, the first decade of the 2000's witnessed a shift in focus towards obtaining additional reductions in energy consumption. There are two main drivers for this renewed interest on energy efficiency: Mobile computing, supported by batteries, and the increasing concerns about energy consumption and cooling requirements (with their own energy demands) in

huge data centers. In this regard, several authors (for example, Koomey et al. [KBSW11]) attribute the emergence of mobile computing to the fact that energy efficiency has increased exponentially during the last sixty years (again, I cover this topic more in depth in Appendix B). This effect has enabled the construction of mobile systems supported by batteries and the apparition of a whole new range of applications with increasing demands for ubiquity and autonomy. In essence, computer architects have been able to trade off between improving performance at the same energy level or maintaining a similar performance, which was already good enough in many cases, with lower energy consumption.

In this way, the success of architectures such as the ones from ARM can be attributed in great part to their ability to operate at low power levels, even if their peak performance is not as high as that of traditional ones. This property is very useful for battery operated devices, but could also serve as a way to increase energy efficiency in data centers while exploiting request-level parallelism.

Energy consumption is a critical parameter for battery operated devices because optimizations in this area may be directly translated into extended operating times, improved reliability and lighter batteries. However, in addition to the total amount of energy consumed, the pace at which it is consumed, that is, power, is also relevant. For instance, battery characteristics should be different when a steady but small current needs to be supplied than when bursts of energy consumption alternate with periods of inactivity. Additionally, temporal patterns added to spatial variations in energy consumption may generate "hot spots" in the circuits that contribute to the accelerated aging and reduced reliability of the system [JED11a]. Furthermore, in order to limit system temperature, expensive (and costly) cooling solutions must be added to the design if instantaneous power requirements are not properly optimized.

Battery operated devices are present in such huge numbers that, even if their individual energy consumption is small, the global effect can still be relevant. On the bright side, every improvement can also potentially benefit vast numbers of units. Therefore, optimizing energy consumption on embedded systems is a fairly critical issue for today's computer architecture.

In a different realm, the shift towards server-side or cloud computing that we are currently experiencing has generated growing concerns about energy consumption in big data centers. Companies such as Google, Amazon, Microsoft or Facebook have built huge data centers that consume enormous amounts of energy: According to estimates by Van Heddeghem et al. [HLL+14], overall worldwide data center energy consumption could have been in the order of 270 TW h in 2012, that is, about 1.4 % of the total worldwide electricity production. The sheer amount of machines in those data centers exacerbates any inefficiencies in energy consumption. However, the good news is that all the machines are under the supervision and control of a single entity (i.e., the owning company), so that any improvements can also be applied to huge numbers of machines at the same time. In this regard, we have seen two interesting trends in the last years: First, as Barroso and Hölzle noticed [BH07], most servers are not energy proportional. That is, their most energy-efficient operating modes are not those with lower (or mild) computing loads, but those with the higher ones. However, as they point out, the typical data center server operates usually at low to mid load levels. This important mismatch is still reported in recent studies such as the ones from Barroso et al. [BCH13] and Arjona et al. [ACFM14], although the first one reports that CPU manufacturers (especially Intel) have done important efforts. Second, during the last years data center architects have speculated with the possibility of favoring simpler architectures. Although these architectures may have lower absolute performance, they typically achieve a higher density of processing

elements. This could make them better adapted to exploit request-level parallelism. But, most importantly, they could help to increase the energy efficiency of data centers.

— Energy consumption in the memory subsystem

The memory subsystem is a significant contributor to the overall energy consumption of computing systems, both in server/desktop and mobile platforms. The exact impact varies according to different measurements, platforms and technologies, but it is usually reported to be between 10 % and 30 %, with a more pronounced effect in data-access dominated applications such as media, networking or big-data processing. For example, Barroso and Hölzle [BH09] report that the share of DRAM on server peak-power[2] requirements in a Google datacenter is 30 %, while in their revised version [BCH13] it is reported as 11.7 % (15.2 % if we exclude the concepts of "power overhead" and "cooling overhead"). In both cases, the relative power demands of the DRAM compared to the CPU are roughly around 90.1 % and 27.8 %, respectively. For a smartphone platform, the measurements presented by Carroll and Heiser [CH10] show that the power required by the DRAM is roughly between 9.1 % and 115.4 % of the CPU power, depending on the configuration and the benchmark chosen.[3] Nevertheless, most of the previous studies consider only the energy consumed by external DRAM modules: They do not discriminate between the energy consumed in the microprocessors by the processing elements themselves (functional units, instruction decoders, etc.) and the integrated cache memories, which are clearly part of the memory subsystem. That means that the energy consumption of the complete memory subsystem may be considerably higher. Unfortunately, I have not been able to get an accurate disclosure of the energy consumed inside the CPU for a commercial platform (an estimate could still be produced for FPGA-based platforms through their static analysis tools, but an accurate dynamic measurement would be considerably more complex).

— Optimizations of the memory subsystem and the three walls

Any optimizations of the memory subsystem may also benefit the overall properties of the system at a global level (Figure 1.2), hence alleviating the pressure against the three walls:

- The memory wall is relieved, reducing access latency to the most accessed data objects and improving overall system performance.
- Reducing the time that the processing pipelines stall waiting for data to arrive eases the ILP wall because there is less need to extract parallelism from the instruction streams. As the performance improvements produced by ILP techniques such as out-of-order execution, speculation and SMT become less critical, the area used to implement these complex features can be exploited for other purposes. Alternatively, a smaller size or number of components can lead to more economic designs.
- The power wall is put off because most of the accesses use preferentially the memories with lower energy consumption per access, but, additionally, a reduced memory latency

[2] Although in this work I usually concern about energy consumption, most existing studies present averaged or peak power ratings instead. Unfortunately, the conversion between power and energy is not always straightforward, especially if peak values are presented.

[3] These data have been manually estimated from Figure 5 of the cited paper.

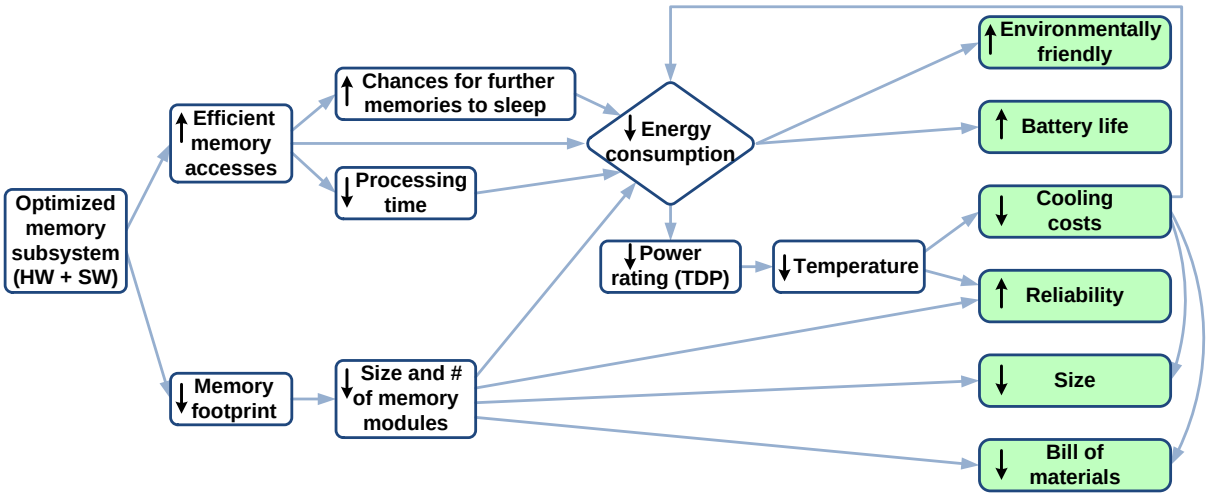


Figure 1.2.: Memory subsystem optimizations can create synergies through the whole system.

means that the processors receive their data faster and are able to finish the different tasks sooner with less wait states. The saved cycles may allow putting the processing elements into a deep sleeping mode with low energy consumption until the next batch of work arrives, saving additional energy. Even better, this lower energy consumption can create a "thermal slack" that can be exploited during short bursts of high performance needs.

— Optimization at the software and platform levels

The memory subsystem can be optimized at two different levels: At the system and application software level, and at the platform hardware level.

When the hardware design can be tailored, as is frequently the case in embedded systems, knowledge on the properties of the applications that will be executed may allow choosing the optimal type and size of the memory elements. This is especially compelling for modern Systems-on-Chip (SoC) that can mix different technologies in the same chip.

At the software level, algorithmic optimizations such as choosing algorithms with the lowest asymptotic cost can reduce the total number of accesses needed to perform a computation. Other optimizations such as the ones presented by Atienza et al. [AMP+04, AMM+06a] may lead to important reductions of the memory footprint (peak amount of allocated memory); hence, it may be possible to use smaller memories with lower energy consumption. Additional algorithmic optimizations may lead to data consolidation during phases of lower memory requirements, with the possibility of turning off some of the unused memory elements for an even lower energy consumption. These optimizations are general for any type of computing system. Additionally, software applications in embedded systems can frequently be tuned to adapt their behavior to the properties of the underlying hardware platform, so that the most accessed data objects are placed in the closest and most energy efficient memories. Other considerations such as whether data accesses are mostly sequential or random can also be considered when deciding the technology of the memory elements in which the data objects will be placed (e.g., a row-based DRAM, a sector-based FLASH or a truly random access SRAM).

— Why focus on embedded systems?

Embedded systems are assuming an important role in our world and it seems that they will be even more pervasive with the advent of ubiquitous computing. Sometimes their purpose is just to make our lives easier or funnier, but in many cases they hold fairly critical responsibilities. That is the case, for instance, of modern pacemakers, insulin pumps or the cochlear implant – a truly amazing achievement that may draw us closer to a better integration of physically impaired people.[4] Whether their failure is a matter of life and death or just a minor annoyance, embedded systems must meet strict requirements: They must be reliable, fast and efficient; they must use little energy and their cost must be as low as possible to make them affordable. For example, many security enhancements in cars are sold as high-end options and therefore, buyers may dispense with them to reduce costs.

Embedded systems present several additional characteristics that make them inviting targets for optimizations. First, many of them operate on batteries or have to meet temperature restrictions, hence their energy consumption and power rating are constrained. Second, most embedded systems present elaborate memory organizations that essentially convert them into Non-Uniform Memory Architecture (NUMA) machines. Finally, the designer has usually more control over the platform and applications running on those systems and so it is possible to apply more advanced techniques than on general purpose systems. On Chapter 7, I show that some of these considerations apply also to big data centers and how future work could provide optimizations on that realm.

— Embedded systems: Necessity and possibility of optimization

The higher need for optimization and the higher degree of control on the final design present in embedded systems go hand-on-hand. In general purpose systems such as a desktop computers, libraries and software components may be optimized for a general case but, too often, generalizations and trade-offs must be done because optimizing for a set of use cases may be counterproductive for a different one. If both sets are equally likely, then a trade-off solution is usually chosen. In this way, none of the use cases is executed in a very bad configuration, but frequently neither is executed in the optimal one.

By contrast, the applications and use cases of embedded systems may frequently be characterized in advance during the design phase and a particular solution, optimized for a particular problem, developed. However, the situation can be different for modern embedded systems that, without falling into the category of general purpose systems, need to react to a changing environment that conditions their behavior. These systems exhibit a moderate to high degree of dynamism in their behavior and are, in consequence, midway from classic, completely static, embedded systems and general purpose ones. However, these dynamic embedded systems may still be partially characterized at design time to prepare them for the most common use cases (Figure 1.3). Therefore, we can say in general that both the necessity and possibility for optimization are present in embedded systems at a higher degree than in general purpose computing systems.

[4] I acknowledge the existence of groups of people who think that this type of technology may actually widen the gap towards those individuals that for any reason do not use it. For example, by reducing the number of people who rely on sign-languages to communicate, these could be perceived as less necessary.

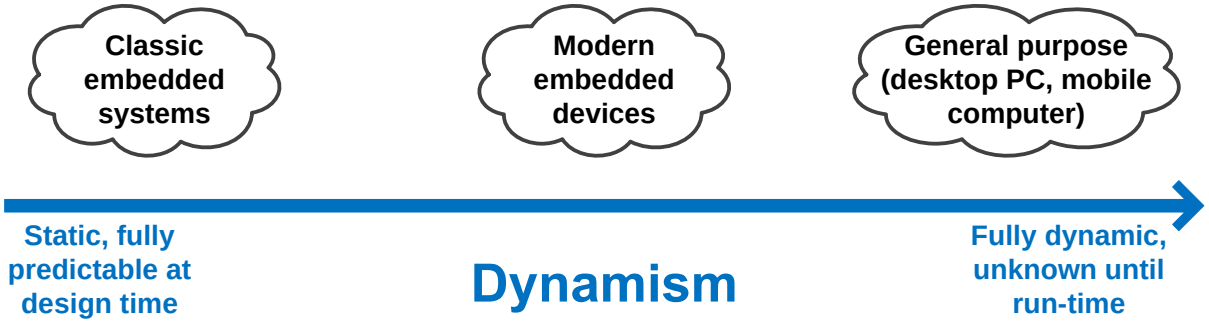


Figure 1.3.: Embedded systems with dynamic inputs can be *partially* characterized at design time, which allows optimizing their typical use cases.

— Goal: Data placement for improved performance and reduced energy consumption in embedded systems

In this work I consider optimizations of the memory subsystem at the data placement level that improve exploitation of platform resources in embedded systems. The benefits include increased performance, due to a lower data latency, and reduced energy consumption, due to a preference for memories with lower energy consumption and avoidance of data movements across elements in the memory subsystem.

1.1. Why are new solutions needed?

Cache memories are an excellent mechanism for the placement of data over several memories with different characteristics that, given the premise of sufficient spatial and temporal locality, have greatly contributed to the continuous increase in computer performance of the last decades alleviating the effects of the memory wall. More importantly, cache memories are completely transparent to the programmer, therefore fulfilling the promise of performance improvements from the side of computer architecture alone.

However, a relevant corpus of existing work [PDN00, BSL+02, KKC+04, VWM04, GBD+05] has shown that scratchpad memories (small, on-chip static RAMs directly addressable by the software) can be more energy-efficient than caches for static data if a careful analysis of the software applications and their data access patterns is done. With these techniques, the software itself determines explicitly which data must be temporarily copied in the closer memories, usually programming a DMA module to perform the transfer of the next data batch while the processor is working on the current one.

The disadvantage of software controlled scratchpad memories is that data placement and migration are not performed automatically by the hardware any longer. The programmer or the compiler must explicitly include instructions to move data across the elements of the memory subsystem. These movements are scheduled according to knowledge extracted during the design phase of the system, usually analyzing the behavior of the applications under typical inputs. This mechanism can provide better performance than generic cache memories, but is costly, platform dependent, relies on extensive profiling and requires an agent (compiler or programmer) clever enough to recognize the data access patterns of the application. Although significant effort was devoted to it, especially in academic environments during the early 2000's, this mechanism has never gained widespread adoption outside of very specific and resource-restricted embedded environments.

Both mechanisms, caches and software controlled scratchpads, are efficient for static data objects amenable to prefetching and with a good access locality. However, applications that confront variable cardinality, size or type of inputs usually rely on dynamic memory and specifically on dynamically linked data structures. Although cache memories can cope with dynamically allocated arrays efficiently, the access pattern to linked structures usually has a devastating effect on access locality, both spatial (thwarting the efforts of prefetching techniques) and temporal (diminishing reuse). Additionally, the dynamic memory mechanism prevents the exact placement in physical memory resources at design time that scratchpad-based techniques require because memory allocation happens at run-time, and the number of instances and their size is potentially unknown until that moment. Although these problems have been known for a long time, most of the previous works on dynamic memory management focused on improving the performance of the dynamic memory managers themselves, reducing the overhead of their internal data structures or palliating the effect of fragmentation, but ignoring the issues derived of placement. Even more important, very little effort has been devoted to the placement of dynamic data objects on heterogeneous memory subsystems.

Figure 1.4 illustrates the dependency of caches and scratchpads on temporal locality to amortize the energy cost of data movements over several accesses. First, on (a) the cost of reading a single word from main memory is depicted. In systems with a cache memory, the word will be read from memory and stored in the cache until the processing element accesses it. Even if data forwarding to the processor may save the read from the cache (and hide the access latency), the cost for the write into the cache is ultimately paid. A more complete situation is presented on (b), where a data word is read from main memory, stored in the cache (assuming no forwarding is possible), read from there by the processor and finally written back, first to the cache and then to the main memory. Energy consumption increases in this case because the cost of writing and reading input data from the cache or scratchpad, and then writing and reading results before posting them to the DRAM, is added to the cost of accessing the data straight from the DRAM. As a result, independently of whether they are performed by a cache controller or a DMA, a net overhead of two writes and two reads to the local memory is created without any reutilization payback. (c) Illustrates a quite unlucky situation that may arise if a word that is frequently accessed is displaced from the cache by another word that is only sporadically used – reducing the likelihood of this situation is the main improvement of associative caches, which are organized in multiple sets. Finally, (d) shows how allowing the processor to access specific data objects directly from any element in the memory subsystem can benefit overall system performance: Modifying a single data word consumes the energy strictly needed to perform a read and a write from main memory. Of course, these considerations would have been different if prefetching could have been applied successfully. Finally, (e) presents a potentially useful situation where a specific data placement is used to put the most frequently accessed data in a closer memory, while data that are seldom accessed reside in the main memory forcing neither evictions nor further data movements.

The fact that cache memories are not optimal for all situations is also highlighted by Ferdman et al. [FAK+12], who show that the tendency to include bigger caches in modern microprocessors can be counterproductive in environments such as warehouse-scale computers (WSCs). In those systems, a myriad of machines work together to solve queries from many users at the same time, exploiting request-level parallelism. The datasets used are frequently so big, and data reuse so low, that caches are continuously overflowed and they just introduce unnecessary delays in the data paths. A better solution might be to reduce the size and en-

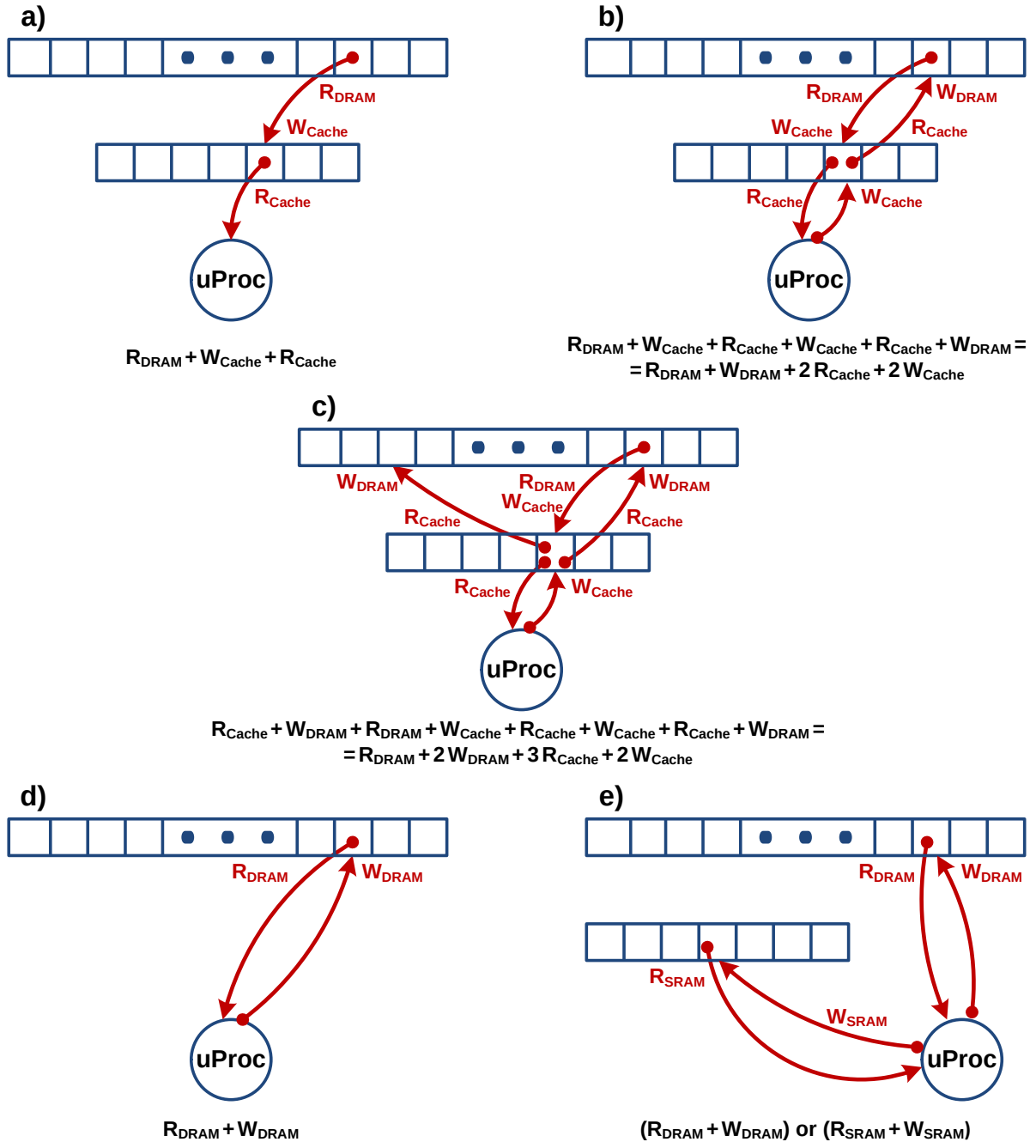


Figure 1.4.: Temporal locality is needed to amortize the cost of data movements across a hierarchical memory subsystem. a) One word is read only once by the processor. b) One word is read and modified by the processor; the result has to be eventually written back to main memory. c) Access to the new word evicts another one previously residing at the same position in the cache. d) Cost of modifying a word if the processing element can access it directly from main memory or e), from any element of the memory subsystem.

ergy consumption of the data caches; as a result, more processing elements could be packed together per area unit to increase the number of threads executed concurrently.

In summary, caches and scratchpads are not always the most appropriate mechanism for applications that make a heavy use of dynamic memory.

— A brief motivational example

As a more elaborate example of the potential struggles of cache memories with dynamic memory, I present here a small subset of the results of the experiment conducted in Section 4.4. The benchmark application uses a trie to create an ordered dictionary of English words, where each node has a list of children directly indexed by letters; it then simulates multiple user look-up operations. This experiment represents a case that is particularly hostile to cache memories because each traversal accesses a single word on each level, the pointer to the next child, but the cache has to move whole lines after every miss.

The performance of the application is evaluated on five different platforms. The reference one (labeled as "Only SDRAM") has just an SDRAM module as main memory. The other platforms add elements to this configuration:

- The platform labeled as "SDRAM, 256 KB Cache" has the SDRAM and a 256 KB cache memory with a line size of 16 words (64 B).
- Platform "SDRAM, 32 KB L1 Cache, 256 KB L2 Cache" has the SDRAM, a first-level 32 KB cache and a second-level 256 KB cache. Both caches have a line size of 16 words (64 B).
- Platform "SDRAM, 256 KB Cache (Line size=4)" has the SDRAM and a 256 KB cache memory with a line size of 4 words (16 B) – in contrast with the previous ones.
- Finally, platform "SDRAM, 256 KB SRAM" has the SDRAM and a 256 KB SRAM memory (also known as a "scratchpad") with an object placement tailored using the techniques presented in this work.

All the caches have an associativity of 16 ways.

Figure 1.5, shows the energy consumption (of the memory subsystem) on each platform, taking the one with only SDRAM as reference. The platform with the SRAM memory ("SDRAM, 256 KB SRAM") is the most efficient, roughly halving the most efficient cache-based platform ("SDRAM, 256 KB Cache (Line size=4)") and achieving important savings in comparison to the reference case.

The size of the cache line has an important impact on the energy consumption of the system: Because of its longer line size, which is a common feature in modern architectures, the penalty paid by platform "SDRAM, 256 KB Cache" due to the lack of spatial locality in the application is exacerbated up to the point that it would be more efficient to access the SDRAM directly. The use of a multilevel cache hierarchy (platform "SDRAM, 32 KB L1 Cache, 256 KB L2 Cache") does not help in this case; indeed, the additional data movements between the levels in the hierarchy incur an even higher penalty. The reason for this effect appears clearly in Figure 1.6: The number of accesses actually required by the application (platforms "Only SDRAM" and "SDRAM, 256 KB SRAM") is significantly lower than the number of accesses due to cache-line-wide data movements across elements in the memory hierarchy of the platforms with caches.

Interestingly, Figure 1.7 shows that, in general, the penalty on the number of cycles spent on the memory subsystem does not increase so dramatically. The reason is that cache memories are much faster than external DRAMs and the latency of some of the superfluous accesses can

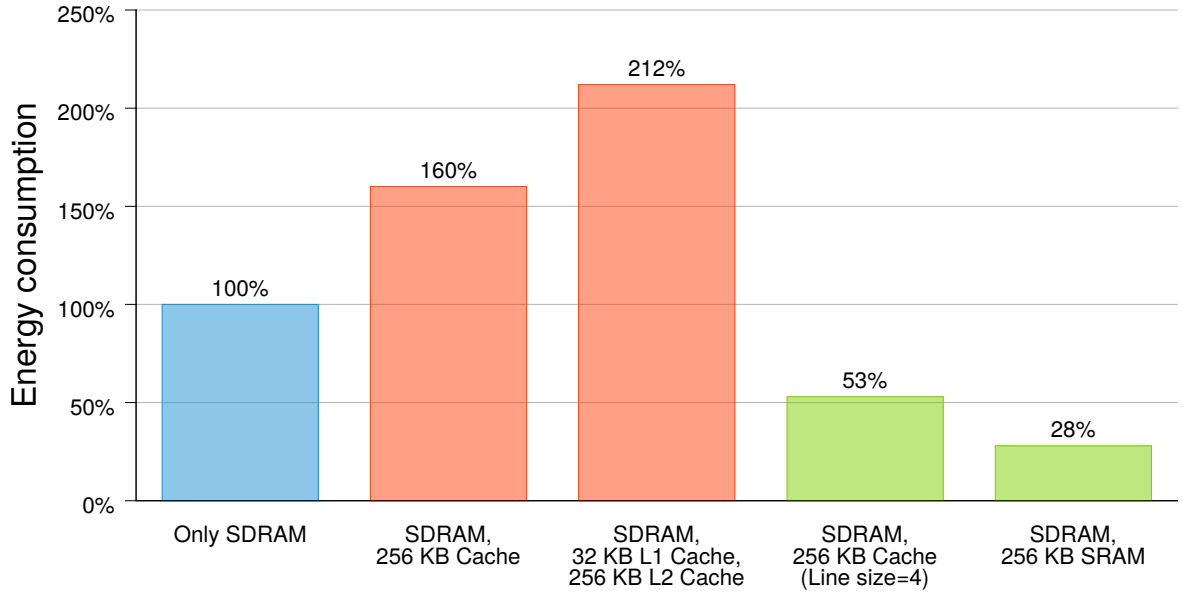


Figure 1.5.: Energy consumption in a DM-intensive benchmark. The results take as reference the energy consumption of the memory subsystem consisting only of an SDRAM module.

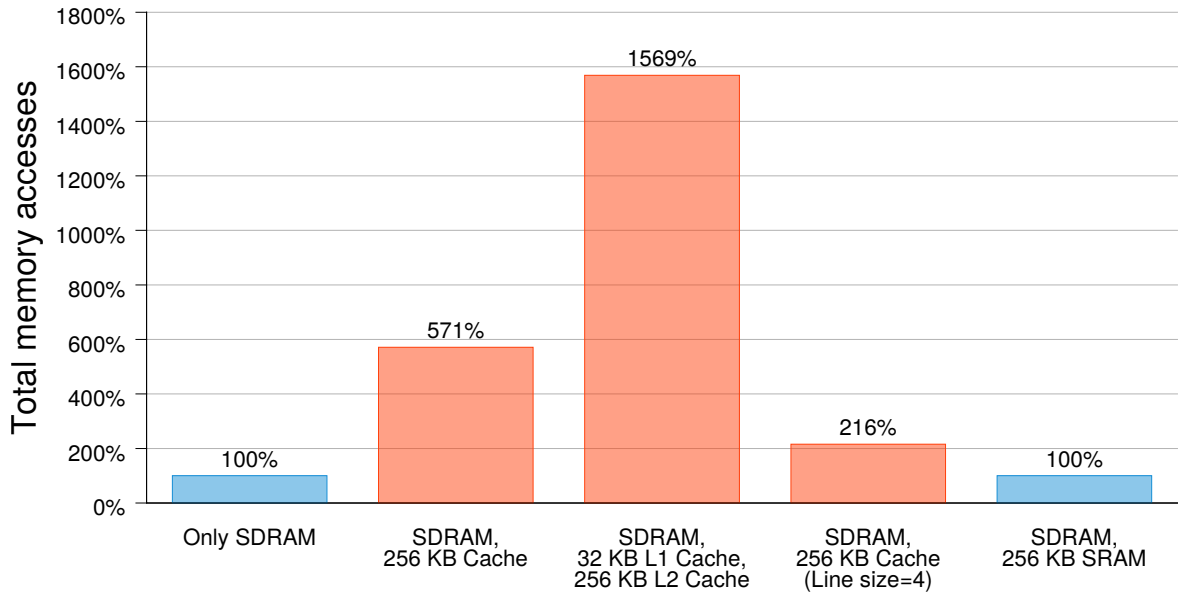


Figure 1.6.: Total number of memory accesses in a DM-intensive benchmark. The results take as reference the number of accesses in the memory subsystem consisting only of an SDRAM module.

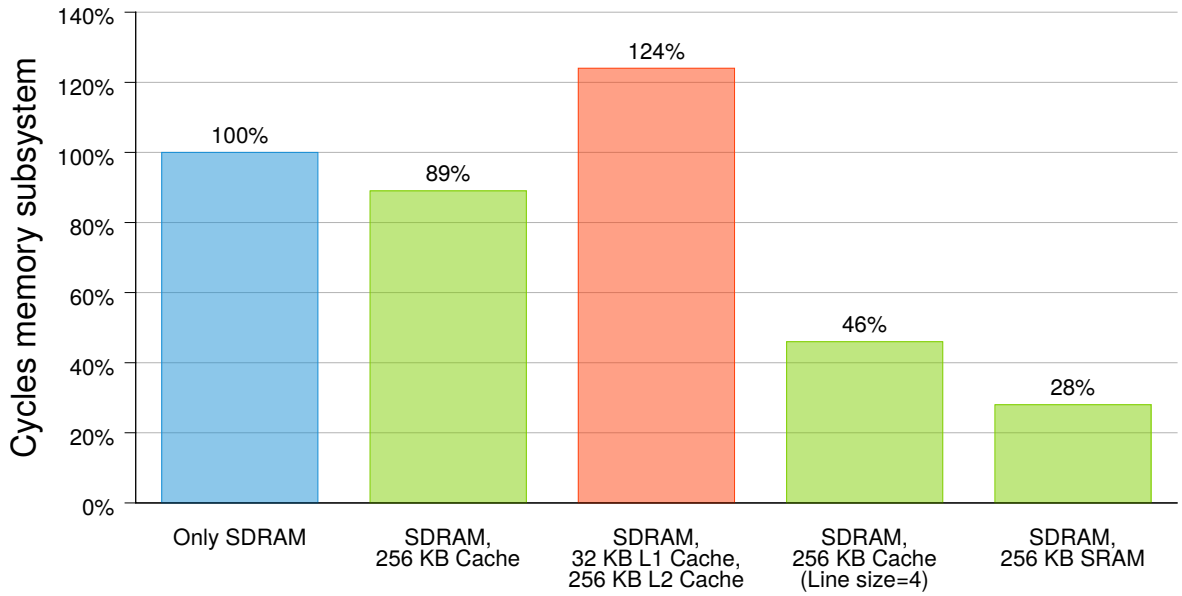


Figure 1.7.: Number of cycles spent on the memory subsystem in a DM-intensive benchmark. The results take as reference the number of cycles on the memory subsystem consisting only of an SDRAM module.

be hidden through techniques such as pipelining, whereas writing a line to a cache consumes a certain amount of energy regardless of whether all the elements in the line will be used by the processor or not. Energy consumption cannot be "pipelined."

To motivate also for the advantages of combining memories of varying sizes in heterogeneous memory subsystems, Figure 1.8 presents an additional experiment with the same application. Now, the reference platform ("SRAM: 512 KB, 4 MB") is configured with enough SRAM capacity to hold all the application dynamic data objects without external DRAMs. This platform has a big module of 4 MB and an additional one of 512 KB – the maximum footprint of the application is in the order of 4.3 MB. Several other configurations that include smaller memories are compared with the previous one. As the energy cost of accessing a word is generally lower for smaller memories, using two 256 KB memories instead of a single 512 KB may reduce energy consumption significantly. Furthermore, with appropriate techniques for dynamic data placement, the most accessed data objects can be placed on the memories that have a lower energy consumption. Thus, heterogeneous memory organizations may attain important savings in energy consumption with a small increase in design complexity.

In conclusion, the use of data-movement techniques with applications that make an important use of dynamic memory can degrade performance (even if pipelining and other techniques can mitigate this effect to some extent), but, more dramatically, increase energy consumption by futilely accessing and transferring data that are not going to be used. Computer architecture has traditionally focused on improving performance by increasing bandwidth but, especially, reducing latency whereas energy consumption was not so relevant. Now that we are struggling against the power wall and that mobile computing is gaining relevance, energy consumption is a new goal that deserves specific treatment by computer architects and everybody involved in the design of computing systems.

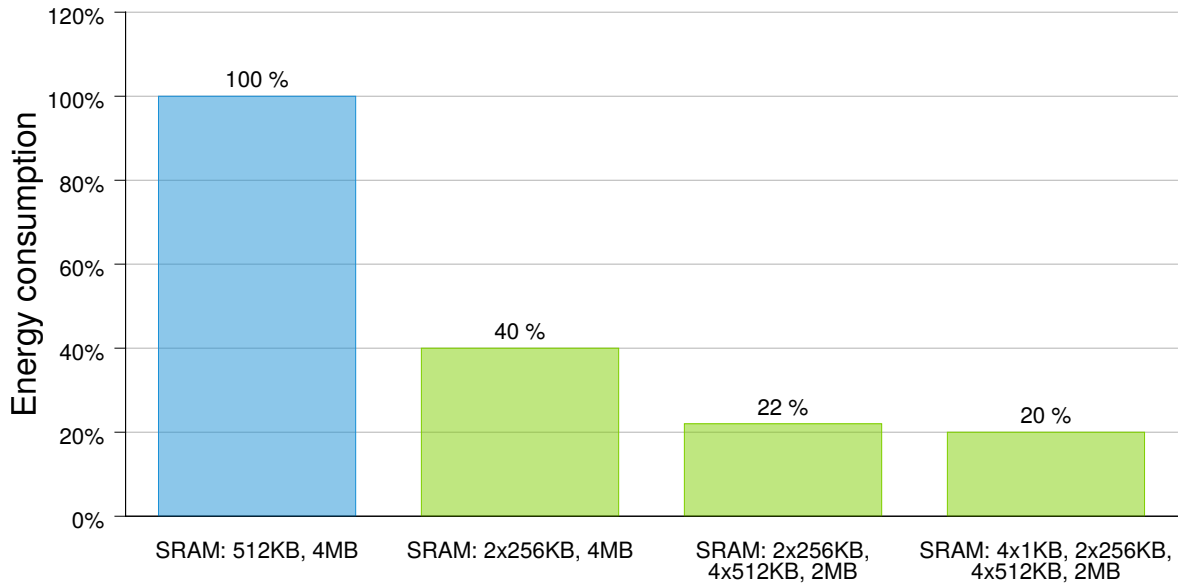


Figure 1.8.: Heterogeneous memory organizations may benefit from the lower energy consumption per access of smaller memories. Also, appropriate placement techniques can place the most accessed data objects on the most efficient memories.

1.2. Problem statement and proposal

In the previous pages I tried to bring forth the complexities that the use of dynamically allocated objects introduces into the placement problem, especially for generic solutions such as cache memories, but also for more specific solutions created to cope with the placement of static data objects using application-specific knowledge. Additionally, I explained why reducing energy consumption is now as important for computer architecture as improving performance was previously. Finally, I also motivated why embedded systems are a good subject choice for optimizations in those areas.

In this work, I present and argue the thesis that for applications that utilize dynamic memory and have a low data access locality, using a specifically tailored data placement that avoids or tries to minimize data movements between elements of the memory subsystem can report significant energy savings and performance improvements in comparison with traditional cache memories or other caching techniques based on data movements. This goal can be summarized as follows:

> *Given a heterogeneous memory subsystem and an application that relies on dynamic memory allocation, produce an efficient (especially in terms of energy consumption) placement without data movements of all the dynamically-allocated data objects into the memory elements that can be easily implemented at run-time by the operating system (Figure 1.9).*

This approach introduces two main questions. First, what is the mechanism that will implement the placement – I propose using the dynamic memory manager itself. Second, what information will be available to make decisions. The interface offered by the usual programming languages does not include enough information to allow the dynamic memory manager to produce an efficient data placement. Therefore, I propose to extend that interface with the data type of the objects under creation. In this way, the dynamic memory manager will be

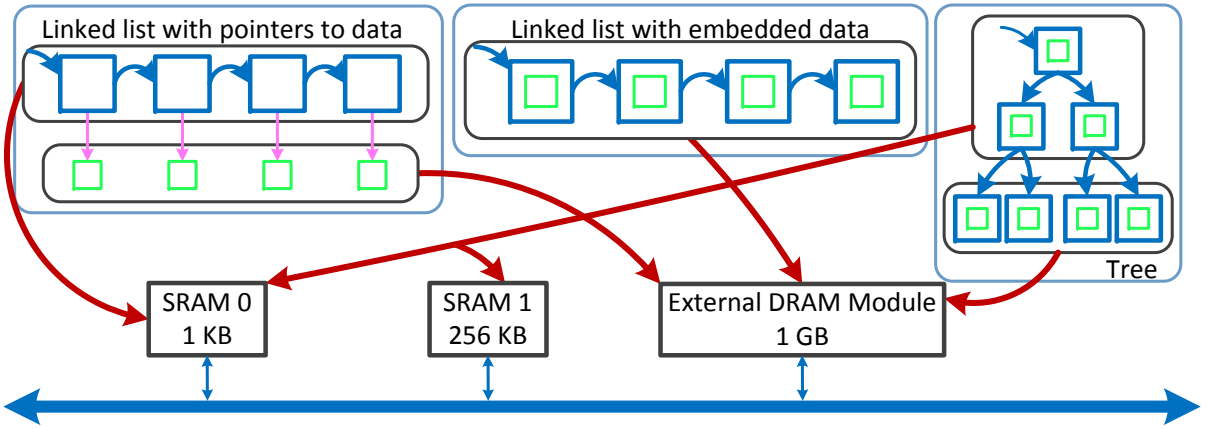


Figure 1.9.: The objective of the methodology presented in this work is to map the dynamic data objects of the application into the elements of the memory subsystem so that the most accessed ones take advantage of the most efficient resources.

able to use prior knowledge about the typical behavior of the instances of the corresponding data type.

Avoiding data movements creates a new challenge: If a resource (or part thereof) is reserved exclusively for the instances of one data type, chances are that it will undergo periods of underutilization when only a small number of instances of that data type are alive. To limit the impact of that possibility, I propose to make a preliminary analysis that identifies which data types can be grouped together in the same memory resources, without significant detriment to system performance. Data types can be placed together if their instances are accessed with similar frequency and pattern (so that, for all practical purposes, which instances are placed in a given resource becomes indifferent) or created during different phases of the application execution – hence, they do not compete for space.

1.2.1. Methodology outline

Figure 1.10 presents the global view of the methodology, whose goal is to produce an explicit placement of dynamic data objects on the memory subsystem. Data movements between memory modules are avoided because they are effective only in situations of high data-access locality – moving data that are going to be used just once is not very useful. Thus, the advantages of the methodology increase in cases where access locality is low, such as when dynamic memory is heavily used.

The most significant challenge for the methodology is balancing between an exclusive assignment of resources to DDTs and keeping a high resource utilization. The final goal is to improve system performance, but keeping the best resources idle will not help to increase it. My proposal, as discussed before, consists on performing an analysis of the DDTs to find out which ones can be combined and assigning resources to the resulting groups, not to isolated DDTs. Grouping helps in provisioning dedicated space for the instances of the most accessed DDTs, while keeping resource exploitation high. This preliminary analysis needs extensive information on the behavior of the DDTs to make the right choices.

Placement is implemented in the methodology by the dynamic memory manager which, to perform that extended role, requires additional information: The data type identifier of the objects being created. Given the previous requirements, the first step of the methodology is to

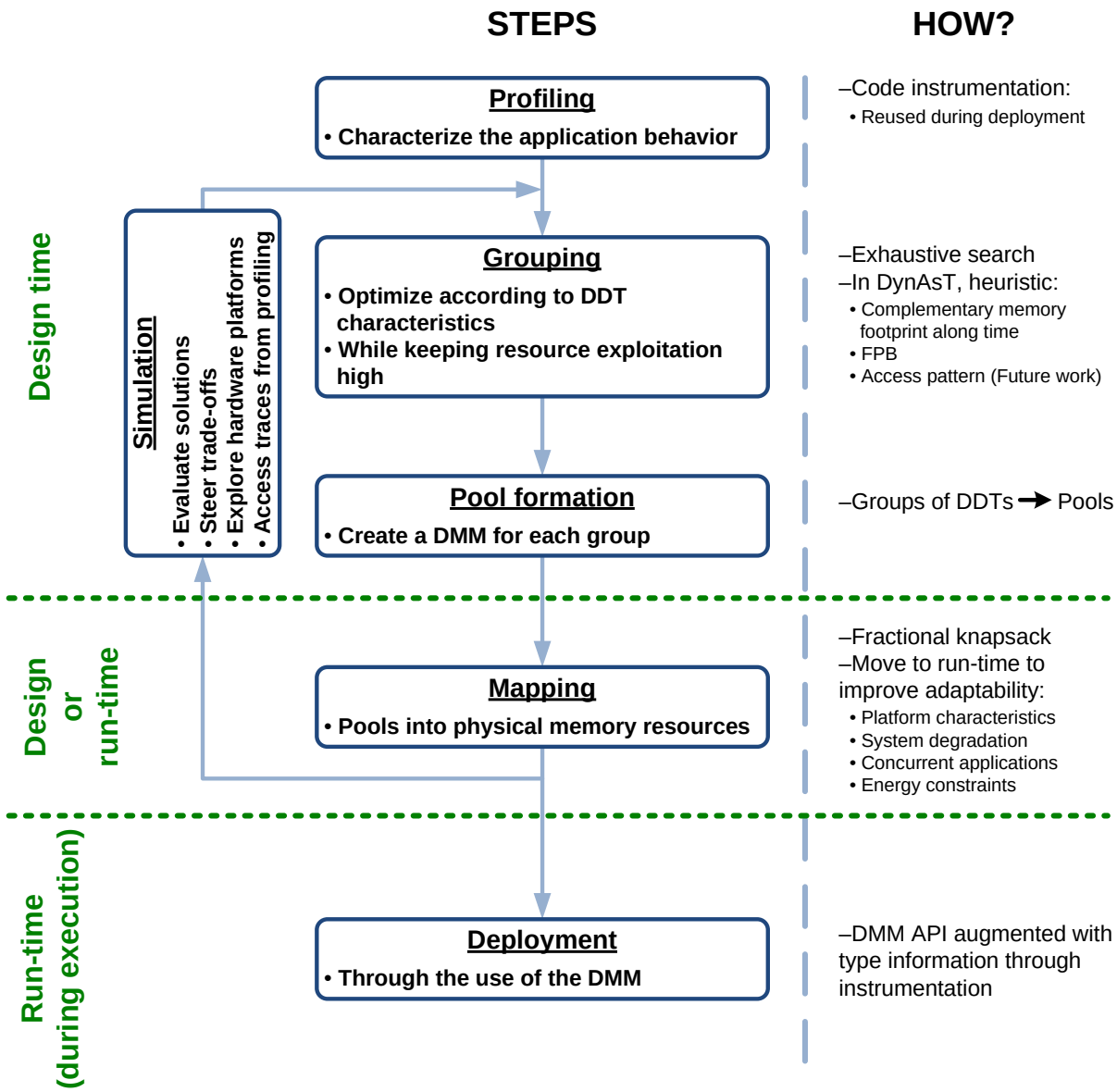


Figure 1.10.: For applications that use dynamic memory and present low access locality, I propose a methodology that places data objects according to their characteristics and that of the platform's memory resources, reducing data movements across the memory subsystem.

instrument the application's source code. This task is designed to reduce overall effort; hence, it serves both to conduct an extensive profiling of the application at design time and to supply the dynamic memory manager with the required additional information at run-time. Most of the burden of manual intervention on the source code involves subclassing the declarations of DDTs to extract during profiling the number of objects created and destroyed, their final addresses (to relate the corresponding memory accesses) and the type of each object created or destroyed – this is the part of the instrumentation that remains at run-time so that the DMM can implement the placement decisions.

The methodology divides the placement process in two steps: First, the DDTs are grouped according to their characteristics; then, the groups (not the individual DDTs) are placed into the memory resources. The heaviest parts are the grouping of DDTs and the construction of the DMMs themselves according to previous knowledge. These processes are performed at design time. The work of mapping every group on the platform's memory resources is a simpler process that can be executed at design time or delayed until run-time. The main benefit of delaying the mapping step would be the ability to produce a placement suited for the resources actually available in the system at run-time. For example, the application could be efficiently executed (without recompilation) on any instantiation of the platform along a product line, or it could adapt itself to yield a graceful degradation of system performance as the device ages and some resources start to fail – more critically, this could help to improve device reliability.

As explained in Chapter 2, placement is a complex problem and there is no guarantee that the approach adopted in this methodology is optimal. Therefore, the methodology foresees an additional step of simulation to evaluate the solutions generated and offer to the designer the possibility of steering trade-offs.

1.2.2. Methodology steps

The methodology is divided in seven steps:

1. **Instrumentation.** The source code of the application is instrumented to generate a log file that contains memory allocation and data access events.

2. **Profiling and analysis.** The application is profiled under typical input scenarios, producing the log file that will be one of the inputs for the tool that implements the methodology.

3. **Group creation.** The number of alive instances of a dynamic data type (DDT) at any given time, that is, the number of objects created by the application and not yet destroyed, varies along time in accordance to the structure and different phases of the application. Therefore, the total footprint of the DDTs will vary along time as well. As a consequence, if each DDT were allocated a memory area in exclusivity, and given that no data movements are executed at run-time, precious memory resources would remain underexploited during relevant fractions of the execution time.

   To tackle that problem, the grouping step clusters the DDTs according to their characteristics (memory footprint evolution along time, frequency of accesses per byte, etc.) as it explores the trade-off between creating a pool for each DDT and merging them all in a single pool. Two important concepts introduced in this step are liveness and exploitation ratio.

The process of grouping also helps to simplify the subsequent construction of efficient dynamic memory managers because each of them will have to support a smaller set of DDTs and, hence, requirements.

4. **Definition of pool algorithms.** Each pool has a list of DDTs, the required amount of memory for the combined – not added – footprint of the DDTs that it will contain, and the description of the algorithms and internal data structures that will be employed for its management. The appropriate algorithm and organization for each pool can be selected using existing techniques for the design of dynamic memory managers such as the ones presented by Atienza et al. [AMP+15]. The DMMs are designed as if each pool were the only one present in the application.

5. **Mapping into memory resources.** The description of the memory subsystem is used to map all the pools into the memory modules of the platform, assigning physical addresses to each pool. This step is an instance of the classic fractional knapsack problem; hence it can be solved with a greedy algorithm in polynomial time. The output of this step, which is the output of the whole methodology, is the list of memory resources where each of the pools is to be placed.

6. **Simulation and evaluation.** The methodology includes a simulation step based on the access traces obtained during profiling to evaluate the mapping solutions before deployment into the final platform and adjust the values of the parameters that steer the various trade-offs. Additionally, if the exploration is performed at an early design stage when the platform is still unfinished, the results of the simulation can be used to steer the design or selection of the platform in accordance to the needs of the applications.

7. **Deployment.** Finally, the description of the pools needed to perform the allocation and placement of the application DDTs is generated as metadata that will be distributed with the application. The size of the metadata should not constitute a considerable overhead on the size of the deployed application. In order to attain maximum flexibility, a factory of DM managers and the strategy design pattern [GHJV95] can be employed at run-time to construct the required memory managers according to their description.

1.2.3. Methodology implementation in the DynAsT tool

The methodology has been implemented as a functional tool, $\mathcal{DynAsT}$, which can be used to improve the placement of dynamic data structures on the memory subsystem of an existing platform, or to steer the design of a new platform according to the particular needs of the future applications. I use it through Chapters 2 and 4 to show the concrete algorithms that implement the methodology and the results obtained in several examples. In the context of the methodology, $\mathcal{DynAsT}$ performs the analysis of the information obtained during profiling, the grouping and mapping steps, and the optional simulation (Figure 1.11).

The first action of the tool is to analyze the traces obtained during profiling to infer the characteristics of each DDT. During the grouping step, which is implemented using several heuristics to limit its complexity, it analyzes the footprint of those DDTs that have a similar amount of accesses and tries to cluster them, matching the "valleys" in the memory footprint of some with the "peaks" of others. Pool formation is currently a stub to introduce any dynamic memory management techniques already available. Finally, $\mathcal{DynAsT}$ produces a mapping of every pool, and hence of the dynamic data objects that it will contain, over the memory modules of

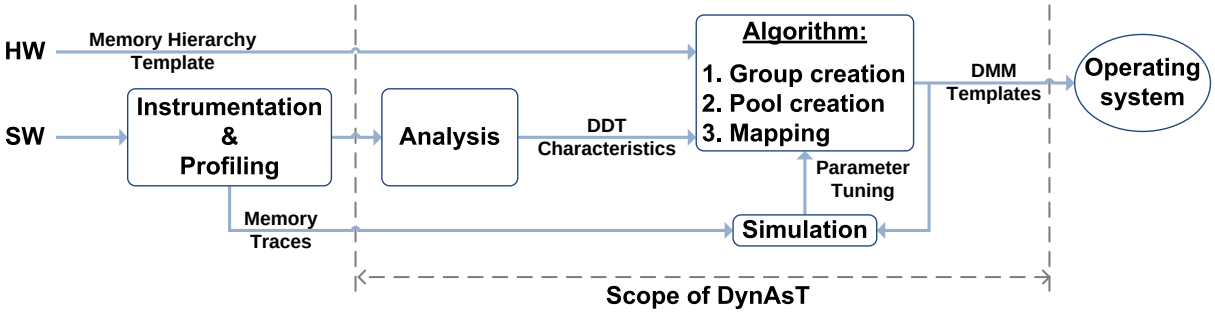


Figure 1.11.: Methodology implementation in $\mathcal{DynAsT}$.

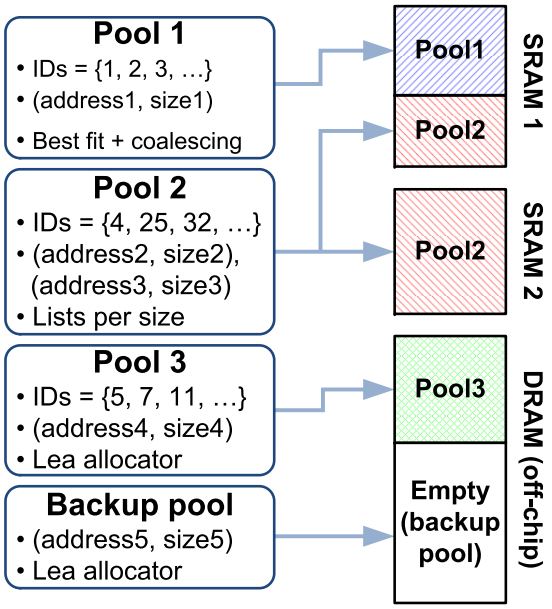


Figure 1.12.: The outcome of the methodology is the description of the pools, the list of DDTs that will be allocated in each one and their placement on memory resources.

the target platform. For this task, the tool considers the characteristics of each group/pool and the properties of each memory module in the platform. The mapping step is simpler because the pools can be split over several memory resources even if their address ranges are not consecutive. These steps interact with each other through data abstractions (e.g., DDT behavior, group, group behavior, pool); hence, they can be improved independently. For example, further research may devise new grouping algorithms or the mapping step could be integrated with the run-time loader of the operating system.

As an optional feature, $\mathcal{DynAsT}$ also includes the memory hierarchy simulator prescribed in the methodology to evaluate the properties of the generated placement solutions. The simulation results can be used to iterate over the grouping, pool formation and mapping steps to tune the placement (e.g., changing the values of the parameters in each algorithm).

The output of the tool is a description of the dynamic memory managers that can be used to implement the solution at run-time. This description includes their internal algorithms, the DDTs that will be allocated into each of them, their size and their position in the memory address space (Figure 1.12).

The data placement generated by $\mathcal{DynAsT}$ is enforced upon the application as follows. During instrumentation, each DDT is assigned a unique numerical ID. The grouping phase of the tool decides which DDTs share a pool: The correspondence between groups and pools is direct. During application execution, the DMM receives both the size of the request and the type of the object being created – that is, the ID assigned to its DDT. With that extended information, the DMM identifies the pool that corresponds to that ID and calls the (sub)DMM the manages that pool. If the local DMM does not have enough resources left in its heaps, the request is redirected to the backup pool. The backup pool resides in the last eligible memory resource – it is important that it remains seldom used as accesses to any objects allocated there will incur the highest cost – and receives allocations that exceed the maximum footprint values measured during profiling for each pool. The backup pool is not the same than pools that are explicitly mapped to a DRAM, something that happens because they are the less accessed or (in future work) because their access pattern makes them the least detrimental for the characteristics of a DRAM and so get less priority for the use of the available SRAMs.

In summary, $\mathcal{DynAsT}$ currently implements a set of simple algorithms and heuristics that can be independently improved in the future. Despite its simplicity, it shows the promising results that can be attained with a data placement that takes into account the characteristics of the application data types.

1.2.4. Novelty

The main contribution of this work is an efficient mapping of dynamic data objects on physical memory resources so that the most accessed ones are placed into the most efficient memories, improving performance and energy consumption in heterogeneous memory subsystems. The produced placement is exclusive (the system contains only one copy of each data object) and static (no data movements are performed across elements of the memory subsystem at run-time). To cope with the high complexity of the problem, I propose to address it at the abstraction level of groups of DDTs and divide it into two steps: Grouping and placement into resources.

This division allows improving resource exploitation without relying on data movements. Instead, the novel grouping step analyzes the evolution of the memory footprint of all the DDTs during the execution time and combines those that are complementary. The subsequent mapping step uses an efficient greedy algorithm to place the pools on memory resources, which clears the path for a future run-time implementation of this step.

My proposal is compatible with existing techniques for the placement of static data (stack and global objects), such as the ones presented by Kandemir et al. [KKC+04], Verma et al. [VWM04] or González-Alberquilla et al. [GCPT10], and the normal use of cache memories (through the definition of non-cacheable memory ranges) when those techniques are efficient. It is also adequate for lightweight embedded platforms that contain only small SRAMs instead of a traditional organization with one or several caches backed by a bigger main memory. The designer can use dynamic memory to allocate all the data objects and leave the management of resources and placement considerations to the tool.

As a final remark, I would like to clarify that the aim of this work is to show the importance of doing an explicit placement of dynamic data structures and the tantalizing opportunities offered by even simple algorithms such as the ones presented here, rather than producing the best possible placement of DDTs on the memory subsystem. For example, the profiling techniques used in this work might be substituted by more efficient mechanisms to characterize

the applications – this being particularly important to tackle data placement at the WSC level. Similarly, the grouping process described here is a convenient way to increase the exploitation of memory resources, but the concrete heuristic-based algorithm proposed is not necessarily optimal. Indeed, grouping itself is just one possibility in the trade-off between creating an independent memory heap for each dynamic data type and a single heap for all of them.

1.2.5. Additional contributions

Besides the main goal of this work, previous experience with several optimization techniques revealed the necessity to count with a solid characterization of application behavior. In that regard, I took part, in close collaboration with other people (see Bartzas et al. [BPP+10] for a list of contributors to this work), in a formalization of the concept of *software metadata* for the systematic characterization of the dynamic-data access behavior of software applications. The software metadata may serve as a central repository that consecutive optimization techniques use to operate on the application in a structured way, reducing the effort required for profiling and characterization.

The insight behind this work, presented in Chapter 5, is that although different optimization tools may need knowledge on a different set of behavior characteristics, a considerable amount of information requirements may be shared among those tools. If each of them has to provide its own characterization mechanism, many resources are wasted in profiling, analyzing and extracting useful information from the same application. Instead, we proposed the creation of a common knowledge repository with the behavior of the applications for each of the relevant use cases of the system. Constructing this knowledge base may still require a significant amount of resources, but then it will be shared among the different optimization tools. Therefore, the overall effort required to optimize the system with each tool is reduced. In addition to saving time-to-market, this may also clear the path for further optimization work because the entry-barrier for the implementation of new tools is lowered.

Finally, although my work on dynamic data placement is focused on embedded systems with limited power budget and no virtual memory mechanisms, in Chapter 7 I explain how these concepts could also make an impact on other environments such as big data centers and cloud computing. Future work might improve performance, but, most importantly, reduce energy consumption in warehouse-scale computers by carefully scheduling applications to match their demands to the available resources of NUMA and scale-out – as opposed to scale-up – systems.

1.3. Related work

In this section I compare the main contributions of this work with other interesting research and provide further references for the interested reader, organized by categories.

1.3.1. Code transformations to improve access locality

Memory hierarchies with cache memories are useful to narrow the gap between the speed of processors and that of memories. However, they are only useful if the algorithms generate enough data access locality. Thus, for many years researchers have used techniques that modify the layout of (static) data objects or the scheduling of instructions in code loops to increase access locality. Examples of those techniques are loop transformations such as tiling (blocking)

that enhance caching of arrays [CM95, AMP00, LLL01], or the Data Transfer and Storage Exploration [CWG+98] methodology, which is one of several proposals to tackle code scheduling at different levels of abstraction (i.e., from the instruction to the task levels), targeting either energy or performance improvements.

1.3.2. SW-controlled data layout and static data placement

The previous techniques aim to improve data access locality, particularly for systems that include cache memories. However, many works showed that scratchpad memories (small, on-chip SRAM memories directly addressable by the software) can be more energy-efficient than caches for static data if a careful analysis of the applications and their data access patterns is done [PDN00, KKC+04, VWM04, GBD+05]; among them, the work of Banakar et al. [BSL+02] is particularly descriptive. Therefore, two main groups of works were conducted to explore the use of scratchpad memories: Those ones that use data movements to implementing a kind of software caching and prefetching, and those others that produce a fixed data placement for statically allocated data such as global variables and the stack.

The fundamental idea of the works in the first group is to exploit design time knowledge about the application so that the software itself determines explicitly which data needs to be temporarily copied to the closer memories, producing a dynamic data layout that is usually implemented programming a Direct Memory Access (DMA) controller to copy blocks of data between main memory and the scratchpad while the processor works on a different data block [WDCM98, APM+04, GBD+05, DBD+06]. As with hardware caches, those methods are only useful if the algorithms present enough data access locality [WDCM98]. Otherwise, performance may still be improved via prefetching, but energy consumption increases.[5]

The works in the second group aimed to statically assign space in the scratchpad to the most accessed data (or code) objects in the application [KRI+01, SWLM02, VSM03, KKC+04, VWM04]. Panda et al. [PDN00] and Benini and de Micheli [BM00] presented good overviews of several techniques to map stack and global variables. Regarding specifically the stack, a hardware structure to transparently map it into a scratchpad memory is presented by González-Alberquilla et al. [GCPT10]. Soto et al. [SRS12] explore, both from the perspective of an exact solver and using heuristics, the problem of placing (static) data structures in a memory subsystem where several memories can be accessed in parallel. Their work can be partially seen as a generalization of the mapping step in the methodology presented through this work if each of the pools is considered as a big static data structure (an array) with a fixed size – however, their approach would prevent splitting a pool over several memory resources because a data structure is viewed as an atomic entity. Nevertheless, that work presents important notions to maximize the chances of parallel accesses when mapping static data structures into several memory modules.

As data objects cannot be usually split during mapping, greedy algorithms based on ordering by frequency of accesses per byte (FPB) are not optimal and most of the works oriented to produce a static data placement resort to more complex methods such as integer linear programming (ILP). This issue is particularly relevant because the amount of individual data

[5] Energy consumption may increase in those cases because the cost of writing and reading input data from the scratchpad, and then writing and reading results before posting them to the DRAM, is added to the cost of accessing the data straight from the DRAM. As a result, independently of whether they are performed by the processor or the DMA, a net overhead of two writes and two reads to the scratchpad is created without any reutilization payback.

objects can increase significantly with the use of dynamic memory (each object-creation location in the code can be executed many times); furthermore, their exact numbers and size is usually not known until run-time. Indeed, the concept of data objects created at run-time is itself not susceptible to such precomputed solutions because further placement choices need to be made when new objects are created, maybe rendering prior decisions undesirable. Either prior placement decisions must be lived with, or a data migration mechanism to undo them would be needed.

In contrast with those works, here I propose optimizations for heap data (allocated at run-time) that altogether avoid movements between memory elements that would be difficult to harness due to the lower locality of dynamic data structures. Nevertheless, those approaches are compatible with this work and may complement it for the overall optimization of access to dynamic and static data.

1.3.3. Dynamic memory management

Wide research effort has also been performed on the allocation techniques for dynamic data themselves to construct efficient dynamic memory managers. Several comprehensive surveys have been presented along the years, such as the ones from Margolin et al. [MPS71] and Bozman et al. [BBDT84] – which were rather an experimental analysis of many of the then-known memory management techniques with their proposals for improvement in the context of IBM servers – and Wilson et al. [WJNB95] – who made a deep analysis of the problem distinguishing between policies and the methods that implement them. Johnstone and Wilson went further in their analysis of fragmentation in general purpose DMMs [JW98]. They argued that its importance is smaller than previously believed and possibly motivated in big part by the use in early works of statistical distributions that do not reflect the hidden pattern behaviors of real-life applications; thus, they advocated for the use of real-program traces and for the focus of future research to be shifted towards performance. Indeed, the focus of most research has moved from fragmentation and performance to performance and energy consumption; and, since energy consumption depends roughly on the number of instructions and memory accesses executed by the processor, we can say that performance became soon the main goal of most research in the area.

Many works in the field have focused on the design of general-purpose DMMs that could be used under a wide variety of circumstances. In that line, Weinstock and Wulf [WW88] presented QuickFit, where they proposed to split the heap in two areas, one managed with a set of lists of free blocks and the other with a more general algorithm. Their insight was that most applications (according to their particular experience) allocate blocks from a small set of different sizes, that those sizes are usually small (i.e., they represent small data records) and that applications tend to allocate repeatedly blocks of the same size. Further experiments confirmed that many applications tend to allocate objects of a small number of different sizes, but that those sizes may change significantly between applications. The DMM design presented by Kingsley[6] generalizes that idea and proposes a fast allocation scheme, where a set of lists is created each corresponding to exponentially increasing block sizes. The design presented by Lea [Lea96] has been the reference design in many Linux distributions for many years as it presents good compromises for different types of applications and their dynamic memory requirements.

[6] See the survey of Wilson et al. [WJNB95] for a description of that dynamic memory manager, which was designed for the BSD 4.2 Unix version.

Other works have proposed the construction of DMMs specifically tailored to the characteristics of each application. Their most relevant feature is that, contrary to the habit of hand-optimizing the DMMs, they explored automated methods to generate the DMMs after a careful profiling and characterization of the applications. For example, Grunwald and Zorn [GZ93] proposed CustoMalloc, which automatically generates quick allocation lists for the most demanded block sizes in each application, in contrast with the predefined set of lists created by QuickFit. As in that work, a more general allocator takes care of the rest of (less common) allocation sizes. Interestingly, Grunwald and Zorn also reported in their work the presence of some sense of temporal locality for allocations: Applications tend to cluster allocations of the same size.

Vmalloc, presented by Vo [Vo96], proposed an extended interface that applications can use to tune the policies and methods employed by the DMM, providing also a mechanism to generate several regions, each with its own configuration. It also provided auxiliary interfaces for debugging and profiling of programs. Some of his proposals are still offered by systems such as the C run-time library of the Microsoft Visual Studio (debugging and profiling [Mic15]) and the Heap API of Microsoft Windows (separate, application-selectable regions with different method choices).

The set of works presented by Atienza et al. [AMC+04a, Ati05, AMM+06a] and Mamagkakis et al. [MAP+06] formalize in a set of decision trees the design space for the construction of DMMs and introduce optimizations specific for the realm of embedded systems that improve on the areas of energy consumption, memory footprint and performance (cost of the allocation operations). They also defined an ordering between the decision trees to prioritize each of those optimization goals. Their work is complementary to the work I present here as they deal with the design of efficient DMMs and their internal organization, not with the placement problem. Indeed, I rely on it for the step of pool formation in my methodology.

Some effort has also been devoted to partial or full hardware implementations of dynamic memory management. Interesting works in this area are the ones presented by Li et al. [LMK06] and Anagnostopoulos et al. [AXB+11].

A common characteristic of all those works is that they do not consider the actual mapping of the pools into physical memory resources. Despite the fact that DDTs become increasingly important as applications in embedded systems grow more complex and driven by external events [Man04], the heap is frequently left out of the placement optimizations and mapped into the main DRAM. That was the main motivation behind the work that I present here.

An interesting work that takes into consideration the characteristics of the memory module where the heap is placed was presented by McIlroy et al. [MDS08], who developed a specialized allocator for scratchpad memories that has its internal data structures highly optimized using bitmaps to reduce their overhead. Their work provided mechanisms to efficiently manage a heap known to reside in a scratchpad memory; it is hence complementary to the proposal of this thesis, which proposes an efficient placement of dynamic data into the system memories. In other words, my work studies the mapping of DDTs into heaps, and of heaps into physical memory resources, whereas their work deals with the internal management of those heaps that have been placed in a scratchpad memory.

Finally, in a different but not less interesting context, Berger et al. [BMBW00] introduced a memory allocator for multithreaded applications running on server-class multiprocessor systems. Their allocator makes a careful separation of memory blocks into per-processor heaps and one global heap. However, their goal is not to take advantage of the memory organiza-

tion, but to avoid the problems of false sharing (of cache lines) and incremented memory consumption under consumer-producer patterns in multiprocessor systems. Although not directly related to the main goal of this work, it is worth mentioning because most middle or high-end embedded devices feature nowadays multiple processors. Exploring the integration of their ideas with techniques for data placement in the context of multiprocessor embedded systems might an interesting research topic in the immediate future.

1.3.4. Dynamic data types optimization

Object-oriented languages offer built-in support for DDTs, typically through interfaces for vectors of variable number of elements, lists, queues, trees, maps (associative containers), etc. A software programmer can choose to use the DDT that has the most appropriate operations and data organization, but this is usually done without considering the underlying memory organization. However, changing how the different DDTs are used, considering the data access patterns of the applications and the characteristics of the cache memories, can produce considerable efficiency improvements. Therefore, many authors looked for ways to improve the exploitation of cache resources when using DDTs. One of the resulting works was presented by Chilimbi et al. [CDL99], who applied data optimization techniques such as structure splitting and field reordering to DDTs. A good overview of available transformations was introduced by Daylight et al. [DAV+04] and a multi-objective optimization method based on evolutionary computation to optimize complex DDT implementations by Baloukas et al. [BRA+09]. Although their authors take the perspective of a programmer implementing the DDTs, it should be fairly easy to apply those techniques to the standard DDTs provided by the language.

A very interesting proposal to improve cache performance was described by Lattner and Adve [LA05]: A compiler-based approach is taken to build the "points-to" graph of the application DDTs and segregate every single instance of each DDT into a separate pool. However, this approach produces a worst-case assignment of pools to DDTs as the free space in a pool cannot be used to create instances of DDTs from other pools. Most importantly, this work was developed to improve the hit ratio of cache memories (e.g., it enables such clever optimizations as compressing 64-bit pointers into 32-bit integer indexes from the pool base address), but it is not specifically suited for embedded systems with heterogeneous memory organizations. Nevertheless, the possibility of combining their analysis techniques with our placement principles in future works is exciting.

1.3.5. Dynamic data placement

An early approximation to the problem of placement for dynamically allocated data was presented by Avissar et al. [ABS01], but they resorted to a worst-case solution considering each allocation place in the source code as the declaration of a static variable and assigning an upper bound on the amount of memory that can be used by each of them. Moreover, they considered each of these pseudo-static variables as independent entities, not taking into consideration the possibility of managing them in a single pool, and adding considerable complexity to their integer linear optimizer; in that sense, their work lacked a full approach to the concept of DM management. Nonetheless, that work constitutes one of the first approaches to the placement challenge.

Further hints on how to map DDTs into a scratchpad memory were offered by Poletti et al. [PMA+04]. In that work, the authors arrived to a satisfactory placement solution that re-

duces energy consumption and improves performance (albeit the latter only in multiprocessor systems) for a simple case; however, significant manual effort was still required from the designer. The methodology presented in this thesis can produce solutions of similar quality, but in an automated way and with a more global approach that considers all the elements in the memory subsystem (versus a single scratchpad memory).

The work presented by Mamagkakis et al. [MAP+06] lays in the boundary between pure DM management, which deals with the problem of efficiently finding free blocks of memory for new objects, and the problem of placing dynamic objects into the right memory modules to reduce the cost of accessing them. There, the authors proposed a method to build DMMs that can be configured to use a specific address range, but leave open the way in which such address range is determined. However, for demonstration purposes, they manually found an object size that received a big fraction of the application data accesses, created a pool specifically for it and then mapped that pool into a separate scratchpad memory, obtaining important energy and time savings. The work I present in this thesis is complementary because it tackles with the problem of finding an appropriate placement of data into memory resources, while leaving open the actual mechanism used to allocate blocks inside the pools. Additionally, it classifies the allocations according not only to the size of the memory blocks, but also to the high-level data type of the objects. This enables a complete analysis of the DDTs that is impossible – or, at least, very difficult – if different DDTs with the same size are not differentiated.

More recently, efforts to create an algorithm to map dynamic, linked, data structures to a scratchpad memory have been presented by Domínguez et al. [DUB05] and Udayakumaran et al. [UDB06], who propose to place in the scratchpad some portions of the heap, called "bins." The bins are moved from the main memory to the scratchpad when the data they hold are known to be accessed at the next execution point of the application. For each DDT, a bin is created and only its first instances will reside in it (and so in the scratchpad), whereas the rest will be kept in a different pool permanently mapped into the main memory. That method requires a careful analysis of the application and instrumentation of the final code to execute the data movements properly. One drawback of this approach is that in order to offset the cost of the data movements, the application must spend enough time in the region of code that benefits from the new data distribution, reusing the data in the bins. Compared to this approach, the method presented here avoids data migration between elements of the memory subsystem and considers the footprint of all the instances of the DDTs, not only the first ones.

The work that I describe here has three important differences in relation with previous techniques. First, it tries to solve the problem of mapping all the DDTs, not only into one scratchpad memory, but into all the different elements of a heterogeneous memory subsystem. Second, it employs an exclusive memory organization model [JW94], which has some advantages under certain circumstances [ZDJ04, Sub09]. As applied here, this model avoids duplication of data across different levels of the memory subsystem: Each level holds distinct data and no migrations are performed. Avoiding data movements reduces the energy and cycles overhead at the possible cost of using less efficiently the memory resources during specific phases of the application execution. However, due to the additional grouping step, which is based on the analysis of the memory footprint evolution and access characteristics of each DDT, and the fact that no resources are wasted in duplicated data – effectively increasing the usable size of the platform memories – I argue that this methodology can overcome the possible inefficiencies in many situations. As a final consideration, this method is also applicable

to multithreaded applications where DDTs are accessed from different threads.

In a different order of things, several algorithms to perform an efficient mapping of dynamic applications (i.e., triggered by unpredictable external events) into DRAM modules were presented by Marchal et al. [MCB+04]. However, they dealt with other aspects of the applications' unpredictability, not with the DDTs created in the heap. Interestingly, they introduced the concept of selfishness to reduce the number of row misses in the banks of each DRAM module. That idea could be easily integrated into this work to control the mapping of DDTs on the different banks of DRAM modules.

1.3.6. Metadata

As an additional contribution, I present an approach to characterize the dynamic-memory access behavior of software applications. In this regard, a significant amount of research has been performed on memory analysis and optimization techniques to reduce energy consumption and increase performance [BMP00, PCD+01] in embedded systems. Traditional optimizations tended toward using only compile-time information. With such techniques, the source code is completely transformed to a specific standardized form such that the analysis can easily happen [CDK+02]. However, for dynamic applications this is not efficient because the variations in behavior due to the changing environment conditions and inputs cannot be captured by source code analysis alone. Therefore, profiling is a crucial resource to focus the optimization process towards the most common use cases.

With respect to profiling, many tools work directly on the binary application without requiring source code instrumentation. For example, Gprof [GKM82] uses debugging information generated during compilation to find out the number of function calls and the time spent in each of them. However, it is not designed to provide insights for optimizations according to memory access patterns. More recent tools such as Valgrind [Net04] are able to look at the memory accesses and use this information to provide consistency checks for the executed programs. Valgrind allows identifying the lines in the source code that are responsible for an access, but cannot give a semantic analysis of the variable that was actually accessed. A framework that is able to perform link-time program transformations and instrumentation to achieve code-size reduction is presented by Van Put et al. [PCB+05]. Traditionally, high-level development environments offer some means to analyze the application during one execution and show the programmer the amount of time spent on every function or block of code. However, the process is usually manual in the sense that the programmer decides the portions of code that need optimization. Finally, modern development environments have the ability to use profiling during the compilation process: The developer executes the application using a set of representative inputs and the instrumentation, usually transparent, records execution data that is later used to apply aggressive optimizations or to solve trade-offs.

In Chapter 5 I show a method to annotate the application DDTs with templates so that accesses to their attributes are logged. After the instrumentation, the application can be executed with different inputs. The main advantage of this annotation method is that the compiler propagates the logging instrumentation automatically, guaranteeing that all the accesses are logged. This method supports multithreading in the sense that it can identify the thread that performs an access to a dynamic object. However, the profiling mechanism may alter the execution trace of the application (e.g., due to "Heisenbugs"), and that may mask race conditions or other types of bugs. This should not be a problem for well-behaved software that imple-

ments proper locking and thread-safety; in any case, other existing mechanisms can be used to identify these conditions.

Eeckhout et al. [EVB03] explain how to choose input data sets to conduct the profiling phase during microprocessor development in an efficient manner. The idea is to reduce the number of input cases that need to be simulated in order to obtain a representative picture of the microprocessor performance. Their work focuses in profiling performance metrics that affect mainly to the microprocessor, such as branch prediction accuracy, cache miss rates, sequential flow breaks, instruction mix, instruction-level parallelism or energy consumption in the microprocessor. In contrast, the work presented here deals with the analysis of the whole system, not only the microprocessor: It takes into consideration factors as diverse as total energy consumption, memory footprint of data structures and the interaction between the processor and other elements that access the memories of the platform. However, that work was quite interesting because it could lead to an efficient method to drive the profiling process.

In a different field, several groups presented results on workload characterization [HKA04] and scenario exploitation [GBC05] that are also relevant in our context. They also extract software metadata information to enable memory subsystem optimizations. These efforts focus on defining the characteristics of the run-time situations that trigger specific application behaviors with significant impact on resource usage and data access behavior. This work presents concepts related to theirs, but instantiated and extended to profile and analyze input-dependent data access behavior and subsequently build its metadata representation.

The use of metadata to describe systems is pervasive in the process of discovery, categorization and later retrieval of any type of information. In the realm of Software Engineering, metadata has been applied to describe several aspects of software applications. For example, the information included in the executable files of many operating systems is considered metadata because it allows them to know how to manipulate the files (e.g., distinguish code and data segments, determine required stack and heap sizes, etc.). A similar case is the information included in the Java class file format which enables dynamic linking and reflection. A common use of metadata in the Software Engineering field is to enable software mining: Discovery of knowledge from software artifacts that allows for their classification and manipulation. This is the case of the Knowledge Discovery Model (KDM) [Objb] from the Object Management Group (OMG/ADM) [Obja], whose purpose is to represent entire existing applications with the goal of modernizing and revitalizing their architecture, disregarding the programming language that was used to create them:

> "KDM is a common intermediate representation for existing software systems and their operating environments, that defines common metadata required for deep semantic integration of Application Lifecycle Management tools. [...] It defines a common vocabulary of knowledge related to software engineering artifacts, regardless of the implementation programming language and runtime platform – a checklist of items that a software mining tool should discover and a software analysis tool can use. KDM is designed to enable knowledge-based integration between tools." [Objb]

One very interesting aspect of KDM is the Micro-KDM concept to represent the (static) behavior of software. It provides a high-level intermediate representation of existing applications, similar to the internal register transfer-level representation used by some compilers, but at a higher abstraction level. The scope of KDM is broad: It aims to represent the whole ecosystem around the applications, not just the properties of their source code. Also, KDM was mainly

designed for use with enterprise software, which has different characteristics than the applications that run on embedded systems. The software metadata presented here is more specific for the analysis and optimization of embedded applications that use dynamic memory. Therefore, I believe both approaches are complementary. It would be interesting future work to analyze the integration of the software metadata structures and tools presented here in the more general framework provided by the Object Management Group.

At the time when our work on characterization techniques was developed, the concept and exploitation of software metadata had started to be explored by several research projects [STR07, Gen07, ISTI07] with the goal of producing a standard description of the characteristics of applications running on embedded platforms that could be interchanged between tools of different vendors and academia. Another interesting use of metadata is the characterization of *commercial off-the-shelf* (COTS) software components [Sta94], which aims to provide a catalogue of software components that can be plugged in like hardware ones. However, that aspect of metadata focuses mainly on a structural description of the software components. In a similar way, the concept of metadata is also applied in the domain of embedded hardware. A relevant example is the use of the IP-XACT [Acc10] standard for the definition of the metadata format that characterizes each hardware component to easily design, test and verify embedded hardware platforms.

The main differentiator of the work presented here is that its approach to metadata does not aim to define the engineering of software applications, nor to analyze or characterize their structure, but to represent the characteristics of their behavior when they are subject to specific inputs in a reusable way. Moreover, this work is specifically limited to the scope of applications dominated by dynamically allocated data types running on embedded systems. The produced software metadata can be used with the platform description to customize resource management and apply different optimization techniques.

1.3.7. Computational complexity

The field of computational complexity is broad, but little is required to understand the work presented here. A good introduction can be obtained from the classic textbook of Cormen et al. [CLRS01, Chap. 34]. My work is closely related to the family of problems that contains the knapsack and general assignment problems. A very good introduction to those problems was presented by Pisinger in his PhD work [Pis95] and several related works where he studied specific instances such as the multiple knapsack problem [Pis99].

Chekuri and Khanna [CK00] present a polynomial-time approximation scheme (PTAS) for the multiple knapsack problem. They also propose that this is the most complex special case of GAP that is not APX-hard – i.e., that is not "hard" even to approximate. In the context of my own work, this means that for placement, which is more complex, no PTAS is likely to be found. Figure 1 in their work shows a brief schematic of the approximability properties of different variations of the knapsack/GAP problems.

Shmoys and Tardos [ST93] present the minimization version of GAP (Min GAP) and propose several approximation algorithms for different variations of that problem, which is relevant to my work because placement represents an effort of minimization and the maximization and minimization versions of the problem are not exactly equivalent – at least as far as I know. Many works deal with the fact that the general versions of GAP are even hard to approximate. In that direction, Cohen et al. [CKR06] explore approximation algorithms for GAP based on approximation algorithms for knapsack problems, while Fleischer et al. [FGMS06] present an

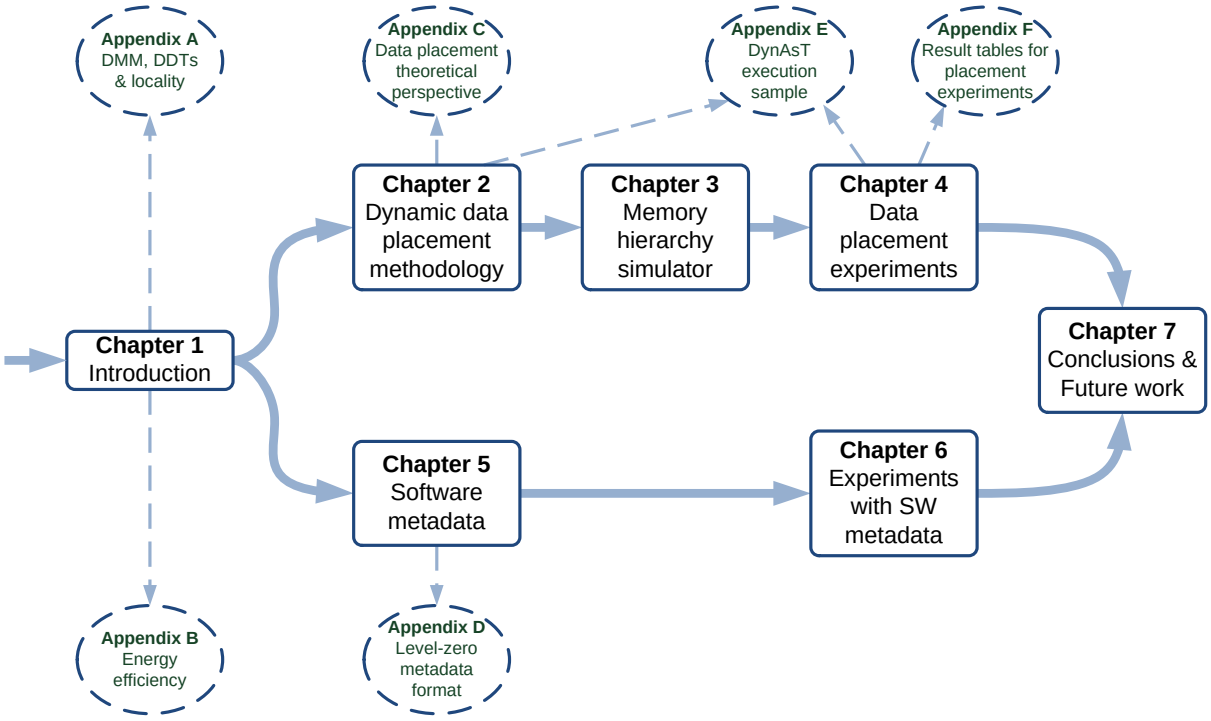


Figure 1.13.: Organization of this text. After the introduction of Chapter 1, Chapters 2, 3 and 4 present the work on dynamic data placement. Chapters 5 and 6 explain the additional contributions on application characterization using software metadata. Finally, Chapter 7 draws conclusions and outlines promising directions for future work. A set of appendixes complement several of the chapters with additional information and interesting insights.

improvement on previous approximation boundaries for GAP.

Finally, virtual machine colocation, which has strong resemblances with the data placement problem, was recently studied by Sindelar et al. [SSS11]. The particularity of this problem is that different virtual machines that run similar operating systems and applications usually have many pages of memory that are exactly identical (e.g., for code sections of the operating system). Placing them in the same physical server allows the hypervisor to reduce their total combined footprint.

1.4. Text organization

The rest of this text is organized as follows (Figure 1.13). In Chapter 2, I describe the methodology for dynamic data placement that forms the core of my PhD thesis. First, I analyze briefly the design space for the construction of dynamic memory managers capable of implementing data placement. Due to the complexity of the problem, I present the simpler approach, consisting of a grouping and a mapping step, that I use through this work to tackle it. The rest of the chapter explains thoroughly each of the steps in the methodology, presenting the corresponding algorithms and the parameters available for the designer to steer the work of the tool.

Chapter 3 describes in deep detail the implementation of the simulator included in $DynAsT$ and that can be used to evaluate the solutions produced (and maybe modify some algorithm parameters) or explore different platform design options.

In Chapter 4, I present the promising results obtained after using $\mathcal{DynAsT}$ to apply the methodology in a set of case studies. Each of these easily understandable examples is explored under many different combinations of platform resources. With them, I try not only to show the improvements that can be attained, but also to explain the reasons behind those improvements. I also include in this chapter an extensive discussion on the properties, applicability and drawbacks of the methodology; the conditions under which my experiments where performed and some directions for further improvement.

Chapter 5 presents the additional work on characterization of the dynamic-object access behavior of applications via a repository of software metadata. This work is supported in Chapter 6 with a case study where several optimization techniques are applied on the same application.

In Chapter 7, I draw my conclusions and present directions for future work on data placement. I also try to motivate the importance of data placement in environments other than embedded systems, specifically in big data centers, under the light of recent technological developments.

Finally, a set of appendixes is included at the end of this work. First, Appendix A introduces the main concepts of dynamic memory and how it is usually employed to build applications that can react to changing input conditions. More importantly, I use several examples to explain why the use of dynamic memory can have a significant negative effect on the performance of cache memories. Appendix B briefs on the improvements on energy efficiency consistently obtained along many decades thanks to the effect known as Dennard scaling. In Appendix C, I give a brief theoretical perspective on the relation between data placement and the knapsack and general-assignment family of problems. Appendix D presents the format of the log files used during profiling both for $\mathcal{DynAsT}$ and for the work on metadata characterization. To conclude, Appendix E contains several tables with the absolute values obtained during the experiments conducted in Chapter 4, and Appendix F presents an execution example of $\mathcal{DynAsT}$ on a simple application.

Chapter 2

Methodology for the placement of dynamic data objects

To palliate the consequences of the speed disparity between memories and processors, computer architects introduced the idea of combining small and fast memories with bigger – albeit slower – ones, effectively creating a memory hierarchy. The subsequent problem of data placement – which data objects should reside in each memory – was solved in an almost transparent way with the introduction of the cache memory. However, the widespread use of dynamic memory can hinder the main property underlaying the good performance of caches and similar techniques: Data access locality. Even though a good use of prefetching can reduce the impact on performance, the increase on energy consumption due to futile data movements is more difficult to conceal. Figure 2.1 summarizes this situation and the solutions proposed in this work.

During one of my stays at IMEC as a Marie Curie scholar, a research engineer approached me to suggest that the placement of dynamic objects could be trivially reduced to the placement of static objects – on which they had been working for a long time: "Simply give us the pool that holds all your dynamic objects as if it were an array with given access characteristics and our tools for the placement of static objects will place them as an additional static array." That proposal encloses a significant simplification: All the dynamic objects are considered as a whole, without distinguishing those heavily accessed from those seldom used. The result is likely a poor placement of dynamic data objects and improvable performance, justifying the development of specific techniques.

The key to the problem is discriminating between dynamic objects with different characteristics instead of mixing them in pools and treating them all as a single big array. In other words, the problem lies in differentiating between DDTs before putting them into pools, rather than in the placement of the resulting pools themselves, which is a more studied problem.

My proposal is based on avoiding data movements between elements in the memory subsystem using the dynamic memory manager to produce a careful placement of dynamic data objects on memory resources. But, how is it possible to achieve such a good placement? What are the new responsibilities of the dynamic memory manager? The rest of this chapter explains the available options to build a dynamic memory manager with support for data placement and proposes simple solutions to the main problems. The outcome is a methodology encom-

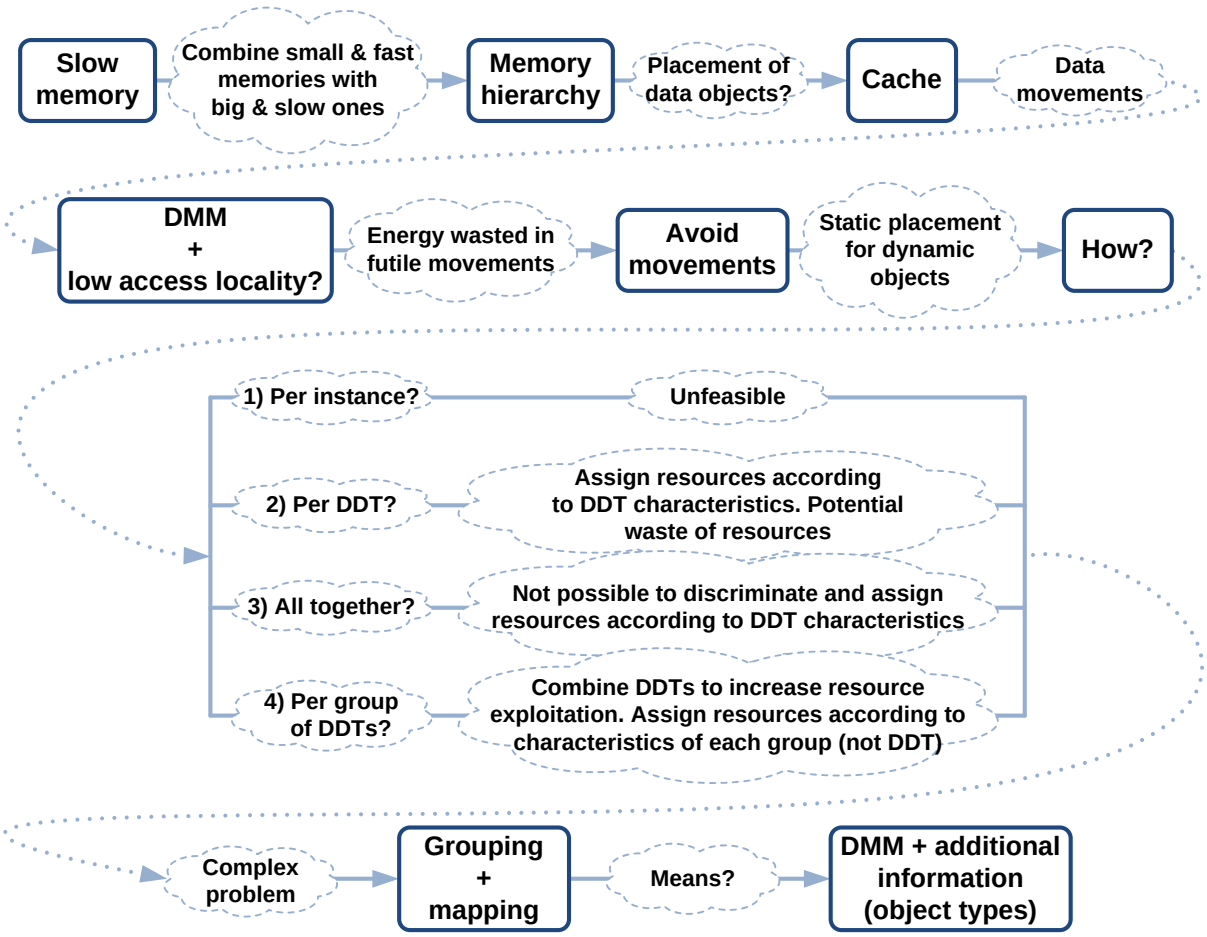


Figure 2.1.: A careful placement of dynamic data objects can improve the performance of the memory subsystem, avoiding the impact of low access locality on cache memories. To cope with the problem's complexity and improve resource exploitation, I propose a two-step process that groups DDTs and uses the DMM to implement a static placement of dynamic data objects.

passing the whole design process – from application characterization to deployment.

Definitions. Through the rest of this chapter, I shall use the term "heap" to denote the address range, "dynamic memory manager" (DMM) for the algorithms, and "pool" for the algorithms, their internal control data structures and the address range as a whole.

2.1. Choices in data placement

This section discusses briefly the different types of data placement and the granularity levels at which it can be implemented. It presents also the main requisites to achieve an efficient data placement.

2.1.1. The musts of data placement

Four are the fundamental requisites for a data placement technique to achieve an efficient use of resources:

Table 2.1.: Data placement according to implementation time and object nature. Static placement options choose a fixed position in the memory subsystem during object creation. Dynamic placement options may move data at run-time to bring closer those that are likely to be more accessed over the next period. Data movements may happen at the object or byte levels.

	STATIC DATA	**DYNAMIC DATA**
STATIC PLACEMENT	Direct placement per object. Problem: Low exploitation.	Direct placement at allocation time. Problem: Resource underuse.
DYNAMIC PLACEMENT	Cache. SW-based movements.	Cache. SW-based movements. Problem: Low access locality.

1. The most accessed data must be in the most efficient resources – or most of the accesses must be to the most efficient memory resources. Caching techniques rely on the property of access locality. Instead, my methodology uses information extracted through profiling to identify the most accessed data objects.

2. Recycling of resources. Caching techniques achieve it through data movements. My methodology uses the grouping step to identify and place together objects that alternate in the use of resources.

3. Avoiding unnecessary overheads. For example, moving data that will not be used (such as prefetching of long or additional cache lines) or with low reutilization. Caching techniques rely on access locality; software-based ones can use knowledge on the application to avoid some useless movements. My methodology produces a static placement of pools with no overheads in terms of data movements.

4. Avoid displacing very accessed data so that they have to be brought back immediately. For example, the victim cache was introduced to palliate this problem with direct-mapped caches [Jou90]. My methodology places data objects according to their characteristics, ensuring that seldom accessed objects are not migrated to closer resources; thus, no other objects can be displaced.

2.1.2. Types of data placement

Data placement in general can be approached in several ways depending on the nature of the data objects and the properties of the desired solutions. Table 2.1 shows the main options.

Placement policy. A static placement policy decides explicitly the memory position that every data object will have during the whole execution. The most used data objects can be placed over the most efficient resources, but accesses to the remaining ones bear higher costs. This happens even when the objects placed in the efficient memories are not being used – even if they hold no valid data at that time – because they have been granted exclusive ownership of their resources.

On the contrary, a dynamic placement policy can move data objects (or portions thereof) to keep track of changes in the application access patterns and obtain a better exploitation of resources. This is especially effective when the moved data are reused, so that the cost of the movement is shared among several subsequent accesses. Techniques such as cache memories

or software prefetching belong to that category. Whereas static placement usually requires less chip area and energy [BSL+02], dynamic placement offers better adaptability for applications that alternate between phases with different behavior or changing working conditions.

Data nature. Static data objects cannot be broken over non-contiguous memory resources because compilers commonly assume that all the fields in a structure are consecutive and generate code to access them as an offset from a base address. Therefore, static placement techniques [KRI+01, SWLM02, VSM03, KKC+04, VWM04] cannot use a greedy algorithm to calculate (optimally) the placement; hence, they resort to dynamic programming or integer linear programming (ILP) to produce optimal results. Although those static placement techniques can still be employed to guarantee that critical data objects are always accessed with a minimum delay, they easily become impractical for large numbers of objects, which is precisely the situation with dynamic data. Anyways, the placement of dynamic data objects cannot be fully computed at design time because their numbers and sizes are unknown until run-time. Consider, for example, that every location at the source code where a dynamic object is created can be executed multiple times and that the life span of the objects can extend further than the scope at which they were created.[1]

Dynamic placement techniques based on data movements (for caching and prefetching) may also be inadequate because the very nature of dynamic data objects reduces access locality and thus their efficiency. They would still cope efficiently for instance with algorithms that operate on dynamically-allocated vectors (e.g., a dynamically allocated buffer to hold a video frame of variable size), but not so much with traversals of linked data structures.

2.1.3. Granularity of dynamic data placement

Whereas the previous paragraphs looked into data placement in general, here I analyze specifically the dynamic data placement problem (placing all the instances of all the DDTs of the application into the memory elements) and the different granularity levels at which it can be tackled. For instance, it can be solved at the level of each individual instance, albeit at a high expense because their numbers are unknown and they are created and destroyed along time, which may render some early decisions suboptimal. The opposite direction would be to group all the DDTs into a single pool, but that would be quite close to the standard behavior of general-purpose DMMs without placement constraints.

The next option that appears intuitively is to characterize the mean behavior of the instances of each DDT and assign memory resources to the DDT itself.[2] In that way placement decisions can be taken for each DDT independently, while keeping the complexity bounded: Space is assigned to the pool corresponding to each DDT once at the beginning of the execution, and classic dynamic memory management techniques are used to administer the space inside the pool. However, avoiding data movements across elements in the memory hierarchy presents the risk of high resource underexploitation when there are not enough alive instances of the DDT assigned to a resource. In this work I propose a mechanism that tries to compensate for it by grouping together DDTs that have complementary footprint demands along time, so that "valleys" from ones compensate for "peaks" of the others and the overall exploitation

[1] A technique to verify whether this is the case is escape analysis. However, a simple example is when dynamic objects are added to a container created out of the current scope.

[2] It may happen that the instances of a DDT have different access characteristics. A mechanism to differentiate between them would be an interesting basis for future work.

ratio of the memory resources is kept as high as possible during the whole execution time. The difficulty of grouping lies then on identifying DDTs whose instances have similar access frequencies and patterns, so that any of these instances equally benefit from the assigned resources.

The following paragraphs describe each of the options mentioned before in more depth:

All DDTs together. The DDTs are put in a single pool, which uses all the available memory resources. Heap space can be reused for instances of any DDT indistinctly through splitting and coalescing. This is the placement option that has the smallest footprint, assuming that the selected DMM is able to manage all the different sizes and allocation patterns without incurring a high fragmentation. However, as the DMM does not differentiate between the instances of each DDT, new instances of very accessed DDTs would have to be placed in less efficient resources when the better ones are already occupied – perhaps by much less accessed instances. The DMM can try to assign resources according to the size of the objects, but unfortunately different DDTs with instances of the same size will often have a quite different number of accesses, limiting the effectiveness of placement: Very accessed and seldom accessed objects of the same size would all use the same memory resources.

This placement is what a classic all-purpose DMM would implement, which is precisely the situation that we want to avoid: The DMM is concerned only with reducing overhead (internal data structures and fragmentation) and finding a suitable block efficiently. An additional drawback of this option for the placement of linked data structures is that the DMM can choose any block from any position in the heap without constraints regarding the memory resource or position where it comes from. Thus, depending on the DMM's policy for reusing blocks (e.g., LIFO), nodes in a linked structure can end up in very different places of the memory map, reducing access locality.

One pool per DDT. This option is the opposite to the previous one: Each DDT gets its own pool that can be tailored for its needs. Placement can select the most appropriate memory resource for the instances of that DDT, without interferences from other ones. On the downside, memory wastage can increase significantly: Even when there are few instances – or none at all – of the DDT that is assigned to an efficient memory resource, that space cannot be used for instances of other DDTs. This is an important issue because movement-based caching techniques can exploit those resources, especially with higher degrees of associativity, by temporarily moving the data that are being used at each moment into the available resources – their problem in this respect is not wasting space, but seldom accessed objects evicting frequently accessed ones.

A possibility to palliate this problem could be temporarily overriding placement decisions to create some instances in the unused space. However, that would simply delay the problem until the moment when the space is needed again for instances of the assigned DDT. What should be done then? It is easy to end up designing solutions equivalent to the very same caching schemes that we were trying to avoid, but without their transparency.

Just for the sake of this discussion, let's examine briefly some of the options that could be considered to face this issue:

- Data migration at page granularity. It can be implemented in systems that support virtual memory for pools bigger than the size of a hardware page (e.g., 4 KB). The pools are divided into page-sized blocks. As the DMM starts assigning space in the pool, it tries to

use space from the minimum number of pages, in a similar way to the old "wilderness preservation heuristic" [WJNB95, pp. 33–34]. If other pool has (whole) unused pages in a more efficient resource, the system maps the pool's new pages in that resource. When the pool that owns the efficient memory resource needs to use that page for its own objects, the whole page is migrated to the correct memory resource.

In principle, this idea seems to allow some pools to exploit better available resources while they are underused by the DDTs that own them. However, as objects become scattered and free space fragmented (assuming that it cannot always be coalesced) the possibility of exploiting whole pages becomes more limited, perhaps making this mechanism suitable only for the first stages of the application execution. Future research can delve deeper into this issue.

- Object migration. Techniques to patch references (pointers) such as the ones used by garbage collectors might be used to migrate individual objects once the borrowed space is claimed by the owner DDT. Future research may evaluate the trade-off between the advantages obtained by exploiting the unused space for short-lived objects that are never actually migrated, and the cost of migrating and fixing references to long-lived objects.[3]
- Double indirection. With support from the compiler, this mechanism could help to migrate some objects without more complex techniques. Considering that it would affect only objects with lower amounts of accesses, the impact on performance might be limited under favorable access patterns. The trade-off between reloading the reference before every access or locking the objects in multithreaded environments would also need to be explored.

All of these options involve some form of data migration. A final consideration is that if objects are placed in a different pool (except for the case of migration at the page level), that pool needs to be general enough as to accommodate objects of other sizes, which partially defeats the whole purpose of DMM specialization. As none of the previous options are entirely satisfying, it seems clear that a completely different approach is needed to make static placement a viable option for dynamic data objects.

Per individual instance. This granularity level represents the ultimate placement option: Choosing the resource where to place each individual data object as it is created. However, contrary to the situation with static data, the number and characteristics of individual dynamic data instances cannot usually be foreseen: The number of instances actually created at a given code location may be unknown (for static data, that number is one). In order to correctly place each object, the DMM would need to know the access characteristics of that specific instance, but I cannot currently devise any mechanism to obtain that information, except perhaps for very domain-specific cases. Therefore, I deem this option as unfeasible for practical purposes, at least at the scope of the techniques presented here.

An interesting consideration is that, as the exact number of instances cannot be calculated until each one is effectively created, the system has to take each placement decision without knowledge about future events. Thus, when a new instance is created, the situation may not be optimal. Had the system known that the new object was indeed going to be created, it could

[3] I believe that the grouping mechanism that I propose in this work identifies such short-lived DDTs and automatically exploits any available space from other pools; hence, it voids the need for migrating this type of objects. The only advantage would be improving the performance of the first accesses to long-lived objects, before they are relocated.

have decided to save a resource instead of assigning it to a previously created one. In the absence of a mechanism to "undo" previous decisions, placement can be suboptimal. Maybe future systems will allow identifying the number of accesses to the objects and migrate them accordingly to adjust their placement a posteriori.

By groups of DDTs. Putting together "compatible" DDTs in the same pool. I propose this option as a means to assign dedicated space to some DDTs (static placement) while maintaining a high resource exploitation.

The last option is intriguing: How are the DDTs in each group selected? And, why group them at all? Although the number of dynamic data instances created by an application depends on conditions at run-time, oftentimes the designer can use profiling or other techniques to identify patterns of behavior. Particularly, it may be possible to obtain typical footprint and access profiles for the DDTs and group those ones that present similar characteristics or whose instances are mostly alive at complementary times – so that either they can share the same space at different times or it does not really matter whose instances use the assigned resources. The advantage of this option is that dissimilar DDTs can be treated separately while the total memory requirements are kept under control. In other words, grouping of DDTs for placement helps in provisioning dedicated space for the instances of the most accessed ones while attaining a high resource utilization.

2.2. A practical proposal: Grouping and mapping

The placement of dynamic data objects is a quite complex problem. Although I myself am not an expert on computational complexity, I believe that it belongs to the family of the (minimization) general assignment problem (GAP). In brief, I assume that tackling the problem at the level of individual dynamic data objects is unfeasible and therefore I propose to approach it as the placement of DDTs. However, this variant of the problem is still complex with respect to other problems from the same family because the number and size of the containers are not fixed. Thus, it is possible to put one object (DDT) in each container, but also to combine several DDTs in the same one and then both the size (memory footprint) of the DDTs (because of space reutilization similar to the case of VM-colocation [SSS11]) and the size of the container itself (which adjusts to the combined, not added, size of the DDTs) vary. Moreover, the cost of the accesses to a DDT (i.e., the cost of the object) depends not only on the container (memory resource) where it is placed, but the presence of other DDTs in the same container can modify it, as is the case of DDTs placed in the same bank of a DRAM. I offer the interested reader a deeper insight into this topic in Appendix C.

My proposal addresses the complexity of the problem in two ways. First, it raises the level of abstraction: Instead of working with individual data objects or individual DDTs, it identifies groups of DDTs with similar characteristics and includes them in the same pool of dynamic memory. This step is the key to improve resource exploitation. Whereas pools are the entities considered during placement, individual objects are allocated in the pools at run-time by a dynamic memory manager. In this way, my proposal implements a static placement of pools that reduces the risk of fruitless data movements while observing the dynamics of object creation and destruction.

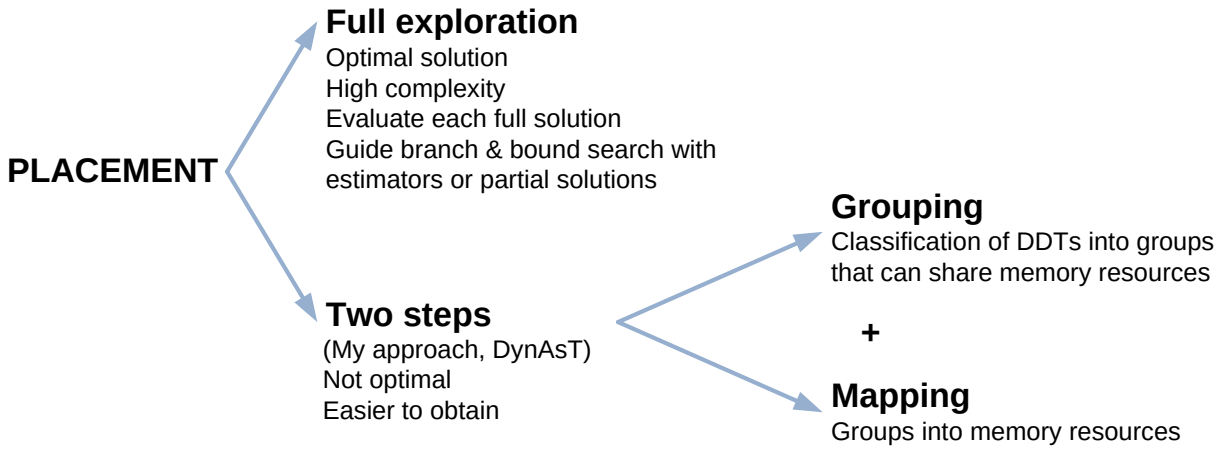


Figure 2.2.: My methodology proposes as a heuristic splitting the original placement problem into two parts that are solved independently. The first one consists on identifying DDTs that can share the same resources; the second assigns the available memory resources to those groups.

Second, I propose to break the original problem into two steps that are solved independently: The aforementioned classification of DDTs that can share a resource into groups, and placement of those groups over the memory resources (Figure 2.2). This constitutes in proper terms a heuristic akin to the ones used to solve other complex forms of the assignment family of problems. Although not guaranteeing optimal solutions, it achieves very good results when compared with traditional options, as shown in the experiments of Chapter 4. Additionally, most of the hard work can be done at design time, limiting hardware support or computations during run-time. The outcome is a process that provisions dedicated space for the instances of the most accessed DDTs, while keeping resource utilization high.

The grouping step analyzes the DDTs of the application and creates groups according to their access characteristics and the evolution of their footprint along execution time. It is a platform-independent step that builds groups according only to the characteristics of the DDTs and the values assigned by the designer to a set of configurable parameters, enabling a generic grouping of DDTs at design time, while exact resources are assigned later. Contrary to problems such as bin packing, the grouping algorithm does not impose a limit on the number or size of the containers (groups): If the designer forces a limit on their number, the DDTs that are not accepted into any group are simply pushed into the last one.

The second step, mapping, is concerned with the correspondence of divisible objects (pools) over a (reasonable) number of containers, without caring for the interactions among the entities that form each object. Thus, the mapping step assigns memory resources to each group using a fractional knapsack formulation.[4] This opens the door for a run-time implementation that places the groups according to the resources available in the exact moment of execution and for that concrete system considering, for instance, the possibility of different instantiations of a general platform or a graceful reduction of performance due to system degradation along its lifetime.

[4]Increasing the abstraction level at which a problem is solved may introduce some inefficiencies, but it offers other advantages. For example, whereas individual variables cannot be broken over non-contiguous memory resources, dynamic memory pools can – an object allocated in a pool cannot use a memory block that spawns over two disjoint address ranges, though, but this can be solved by the DMM at the cost of possibly slightly higher fragmentation. In this way, the placement of divisible pools can be solved optimally with a greedy algorithm.

This scheme works well when the application has DDTs with complementary footprints or if the number of instances extant at a given time of each DDT is close to the maximum footprint of the DDT. In other cases, compatible DDTs may not be found; hence, the exploitation ratio of the memory resources will decrease and performance will suffer. In comparison with other techniques such as hardware caches, a part of the resources might then be underused. Whether the explicit placement implemented through grouping and mapping is more energy or performance efficient than a cache-based memory hierarchy in those situations or not – avoiding data movements under low-locality conditions can still be more efficient even if the resources are not fully exploited all the time – is something that probably has to be evaluated on a case-by-case basis.

It is also interesting to remark that, as the break up into the steps of grouping and mapping is a form of heuristic to cope with the complexity of placement, so are the specific algorithms that I use to implement each of these steps. For example, to limit the cost of finding solutions the algorithm that I propose for grouping uses a straightforward method with some parameters that the designer can use to refine the solutions found. Future research may identify better options for each of the algorithms that I propose.

2.3. Designing a dynamic memory manager for data placement with DDT grouping

My proposal relies on a dynamic memory manager that uses knowledge on request sizes, DDT characteristics and compatibilities between DDTs (i.e., grouping information) to assign specific memory resources to each data object at allocation time. This section explores how all these capabilities can be incorporated into the dynamic memory manager.[5] The final goal is to supply the DMM with the information required for data placement:

- DDTs that can be grouped.
- Correspondence between DDTs/pools and memory resources.
- The data type of each object that is allocated.

Figure 2.3 presents the choices during the construction of a DMM that concern specifically data placement. The decision trees shown in the figure are independent from each other: It is possible to pick an option at one of the trees and combine it with any other decisions from the rest of them. However, the decision taken at one tree can influence the implementation of the decisions taken in the next ones. I elaborate more on the ordering of choices in Section 2.3.4 after all of them are presented.

2.3.1. Placement according to memory resources

The first decision, it may seem obvious in this context, is whether the dynamic memory manager should implement data placement or not (Figure 2.3(a)). That is, whether the DMM has to take into account the characteristics of the memory resources underlaying the available memory blocks and the object that will be created, or ignore them and treat all the blocks as equal in this regard – as is the most frequent case with general purpose managers for desktop systems.

[5] A comprehensive analysis of the full design space for dynamic memory managers, without data placement, was presented by Atienza et al. [Ati05, AMM+06b].

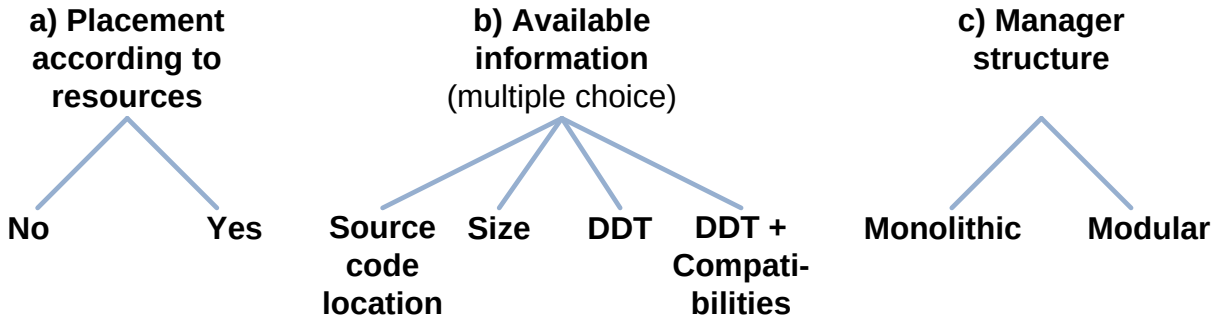


Figure 2.3.: Decisions relevant for data placement during the design of a dynamic memory manager: a) Has the DMM to include data placement capabilities? b) What kind of information is available for the DMM at allocation time? c) Does the DMM take global decisions or is it a modular collection of particularly optimized DMMs?

If the DMM does not have to care about data placement, its focus will be mainly to reduce wasted space due to fragmentation (similar to the scraps produced while covering a surface with pieces of restricted sizes), time required to serve each request (usually linked to the number of memory accesses performed by the manager to find the right block) and the overhead introduced by the manager itself with its internal data structures. Those structures are usually built inside the heap taking advantage of the space in free blocks themselves, but frequently require also some space for headers or footers in used blocks (e.g., the size of a used block may be stored in front of the allocated space) and thus impose a lower limit on the size of allocated blocks. Depending on the available information, the memory manager may rely on independent lists for blocks of different sizes or even different heaps for each DDT. Even if the DMM does not produce the data placement, other software techniques such as array blocking or prefetching can still be used for dynamic objects with predictable access patterns such as dynamically allocated vectors.

If the duties of the DMM include data placement, it has to place each object according to its size and number and pattern of accesses. The different granularity levels at which placement can be implemented have already been discussed in Section 2.1.3; the conclusion was that placement of individual instances seems unfeasible and thus, some aggregation must be used. This creates the need for additional information to group data objects.

2.3.2. Available information

This decision defines the amount of information that is available to the dynamic memory manager during an allocation. The tree in Figure 2.3(b) shows several non-exclusive options that can be combined according to the needs of the design. Traditionally, the DMM knew only the size of the memory request and the starting address of the blocks being freed because that was enough for the tasks entrusted to the dynamic memory mechanism of most general purpose languages. Typical considerations were building independent lists of blocks based on sizes to find a suitable free block quickly or the inclusion of coalescing and splitting to reduce the impact of fragmentation.

That simple interface is clearly not enough to implement data placement through the DMM because it would not have enough information to distinguish and allocate in different heaps instances of the same size but from different DDTs. Without the ability to exploit additional information, the designer has three main options: a) Place all the dynamic objects in a single

pool and rely on caching at run-time; b) place all the dynamic objects in a single pool, measure the total number of accesses and use static-data techniques to assign resources to the whole pool; and, c) analyze the amount of accesses for each allocation size and assign resources accordingly, disregarding that instances of different DDTs can receive each a quite different number of accesses even if they have a similar size.

A second possibility is that the DMM receives the size of the request and the type of the object that is being allocated (or destroyed). With that information the DMM can use the characterization of the typical behavior of the instances of the corresponding DDT, such as allocation and deallocation patterns and mean number of accesses to each instance, and assign resources from specific heaps to the instances of different DDTs. Here lies a key insight to implement data placement using the dynamic memory manager: The dynamic memory manager needs to know the type of the objects on which it operates to establish a correlation with the known properties of their DDTs. However, I explained previously that this approach has a serious drawback: Heaps assigned in exclusivity to instances of a single DDT can undergo significant periods of underutilization and those resources cannot be used to create instances of other DDTs.

In this work I propose a third option: Providing the DMM with knowledge about the compatibilities between DDTs. With that information, the DMM can implement a controlled amount of resource sharing to increase exploitation without impacting performance – as it will be able to limit the likelihood of seldom accessed DDTs blocking resources from more critical ones. This option allows implementing a trade-off between exclusivity (in resource assignment) and exploitation. In compensation for the need to modify the dynamic memory API to include the additional information, the run-time implementation of this mechanism requires no additional hardware at run-time.

Finally, Figure 2.3(b) shows also a fourth option: Making the source-code location of the call to the dynamic memory API available to the DMM itself. Although this information is not exploited by the methodology that I propose in this text, it could be used in the future to distinguish instances of the same DDT with dissimilar behaviors. For example, nodes of a container structure used in different ways in various modules of the application. However, this information would not be enough on its own to cope with objects – such as dynamically-allocated arrays – whose placement needs to change according to the size of each instance. Location information can be easily obtained in languages such as `C` or C++ through macros. For example, hashing the values of the macros `__FILE__`, `__LINE__` and `__func__`.

2.3.3. Manager structure

A pool can be managed as a global monolithic entity or as a collection of individual pools (Figure 2.3(c)). A monolithic pool consists of a single all-knowing dynamic memory manager handling one or more heaps of memory. The space in the heaps can be divided, for example, into lists of blocks of specific sizes; coalescing and splitting operations can move blocks between lists as their size changes. The DMM can take more global decisions because it knows the state of all the heaps.

On the contrary, a modular pool comprises several independent dynamic memory managers, each controlling one or more heaps. The main advantage is that each DMM can be tailored to the characteristics of the DDTs assigned to it, providing optimized specific data structures and algorithms for the heaps that it controls as if no other DDTs were present in the application. The DMM that has to serve each application request can be selected according to

the request properties (e.g., using the type of the object that is being allocated if DDT information is available) or through a cascaded mechanism where the DMMs are asked in turn if they can handle it.

Finally, modular designs that include a top layer to take global decisions are also possible. For example, a global manager that moves free blocks from the heap of the consumer thread to the heap of the producer thread in a producer-consumer scheme.

2.3.4. Order of choice

The order in which the trees of Figure 2.3 are considered is relevant because a choice preference is placed on the later trees by the former ones, and the most expensive options in a tree need only to be chosen if required by previous decisions. Therefore, I propose that the most useful order in the decision trees is determining first whether the DMM has to produce the placement of dynamic data objects or not. Then, the required amount of information can be determined – if that is not affordable, the next best option can be chosen, but the result may be suboptimal. Next, the designer can opt for a monolithic or modular approach. As the duties and complexity of the DMM increase, the most likely it becomes that the modular approach will be easier and more efficient.

Finally, the rest of decisions required to produce a working dynamic memory manager can be taken, as explained in previous work such as by Atienza et al. [Ati05, AMM+06b]: Use of coalescing and splitting, keeping independent lists for different block sizes, fixed block sizes or ability to adjust them (via splitting), order of blocks in the lists of free blocks (influences the effort needed to find a suitable block for every allocation), the policy for block selection (LIFO, FIFO, first fit, exact fit, next fit, best fit, worst fit), when to apply coalescing and/or splitting (at block request or discard), etc. Those decisions are evaluated once for a monolithic pool, or once for every DMM in a modular pool.

2.4. Putting everything together: Summary of the methodology

My proposal for placement of dynamic data objects constitutes an intermediate point between dynamic placement techniques, which can introduce an excess of data movements whose cost may not be compensated with a high access locality, and static placement of individual DDTs, which risks a low exploitation of resources. It consists on creating groups of DDTs with similar characteristics or dissimilar footprint requirements before performing the assignment of physical resources. As these processes are complex, I propose a set of heuristics to solve them efficiently; experimental data show the improvements that can be attained.

An independent dynamic memory manager controls the space assigned to each group, making up the application's pools. I further propose to use it to implement the calculated placement at run-time. This choice stems from the fact that the size and creation time of dynamic objects is unknown until the moment when they are allocated. Therefore, it seems appropriate to seize that chance to look for a memory block not only of the appropriate size, but also in the most suitable memory resource. Nevertheless, the DMM can affect seriously the performance of the whole system in some applications, so the process must introduce as little overheads as possible.

Extensive profiling and analysis supplies information about the DDT characteristics and the analysis of compatibilities. Object size and type are provided to the DMM (in the case of C++) via an augmented dynamic memory API. To this end, class declarations in the application are

instrumented (as explained in Section 2.5) to first profile the application and, then, transparently implement the extended API. At run-time, the DMM uses this extra information to select the appropriate pool to serve each request. The selection mechanism can be implemented as a simple chain of selections favoring the common cases or, in more complex situations, as a lookup table and should not introduce a noticeable overhead in most cases.

I propose to use a modular structure not only to ease the creation of DMMs specifically tailored for the DDTs in a pool, but also because the very structure of a modular DMM includes implicitly the knowledge about the DDTs that can be grouped together: The DDTs in a group are all processed by the DMM of the corresponding (sub)pool. In other words, two DDTs share memory resources if they are in the same group. Each (sub)DMM can look for blocks to satisfy memory requests disregarding object types because the mapping phase provided it with the most appropriate memory resources for its assigned DDTs and it will not be asked to allocate extraneous ones. Thus, the grouping information generated at design time can be exploited at run-time with a simple mechanism for DMM selection.

The choice of a modular design has two additional benefits. First, free blocks in a pool cannot be used by a different DMM to satisfy any memory request, which is a desirable effect because it saves further mechanisms to reclaim resources when more important objects are created. Second, the DMM of a pool that hosts objects of several sizes may use coalescing and splitting as suitable, disregarding the block sizes served by other pools.

The following sections describe in detail each of the steps of the methodology as they are implemented in $\mathcal{DynAsT}$ (Figure 1.10).

2.5. Instrumentation and profiling

In this section I document the instrumentation techniques specifically used for my work on data placement, which have the dual purpose of profiling the applications during the design phase and providing extended (DDT) information for the DMM at run-time. A more general approach for the characterization of the dynamic data behavior of software applications is documented in Chapter 5 as part of a framework for extraction of software metadata.

These techniques allow extracting data access information at the data type abstraction level with minimal programmer intervention. Obtaining information at the data type level is important to implement data type optimizations [PAC06] and dynamic data object placement on memory resources. In turn, reduced programmer intervention is important, particularly during early design phases, to minimize the overhead of adapting the instrumentation to later modifications that may affect significantly the structure of the code. The impact of instrumentation is limited to:

- The inclusion of one header file (.hpp) per code file (.cpp) – avoidable if precompiled headers are used.
- The modification of one line in the declaration of each class whose instances are created dynamically.
- Every allocation of dynamic vectors through direct calls to `new`.
- Every direct call to `malloc()`.

The instrumentation phase requires modifying those classes whose instances are created dynamically so that they inherit from a generic class that overloads the `new` and `delete` operators and parameterizes them with a univocal identifier for each DDT. Direct calls to

the functions `malloc()` and `free()` need also to be modified to include such a univocal identifier. That modifications alone are enough to track the use of dynamic memory in the application. However, to profile also accesses to data objects, additional measures are required. Section 5.4.1 presents a template-based technique to profile data accesses in an application. However, I developed an alternative mechanism based on virtual memory protection to profile data accesses avoiding further source code modifications – which would anyways be useless for the final deployment. In this section, I explain the basics of the template-based mechanism for extraction of allocation information and the use of virtual memory support to obtain data access information without additional source code modifications.

During the profiling phase, the designer has to identify the most common execution scenarios and trigger the application using representative inputs to cover them. This enables the identification of different system scenarios and the construction of a specific solution for each of them. The instrumentation generates at this stage a log file that contains an entry for every allocation and data access event during the application execution. This file, which generally has a size in the order of gigabytes (GB), is one of the inputs for $\mathcal{DynAsT}$.

At run-time, the same instrumentation is reused to supply the DMM with the DDT of each object allocated or destroyed. Therefore, the overhead imposed by the methodology on the designers should be minimal given the significant improvements that can be attained. Deployment in environments different to C++ is possible given an equivalent method to supply DDT information to the DMM.

2.5.1. Template-based extraction of allocation information

Section 5.4.1 presents comprehensive techniques to characterize the behavior of an application. Here, I summarize briefly the minimum instrumentation required to record just the events related to allocation operations. The code fragments shown here differ syntactically from those presented in Section 5.4.1 to simplify this explanation and illustrate how easily they can be used.

The following fragment of code shows the class that implements the logging of allocation events:

```
template <int ID>
class allocatedExceptions {
public:
  static void * operator new(const size_t sz) {
    return logged_malloc(ID, sz);
  }
  static void operator delete(void * p) {
    logged_free(ID, p);
  }
  static void * operator new[](const size_t sz) {
    return logged_malloc(ID, sz);
  }
  static void operator delete[](void * p) {
    logged_free(ID, p);
  }
};
```

The class `allocatedExceptions` defines class-level `new` and `delete`. During profiling, the original requests are normally forwarded (after logging) to the system allocator through the underlying `malloc()` and `free()` functions. In contrast, at run-time the requests are handled

by the custom memory manager used to implement data placement. Thus, the promise of serving both purposes with the same instrumentation is fulfilled.

To instrument the application, the designer needs only to modify the declaration of the classes that correspond to the dynamic data objects of the application so that they inherit from `allocatedExceptions`, parameterizing it with a unique identifier. No other lines in the class source code need to be modified:

```
class NewClass : public allocatedExceptions<UNIQUE_ID> {
  ...
};
```

2.5.2. Virtual memory support for data access profiling

The human effort of propagating the modifications required to implement the template-based technique for profiling of dynamic-data accesses presented in Section 5.4.1 may be significant, particularly if the application aliases pointers to objects frequently. To tackle this issue, I developed the technique presented here that trades human effort for execution performance during profiling. The main drawback of this technique is that it requires support for virtual memory in the platform. If the target platform lacks this functionality, it may still be possible to get an approximation of the overall application behavior using a different platform.

2.5.2.1. Mechanism

This technique, from now on referred as "exception-based profiling," consists on creating a big heap of memory and removing access permissions to all its memory pages. An exception handler is defined to catch subsequent application accesses. To manage the space in the heap, a custom DMM is designed and used by the application through the normal instrumentation. When the application accesses a data object in the heap, the processor generates automatically an exception. The exception handler uses the information provided by the operating system in the exception frame to identify the address that was being accessed by the application (and whether it was a read or a write access). Then, it enables access to the memory page containing that address. Before instructing the operating system to retry the offending instruction in the application, the exception handler activates a special processor mode that generates an exception after a single instruction is executed. This mechanism is commonly used by debuggers to implement "single-step execution."

After the data-access instruction is executed by the application, the processor generates an exception and the exception handler recovers control. The next action is to revoke again access permissions to the heap page accessed by the application. Then, the single-step mode is disabled, the access is recorded in the log file that contains the profiling information and execution of the application is resumed normally until the next access to a data object in the heap. Figure 2.4 illustrates the whole process.

Although those details are irrelevant for the purpose of profiling the number of data accesses, it may be interesting to mention that the custom designed DMM uses a tree (specifically, an `std::multimap`) built outside of the heap that contains pairs <address, size> ordered by block address. This setup avoids that accesses performed by the DMM itself, which is yet to be defined, are included with the rest of accesses of the application.[6]

[6] A custom DMM is used instead of the standard one in Windows because if `HeapAlloc()` is asked to commit all the pages since the beginning (needed to access-protect them), allocations bigger than 512 KB for 32-bit

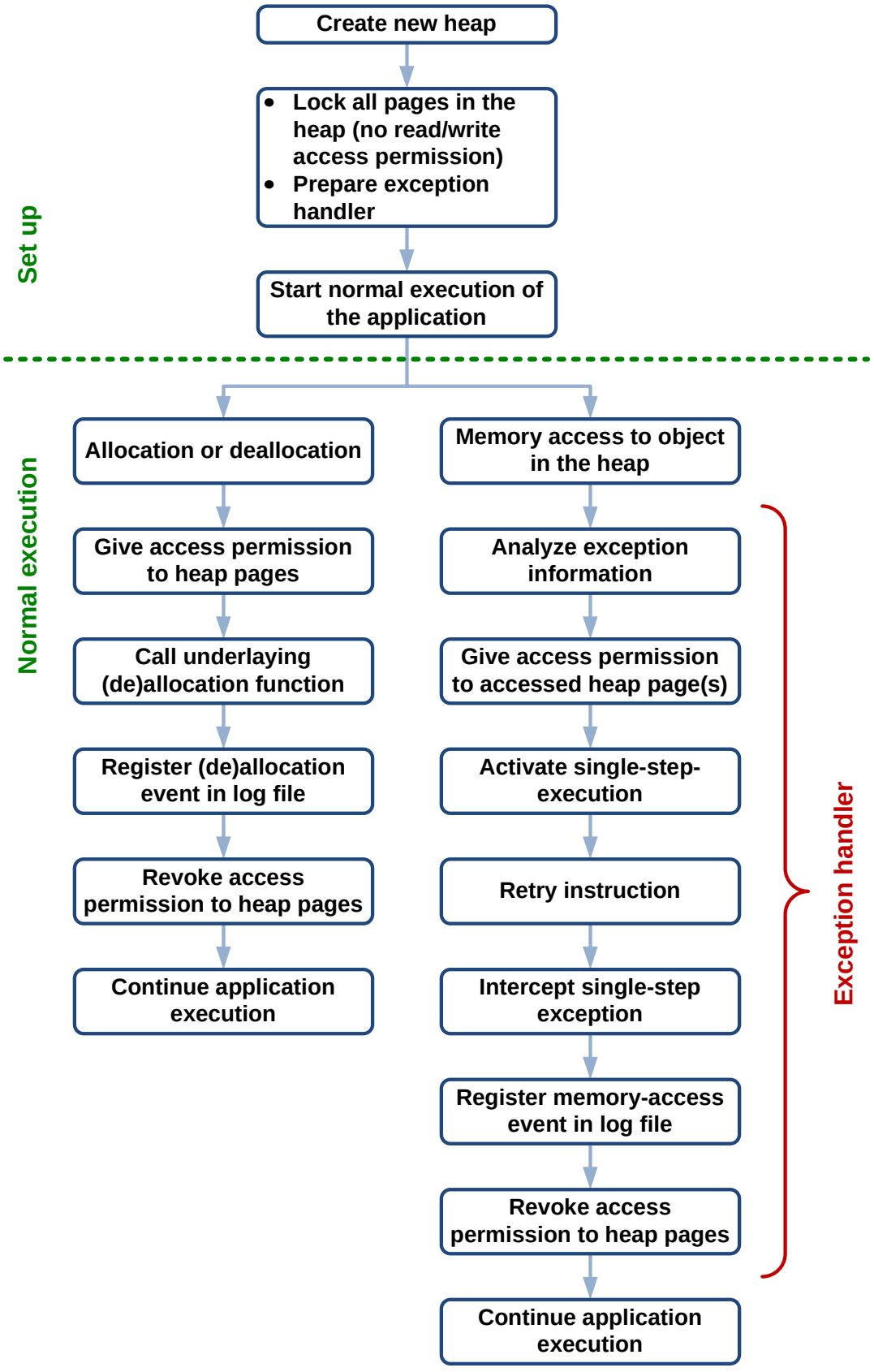


Figure 2.4.: Profiling with virtual memory support and processor exceptions.

2.5.2.2. Implementation

To add exception-based profiling to an application, the only modification required is to substitute its `main()` function with the one contained in the profiling library, renaming it. After initialization, the `main()` method provided by the library calls the entry point of the application. The following code fragments illustrate the implementation of the core methods in the library for the Microsoft Windows© operating system:[7]

```
static volatile void * theHeap;
static volatile bool inException = false;
static FILE * logFile;

int main(int argc, char ** argv) {
  unsigned long foo;

  // Create the heap
  theHeap = VirtualAlloc(NULL, HEAP_SIZE, MEM_RESERVE, PAGE_NOACCESS);
  if (theHeap == NULL)
    return -1;

  // Commit individual pages
  for (unsigned long offset = 0; offset < HEAP_SIZE; offset += 4096) {
    if (VirtualAlloc((unsigned char*)theHeap + offset, 4096, MEM_COMMIT,
                PAGE_READWRITE) == NULL)
      exit(-1);
  }

  // Protect heap pages
  VirtualProtect((void *)theHeapStart, HEAP_SIZE, PAGE_NOACCESS, &foo);
  inException = false;

  // Create logging file
  logFile = CreateLogFile(PATH_TO_LOG_FILE);
  atexit(CloseProfiling);

  // Start application inside an exception handler
  __try {
    MainCode(argc, argv);
  }
  __except (ResolveException(GetExceptionCode(),
          GetExceptionInformation())) {
    // The handler always resumes execution, so nothing to do here.
  }

  return 0;
}

...

void CloseProfiling() {
  unsigned long foo;
  ...
```

systems or than 1 MB for 64-bit ones will fail. In any case, it is fairly easy to substitute the custom DMM with the standard one using the functions `HeapCreate()` to create the heap and `HeapWalk()` to find its starting address.

[7] The source code presented through this work is for illustrative purposes only and should not be used for any other purposes. In particular, most checks for return error codes are omitted.

```
  fclose(logFile);

  // Unprotect heap pages so it can be deleted.
  VirtualProtect((void*)theHeapStart, HEAP_SIZE, PAGE_READWRITE, &foo);
  // Free the whole heap.
  VirtualFree((void*)theHeap, 0, MEM_DECOMMIT | MEM_RELEASE);

  ...
}

...

int ResolveException(int exceptCode, EXCEPTION_POINTERS * state) {
  unsigned long foo;

  // Extract context information for the exception
  EXCEPTION_RECORD * infoExcept = state->ExceptionRecord;
  CONTEXT * infoContext = state->ContextRecord;

  if (exceptCode == EXCEPTION_ACCESS_VIOLATION) {
    // First access attempt by the application
    if (inException) // Error: double virtual exception
      exit(-1);

    // Enable access for the correct memory page
    lastAddress = (void *)infoExcept->ExceptionInformation[1];
    VirtualProtect((void*)lastAddress, 16, PAGE_READWRITE, &foo);

    infoContext->EFlags |= 0x0100; // Activate single-step flag.

    // Register operation into log file.
    // If (infoExcept->ExceptionInformation[0] == 0), it was a read.
    // Otherwise, it was a write.

    inException = true;
    // Return to the application to retry the memory access
    return EXCEPTION_CONTINUE_EXECUTION;
  }
  else if (exceptCode == EXCEPTION_SINGLE_STEP) {
    // Single-step after the application executes the access

    // Protect again the affected memory page
    VirtualProtect((void*)lastAddress, 16, PAGE_NOACCESS, &foo);

    infoContext->EFlags &= 0xFEFF; // De-activate single step flag.

    inException = false;
    return EXCEPTION_CONTINUE_EXECUTION;
    // Return to the application and continue normal execution
  }
  else { // Unhandled exception
            exit(-1);
  }
}
```

Heap space is reserved once at the beginning of the execution to ensure a continuous range of addresses. However, its pages are committed individually because `VirtualProtect()`

works (empirically) much faster in that way.

2.5.2.3. Performance optimization

The performance of the exception-based profiling mechanism can be improved with a low-level trick. The previous implementation raises two exceptions for every access to an object in the heap. However, the designer can easily identify the most common operation codes that raise the access exceptions and emulate them explicitly in `ResolveException()`. In that way, a single exception is raised for the most common operations in the application.

This technique can also be used to tackle memory-to-memory instructions that might otherwise access two pages.[8] The following fragment of code shows how to emulate the opcode sequence "8B 48 *xx*," which corresponds in x86 mode to the instruction `mov ecx, dword ptr [eax + xx]`:

```
unsigned char * ip;
ip = (unsigned char *)infoContext->Eip;

// Examine the bytes at *ip
// 8B 48 xx mov ecx, dword ptr [eax + xx]

// Get the offset
signed char offset = (signed char)*(ip + 2);

// Read the value from memory
UINT32 * pValue = (UINT32 *)(infoContext->Eax + offset);

// Update the destination register
infoContext->Ecx = *pValue;

// Update eflags!
if (infoContext->Ecx == 0)
      SET_BIT(infoContext->EFlags, FLAG_ZERO);
else
      CLEAR_BIT(infoContext->EFlags, FLAG_ZERO);
if (infoContext->Ecx & 0x80000000)
      SET_BIT(infoContext->EFlags, FLAG_SIGN);
else
      CLEAR_BIT(infoContext->EFlags, FLAG_SIGN);

// Skip the emulated instruction
infoContext->Eip += 3;

// Protect the memory page and continue without
// entering the single-step mode
vRes = VirtualProtect(lastAddress, 16, PAGE_NOACCESS, &foo);
infoContext->EFlags &= 0xFEFF; // De-activate single step flag.
inException = false;
```

[8] Other possible solutions to this problem are identifying the corresponding opcodes and enabling accesses to both pages at the same time, or allowing nested access exceptions, granting and revoking permissions successively to both pages.

2.5.3. In summary

Profiling memory allocation operations can be done easily with multiple techniques and tools. However, profiling memory accesses with no dedicated hardware support in an exact way and without impact on application performance is more complex. In this text I propose two different techniques for profiling memory allocations and memory accesses, each with its own advantages and disadvantages, that can be reused to pass data type information to the DMM at run-time.

Both techniques may alter slightly the behavior of the application. The template-based technique will record every access, including those to variables that would normally be kept in processor registers. With the exceptions-based technique, processor registers are used normally and accesses to variables (or class members) stored in them will not be recorded, which may be relevant if the initial profiling is performed on a different processor architecture than that used in the final platform. However, due to the nature of dynamic structures and the extra level of indirection used to access dynamic objects through a pointer (or reference), it is quite possible that the interference of any of these techniques is minimal.

2.6. Analysis

The analysis step, which is the first one in $\mathcal{DynAsT}$, is based on the techniques presented in Chapter 5. It extracts the following information for each DDT:

- **Maximum number of instances** that are concurrently alive during the application execution.
- **Maximum footprint:** Size of data element $\times$ maximum number of instances alive.
- **Number of accesses** (reads and writes). This information is extracted for every single instance, but is later aggregated for the whole DDT because $\mathcal{DynAsT}$ does not currently process instances individually.
- **Allocation and deallocation sequence,** counting every instance of the DDT ever created.
- **Frequency of accesses per byte (FPB)**: Number of accesses to all the instances of a DDT divided by its maximum footprint.
- **"Liveness:"** Evolution of the DDT footprint along execution time as a list of footprint variations.

The analysis tool can distinguish between instances of the same DDT created with different sizes (e.g., multiple instances of a dynamically-allocated array created at the same location in the source code). In the rest of this text this is used as a specialization of the DDT concept, that is, different sized instances are considered as distinct DDTs.

The analysis algorithm is straightforward: Each packet in the log file produced during profiling is analyzed. For every allocation event, an object representing the allocated block is created and introduced in a tree (`std::map<>`) ordered by allocation address (Figure 2.5(a)) – that is, the address assigned in the original application. Then, for every memory access event the algorithm looks in the tree for the block that covers the accessed address and updates its access counters. Finally, when a deallocation event is found, the object representation is extracted from the tree and destroyed. This mechanism allows identifying the instance (and its DDT) that corresponds to each memory access recorded in the log file, tracking the addresses that were assigned to each one during the execution of the original application.

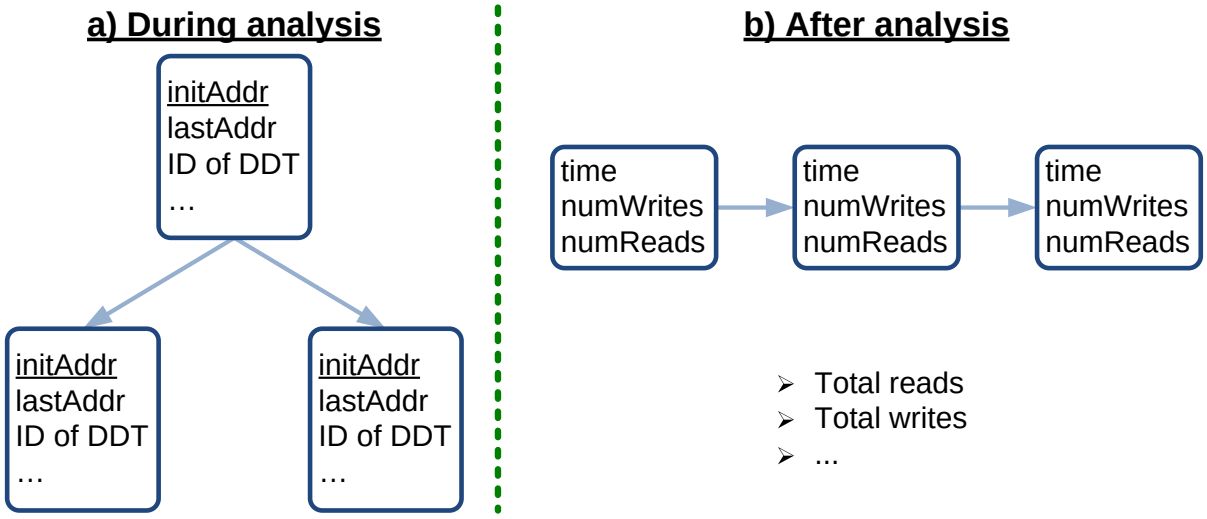


Figure 2.5.: Data structures during the analysis step: a) Ordered tree with blocks active up to the current profiling event. b) Behavior for each DDT. In this context, "time" refers to the number of allocation events since the beginning of the execution.

The output of the analysis step (Figure 2.5(b)) is a list of allocation events for each DDT, where each event includes the number of read and write accesses to all the DDT's alive instances since the previous event for that DDT. This list is deemed as the DDT behavior. Liveness information, which can be extracted directly from the allocation events in the DDT behavior, is used during the grouping step to identify DDTs that can share the same pool.

The use of an independent list of events for each DDT, instead of a single list for all the DDTs in the application with empty entries for the DDTs that do not have footprint variations on a given instant, reduces the memory requirements of the analysis tool itself and improves its access locality, which is an interesting issue on its own to reduce the time required to complete or evaluate a design.[9]

2.7. Group creation

As I have explained previously, placement of dynamic data objects over a heterogeneous memory subsystem is a hard problem. The very dynamic nature of the objects makes foreseeing techniques to produce an exact placement highly unlikely. For that reason, I have proposed a methodology based on three concepts: First, performing placement at the DDT level. Second, analyzing the properties of the DDTs to group those with similar characteristics and place them indistinctly. Third, mapping the resulting groups into memory resources according to their access characteristics.

Grouping is the central idea in this approach. It selects a point between assigning each DDT to a separate group (optimal placement, worst exploitation of resources) and assigning all of them to a single group (no specific placement, best exploitation of resources). DDTs assigned to different groups are kept in separate pools and, if possible, they will be placed in different memory resources during the mapping step. Similarly, DDTs with complementary characteristics that are included in the same group will be managed in the same pool; thus,

[9]Interestingly, some details of *DynAsT*'s internal implementation profited from the profiling and analysis of the tool itself with techniques similar to the ones presented in Chapter 5.

their instances will be placed on the resources assigned to the pool indistinctly.

Grouping has two main tasks: First, identifying DDTs whose instances have similar access frequencies and patterns, so that any of them will benefit similarly of the assigned resources. Second, balancing between leaving resources underused when there are not enough alive instances of the DDTs assigned to the corresponding group, and allowing instances from less accessed DDTs to use better memory resources when there is some space left in them. The grouping algorithm approaches this second task analyzing DDTs that have complementary footprint demands along time, so that valleys in the footprint of some DDTs compensate for peaks of the others and the overall exploitation ratio of the memory resources is kept as high as possible during the whole execution time.

To reduce the complexity of grouping, the approach that I present here is based on a greedy algorithm that offers several "knobs" that the designer can use to steer the process and adapt it to the specific features of the application under development. Even if this solution is not optimal, in Chapter 4 I show the performance advantages that can be attained for several case studies.

2.7.1. Liveness and exploitation ratio

The following two metrics are relevant to minimize the periods when memory resources are underused:

- **Group liveness:** Similarly to the case of individual DDTs, the liveness of a group is the evolution along time of the combined footprint of all the DDTs that it contains. It is implemented in $\mathcal{DynAsT}$ through a list of memory allocation events that represents the group "behavior."
- **Exploitation ratio:** The occupation degree of a group (or pool) along several time instants:

$$Exploitation\ ratio = \frac{\sum_{t=1}^{N} \frac{Required\ \ footprint(t)}{Pool\ \ size}}{N}$$

In essence, the exploitation ratio provides a measure of how well the grouping step manages to combine DDTs that fill the space of each memory resource during all the execution time. During the grouping step, as the size of each group is not yet fixed, it is calculated as the occupation at each instant respect the maximum footprint required by the DDTs already included in the group. In this way, the grouping algorithm tries to identify the DDTs whose liveness is complementary along time to reduce their combined footprint, but it may also add DDTs with a lower FPB to a group if the exploitation ratio improves and the total size does not increase (more than a predefined parameter) – the "filling the valleys" part. During simulation, the exploitation ratio can be calculated for the final pools (which do have a fixed size) to check the effectiveness of a solution.

These concepts can be illustrated using a hypothetical application with two threads as an example. The first thread processes input events as they are received, using DDT_1 as a buffer. The second one consumes the instances of DDT_1 and builds an internal representation in DDT_2, reducing the footprint of DDT_1. Accordingly, the footprint of DDT_2 is reduced when the events expire and the related objects are deleted. Figure 2.6(a) shows the liveness of each DDT. The maximum footprints of the DDTs are 7 KB and 10 KB, respectively. Therefore, if both DDTs were placed independently (that is, in two different pools), the total required footprint would

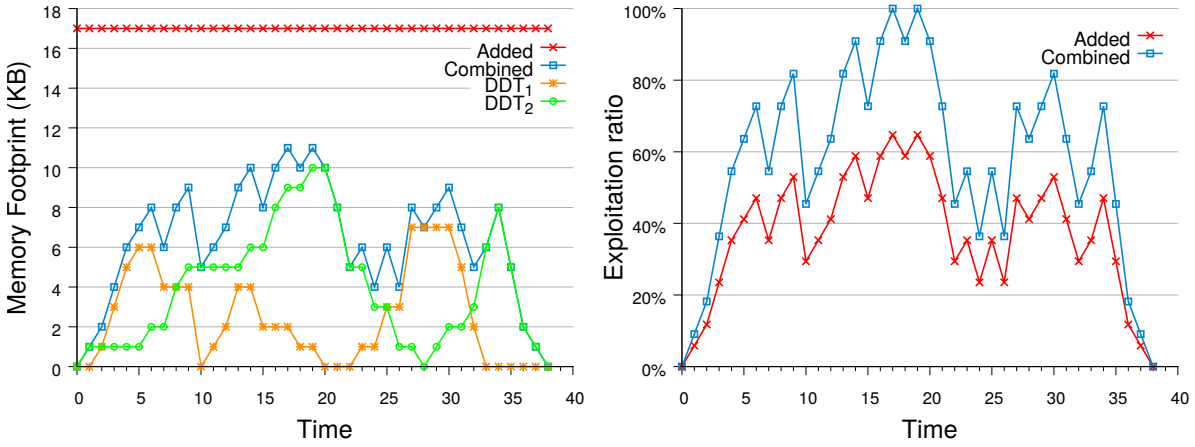


Figure 2.6.: Liveness and exploitation ratio for two hypothetical DDTs. a) Liveness (evolution of footprint) for the two DDTs considered independently, their added maximum footprints and the footprint of a group combining them. b) Exploitation ratio for the case of an independent group for each DDT (38.2 % on average) and for one combined group (59.1 % on average).

be 17 KB (labeled as "Added" in the figure). However, grouping both DDTs together reduces the maximum required footprint to 11 KB (labeled as "Combined"). Figure 2.6(b) compares the exploitation ratio of the group with the exploitation ratio that would result if the two DDTs were kept apart. In this hypothetical case, grouping would increase the exploitation ratio of the memory resources, thus reducing the required footprint. These two DDTs can be kept apart from the rest of DDTs of the application, or they might be further combined with others.

2.7.2. Algorithm parameters

The designer can use the following parameters to guide the grouping algorithm:

1. **MaxGroups.** The maximum number of groups that the algorithm can create. This parameter controls the trade-off between minimizing memory footprint (for a value of 1) and creating multiple pools (for values bigger than 1). It should be at least as big as the number of memory modules of distinct characteristics present in the platform; otherwise, the chance to separate DDTs with different behaviors into independent memory modules would be lost.

   In comparison with the bin-packing problem, the grouping algorithm does not try to fit all the DDTs in the maximum allowed number of groups; instead, DDTs that are not suitable for inclusion in any group are simply pushed into the last group, which contains the DDTs that will probably be placed on main memory.

2. **MaxIncMF$_G$.** The maximum ratio that the footprint of group G is allowed to increase when a DDT is added to that group. This parameter introduces some flexibility while combining DDTs, which is useful because their footprints will usually not be a perfect match.

3. **MinIncFPB.** Minimum ratio between the old and the new FPB of a group that needs to be achieved before a DDT is added to that group. The default value of 1.0 allows any DDT that increases the FPB of the group to be included.

4. **MinIncExpRatio.** The minimum increase in the exploitation ratio that allows a DDT with a lower FPB to be included in a group. The default value of 1.0 allows any DDT that increases the exploitation ratio of the group to be included in it.

5. **ExcludedDDTs.** If the designers have good knowledge of the application, they can decide to exclude a DDT from the grouping process and manually push it into the last group for placement on main memory.

The combination of MaxIncMF, MinIncFPB and MinIncExpRatio allows balancing between increasing the exploitation ratio of the pools, maximizing the FPB of their DDTs and bounding the number of distinct DDTs that can be added to a group. As an example, a high MaxIncMF and a MinIncFPB of 1.0 will favor grouping many DDTs together. Increasing the value of MinIncFPB will prevent the addition of DDTs with very few accesses, even if they apparently match the valleys in the footprint of the group (this can be useful, for example, if the designers know that some DDT has a very dynamic behavior that is difficult to capture during profiling).

2.7.3. Algorithm

Algorithm 1 presents the pseudo-code for the grouping process. The main idea is building groups until all the DDTs have been included in one or the maximum number of groups is reached. Each time the algorithm builds a new group, it checks all the remaining DDTs in order of descending FPBs. If the DDT is compatible with the current contents of the group, that is, its peak requirements match the footprint minima of the group, then it is included. Otherwise, it is discarded and the next one is considered. This increases the exploitation ratio of the group as more instances will be created using the same amount of resources. The available parameters allow relaxing the restrictions applied for joining DDTs to existing groups so that, for instance, a small increase on the group footprint is allowed for DDTs with similar FPBs that do not have a perfectly matching liveness.[10]

The DDTs are evaluated in order of decreasing FPB to ensure that if a DDT matches the liveness of a group, it is the DDT with the highest FPB among the remaining ones – thus, observing the heuristic of placing first the DDTs with the highest density of accesses per size unit. Once the FPB of the resulting group is known, a check is made to verify that a minimum increment is achieved. This is useful for instance to avoid including DDTs with low FPBs that could hinder the access pattern of the group once it is placed into memory resources such as DRAMs.

The group liveness is kept updated as a combination of the liveness of the DDTs in it. This ensures that no comparisons between individual DDTs, but between the DDTs and the groups, are executed, reducing the algorithm complexity. To evaluate the inclusion of a DDT in a group, the new combined behavior is calculated (lines 22–34). This is a straightforward process that involves combining two ordered lists (the lists of allocation events of the group and the new DDT) and accounting for the accumulated footprint and accesses of the group and the DDT. Then, the constraints imposed by the grouping parameters are evaluated for the new behavior (lines 16–21). New criteria such as the amount of consecutive accesses or specific access patterns can be easily incorporated into this check in the future.

[10] This is one example of the difficulty of the grouping problem: What is a better option, to combine those two DDTs with similar FPBs at the expense of increasing the size of the group (i.e., requiring more resources during mapping), or leave them apart in case that other DDT with a better footprint match with the group may be found later?

Algorithm 1 Grouping

```
1: function Grouping(DDTs : List of DDTs) : List of Groups
2:   Order the DDTs on descending FPB
3:   Exclude the DDTs that were marked by the designer (ExcludedDDTs)
4:   While there are any DDTs remaining and MaxGroups is not reached do
5:     Create a new group
6:     For each remaining DDT do
7:       Calculate the liveness and FPB that would result if the DDT were
           added to the group (CalcNewFootprintAndFPB)
8:       If the new behavior passes the tests in CheckCompatibility then
9:         Add the DDT to the group
10:        Remove the DDT from the list of DDTs
11:  Push any remaining DDTs into the last group
12:  Add the DDTs that were excluded to the last group
13:  Order the groups on descending FPB
14:  Return the list of groups
15: end function

16: function CheckCompatibility(newBehavior : Behavior) : Boolean
17:   Return true if the new footprint does not exceed the maximum footprint for any group
18:     and the footprint is not incremented more than MaxIncMF_G
19:     and the FPB of the group is increased by at least MinIncFPB
20:     and the exploitation ratio is increased by at least MinIncExpRatio
21: end function

22: function CalcNewFootprintAndFPB(Group, DDT) : Behavior
23:   Create a new behavior
24:   While there are events left in the behavior of the group or the DDT do
25:     Select the next event from the group and the DDT
26:     If both correspond to the same time instant then
27:       Create a new event with the sum of the footprint of the group and the DDT
28:     Else if the event of the group comes earlier then
29:       Create a new event with the addition of the current footprint of the group
            and the last known footprint of the DDT
30:     Else // ( if the event of the DDT comes earlier )
31:       Create a new event with the addition of the current footprint of the DDT
            and the last known footprint of the group
32:   Update the FPB with the maximum footprint registered and the sum of reads and writes
        to the group and the DDT
33:   Return new behavior
34: end function
```

The output of this step is a list of groups with the DDTs included in each of them and their aggregated characteristics (maximum footprint, liveness, FPB and exploitation ratio). The worst-case computational complexity of the algorithm is bounded by $\mathcal{O}(n^2m + nm)$, where n is the number of DDTs in the application and m is the number of entries in the liveness of the DDTs. However, as normally n is in the order of tens of DDTs while m is in the order of millions of allocation events per DDT, $m \gg n$ and the complexity is in practical terms closer to $\mathcal{O}(m)$.

Justification. The grouping algorithm presents two extreme cases. In the first one, it creates as many groups as DDTs in the application. In the second, all the DDTs require memory mostly at disjoint times and can be combined in a single pool. In the first case, the algorithm performs $\mathcal{O}(n^2)$ tests, each test requiring $\mathcal{O}(m + m)$ operations to produce the behavior of the new (hypothetical) group – merging two sorted lists with v and w elements respectively has a complexity of $\mathcal{O}(v + w)$. Thus results a complexity of $\mathcal{O}(n^2m)$.

However, in the second case the algorithm performs $\mathcal{O}(n)$ tests (one test for each DDT against the only extant group). Each of those tests requires $\mathcal{O}(nm + m)$ operations to generate the new behavior – the m events in the liveness list of the n DDTs accumulate in the liveness of the group, which is tested against the m elements in each newly considered DDT. Since the time of the tests dominates the time of sorting, the worst case complexity is $\mathcal{O}(n(nm + m)) \equiv \mathcal{O}(n^2m + nm)$.

2.8. Definition of pool structure and algorithms

The concept of pool represents in this methodology one or several address ranges (heaps) reserved for the allocation of DDTs and the data structures and algorithms needed to manage them. For every group from the previous step, a pool that inherits its list of DDTs is generated. During this step, considerations like the degree of block coalescing and splitting, choice of fit algorithms, internal and external fragmentation, and number of accesses to internal data structures and their memory overhead take place. The result of this step is a list of pools ordered by their FPBs and the description of the chosen algorithms in a form of metadata that can be used to build the memory managers at run-time.

The construction of efficient DMMs is a complex problem that, as outlined in Section 1.3.3, has been the subject of decades of study. Among other resources widely available in the literature, the Lea allocator used in Linux systems is described by Lea [Lea96], an extensive description of a methodology that can be used to implement efficient custom dynamic memory management is presented by Atienza et al. [AMM+06a, AMP+15] and a notable technique to efficiently manage a heap that has been placed on a scratchpad memory is presented by [MDS08]. Therefore, this process is not further described here.

Similarly, efficient composition of modular dynamic memory managers can be achieved with techniques such as the mixins used by Berger et al. [BZM01] and Atienza et al. [AMC+04b, AMM+06a] or through call inlining as used by Grunwald and Zorn in CustoMalloc [GZ93] – although the latter can only be used for objects whose size is known by the compiler, that is the case for common constructs such as struct (record) types.

As an example of the type of decisions involved during the construction of an efficient DMM, consider the case of a pool that receives allocation requests of 13 B and 23 B. The designer of the DMM can take several options. For instance, the DMM can have two lists

of free blocks, one for each request size. If the heap space is split between the two sizes, then the DMM does not need to add any space for the size of each block: The address range of the block identifies its size. However, in that case the DMM will not be able to reuse blocks of one size for requests of the other – in line with the idea that the DDTs in one pool can share the space assigned to it. To implement coalescing and splitting, the DMM will need to add a size field to each memory block. Assuming it uses 4 B for this purpose, the size of the blocks will be 17 B and 27 B, with an overhead of 31 % and 17 %, respectively. To complicate things further, coalescing two smaller blocks will create a block of 34 B that is slightly big for any request. Depending on the characteristics of the application and the success of the grouping process in finding DDTs with complementary liveness, the DMM may be able to coalesce many blocks of one size at some point during execution, recovering most of the space as a big area suitable for any number of new requests.

After such considerations, one could be tempted to execute the pool construction step before grouping, so that the DDTs in a pool are chosen to ease the implementation of the DMM. However, such decision would defeat the whole purpose of placement because DDTs with many accesses might be joined with seldom accessed ones, creating groups with lower overall FPB. Although this could be the topic of future research, in general it seems preferable to sacrifice some space to obtain better access performance – especially because the grouping step is designed with the specific purpose of finding DDTs with complementary liveness.

In summary, the considerations during the construction of the DMM are numerous and complex. Not being myself an expert on the construction of dynamic memory managers, in $\mathcal{DynAsT}$ this step is currently included only as a stub. Therefore, in its current implementation, the groups pass directly to the mapping step. The experiments presented in Chapter 4 using the simulator included in $\mathcal{DynAsT}$ use simple ideal DMMs that help to compare the relative quality of different placement options.

2.9. Mapping into memory resources

The mapping step produces a placement of the pools into the available memory resources. The input to this step is the ordered list of pools with their internal characteristics (e.g., size, FPB) and a description of the memory subsystem of the platform. In $\mathcal{DynAsT}$ this description is very flexible and allows specifying different types of organizations based on buses and memory elements. The result of this step is a list of pools, where each pool is annotated with a list of address ranges that represent its placement into memory resources. The computational complexity of the mapping algorithm is in the order of $\mathcal{O}(n)$, being n the number of pools – and assuming that the number of pools is higher or in the order of the number of memory resources.

The design of the mapping algorithm makes some assumptions. First, that pools can be split over several memory resources even if they are not mapped on consecutive addresses. This is technically feasible with modern DMMs at the expense of perhaps a slightly larger fragmentation – however, if the pools are mapped into memory blocks with consecutive memory addresses, this overhead disappears because the blocks become a single entity at the logical level. With this consideration, the mapping part of the dynamic data placement problem can be seen as an instance of the fractional (or continuous) knapsack problem [BB96], which can be solved optimally with a greedy algorithm. This is an important result because the mapping step could be moved in the future to a run-time phase, allowing the system to adapt

to different implementations of the same architecture or to achieve a graceful degradation of performance as the system ages and some components start to fail. It could even be useful to adapt the system to changing conditions, such as powering down some of the memory elements when energy is scarce (e.g., for solar-powered devices on cloudy days).

Second, the mapping step assumes that only one memory module can be accessed during one clock cycle. It is possible to imagine situations where the processor has simultaneous access to several independent buses or where at least slow operations can be pipelined, which may be useful for memories that require several cycles per access (DRAMs fall in this category for random accesses, but they can usually transfer data continuously in burst mode). However, to efficiently exploit those extra capabilities a more complex approach would be needed. In this regard, an interesting study, limited to static data objects, was presented by Soto et al. [SRS12]. This may constitute an interesting topic for future research.

The third assumption is that all the instances created in the pool have the same FPB. As explained in Section 2.1.3, discriminating between instances with a different number of accesses seems to be a complex problem, specially for short-lived instances created and destroyed in big numbers. For longer-lived ones, future research could consider hardware or compiler-assisted methods to identify very accessed instances (although such techniques would likely introduce unacceptable costs), and techniques similar to the ones used by garbage collectors to migrate them to the appropriate memories.[11]

Finally, the mapping algorithm does not take into account possible incompatibilities between pools. Such situations might arise, for example, when instances of DDTs assigned to different pools are accessed simultaneously during vector reduction operations. If the pools were both assigned to a DRAM, it would be preferable to place each of them in a different bank to reduce the number of row misses. Although the current implementation of *DynAsT* does not offer explicit support, the parameter `SpreadPoolsInDRAMBanks` can be used by the designer to manually implement this functionality in simple cases – indeed, such is the case in the experiments presented in Chapter 4. Section 7.3 elaborates more on how this issue could be approached with *DynAsT*.

2.9.1. Algorithm parameters

The mapping algorithm offers several parameters that the designer can use to tune the solution to the application:

1. **MinMemoryLeftOver.** Minimum remaining size for any memory block after placing a pool in it. If the remaining space is lower than this value, it is assigned to the previous pool. This parameter, which prevents dealing with tiny memory blocks, should be set to the minimum acceptable heap size for any pool (the default is 8 B, but imposing a bigger limit may be reasonable).

2. **Backup pool.** This parameter activates the creation of the backup pool. Accesses to objects in this pool will typically incur a penalty, as the backup pool is usually located in an external DRAM. Hence, although the backup pool adds robustness to the system, its presence should not be used as an excuse to avoid a detailed profiling.

[11] Generational pools can be used to identify long-lived objects. However, for data placement the interest is on identifying very accessed objects among those that are long-lived.

3. **PoolSizeIncreaseFactor.** The designer can adjust the amount of memory assigned to each pool with respect to the maximum footprint calculated through the analysis of group liveness. This allows balancing between covering the worst-case and reducing the amount of space wasted during the most common cases. However, the experiments show that usually the liveness analysis packs enough DDTs in each group to keep a high exploitation ratio; thus, a value of 1.0 gives normally good results. The designer may still use it to explore different conditions and what-ifs.

4. **PoolSizeIncreaseFactorForSplitPools.** When a pool is split over several memory blocks (with non-contiguous address ranges), it may be necessary to increase its size to overcome the possibly higher fragmentation. The default value is 1.0, but a value as high as 1.3 was needed in some preliminary experiments, depending on the size of the DDT instances.

5. **MappingGoal.** Memory modules can be ordered according either to access energy consumption or latency. Although the case studies in Chapter 4 use memory technologies that improve both simultaneously (increasing the area per bit cost), either goal may be specified explicitly.

6. **SpreadPoolsInDRAMBanks.** This parameter spreads the pools placed in the last DRAM module (including the backup pool, if present) over its banks according to FPB ordering. The idea is to increase the probability that accesses to DDT instances from different pools hit different DRAM banks, thus reducing the number of row misses (i.e., row activations). If there are more pools than DRAM banks, the remaining ones are all placed in the last bank – of course, better options that take into account the interactions between accesses to the pools could be explored in the future.

2.9.2. Algorithm

Algorithm 2 presents the pseudo-code for the mapping algorithm. In essence, it traverses the list of pools placing them in the available memory resources, which are ordered by their efficiency. After the placement of each pool, the remaining size in the used memory resources is adjusted; memory resources are removed from the list as their capacity is exhausted.

The mapping algorithm starts by ordering the memory resources according to the target cost function, be it energy or latency. The list of pools remains ordered since the grouping step. For every pool, the algorithm tries to map as much of it as possible into the first memory resource in the list. If the pool is bigger than the space left in the memory resource (lines 7–13), it is split and the remaining size goes for the next round of placement. However, splitting a pool can introduce some fragmentation. Therefore, the size of the remaining portion is rounded up to the size of an instance if the pool contains only one DDT; otherwise, the designer can specify a small size increase with the parameter `PoolSizeIncreaseFactorForSplitPools` (lines 10–13).[12]

On the contrary, if the remaining portion of a pool fits into the current memory resource, the placement of that pool is concluded; the remaining capacity of the resource is adjusted

[12] In platforms with many small memory resources, this adjustment can lead to an unbounded increase of the pool size. In such cases, either a smaller value should be used or the algorithm could be changed to increase the size only after the first time that the pool is split.

Algorithm 2 Mapping

```
 1: function MAPPING(List of pools, List of memory blocks, Cost function)
 2:   Order memory blocks using the cost function (energy / access time)
 3:   For each pool in the list of pools do
 4:     Multiply pool size by PoolSizeIncreaseFactor
 5:     While the pool is not fully mapped do
 6:       Select the first block in the list of available blocks
 7:       If the remaining size of the pool > available space in the block then
 8:         Assign the available space in the block to the pool
 9:         Remove the block from the list of memory blocks
10:         If the pool has only one DDT then
11:           Round up pool's remaining size to a multiple of the DDT instance size
12:         Else
13:           Increase the pool's remaining size by PoolSizeIncreaseFactorForSplitPools
14:       Else
15:         Assign the rest of the pool to the block
16:         If block is DRAM and SpreadPoolsInDRAMBanks then
17:           Assign the whole DRAM bank
18:         Else
19:           Reduce available space in the block
20:           If available space in the block < MinMemoryLeftOver then
21:             Assign everything to the pool
22:             Remove block from the list of memory blocks
23:   return (blocks, pools)
24: end function
```

appropriately (lines 15–22). When a pool is placed on a DRAM, the parameter `SpreadPools-InDRAMBanks` is checked; if active, the whole DRAM bank is assigned to the pool in exclusivity, regardless of its actual size. In the absence of a more elaborate mechanism, the designer can use this parameter to reduce the number of row misses when the DRAM has extra capacity. For the rest of memory resources (or if the parameter is not active) the algorithm makes a final check to avoid leaving memory blocks with very small sizes (line 20).

Finally, the mapping algorithm can produce an additional "backup pool," which will lodge all the instances that, due to differences in the actual behavior of the application in respect to the characterization obtained during the profiling phase or to excessive pool fragmentation, cannot be fit into their corresponding pools. If present, the backup pool uses all the remaining space in the last memory resource, usually a DRAM. When the mapping step finishes, each pool has been annotated with a set of tuples in the form (`memoryResourceID`, `startAddress`, `size`) that represent the position in the platform's address map of each fragment of the pool.

2.9.3. Platform description

Platform descriptions for mapping and simulation are provided to $\mathcal{DynAsT}$ through simple text files. As can be seen in the following examples, the syntax is very simple. It has one entry (in square brackets "[]") for every element in the memory subsystem. Valid memory types are "SRAM," "SDRAM" and "LPDDR2S2_SDRAM." Both DRAMs and SRAMs, that is, directly addressable memories, bear the label "Memory" and a unique identifier. The designer must define for them the attributes "PlatformAddress" and "Size," which define their position in the platform's address space, and "ConnectedTo," which defines their connection point in the memory subsystem. Each memory module has to define also a set of working parameters such

as voltage and timing that will be used during mapping (to order memory modules according to the target cost function) and simulation.

DRAMs require several additional parameters. For example, "NumBanks" (the number of internal memory banks) and "WordsPerRow" (the number of words in the rows of each bank) control the internal configuration of the memory, while "CPUToDRAMFreqFactor," which defines the memory-bus-to-CPU frequency ratio, and "CPUFreq," which defines the CPU frequency in Hz, are used to find the memory cycle time:

$$MemCycleTime = \frac{1}{CPUFreq \,/\, CPUToDRAMFreqFactor}$$

Cache memories use instead the label "Cache." Valid cache types are "Direct" and "Associative," with "NumSets" defining the degree of associativity. Valid replacement policies are "LRU" and "Random." $\mathcal{DynAsT}$ supports up to four cache levels per memory and multiple cache hierarchies. However, a minor limitation is currently that a cache hierarchy can be linked only to one memory module. Therefore, multiple memory modules require each their own cache hierarchy. Although this should not constitute a major applicability obstacle – most platforms contain a single DRAM module – modifying it should not represent a big challenge, either. Each cache memory defines its "Size," the length of its lines in memory words ("WordsPerLine"), its master memory ("ConnectedToBlockID") and its level in the local hierarchy ("Level"). The range of memory addresses covered by each cache memory is defined through the attributes "CachedRangeStart" and "CachedRangeSize;" using them, the designer can easily define uncached address ranges as well.

The current implementation of $\mathcal{DynAsT}$ assumes that SRAMs are always internal (on-chip) and never sport caches. However, although not a common practice in the design of embedded platforms nowadays, it would be very easy to update it to consider the case of external (off-chip) SRAMs used as main memories with (optional) internal caches. Similarly, on-chip DRAMs (i.e., eDRAM) and other memory technologies could also be included if needed in the future.

Finally, interconnections use the label "Interconnection," also with a unique identifier. Bus hierarchies can be defined through "RootConnection," with a value of 0 signifying that the bus is connected directly to the processor. The following fragment defines a platform with a 128 MB SDRAM and a 32 KB cache with associativity of 16 ways and lines of 4 words:

```
####################
[Memory=0]
# DRAM 128 MB
# Micron Mobile SDRAM 1 Gb
# (128 MB) MT48H32M32LF
# -6, 166 MHz, CL=3
Type="SDRAM"
ConnectedTo=1 # Interconnection ID
PlatformAddress=2147483648
Size=134217728

CPUToDRAMFreqFactor=(double)8.0
CPUFreq=(double)1332e6
NumBanks=4
WordsPerRow=1024

CL=3
tRP=3
tRCD=3
tWR=3
tCDL=1
tRC=10

vDD=(double)1.8
vDDq=(double)1.8
iDD0=(double)67.6e-3
iDD1=(double)90e-3
iDD2=(double)15e-3
iDD3=(double)18e-3
iDD4=(double)130e-3
iDD5=(double)100e-3
iDD6=(double)15e-3
cLOAD=(double)20e-12
```

```
####################
[Cache=0]
# L1-Data 32 KB
Type="Associative"
ReplacementPolicy="LRU"
ConnectedToBlockID=0
Level=0 # Caches must be kept oredered
# Address range cached by this memory:
CachedRangeStart=2147483648
CachedRangeSize=134217728
NumSets=16
Size=32768
WordsPerLine=4

EnergyRead=
   (double)0.02391593739800e-9
EnergyWrite=
   (double)0.02391593739800e-9
EnergyReadMiss=
   (double)0.02391593739800e-9
EnergyWriteMiss=
   (double)0.02391593739800e-9
DelayRead=1
DelayWrite=1  # All words in parallel
DelayReadMiss=1
DelayWriteMiss=1

####################
[Interconnection=1]
AcquireLatency=0
TransferLatency=0
RootConnection=0 # Processor
ConcurrentTransactions=0
TransferEnergy=(double)0.0
```

The text fragment below defines a platform with a low-power 256 MB DDR2-SDRAM and a 32 KB SRAM:

```
####################
[Memory=0]
# SRAM 32KB
Type="SRAM"
ConnectedTo=1 # Interconnection ID
PlatformAddress=0
Size=32768

EnergyRead=(double)0.00492428055021e-9
EnergyWrite=(double)0.00492428055021e-9
DelayRead=1
DelayWrite=1

####################
[Memory=1]
# LPDDR2-S2 SDRAM 256MB
# (64Mx32) at 333MHz (-3)
Type="LPDDR2S2_SDRAM"
ConnectedTo=1 # Interconnection ID
PlatformAddress=2147483648
Size=268435456

CPUToDRAMFreqFactor=(double)4.0
CPUFreq=(double)1332e6
NumBanks=8
WordsPerRow=2048
MaxBurstLength=4
DDR2Subtype="S2"

tRCD=6
tRL=5
tWL=2
tDQSCK_SQ=1
tDQSS=1
tCCD=1
tRTP=3
tRAS=14
tRPpb=6
tWTR=3
tWR=5
vDD1=(double)1.8
vDD2=(double)1.2
vDDca=(double)1.2
vDDq=(double)1.2
iDDO1=(double)0.02
iDDO2=(double)0.047
iDDOin=(double)0.006
iDD3N1=(double)1.2e-3
iDD3N2=(double)23e-3
iDD3Nin=(double)6e-3
iDD4R1=(double)0.005
iDD4R2=(double)0.2
iDD4Rin=(double)0.006
iDD4Rq=(double)0.006
iDD4W1=(double)0.01
iDD4W2=(double)0.175
iDD4Win=(double)0.028
cLOAD=(double)20e-12

####################
[Interconnection=1]
AcquireLatency=0
TransferLatency=0
RootConnection=0 # Processor
ConcurrentTransactions=0
TransferEnergy=(double)0.0
```

2.10. Deployment

Application deployment can be accomplished in two ways. The first one derives directly from the work previously done on DMM composition with mixins by Berger et al. [BZM01] and Atienza et al. [AMC+04b, AMM+06a, AMP+15]. Using a library of modular components, a DMM can be built for each pool using the characteristics determined during the pool construction step. Mixins lie on one end of the trade-off between efficiency and flexibility: Although they allow for a good degree of optimization, each DMM is assembled during compilation; hence, its design is immutable at run-time – *parameters*, such as pool position (memory address) or size, can still be configured.

The second option is to define the characteristics of the DMMs and provide a set of assemblable components to construct them when the application is loaded at run-time. A combination of the factory and strategy design patterns [GHJV95] may be used to provide a set of components (strategy implementations) that can be assembled using a recipe by a factory when each pool is created. The code for the library of strategy implementations and the factory object can be shared by all the applications in the system as a dynamically-linked library or shared object. This approach lies at the opposite side of the trade-off, offering adaptability to changing conditions at a possibly higher cost – that future research can quantify – due to the overhead of the indirect calls introduced by the strategy pattern. Additionally, the size of the code required for the components should be lower than the sum of the code of several full DMMs.

The final deployment of the application comprises its compiled code, including the mixins-based library of DMMs, or, alternatively, its compiled code, metadata with the description of the pools, the library of DMM components and the code for the factory object to assemble them. Although the application source code has to be modified to use the new memory managers – they need to receive the identifier of the DDT to which the new instances belong – these modifications are the same introduced during the profiling phase and thus, no additional effort is required.

Although $\mathcal{DynAsT}$ does not currently implement this step – the experiments in Chapter 4 use the integrated simulator – the feasibility of deployment should be sufficiently proven by the numerous times when the library-based approach has been used.

Example 2.10.1 Example of deployment with the mixin-based library[13]
Just for the sake of this example, let's assume that we have an application with five DDTs that $\mathcal{DynAsT}$ separates in two pools, as illustrated in Figure 2.7. The first one contains the DDTs with identifiers 1 *and* 5*, is assigned to a scratchpad and uses coalescing to make good use of the space. The second pool contains the DDTs with identifiers* 2*,* 3 *and* 4*, with instances of size* 24 B*,* 32 B *and several more, is placed on the main DRAM and does not use coalescing to reduce the number of non-sequential accesses.*

The following simplified sketch of code shows how the first pool could be defined using the mixins library:

```
typedef
  CoalesceHeap<
    HeapList<
```

[13]The library for the modular design of DMMs based on mixins was developed by David Atienza, Stylianos Mamagkakis and Christophe Poucet during their PhD stays at IMEC, based on previous work by Emery Berger. I have used that library in several works [Peó04, BPP+10] and even contributed a tiny amount of work to it, such as the `FixedAddressHeap` needed to fix heaps to memory ranges in embedded platforms.

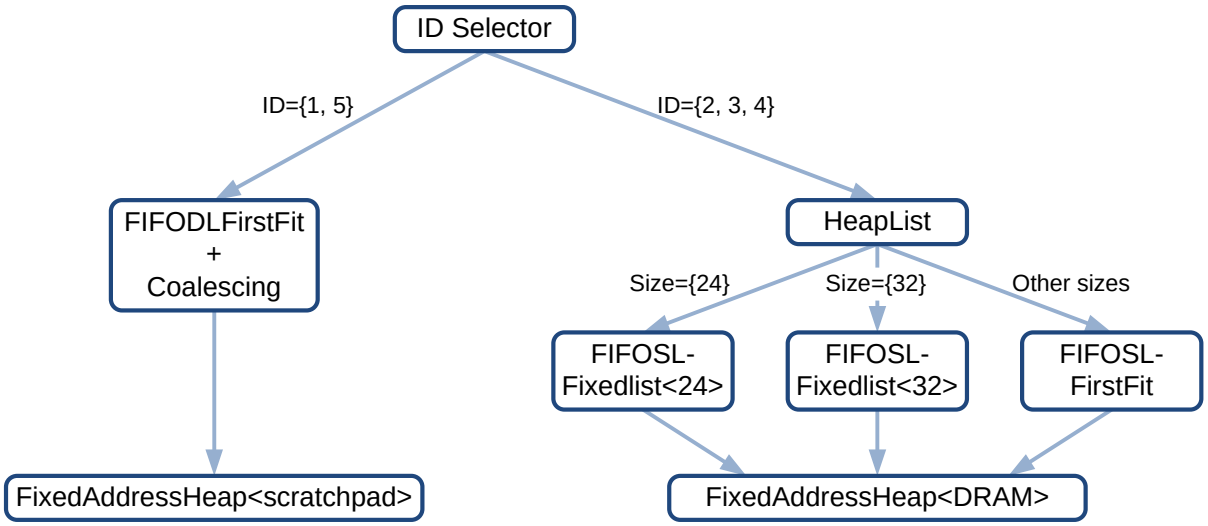


Figure 2.7.: Structure of the DMM in Example 2.10.1. DDTs with IDs 1 and 5 are placed in the scratchpad with a coalescing heap. DDTs with IDs 2, 3 and 4 are placed in the DRAM, with blocks of sizes 24 B and 32 B each apart from the rest.

```
    FIFODLFirstFitHeap,
    FixedAddressHeap<SCRATCH_ADDR, SCRATCH_SIZE>
   >,
   MIN_BLOCK_SIZE, MAX_BLOCK_SIZE
  >
  PoolA;
```

PoolA employs a coalesceable heap that uses space from either a list of free blocks or from unused space in the scratchpad. The list of free blocks uses the first-fit mechanism to locate a block for a new request. If the list does not contain a big enough block, new space is allocated from the unused part in the scratchpad. The blocks in the list are doubly-linked (next and previous) to ease the process of extracting a block from the list during coalescing. Splitting happens at allocation time, as long as the remaining block is bigger than `MIN_BLOCK_SIZE` *bytes. Similarly, coalescing happens at deallocation time, as long as the resulting block is smaller than* `MAX_BLOCK_SIZE` *bytes.* `HeapList<Heap1, Heap2>` *is a simple mechanism that asks Heap1 to handle the request; if Heap1 refuses it, then it tries with Heap2.*

The second pool can be defined with the following simplified code:

```
typedef
  HeapList<
    SelectorHeap<FIFOSLFixedlistHeap, SizeSelector<24> >,
    HeapList<
      SelectorHeap<FIFOSLFixedlistHeap, SizeSelector<32> >,
      HeapList<
        FIFOSLFirstFitHeap, % SL because no coalescing
        FixedAddressHeap<DRAM_ADDR, DRAM_SIZE>
      >
    >
  >
  PoolB;
```

Here, the pool uses first two lists of free blocks for the most common allocated sizes. The lists are singly-linked to reduce memory overhead because they are not involved in coalescing operations. As each of them contains blocks of exactly one size, free blocks do not contain any header to record their size.

Therefore, the overhead is reduced to one word, to store the pointer to the next block when they are in the list, or to store their size (so that it can be determined when they are released) when they are in use by the application. The pool has a last heap organized as a list of blocks that uses first-fit to find a suitable block for the remaining object sizes. Finally, extra space is obtained from the assigned space in the DRAM when needed.

The global DMM of the application can be composed as follows:

```
typedef
 HeapList<
  SelectorHeap<
   PoolA,
   OrSelector<IDSelector<1>, IDSelector<5> >
  >,
  SelectorHeap<
   PoolB,
   OrSelector<
    IDSelector<2>,
    OrSelector<IDSelector<3>, IDSelector<4> >
   >
  >
 >
 GlobalDMM;
```

In essence, the global DMM uses the ID corresponding to the DDT of the object to find the pool that must attend the petition. `IDSelector<ID>` *returns* `true` *if the DDT of the allocated object coincides with its parameter.* `OrSelector<A, B>` *returns* `A || B`*. Finally,* `SelectorHeap<pool, condition>` *attends a memory request if its condition test returns* `true`*; otherwise, it rejects the request.*

The parameters used to configure the different heaps are passed as template arguments. Therefore, the compiler knows them and can apply optimizations such as aggressive inlining to avoid most function calls. For example, a long chain of `OrSelector` *objects can be reduced to a simple* `OR` *expression.*

Chapter 3

Design of a simulator of heterogeneous memory organizations

y methodology for placement of dynamic data objects includes a memory-hierarchy simulation step for evaluation of the generated solutions and platform exploration. For example, if the design is at an early design phase and the actual platform is not yet available, the designer can use it to get an estimation of the placement performance. In systems such as FPGAs or ASICs in which the hardware can be modified, the designer can use the simulation to explore different architectural options. Alternatively, the simulator can also be used to estimate the performance of the applications on different models of a platform, either to choose the most appropriate one from a vendor, or to steer the design of several models at different cost-performance points. After simulation, the designer can iterate on the previous steps of the methodology or, if the outcome looks satisfactory, use the output of the mapping step to prepare the deployment of the application.

3.1. Overview

The simulator takes as input the pool mappings, which include the correspondence between application IDs and memory resources, the trace of allocation operations and memory accesses obtained during the profiling step and the template of the memory hierarchy with annotated costs that was used during the mapping step. It then calculates the energy and number of cycles required for each access.

The simulator implemented in $\mathcal{DynAsT}$ is not based on statistical analyses; instead, it uses a complete simulation model of the memory subsystem to reproduce the behavior of the profiled application and compute the energy and number of cycles required for each access. For example, DRAMs are modeled using the current, voltage, capacitive load and timing parameters provided by a well known manufacturer; the simulator tracks the state of all the banks (which row is open in each of them) and includes the cost of driving the memory pins and the background energy consumed. These calculations are performed according to the rules stated by the JEDEC association – except for mobile SDRAM modules because they appeared before standardization and were specified independently by each manufacturer.

3.1.1. Elements in the memory hierarchy

As explained in Chapter 2 for the mapping step, the platform template can include any number of the following elements:

- **Multi-level bus structures.** The memory subsystem can be connected to the processor via a common bus, or through a multi-level hierarchy of buses with bridges between them. For each bus or bridge, it is possible to specify the energy and cycles that are required to pass through it.
- **Static RAMs.** The simulator supports SRAMs of any size, parameterized with their energy cost and latency per access. The main distinctive characteristic of SRAMs is that they are truly random-access memories: Accessing a word costs the same disregarding previous accesses to that module. As an example, scratchpad memories (SPMs) are usually implemented as SRAMs.
- **Dynamic RAMs.** Main memory is usually implemented as DRAM due to its higher density and lower cost per bit. The simulator integrated in $\mathcal{DynAsT}$ supports currently two types of DRAMs, Mobile SDRAM and LPDDR2-SDRAM, and performs calculations according to the rules for state transitions defined by the JEDEC association [JED11b] and the manufacturer's datasheets [MIC10, MIC12]. The DRAMs can be organized in any number of banks; the simulator calculates the correspondence of addresses, rows and columns automatically.
  - **Mobile SDRAM.** Single data rate, low power mobile version also known as LPSDR-SDRAM. Multiple consecutive read or write accesses can be issued to random addresses in different banks and the memory outputs one word of data per cycle as long as all the involved banks are active and no row misses happen. This technology allows burst sizes of 1, 2, 4, 8 words or full-row.
  - **LPDDR2-SDRAM.** Double data rate, low power version. This technology transfers two data words per cycle, using both the rising and falling edges of the clock. Consecutive read or write bursts may be executed to any number of banks without extra latency as long as all the banks are active with the appropriate rows selected.
- **Cache memories.** Every DRAM module can have an associated cache hierarchy covering its complete address range, or just a part. Cache memories may be modeled according to several parameters: Size, associativity (from direct-mapped up to 16-ways), line length (4, 8 or 16 words), replacement policy (random or LRU, that is, Least-Recently-Used), cached address range, latency and energy consumption per access. The simulator supports hierarchies of up to ten levels.

SRAMs and caches are assumed to work at the CPU frequency; hence, a latency of 1 cycle counts as one CPU cycle in the simulator. In comparison, DRAMs work usually at a lower frequency and their latencies are multiplied by the factor between both frequencies.

3.1.2. DMM and memory addresses translation during simulation

The simulator reads the traces obtained during profiling. It uses memory allocation events, which contain the IDs of the DDTs to which the affected instances belong, to create and destroy representations of the objects mimicking the original execution of the application. This

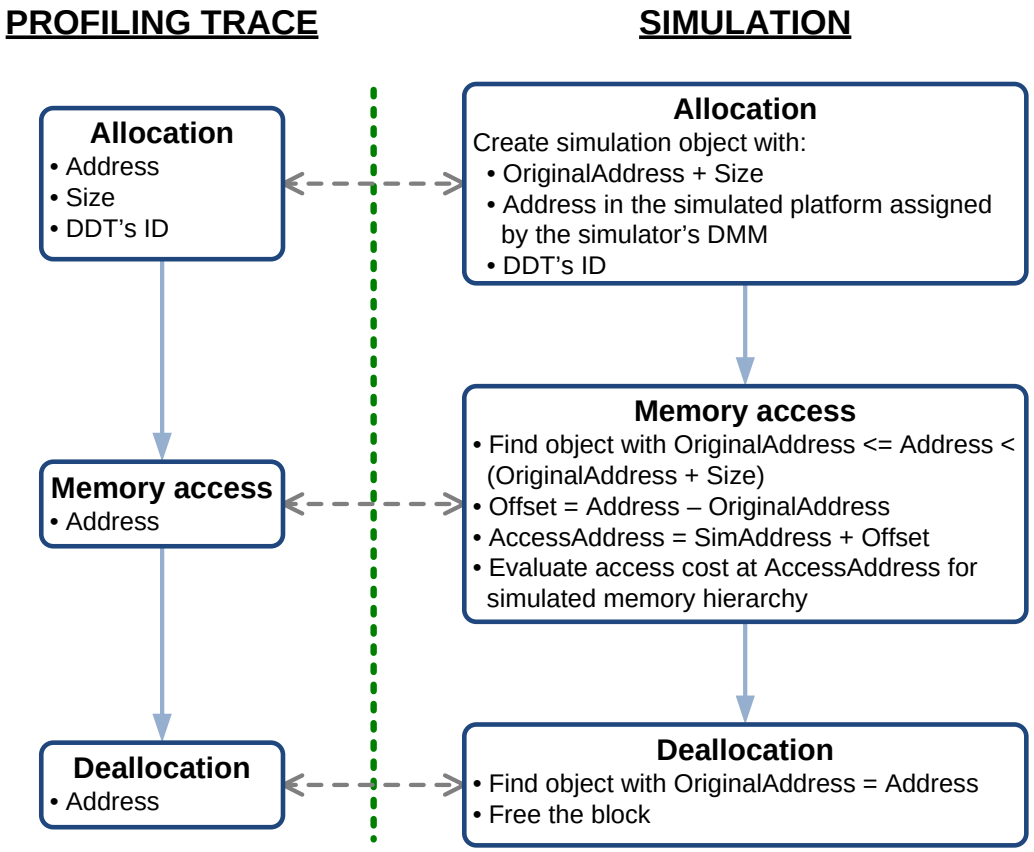


Figure 3.1.: Translation of profiling addresses during simulation. The simulator creates an object in the platform address space for every object created in the original execution. Then, every recorded memory access is mirrored over the appropriate address in the simulated platform.

step is fundamental to reflect the behavior of the application because the memory accesses recorded in the traces were not executed over absolute memory addresses, but on concrete data instances. As the memory organization used during profiling is different than the one used during mapping and simulation – after all, that was the very purpose of *DynAsT*! – the addresses assigned in the original execution of the application are meaningless in the context of simulation. Therefore, the simulator uses its own implementation of a dynamic memory manager to assign addresses to the instances of each DDT in the memory space of the simulated platform.[1] Figure 3.1 illustrates the address translation mechanism.

The memory allocator used by the *DynAsT*'s simulator is an always splitting and coalescing allocator. However, it does not affect the execution of the application as the DMM is ran entirely by the simulator, out of the simulated platform's memory subsystem.[2] It uses two

[1] Although it should be possible to use directly the description of DMM produced during previous phases, the current implementation of the simulator in *DynAsT* uses a simplified approach.

[2] In this way, memory accesses executed by the concrete DMMs created during execution are not accounted. Future work could easily change this by implementing in the simulator the exact algorithms designed by the previous phases, working on the pool space of the simulation. However, it could also be used to evaluate the benefits of using a separate SRAM memory to hold the internal data structures used by the allocators. For example, it is not uncommon to avoid coalescing and splitting or full block searches in pools that are mapped in a DRAM to reduce the energy consumption of random accesses (which may force additional row activations). Holding the data structures used by the DMM in a different small memory may widen the possibilities reducing the cost of more complex organizations such as ordered trees instead of linked lists.

trees (`std::multimap`, which is really an associative container with element ordering) to keep track of blocks: One, ordered by block size, for free blocks; the other, ordered by addresses, for used blocks. In that way, finding the best fit for a memory request is just a matter of traversing the tree of free blocks (with the method `std::multimap::lower_bound()`). Similarly, finding the block that corresponds to a deallocation operation consists on traversing the tree of used blocks, but using the request address instead of its size as the search key.

When a pool is divided between several memory modules, the allocator tries first the heap in the most efficient memory. If it does not have enough space, the next heap is then probed. Finally, if none of the heaps of the pool have enough space, the memory request is forwarded to the backup pool. Objects are not reallocated, even if enough space becomes available in one of the other heaps of the pool, for the same issues with data migration as outlined previously.

The simulator keeps also a tree (`std::map`) with the active data instances, ordered by their address in the profiling trace. Each entry contains the ID of its DDT, its starting address in the original run, starting address in the simulated platform and size. When the simulator encounters a memory access in the profiling trace, it looks for the instance that spawned over that address in the original execution and calculates the access offset from the object starting address. Then, it uses the calculated offset and the starting address of the instance in the simulated platform to determine the exact address and memory module that corresponds to that access in the simulation. In this way, the simulator correlates accesses recorded during profiling with accesses that have to be accounted during simulation. The IDs of the objects are used to count accesses to the different DDTs during simulation – the translated address identifies the memory resource and the pool of the accesses, but not the DDT of the object.

3.1.3. Global view of a memory access simulation

Figure 3.2 shows a high-level view of how memory accesses are evaluated by the simulator. For every memory access in the trace file, the simulator translates its address in the original platform to its address in the simulation platform. With this address, it can identify the affected memory module. For SRAMs and uncached DRAMs, the simulator uses directly the corresponding memory model to calculate the cost of the access. For cached ranges of DRAMs, the simulator checks if the accessed word is in the cache. If so, the cost of the access in the cache memory is calculated. On the contrary, if the word is not contained in the cache, the simulator has to evaluate the behavior of a complete memory hierarchy that brings it there, taking into account issues such as writing back modified lines that need to be evicted and data copies from as far as main memory to the first cache level.

The memory simulator included in $DynAsT$ makes a few assumptions that are interesting to know. First, consecutive accesses to a memory word (or to bytes thereof) that may be registered by the profiling mechanisms are ignored because they are assumed to be handled inside the load/store queues of the processor. The processor is assumed not to access individual bytes through the system bus. Second, multi-cycle read accesses stall the processor; however, DRAM writes do not necessarily. DRAMs require normally that data words are presented on the bus for one *bus* cycle; only the last access in a write burst bears a complete latency, and only when the memory is immediately used for a different operation before the time required by the memory to complete that last write. However, SRAMs and caches are neither pipelined in the current implementation.[3] Finally, the simulation does not reflect delays between memory

[3] Any future modification in this aspect should be introduced simultaneously for SRAMs and caches to produce fair comparisons because pipelining write accesses may change the relative performance of both types of

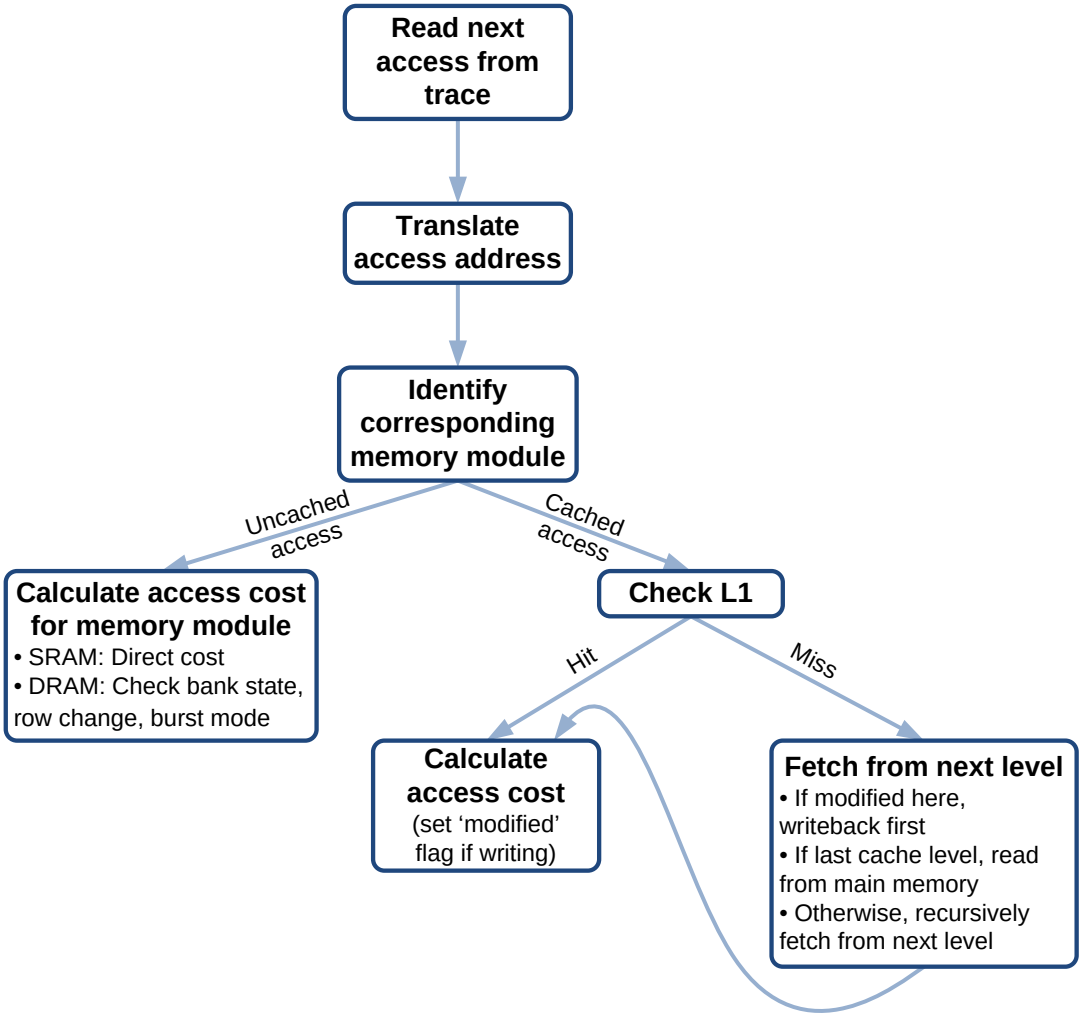


Figure 3.2.: Simulation diagram.

accesses due to long processing times (e.g., long chains of floating-point operations) as they are not recorded in the traces. However, most data-dominated applications are usually limited by memory bandwidth and latency rather than by processing time; therefore, this factor should not have a significant impact on the simulation outcome.

3.2. Simulation of SRAMs

Static memories are the easiest to simulate. As they are truly random-access memories, the cost of reading or writing a word does not depend on the previous operations. In that sense, those memory modules do not have a notion of state.

Every SRAM in the platform template covers a range of addresses defined as a base address and a size. No other memory module's address range may overlap with the address range covered by a particular SRAM module. Additionally, SRAMs cannot be cached in the current implementation.

memories; energy consumption should not be affected, though.

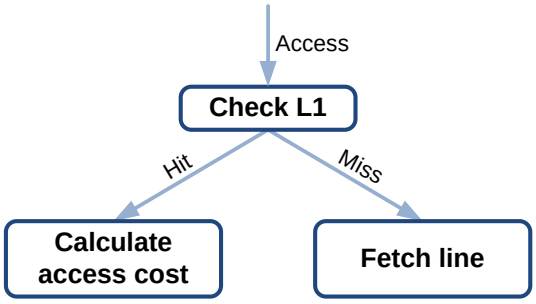


Figure 3.3.: Cached memory-access simulation. In the case of a cache miss, the simulator traverses the cache hierarchy as a real system would do, annotating energy consumption and latencies along the way.

The definition of an SRAM in a platform template has four parameters: `EnergyRead` and `EnergyWrite`, which define the energy consumed during a read or a write access, respectively; and `DelayRead` and `DelayWrite`, which define the number of cycles required to complete a read or write access, respectively. The simulator simply accumulates the cost of each access to each SRAM. More complex cases, such as pipelined write accesses for devices with latencies higher than 1 cycle, can be easily included in the future.

3.3. Simulation of cache memories

The methodology and *DynAsT* are agnostic about caches: They are not considered when placing dynamic data objects into DRAMs, but the designer is free to include them either during simulation or in the actual hardware platform, as long as they do not interfere with other address ranges (specifically, with those covered by any SRAMs). However, the simulator supports the inclusion of cache memories in the platform templates to evaluate their effect on the remaining DRAM accesses or simply to compare *DynAsT*'s solutions with traditional cache-based ones (as I do in Chapter 4).

The simulator supports direct-mapped and associative caches, both with a configurable line size. Cache hierarchies can be "weakly" or strongly inclusive. In strongly inclusive cache hierarchies, every position contained in a level closer to the processor is guaranteed to be contained in the next level. For example, a word in L1 is also contained in L2. Exclusive cache hierarchies (not currently supported) guarantee that words are never contained in more than one level in the hierarchy. As an intermediate point, "weakly inclusive" (for lack of a standard term) hierarchies do not guarantee that a word in a level closer to the processor (e.g., L1) is also contained in the next level (e.g., L2).

Enforcing complete inclusiveness eases maintaining cache coherence between multiple processors, but introduces extra data movements because a line fetch in a further level that produces an eviction must also evict the corresponding line(s) in the closer levels, with the potential associated write backs.

When the simulator processes a memory access from the trace file that corresponds to an object placed on a DRAM in the simulated platform, it first checks if its offset corresponds to a cached area. The simulator uses either the corresponding DRAM or cache model. If a cached access results in a "cache hit," the associated energy and latency are accounted. If it results in a "cache miss," then a recursive process is triggered to bring the cache line that contains that

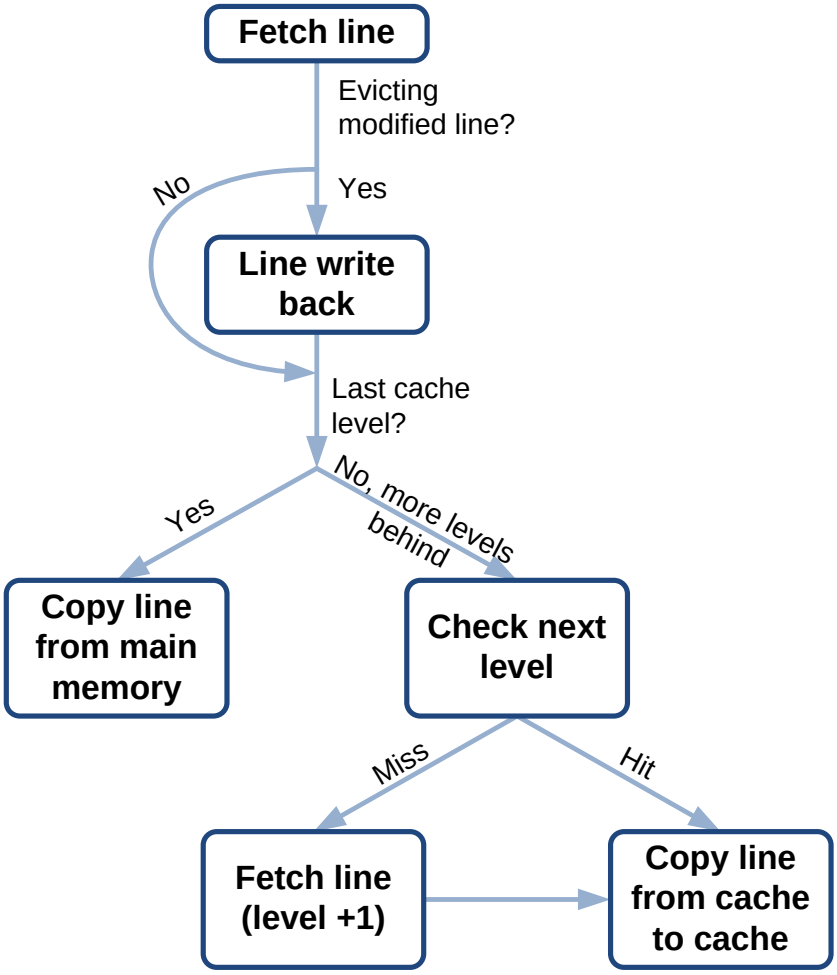


Figure 3.4.: Simulation of cache line fetch.

word from as far in the memory subsystem as necessary, accounting for potential evictions at each cache level. This process is shown in Figure 3.3.

Figure 3.4 illustrates the simulation of line fetches. First, the simulator checks if the cache has to evict a modified line (unmodified lines can be discarded directly). If so, it starts a line write-back procedure. When the line can be safely reused, it checks if the cache is the last level, in which case it simply accounts for copying the data from the main memory module. However, if the current cache level (l) is backed by other ones, the simulator has to recursively repeat the process for them: If the access is a hit for the next level ($l+1$), the line is simply copied; but if it is a miss, then the address is recursively fetched starting at level $l+1$.

Finally, Figure 3.5 shows the process of line write back. Again, if the current cache level (l) is the last (or only) one, the line is written straight to main memory. The simulator invalidates automatically lines written back. If the current cache level is backed by another one, the simulator checks whether the cache at $l+1$ contains a copy. If the line is present at level $l+1$, the updated contents are simply passed from level l to level $l+1$. Otherwise, the line is copied to $l+1$, with the exception that if it has longer lines, then the complete line must be first fetched there to ensure a correct partial update. Finally, if level $l+1$ does not have a free line to accommodate the line from level l – which may happen for weak inclusion – then a write back procedure is recursively started to free one cache entry at level $l+1$.

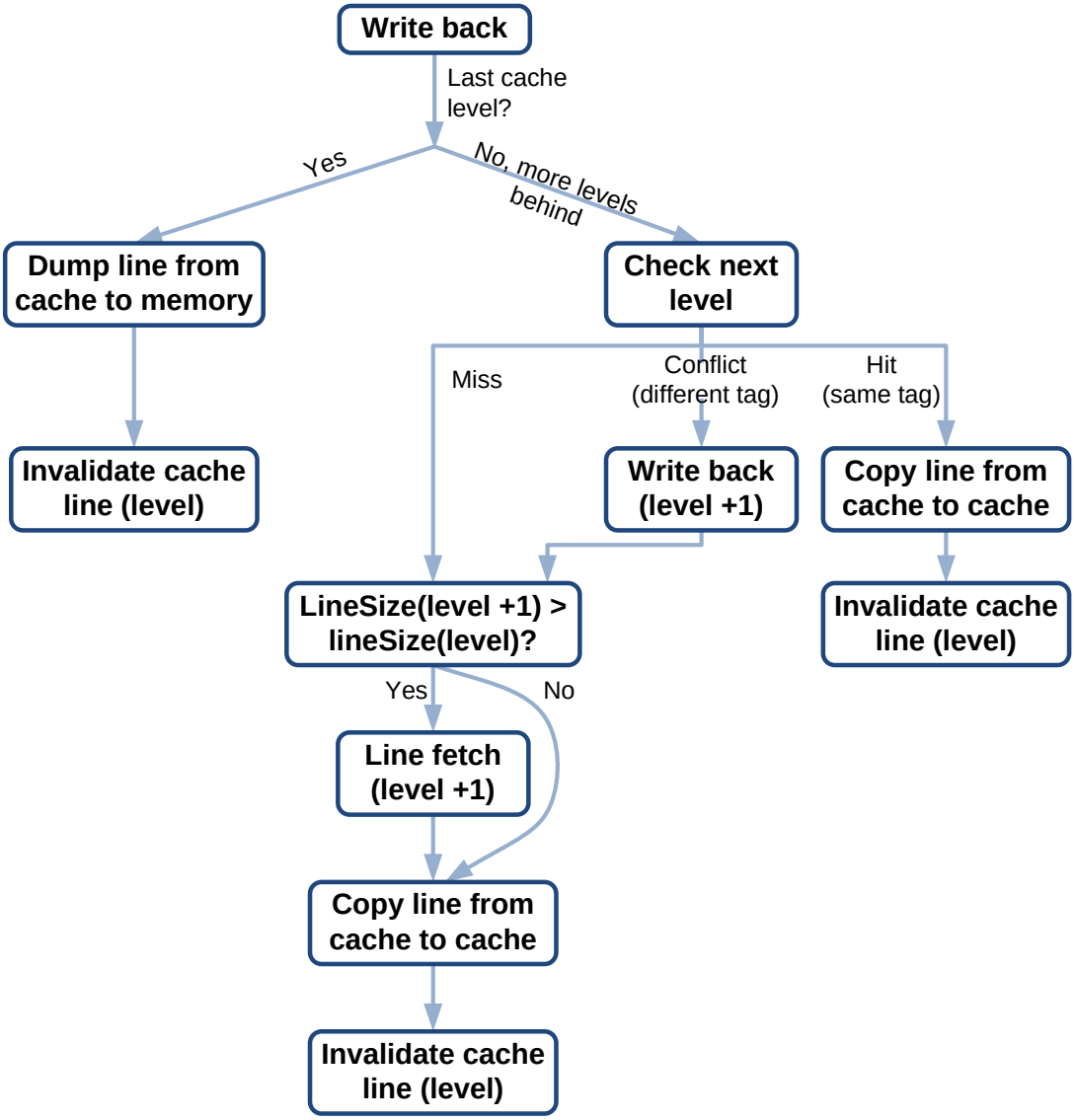


Figure 3.5.: Simulation of cache line write-back.

Figures 3.4 and 3.5 are modified to force complete inclusion as follows. When a cache line is going to be overwritten at cache level l (either to receive data from main memory or because of the invalidation after a write back), the simulator checks if the same address is also present at level $l-1$. If so, the line is also purged (written back if needed) from $l-1$. This extra work guarantees that an address is not present at a level if it is not also present at the next ones.

Checking whether an address is present at a cache level counts as an access. The simulator assumes that caches read the tag and data in parallel, which is the standard behavior to improve access time for hits. A possible extension could be considering that the cache may read just the tag data and then, in case of a hit, the actual data contents, which is sometimes implemented in big caches with high associativity degrees to reduce cache energy consumption.

The previous operations work all at the logical level in the simulator: No real data values are moved across the simulated hierarchy levels. The simulator only accounts for energy con-

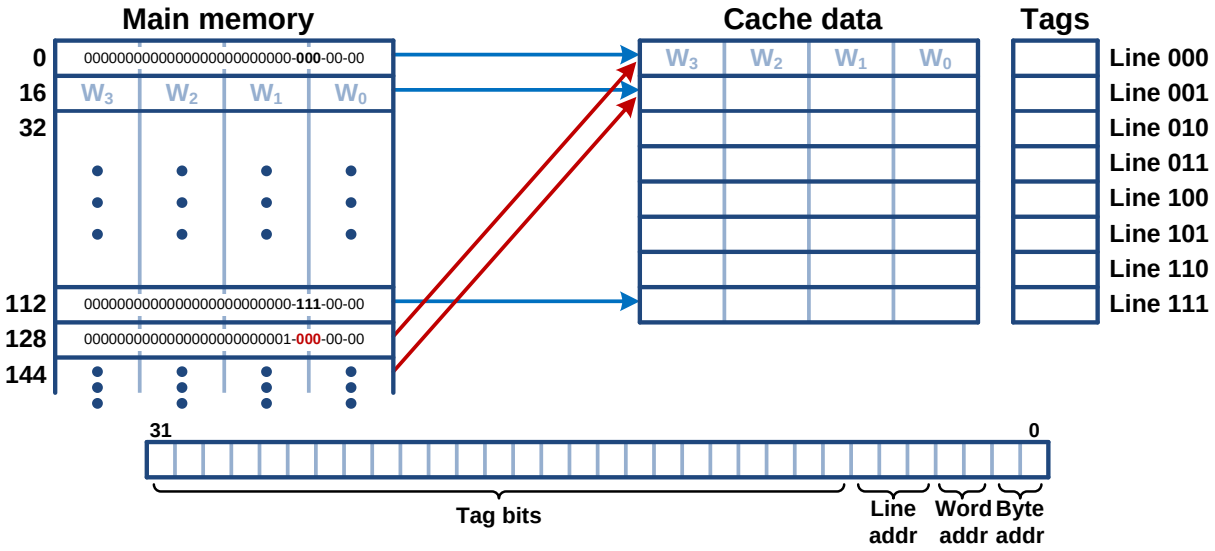


Figure 3.6.: Direct-mapped cache with 8 lines of 4 words each (total size = 128 B). Every 128 B, memory addresses collide on the same cache line. The cache controller checks the tag bits stored with each line against the access' address to discriminate between hits and misses.

sumption and latencies, and sets the state of each cache line (address tag, and validity and modification bits).

3.3.1. Overlapped accesses

Transfer operations between cache levels or main memory are overlapped. The simulator accumulates the energy required to perform each access to each memory, but latencies are determined by the slowest one. Therefore, the simulator assumes that words are transferred one at a time at the pace of the slowest memory. Future work may easily include the option of wider inter-cache buses.

Energy consumption when transferring complete lines is calculated as the addition of the energy required to transfer individual words. The simulator does not currently support "hot-word first" operation – as the processor is not simulated, it is difficult to assess the benefits of this technique in the execution pipeline – nor multiport caches.

3.3.2. Direct mapped caches

Direct-mapped caches are the simplest type of cache memories. The cacheable address space is divided in blocks of the same size than the cache line (e.g., 64 B for lines of 16 32-bit words). A mapping function is used to assign each line in the address space to a line in the cache memory. The simplest case divides the address bits into fields for byte, word-in-line and line. The bits for the line are used to address directly into the cache memory.

This simple mapping of the complete address range into the address range of the cache memory produces conflicts for addresses separated exactly the size of the cache memory. No matter what mapping scheme is used, conflicts are inevitable (by the pigeonhole principle). To avoid mistakes, the cache memory stores the upper part of the address ("address tag") along the data values. Thus, the cache knows implicitly part of the address of each line contents and stores explicitly the rest of the address bits. The combination of line addressing and tag

comparison allows the cache controller to determine if an address is contained in a cache. Figure 3.6 shows this simple scheme.

The simulator receives as parameters the total cache capacity and the number of words per line, and calculates automatically the number of bits used for word addressing, the number of lines in the cache and the number of tag bits:

$$byteAddrSize = \log_2 \overbrace{bytesPerWord}^{4} \tag{3.1}$$

$$wordAddrSize = \log_2 wordsPerLine \tag{3.2}$$

$$numLines = (cacheSize / bytesPerWord) / wordsPerLine \tag{3.3}$$

$$lineAddrSize = \log_2 numLines \tag{3.4}$$

$$tagSize = \underbrace{addrSize}_{32} - wordAddrSize - lineAddrSize - \underbrace{byteAddrSize}_{2} \tag{3.5}$$

3.3.3. Associative caches

Associative caches try to palliate the problem of address conflicts in direct mapped caches. In essence, the space is divided in n sets that correspond conceptually to n direct-mapped caches of reduced capacity. Memory addresses are mapped into cache lines as with direct-mapped caches, using an equivalent mapping function (frequently, just a subset of the address bits). However, a given memory address may reside in any of the n sets.

The benefit of this approach is that circumstantial address-mapping conflicts do not force mutual evictions because the lines can be stored at the same time in the cache, each on one of the sets. The disadvantage is that all the sets must be probed for a hit and a mechanism for set selection when storing a new line is needed. Frequent choices are LRU, which evicts the line in the Least-Recently-Used set with the hope that it will not be needed soon, and random replacement.

Associativity degrees of 2 and 4 tend to produce the biggest improvements in comparison with direct-mapped caches, trading between hit rate and cost per access (because of the multiple tag comparisons), while higher degrees tend to provide diminishing returns. Figure 3.7 illustrates the working of an associative cache with 2 sets.

The simulator receives as parameters the total cache capacity, number of words per line and number of sets (ways), and calculates automatically the number of bits used for word addressing, the number of lines in the cache and the number of tag bits:

$$byteAddrSize = \log_2 \overbrace{bytesPerWord}^{4} \tag{3.6}$$

$$wordAddrSize = \log_2 wordsPerLine \tag{3.7}$$

$$numLines = ((cacheSize / bytesPerWord) / wordsPerLine) / numSets \tag{3.8}$$

$$lineAddrSize = \log_2 numLines \tag{3.9}$$

$$tagSize = \underbrace{addrSize}_{32} - wordAddrSize - lineAddrSize - \underbrace{byteAddrSize}_{2} \tag{3.10}$$

Sets are not identified by any addressing bits. Instead, the full tag is stored for each set and compared when a match for a line is searched. Of course, the number of lines is reduced by the number of sets in comparison with a direct-mapped cache.

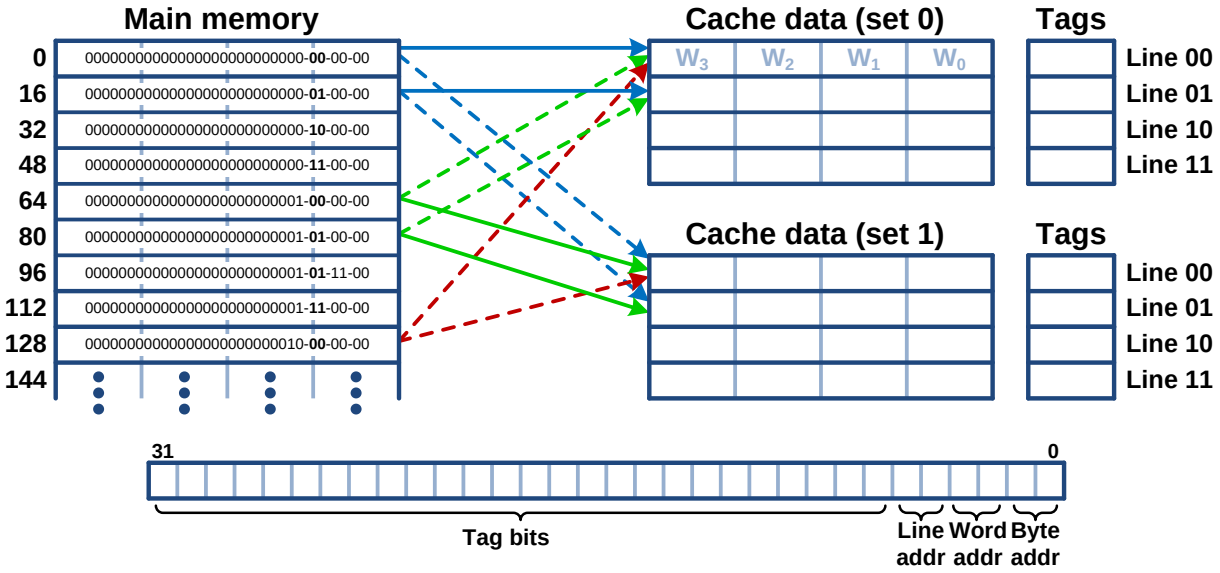


Figure 3.7.: 2-Way associative cache with 4 lines of 4 words each (total size=128 B). Every 64 B, memory addresses collide on the same cache lines, but they can be stored on any of the two sets. The cache controller checks the tag bits stored in each set for the corresponding line against the access' address to discriminate between hits to any of the sets and misses.

3.4. Overview of dynamic memories (DRAMs)

Accurate simulation of DRAM modules is a complex and tricky process. Although I have been as careful as possible and I have verified the results of the simulation whenever possible (for example, against the spreadsheets provided by Micron), I cannot completely rule out the possibility of bugs in the simulation. Therefore, I offer as many details as it seems reasonable so that readers of this text may evaluate themselves the conditions on which the experiments of Chapter 4 were performed.

3.4.1. Why a DRAM simulator?

Manufacturers such as Micron provide simple spreadsheets to estimate the average power required by a DRAM during the whole execution of an application. However, calculating energy consumption requires knowing also the execution time ($E = P \cdot t$). The main factors that affect DRAM performance are the number of row changes,[4] the exact combination and number of switches between reads and writes, and the number of accesses that are executed in burst or individually (the latency of the first access is amortized over the length of a burst). $\mathcal{DynAsT}$ includes a DRAM simulator that tracks the current state of each memory bank to calculate more accurately the number of cycles that each memory access lasts and compare the performance of different solutions.

At the time of implementing $\mathcal{DynAsT}$, Micron provided a spreadsheet for SDRAM power calculations that could be also applied to their mobile (low power) devices. However, the spreadsheets for DDR2-SDRAMs were at that time not easily applicable to LPDDR2-SDRAMs,

[4]The worst case happens when every operation accesses a different row in a single bank. Assuming that t_{RC} is 10 cycles and t_{RCD}, t_{RP} and CL are 3 cycles in an SDRAM at 166 MHz, each memory access lasts 10 cycles ($t_{RC} \geq t_{RCD} + t_{RP} + CL$). Thus, only a 10 % of the maximum bandwidth is actually available. In comparison, maximum bandwidth can be attained in burst mode.

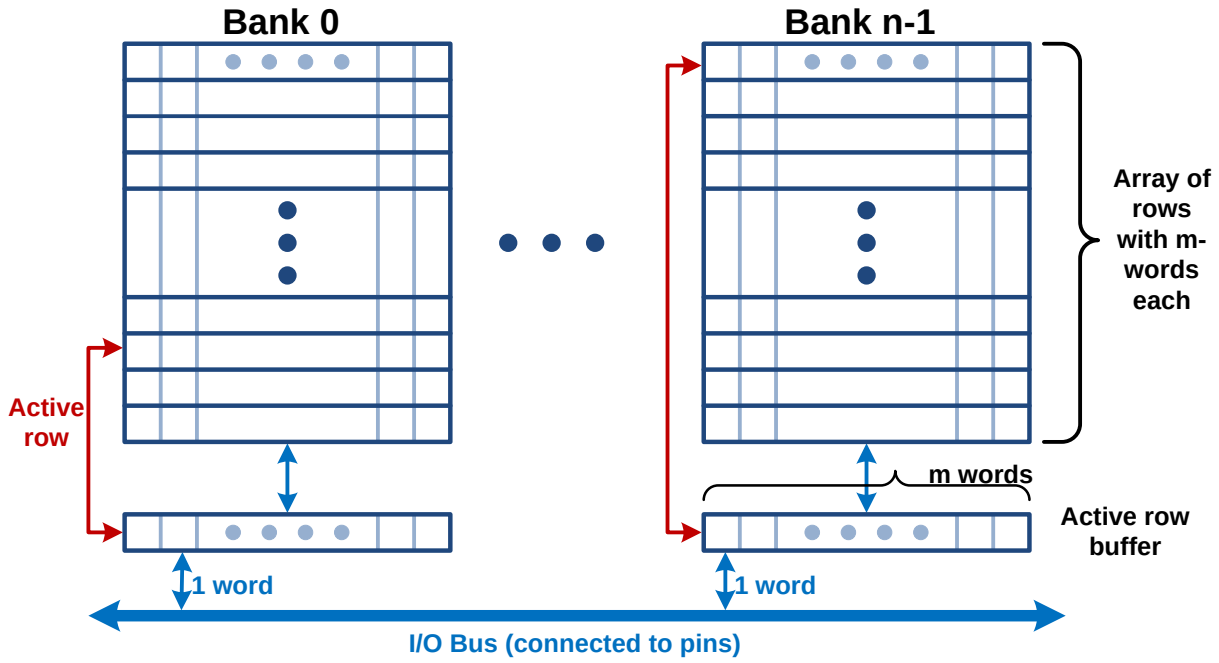


Figure 3.8.: The cells in the DRAM matrix cannot be accessed directly. Instead, DRAMs are divided in several banks and each one has a buffer that holds at most an active row. Only the words in the active row buffer can be accessed directly.

which separate currents to more easily control energy consumption. Thus, a method to calculate energy consumption in LPDDR2-SDRAM devices was required and $\mathcal{DynAsT}$'s simulator was the perfect candidate.

An additional interesting improvement of a more precise model of DRAM activities that reflects energy consumption on an operation basis is that the designer can identify peaks that affect negatively system performance. Indeed, $\mathcal{DynAsT}$ might be extended in the future to include (already existing) temperature calculations based on this cycle-accurate energy consumption and track both spatial and temporal temperature variations.

3.4.2. DRAM basics

Main memory is usually implemented as DRAM due to its higher density and lower cost per bit. However, the elements in the DRAM cell array cannot be accessed directly. Memory cells are grouped in rows of usually 1024 to 8192 words (Figure 3.8). At the memory module level, an internal buffer latches the contents of one active row. Words in the active row buffer can be accessed efficiently.

To access a different row, the module undergoes a two-step process: First, the pairs of bit lines that traverse the columns in the cell array are "precharged" at specific voltage values. Then, the transistors of the cells in the selected row are "activated" to connect the cells to one of the bit lines. Sense amplifiers detect the voltage difference between the two bit lines, amplifying it. The read values are latched in the row buffer so that the individual words can be accessed efficiently. The operation of the sense amplifiers "refreshes" the cells that have been read, but this operation requires some time that must be respected between an activation and the next precharge.

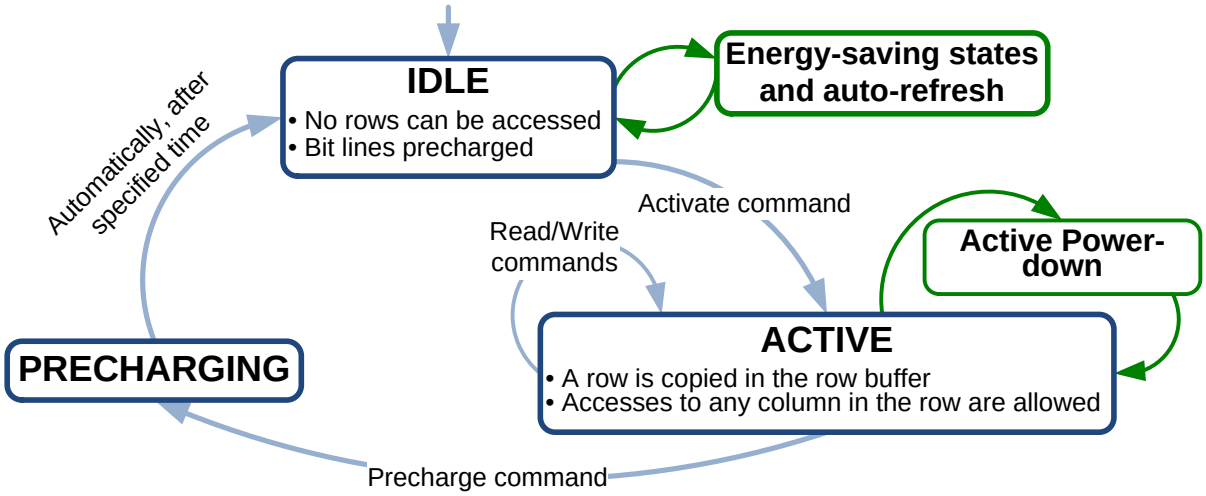


Figure 3.9.: Simplified state diagram for DRAMs.

Additionally, as DRAMs are based on capacitors that lose their charges over time, the cells in every row have to be "refreshed" periodically in order to keep their charges. As a result, the DRAM becomes inaccessible periodically (every few microseconds) as the controller refreshes one row each time (which basically means precharging the bit lines and activating that row). The time that the rows can endure before losing their data diminishes as temperature increases; hence, this is another reason to save energy and keep the device's temperature low.

To reduce the need for switching active rows, DRAM modules are divided into several (usually 4 to 16) banks. Each bank has one buffer, that is, at most one row can be active at a time in each bank. Words from the open rows in any of the banks can be accessed seamlessly, but changing the active row in a bank incurs an extra cost – however, oftentimes a careful block and row management can hide some of the latencies by reading data from a bank while other is preparing a new row. Therefore, the cost of an access depends on the currently active row for the corresponding bank. Interestingly, Marchal et al. [MGP+03] showed how a careful allocation of data to DRAM banks can reduce the number of row misses and improve latency and energy consumption.

Figure 3.9 presents a generic (and simplified) diagram of DRAM states. The default state for the banks of a DRAM is the *IDLE* state, where no row is open. When an access needs to be performed, the memory controller has first to activate the row that corresponds to the memory address. This process moves the bank to the *ACTIVE* state, copying the data corresponding to the required row from the main DRAM array to the active row buffer. Any number of accesses, in any order, can be performed to any number of columns in the row buffer.[5] When the controller needs to access a word in a different row, it must issue a `PRECHARGE` command to the bank. The bank remains in the *PRECHARGING* state for a specified period of time and, after that, the controller can issue the activation command needed to access the new row and move the bank to the *ACTIVE* state again. The JEDEC association publishes a standard regulating the logical behavior of each DRAM technology and the minimum physical parameters (timing, voltages, currents, etc.) that the manufacturers must respect to claim compliance with it.

[5]Read accesses are performed directly on the active row buffer. However, writes need to propagate to the cells in the DRAM matrix. Hence, there is a "write-recovery" time that must be respected before issuing any other command after a write.

In general, accesses to words in the active rows of any of the banks can be alternated efficiently in almost every type of DRAM. In particular, the simulated types support continuous bursts of reads or writes. Burst modes are active for a number of words; individual or burst accesses can be linked if the respective commands are presented on the command lines of the bus at the appropriate times. Accessing a word in a row that is not active requires precharging the bit lines and activating that row. A flag on every access command allows the memory controller to specify if the row should be closed (i.e., a `PRECHARGE` automatically executed) after it.

Currently, the simulator works with an "open-row" policy, that is, rows are kept active until an access forces a change. However, other policies can be explored. For example, in the "closed-row" policy the controller closes the active row after every burst; however, this technique can incur significant increases on energy consumption and instantaneous power requirements (currents in the circuit are much higher during activations). Other intermediate possibilities close the active rows if they are not used after a certain time, which allows the bank to enter a lower-power mode. An interesting option would be modifying the simulator to assume that the memory controller has enough information to guess if a row should be closed during the last access to the previous row. Thus, given enough time between accesses to different rows, the delays could be practically hidden (of course, energy would be consumed equally). A similar technique seems to be applied by current Intel processors [Dod06].

DRAM timings (e.g., CL, t_{RCD} or t_{Read}) are expressed in terms of bus cycles. As the simulator works with CPU cycles, all timing parameters are multiplied by *CPUToDRAMFreqFactor* during simulator initialization.

3.5. Simulation of Mobile SDRAMs

At the time of implementing $\mathcal{DynAsT}$'s simulator, a standard for low power SDRAMs was not available. Instead, each manufacturer implemented their own variation. For this work, I chose to follow Micron's datasheets for their Mobile LPSDR-SDRAM [MIC10], whose most fundamental characteristic is being a pipelined architecture instead of a prefetch one. The datasheet makes the following remarks in this regard:

> "Mobile LPSDR devices use a pipelined architecture and therefore do not require the $2n$ rule associated with a prefetch architecture. A READ command can be initiated on any clock cycle following a READ command. Full-speed random read accesses can be performed to the same bank, or each subsequent READ can be performed to a different bank." [MIC10, p. 43]

And

> "Each READ command can be issued to any bank." [MIC10, p. 44 Note 1; p. 45 Note 1]

Similar remarks are made for write accesses in subsequent pages of the datasheet.

With the previous information, the simulator assumes that all consecutive (in time) read accesses form a burst; in particular, bursts can include reads to any words in the set of active rows, no matter their absolute addresses (Figure 3.10). The first access in a burst reflects the complete delay of the pipeline (CL), but the memory outputs one word in each subsequent bus cycle (t_{Read} is normally one cycle). Energy consumption is calculated for the whole duration of the access (CL for the first, 1 for the next) because I_{DD4} is measured during bursts and thus,

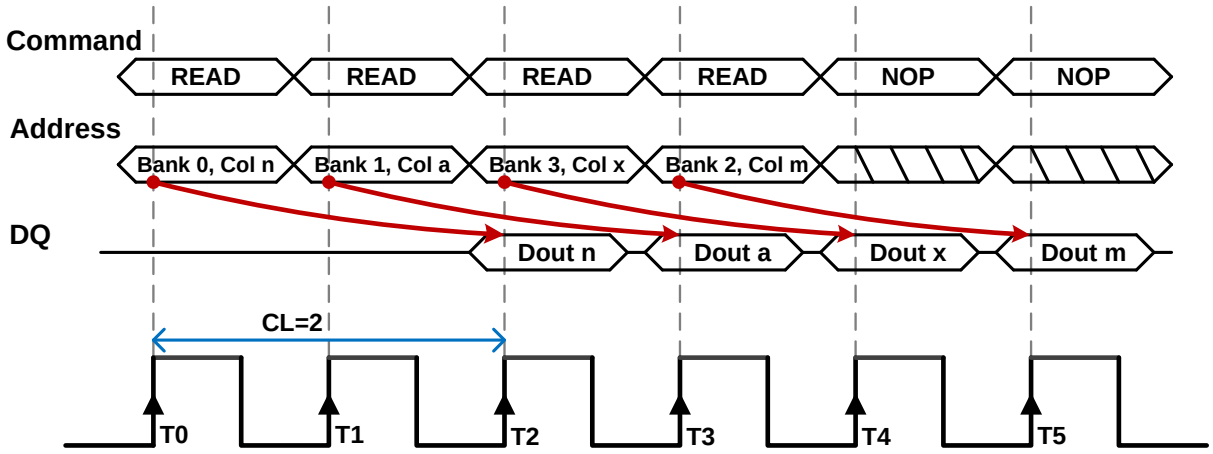


Figure 3.10.: "Seamless burst" access in an SDRAM with CL=2, random read accesses. Each READ command can be issued to any bank as long as they are active. Successive READ commands can be issued on every cycle. The result of each operation is available *CL* cycles after the command is issued. This example shows linked bursts of length 1; longer burst sizes retrieve several consecutive data words with a single READ command. In that case, continuous reading can be achieved issuing each READ command *CL* cycles before the preceding burst finishes.

it reflects the energy consumed by all the stages of the pipeline. A possible error source is that I_{DD4} might be slightly lower during the *CL* cycles of the first access as the operation is progressing through the pipeline, but there seems to be no available data in this respect.

A burst is broken by an inactivity period or an access to a different row. In those cases, the first access of the next burst will bear the complete latency. Inactivity periods (i.e., after the last word of a burst is outputted) happen if the processor accesses other memories for a while; for instance, because most of the accesses happen to an internal SRAM or cache memories are effective and thus DRAMs are seldom accessed. Accesses to words in a different row on any bank also break a burst (because the simulator inserts then the precharge and activation times). The simulator can be extended assuming that the PRECHARGE command was sent to the affected bank as far back in time as the last access to that bank (every access command may include an optional PRECHARGE operation) and, therefore, simulate interleaving of bank accesses with no intermediate latencies.

Writes are only slightly different. Once starting delays are met (e.g., after a row activation), the pipeline does not introduce additional delays for the first access; each data word is presented on the bus during one cycle (t_{Write} is 1 in the used datasheets). However, the banks require a "write-recovery time" (t_{WR}) after the last write before a PRECHARGE command can be accepted. Every time that the simulator has to change the active row in a bank, it checks if that bank is ready to accept new commands by evaluating $t_{LastBankWrite} + t_{WR} \geq CurrentTime$. Energy consumption is calculated for one cycle (t_{Write}) for normal write accesses; however, when a burst is finished (due to inactivity) or the activity switches to a different bank, the extra energy consumed by the bank finishing its work in the background during t_{WR} cycles is also accounted.

The simulator does not implement any policy for proactively closing rows and thus does not currently account for the potential energy savings. This topic is worth future expansion.

Address organization in a DRAM module is a matter of memory controller design. A common choice is to organize address lines as "row-bank-column" so that logical addresses jump

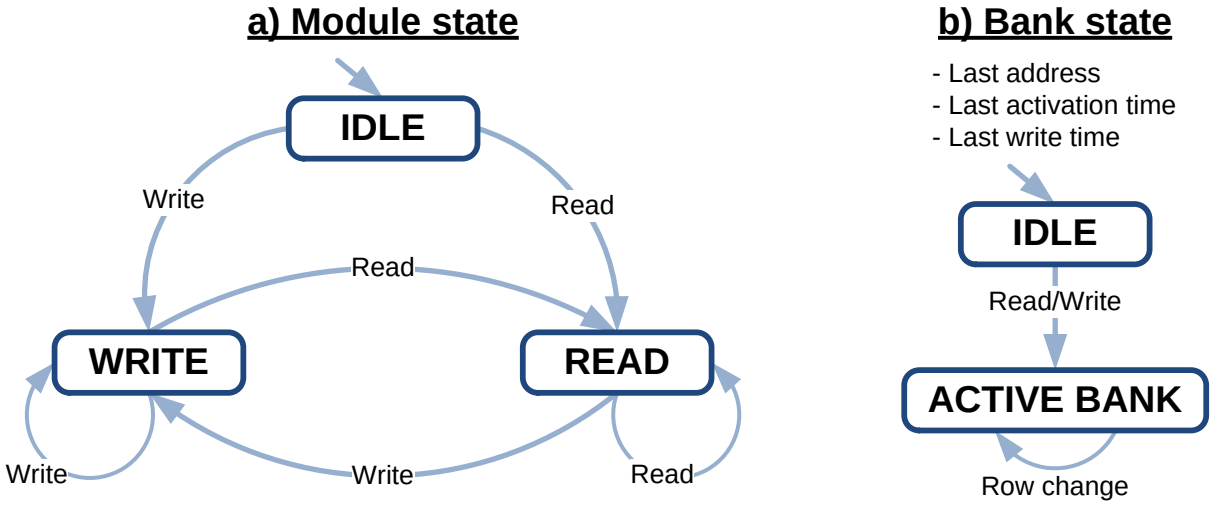


Figure 3.11.: Module and bank state in the simulator. All transitions check if a change of active row is needed. Transitions from write that require a row change have to observe t_{WR}.

from row j in bank m to row j in bank $m+1$. This option is beneficial for long sequential accesses because the memory controller can precharge and activate the row in the next bank on advance, so that when the words in the active row of the current bank are read, it can continue immediately with the next bank. However, $\mathcal{DynAsT}$ organizes address lines as "bank-row-column:" Addresses cover first a complete bank, then continue in the next one. The downside of this choice is that long sequential accesses that spawn several rows must wait for the precharge and activation times after exhausting each row. However, it enables pool placement on DRAM banks because the banks can be seen as continuous ranges of memory addresses.

Example 3.5.1 Addressing in a 256 MB 4-bank DRAM.
With $\mathcal{DynAsT}$'s view of memory space (bank-row-column), each DRAM bank is seen as a continuous range of 64 MB. *Assuming each row has* 1024 *words, the application can access* 4 KB *of data before encountering a precharge-activate delay. Less innocuous is the case of small objects that cross the boundary between rows and that are accessed entirely.*

With a row-bank-column organization, DMM pools would have to be organized in 1024*-word heaps, being impossible to allocate objects across different heaps (i.e., rows) without crossing to other banks. Therefore, each heap would have to be completely independent and serious fragmentation issues might appear, affecting not only to objects bigger than* 4 KB. *Then, there is also the complexity of managing thousands of heaps, one per row in each bank.*

In any case, the effect of different addressing options could be explored in the future. For example, Zhang et al. [ZZZ00] presented an address organization based on permutations that reduces the number of row conflicts due to L2-cache misses.

Finally, the simulator ignores the effect of DRAM refreshing on energy consumption and row activations. The datasheet of the modeled device mandates a refresh command to each row in a bank every 64 ms, which can be spread as a one-row refresh every 7.8125 µs or 8192 row refreshes clustered at the end of the 64 ms period (t_{REF}). The duration of a row refresh is specified by the t_{RFC} parameter (72 ns for the modeled device). The total accumulated time spent refreshing rows is $8192 \cdot 72\,\text{ns} / 64\,\text{ms} \approx 0.9\,\%$, which represents a small percentage

Table 3.1.: Definition of standard working parameters for SDRAMs. Timing parameters are usually provided in ns and rounded up to DRAM bus cycles.

NAME	UNITS	TYPICAL	DESCRIPTION
t_{CK}	ns	6	Clock cycle time.
CL	bus cycles	3	CAS (Column-Address Strobe) latency.
t_{CDL}	bus cycles	1	Last data-in to new READ/WRITE command.
t_{RAS}	bus cycles	7	ACTIVE-to-PRECHARGE command.
t_{RC}	bus cycles	10	ACTIVE-to-ACTIVE command period (same bank).
t_{RCD}	bus cycles	3	ACTIVE-to-READ-or-WRITE delay (**r**ow address to **c**olumn address **d**elay).
t_{RP}	bus cycles	3	PRECHARGE command period.
t_{WR}	bus cycles	3	Write recovery time.
I_{DD0}	mA	67.6	ACTIVE-PRECHARGE current (average, calculated).
I_{DD1}	mA	90.0	Operating current.
I_{DD3}	mA	18.0	Standby current, all banks active, no accesses in progress.
I_{DD4}	mA	130.0	Operating current, read/write burst, all banks active, half data pins change every cycle.
V_{DD}	V	1.8	Supply voltage.
V_{DDq}	V	1.8	I/O supply voltage (usually, $V_{DDq} = V_{DD}$).
C_{L0}	pF	20.0	Input/output pins (DQs) capacitance (for input, i.e., writes).
C_{LOAD}	pF	20.0	Capacitive load of the DQs (for output, i.e., reads).

of the total. Similarly, it usually represents around or well below 0.1 % of the total energy consumption. In any case, this factor should be easy to add in future revisions.

Given the previous assumptions, the simulator tracks (Figure 3.11):

- For the complete module, the state (reading or writing) and last access time (to know if new accesses extend the current burst or start a new one).
- For each bank, its active row, the time of the last write (to account for the write recovery time) and the time of the last row activation (to respect t_{RAS} and t_{RC}, Table 3.1).

The simulator does not track the specific read or write state of each bank, only if a different row needs to be activated. This is because I_{DD4} is measured for bursts of reads or writes to any bank and is presented in the datasheets as equal for both types of accesses.

3.5.1. Memory working parameters

The simulator uses the parameters defined in Table 3.1 to simulate SDRAMs. Timing parameters are usually provided by the manufacturers in nanoseconds (ns) and rounded up to DRAM bus cycles by the system designers.

CL (CAS or "Column-Address Strobe" latency) is the latency since a `READ` command is presented and the data are outputted. t_{RP} and t_{RCD} determine the time since a bank is precharged until the new row is active and ready to be accessed. t_{WR} is counted when the active row of a bank is changed to ensure that the last write succeeds. t_{RC} defines the minimum time between two row activations in the same bank. t_{RAS} defines both the minimum time between an `ACTIVATE` command and the next `PRECHARGE` to the same bank, and the maximum time that a row may remain active before being precharged. The simulator observes implicitly the minimum period for t_{RAS} as normally $t_{RC} = t_{RP} + t_{RAS}$. The maximum period, which forces

to reopen a row periodically, is ignored as typical values are in the order of 120 000 ns (the simulator assumes that rows may stay active indefinitely).

I_{DD4}, the current used during bursts, is calculated assuming that "address transitions average one transition every two clocks." This value applies for random reads or writes from/to any bank with no other banks precharging or activating rows at the same time.

3.5.2. Calculations

The following paragraphs explain how the simulator calculates some derived quantities.

***IDD*$_0$: Maximum operating current.** I_{DD0} is normally defined as the average current during a series of `ACTIVATE` to `PRECHARGE` commands to one bank. Micron's power estimation spreadsheet calculates it as follows:

$$I_{DD0} = I_{DD1} - 2 \cdot \frac{t_{CK}}{t_{RC}} \cdot (I_{DD4} - I_{DD3}) \tag{3.11}$$

Shortcuts. During the description of the simulation I use the following shortcuts to simplify the writing of long equations:

$$P_{ActPre} = I_{DD0} \cdot V_{DD} \tag{3.12}$$
$$P_{Read} = I_{DD4} \cdot V_{DD} \tag{3.13}$$
$$P_{Write} = I_{DD4} \cdot V_{DD} \tag{3.14}$$
$$t_{CPUCycle} = 1/CPUFreq \tag{3.15}$$

P_{ActPre} gives the power used during precharge and activation operations. As I_{DD0} is measured as an average over consecutive pairs of both commands, P_{ActPre} is also an average. In reality, the power required during precharging is much less than during a row activation; however, as both commands come in pairs, the overall result should be acceptable.

P_{Read} and P_{Write} represent the power required during a burst of reads or writes, respectively. $t_{CPUCycle}$ is calculated using the CPU frequency defined in the platform template file and is normally measured in ns.

I also use t_{Read} as a shortcut to represent the number of bus cycles that read data are presented by the memory module on the external pins. Correspondingly, t_{Write} represents the number of bus cycles that written data must be presented by the memory controller on the external data pins. Normally, both values are one cycle: $t_{Read} = t_{Write} = 1$.

Power of driving module pins. Equations (3.16) and (3.17) give the power required to drive the module pins during read or write accesses, respectively. The energy required for a complete $0 \to 1 \to 0$ transition corresponds to $C \cdot {V_{DDq}}^2$. Since the signals toggle at most once per clock cycle, their effective frequency is at most $0.5 \cdot CPUFreq/CPUToDRAMFreqFactor$.[6]

[6]The amount of transitions at the DRAM pins is completely data-specific. Therefore, a much better approximation could be achieved in future work by including during profiling the actual data values read or written to the memories, at the expense of a bigger log file size.

$$PDQ_r = \underbrace{32}_{\text{32 pins}} \cdot 0.5 \cdot \underbrace{C_{LOAD} \cdot {V_{DDq}}^2}_{\text{Transition energy}} \cdot \frac{CPUFreq}{CPUToDRAMFreqFactor} \tag{3.16}$$

$$PDQ_w = \underbrace{32}_{\text{32 pins}} \cdot 0.5 \cdot \underbrace{C_{L0} \cdot {V_{DDq}}^2}_{\text{Transition } E} \cdot \frac{CPUFreq}{CPUToDRAMFreqFactor} \tag{3.17}$$

To calculate the power needed for writes, the simulator assumes that the capacitive load supported by the memory controller (C_{L0}) is the same than the load driven by the memory during reads: $C_{L0} = C_{LOAD}$. Although this does not strictly correspond to energy consumed by the memory itself, it is included as part of the energy required to use it.

Finally, energy consumption is calculated multiplying P_{DQ} by the length of an access. Alternatively, the simulator could simply use $E = 32 \cdot C \cdot {V_{DDq}}^2$ and multiply by the number of (complete) transitions at the data pins.

Background power. Micron datasheets calculate the power required by each operation subtracting the current used by the module in standby mode, I_{DD3} (with all the banks active), from the total current used: $P_{ActPre} = (I_{DD0} - I_{DD3})V_{DD}$ and $P_{Read} = P_{Write} = (I_{DD4} - I_{DD3})V_{DD}$. Then, background power (the power required just by having all the modules active without doing anything) is calculated apart.

However, the simulator calculates directly the total energy consumed during these operations, using hence I_{DD0} and I_{DD4} directly. The energy consumed during real standby cycles is calculated later (Equation 3.18) using the total number of standby cycles counted by the simulator (i.e., cycles during which the DRAM banks were active but the DRAM was not responding to any access). I believe that both approaches are equivalent.

$$E_{Background} = I_{DD3} \cdot V_{DD} \cdot EmptyCycles \cdot t_{CPUCycle} \tag{3.18}$$

3.5.3. Simulation

The simulation is organized according to the state of the module and the type of the next operation executed. Every time that a row is activated, the simulator saves the operation time to know when the next activation can be initiated. Similarly, the simulator tracks the time of the last write to each bank to guarantee that the write-recovery time (t_{WR}) is met.

The following diagrams present all the timing components required during each access. Some parameters limit how soon a command can be issued, and are counted since a prior moment. Thus, they may have already elapsed at the time of the current access. These parameters are surrounded by square brackets ("[]"). This applies particularly to t_{RC}, which imposes the minimum time between two `ACTIVATE` commands to a bank required to guarantee that the row is refreshed in the cell array after opening it.

The simulator identifies periods of inactivity for the whole module, providing the count of those longer than 1000 CPU cycles and the length of the longest one. This information may be used to explore new energy-saving techniques.

3.5.3.1. From the IDLE state

The transition from the *IDLE* state happens in the simulator's model only once, at the beginning of every DRAM module simulation (Figure 3.12).

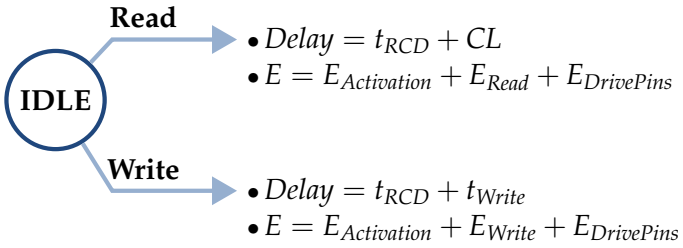


Figure 3.12.: Initial transition for an LPSDRAM module.

READ from IDLE. The banks are assumed to be precharged, so that the access has to wait for the row-activation time (t_{RCD}) and the time to access a word in the row buffer (CL). Equation (3.19) gives the energy required to complete the access. E_{Read} is calculated for the whole CL time to reflect the latency of the first access in a burst. As data are presented on the bus for just one cycle, the energy consumed driving external pins is confined to that time, represented by t_{Read}.

$$\begin{aligned} E &= E_{Activation} + E_{Read} + E_{DrivePins} = \\ &= (\underbrace{P_{ActPre} \cdot t_{RCD}}_{\text{Activation}} + \underbrace{P_{Read} \cdot CL}_{\text{Read}} + \underbrace{PDQ_r \cdot t_{Read}}_{\text{Drive pins}}) \cdot t_{CPUCycle} \end{aligned} \tag{3.19}$$

Notice that t_{RCD}, CL and t_{Read} are measured in bus cycles, whereas $t_{CPUCycle}$ is the length in seconds of each cycle. Ergo, each of t_{RCD}, CL and t_{Read} multiplied by $t_{CPUCycle}$ expresses a time period measured in seconds.

WRITE from IDLE. Similarly, a `WRITE` access has to wait until the required row is active and then one extra cycle (t_{Write}) during which data are presented on the bus to the memory module. Equation (3.20) gives the energy required to complete the write. The memory controller consumes energy driving the external pins during one cycle (t_{Write}).

$$\begin{aligned} E &= E_{Activation} + E_{Write} + E_{DrivePins} = \\ &= (\underbrace{P_{ActPre} \cdot t_{RCD}}_{\text{Activation}} + \underbrace{P_{Write} \cdot t_{Write}}_{\text{Write}} + \underbrace{PDQ_w \cdot t_{Write}}_{\text{Drive pins}}) \cdot t_{CPUCycle} \end{aligned} \tag{3.20}$$

3.5.3.2. From the READ state

Figure 3.13 shows the possible transitions from the `READ` state.

READ after READ with row change. A `READ` command that accesses a different row than the one currently active requires a full `PRECHARGE-ACTIVATE-READ` sequence. Additionally, a minimum of t_{RC} cycles must have elapsed since the last `ACTIVATE` command to that bank. Therefore, the simulator checks the last activation time for the bank and calculates the remaining part of t_{RC} that still needs to pass (under most normal conditions, it will be zero). Equation (3.21) shows how the components of the energy consumption of this operation are calculated:

$$E = (\overbrace{P_{ActPre} \cdot (t_{RP} + t_{RCD})}^{\text{Activation}} + \overbrace{P_{Read} \cdot CL}^{\text{Read}} + \overbrace{PDQ_r \cdot t_{Read}}^{\text{Drive pins}}) \cdot t_{CPUCycle} \tag{3.21}$$

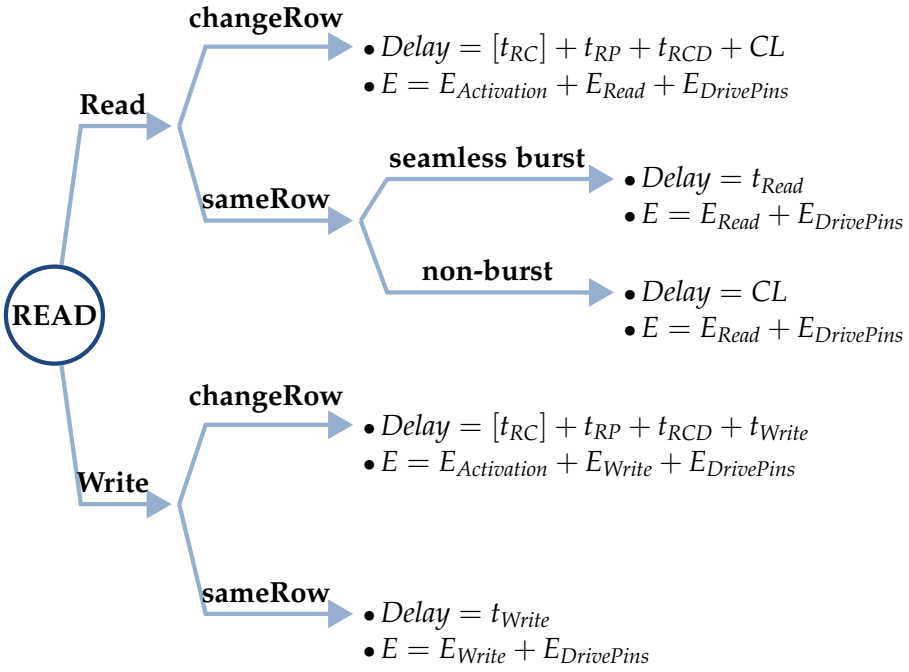


Figure 3.13.: Transitions after previous read accesses (LPSDRAM).

READ after READ in the active row. A `READ` command that accesses a word in the active row can proceed directly without more delays. However, the simulator makes a distinction between access that are consecutive in time and accesses that are separated by a longer time.

In the first case, the simulator assumes that the `READ` command belongs to the previous burst access. As the time to fill the pipeline (CL) was already accounted during that access, the delay for the current access is one cycle (t_{Read}). In other words, the memory controller will receive the data after CL cycles, but the delay of this access with respect to the previous is only one cycle. Figure 3.10 presented this case: The first read command at T_0 was answered at T_2; however, the accesses started at T_1, T_2 and T_3 received their data consecutively at T_3, T_4 and T_5. Equation (3.22) shows the detailed equation for energy consumption:

$$E = (\overbrace{P_{Read} \cdot t_{Read}}^{\text{Read}} + \overbrace{PDQ_r \cdot t_{Read}}^{\text{Drive pins}}) \cdot t_{CPUCycle} \tag{3.22}$$

In the second case, the current access is the first of a new burst. Therefore, the simulator accounts the full pipeline delay to it, CL. Equation (3.23) details energy consumption in this case:

$$E = (\overbrace{P_{Read} \cdot CL}^{\text{Read}} + \overbrace{PDQ_r \cdot t_{Read}}^{\text{Drive pins}}) \cdot t_{CPUCycle} \tag{3.23}$$

WRITE after READ with row change. A `WRITE` command that accesses a row different to the currently active one starts a full `PRECHARGE-ACTIVATE-WRITE` cycle. Similarly to other cases, the simulator checks whether the minimum t_{RC} time between to `ACTIVATE` commands has

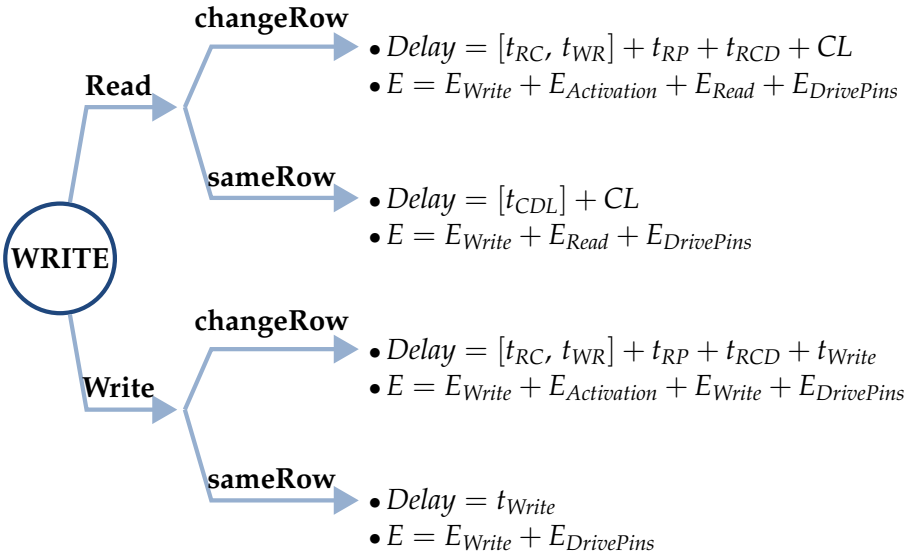


Figure 3.14.: Transitions after previous write accesses (LPSDRAM).

already been met or not. Equation (3.24) shows the details of energy consumption:

$$E = (\overbrace{P_{ActPre} \cdot (t_{RP} + t_{RCD})}^{\text{Activation}} + \overbrace{P_{Write} \cdot t_{Write}}^{\text{Write}} + \overbrace{PDQ_w \cdot t_{Write}}^{\text{Drive pins}}) \cdot t_{CPUCycle} \tag{3.24}$$

WRITE after READ in the active row. `WRITE` commands do not wait for a result, so they take one bus cycle as long as they access one of the active rows. Equation (3.25) shows the details of energy calculation in this case:

$$E = (\overbrace{P_{Write} \cdot t_{Write}}^{\text{Write}} + \overbrace{PDQ_w \cdot t_{Write}}^{\text{Drive pins}}) \cdot t_{CPUCycle} \tag{3.25}$$

3.5.3.3. From the WRITE state

Figure 3.14 shows the possible transitions from the `WRITE` state.

READ after WRITE with row change. A `READ` command after a `WRITE` that changes the active row has two peculiarities. First, before the `PRECHARGE` command can start, the minimum write-recovery time (t_{WR}) must be respected. Second, the new row cannot be activated until a minimum t_{RC} time since the previous `ACTIVATE` command has elapsed. Therefore, the simulator checks both times and delays the new `READ` operation until both times have been met. From this point, the read proceeds normally as in previous cases.

Equation (3.26) details how energy consumption is calculated in this case. Write accesses are accounted by the simulator in the reverse way than read accesses: Each of them has a delay of one cycle (t_{Write}), except the last one, which requires a few extra cycles to complete. As I_{DD4} is measured during bursts, it reflects the current that circulates during every cycle of a write burst, including all the stages of the pipeline. The last access adds the time required to empty the writing pipeline and meet the recovery time of the last write (i.e., its propagation time).

That explains the first term (E_{Write}) in the equation with P_{Write}.

$$E = (\overbrace{P_{Write} \cdot t_{WR}}^{\text{Finish prev. write}} + \overbrace{P_{ActPre} \cdot (t_{RP} + t_{RCD})}^{\text{Activation}} + \\ + \underbrace{P_{Read} \cdot CL}_{\text{Read}} + \underbrace{PDQ_r \cdot t_{Read}}_{\text{Drive pins}}) \cdot t_{CPUCycle} \tag{3.26}$$

READ after WRITE in the active row. When a `READ` command follows a `WRITE`, but in the same active row, the new access can proceed after meeting t_{CDL}, which represents the time needed from the last "data-in" to the next `READ` or `WRITE` command.[7] Equation (3.27) details the energy consumption for this case:

$$E = (\overbrace{P_{Write} \cdot t_{CDL}}^{\text{Finish write}} + \overbrace{P_{Read} \cdot CL}^{\text{Read}} + \overbrace{PDQ_r \cdot t_{Read}}^{\text{Drive pins}}) \cdot t_{CPUCycle} \tag{3.27}$$

WRITE after WRITE with row change. A `WRITE` command that changes the active row after a previous `WRITE` command has to wait for the write-recovery time (t_{WR}) and the minimum time between activations (t_{RC}). The simulator ensures that both timing restrictions have elapsed before continuing; both restrictions are served concurrently, not one after the other – that is why they are separated by commas and not added in Figure 3.14. Equation (3.28) shows each of the terms that add up to the energy consumption of this case:

$$E = (\overbrace{P_{Write} \cdot t_{WR}}^{\text{Finish prev. write}} + \overbrace{P_{ActPre} \cdot (t_{RP} + t_{RCD})}^{\text{Activation}} + \\ + \underbrace{P_{Write} \cdot t_{Write}}_{\text{Write}} + \underbrace{PDQ_w \cdot t_{Write}}_{\text{Drive pins}}) \cdot t_{CPUCycle} \tag{3.28}$$

WRITE after WRITE in an active row. Consecutive `WRITE` commands to any of the active rows can proceed normally one after the other, in consecutive cycles. Energy consumption during writes is calculated using I_{DD4}, which accounts for the total current during write bursts; thus, the energy consumed by chained writes each in a different stage is appropriately reflected. Equation (3.29) shows how energy consumption is calculated in this case:

$$E = (\overbrace{P_{Write} \cdot t_{Write}}^{\text{Write}} + \overbrace{PDQ_w \cdot t_{Write}}^{\text{Drive pins}}) \cdot t_{CPUCycle} \tag{3.29}$$

3.6. Simulation of LPDDR2-SDRAMs

DDR2-SDRAM devices use both edges of the clock signal to transmit data on the bus. Figure 3.15 illustrates how LPDDR2 devices use both edges of the clock signal to transfer data and the timing components involved in a burst of read operations. The simulation of LPDDR2-SDRAM devices is based on the specifications published by the JEDEC [JED11b]. Concrete data values were obtained from Micron datasheets [MIC12].

[7] Normally, t_{CDL} is 1. Therefore, the next access can proceed in the next cycle. If its value were bigger in a device, then the following cases of `WRITE`-to-`WRITE` should also be reviewed as t_{CDL} affects both reads and writes after a write.

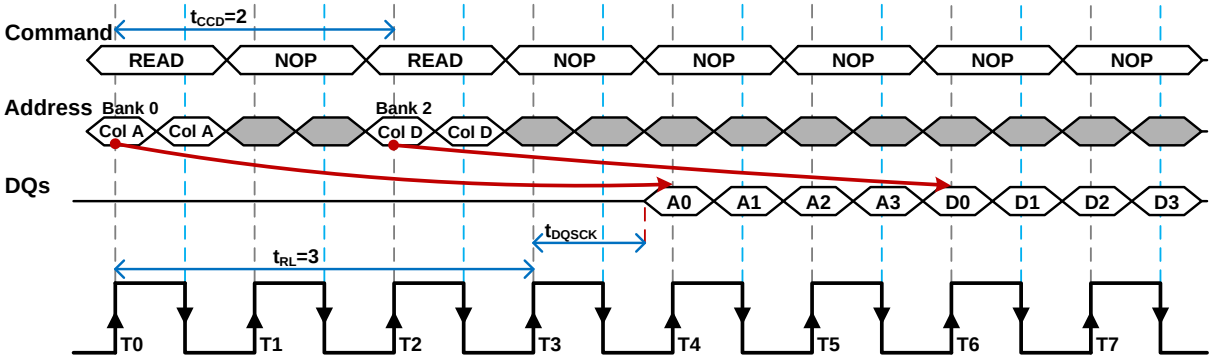


Figure 3.15.: Seamless burst read in an LPDDR2-Sx SDRAM with $t_{RL} = 3$, $BL = 4$ and $t_{CCD} = 2$. Each read command can be issued to any bank as long as they are active. LPDDR2 devices are worth two remarkable observations. First, data are presented on the *DQ* pins after $t_{RL} + t_{DQSCK}$ cycles; t_{DQSCK}, which represents the skew between the data-pins strobe (*DQS*) and CLK, may be longer than the clock period and is accounted *after* t_{RL} is over. Second, t_{CCD} may be longer than one cycle, which delays the moment when new commands can be issued.

The most relevant characteristic of LPDDR2-SDRAM devices is their n-prefetch architecture, in contrast with the pipelined architecture of Mobile SDRAMs. LPDDR2 devices are classified into S2 (2-prefetch) or S4 (4-prefetch) devices. This means that the devices must fetch 2 or 4 words before executing an operation, respectively. As data are transferred at both edges of the clock, operations can be canceled after any cycle for S2 devices, but only at every other cycle for S4 devices. $\mathcal{DynAsT}$'s simulator supports currently LPDDR2-S2 devices only.

The previous note is relevant because the minimum burst size for DDR2 devices is 4 words, which is in accordance with a memory hierarchy model in which DRAMs are mainly accessed to transfer cache lines. However, with the solutions generated by my methodology, the processor may execute accesses over individual memory words (e.g., when accessing just a pointer in a node). With an S2 device, the memory controller can issue burst-terminate (BST) commands to limit the size of an access to 2 words or simply chain consecutive accesses at every cycle (for $t_{CCD} = 1$) to achieve an effective $BL = 2$. Single-word writes can be accomplished using the write mask bits (*DM*) to disable writing of the odd (or even) word.

The results obtained in Chapter 4 for $\mathcal{DynAsT}$ solutions with LPDDR2-S2 memories incur an extra overhead for every individual access in terms of energy consumption. Future work may study performance with S4 devices or others with a higher prefetch index, taking into account the proportion of 1, 2 and 3-word accesses for each concrete application.

The working of LPDDR2 devices is considerably more complex than that of LPSDR devices. Therefore, the following paragraphs explain some key concepts in the simulation using excerpts directly from the JEDEC's specification.

Row activations.

> "The LPDDR2-SDRAM can accept a READ or WRITE command at time t_{RCD} after the ACTIVATE command is sent. [...] The minimum time interval between successive ACTIVATE commands to the same bank is determined by the RAS cycle time of the device (t_{RC})." [JED11b, p. 81]
>
> "The bank(s) will be available for a subsequent row access [...] t_{RPpb} after a single-bank `PRECHARGE` command is issued." [JED11b, p. 109]

The simulator assumes an "open-row" policy; hence, rows remain active until an access to a different one arrives. Opening a new row requires a PRECHARGE-ACTIVATE command sequence. Subsequent READ or WRITE commands can be sent by the memory controller after t_{RCD}. Table 3.2 details the timing components that intervene in the process: t_{RPpb}, the single-bank precharging time; t_{RAS}, the row-activation time and t_{RCD}, the ACTIVATE-to-READ-or-WRITE time.

The simulator does not implement any policy for proactively closing rows and thus does not currently account for the potential energy savings. This topic is worth future expansion.

Memory reads. The simulator assumes that all time-consecutive read accesses form a burst; in particular, bursts can include reads to any words in the set of active rows, no matter their absolute addresses. However, LPDDR2 devices have a minimum burst size of 4 words, transmitted during two bus cycles. Smaller transfers can be achieved issuing a TERMINATE command or interrupting the current read with a new READ command:

> "The seamless burst read operation is supported by enabling a READ command at every other clock for $BL = 4$ operation, every 4 clocks for $BL = 8$ operation, and every 8 clocks for $BL = 16$ operation. For LPDDR2-SDRAM, this operation is allowed regardless of whether the accesses read the same or different banks as long as the banks are activated." [JED11b, p. 91]
>
> "For LPDDR2-S2 devices, burst reads may be interrupted by other reads on any subsequent clock, provided that t_{CCD} is met." [JED11b, p. 91]

If $t_{CCD} > 1$, single or double-word accesses will incur extra overheads.

The first access in a burst reflects the complete delay of the pipeline ($t_{RL} + t_{DQSCK} + t_{DQSQ}$), but the memory outputs one word in each subsequent bus cycle (t_{Read} is normally one cycle). The simulator unifies both skew terms ($t_{DQSCK\_SQ} \equiv t_{DQSCK} + t_{DQSQ}$) and requires them in bus cycles (instead of ns as usually expressed in the datasheets). Figure 3.15 illustrates this situation:

> "The Read Latency (t_{RL}) is defined from the rising edge of the clock on which the READ command is issued to the rising edge of the clock from which the t_{DQSCK} delay is measured. The first valid datum is available $t_{RL} \cdot t_{CK} + t_{DQSCK} + t_{DQSQ}$ after the rising edge of the clock where the READ command is issued." [JED11b, p. 86]

Energy consumption is calculated for the whole duration of the access (CL cycles for the first one, 1 cycle for the next) because the family of $I_{DD4R\_}$ currents is measured during bursts and thus, it reflects the energy consumed by all the stages of the pipeline. A possible error source is that $I_{DD4R\_}$ might be slightly lower during the CL cycles of the first access as the operation is progressing through the pipeline, but there seems to be no available data in this respect.

A burst is broken by an inactivity period or an access to a different row. In those cases, the first access of the next burst will bear the complete latency. Inactivity periods (i.e., after the last word of a burst is outputted) happen if the processor accesses other memories for a while; for instance, because most of the accesses happen to an internal SRAM or cache memories are effective and thus DRAMs are seldom accessed. Accesses to words in a different row on any bank also break a burst. The simulator can be extended assuming that the PRECHARGE

command was sent to the affected bank as far back in time as the last access to that bank (every access command may include an optional PRECHARGE operation) and, therefore, simulate interleaving of bank accesses with no intermediate latencies.

Contrary to the case with Mobile SDRAMs, transitions from reads to writes, and vice versa, require extra delays:

> "For LPDDR2-S2 devices, reads may interrupt reads and writes may interrupt writes, provided that t_{CCD} is met. The minimum CAS-to-CAS delay is defined by t_{CCD}." [JED11b, p. 85]
>
> "The minimum time from the burst READ command to the burst WRITE command is defined by the read latency (t_{RL}) and the burst length (BL). Minimum READ-to-WRITE latency is $t_{RL} + \lceil t_{DQSCK_{max}}/t_{CK} \rceil + BL/2 + 1 - t_{WL}$ clock cycles. Note that if a read burst is truncated with a burst terminate (BST) command, the effective burst length of the truncated read burst should be used as BL to calculate the minimum READ-to-WRITE delay." [JED11b, p. 90]

The simulator observes these restrictions. It also assumes that the memory controller issues TERMINATE commands as needed for single or double-word accesses; thus, the equation is simplified because the effective BL is 2 and $BL/2 = 1$. This term is kept in this simplified form in the delay and energy consumption equations presented in the rest of this section to make it explicit.

Finally, LPDDR2 devices introduce additional restrictions for READ-to-PRECHARGE transitions, which were not present for Mobile SDRAM devices:

> "For LPDDR2-S2 devices, the minimum READ-to-PRECHARGE spacing has also to satisfy a minimum analog time from the rising clock edge that initiates the last 2-bit prefetch of a READ command. This time is called t_{RTP} (read-to-precharge). For LPDDR2-S2 devices, t_{RTP} begins $BL/2 - 1$ clock cycles after the READ command. [...] If the burst is truncated by a BST command or a READ command to a different bank, the effective BL shall be used to calculate when t_{RTP} begins." [JED11b, p. 110]

Memory writes. Write latencies are slightly more complex in LPDDR2 devices than in Mobile SDRAMs. The first write in a burst has an extra starting delay (t_{DQSS}). Once this delay is met, the memory controller has to provide one data word at each clock edge up to the length of the burst:

> "The write latency (t_{WL}) is defined from the rising edge of the clock on which the WRITE command is issued to the rising edge of the clock from which the t_{DQSS} delay is measured. The first valid datum shall be driven $t_{WL} \cdot t_{CK} + t_{DQSS}$ from the rising edge of the clock from which the WRITE command is issued." [JED11b, p. 94]

As expected, WRITE commands can be chained in consecutive bursts:

> "The seamless burst write operation is supported by enabling a WRITE command every other clock for $BL = 4$ operation, every four clocks for $BL = 8$ operation, or every eight clocks for $BL = 16$ operation. This operation is allowed regardless of same or different banks as long as the banks are activated." [JED11b, p. 97, Fig. 47]

> "For LPDDR2-S2 devices, burst writes may be interrupted on any subsequent clock, provided that $t_{CCD(min)}$ is met." [JED11b, p. 97]
>
> "The effective burst length of the first write equals two times the number of clock cycles between the first write and the interrupting write." [JED11b, p. 98, Fig. 49]

This last property and the write masks (*DM*) are used in the simulator to implement single-word writes. This aspect may be incorrect for devices with $t_{CCD} > 1$.

As with reads, LPDDR2 devices require a minimum time to transition from writing to reading in a bank, even when no active row changes are involved:

> "The minimum number of clock cycles from the burst WRITE command to the burst READ command for any bank is $\lceil t_{WL} + 1 + BL/2 + \lceil t_{WTR}/t_{CK} \rceil \rceil$.
>
> [...] t_{WTR} starts at the rising edge of the clock after the last valid input datum.
>
> [...] If a WRITE burst is truncated with a burst TERMINATE (BST) command, the effective burst length of the truncated write burst should be used as *BL* to calculate the minimum WRITE-to-READ delay." [JED11b, p. 96, Fig. 45]

Finally, LPDDR2 devices introduce also additional restrictions for WRITE-to-PRECHARGE transitions, which were neither present for Mobile SDRAM devices:

> "For write cycles, a delay must be satisfied from the time of the last valid burst input data until the PRECHARGE command may be issued. This delay is known as the write recovery time (t_{WR}) referenced from the completion of the burst write to the PRECHARGE command. No PRECHARGE command to the same bank should be issued prior to the t_{WR} delay. LPDDR2-S2 devices write data to the array in prefetch pairs (prefetch = 2) [...]. The beginning of an internal write operation may only begin after a prefetch group has been latched completely. [...] For LPDDR2-S2 devices, minimum WRITE-to-PRECHARGE command spacing to the same bank is $t_{WL} + \lceil BL/2 \rceil + 1 + \lceil t_{WR}/t_{CK} \rceil$ clock cycles. [...] For an [sic] truncated burst, *BL* is the effective burst length." [JED11b, p. 112]

Every time that the simulator has to change the active row in a bank, it checks if that bank is ready to accept new commands by evaluating if the last writing cycle plus the additional delays have already elapsed.

Energy consumption is calculated for one cycle (t_{Write}) for normal write accesses; however, when a burst is finished (due to inactivity), the next command is a `READ` or the activity switches to a different bank, the extra energy consumed by the bank finishing its work in the background during t_{WR} cycles is also accounted.

Write data mask. LPDDR2-SDRAM devices have a set of pins that act as byte masks during write operations. They can be used to inhibit overwriting of individual bytes inside a word. The simulator assumes that the memory controller exploits this capability so that individual words can be written seamlessly in one cycle (half cycle is used to access the word and the other half is wasted). Internally, the simulator does not distinguish if the written word is the even or the odd one. Instead, it counts the write access; if the next command is a write to the odd word, then it is ignored. This scheme covers the cases of writing the even, the odd or both words.

> "One write data mask (*DM*) pin for each data byte (*DQ*) will be supported on LPDDR2 devices, consistent with the implementation on LPDDR SDRAMs. Each data mask (*DM*) may mask its respective data byte (*DQ*) for any given cycle of the burst. Data mask has identical timings on write operations as the data bits, though used as input only, is internally loaded identically to data bits to insure matched system timing." [JED11b, p. 103]

As with Mobile SDRAMs, *DM* pins are accounted for energy consumed by the memory controller during writes.

Address organization. As explained for Mobile SDRAM devices,the simulator organizes memory addresses as "bank-row-column." Nevertheless, different addressing options could be explored in the future.

DRAM Refreshing. Similarly, the simulator ignores the effect of DRAM refreshing on energy consumption and row activations. LPDDR2-SDRAM devices reduce the maximum refresh period (t_{REF}) to 32 ms. However, the effect should still be small. This point should be easily modifiable if needed in the future.

3.6.1. Memory working parameters

The simulator uses the parameters defined in Table 3.2 to simulate SDRAMs. Timing parameters are usually provided by the manufacturers in ns and rounded up to DRAM bus cycles by system designers. t_{DQSCK} and t_{DQSQ} are used in ns for calculations in the JEDEC's specification; however, here they are converted (rounding up) to bus cycles as well.

3.6.2. Calculations

Shortcuts. During the description of the simulation I use the following shortcuts to simplify the writing of long equations:

$$P_{ActPre} = (I_{DDO1} \cdot V_{DD1} + I_{DDO2} \cdot V_{DD2} + I_{DDOin} \cdot V_{DDca}) \tag{3.30}$$

$$P_{Read} = (I_{DD4R1} \cdot V_{DD1} + I_{DD4R2} \cdot V_{DD2} + \\ + I_{DD4Rin} \cdot V_{DDca} + I_{DD4RQ} \cdot V_{DDq}) \tag{3.31}$$

$$P_{Write} = (I_{DD4W1} \cdot V_{DD1} + I_{DD4W2} \cdot V_{DD2} + I_{DD4Win} \cdot V_{DDca}) \tag{3.32}$$

$$t_{CPUCycle} = 1.0/CPUFreq \tag{3.33}$$

P_{ActPre} is calculated using I_{DD0} in a similar way than for Mobile SDRAM devices. However, LPDDR2 devices may use two different voltage sources and thus multiple currents appear: I_{DDO1}, I_{DDO2} and I_{DDOin} (for the input buffers). P_{ActPre} is the average power required during a series of `ACTIVATE`-`PRECHARGE` commands; thus, separating the power required for each operation is not feasible.

Similarly, LPDDR2-SDRAM devices distinguish read and write burst currents; even more, the specification separates them into their respective components. This can be seen in the equations for P_{Read} and P_{Write} in comparison with the case for Mobile SDRAMs.

$t_{CPUCycle}$ is calculated using the CPU frequency defined in the platform template file and normally measured in ns.

Table 3.2.: Definition of standard working parameters for LPDDR2-SDRAMs. Timing parameters are usually provided in ns and rounded up to DRAM bus cycles.

NAME	UNITS	TYPICAL	DESCRIPTION
BL	32-bit words	4	Burst length (programmable to 4, 8 or 16).
t_{CK}	ns	3	Clock cycle time.
t_{CCD}	bus cycles	1	CAS-to-CAS delay.
t_{DQSCK_SQ}	bus cycles	1	DQS output access time from $CK/CK\#$ plus $DQS - DQ$ skew.
t_{DQSQ}	bus cycles	< 1	$DQS - DQ$ skew.
t_{DQSS}	bus cycles	1	WRITE command to first DQS latching transition.
t_{RAS}	bus cycles	14	Row active time.
t_{RCD}	bus cycles	6	ACTIVE-to-READ-or-WRITE delay (RAS-to-CAS).
t_{RL}	bus cycles	5	READ latency.
t_{RPab}	bus cycles	6 to 7	Row PRECHARGE time (all banks).
t_{RPpb}	bus cycles	6	Row PRECHARGE time (single bank).
t_{RRD}	bus cycles	4	`ACTIVATE` bank A to `ACTIVATE` bank B.
t_{RTP}	bus cycles	3	READ-to-PRECHARGE command delay.
t_{WL}	bus cycles	2	WRITE latency.
t_{WR}	bus cycles	5	WRITE recovery time.
t_{WTR}	bus cycles	3	WRITE-to-READ command delay.
I_{DDO1}	mA	20.0	Operating one bank ACTIVE-PRECHARGE current (V_{DD1}).
I_{DDO2}	mA	47.0	Operating one bank ACTIVE-PRECHARGE current (V_{DD2}).
I_{DDOin}	mA	6.0	Operating one bank ACTIVE-PRECHARGE current (V_{DDca}, V_{DDq}).
I_{DD3N1}	mA	1.2	Active non power-down standby current (V_{DD1}).
I_{DD3N2}	mA	23.0	Active non power-down standby current (V_{DD2}).
I_{DD3Nin}	mA	6.0	Active non power-down standby current (V_{DDca}, V_{DDq}).
I_{DD4R1}	mA	5.0	Operating burst READ current (V_{DD1}).
I_{DD4R2}	mA	200.0	Operating burst READ current (V_{DD2}).
I_{DD4Rin}	mA	6.0	Operating burst READ current (V_{DDca}).
I_{DD4RQ}	mA	6.0	Operating burst READ current (V_{DDq}).
I_{DD4W1}	mA	10.0	Operating burst WRITE current (V_{DD1}).
I_{DD4W2}	mA	175.0	Operating burst WRITE current (V_{DD2}).
I_{DD4Win}	mA	28.0	Operating burst WRITE current (V_{DDca}, V_{DDq}).
V_{DD1}	V	1.8	Core power 1.
V_{DD2}	V	1.2	Core power 2.
V_{DDca}	V	1.2	Input buffer power.
V_{DDq}	V	1.2	I/O buffer power.
C_{IO}	pF	5.0	Input/output pins (DQ, DM, DQS_t, DQS_c) capacitance (for input, i.e., writes).
C_{LOAD}	pF	5.0	Capacitive load of the DQs (for output, i.e., reads).
t_{Read}	bus cycles	1	
t_{Write}	bus cycles	1	
CPUToDRAMFreqFactor	n/a	4 to 8	$CPUFreq/DRAMFreq$.
DQ	bit	32	Data pins.
DQS	bit	4×2	Strobe signals for DQs (differential).
DM	bit	4	Write data mask pins, one per each DQ byte.

Power of driving module pins. Equations (3.34) and (3.35) give the power required to drive the module pins during read or write accesses, respectively. The energy required for a complete $0 \rightarrow 1 \rightarrow 0$ transition corresponds to $C \cdot {V_{DDq}}^2$. Since DDR signals can toggle twice per clock cycle, their effective frequency is *CPUFreq*/*CPUToDRAMFreqFactor*, in contrast with the case of Mobile SDRAMs.[8]

The simulator calculates the power required to drive the data pins and the data-strobe signals for reads and writes. For writes, it adds also the power required to drive the write-mask pins.

$$PDQ_r = \underbrace{C_{LOAD} \cdot {V_{DDq}}^2}_{\text{Transition energy}} \cdot \underbrace{(DQ + DQS)}_{\text{Number of pins}} \cdot \frac{CPUFreq}{CPUToDRAMFreqFactor} \tag{3.34}$$

$$PDQ_w = \underbrace{C_{IO} \cdot {V_{DDq}}^2}_{\text{Transition } E} \cdot \underbrace{(DQ + DQS + DM)}_{\text{Number of pins}} \cdot \frac{CPUFreq}{CPUToDRAMFreqFactor} \tag{3.35}$$

To calculate the power needed for writes, the simulator assumes that the capacitive load supported by the memory controller (C_{IO}) is the same than the load driven by the memory during reads: $C_{IO} = C_{LOAD}$. Although this does not strictly correspond to energy consumed by the memory itself, it is included as part of the energy required to use it.

Finally, energy consumption is calculated multiplying P_{DQ} by the length of an access. Alternatively, the simulator could simply use $E = C \cdot {V_{DDq}}^2$ and multiply by the number of transitions and the number of data pins.

Background power. As for Mobile SDRAMs, $\mathcal{DynAsT}$'s simulator calculates the total energy consumed during each memory operation. The energy consumed during standby cycles (i.e., cycles during which the DRAM banks were active but the DRAM was not responding to any access) is calculated later (Equation 3.36) using the total number of standby cycles counted by the simulator:

$$\begin{aligned} E_{Background} = (&I_{DD3N1} \cdot V_{DD1} + I_{DD3N2} \cdot V_{DD2} + I_{DD3Nin} \cdot V_{DDca}) \cdot \\ &\cdot EmptyCycles \cdot t_{CPUCycle} \end{aligned} \tag{3.36}$$

3.6.3. Simulation

The simulation is organized according to the state of the module and the type of the next operation executed. Every time that a row is activated, the simulator saves the operation time to know when the next activation can be initiated. Similarly, the simulator tracks the time of the last write to each bank to guarantee that the write-recovery time (t_{WR}) is met.

The following diagrams present all the timing components required during each access. Some parameters limit how soon a command can be issued, and are counted since a prior moment. Thus, they may have already elapsed at the time of the current access. These parameters are surrounded by square brackets ("[]"). This applies particularly to t_{RC}, which imposes the minimum time between two `ACTIVATE` commands to a bank required to guarantee that the row is refreshed in the cell array after opening it.

[8] The amount of transitions at the DRAM pins is completely data-specific. Therefore, a much better approximation could be achieved in future work by including during profiling the actual data values read or written to the memories.

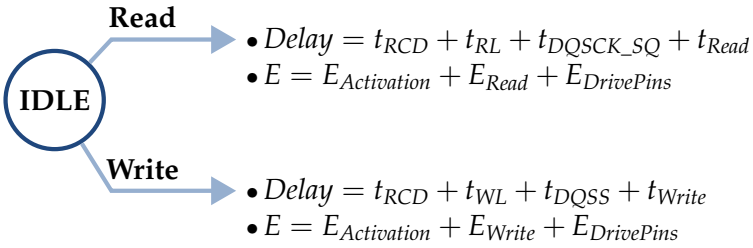


Figure 3.16.: Initial transition for an LPDDR2-SDRAM module.

The simulator identifies periods of inactivity for the whole module, providing the count of those longer than 1000 CPU cycles and the length of the longest one. This information may be used to explore new energy-saving techniques.

3.6.3.1. From the IDLE state

The transition from the *IDLE* state happens in the simulator's model only once, at the beginning of every DRAM module simulation (Figure 3.16).

READ from IDLE. The banks are assumed to be precharged, so that the access has to wait for the row-activation time (t_{RCD}) and the time for the first word to appear on the data bus. Equation (3.37) gives the energy required to complete the access. E_{Read} is calculated for the whole operation time to reflect the latency of the first access in a burst. However, as data are presented on the bus for just one cycle, the energy consumed driving external pins is confined to that time (t_{Read}).

$$\begin{aligned} E &= E_{Activation} + E_{Read} + E_{DrivePins} = \\ &= (\overbrace{P_{ActPre} \cdot t_{RCD}}^{\text{Activation}} + \overbrace{P_{Read} \cdot (t_{RL} + t_{DQSCK\_SQ} + t_{Read})}^{\text{Read}} + \\ &+ \underbrace{PDQ_r \cdot t_{Read}}_{\text{Drive pins}}) \cdot t_{CPUCycle} \end{aligned} \tag{3.37}$$

WRITE from IDLE. Assuming that all banks are precharged, a write access has to wait until the required row is active and then for the initial write latency ($t_{WL} + t_{DQSS}$) before the memory controller can present the first data word on the bus. Each pair of data words is then presented on the bus for one cycle (t_{Write}).

Equation (3.38) gives the energy required to complete the write. The memory controller is assumed to consume energy driving the external pins during one cycle (t_{Write}).

$$\begin{aligned} E &= E_{Activation} + E_{Write} + E_{DrivePins} = \\ &= (\overbrace{P_{ActPre} \cdot t_{RCD}}^{\text{Activation}} + \overbrace{P_{Write} \cdot (t_{WL} + t_{DQSS} + t_{Write})}^{\text{Write}} + \\ &+ \underbrace{PDQ_w \cdot t_{Write}}_{\text{Drive pins}}) \cdot t_{CPUCycle} \end{aligned} \tag{3.38}$$

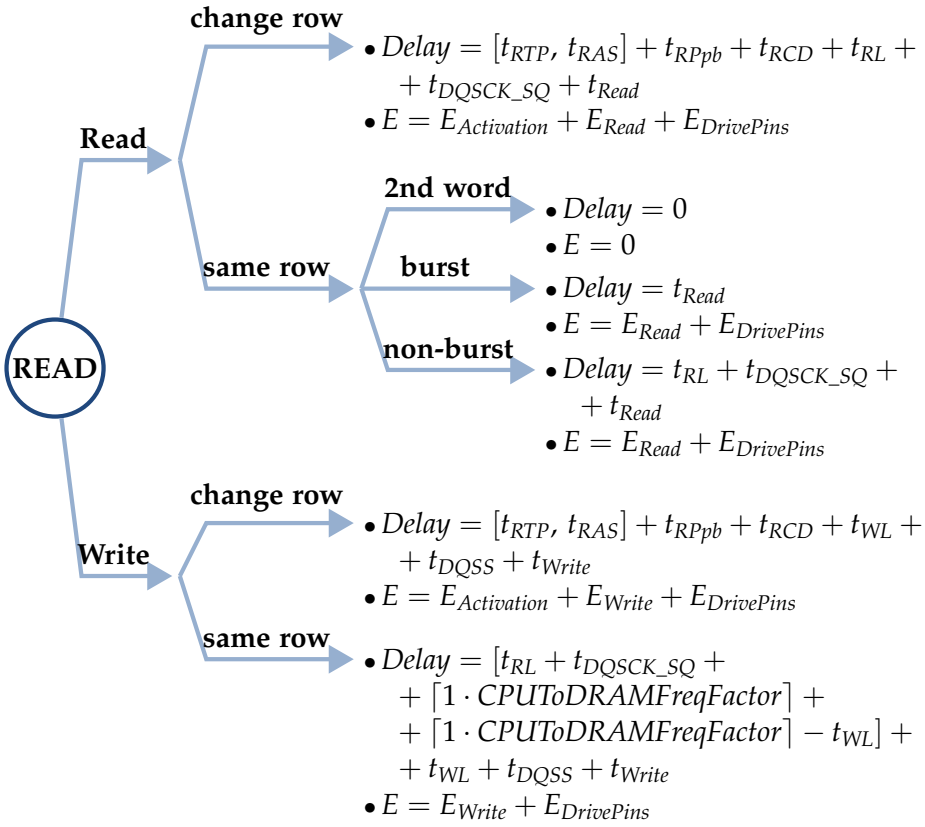


Figure 3.17.: Transitions after previous read accesses (LPDDR2-SDRAM).

3.6.3.2. From the READ state

Figure 3.17 shows the possible transitions from the READ state.

READ-to-READ with row change. A READ command that accesses a different row than the one currently active requires a full PRECHARGE-ACTIVATE-READ sequence. Additionally, the new access has to meet both the READ-to-PRECHARGE time (t_{RTP}) and the minimum time (t_{RAS}) that a row must be active (to ensure that the bit cells in the DRAM array are restored). Therefore, the simulator checks the last activation and operation times for the bank and calculates the remaining part that still needs to pass (under most normal conditions, it will be zero). Energy consumption is calculated with Equation (3.39):

$$E = (\overbrace{P_{ActPre} \cdot (t_{RPpb} + t_{RCD})}^{\text{Activation}} + \overbrace{P_{Read} \cdot (t_{RL} + t_{DQSCK\_SQ} + t_{Read})}^{\text{Read}} + \\ + \underbrace{PDQ_r \cdot t_{Read}}_{\text{Drive pins}}) \cdot t_{CPUCycle} \tag{3.39}$$

READ-to-READ in the active row. Consecutive reads present three possibilities. The first one is that the access corresponds to the second word of a two-word transfer: The LPDDR2 module transfers two words per bus cycle, but the simulator sees the individual accesses from the memory access trace. The simulator tackles with this situation by accounting for the total delay and energy during the first access and, if the immediately consecutive access corresponds to

the next word, both the latency and the energy consumption are counted as zero.

The second possibility is that the access belongs to an ongoing burst or to the first access in a chained ("seamless") burst. As the time to fill the pipeline was already accounted for the first access in the burst, the delay of this access with respect to the previous is only one cycle. Figure 3.15 presented this case: The first READ command with $BL = 4$ at T_0 was answered during T_4 and T_5, and the next READ (originated at T_2) was answered during T_6 and T_7. To achieve $BL = 2$ accesses in $t_{CCD} = 1$ devices, READ commands have to be presented in consecutive cycles, thus effectively terminating the previous burst ($BL = 4$ is the minimum supported by the standard, but for LPDDR2-S2 devices with $t_{CCD} = 1$ this mode of operation is allowed).

Equation (3.40) accounts for the energy consumption of this case. This access corresponds to reading and transferring two words; thus, the simulator saves the address to check if the next access corresponds to the word transferred in the second half of the bus cycle (i.e., the previous paragraph).

$$E = (\overbrace{P_{Read} \cdot t_{Read}}^{\text{Read}} + \overbrace{PDQ_r \cdot t_{Read}}^{\text{Drive pins}}) \cdot t_{CPUCycle} \tag{3.40}$$

Finally, if the access starts a new burst that is not consecutive (in time) to the previous one, it bears the full starting cost for the burst. Equation (3.41) details the energy consumption for this case:

$$E = (\overbrace{P_{Read} \cdot (t_{RL} + t_{DQSCK\_SQ} + t_{Read})}^{\text{Read}} + \overbrace{PDQ_r \cdot t_{Read}}^{\text{Drive pins}}) \cdot t_{CPUCycle} \tag{3.41}$$

READ-to-WRITE with row change. A WRITE command that accesses a row that is not the currently active one starts a full PRECHARGE-ACTIVATE-WRITE cycle. Again, the new access has to meet both the READ-to-PRECHARGE time (t_{RTP}) and the minimum time (t_{RAS}) that a row must be active. Therefore, the simulator checks the last activation and operation times for the bank and calculates the remaining part that still needs to pass. The memory controller can present the first data word on the data bus once the new row is active ($t_{RPpb} + t_{RCD}$) and the starting delays for the write burst are met ($t_{WL} + t_{DQSS}$). Each pair of words is presented on the bus during one cycle (t_{Write}).

Equation (3.42) shows the details of energy consumption. The energy consumed during the first cycles of a burst is accounted for this access.

$$\begin{aligned} E = (&\overbrace{P_{ActPre} \cdot (t_{RPpb} + t_{RCD})}^{\text{Activation}} + \overbrace{P_{Write} \cdot (t_{WL} + t_{DQSS} + t_{Write})}^{\text{Write}} + \\ &+ \underbrace{PDQ_w \cdot t_{Write}}_{\text{Drive pins}}) \cdot t_{CPUCycle} \end{aligned} \tag{3.42}$$

READ-to-WRITE in the active row. A WRITE command can follow (or interrupt) a previous READ command to the active row after a minimum delay calculated as shown in Figure 3.17. As explained previously, the simulator assumes that the minimum burst size for LPDDR2-S2 devices is in effect $BL = 2$, so that the factor $BL/2$ in the specification is simplified to 1 in the figure. However, all the terms (and rounding operators) are kept to make explicit their existence – the simulator multiplies internally all timings by *CPUToDRAMFreqFactor* so this factor remains here. After that, the write operation has the usual starting delay.

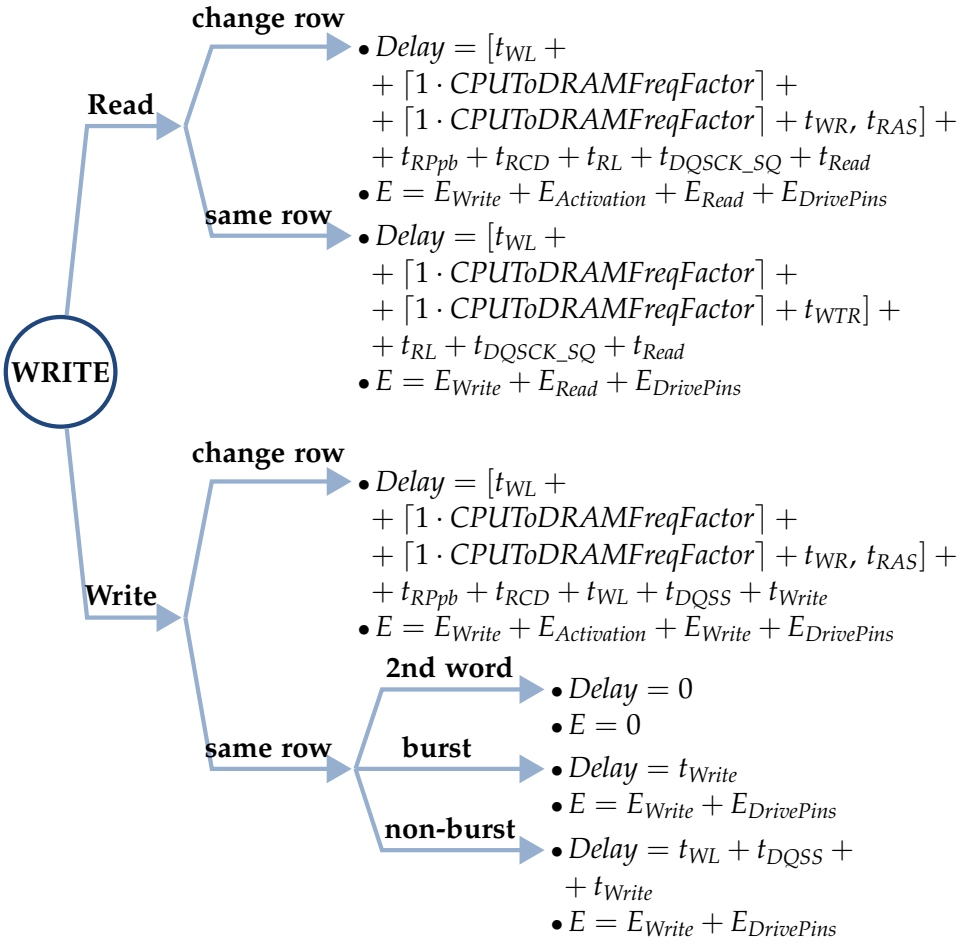


Figure 3.18.: Transitions after previous write accesses (LPDDR2-SDRAM).

Equation (3.43) shows the details of energy consumption:

$$E = (\overbrace{P_{Write} \cdot (t_{WL} + t_{DQSS} + t_{Write})}^{\text{Write}} + \overbrace{PDQ_w \cdot t_{Write}}^{\text{Drive pins}}) \cdot t_{CPUCycle} \tag{3.43}$$

3.6.3.3. From the WRITE state

Figure 3.18 shows the possible transitions from the `WRITE` state.

WRITE-to-READ with row change. A `READ` command that requires a row-change in a bank can follow (or interrupt) a previous writing burst after a minimum delay (Figure 3.18), which includes the write-recovery time (t_{WR}). Additionally, the simulator checks that t_{RAS} has already elapsed since the last `ACTIVATE` command to that bank. Both conditions are independent and thus appear separated by commas in the diagram. After that, the normal `ACTIVATE`-to-`READ` delays apply.

Equation (3.44) shows each of the terms that add up to the energy consumption of this case, including the energy consumed during the final phase of the previous write operation (E_{Write}):

$$E = (\overbrace{P_{Write} \cdot t_{WR}}^{\text{Finish prev. write}} + \overbrace{P_{ActPre} \cdot (t_{RPpb} + t_{RCD})}^{\text{Activation}} + \\ + \overbrace{P_{Read} \cdot (t_{RL} + t_{DQSCK\_SQ} + t_{Read})}^{\text{Read}} + \overbrace{PDQ_r \cdot t_{Read}}^{\text{Drive pins}}) \cdot t_{CPUCycle} \tag{3.44}$$

WRITE-to-READ in the active row. As in the previous case, A `READ` command in the active row of a bank can follow (or interrupt) a previous writing burst after a minimum delay (Figure 3.18). In this case, t_{WR} is substituted by t_{WTR}, the `WRITE`-to-`READ` command delay. However, as a change of active row is not involved in this case, the normal delay for a `READ` command applies directly after that.

Equation (3.45) details the energy consumption for this case, including the energy consumed during the final phase of the previous write operation (E_{Write}):

$$E = (\overbrace{P_{Write} \cdot t_{WTR}}^{\text{Finish write}} + \overbrace{P_{Read} \cdot (t_{RL} + t_{DQSCK\_SQ} + t_{Read})}^{\text{Read}} + \\ + \underbrace{PDQ_r \cdot t_{Read}}_{\text{Drive pins}}) \cdot t_{CPUCycle} \tag{3.45}$$

WRITE-to-WRITE with row change. A `WRITE` command to a different row can follow (or interrupt) a previous writing burst to the same bank after a minimum delay (Figure 3.18), which includes the write-recovery time (t_{WR}). Additionally, the simulator checks that t_{RAS} has already elapsed (in parallel, not consecutively) since the last `ACTIVATE` command to that bank. After that, the normal `ACTIVATE`-to-`WRITE` delays apply.

Equation (3.46) shows each of the terms that add up to the energy consumption of this case, including the energy consumed during the final phase of the previous write operation (E_{Write}):

$$E = (\overbrace{P_{Write} \cdot t_{WR}}^{\text{Finish prev. write}} + \overbrace{P_{ActPre} \cdot (t_{RPpb} + t_{RCD})}^{\text{Activation}} + \\ + \underbrace{P_{Write} \cdot (t_{WL} + t_{DQSS} + t_{Write})}_{\text{Write}} + \underbrace{PDQ_w \cdot t_{Write}}_{\text{Drive pins}}) \cdot t_{CPUCycle} \tag{3.46}$$

WRITE-to-WRITE in the active row.

Consecutive `WRITE` commands to the active row of a bank come in three flavors. First, if the access corresponds to the second word of a DDR two-word transfer, both the latency and the energy consumption are counted as zero because the simulator already accounted for them when it encountered the first access.

Second, if the access belongs to an ongoing burst or is the first access in a chained ("seamless") burst, the delay of this access with respect to the previous is only one cycle. LPDDR2-S2 devices support $BL = 2$ bursts if a `WRITE` command follows immediately the previous, provided that the device supports $t_{CCD} = 1$.

Equation (3.47) accounts for the energy consumption of this case. This access corresponds to the first of two words; thus, the simulator saves the address to check if the next access corresponds to the word transferred in the second half of the bus cycle (i.e., as per the previous paragraph).

$$E = (\overbrace{P_{Write} \cdot t_{Write}}^{\text{Write}} + \overbrace{PDQ_w \cdot t_{Write}}^{\text{Drive pins}}) \cdot t_{CPUCycle} \tag{3.47}$$

Finally, if the access starts a new burst that is not consecutive (in time) to the previous one, it bears the full starting cost for the burst. Contrary to the case of Mobile SDRAMs, in LPDDR2-DRAMs this first `WRITE` access in a burst has an extra delay of $t_{WL} + t_{DQSS}$. Equation (3.48) details the energy consumption for this case:

$$E = (\overbrace{P_{Write} \cdot (t_{WL} + t_{DQSS} + t_{Write})}^{\text{Write}} + \overbrace{PDQ_w \cdot t_{Write}}^{\text{Drive pins}}) \cdot t_{CPUCycle} \tag{3.48}$$

3.6.4. A final consideration

The simulation of LPDDR2 devices is a complex topic that depends on many factors that are difficult to measure, such as the currents that circulate through the device during the initial cycles of a read burst. Additionally, the reader might find errors or inaccuracies in my interpretation of the latency and energy costs for some of the cases. Any mistakes that affected access costs might change the relative performance of cache or SRAM-based solutions as the first ones tend to execute many more accesses over the DRAM. For example, if the cost of the operations were found to be lower than as calculated by $\mathcal{DynAsT}$'s simulator, the distance between cache and SRAM solutions would narrow. Correspondingly, if the real cost were found to be higher, the distance would increase.

To provide some assurance on error-bounding, I would like to point out that the energy consumption of both solutions is approximately proportional to the factor between the number of DRAM accesses performed by each one. Therefore, if an SRAM solution produces 0.75 times the number of DRAM accesses than a cache solution, the distance between them should be bounded by at least that same factor. This distance is increased with the factor of SRAM-accesses times their energy consumption and cache-accesses times their cost, the latter being usually higher for the same capacity.

Chapter 4

Experiments on data placement: Results and discussion

In this chapter I present several case studies composed of application models and synthetic benchmarks to evaluate the improvements attained with the use of *DynAsT* and my methodology. The first case study takes a detailed journey through all the phases of the optimization process. The other two cases present the improvements obtained, highlighting the effects of data placement with explicitly addressable memories in comparison with traditional cache memories. In summary, this chapter presents a broad set of experiments on three case studies, using two technologies of main memory (Mobile SDRAM and LPDDR2-SDRAM) and evaluating more than a hundred platforms for each case, including multiple combinations of DRAM, SRAMs and caches. The experiments compare the cost of executing an application in a platform with hardware-based caches or in a platform with one (or several) explicitly addressable on-chip SRAMs managed via the dynamic memory manager as proposed in this text.

The first case models a network of wireless sensors where the devices have to process a moderate amount of data; the focus is here on reducing energy consumption to prolong battery life. The second experiment uses the core of a network routing application as an example of high performance data processing. Finally, the third experiment is a small benchmark representative of DDT-intensive applications.

In this text I have tried to provide as much information as possible to ease the evaluation of the methodology advantages. Thus, the interested reader should be able to reproduce the results presented either with the same simulator or building an equivalent one based on the descriptions of the previous chapters.

DynAsT was configured with the following parameters for all the experiments:

- $MaxGroups = +\infty$;
- $MinIncFPB = 1.0$ and $MinIncExpRatio = 1.0$: Any increase on FPB or exploitation ratio is accepted;
- $SpreadPoolsInDRAMBanks = True$;
- $MinMemoryLeftOver = 8\,\text{B}$;
- $PoolSizeIncreaseFactor = 1.0$;
- $PoolSizeIncreaseFactorForSplitPools = 1.3$;

- $UseBackupPool = True$.

With these values as a common base line for the experiments, $\mathcal{DynAsT}$ is able to produce solutions that improve on those obtained with traditional caching techniques. Further improvements may be achieved with a careful tuning of these parameters to the characteristics of each particular application.

4.1. Description of the memory hierarchies

During our experiments, we used a big number of memory organizations to test the methodology with each of the applications. Here, I present only the most relevant configurations for each one.

The technical parameters of the SRAM and cache memories were obtained via Cacti [HP 08] using a 32 nm feature size. Tables 4.1 and 4.2 detail their respective technical parameters. The values for energy consumption in Table 4.1 present some unexpected variations. For example, the energy consumed during an access by a 4 KB direct mapped cache is higher than that consumed by a 32 KB 2-way associative cache. As far as I have checked them, these values are consistent and correspond to the particular characteristics of signal routing inside big circuits. As a backup to my assumption, the following quote by Hennessy and Patterson [HP11, p. 77, Fig. 2.3] points in the same direction regarding the evolution of access times:

> "Access times generally increase as cache size and associativity are increased. [...] The assumptions about cache layout and the complex trade-offs between interconnect delays (that depend on the size of a cache block being accessed) and the cost of tag checks and multiplexing lead to results that are occasionally surprising, such as the lower access time for a 64 KB with two-way set associativity versus direct mapping. Similarly, the results with eight-way set associativity generate unusual behavior as cache size is increased. Since such observations are highly dependent on technology and detailed design assumptions, tools such as CACTI serve to reduce the search space rather than precision analysis of the trade-offs."

I configured the cache memories for the experiments using as reference the ARM Cortex-A15 [ARM11]: 64-byte (16 words) line size, associativity of up to 16 ways, up to 4 MB in size (for the L2 caches) and Least Recently Used (LRU) replacement policy. I present multiple different configuration cases to explore the implications of these design options (Figure 4.1). Among others, I used configurations with sizes from 4 KB up to 4 MB; direct mapped ("D"), 2-way ("A2"), 4-way ("A4") and 16-way ("A16") associative; 16 (default) and 4 ("W4") words per line. All the cache memories use an LRU replacement policy. Finally, I include in all the experiments special configurations labeled as "lower bound" that represent the minimum theoretical cost for a cache memory with a big size (256 MB), but a small cost (comparable to that of a 4 KB one).

$\mathcal{DynAsT}$ generates solutions for configurations that consist of one or several on-chip SRAMs (Figure 4.2) and an external DRAM. These configurations are labeled as "SRAM," where their labels enumerate the independent memory modules that are included. For instance, a configuration labeled as "SRAM 8x512KB" has 8 modules of 512 KB each. I include a base configuration with SRAM modules of 512 B, 1 KB, 32 KB and 256 KB in all the experiments to serve as a common reference. Finally, I also include a configuration labeled as "lower bound" with an on-chip SRAM of 1 GB and the properties of a 4 KB memory to provide a lower cost bound.

Table 4.1.: Technical parameters of the cache memories.

Size	Associativity	Line size Words	Energy nJ	Latency Cycles	Area mm^2
4 KB	Direct	16	0.154	1	0.021
32 KB	2 ways	16	0.102	1	0.075
64 KB	4 ways	16	0.119	1	0.140
16 KB	16 ways	16	0.166	1	0.137
32 KB	16 ways	16	0.166	1	0.203
32 KB	16 ways	4	0.024	1	0.100
64 KB	16 ways	16	0.179	1	0.263
64 KB	16 ways	4	0.030	1	0.158
256 KB	16 ways	16	0.250	2	0.609
256 KB	16 ways	4	0.068	2	0.533
512 KB	16 ways	16	0.345	2	1.069
1 MB	16 ways	16	0.509	4	2.088
4 MB	16 ways	16	2.124	6	8.642

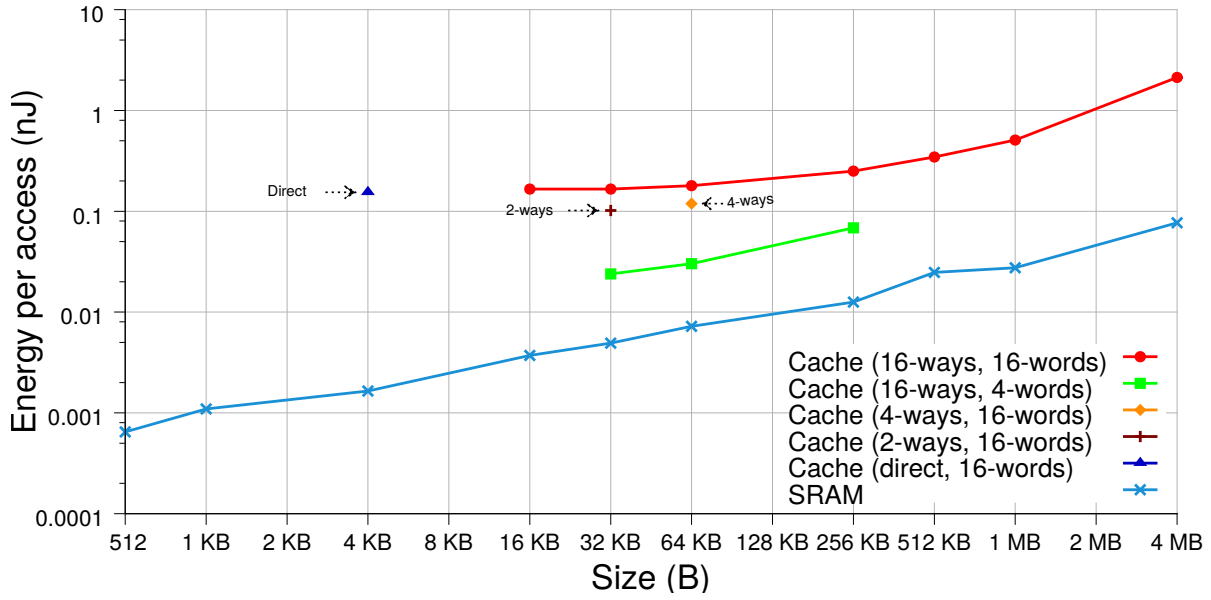


Figure 4.1.: Energy per access for the cache and SRAM configurations used in the experiments (logarithmic scale).

Table 4.2.: Technical parameters of the (on-chip) SRAMs.

Size	**Energy** nJ	**Latency** Cycles	**Area** mm^2
512 B	< 0.001	1	0.003
1 KB	0.001	1	0.005
4 KB	0.002	1	0.012
16 KB	0.004	1	0.045
32 KB	0.005	1	0.112
64 KB	0.007	1	0.185
256 KB	0.013	2	0.781
512 KB	0.025	2	1.586
1 MB	0.028	4	2.829
4 MB	0.077	6	11.029

As I have explained previously, my methodology allows the designer to easily combine SRAMs with caches. Although in this set of experiments the effect of these combinations is not relevant, they may become useful in cases where a small set of data instances gets a low but still significant number of accesses (with some locality) in the DRAM modules.

Finally, the Mobile SDRAM and LPDDR2-SDRAM modules were configured according to manufacturer datasheets as presented in Sections 3.5 and 3.6.

4.2. Case study 1: Wireless sensors network

In this first case study I apply the methodology step by step, optimizing the application for several platform configurations with SRAMs of varying sizes, to provide a global view of the process. The subject application is a model of a network of wireless sensors that can be distributed over wide areas. Each sensor extracts information of its surroundings and sends it through a low-power radio link to the next sensor in the path towards the base station. The networks constructed in this way are very resilient to failures and terrain irregularities because the nodes can find new paths. They may be used for applications such as weather monitoring [IBS+10] and fire detection in wild areas or radio signal detection in military operations. Each sensor keeps several hash tables and input and output queues to provide some basic buffering capabilities. The sensors use dynamic memory due to the impossibility of determining the network dependencies and the need of adjusting the use of resources to what is strictly needed at each moment. As a consequence, the sizes of the hash tables and network queues, among others, need to be scaled. The application model creates a network of sensors and monitors one in the middle of the transmission chain; that node forwards the information from more distant ones and gathers and sends its own information periodically. In the next paragraphs I explain how the different steps of the methodology are performed and their outcome.

4.2.1. Profiling

Instrumenting the application requires modifying 9 lines out of 3484. That means that just a 0.26 % of the source code has to be modified both for profiling and deployment. After

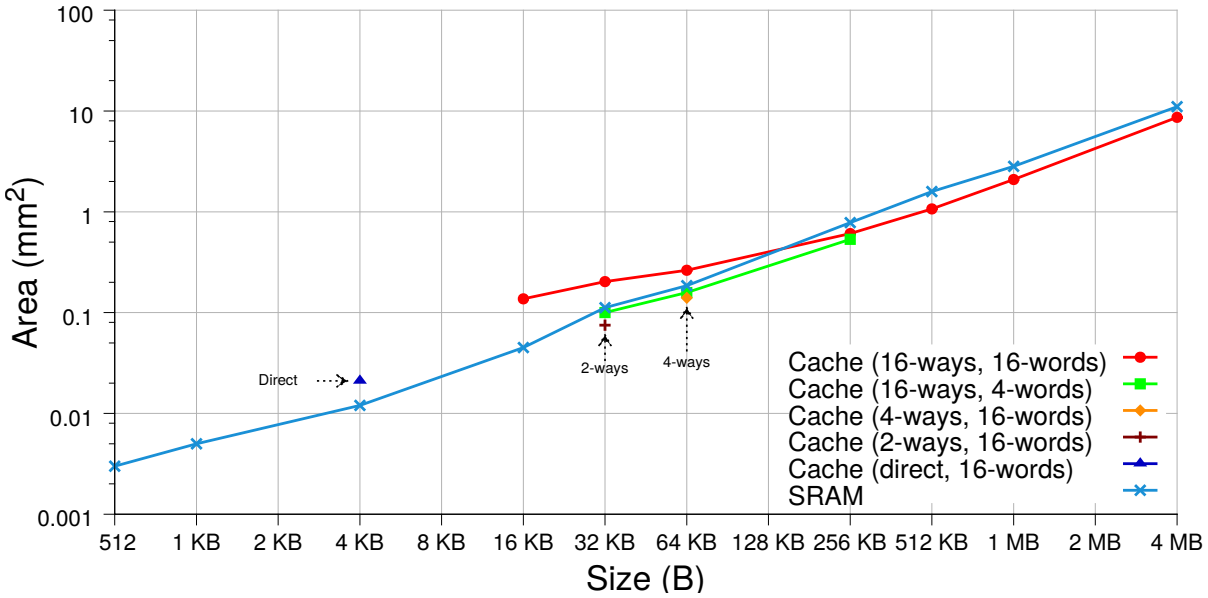


Figure 4.2.: Area of the different cache and SRAM configurations used in the experiments (logarithmic scale).

instrumentation, the model ran normally during 4 hours of real time, producing a log file of 464 MB.

4.2.2. Analysis

DynAsT analyzes the log file obtained through profiling in about 29 seconds, identifying 21 distinct DDTs. The FPBs of the DDTs range from 8.8×10^5 accesses per byte down to 0.61 accesses per byte. The maximum footprint required for a given DDT is 24 624 B and the minimum, 12 B. The size of the hash tables varies during the application execution; for instance, the hash table for the active neighbors uses internal arrays of 804 B, 1644 B, 3324 B, 6684 B and 13 404 B. *DynAsT* detects this and separates the different instances of the hash table DDT (classifying them according to their different sizes) for the next steps.

4.2.3. Grouping

The grouping step runs in about 70 seconds with the parameters explained at the beginning of this section. *DynAsT* builds a total of 12 groups from the initial set of 21 DDTs. One of the groups has 5 DDTs, one group gets 3 DDTs, three groups hold 2 DDTs and the last seven groups contain just 1 DDT. The grouping step manages to reduce the total footprint of the application (compared to the case of one DDT per pool) from 110 980 B to 90 868 B, thus achieving an overall footprint reduction of an 18.12 % and reducing from 21 to 12 the number of pools that have to be managed (−42.9 %). For the five groups that the tool generates combining several DDTs, the respective memory footprint reductions are 32.6 %, 56.4 %, 19.9 %, 51.7 % and 19.6 % when compared to the space that would be required if each DDT were mapped into an independent pool.

Figure 4.3 gives more insights into the grouping step. *DynAsT* identifies DDT_2 and DDT_6 as compatible and combines them in the same group, yielding a significant reduction in memory usage. If each DDT were mapped into an independent pool, that is, building per-DDT

pools, the total memory space required would be 1032 B (thick black horizontal line in the graph). With grouping, the required footprint is reduced to just 696 B (green horizontal line), a reduction of 32.6 %.

The figure shows also how the maximum memory footprint of the DDTs and the group is determined by instantaneous peaks in the graph. The designer might use the `PoolSizeIncreaseRatio` parameter during the mapping phase to adjust the final footprint so that it covers only the space required during most of the time, relying on the use of a backup pool in DRAM to absorb the peaks. However, there is an important drawback to this. Although the freed space could then be exploited by other DDTs (instead of staying unused when the footprint of the group is lower), it is possible that the instances created during the peak footprint periods, and that would have to be allocated in the backup pool, get so many accesses that the total system performance is reduced. After all, the instances of the DDTs included in the group supposedly have a higher FPB than the instances of DDTs included in the next groups. This consideration shows the importance of the grouping step to improve the exploitation ratio of the memory space assigned to each DDT and of the simulation step to predict the consequences of different parameter values.

Finally, the footprint evolution of the two DDTs and the combined group can be observed in more detail in the inset in Figure 4.3, which presents a randomly-chosen period during the execution time of the application. The fact that the footprint peaks of DDT_2 are not coincident with the peaks of DDT_6 is the factor that enables their grouping.

4.2.4. Pool formation

For this experiment, an always-coalesce-and-split scheme with free lists per size was used. When a block is allocated, any extra space is separated and inserted into a list of free blocks; when a block is deallocated, it is fused with the previous and/or next blocks if they are also free before being inserted into the matching list of free blocks.

$DynAsT$'s simulator does not currently consider the accesses of the dynamic memory managers (only the application accesses are traced). Therefore, it cannot be used to analyze the impact of the different DMMs generated during this phase. Nevertheless, it should be straightforward to introduce this functionality by linking the library of DMMs with the simulator itself so that every time that the DMM code needs to read or update an internal structure, these accesses are reproduced in the simulated platform.

4.2.5. Mapping

Mapping is the first platform-dependent step in the methodology. In this case, the mapping and simulation steps were run for more than 200 different platforms to verify that the methodology achieves better results than cache memories for many different potential platform configurations. Due to the small footprint of this application, I include in these experiments several configurations with reduced memory subsystems (down to just 32 KB). However, it is important to stress that, if the platform is already decided, the designer has to run the mapping and simulation steps just once with the platform parameters. $DynAsT$ executes this step with the parameters detailed at the beginning of this section in less than one second per configuration.

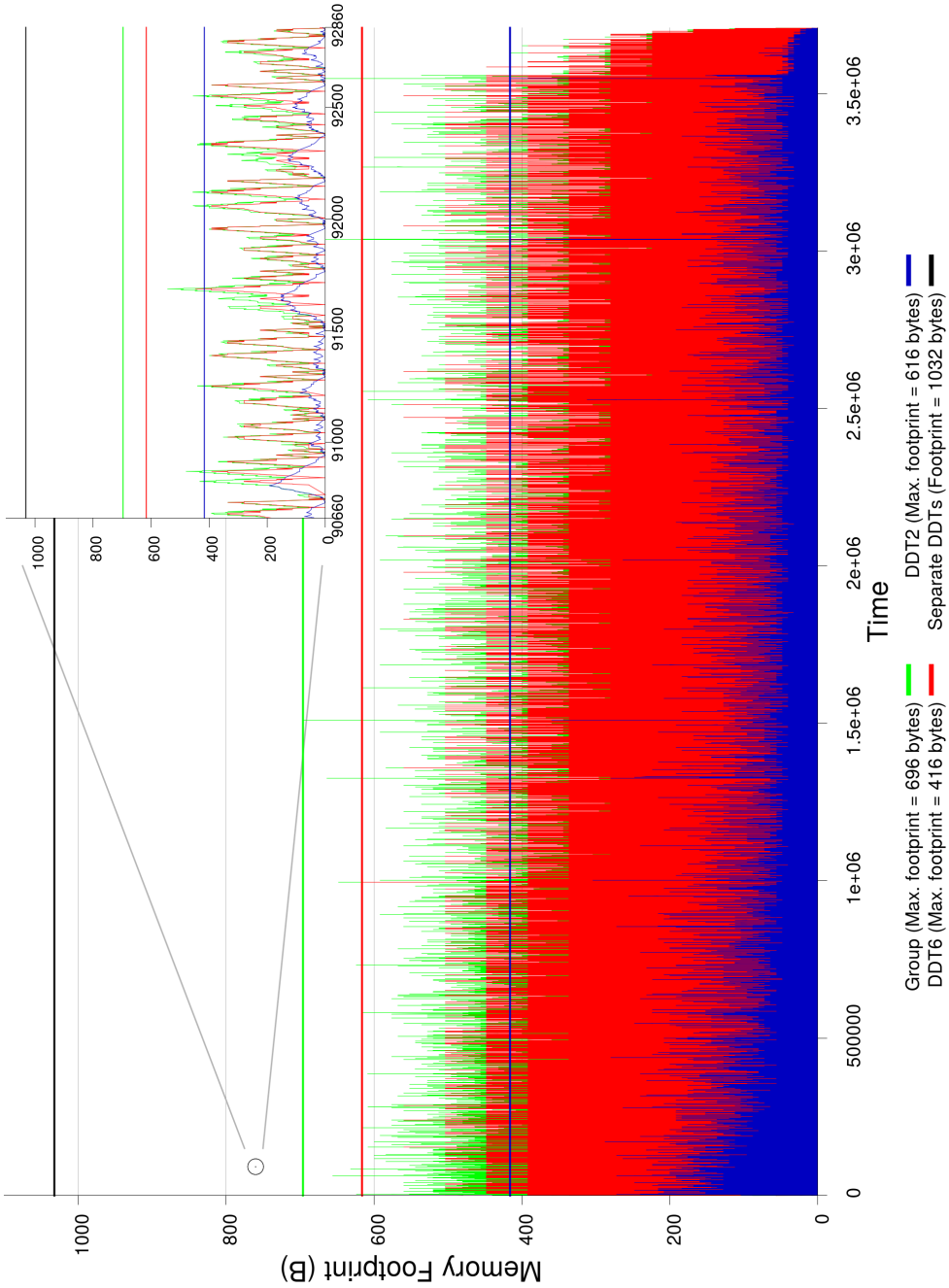


Figure 4.3.: Case study 1: Analysis of the footprint of two DDTs in the application (red and blue plots) and the group that holds both of them (green plot). The combined footprint of the group (green horizontal bar) is significantly lower than the added footprint that would be needed if the DDTs were assigned to independent pools (black horizontal bar). Inset: Zoom over a randomly-chosen area of 2200 (allocation-time) instants.

Table 4.3.: Case study 1: Performance of the solutions obtained with my methodology versus cache-based solutions. The entries in the first half of the table, which correspond to platforms with a Mobile SDRAM, are normalized using as reference platform 00. Correspondingly, the entries in the second part (platforms with an LPDDR2), are normalized taking as reference platform 17.

Platform	**Energy** mJ	**Energy** %	**Time** ($\times 10^6$) Cycles	**Time** %	**Page misses** %	**DRAM accesses** %	**Total accesses** %
(Mobile SDRAM)							
00. Only DRAM	360.01	100.0	2 132.5	100.0	100.0	100.0	100.0
01. Cache: L1=256KB(A16)	21.88	6.1	174.5	8.2	< 0.1	< 0.1	100.1
02. Cache: L1=256KB(A16,W4)	6.03	1.7	174.5	8.2	< 0.1	< 0.1	100.1
03. Cache: L1=16KB(A16), L2=256KB(A16)	51.47	14.3	267.6	12.5	< 0.1	< 0.1	300.5
04. Cache: L1=32KB(A16), L2=256KB(A16)	33.29	9.2	178.8	8.4	< 0.1	< 0.1	201.8
05. SRAM: 512B, 1KB, 32KB, 256KB	0.24	0.1	90.1	4.2	0	0	100.0
06. Cache: L1=64KB(A16)	15.94	4.4	88.3	4.1	< 0.1	0.1	100.2
07. Cache: L1=64KB(A16,W4)	2.94	0.8	88.8	4.2	0.1	0.1	100.2
08. SRAM: 512B, 1KB, 64KB	0.46	0.1	87.7	4.1	< 0.1	0.1	100.0
09. SRAM: 64KB	0.93	0.3	88.1	4.1	< 0.1	0.1	100.0
10. Cache: L1=32KB(A16)	106.71	29.6	434.3	20.4	9.7	35.0	168.7
11. Cache: L1=32KB(A16,W4)	36.08	10.0	275.4	12.9	11.5	9.0	118.0
12. SRAM: 512B, 1KB, 32KB	10.62	2.9	132.9	6.2	2.2	3.4	100.0
13. SRAM: 512B, 1KB, 16KB, 32KB	6.39	1.8	112.3	5.3	1.0	2.1	100.0
14. SRAM: 32KB	12.23	3.4	141.1	6.6	2.9	3.7	100.0
15. SRAM: LowerBound	0.14	< 0.1	87.1	4.1	0	0	100.0
16. Cache: LowerBound(D)	13.52	3.8	87.3	4.1	< 0.1	< 0.1	100.1
(LPDDR2-SDRAM)							
17. Only DRAM	229.57	100.0	1 315.6	100.0	100.0	100.0	100.0
18. Cache: L1=256KB(A16)	21.84	9.5	174.4	13.3	< 0.1	< 0.1	100.1
19. Cache: L1=256KB(A16,W4)	6.01	2.6	174.5	13.3	< 0.1	< 0.1	100.1
20. Cache: L1=16KB(A16), L2=256KB(A16)	54.64	23.8	283.1	21.5	< 0.1	< 0.1	318.0
21. Cache: L1=32KB(A16), L2=256KB(A16)	32.80	14.3	176.5	13.4	< 0.1	< 0.1	199.4
22. SRAM: 512B, 1KB, 32KB, 256KB	0.24	0.1	90.1	6.8	0	0	100.0
23. Cache: L1=64KB(A16)	16.21	7.1	90.0	6.8	0.5	0.6	101.1
24. Cache: L1=64KB(A16,W4)	2.89	1.3	89.1	6.8	0.4	0.1	100.2
25. SRAM: 512B, 1KB, 64KB	0.37	0.2	87.4	6.6	0.1	0.1	100.0
26. SRAM: 64KB	0.83	0.4	88.0	6.7	0.1	0.1	100.0
27. Cache: L1=32KB(A16)	49.30	21.5	258.5	19.6	26.9	34.3	167.3
28. Cache: L1=32KB(A16,W4)	34.85	15.2	323.6	24.6	47.6	15.8	129.7
29. SRAM: 512B, 1KB, 32KB	7.35	3.2	117.6	8.9	0.3	3.4	100.0
30. SRAM: 512B, 1KB, 16KB, 32KB	4.65	2.0	106.0	8.1	0.1	2.1	100.0
31. SRAM: 32KB	8.85	3.9	126.8	9.6	3.2	3.7	100.0
32. SRAM: LowerBound	0.14	0.1	87.1	6.6	0	0	100.0
33. Cache: LowerBound(D)	13.49	5.9	87.2	6.6	< 0.1	< 0.1	100.1

4.2.6. Simulation

The simulator evaluates the performance of the mapping solution for every platform using the memory trace obtained during profiling. Each simulation requires about 16 s (the simulation process is considerably faster than the analysis). As the DRAM modules are seldom accessed in most platforms, I assume that the memory controller can drive the chips into one of the power-saving modes and thus I configured the simulator to discard their active-idle energy consumption. Table 4.3 presents the results, normalized taking as reference the platforms with only DRAM modules (that is, without cache memories or SRAMs), which correspond in this experiment to platforms 00 and 17.

The experiments show that even for small sizes, cache memories improve significantly the performance of the system. However, the solutions obtained with my methodology achieve even bigger gains. To get a better measure of this improvement, Figures 4.4 and 4.5 show a direct comparison between both types of solutions for various memory subsystem sizes. Each of the figures shows three different comparisons, where the results of each group are normalized to the value of the first bar in the group (in blue).

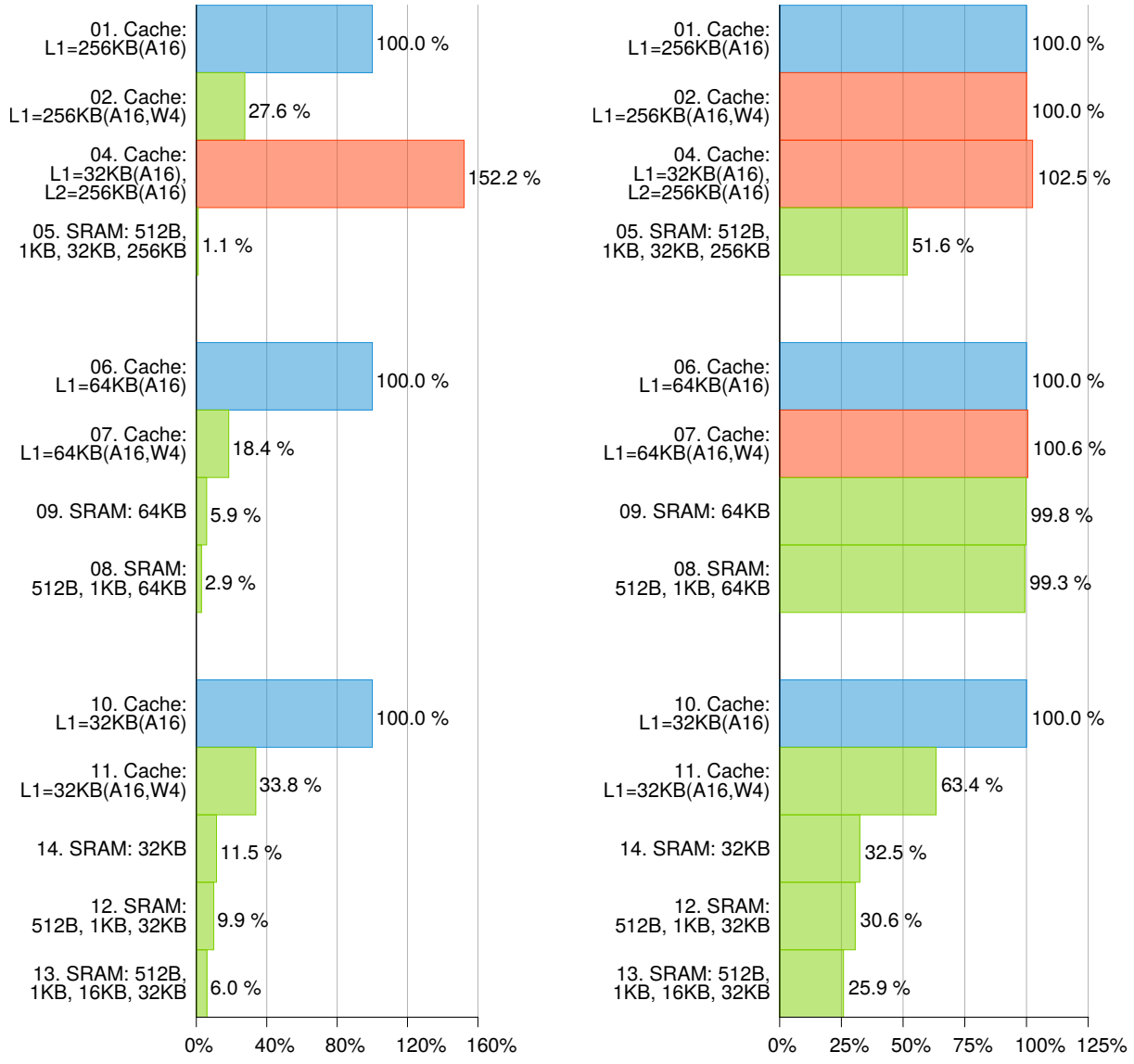
For instance, in Table 4.3 we saw that platform "06. Cache: L1=64KB(A16)" reduces the energy consumption in comparison with the platform that has only DRAM, platform 00, down to a 4.4 %. In Figure 4.4a, this is taken as the base case for the second group of bars. There, we see that modifying the length of the cache lines (platform "07. Cache: L1=64KB(A16,W4)") reduces energy consumption down to an 18.4 % – with respect to the reduction already achieved by platform 06. However, using instead an SRAM of the same size (platform "09. SRAM: 64KB"), energy consumption is reduced down to 5.9 % of the energy consumption with the 64 KB cache. Furthermore, this still represents a reduction in energy consumption of a 68.4 % from platform 07 (the one with shorter cache lines). Interestingly, platform "08. SRAM: 512B, 1KB, 64KB" shows that SRAMs can be seamlessly composed into hierarchies to achieve even greater improvements. Facing the designers, $\mathcal{DynAsT}$ generates automatically solutions to exploit them.

Similar trends are shown in Figure 4.5a for the platforms with LPDDR2 SDRAM. One interesting point is that the relative improvement of platform "28. Cache: L1=32KB(16,W4)" with respect to platform "27. Cache L1=32KB(A16)" is smaller than in the case of the platforms with Mobile SDRAM. This effect is likely due to the prefetch nature of LPDDR2, which eases the transfer of long cache lines while eliminating the difference between single and double word transfers.

Figures 4.4b and 4.5b show a different evolution for the number of cycles than for the energy consumption. These differences owe to latencies not scaling linearly with the size of the caches. For instance, I used a latency of one clock cycle for all the memories smaller than 128 KB, whether caches or SRAMs. Also, I assumed that SRAMs and caches of the same size have the same latency. Nevertheless, the general observation that platforms with SRAMs managed through $\mathcal{DynAsT}$ have better performance is kept in most cases.

4.3. Case study 2: Network routing

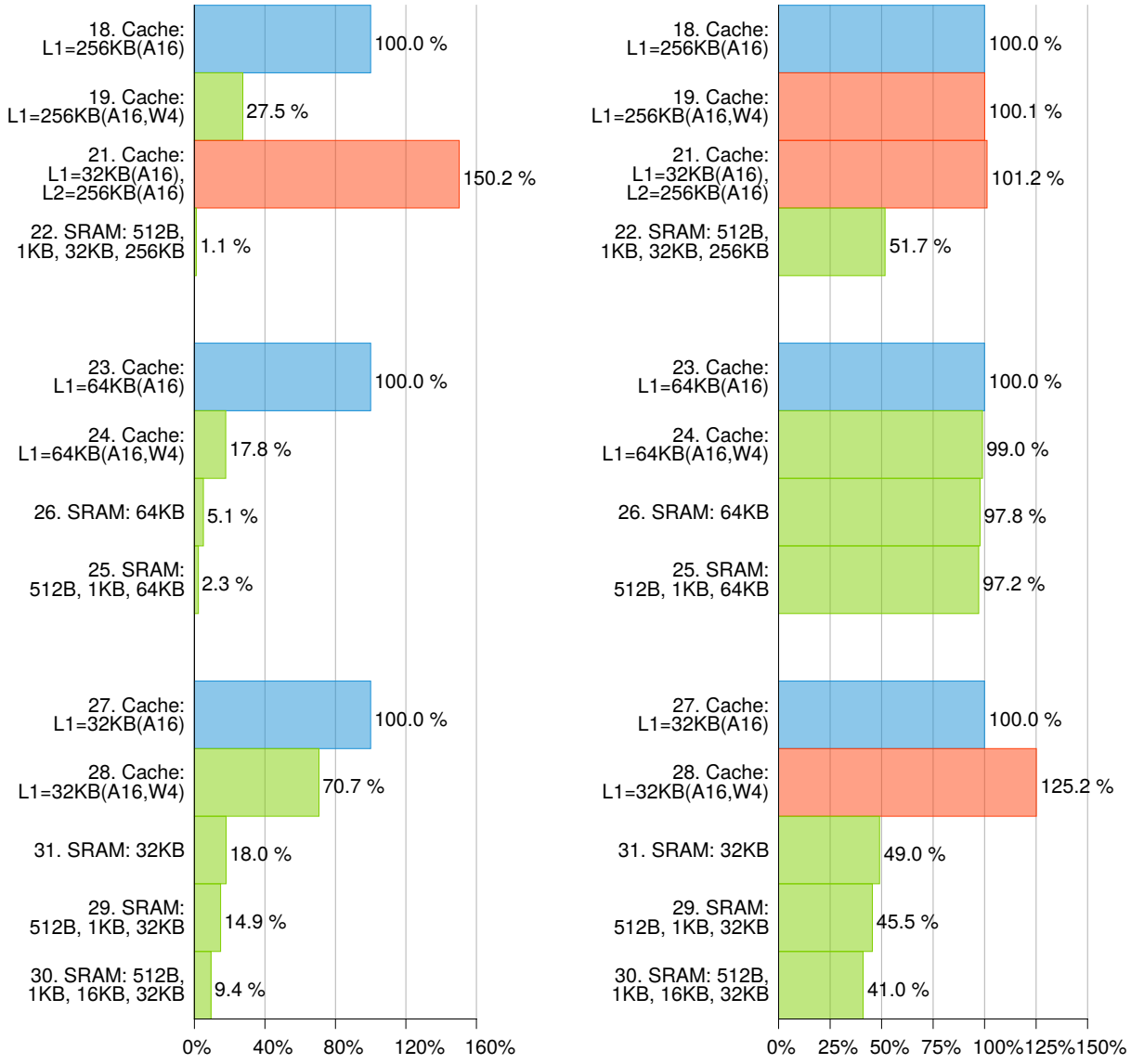
In this case study I explore the application of the methodology to a multithreaded implementation of the core algorithms in the network dispatcher of an operating system. The model is similar to the one presented in the case study of Chapter 6. Embedded systems may execute threads from several applications concurrently; the operating system has then to arbitrate be-



(a) Energy consumption

(b) Cycles memory subsystem

Figure 4.4.: Case study 1: Comparison of SRAM-based solutions with solutions based on caches of equivalent capacity for platforms with Mobile SDRAM. Each group of results is normalized to the uppermost bar (in blue), which represents the performance for a cache of a given size. Red bars mark configurations that have worst performance than their reference.



(a) Energy consumption

(b) Cycles memory subsystem

Figure 4.5.: Case study 1: Comparison of SRAM-based solutions with solutions based on caches of equivalent capacity for platforms with LPDDR2 SDRAM. Each group of results is normalized to the uppermost bar (in blue), which represents the performance for a cache of a given size. Red bars mark configurations that have worst performance than their reference.

tween the threads that need to send data through the network. As explained there, a common choice in these cases is the Deficit Round Robin (DRR) algorithm [SV96]. The operating system keeps a list of active destinations and, when a thread sends a packet towards one of them, it is stored in the corresponding queue. Packets are extracted from the queues in order and forwarded to the network adaptor, reducing the credit of the queue proportionally to the size of the packet. This mechanism enables the implementation of Quality-of-Service (QoS) mechanisms that can prioritize among applications and destinations, for instance, to guarantee a minimum bandwidth for certain connections while avoiding starvation in the rest.

The system was implemented as a multithreaded application. Therefore, several instances of the application DDTs are alive and accessed at the same time by different threads; moreover, memory accesses do not happen in a sequential manner. The main issue here is that it is not possible (at least in an easy way) to analyze the memory access behavior of each thread independently from each other and then, by combining these individual behaviors, "recreate" the behavior of the whole system, because that would leave out the interaction among the threads and, most importantly, the interleaved thread execution. Consider, for example, the case of two threads where each of them generates a stream of sequential accesses to memory: The memory subsystem would receive the accesses interleaved. Indeed, if the system has multiple processors or SMT (simultaneous multithreading) capabilities, the accesses will be interleaved every few words. If the involved DDTs have been placed on the same bank of a DRAM, these streams, that would otherwise execute efficiently, will generate a big number of different row activations.

The system implementation is organized in five modules with asynchronous queues between them:

Packet injection: A collection of real wireless network traces is used to generate the packets sent.

Packet formation: The TCP/IP header is added to the data supplied by the applications.

Encryption: Only for packets from applications that require encryption.

TCP Checksum computation.

Scheduling and quality-of-service management: The DRR algorithm classifies the packets in priority queues and schedules them according to available bandwidth.

In order to reproduce the conditions of a real system, we used a set of network traces collected from the wireless access points of the Dartmouth University campus [HKA04]. We identified traces from individual, yet anonymous, users and applications; some of them represent sessions lasting a few minutes while others represent sessions of up to 24 hours. This allows reproducing part of the original system use cases, with the exception of accurate packet rate control.

Table 4.4 presents the results obtained after applying my methodology, in comparison with the ones obtained with cache memories of similar sizes. Each trace was fed to the system and executed on all the different platforms. In total, 32 different platform configurations were evaluated with 14 traces. The results for every platform with every input are normalized against the results of the reference platform with the same input, and the normalized values are then averaged for each platform. Thus, if a platform has a 30 % figure for energy consumption with respect to the reference platform, that platform consumes a 30 % of the energy consumed by the reference platform across all the input traces. Appendix E contains tables with the results of all the experiments, unaggregated.

Table 4.4.: Case study 2: Performance of the solutions obtained with my methodology versus cache-based solutions. Average normalized improvements with sample standard deviations (σ_{n-1}, sample size $N = 14$). All figures are percentages. The entries in the first half of the table, which correspond to platforms with a Mobile SDRAM, are normalized using as reference platform 00. The entries in the second part (platforms with an LPDDR2), are normalized taking as reference platform 14.

Platform	Energy	σ_{n-1}	Time	σ_{n-1}	Page misses	σ_{n-1}	DRAM accesses	Total accesses
(Mobile SDRAM)								
00. Only DRAM	100.0	0.0	100.0	0.0	100.0	0.0	100.0	100.0
01. Cache: L1=256KB(A16)	46.6	30.3	39.5	26.1	7.2	3.5	43.2	184.1
02. Cache: L1=256KB(A16,W4)	38.9	25.6	40.9	23.5	16.5	7.0	41.3	173.1
03. Cache: L1=512KB(A16)	42.8	37.1	34.4	29.6	4.3	4.8	33.5	164.9
04. Cache: L1=4MB(A16)	107.0	66.5	54.7	36.2	3.1	4.3	28.6	155.5
05. Cache: L1=16KB(A16), L2=256KB(A16)	51.4	34.4	41.2	29.7	7.2	3.5	43.2	292.4
06. Cache: L1=32KB(A2), L2=256KB(A16)	48.6	33.4	41.3	29.7	7.2	3.6	43.2	293.5
07. SRAM: 512B, 1KB, 32KB, 256KB	25.0	25.8	22.7	19.6	0.8	0.9	31.4	100.0
08. SRAM: 256KB	25.9	25.6	25.8	18.8	0.9	0.9	32.0	100.0
09. SRAM: 4MB	15.3	14.3	36.5	12.9	0.3	0.4	12.6	100.0
10. SRAM: 8x512KB	11.3	14.9	17.6	11.8	0.3	0.4	12.6	100.0
11. SRAM: 512B, 1KB, 32KB, 256KB, 8x512KB	10.1	13.9	14.2	11.9	0.3	0.4	11.7	100.0
12. SRAM: LowerBound	< 0.1	< 0.1	5.7	1.7	0	0	0	100.0
13. Cache: LowerBound(D)	6.9	3.2	7.0	2.5	< 0.1	< 0.1	2.7	105.3
(LPDDR2-SDRAM)								
14. Only DRAM	100.0	0.0	100.0	0.0	100.0	0.0	100.0	100.0
15. Cache: L1=256KB(A16)	47.9	36.4	34.9	24.6	5.2	2.8	43.2	184.1
16. Cache: L1=256KB(A16,W4)	57.4	50.8	56.9	44.5	10.1	5.1	41.3	173.2
17. Cache: L1=512KB(A16)	50.8	44.3	32.0	26.4	2.5	3.0	33.5	165.0
18. Cache: L1=4MB(A16)	187.6	139.2	64.0	42.5	1.4	2.0	28.6	155.5
19. Cache: L1=16KB(A16), L2=256KB(A16)	58.5	47.3	38.2	30.5	5.2	2.8	43.2	292.4
20. Cache: L1=32KB(A2), L2=256KB(A16)	53.0	44.0	38.2	30.5	5.2	2.8	43.2	293.6
21. SRAM: 512B, 1KB, 32KB, 256KB	12.7	13.4	12.9	8.1	0.9	0.9	31.4	100.0
22. SRAM: 256KB	13.7	13.4	16.7	7.9	0.9	0.9	32.0	100.0
23. SRAM: 4MB	15.6	10.3	43.7	15.3	0.3	0.4	12.6	100.0
24. SRAM: 8x512KB	7.7	9.5	17.0	8.4	0.3	0.4	12.6	100.0
25. SRAM: 512B, 1KB, 32KB, 256KB, 8x512KB	6.6	8.9	13.2	8.7	0.2	0.4	11.7	100.0
26. SRAM: LowerBound	0.1	< 0.1	8.4	4.1	0	0	0	100.0
27. Cache: LowerBound(D)	10.9	4.5	9.1	4.2	< 0.1	< 0.1	2.7	105.3

The sample standard deviation in the table shows important fluctuations from the average values for almost every platform because of the different nature, transmitted data length and duration of the inputs. However, the solutions generated with my methodology improve always on the results obtained with caches. Interestingly, the standard deviation of $\mathcal{DynAsT}$'s solutions is usually smaller than that of cache solutions, suggesting a more uniform system performance. Nevertheless, this high variation hints that this application is a good candidate for a mechanism to cope with variability such the system scenarios presented in Section 4.5.3.

In this experiment, footprint varies approximately from 242 KB to 8.7 MB, with 13 out of the 14 cases over 512 KB. This situation attests that the solutions produced by my methodology have good performance also when the application footprint exceeds the size of the available on-chip SRAMs. The reason for this good performance is that the data placement puts in the DRAM only instances of the least accessed DDTs and, in this case, those instances (the body of the network packets) are accessed mostly sequentially. Temporal locality is low because

Table 4.5.: Case study 2: Detailed comparison between SRAM and cache-based solutions. The execution cost of $DynAsT$'s solutions is normalized for each input against the cost of a solution with a cache memory of equivalent size. These results correspond to 4 of the 14 inputs used in the experiments, not aggregated. All numbers are percentages.

Platform	(Mobile SDRAM) Energy	Time	DRAM accesses	Page misses	(LPDDR2-SDRAM) Energy	Time	DRAM accesses	Page misses
INPUT 1								
Cache: L1=256KB(A16)	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
Cache: L1=64KB(A4)	98.9	92.9	123.8	130.2	79.6	80.4	123.9	141.1
SRAM: 64KB	52.4	56.7	79.5	30.6	45.8	61.1	79.6	43.3
SRAM: 256KB	24.3	53.2	32.5	2.5	16.9	61.4	32.6	3.1
SRAM: 512B, 1KB, 32KB, 256KB	19.7	35.5	28.7	1.9	11.4	37.5	28.7	2.2
INPUT 2								
Cache: L1=256KB(A16)	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
Cache: L1=64KB(A4)	94.1	93.5	104.8	131.1	80.7	83.1	104.8	134.7
SRAM: 64KB	72.3	68.8	95.9	40.1	38.7	41.8	95.9	71.9
SRAM: 256KB	68.3	68.2	90.7	14.5	33.2	41.4	90.7	20.1
SRAM: 512B, 1KB, 32KB, 256KB	67.6	65.0	90.2	14.4	32.6	36.1	90.2	20.0
INPUT 3								
Cache: L1=256KB(A16)	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
Cache: L1=64KB(A4)	86.5	83.0	103.9	102.9	71.7	72.9	103.9	102.3
SRAM: 64KB	46.8	50.9	69.5	20.2	42.6	56.7	69.5	29.7
SRAM: 256KB	22.2	49.1	27.8	5.2	17.7	59.3	27.8	7.7
SRAM: 512B, 1KB, 32KB, 256KB	17.8	32.8	24.2	4.7	12.6	37.1	24.2	6.9
INPUT 4								
Cache: L1=256KB(A16)	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
Cache: L1=64KB(A4)	92.4	91.9	102.8	122.4	79.4	81.5	102.8	119.9
SRAM: 64KB	73.2	69.5	96.8	38.6	37.8	40.3	96.8	78.5
SRAM: 256KB	70.3	69.6	92.8	20.3	34.2	41.3	92.8	34.6
SRAM: 512B, 1KB, 32KB, 256KB	69.7	66.8	92.5	20.2	33.1	36.1	92.5	34.4

each instance is accessed just twice, first to calculate the CRC and then to forward it to the buffers of the network adaptor. Cache memories cannot amortize the cost of data movements and even risk evicting more useful data. An alternative in platforms based on caches could be mapping these DDTs in a non-cacheable area, deactivating caching for all the packet bodies.

In contrast, as $DynAsT$'s solutions can split the pools over different memory resources, some of the packet-body instances are still placed on the on-chip SRAMs, but that no instances of other more accessed DDTs are evicted is still ensured.[1] If the application has a memory allocation pattern that alternates peaks with periods of lower consumption then, during a potentially significant fraction of the execution time, all the instances of all the DDTs may reside in the on-chip memories without conflicts (that is, if the footprint of the packet bodies is small enough to fit in the part of their pool mapped on the closer memories). As the number of packet bodies increases, some of them are allocated in the DRAM, but accessing them does not evict more accessed data from other pools.

In a different consideration, the observed increase in energy consumption related to the

[1] It is possible to split the pool of packet bodies in two areas, one cacheable and the other non-cacheable, allocating space from the first one as long as possible. However, how big should the cacheable pool be? The answer to this question would probably require an analysis similar to the one proposed in this text!

bigger cache memories is due to the fact that bigger caches consume more energy per access, and the smaller ones are already capable of the most significant reduction of accesses to the DRAM. This effect is less clear in the number of cycles required for execution because the cache latencies used in the experiments do not increase linearly with their size. Moreover, cache hierarchies perform quite badly in this experiment. Consider for example the case of platforms 01 and 05. Despite having more capacity, the second platform has considerably higher energy consumption. This is due to the continuous transfers of data with low locality between the small L1 and the L2. As a result, the total number of memory accesses, that is, the addition of accesses to all the memory modules, including transfers between levels in the cache hierarchy, is much higher in the second platform. Compare these results with the ones obtained for platform 08, which has an SRAM of the same size than the cache of platform 01.

An interesting observation is that the energy consumption values shown in Table 4.4 are compared independently for the platforms with Mobile SDRAM and LPDDR2. However, a direct comparison between them exposes a net reduction of a 39.8 % on average for all platforms (s = 26.7 %) when using an LPDDR2 instead of the older technology (specifically, 71.0 % with s = 8.3 % when comparing platforms 21 and 7, excluding the experiments that fit entirely in the on-chip memories). This highlights the important effort invested by the industry in reducing the energy consumption of the memories designed for embedded systems. More recent technologies such as LPDDR3 and LPDDR4 (introduced in 2014) should be able to reduce even further the cost of DRAM accesses.

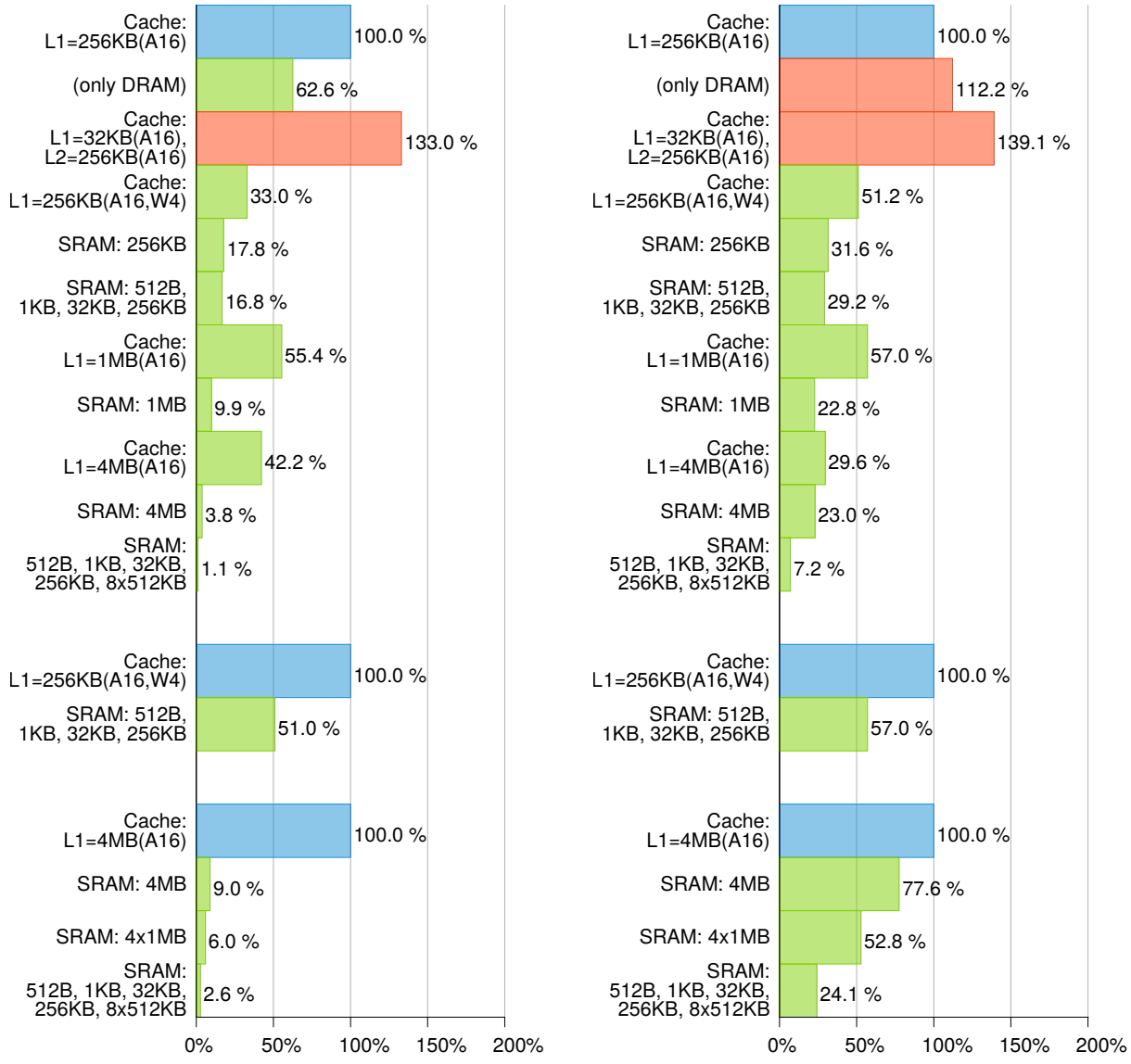
Finally, Table 4.5 compares non-aggregated data for several input cases to compensate for the big standard deviation in the aggregated results of Table 4.4. Here, I compare the performance of $\mathcal{DynAsT}$'s solutions for several SRAM-based platforms with standard configurations that include caches of equivalent sizes.

4.4. Case study 3: Synthetic benchmark – Dictionary

The goal of this last benchmark is to show that DM-intensive applications can limit the effectiveness of cache memories because of their low spatial and temporal localities. The benchmark uses a trie to create an ordered dictionary of English words, and then simulates multiple user look-up operations. The trie DDT [Fre60] belongs to the category of ordered trees and is useful to store any type of information that can be organized using prefixes, especially if it presents a high degree of redundancy, such as words of a dictionary, compression tables and DNA sequences. In this benchmark each node has a list of children indexed by letters.

This experiment models a case that is particularly hostile to cache memories because each traversal accesses a single word on each level, the pointer to the next child, but the cache has to move whole lines after every conflict. This is a well known side effect of the use of dynamically-linked data structures on cache architectures, including desktop and server computers, and is thus an area of intense research.

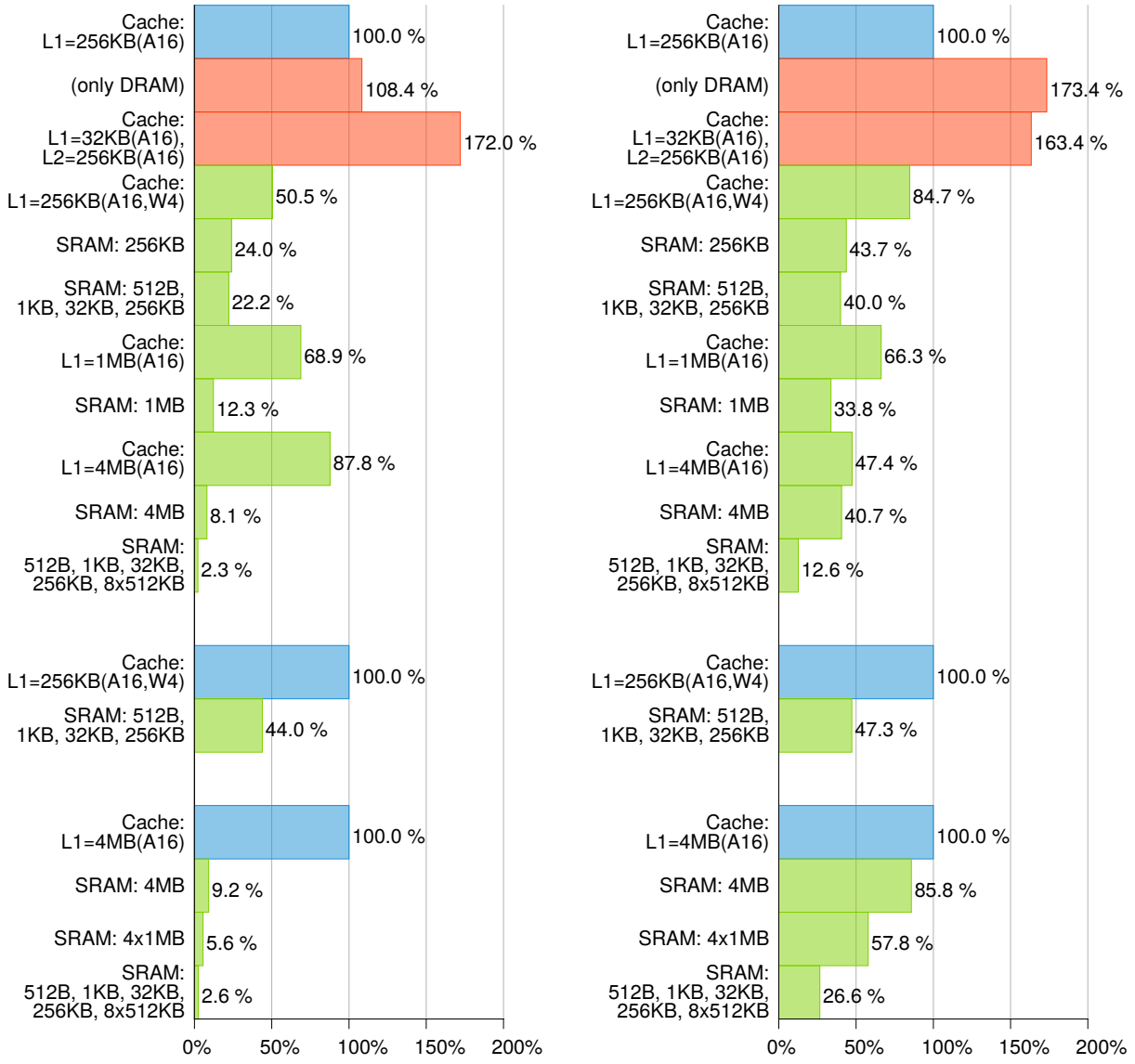
Figures 4.6 and 4.7 show the improvements attained by $\mathcal{DynAsT}$ in comparison with cache memories. As in previous cases, each of the figures shows three different comparisons, where the results of each group are normalized to the value of the first bar in the group (in blue). In both figures, the first group of bars compares the performance of caches and SRAMs of increasing sizes to that of a 256 KB cache. A direct comparison between platforms "SRAM: 512B, 1KB, 32KB, 256KB" and "Cache: L1=256KB(A16)" shows that $\mathcal{DynAsT}$'s solution achieves a relative improvement of 83.2 % for the LPSDRAM case and 77.8 % for the LPDDR2 case in



(a) Energy consumption

(b) Cycles memory subsystem

Figure 4.6.: Case study 3: Comparison of SRAM-based solutions with solutions based on caches of equivalent capacity for platforms with Mobile SDRAM. Each group of results is normalized to the uppermost bar (in blue), which represents the performance for a cache of a given size. Red bars mark configurations that have worst performance than their reference.



(a) Energy consumption

(b) Cycles memory subsystem

Figure 4.7.: Case study 3: Comparison of SRAM-based solutions with solutions based on caches of equivalent capacity for platforms with LPDDR2 SDRAM. Each group of results is normalized to the uppermost bar (in blue), which represents the performance for a cache of a given size. Red bars mark configurations that have worst performance than their reference.

energy consumption, 70.8 % and 60 % when considering the cycles spent accessing the memories. As discussed in the previous case study, cache hierarchies (platform "Cache: L1=32KB(A16), L2=256KB(A16)") may perform poorly with this type of applications because the small size of the caches in the closer levels forces many evictions of whole lines, and the transfers between cache levels accumulate to the accesses to the DRAM themselves.

One of the most interesting results shown in these figures is obtained reducing the size of the cache lines. The base case is the configuration offered by many processors such as the ARM Cortex-A15, which has 64-byte cache lines. Although the trend for larger line sizes may be beneficial for applications that process streams of data (because it increases the length of prefetching), for applications that rely heavily on the use of DDTs the effect is quite different: Every line fill and write-back is more costly because more words are moved between memories even if the application accesses just one or two of them, and the longer lines reduce the number of different cache lines (i.e., different memory positions) that can be stored with the same cache size. My experiments show that using a line size of just 16 B (4 words) can improve, for this type of applications, energy consumption by 67 % with LPSDRAM and 49.5 % with LPDDR2, and the cycles spent in the memory subsystem by 48.8 % and 15.3 %, respectively (platform "Cache: L1=256KB(A16,W4)" versus "Cache: L1=256KB(A16)"). Nevertheless, my approach with explicitly addressable memories is even better suited for this type of applications, still improving energy consumption over the 16-byte-lines cache memory by 49 % and 56 %, and the cycles used by 43 % and 52.7 %, respectively (platform "SRAM: 512B, 1KB, 32KB, 256KB" versus "Cache: L1=256KB(A16,W4)," second block of bars in the figures).

Finally, the performance of $\mathcal{DynAsT}$'s solutions also scales better with the size of the memories. The third group of bars in the figures shows this effect. Moreover, this last group of bars shows how multiple on-chip SRAMs of a smaller size can be combined to further improve performance, in contrast with the difficulties encountered while trying to harness cache hierarchies.

4.5. Additional discussion

4.5.1. Suitability and current limitations

The methodology that I have presented in this text is best suited in its current form for applications that use DDTs in phases or traverse data structures accessing only a small amount of data at each node, and whose DDTs have very different FPBs. These cases may hinder the performance of hardware caches as they have to move more data around the memory hierarchy than is really needed, possibly evicting very accessed objects with seldom accessed ones. As an example, the traversal of a structure could force a cache to move complete nodes back and forth, with an increasing waste of energy as the size of the cache lines increases. Splitting of data structures may allow packing the node pointers tightly and accessing only the data of the nodes that are actually needed during the traversal. This effect is particularly beneficial with my methodology because the pool containing the pointers is guaranteed to be always in the correct memory module, independently of which other data accesses are performed by the application.

On the contrary, my methodology may not be adequate for applications that keep instances of many DDTs alive simultaneously and alternate between phases that access each of them, particularly if each phase creates periods of high access locality. The reason is that the objects do not free their space and thus, grouping cannot reuse it. In comparison, cache memories are

specifically designed for this situation: They use data movements to recycle storage and keep the most currently accessed data closer to the processor. For these situations, cache memories have proven themselves useful during decades.

One exception to the previous consideration arises when the application has low spatial locality because my methodology avoids moving data words that will not be reused. Even if accessing the objects of the group in the worst resource incurs a high cost, saving fruitless data movements may produce important energy savings.

My methodology assumes that most instances of a given DDT have a similar FPB. Therefore, it may not be adequate for applications whose DDTs have instances with very different FPB.

The methodology leaves the least accessed DDTs in the most distant memory modules; this is appropriate for data streams that are processed sequentially and without reuse, or for small data elements that are seldom accessed. However, for other access patterns to big arrays, it may be convenient to supplement my approach with software techniques such as array tiling or blocking on a small dedicated SRAM.

As a final consideration, the performance of the solutions generated with my methodology degenerates in the worst case to the performance of a platform with only DRAM memories (or the most inefficient technology used in the memory subsystem, if other). A simple justification is that accesses happen either to an SRAM or to the DRAM; any access to an SRAM improves performance. As the total number of accesses does never increase, the total cost is bounded by the cost of executing all the accesses on the DRAM – small variations might nonetheless happen because of specific data layouts that produce slightly different numbers of DRAM row-misses, even if the number of accesses to the DRAM is reduced with my methodology. In contrast, cache hierarchies may introduce higher costs due to excessive data movements, as demonstrated in the previous case studies.

4.5.2. Use cases for the methodology

My methodology can be used in two different ways, depending on whether the hardware platform is fixed or not:

4.5.2.1. Application optimization for a fixed platform.

When the hardware platform is fixed, the methodology can be used to produce the placement of the dynamic data objects of the application into the available memory resources and improve energy consumption and performance. It may also help to increase the duration of the periods that the external DRAMs spend on low-power modes.

4.5.2.2. Hardware platform exploration and evaluation.

Since $\mathcal{DynAsT}$ includes a complete memory organization simulator, it can be used by the designer to explore application performance on different platforms and choose the most suitable from the available options. Alternatively, if the design is going to be implemented on an ASIC (or to a lesser extent on an FPGA), the designer may have complete control to adapt the platform exactly to the characteristics of the application, maybe including multiple small memories.

An interesting possibility in this regard is integrating as many memory modules as groups defined by the tool, with the size of each one close to the size of the corresponding group. The size and FPB of the groups may help to decide the size of the memories. For example,

if $\mathcal{DynAsT}$ creates a very small group with a high FPB, the designer may instantiate a small SRAM macro to contain it, possibly creating an important reduction in energy consumption. Bigger groups with similar FPBs may be included in a single SRAM, or the designer may instantiate separate ones, depending perhaps on the possibility of shutting down some of them during specific phases of the application.

Interestingly, increasing the number of elements in the memory subsystem may introduce some overheads due to address decoding in the bus (although probably offset by the lower cost of the memory decoders), but it does not introduce any of the complexity issues that arise with cache hierarchies. This is especially true for "miss-hit" chains in cache hierarchies, which are completely absent in the solutions produced with my methodology.

Other design aspects that can be explored are the advantages of DRAMs with more banks, or using more DRAM modules instead of a bigger single one to increase the number of banks that can be active at the same time. Or the possibility of introducing additional SRAM capacity in the design with the advantage of reducing DRAM size or even completely removing it. For example, if an application uses exactly 257 MB of memory, the designer may evaluate including a 1 MB SRAM in the ASIC instead of using a bigger DRAM. While such a decision may be out of question with caches (the system will still require the 257 MB of DRAM unless the cache is exclusive respect it), $\mathcal{DynAsT}$ enables it and even takes care automatically of all the data placement issues transparently.

4.5.3. Scenario based system design

Many embedded systems are subject to changing working conditions and those designed with the data placement optimization techniques that I present in this text are no exception. One technique to cope with this issue is *system-scenario based* design [GPH+09]. Therefore, I propose using it to identify the different run-time situations that may arise, generate the required solutions and activate them at run-time.

Embedded systems (and almost any computing system for that case) may be optimized at design time or during run-time. However, none of these options are perfect. If the system is fully configured at design time, it will have to be prepared for the worst possible case. That means that most of the time the system will be running with a suboptimal configuration for the actual run-time situation, which may be more benign. Alternatively, the system may use a run-time mechanism to configure itself for the current conditions. The problem is that the configuration process can be complex and slow, demanding many resources itself. Therefore, either the system employs suboptimal (but quick) algorithms or the cost of run-time configuration may be even higher than the cost of a worst-case solution optimized at design time.

Scenario-based system design taps into both paradigms by clustering together run-time situations that can be executed with the same system configuration without incurring a high resource consumption or deadline penalty. The run-time situations that belong to a system scenario are executed using the configuration required for the worst of them, but different scenarios define their own system configuration. Each of the individual system scenarios can be thoroughly optimized at design time. At run-time, the system only needs to identify the current situation, the corresponding scenario, decide if the switch is worthy and, if so, transition itself into the new configuration. Unforeseen run-time situations can be tackled with a backup scenario that uses a worst-case configuration or, if possible, a lightweight dynamic optimization process. The final goal of the system scenarios approach is to enjoy complex situation-specific optimizations without the unbounded cost of storing a huge number of con-

figurations or executing complex optimization algorithms at run-time.

System scenarios are often confused with use-case scenarios, but the distinction between them is relevant. System scenarios:

> "[...] group system behaviors that are similar from a multidimensional cost perspective – such as resource requirements, delay and energy consumption – in such a way that the system can be configured to exploit this cost similarity. At design-time, these scenarios are individually optimized. Mechanisms for predicting the current scenario at run-time, and for switching between scenarios, are also derived." (p. 1)[2]

Whereas use-case scenarios:

> "[...] focus on the application functional and timing behaviors and on its interaction with the users and environment, and not on the resources required by a system to meet its constraints." (p. 2)

Usually, one use case will generate one or more system scenarios. For instance, the system may configure itself differently for the same use case when running on batteries than when connected to the mains. Also, although less obvious, several use cases may be executed using the same system scenario if their resource demands are similar. The important concept is that system scenarios are sets of run-time circumstances that can be tackled with a similar configuration of system resources:

> "[System scenarios] are derived from the combination of the behavior of the application and the application mapping on the system platform. These scenarios are used to reduce the system cost by exploiting information about what can happen at run-time to make better design decisions at design-time, and to exploit the time-varying behavior at run-time." (pp. 2–3)

The design process with system scenarios has several phases (full details are presented in [GPH+09]). The main one is identifying the different run-time situations using the observable parameters in the platform: A constant value for the set of parameters during a period of time is used to define the run-time situations. Then, in order to reduce the number of different cases to consider, similar run-time situations are clustered into scenarios. If the cost of scenario switching is high, run-time situations that alternate frequently may also be clustered in the same scenario:

> "The number of distinguishable RTSs [Run-Time Situations] from a system is exponential in the number of observable parameters. Therefore, to avoid the complexity of handling all of them at run-time, several RTSs are clustered into a single *system scenario*. A trade-off is present here between optimisation quality and run-time overhead of the scenario exploitation." (p. 9)

In summary, the design process with system scenarios consists of design time and run-time phases. The design time phase includes the definition of the observable parameters of the platform, identification of the possible run-time situations, clustering of those situations into

[2] All the quotations in Section 4.5.3 have been extracted from [GPH+09].

system scenarios with similar resource requirements and optimization of the system configuration for each scenario (potentially including different code generation). During the run-time phase, a special component of the system monitors the current situation and identifies the corresponding scenario. The system has to execute a scenario switch if the conditions demand it. Important additional considerations are the overhead of scenario switching and the possibility of continuous run-time calibration of the scenario configuration beyond the prescriptions set at design time.

The integration of the placement optimizations for dynamic data presented here and system scenarios has moderate complexity. A mapping solution has to be produced for every system scenario identified because the dynamic data types of the application may be used differently during each scenario, or platform resources may have different availability. As each solution has a moderate size, this should not constitute a big overhead on the total system cost. In any case, the designer will have to find a balance between a placement finely tuned for every possible situation and the overhead of storing and managing multiple scenarios.

One point that may require additional research is the switching between scenarios if different dynamic memory managers are employed: While it may be relatively easy to migrate an entire pool from a memory resource to another in systems with an MMU (Memory Management Unit), changing the internal algorithms and structure of the pools may be more difficult. The new DMM would need to be able to manage the memory blocks allocated by the old one as they are freed. Additionally, the movement of entire pools without the use of an MMU can also be quite costly; a mechanism to update the pointers in the application may be needed.[3] Therefore, scenario switching may be constrained to certain moments in the execution or require the collaboration of the applications, depending on the platform resources and specific techniques available.

4.5.4. Simulation versus actual execution

Although the design of $\mathcal{DynAsT}$'s simulator has received as much care and attention as seemed reasonable, multiple factors may affect the accuracy of the results obtained with it. Thus, it is important to keep in mind that simulation is just a convenient way to assess the properties of the placement solutions generated with the methodology, or to evaluate multiple platform variations quickly.

Additional experimentation on real hardware platforms would be required to validate the improvements that the experiments presented in this chapter promise. Nevertheless, and subject to further tests, I humbly believe that the significant improvement margins obtained for most cases support the plausibility of this work.

4.5.5. Reducing static data placement to dynamic data placement – or vice versa

The techniques for dynamic data placement presented in this text can be applied also to produce the placement of static data. The implementation would be technically easy: Each global variable or object declaration is simply replaced by a call to `malloc()` or the `new` operator – a function for allocating all these objects at the beginning of the execution may also be necessary. With this simple change, $\mathcal{DynAsT}$ will analyze the properties of each variable as

[3] Interesting options are the algorithms employed by garbage collectors to update application references after relocating heaps or the compiler-based techniques presented in [LA05].

a DDT with a single instance (i.e., a "singleton") and consider its properties during placement. Thus, static global objects will be allocated at run-time in SRAM or DRAM resources according to their access characteristics together with the other objects of the application. Moreover, further improvements in the methodology and $\mathcal{DynAsT}$ may also help to separate global data objects into different DRAM banks without explicit attention.

The designer may exclude explicitly these pseudo-DDTs from grouping to ensure that they are not mixed in pools with other objects, but this is probably unnecessary because they will be allocated at the beginning of the execution and never destroyed.

An important advantage that can be obtained by managing global static data objects through $\mathcal{DynAsT}$ is adaptability to platform changes. For example, if the application is executed in a platform with bigger SRAMs, static data objects will also benefit from them according to their characteristics. Similarly, in platforms with smaller resources the complete data placement solution will be automatically reevaluated for all the data objects. More importantly, the system will be able to run even if the SRAM in which a global data object would have been mapped fails.

Stacks may also be considered as dynamic arrays and placed accordingly (depending on the concrete processor architecture requirements), but other existing approaches that include specific hardware support [GCPT10] seem more adequate. Special requirements such as stack management during interruptions may prevent the applicability of $\mathcal{DynAsT}$ to these data objects.

In the other direction, it is also possible to use existing techniques for placement of static data objects instead of the mapping phase in my methodology. As explained at the beginning of this text, the problem lies not so much on placing the pools themselves on memory resources, but on classifying the dynamic data objects of the application into different pools: Creating a big pool for all the objects would blindly mix very and seldom accessed objects, hence making differentiation impossible. With hardware caches, this solution would correspond to the model traditionally used in desktop computers. Software caching techniques would not be very efficient due to the impossibility of identifying individual objects in the pool address space and the lack of locality.

However, after the DDTs are classified with a mechanism such as the grouping step in my methodology, it would be possible to place each complete pool into memory resources as if it were a static array. Software caching techniques might also be applied to specific pools that contain dynamic data objects with particular characteristics.

In a sense, both problems are complementary with plausible reductions in both directions, although these reductions may introduce some overheads.

4.5.6. Order between mapping and pool formation

In my methodology as it currently stands, pool formation, that is, the design of the dynamic memory manager for each pool, is performed at design time before the mapping step. The reason is that this step is usually complex, frequently requiring some exploration of the design space. One of the main goals of my methodology is that the mapping step can be moved to run-time so that the application can be adapted to the concrete resources available in the platform at the time of execution, maybe with variations between successive executions. Thus, any run-time parts of the methodology executed at run-time must be very efficient. This decision may be reconsidered if efficient techniques for the design of dynamic memory managers are found.

Traditionally, DMMs have been designed taking into account the number and size of the

allocated blocks and the allocation and deallocation patterns of the application. However, it may be possible to improve the performance of the DMM itself by taking into consideration the characteristics of the memories where the pools are placed. For example, a DMM for a pool located in an SRAM may employ aggressive coalescing and splitting mechanisms to reduce fragmentation at the expense of some more – relatively cheap – memory accesses. In contrast, a DMM for a pool located in a big DRAM may relax anti-fragmentation measures to restrain the number of costly random memory accesses.

To exploit this information, pool formation must be executed after the pools are placed on memory resources. Therefore, either the mapping step is always executed at design time, or pool formation is delayed until run-time. The alternative, if no efficient techniques for DMM design are available, may be to generate at design time multiple DMM descriptions, one for each type of memory technology that deserves special attention. Then, the mapping step could be executed at design or run-time as convenient. Assuming that this does not constitute a big burden for deployment, it may be an interesting option for future research. Nevertheless, the benefits of this order change need still to be assessed.

Chapter 5

Characterization of application dynamic memory behavior via software metadata

Embedded systems are normally subject to more restrained operating conditions than general purpose systems and their behavior often relies on the nature of the input data samples, operating environment and user behavior. The trick to avoid relying on worst case – usually static – solutions is to exploit as much static (design time) knowledge on the applications as possible, but still leaving enough freedom for run-time considerations to tackle with dynamic variations.

One technique that can be used for this purpose is system scenarios (discussed in Section 4.5.3), which group different working conditions with similar characteristics so that the system and its applications can be analyzed and optimized for each of the scenarios. This type of analysis, or similar ones, requires extensive information about the static and dynamic characteristics of the applications. However, (when this research was conducted) there is not a standard definition or representation to typify the characteristics of the dynamic data access behavior of applications due to varying inputs.

In Chapters 5 and 6, I propose a uniform representation of the dynamic data access and allocation behavior of the applications: Software metadata. The purpose of this representation is to enable optimization, rather than to analyze the structure of the applications. I also describe appropriate techniques to obtain this information from the original applications.

The concept of software metadata is a generalization born from previous experience, mine and from other people, in areas such as dynamic memory management, dynamic data types, memory access scheduling and dynamic data placement optimization for embedded systems. In these two chapters I explain how that previous experience was exploited to design a work flow that reduces the total effort required to apply several optimizations to an application. This presentation is limited to techniques with which we were already familiar, but surely the method can be extended with new ones, adapting the data model to their needs, trying to identify common requirements and minimizing the amount of specific work. The work method presented in this chapter makes sense only when the intention is to apply multiple optimization techniques to an application. Otherwise, limiting the effort to the strictly needed for the planned optimization would result more practical.

Given the extension of the previous chapters on placement of dynamic data objects, which is the main subject of this text, I opted for limiting their complexity and avoiding more concepts than those strictly necessary (profiling, analysis) for the development of the methodology and its implementation as a tool. However, that work integrates seamlessly with the work flow that I now propose in Chapters 5 and 6. For example, $\mathcal{DynAsT}$ only needs information concerning the management of dynamic memory and data object accesses and hence, in Chapter 2 I introduced an exception-based profiling technique that demands less effort from the designer than the template-based technique that I explain in depth in this chapter. However, $\mathcal{DynAsT}$ can use the information produced with the template library, and the exception-based technique can be employed in the context of metadata extraction (substituting the `var` template), taking into account that it is limited to memory accesses.

Static analysis is usually not enough to describe systems subject to dynamic environments. Thus, many optimization techniques start with a careful profiling of the applications with representative samples of actual input data. However, profiling is a time consuming process: First, applications need to be instrumented. Second, relevant inputs must be identified and supplied to the system. Then, the information generated has to be analyzed to infer the application characteristics. An important part of the profiling and analysis processes is done at the dynamic data type level, which can steer the designers choice of implementation for those data types. Finally, the results need to be categorized.

The complexity of this process is aggravated by the fact that different optimization tools may require information of different nature, adding to the overall effort required. Without a common information repository, each optimization team has to study the application independently to extract the characteristics that are relevant for their work, even if a considerable amount of information demands are usually shared.

A central repository of information with the characteristics of the applications would reduce significantly the effort required for optimization, even if the information collected is restricted to a single aspect of the design such as the memory subsystem. Although the process of metadata mining may be more complex than the analysis required by each team in isolation, the accumulated effort can be reduced because information is shared among the different teams. Figure 5.1 illustrates the potential gains. An important bonus of this approach is that once the software metadata have been extracted, the information is readily available, reducing the cost of entry of any further optimizations.

The contributions presented in this chapter and the next one are:

1. A uniform representation for the dynamic memory characteristics of applications – the software metadata.
2. A methodology to extract that information.
3. Concrete profiling and analysis techniques to implement the methodology.
4. An example of how software metadata can be used to implement several dynamic memory optimizations.

In this chapter I first present the metadata representation (Section 5.1) and the methodology proposed for its extraction (Section 5.3). Then, I study the feasibility of software metadata extraction with the discussion of specific methods to obtain the profiling information (Section 5.4) and the analysis techniques required to turn it into metadata (Section 5.5). Then, in

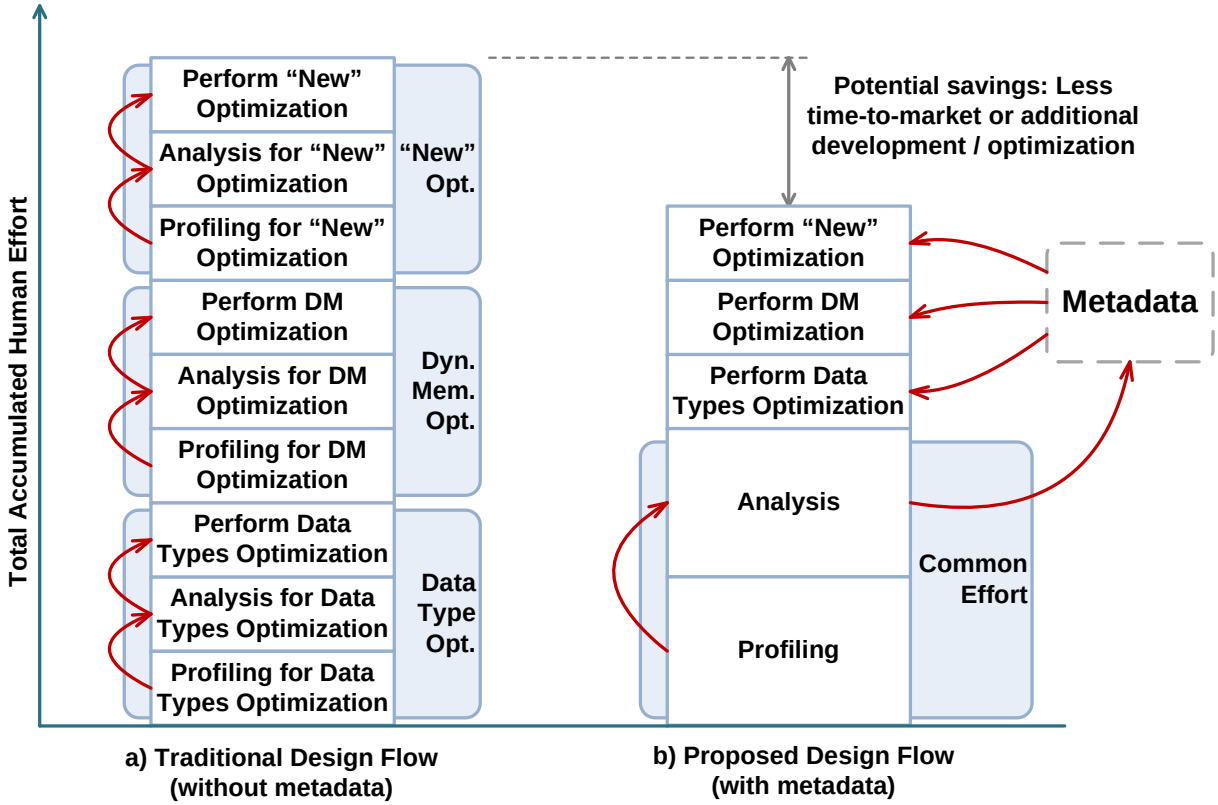


Figure 5.1.: Qualitative comparison of the effort invested when using a) a design flow with no information sharing or b) a common representation of software metadata to share characterization information.

Chapter 6 I show one case-study that employs the software metadata in various optimizations of energy consumption and memory footprint.

5.1. Software metadata structure

The knowledge about a software application can be seen from different levels of abstraction. I propose to classify it in the following three levels:

Level-zero metadata: Extensive characterization of the application behavior generally obtained through profiling.

Level-one metadata: Aggregate representation of the information at the previous level created by the analysis tools. This information is used and updated by the optimization tools during design time.

Level-two metadata: Information that is deployed with the final application detailing its resource needs. The run-time manager of the embedded system can use it to adapt the performance of the system to the needs of the applications currently running. This information can be completed with the variations of available resources during time (e.g., battery capacity, new modules that are plugged-in, etc.).

The focus of this chapter is on the metadata at level one. Hence, through the rest of this chapter the terms "raw" or "profiling information" are used when referring to the information

at level-zero, whereas the term "metadata" is reserved for the level-one information. The systematic generation and run-time exploitation of the information in level-two metadata, such as proposed in Chapter 2 for the deployment of placement solutions, is left for future work.

The concept of metadata for software running on embedded systems involves two parts: The metrics and the values assigned to them. The metrics present in the set of metadata information can be classified according to their main usage. Although software metadata is defined as a whole set, different optimization tools may employ different, potentially overlapping, subsets. An optimization tool may take as input the values of any of the metrics, use them to transform the application and update the affected metrics with the values derived from the new behavior of the transformed application.

5.2. Definition and categorization of metadata

The first question to define the software metadata is: What are the metrics present in the metadata? Any information regarding the behavior of an application that could be potentially used by an optimization tool should be included. For software applications this mainly concerns their resource requirements (memory footprint, memory bandwidth, cycle budget, dedicated hardware needs, etc.), but also any applicable deadlines, dependencies on other software modules, events that trigger specific behavior, etc. In some cases, the metadata needed by the optimization tools can be extracted from the profiling information (the raw data) in a straight-forward manner, but in most other cases more elaborate extraction techniques must be employed. Although the metadata metrics may cover all the relevant aspects of application behavior, the focus of this work is on the analysis of the memory behavior of the applications.

Figure 5.2 shows a complete view of the proposed software metadata structure for the dynamic memory behavior of embedded systems, including the semantic interconnections between different categories. The diagram is composed of three types of elements:

Abstract entities: Represented as boxes with a double line at the top. Abstract entities represent a *concept*, not an actual item in the metadata information. They are included in the diagram to make explicit that some concepts belong to the same category. For example, dynamic objects can be classified in simple variables and (composed) dynamic data types (represented by sequences and trees in the analysis). However, all of them have an associated identifier (data type ID) and information on the number of accesses performed on all of their instances. This is represented in the diagram via inheritance from the abstract entity "dynamic data." Other properties, such as the relationships of the abstract class itself, are also inherited: A memory pool is the place where dynamic data reside in; thus, each variable, sequence or tree can be related to the pool where they are hold.

Data entities: Represented with a simple box, as is the case of "control flow," "block transfers," "pool," "variables," etc. Each of these entities have associated attributes such as the size of the block transfers or the memory footprint of a dynamic data type instance. Regarding different DDTs, the figure includes both abstract entities and concrete implementations for sequences and trees, but any other dynamic data type could be added to the schema.

Relations: Including inheritance, composition and association. Inheritance, represented with a hollow arrow, expresses that the "child concept" replaces the parent concept in the metadata information while preserving all of the parental attributes. The parent concept groups characteristics common to several related categories. Association and composition are represented with a directed open arrow and with an arrow finished in a diamond, respectively. Relations can have attributes on its own. This is the case, for instance, of the relation "allocation information" between control flow and dynamic data entities, where the attributes of the relation express the number of reads, writes, etc., performed on an specific dynamic data instance inside a concrete control flow.

The metrics present in the metadata information can be classified according to their main usage, which embraces aspects as distinct as the number of accesses to a variable or the relationships between execution scopes and the dynamic data types that are accessed from them. The top level entity in the metadata is the "control flow." This entity holds information about the dynamic data ("dynamic data" entity) that are accessed within each scope in the form of individual memory accesses and data block transfers ("block transfer" entity). These accesses happen into actual physical memory locations that are represented by the "pool" entity, where the instances of the various dynamic data are allocated. Additionally, the control flow entity contains aggregated information in the form of access and allocation histograms, requested bandwidth and frequency of accesses per allocated byte.

The "dynamic data" entity contains information specific to each dynamic object, such as number of reads, writes or maximum required memory footprint. The nature of this information is the same for variables and structured data types and includes details about each individual instance, but also overall aggregated information. However, dynamic data types – as opposed to simple, generally scalar, dynamically allocated variables – are grouped under the "dynamic data type" instance, which encloses common information for all of them. This entity shows also the relation between dynamic data types and the operations that can be performed on them through the relation with the "dynamic data type operations" entity. In summary, the information regarding the dynamic data structures and variables of the application is presented at three levels of abstraction: Aggregated at the data type level, explicit for each concrete instance of each data type or variable, and specific to the operations performed on them.

Each variable or dynamic data type entity is associated to a pool where its concrete instances are allocated through the system's dynamic memory manager. Therefore, the "pool" entity aggregates the allocation, access, frequency per byte and bandwidth information for all the instances of the variables and dynamic data types that are created inside it. Finally, the information about potential data block transfers (such as the size and the number of times each individual block transfer is executed) is encapsulated in the "block transfer" entity. The following paragraphs explain each of the entities in more detail.

5.2.1. Control flow metadata

The control flow entity represents information specific to the application control flow, with a focus on the dynamic memory behavior at each location. Its instances can be seen as a series of invocations of source code blocks for each thread in the application. Starting from the entry that represents the `main()` function, each control flow instance can spawn an indeterminate number of independent control flows, either by entering into different scopes sequentially, or

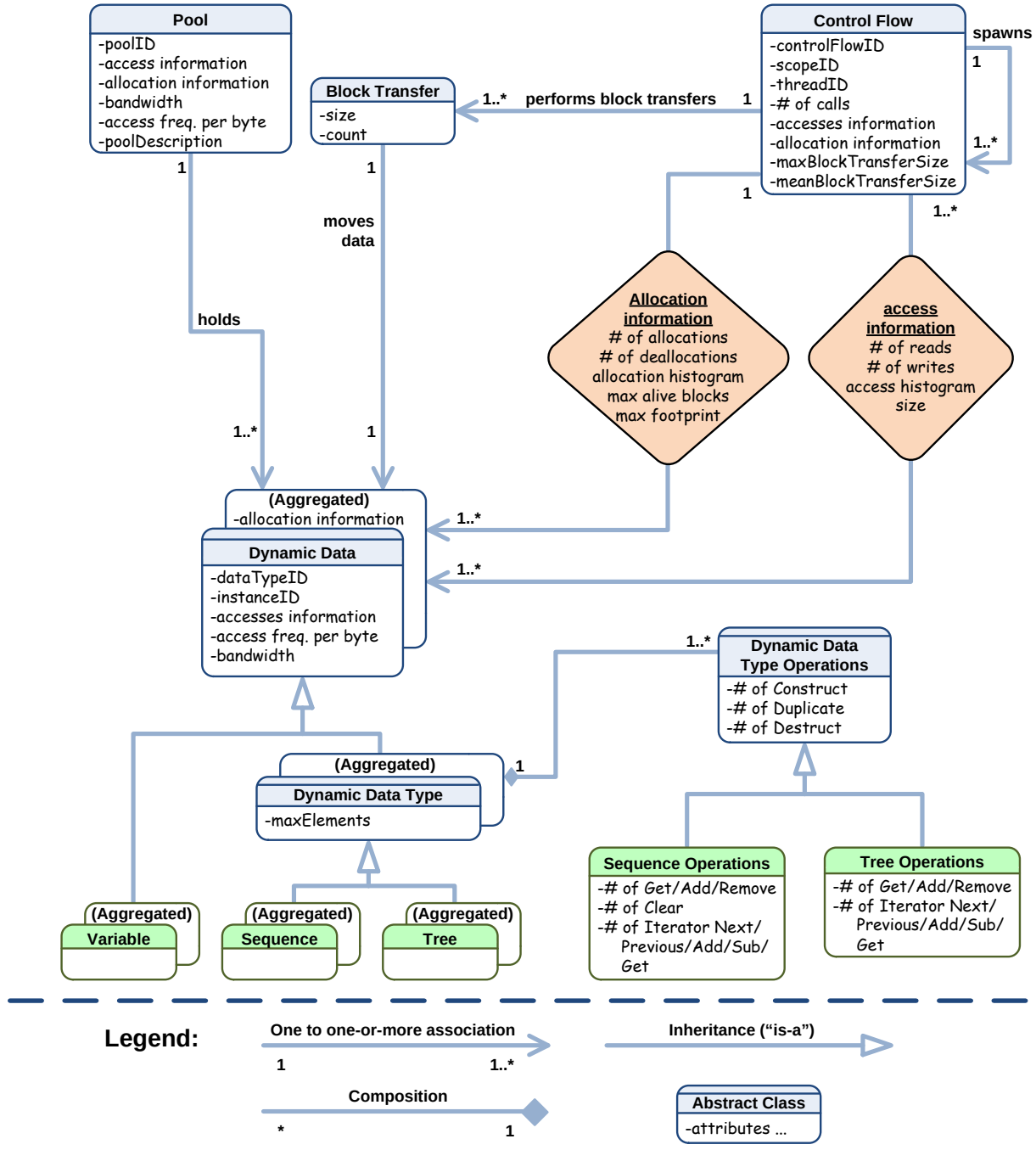


Figure 5.2.: Structure of the software metadata needed to characterize the dynamic memory behavior of applications running on embedded systems.

by launching new threads in parallel. With this information, the memory allocation or access behavior of any thread (through all the different scopes that it activates) can be analyzed. Other information such as deadlines or cycle budget was not included in this work to adhere to its main target – dynamic memory analysis – but including it should be doable.

Each control flow instance, referred as "scope," represents a designer defined location in the code – functions, loops, conditional structures. The information regarding the number of accesses to each dynamic data object is detailed in Figure 5.2 inside a diamond along the arrow connecting the various data entities to the control flow ("access information" relationship). As a dynamic data instance may be accessed from several code locations, the accesses performed by each of them is associated to the specific relationship between the concrete data instance and the control flow instance. This is the reason for the "1..*" cardinality at both extremes of that arrow. The other relationship between control flows and dynamic data objects is "allocation information," which details where each instance is allocated.

5.2.2. Dynamic memory allocation and access metadata

The concept of pool [WJNB95] is crucial for applications that use dynamic memory. Therefore, the information regarding the behavior of the different application pools (represented by the "pool" entity) is essential to enable optimization techniques such as dynamic memory refinement, dynamic memory assignment on memory resources or access scheduling. The pool entity represents a portion of the address space where some of the application dynamic data types are allocated. It stores information such as the description of the physical properties of the pool and the algorithms employed to manage it, total number of allocations and deallocations inside that pool, maximum number of objects simultaneously alive or the histogram of allocations along time. The pool information is usually not extracted during the analysis phase of the original application code itself; instead, it will be created, used and updated by different tools (e.g., for DMM optimization) that will use the metadata information to communicate and pass constraints between them. A relationship is established between a pool and the dynamic data entities that are allocated inside it (relation "holds"). The cardinality of that relationship illustrates that a single pool can store multiple dynamic data entities whereas, usually, a given dynamic data entity is always stored inside the same pool (in some cases, however, it may be possible to allocate instances of one dynamic data entity in different pools; the association cardinality would then be modified to "1..*" at the pool side).

The "variable" entity contains information related to accesses to dynamic variables, with a granularity down to every dynamic variable declared in the application. Additionally, information regarding maximum length, mean length and number of block data transfers performed in a control flow entity (i.e., a scope), which is different than the number of individual memory accesses, can also be extracted. This information, held in the "block transfers" entity, allows identifying the dynamic data types involved in bulk memory movements, the transfer sizes and the number of times such block data transfers are performed. Finally, a relationship with a control flow instance identifies the source code locations responsible for each transfer.

5.2.3. Dynamic data type metadata

The "dynamic data" entity represents any data type (simple or compound) whose instances are allocated inside a pool at run-time in numbers and sizes unknown at design time. This is an abstract entity that does not appear directly in the metadata, but has two useful purposes:

- Represent in the diagram relations that affect any of the derived entities. For example, a control flow executes accesses over some data types.

- Obtain aggregated information for all the derived entities. The metadata contains information on every concrete instance of variable ("variable" entity) or data type – "sequence" and "tree" entities, or any other dynamic data type if the model is extended – but it is also possible to obtain aggregated information for all the simple (scalar) variables, for all the sequences, for all the trees, for all the dynamic data types (i.e., sequences and trees) and for all the dynamic objects (i.e., variables, sequences and trees together). Therefore, an optimization tool can obtain the number of instances that are created of each dynamic data type, their total footprint and the total number of accesses to them, but also the number of accesses and footprint of each concrete instance of a dynamic data type.

Many instances of each of the derived entities can be created. The metadata of an application contains one instance of the corresponding entity for every dynamic data object – single variable or compound data type instance – created by the application. Some simple relations arise, such as that a control flow can create many dynamic data instances whereas an instance is created by only one control flow; this is reflected by the cardinality of the respective relationships in the diagram.

Dynamic data types support different operations depending on their nature. However, some of them are common: Creation, duplication or destruction. Generic operations are represented in the "dynamic data type operations" entity, while concrete operations are represented in the derived classes (sequences and trees in the figure). At the aggregate level, it is possible to determine the number of operations from the "dynamic data type operations" executed on all the dynamic data types of the application. At closer inspection, specific operations on sequences or trees can be accounted, both globally and for every instance.

5.3. Software metadata mining

An overview of the methodology used to extract and exploit the software metadata is shown in Figure 5.3. The starting point is the application source code.[1] The raw information about the dynamic memory behavior of the application is gathered through an extensive profiling at the dynamic data type level, using representative input data sets to trigger different aspects of the application behavior. Then, the analysis phase extracts the values for the metadata metrics from the raw data. Once the metadata values are defined, different tools can be linked in a pipe-line manner where they use as input the current metadata and generate an updated version that can be used by the next tool.

Identification of relevant input data sets is crucial to obtain meaningful profiling information because the behavior of dynamic applications is often influenced by the nature of the input. Sometimes a single characterization will be enough because the values of the different software metadata metrics will be similar, but in other cases different sets of metadata values will be needed for each set of input data. If each set of values is optimized independently, a

[1]This work is focused on C++ applications and hence, the presented profiling technique is adapted for that language. However, the metadata concepts are independent of the programming language and are valid for other (imperative) languages, provided that equivalent profiling methods are available.

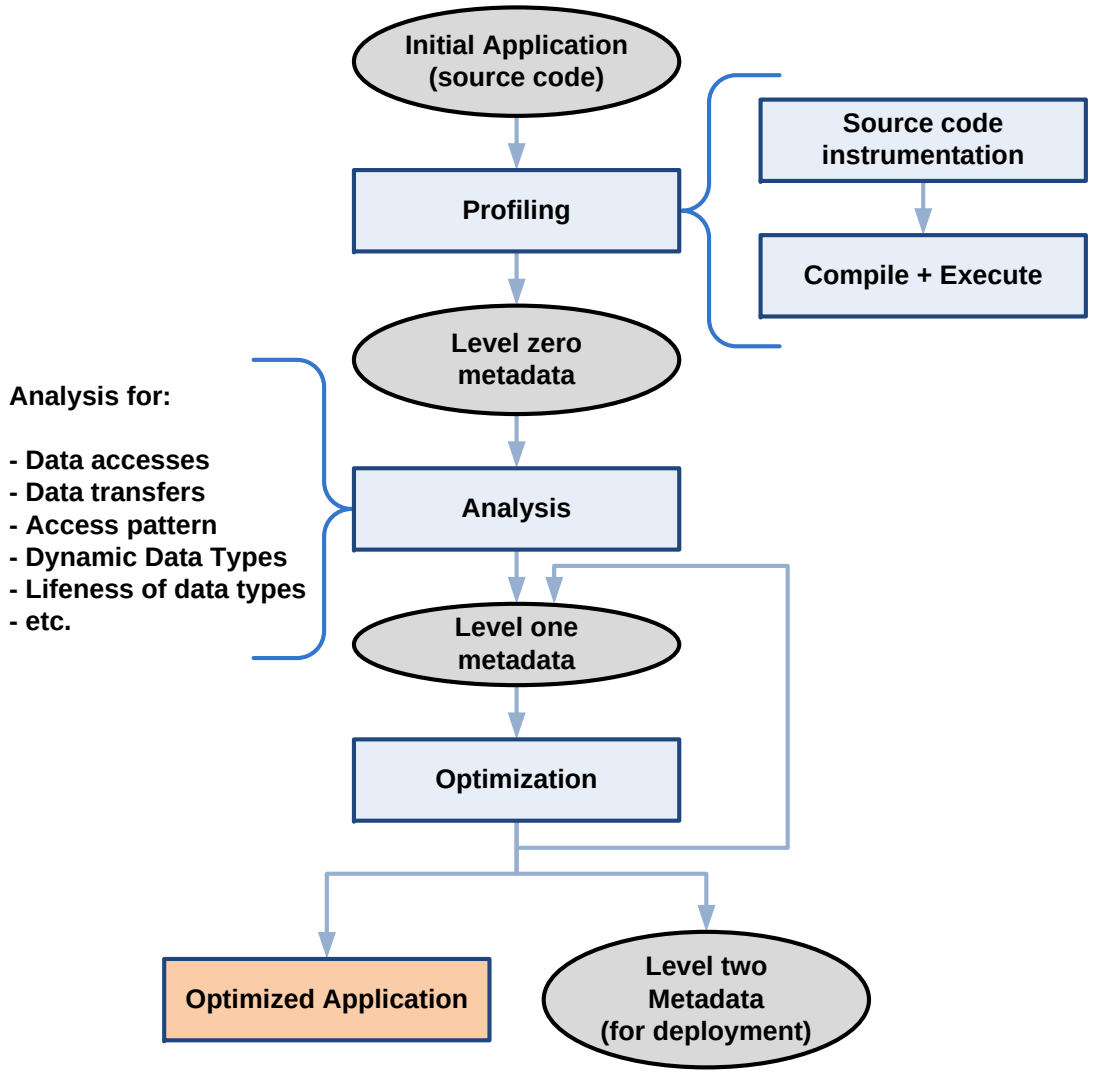


Figure 5.3.: Overview of the methodology.

mechanism to identify the characteristics of the current input instance at run-time and the corresponding set of metadata will be needed to apply the right optimizations. System scenarios, as presented in Section 4.5.3, are an option for this task.

5.4. Profiling for raw data extraction

The information related to the dynamic data behavior of the application that will be used by the analysis step to extract the proper metadata is collected during the preliminary profiling step. For the metadata as used in this chapter, the profiling step has to extract information on: a) allocation and deallocation of dynamic memory; b) memory accesses (reads and writes), both for scalar variables and for dynamic data types; c) operations on dynamic data types; d) control-flow paths (changes of scope) that lead to the locations where those operations are performed; and e) identifiers of the threads that perform these operations. The extracted raw information has to keep a sense of order, so that aggregated constructs such as histograms of memory footprint variations along time can be created.

Profiling information can be obtained through several techniques. The two main groups are based on binary code interpretation and source code instrumentation. Both approaches have advantages and disadvantages. Techniques that work directly on binary code do not require modifications to the source code, but the extracted information may be difficult to relate to the application DDTs – see Section 7.3.2 for a proposal to improve this situation through compiler-based analysis. By contrast, those based on source code instrumentation usually provide information that can be related to concrete items in the source code. This second category can be further split into automated methods – the compiler or other tool, which may even work in the same framework than the compiler, introduces the instrumentation – and manual ones – the programmer modifies the source code to introduce more specific instrumentation.

For the work described here, a semi-automated type-based approach as presented by Poucet et al. [PAC06] was selected to profile dynamic data type behavior, allocation and deallocation patterns and number of accesses. With this solution, the designer has to annotate the types of the relevant variables, but the compiler takes care automatically of annotating all accesses to them through the source code, guaranteeing that no access is overlooked. Appendix D details the format of the log file that contains the raw metadata information.

5.4.1. Profiling using a template-based library

This section describes the profiling method[2] that was used during the experiments presented here to profile applications written in C++. However, it is in no way the only profiling mechanism that could be used for the job. The type annotation is performed through templates, to wrap types and give them a new type, and operator-overloading, to capture all the accesses to the wrapped variables. Templates are compile-time constructs that describe the general behavior of a class (or function) based on type parameters, thus they are sometimes known as "parameterized types." When the class template is instantiated with the desired type, the compiler generates the correct instructions to deal with it.

The profiling works as follows. At design time, the programmer introduces three types of annotation. First, dynamic classes are modified to inherit from a special class that will overload their `new` and `delete` operators. Second, attribute members of profiled classes – but only those of basic or primitive types – are wrapped in a template that overloads all accessors to that variable. If an attribute is an instance of other class, then the attributes of that class need to be recursively wrapped – this being the potentially tedious part of the process. When the compiler generates the binary code, for every access operation it will introduce the code contained in the template. Finally, temporary "scope" objects are introduced to mark the beginning and end of each region of code of interest. The profiling library includes classes that build a logger in several flavors, mainly binary or textual output.

At run-time, a global instance of the logging class is generated. When a new object is created the overloaded `new` operator accesses the logger class to dump the related information. In an analogous way, the `delete` operator dumps the relevant information when the object is destroyed. Finally, when a scalar variable or class attribute is accessed, the compiler-inserted

[2]The profiling library presented in this section was originally developed by Christophe Poucet at IMEC. As far as I know, it was first published in 2006 [PAC06], but we were using it internally at IMEC even before that. Later, we collaborated during the work that produced most of the content of this chapter and was published in [BPP+10] and [AMP+15]. I explain this library here, even if I did not develop it, because it is a fundamental part of the methodology.

code calls the logger with the access information.

The potential drawbacks of this technique are:

1. Annotating all the attributes of all the classes in multiple nested levels can be tedious. However, its complexity is much lower and the process is less error-prone than annotating manually all accesses to the objects.

2. The execution time of the application is modified. Although not a serious issue for most applications, real-time communication may be difficult to deal with.

3. Logging all variable accesses is a space consuming process that generates big log files. It could be worth investigating the possibility of a joint profiling-analysis phase that processes each access directly at run-time. For applications not bounded by real-time constraints, this could solve the need for storing huge log files.

Description and use of the profiling library

The profiling library consists of several orthogonal class templates, built to be as little obtrusive as possible. Each of them logs different information. The library contains also a set of auxiliary classes for information formatting and recording. Here, I explain only the basic elements needed to understand the structure of the library.

The following code fragment shows the basic structure of the logging class that writes a binary record for every application event:

```
class DMMLogger {
  enum LogType {
    LOG_VAR_READ = 0,
    LOG_VAR_WRITE = 1,
    ...
    LOG_MALLOC_END = 5,
    ...
  };

public:
  inline static void log_read(const void * addr, const unsigned int id,
           const size_t sz) {
    write_header(LOG_VAR_READ, 4 * sizeof(unsigned int)) << id <<
      addr << sz << (unsigned long)pthread_self();
  }

  inline static void log_malloc_end(const unsigned int id,
           const size_t sz, const void * addr) {
    write_header(LOG_MALLOC_END, 4 * sizeof(unsigned int)) << id <<
      addr << sz << (unsigned long)pthread_self();
  }

private:
  DMMLogger & write_header(LogType logType, unsigned short logSize) {
    unsigned short logType_s = (unsigned short) logType;
    if (logFile_ != NULL) {
      fwrite(&logType_s, sizeof(unsigned short), 1, logFile_);
      fwrite(&logSize, sizeof(unsigned short), 1, logFile_);
    }
    return *this;
  }
```

```
  DMMLogger & operator<<(const unsigned int num) {
    if (logFile_ != NULL)
      fwrite(&num, sizeof(unsigned int), 1, logFile_);
    return *this;
  }

  DMMLogger & operator<<(const void * addr) {
    if (logFile_ != NULL)
      fwrite(&addr, sizeof(unsigned int), 1, logFile_);
    return *this;
  }

  FILE * logFile_; // Initialized in the constructor.
};
```

Additional methods, such as `log_write`, `log_malloc_begin`, `log_free_begin`, `log_free_end`, `log_scope_begin`, `log_scope_end`, `sequence_get`, `sequence_add`, `sequence_remove`, `sequence_clear`, `map_get`, `map_add`, `map_remove` or `map_clear`, are constructed in an analogous way. The set of entries in the `logType` enumerated contains all the needed definitions.

The class that adds logging capabilities to the `malloc` and `free` functions is implemented as follows:

```
template <int ID, typename Logger = DMMLogger>
class logged_allocator {
public:
  inline static void * malloc(const size_t sz) {
    Logger::log_malloc_begin(ID, sz);
    void * ptr = ::malloc(sz);
    Logger::log_malloc_end(ID, sz, ptr);
    return ptr;
  }

  inline static void free(void * ptr) {
    Logger::log_free_begin(ID, ptr);
    ::free(ptr);
    Logger::log_free_end(ID, ptr);
  }
};
```

For the actual code instrumentation, the designer uses the following class templates:

- **allocated:** Class instances are normally created and destroyed through the `new` and `delete` operators. This class template overloads them to transparently generate profiling tokens. The original requests are typically forwarded to the system allocator through the underlying `malloc()` and `free()` functions. Alternatively, allocation and deallocation requests may be forwarded to a custom memory allocator.

  ```
  template <class Allocator>
  class allocated {
  public:
    void * operator new(const size_t sz) {
      return Allocator::malloc(sz);
    }
    void operator delete(void * p) {
      return Allocator::free(p);
  ```

```
  }
  void * operator new[](const size_t sz) {
    return Allocator::malloc(sz);
  }
  void operator delete[](void * p) {
    return Allocator::free(p);
  }
};
```

In order to log all the dynamic memory events related to the instances of a class (or structure), the designer needs only to modify its declaration so that it inherits from `allocated`. No other lines in the class source code need to be modified:

```
class NewClass : public allocated<logged_allocator<1> > {
  ...
};
```

- **var:** This class template wraps the declaration of individual variables or class attributes and generates a profiling token for each memory access to them. Additionally, it derives from the "allocated" template, thereby providing also allocation and deallocation logging for scalar wrapped variables. The following is an excerpt of the template implementation:

```
template <typename T, // Type of the wrapped object
  int ID, // Id used in the profiling tokens
  class Logger = DMMLogger,
  class Allocator = logged_allocator<ID, Logger> >
class var : public allocated<typename Allocator> {
public:
  T data_;

  // Constructor
  template <typename T2>
  var(const T2 & data)
     : data_(data) {
    Logger::log_write(&(data_), ID, sizeof(T));
  }

  // Assignment operator from basic type
  template <typename T2>
  var & operator= (const T2 & data) {
    data_ = data;
    Logger::log_write(&(data_), ID, sizeof(T));
    return *this;
  }

  // Assignment operator from wrapped type
  template <typename T2, int ID2>
  var & operator= (const var<T2, ID2> & other) {
    Logger::log_read(&(other.data_), ID2, sizeof(T2));
    data_ = other.data_;
    Logger::log_write(&(data_), ID, sizeof(T));
    return *this;
  }
};
```

This template can be used to wrap variables of any basic type, including pointers and class attributes. To profile accesses to the attributes of a class or structure, it is necessary to go inside that class and wrap all the relevant attributes. If any of the attributes are also structures, then the technique can be applied recursively until the basic types are reached:

```
var<int, 1> oneVariable;
var<int, 2> * onePointer = new var<int, 2>; // The new operator is
                              // overloaded by the var<> template

struct A {
 var<int, 3> value;
};
struct B {
 A a;
 var<int *, 4> bar;
};
```

A special consideration is needed when wrapping pointers. In the following code fragment, the first declaration means that only accesses to the pointer `quux` itself, but not the accesses to the integers that are pointed to, are logged. However, when both accesses are of interest, the pattern given for the definition of `baz`, which is a combination of the previous patterns, should be used instead:

```
var(int, 1) * quux;
var(var(int, 2) *, 3) baz;
```

In summary, with the "var" template, the designer does not need to worry about identifying all the positions in the source code where the variables are accessed: The compiler will ensure that they are properly logged.

- **vector:** This class template replaces the `vector` container from the STL (C++ Standard Template Library [SGI06]) in order to profile sequence usage behavior at the dynamic data type abstraction level. The implementation of the template matches that of the STL one from a logical point of view, but adds logging at the right places to give a high-level view of the data structure usage pattern (e.g., how many insertions, linear traversals, or random accesses). This information can be used to determine the concrete data type that best suits the observed usage pattern, as shown by Atienza et al. [ABP+07]. The declaration of a vector with this template is straightforward:

  ```
  vector<int, 1> oneVector;
  ```

  The "vector" and "var" templates can be combined to profile also accesses to the actual data items inside a sequence.

- **scope:** This class template is used to mark different sections of the control-flow. The following code fragment illustrates its implementation:

  ```
  template <class Logger = std_scope_logger>
  class scope {
  public:
    scope(std::string name)
       : name_(name) {
     Logger::log_scope_begin(name_);
    }
  ```

```
  ~scope() {
    Logger::log_scope_end(name_);
  }

private:
  std::string name_;
};
```

The designer can create an instance of this class template whenever an interesting location of the application code is entered. The compiler-generated constructor and destructor will generate the profiling token, which the analysis tools can use to determine the portions of code executed, the sequence of activations and, most importantly, the relation between code locations and the different profiling events (allocations, accesses to dynamic data types, etc.). Additionally, since the scope template can be combined with thread identification, those events can be associated with the specific threads that performed them. The following is an example of utilization for scope:

```
void Algorithm1() {
  scope algorithm1("Algorithm␣1");
  // Declare variables...
  var(int, 1) * quux;

  // ... and do something interesting on them...
}

void ControlFunction() {
  scope a("Control␣function");

  for (int ii = 0; ii < 2; ++ ii)
    Algorithm1();
}
```

The output of the logger (once translated into textual form) would be:

```
1  Scope "Control function" begins
2  Scope "Algorithm 1" begins
3  ... accesses to quux logged here
4  Scope "Algorithm 1" ends
5  Scope "Algorithm 1" begins
6  ... accesses to quux logged here
7  Scope "Algorithm 1" ends
8  Scope "Control function" ends
```

5.5. Analysis techniques for (level-one) metadata inference

Once the profiling information has been obtained, several analysis steps can be applied to extract and compute the relevant metadata metrics that will be used by the optimization tools to reduce energy consumption, memory accesses and memory footprint. Each of these optimization tools will use specific parts of the metadata set. Therefore, the metadata information can be seen as structured in different, potentially overlapping, slices of interest. This section presents several techniques that can be used to extract different portions of the metadata information.

Algorithm 3 Access analysis

```
1: process(event)       // event.type is variable-read or variable-write
2:    Update application reads/writes counter
3:    Update reads/writes counters of current control-flow point
4:    with findBlockAtAddress(event.address)
5:       Update block reads/writes counter
6:       Update DDT & DDT-instance reads/writes counter
```

An important consideration is that the profiling system does not include information about "wall time." Part of the reason is that the profiling mechanism itself alters the application timing, slowing it down;[3] absolute timestamps are hence not meaningful. Instead, timing is defined in terms of events of specific types, usually allocation events. As the optimization processes targeted with this approach to metadata are focused on optimizing dynamic memory behavior – performance improvements are indirectly attained through optimization of the accesses to dynamic objects – this choice should be appropriate.

The analysis process is structured as a set of objects that perform specific analysis tasks. The main driver reads every packet from the log file produced during the profiling phase and invokes in turn all the analysis objects. After all of them have processed the packet, the main driver moves forward to the next packet.

5.5.1. Accesses to dynamic objects

Algorithm 3 shows the work performed by the analyzer of accesses to variables. Whenever a read or write event is found in the log file, the global read or write counters (for the DDTs) of the application and the current scope and thread are updated. Using the address of the memory access, the concrete dynamic data type instance that was being accessed can be identified. This information allows updating also the number of accesses of the corresponding dynamic data type and DDT-instance.

5.5.2. Dynamic memory behavior

Another important analysis target is the allocation behavior as such information can be used to design highly-tuned application-specific dynamic memory managers. Therefore, the tools extract also the number of allocations per block size, the different block sizes and the number of accesses per block size, which enables the automatic exploration of application-specific dynamic memory managers, as studied by Mamagkakis et al. [MAP+06]. In this regard, the accesses to variables encountered between `MallocBegin` and `MallocEnd`, or between `FreeBegin` and `FreeEnd` can be used to evaluate the computational overhead of different dynamic memory managers. Finally, these data can also be used to detect memory leaks (although there

[3] This is a relevant factor that should be taken into consideration. Logging every access to every dynamic object introduces a high overhead on execution time. The case study shown in the next chapter is based on network traces that were collected off-line and replayed during the experiments without a notion of "wall time." Indeed, it seems close to impossible to profile an application without interfering on its timing unless external hardware devices are connected to the profiled system. However, this should not constitute a problem except for systems whose behavior changes completely subject to real-time conditions. For those cases, more advanced profiling techniques out of the scope of this chapter would be needed. For example, the memory subsystem could be augmented during development with a parallel structure to log the different data accesses in real-time, as presented by García et al. [GAM+06] and Atienza et al. [AGP+08].

Algorithm 4 Identification of dynamic objects

```
1:  process(event)
2:    case event of
3:      · AllocEnd →
4:        Create new live-block with event.size, event.address and the information
5:          for the currently active control flow in this thread
6:        Increase allocation count for event.size
7:      · DeallocEnd →
8:        with findBlockAtAddress(event.address)
9:          Increase deallocation count for event.size
10:         Add number of accesses to accesses for blocks of size event.size
11:         Destroy live-block
12:     · VarRead or VarWrite →
13:       with findBlockAtAddress(event.address)
14:         Increment read or write counter for this live-block
15: finalize()
16:   forall block ∈ live-blocks
17:     Report memory leak for this block and where it was allocated
```

are arguably easier methods to extract just this information). Algorithm 4 shows how this information is extracted.

5.5.3. Block transfer identification

Identifying block transfers amongst the individual memory-access events enables the application of optimizations at the data transfer level. For example, dedicated hardware resources (e.g., DMA engines) can be used to perform the longest data transfers, saving computing cycles from the main processing elements. In this area I proposed, in collaboration with the rest of authors, a technique to selectively perform data transfers using a DMA module in embedded systems [PBM+07]. Block transfers are not distinguished in the raw information, for they are just a collection of independent memory accesses. Therefore, their existence must be inferred from the properties of individual accesses. For static data types, this type of analysis is relatively easy because the size and address (placement) of the different structures is predefined; hence, individual accesses can be immediately related to the corresponding static object. However, dynamically allocated data objects do not have a predefined address. Indeed, it is not even possible to know how many instances of each one will be created at run-time, each of them participating in its own data transfers.

Algorithm 5 can be used to identify the data transfers that involve dynamic objects, where a data transfer is defined as a set of strictly consecutive accesses to a given instance of a data type, performed by one thread. In essence, the algorithm keeps the last accessed address for every dynamic object and the thread that performed it. For every new memory access event in the log, if the address is consecutive and comes from the same thread, the transfer is updated – that is, enlarged. When the next access is no longer consecutive, or the object is accessed by a different thread, the block transfer is closed and a new one is started. Here, the assumption that a block transfer is executed uninterrupted by a single thread stems from the fact that potential block transfers are mixed with scalar accesses in the level-zero (log file) metadata; therefore, the algorithm assumes that the object is locked during a block transfer. If a new thread is seen accessing the object, it infers that the lock on the object was released. The possibility that a single thread participates in multiple block transfers at the same time (e.g.,

Algorithm 5 Data block transfer identification

```
1: process(event)
2:   case event of
3:     · AllocEnd →
4:       Create new block with event.address
5:     · DeallocEnd →
6:       with findBlockAtAddress(event.address)
7:         Record last active transfer for the block
8:         Destroy block
9:     · VarRead or VarWrite →
10:      with findBlockAtAddress(event.address)
11:        if is consecutive access from same event.threadID and same direction
                                                                    (read/write)
12:          Update active transfer with event.address
13:        else
14:          Record last active transfer (if any)
15:          Create new transfer with event.address
```

when copying data between buffers) is implicit in this approach. This interpretation may be modified if needed for future work. The algorithm follows these guidelines:

1. The algorithm keeps information on the current data transfer, including the last address accessed (to check the consecutiveness of the next accesses), for every alive dynamic data object. In the case of scalar accesses, this is equivalent to storing the information of the last access. In parallel, the algorithm builds a record with finished data transfers.

2. Each time a `malloc()` memory allocation primitive is encountered, the analyzer creates a new dynamic data object representation, univocally identified by its starting address and size.

3. Similarly, the analyzer uses the address parameter of each `free()` deallocation primitive in the log file to identify and destroy the corresponding dynamic data object representation. If the object had an open transfer, then that transfer is considered as finished and moved to the record of transfers.

4. The analyzer checks the address of every access in the log file against the starting and ending addresses of the currently alive dynamic objects. If the address lies within the boundaries of an alive object, then it checks whether the new access is the continuation of the active transfer – by the same thread – for that block or not. If so, the analyzer updates the information of the active transfer for that block. Otherwise, the transfer is closed and, if its length is bigger than one word, moved to the record of data transfers; afterwards, the analyzer creates a new active transfer for the object using the address of the access that is being evaluated as the transfer starting address. If the analyzer encounters an access for a dynamic object without an active transfer, then a new active transfer is created for that object. Finally, if the analyzer does not find an alive dynamic data object spanning the address of the data access, then it is qualified as an access to a block of static data and discarded – in the assumption that other techniques already cope with those ones.

Algorithm 6 Analysis of sequence operations

```
1: process(event)
2:   case event of
3:     · VectorConstruct →
4:       Create new live-sequence with event.seqId, event.instanceId and the
5:         information for the currently active control flow in this thread
6:       Increase creation count for event.seqId
7:     · VectorDestruct →
8:       Increase deallocation count for event.seqId
9:       with findSeqInstance(event.instanceId)
10:        Destroy live-sequence
11:    · VectorResize →
12:      Increase resize count for event.seqId
13:      with findSeqInstance(event.instanceId)
14:        Update sequence instance size
15:        Increase resizing count for that instance
16:    · VectorGet/Add/Remove/Clear →
17:      Increase Get/Add/Remove/Clear count for event.seqId
18:      with findSeqInstance(event.instanceId)
19:        Increase Get/Add/Remove/Clear count for that instance
20:    · IteratorNext/Previous/Add/Sub/Get →
21:      Increase IteratorNext/Previous/Add/Sub/Get count for event.seqId
22:      with findSeqInstance(event.instanceId)
23:        Increase IteratorNext/Previous/Add/Sub/Get count for that instance
```

5.5.4. Sequence (DDT) operations

Finally, gathering information on the statistical behavior of the application sequences allows discerning the average number of elements in the sequences of each specific type and the frequency of each operation. This information enables optimization techniques that choose the most efficient implementation for each sequence: Statically allocated arrays or linked lists, type of iterators, etc. Algorithm 6 shows how this information can be extracted in a straightforward way.

The tags `seqId` and `instanceId` (see Appendix D for reference) serve a similar purpose as the `varId` and `address` for discrete objects: `seqId` is introduced at design time and identifies a family of sequences. If the sequence is statically declared, the family contains one element; however, if the sequence is dynamically allocated itself, for example, using `new` inside a loop, then the family of sequences with a given `seqId` contains multiple instances each with a different `instanceId`. The information for individual sequence instances is not currently employed, though, due to the difficulty of identifying different instances created at the same code location along execution.

The number of resizing events for a family of sequences is useful to decide whether they can be implemented as a fixed-size vector (which requires reallocation and copy of elements in case its capacity is exceeded) or whether a linked structure is more appropriate. The number of operations for each iterator can be used also to identify the most efficient implementation. For example, a significant number of `IteratorPrevious` events would suggest using a doubly-linked list. Typical sequence operations such as `push_back()` are implemented in terms of iterators and appear correspondingly in the log file.

Chapter 6

Experiments on software metadata: An integrated case study

In this chapter I present an integrated example in which different methods for dynamic data type, dynamic memory management and dynamic-data block transfer optimization are applied on a single application to obtain significant gains in energy consumption and memory footprint. The driver application is a network TCP/IP-like stack that includes Deficit-Round-Robin (DRR) scheduling of outgoing packets. This application is similar to one of the cases studied in Chapter 4 for data placement. However, the emphasis is placed here on how different optimization methods exploit different parts of the metadata.

6.1. Goal and procedure

The goal of this example is to show how the software metadata can be used to optimize an application at three levels: Dynamic data type, dynamic memory management and block transfers of dynamic data. The optimizations are applied in order of decreasing abstraction level. This consideration is important when optimizing a complete system to avoid loops; for example, the number of operations over each DDT and their type depends on algorithmic decisions and holds independently of the concrete DDTs selected. Similarly, DMM selection affects application performance, but not the number of DDT operations nor of memory allocations. The work is divided in the following steps:

1. Extraction of software metadata from the original application using the profiling and analysis techniques presented in Chapter 5.

2. Optimization of the dynamic data types to reduce the number of memory accesses and memory footprint of the dynamic data structures used in the application (e.g., linked lists, double-linked lists).

3. Optimization of the dynamic memory management to support a more efficient implementation of the `malloc()` and `free()` operations. After this step, the number of memory accesses performed in the application to manage dynamic memory is minimized.

Correspondingly, the total memory footprint required by the dynamic memory manager to serve all the demands of the application is also minimized (i.e., by reducing the internal and external fragmentation caused by the manager itself [WJNB95]).

4. Optimization of the block transfer behavior of the application to exploit the DMA resources for transfers of blocks of dynamic data (as opposed to the transfer of static arrays or variables). This step takes into consideration the effect of the concurrent accesses from the processor and the DMA on the banking scheme of DRAM memories.

The memory behavior of the application is analyzed after each step to evaluate the impact of each optimization technique. In the end, the application is simulated to assess the accumulated effect of all the optimizations.

This case study focuses more on showing the application of each technique and its effects, and not so much on how the metadata values are updated after each phase because most of the changes are minor: DDT optimizations do not change the type of operations executed by the application but how they are handled, nor do they affect the number of objects allocated by the application; DMM optimizations do not alter the number of accesses executed by the application on the objects, whatever their positions in the pools are.

An important remark is that a designer would not need to conduct exhaustive explorations as the ones presented in this chapter to assess the impact of each optimization. Instead, each optimization tool would normally work on the metadata and the application to produce a final solution. For example, during the experiments of this chapter we activated specific profiling options to monitor memory accesses between calls to the dynamic memory manager, which correspond to work done by the DMM itself. These accesses would normally not appear in the software metadata of the application as they are not necessary for DMM optimizations – only the allocation pattern of the application is required. The experiments include these values to verify and analyze the result of the decisions.

6.2. Description of the driver application

The driver application of this example is a simplified TCP/IP-like network subsystem. The application is organized in several threads that communicate through asynchronous FIFO queues (Figure 6.1) so that the output of one thread is the input for the next one:

- Packet injection. To simulate traffic generated by client applications, this subsystem uses a collection of (wireless) network traces from the repository built by Henderson et al. [HKA04].
- Packet formation. Using data from the network traces, this subsystem adds a TCP/IP header to each packet that enters the system.
- Encryption (DES). During the experiments, we identified a number of service port numbers that belonged to encrypted sessions. Packets in the traces sent to those ports are passed through this encryption subsystem to approximate the work of a block cipher.
- TCP Checksum.
- Scheduling using Deficit Round Robin.

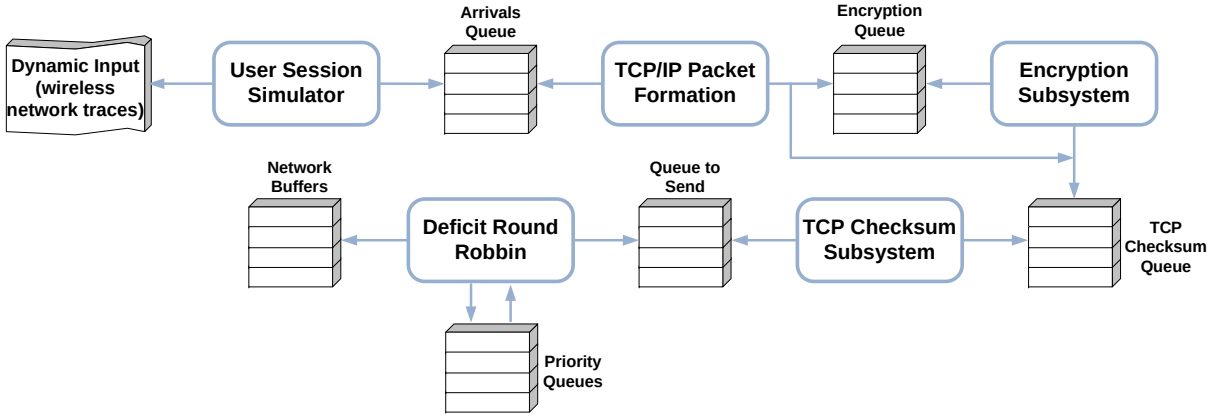


Figure 6.1.: The application framework used in this case study. The boxes represent the different threads/kernels that communicate through asynchronous FIFO queues.

Deficit-Round-Robin (DRR) is a network fair scheduling algorithm that splits the available bandwidth evenly among a number of destinations [SV96]. In DRR, the service manager keeps a list of active destinations and assigns a quota to each of them, possibly taking into account relative priorities. As packets become ready for sending, they are added to the queue corresponding to their destination. When a packet is forwarded to the network adaptor, the credit of its queue is reduced proportionally to the size of the packet. In this setup, the forwarding of packets to the network adaptor is simulated as a copy to a circular buffer in memory that can be traced by the profiling and analysis tools. In most real systems, those transfers would be performed by a dedicated DMA engine. However, they are included in the profiling of the system because this and previous works [BPM+09] include considerations about scheduling of processor and DMA accesses over DRAMs.

These subsystems form the basis of a simplified network stack. Due to the fact that the experiments used anonymized network traces collected from the wireless access points of a university campus [HKA04], and not from the actual devices, reproducing details like packet retransmission or window-based TCP rate control is very difficult; hence, they were left out of the experiments.

The modeled system is multithreaded, which means that multiple packets are alive at the same time during execution; thus, memory accesses happen concurrently from multiple threads. Multithreading eventually means that memory accesses from different threads interleave in a fine-grained way and the overall behavior cannot be described from the independent behavior of each thread. Therefore, the whole system must be optimized instead of each thread independently.

The target architecture for this system consists of a processing element connected to an internal SRAM and to an external DRAM module – Table 6.1 describes their working parameters. A Direct Memory Access (DMA) engine takes care of data movements between the external and the internal memories, and from any of the memories towards the external devices (i.e., hardware buffers in the network adapters). The processor can access both memories directly; however, access to the external DRAM has to be coordinated with the DMA to avoid incurring unnecessary energy and latency penalties due to row-level interferences.

The network traces used as input contained anonymized data from multiple users and applications. A simple analysis based on source and destination addresses, port numbers and temporal windows was used to extract the individual sessions employed during the exper-

Table 6.1: The SDRAM is modeled according to Micron PC100 specifications assuming CL = 2 and a system and processor clock of 100 MHz.

Energy / access	3.5 nJ
Energy activate / precharge	10 nJ
CAS latency	2 cycles
Precharge latency	2 cycles
Active to read or write	2 cycles
Write recovery	2 cycles
Last data-in to new read/write	1 cycle
Max burst length	1024 words

iment. Those sessions had varied characteristics; for example, the number of packets sent varies from 4687 up to 123 574 and the total number of bytes sent varies from about 14 KB up to 140 MB.

6.3. Profiling and analysis

Using the techniques explained in the previous chapter, the driver application was instrumented and profiled, producing the initial set of software metadata:

- For each dynamic data structure, the number of operations of each type executed, number of memory accesses needed to accomplish them and total memory footprint.

- For the dynamic memory manager, the number of memory accesses executed to manage all the free and used blocks of dynamic memory, and total amount of memory actually used to serve all the petitions from the application – that is, including the overheads due to internal and external memory fragmentation, which are caused by the manager when subject to the application behavior.

- Finally, the number of accesses to each dynamic object.

This information allows identifying the most relevant dynamic data types in the application. The first one corresponds to the body of the network packets. Their huge number demands high efficiency from the dynamic memory manager. The next most relevant data structures are the list of nodes and the queue of packets built by the DRR module. The list of nodes represents the hosts to which a connection is active, i.e., there is an entry for each destination host for which there are packets waiting to be sent. Each entry in the list contains the queue of packets waiting to be sent towards that destination. Both data structures are dynamic because their number of elements varies at run-time; thus, they are good candidates for data type optimization techniques. Finally, the last of the relevant dynamic data structures used in the application is the asynchronous FIFO queue used by the threads to pass messages between them. However, that data structure makes heavy use of synchronization primitives, which are out of the scope of this work. Nevertheless, its best implementation may be determined in a straightforward way because it presents a regular FIFO access pattern.

6.4. Dynamic data type refinement

The first optimization applied in this example is the refinement of the dynamic data structures of the application.[1] After the profiling and analysis steps, the metadata entities derived from Dynamic Data Type and Dynamic Data Type Operations contain information on the type and number of operations performed for each data type. This information is also available for their concrete instances, which makes it possible to study and optimize each of them according to the operations performed more frequently. The profiling and analysis methods allow differentiating between operations performed on the data structures themselves and accesses to the internal elements of their implementation. This differentiation makes it possible to identify the data types that are the best candidates for optimization independently of their initial implementation.

An important consideration is that the optimization techniques presented here focus on optimizing the overhead imposed by each data structure, but they cannot reduce the number of operations that the application algorithms execute on them. In order to reduce the number of operations – in contrast to the number of memory accesses actually performed on the memory subsystem – optimizations at a higher abstraction level than the ones presented through this work would be needed.

The analysis of the application reveals that the two dynamic data instances that concentrate most of the application accesses are the list of nodes in the DRR algorithm (dynamic structure "A") and the queue of pending packets for each of the nodes (dynamic structure "B"). The best option for the implementation of each DDT was chosen using the methods explained by Bartzas et al. [BMP+06]. Although running an exhaustive exploration of all cases is not needed to get the optimal solution, in this experiment we ran a complete sweep of all the combinations to perform an additional comparative test and validate the optimization approach. For each of the dynamic data structures, one of the following implementations may be chosen (more details on each of the implementations can be found in the work by Mamagkakis et al. [MBP+07]):

1. Array of pointers
2. Array of objects
3. Single-linked list
4. Double-linked list
5. Single-linked list with roving pointer
6. Double-linked list with roving pointer
7. Single-linked list of arrays
8. Double-linked list of arrays
9. Single-linked list of arrays with roving pointer
10. Double-linked list of arrays with roving pointer

Trees are not represented in this hierarchy, which is more focused on sequence-like behavior. Although they might be implemented using a combination of these data structures, a comprehensive study of the interfaces exposed by trees would be required to realize an efficient exploration of their design space. Such study would have first to classify the different tree-like data structures before abstracting common points in their interfaces. For example,

[1] The optimizations presented in this section were driven by Christos Baloukas during our joint works [BPP+10, AMP+15].

some tree data structures, such as ordered trees, behave basically as sets, while others encode more fundamental properties such as parent-child relations.

In the rest of this section, A_i-B_j represents the combination of the implementation i for A (the list of nodes in the DRR algorithm) and the implementation j for B (the queue of pending packets for each of the nodes). For example, A_1-B_3 represents that an array of pointers is used as implementation for the list of nodes in the DRR algorithm and a single-linked list is used as implementation for the queue of pending packets for each node.

To validate the chosen solution an exhaustive test over 13 000 combinations was performed: Ten possible choices per dynamic structure, thirteen input traces and ten repetitions per input – to tackle with statistical variations. However, such an exhaustive test is not typically required for practical cases. The results of these experiments reveal which are the most efficient data structures regarding total number of memory accesses and amount of memory (memory footprint) required to execute the application, as presented in the following paragraphs.

6.4.1. Reducing the number of memory accesses

The experimental results show that the most efficient combination of dynamic data structures is A_3-B_3, which means that a single-linked list implementation should be used for both the list of active nodes in DRR and the queue of packets waiting to be sent for each of the nodes (Figure 6.2). Table 6.2 shows the number of accesses required by the optimal solution for each input, and the number of memory accesses required by the selected solution, A_3-B_3. The forth column shows the difference between both: The average difference from A_3-B_3 to a hypothetical "perfect" solution is 0.3 %.[2]

6.4.2. Reducing memory footprint

In the case of memory footprint, we evaluated a Pareto optimal solution considering also the effect on the number of memory accesses. Table 6.3 shows the results obtained from the experiments. Although there is not a clear solution that reduces the memory footprint of the application for all cases, a situation which is clearly reflected in Figure 6.3, the solution selected in the previous step to reduce the number of accesses does not inflict a significant penalty on the total amount of memory required: The average difference from this solution to the optimal memory footprint solution (calculated on a case-by-case basis) is only 3.3 %.

6.5. Dynamic memory management refinement

Once the dynamic data structures of the application have been optimized and the corresponding metadata information updated, the next step is the optimization of the dynamic memory manager.[3] We refer to the dynamic memory manager as an integrated set of application specific dynamic memory managers (i.e., application level DM managers compiled with each software application). If more than one application are present, then each one is compiled with its own customized DM manager. The customized DM managers share common OS level

[2]This "perfect" solution is calculated by selecting the optimal DDT combination for each input case as if foreseen by an oracle. Then, the selected configuration, A_3-B_3 in this case, is compared against each of them to calculate how much it deviates for each input case.

[3]The optimizations presented in this section were performed by Alexandros Bartzas and myself based on previous work by David Atienza and Stylianos Mamagkakis [Ati05, AMP+15].

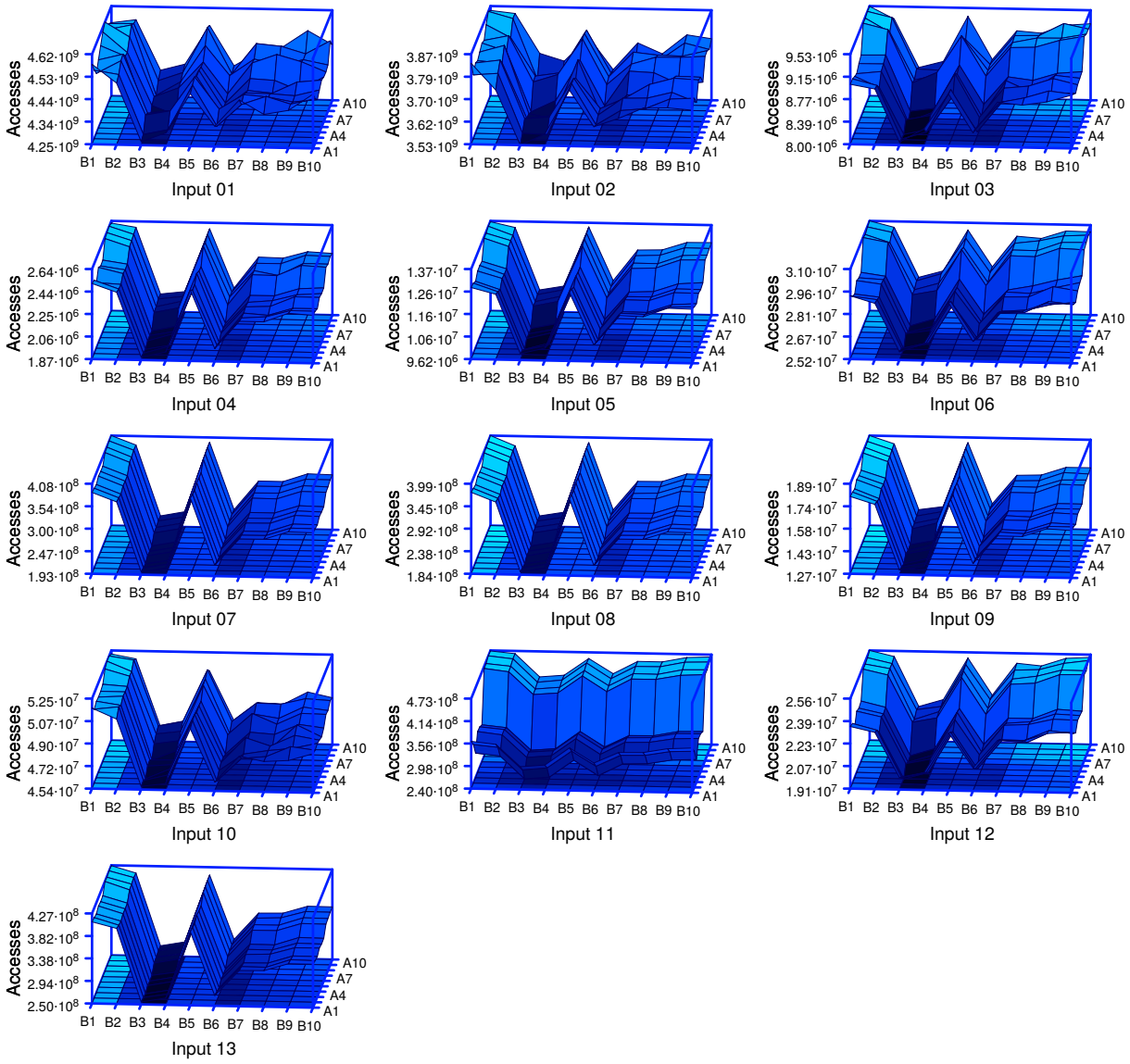


Figure 6.2.: Exploration of the number of memory accesses executed by the application with each combination of A_i-B_j data structures, for each of the thirteen input cases explored. The area corresponding to B_3 presents clearly a lower cost, with A_3 being a minima in most cases.

Table 6.2.: Comparison of the solution A_3-B_3 to a hypothetical "perfect" solution for each input case, in terms of memory accesses. The selected solution is optimal in 11 out of the 13 input cases considered and the average difference is 0.3 %.

Input set	Memory accesses A_3-B_3 ($\times 10^6$)	Memory accesses optimal per case ($\times 10^6$)	Difference %
01	4 328.50	4 247.38	1.9
02	3 589.77	3 530.75	1.7
03	8.00	8.00	0
04	1.87	1.87	0
05	9.62	9.62	0
06	25.23	25.23	0
07	192.87	192.87	0
08	183.64	183.64	0
09	12.70	12.70	0
10	45.43	45.43	0
11	239.76	239.76	0
12	19.08	19.08	0
13	250.19	250.19	0
Average			0.3

Table 6.3.: Comparison of the solution A_3-B_3 with a hypothetical "perfect" solution that minimizes the memory footprint for each input case.

Input set	Memory footprint A_3-B_3 KB	Memory footprint optimal per case KB	Difference %
01	8 816.8	8 781.5	0.4
02	7 697.9	7 688.0	0.1
03	1 932.0	1 767.4	9.3
04	437.3	437.3	0
05	969.3	931.3	4.1
06	2 127.9	1 977.7	7.6
07	182.1	181.5	0.3
08	170.2	167.4	1.7
09	1 552.3	1 497.1	3.7
10	3 238.8	3 194.6	1.4
11	742.5	700.1	6.1
12	754.1	715.3	5.4
13	460.2	445.7	3.2
Average			3.3

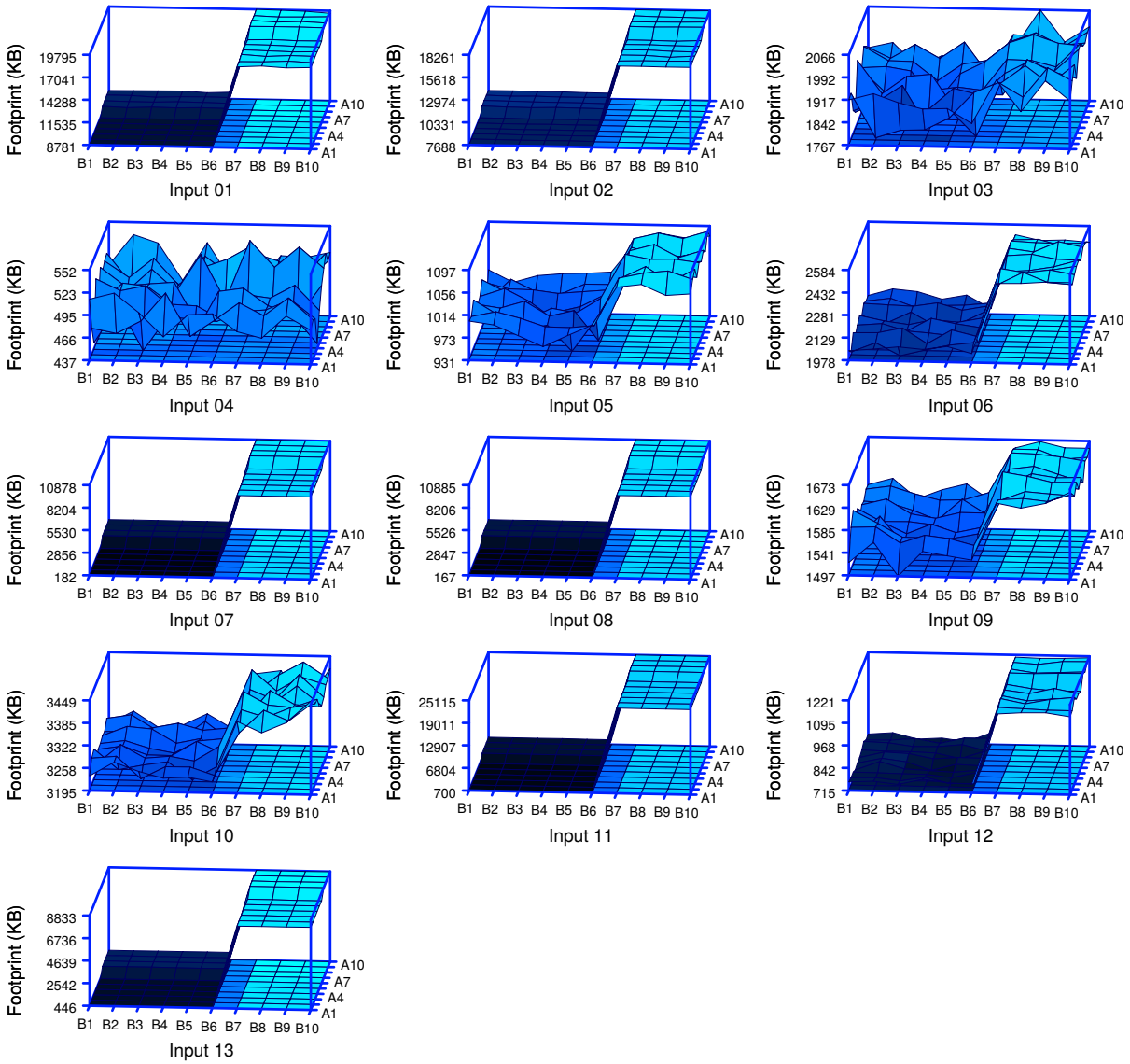


Figure 6.3.: Exploration of the memory footprint required by the application with each combination of A_i-B_j data structures, for each of the thirteen input cases explored. The larger inputs (longer network traces) behave well with several configurations (left-front quadrant), while smaller ones display a somewhat chaotic behavior (inputs 03 and 04).

services for providing big memory blocks (e.g., `sbrk` and `mmap`). The metadata information for this step includes the number of blocks of varying sizes allocated by the application for the network packets (data bodies and network headers independently), the list of destination nodes for the DRR algorithm and the queues of pending packets for each node.

This step is performed after the dynamic data structures have been optimized because the number of memory allocations and deallocations will not change anymore due to modifications in their implementation. Every time a new instance of any of these dynamic data types is created, the dynamic memory manager must search for a suitable free block, i.e., a block of sufficient size that spans over a range of memory addresses that are not currently occupied by any other object. Conversely, when an instance is destroyed, the memory space that it was using must be reintegrated to the pool of available addresses for new instances. These operations produce an overhead in the total amount of work that the system has to do. Additionally, the memory manager requires some extra memory for its own operation and the book-keeping of the whole process increases the total number of memory accesses. Therefore, the suitability of a given dynamic memory manager depends on the overhead that it imposes on the system and the extra memory needed due to the existence of internal and external fragmentation [WJNB95].

The size of the allocated memory blocks and their frequency of appearance, together with the pattern of allocations and deallocations, determine the characteristics of the most suitable memory allocator. These numbers are defined in the software metadata of the application (Dynamic Data and Pool entities). Figure 6.4 shows the frequency of appearance of each allocation size in the driver application, for the most popular sizes: The memory manager for this application must be primarily optimized to allocate large quantities of blocks from a small set of sizes.

In order to reduce the number of memory accesses, the memory manager has to locate the most appropriate free block with the least amount of memory accesses. For this reason, a memory manager that gets free space from a global pool at first, but then frees the blocks into lists of specific sizes is chosen. Additionally, internal fragmentation can be reduced creating lists of free blocks for a small number of additional sizes so that the amount of wasted memory for uneven sizes is limited. With the previous considerations, and the methods presented by Mamagkakis et al. [MAP+06], the design space of the dynamic memory manager may be narrowed to a few options:

Block sizes: Special memory block sizes equal to each packet size that represents at least 10 % of the overall packet sizes should be predefined. The rest of the predefined memory blocks should be power-of-two sizes up to the MTU size. In the example, this means the creation of blocks of 40 B, 1460 B and 1500 B in addition to blocks of 256 B, 512 B and 1024 B. In some of the managers considered in the experiments, two smaller application-specific block categories of 92 B and 132 B were added. With these block sizes, the most popular memory requests can be satisfied without any internal fragmentation and the remaining less popular requests can be satisfied with reasonable internal fragmentation. Additionally, having blocks of fixed sizes, predefined at compile time, gives the performance advantage of not having to calculate the corresponding block size for each request at run-time.

Some of the managers include a block for allocations of 0 B, used by the system at the TCP-level to send acknowledgment-only (ACK) packets when it receives big amounts of data without significant outbound traffic. This application-specific optimization im-

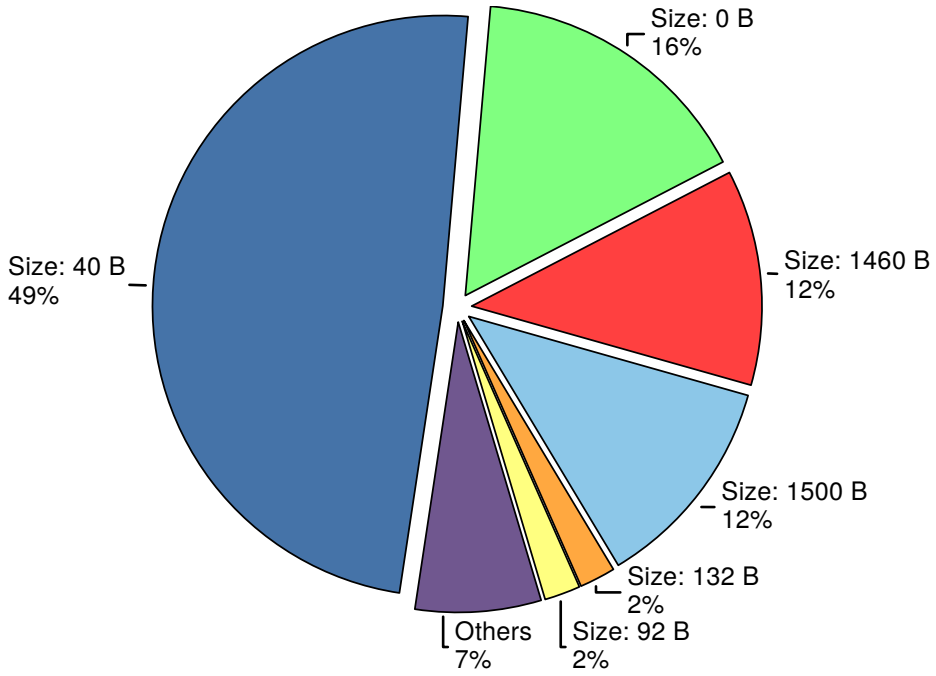


Figure 6.4.: Allocation sizes that represent each more than 1 % of the requests between all the experiments. The rest of allocation sizes, which add up to a total of 1481 different sizes, can be handled by a general DMM. However, that DMM has to be optimized for handling a varied range of sizes as it will still serve around a 7 % of the requests.

proves slightly the performance of the system. However, its impact is minor, so it can be removed if the resulting DMM behavior is not acceptable.[4]

Coalescing and splitting of blocks: Splitting and coalescing are computationally intensive processes that may slow down allocation and deallocation and incur a significant number of memory accesses to transform the old block sizes into the new ones. As the maximum requested block size in the subject application is already known (the MTU of the packets), there is no need to coalesce blocks to deal with external fragmentation. Additionally, defining block sizes that prevent most of the internal fragmentation (i.e., the internal fragmentation produced by the popular requests) reduces the need to split blocks. Consequently, no splitting or coalescing of memory blocks should be used for this application.

Pools: A number of pools equal to the number of the predefined block sizes should be created. In this example, one pool for each of the predefined block sizes is created. This organization allows a faster access to the specific memory pool that will service each request: In the worst case, the number of memory accesses that will be needed to find the pool that holds the block of the appropriate size is in the order of the number of pools. Finally, once the decision of not supporting coalescing nor splitting of blocks is

[4] According to the rules of the C/C++ programming language, an allocation of zero bytes is valid and must return a valid (not `NULL`) memory block. However, the actual size of the object is zero bytes and the application cannot access any bytes at that address. Indeed, previous versions of the C++ standard did not require that a distinct block was returned for zero-size blocks. Relying on this interpretation, an additional optimization is introduced in the memory manager to ease the allocation of zero-byte blocks: That the memory manager may use a fixed block to host all of these requests. This method could also be used to detect application errors during the developing phase by making this special block access-protected.

Table 6.4.: Configuration of the dynamic memory managers evaluated in this experiment.

DMM	Description
DMM 1	Kingsley-like [Mic04] memory manager with bins for blocks of 128 different sizes, from 8 B to 16 384 B. This popular memory manager is used throughout the rest of this section as reference for comparison with the customized dynamic memory managers.
DMM 2	Custom manager with lists of free blocks of sizes 40 B, 1460 B and 1500 B.
DMM 3	Custom manager with lists of free blocks of sizes exactly 40 B, exactly 1460 B, exactly 1500 B, up to 92 B, up to 132 B, up to 256 B, up to 512 B and up to 1024 B. The particular order and constrains of "up to" and "exactly" ensure that the most common allocation sizes require the minimum number of accesses to find a suitable block.
DMM 4	Custom manager with lists of free blocks of sizes 40 B, 1460 B and 1500 B, plus support for splitting and coalescing.
DMM 5	Custom manager with lists of free blocks of sizes 40 B, 1460 B, 1500 B, 92 B and 132 B (in order of search priority), plus support for splitting and coalescing.
DMM 6	Like DMM 2, plus special treatment for blocks of zero bytes (app. specific optimization).
DMM 7	Like DMM 3, plus special treatment for blocks of zero bytes (app. specific optimization).
DMM 8	Like DMM 4, plus special treatment for blocks of zero bytes (app. specific optimization).
DMM 9	Like DMM 5, plus special treatment for blocks of zero bytes (app. specific optimization).

made, the DMM does not need to support (performance-costly) movements of blocks between pools as their sizes change.

Fit algorithms: In order to choose a pool, the Exact Fit and First Fit algorithms are proposed. Exact Fit is used to discriminate between the special pools, while First Fit is used with the rest. Once a pool is chosen, only First Fit is used to choose the appropriate block. This decision is based on the existence of specific pools for the sizes that represent at least a 70 % of the memory requests as it requires the least accesses to find a suitable memory block.

The previous points are summarized in Table 6.4, which describes each of the dynamic memory managers evaluated in the experiments. Once we have explored the key parameters of the memory managers, reducing the design space to a manageable size, we can run several simulations to analyze the performance of each option in terms of number of memory accesses and memory footprint. A reduced design space allows performing an exhaustive analysis and obtaining the most suitable customized dynamic memory manager.

A relevant question that arises at this point is: "How much is the impact of the dynamic memory manager on the total number of memory accesses of the application?" To justify the importance of dynamic memory management optimizations, Figure 6.5 shows the weight of the memory accesses[5] due to the dynamic memory management over the total number of memory accesses, for each of the memory managers used in this experiment. The dynamic memory managers 2, 3, 6 and 7 require a relatively low number of memory accesses to perform their work. On the contrary, DMMs 4, 5, 8 and 9 introduce more than one third of the memory accesses required by the application just to manage the dynamic memory. DMM 1, the design used as reference through this experiment, introduces a moderate 10 % of additional accesses.

[5]This weight is calculated considering the number of memory accesses logged between the `MallocBegin` and `MallocEnd` profiling tokens, and between `FreeBegin` and `FreeEnd`, which correspond to the work done by the memory manager to handle each application request. The weight, i.e., the fraction of accesses that correspond to the overhead of doing memory management, is obtained dividing this number by the total number of memory accesses.

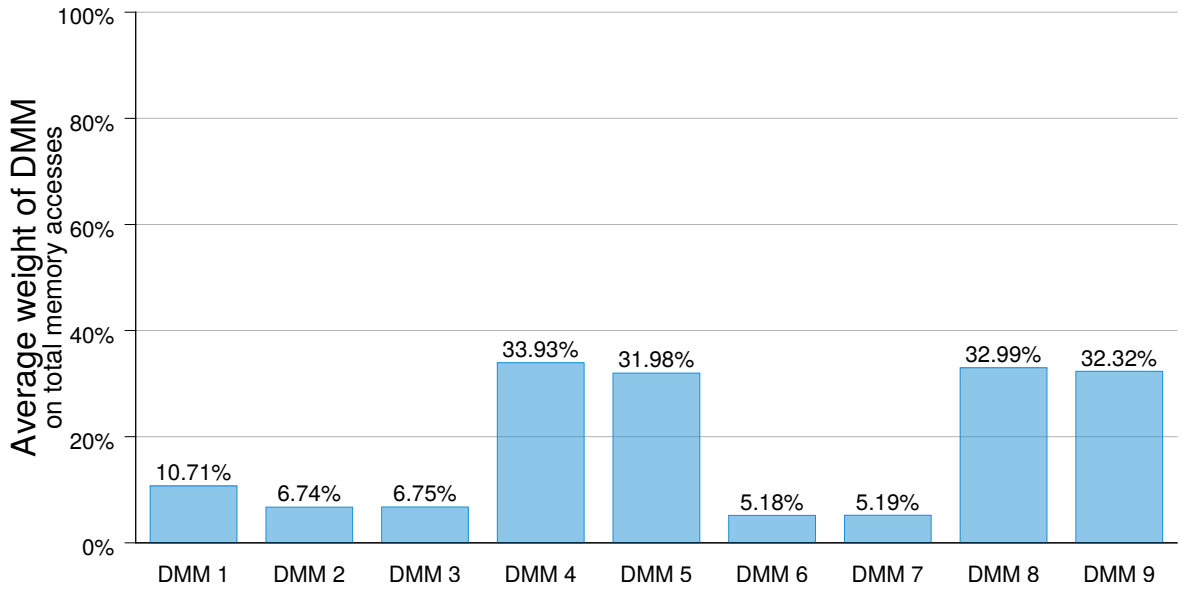


Figure 6.5.: Percentage of memory manager accesses over total application accesses.

Figures 6.6 and 6.7 present a more detailed analysis for the most promising managers, DMM 6 and DMM 7, and the reference one, DMM 1. The first figure gives an idea of proportion showing the total amount of memory accesses executed during each input case. The second one reveals that for the data intensive cases, where mostly big packets are sent and the work required to process the data supersedes the work needed to allocate the blocks, the number of accesses due to dynamic memory management represents less than 1 % of the total (inputs 01 and 02 in both figures). However, when the system sends many small packets (e.g., small TCP segments may be sent due to the necessity of bounding delays under low application traffic), the number of accesses due to memory management can scale up to a 24 % of the total (for input number 8 using DMM 1). Interestingly, the overhead of DMMs 6 and 7 is approximately half of that, underlining the importance of optimizing the performance of the dynamic memory manager.

6.5.1. Reducing the number of memory accesses

Having as target the minimization of memory accesses, the dynamic memory manager needs to be able to fetch blocks for the most popular sizes quickly. Similarly, the process of marking as free the blocks that become unused must be quick. This suggests the specialization of lists of free blocks for popular sizes. However, using too many lists increases searching times; the set of lists must hence be reduced to the ones that receive most of the allocations.

Table 6.5 shows the difference in the number of memory accesses directly related to the management of dynamic memory to the optimal solution for each input case. DMMs 6 and 7 are the memory managers that incur the lowest overhead, with DMM 6 being the best one for most of the input cases. Compared to DMM 1, DMM 6 and DMM 7 obtain an improvement of up to 62.35 % and 62.33 %, respectively (for input 05). On average, both managers enable a reduction of 45.67 % and 45.62 % of the memory accesses due to the management of dynamic memory, respectively (Figure 6.8).

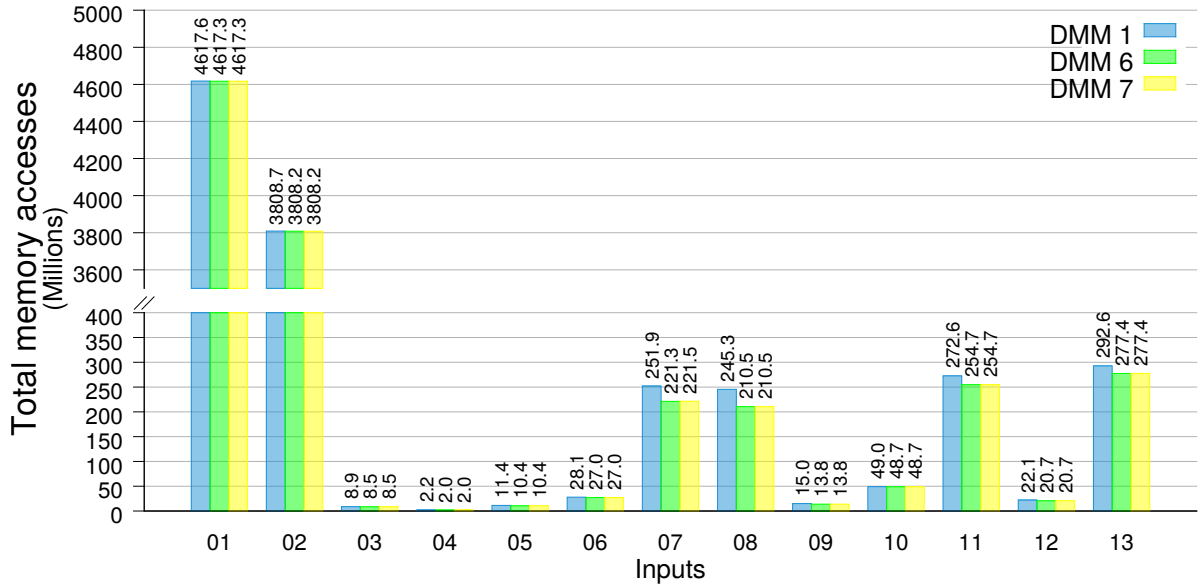


Figure 6.6.: Total number of memory accesses for each input.

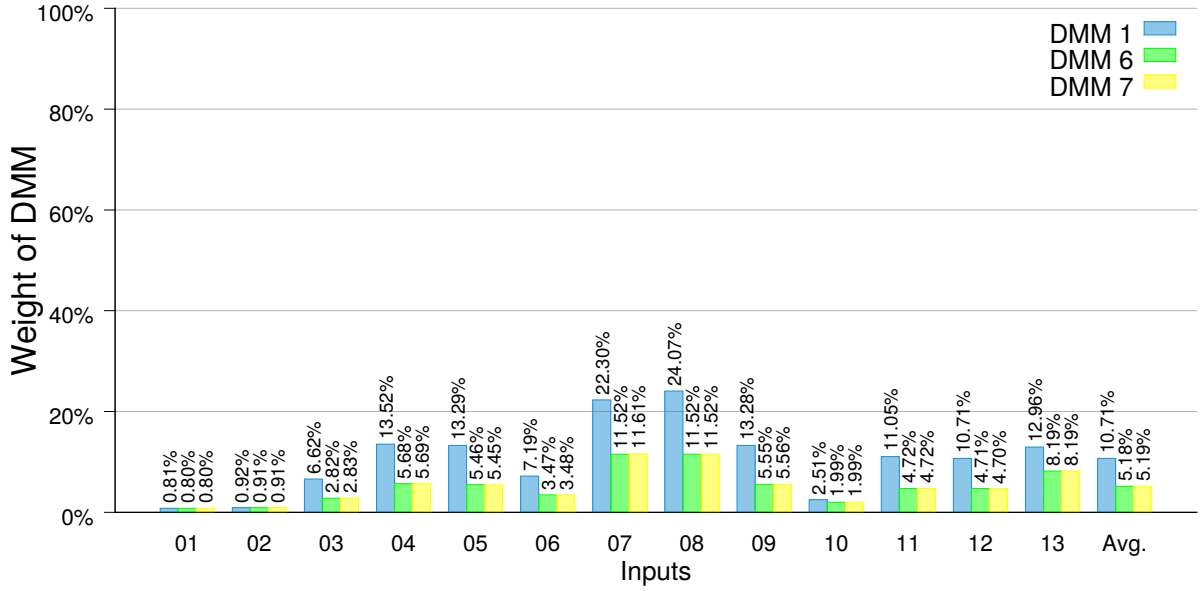


Figure 6.7.: Impact of dynamic memory management on the total number of memory accesses: Percentage of the total memory accesses that correspond to the dynamic memory manager.

Table 6.5.: Difference in memory accesses (during DMM operations only) between each DMM and the optimum for each input. DMM 6 and DMM 7 cause the lowest overhead.

Input	DMM1 %	DMM2 %	DMM3 %	DMM4 %	DMM5 %	DMM6 %	DMM7 %	DMM8 %	DMM9 %
01	0.67	0.01	0.01	5 024.59	4 881.74	0	0	4 986.61	4 454.37
02	1.30	0.18	0.19	2 240.55	2 220.80	0	0.01	2 268.15	2 239.02
03	144.17	26.92	25.39	720.54	735.17	0	0.29	547.52	572.22
04	159.45	32.33	33.54	744.68	730.43	0	0.04	523.58	514.36
05	165.61	43.62	43.37	844.33	862.57	0	0.05	650.53	698.31
06	115.39	27.89	28.19	865.83	868.86	0	0.16	598.72	629.96
07	120.39	35.03	35.96	551.71	552.20	0	0.94	590.30	614.06
08	143.51	42.13	42.13	572.62	572.98	0	0	566.18	567.98
09	160.48	42.85	42.97	820.09	820.18	0	0.12	651.85	575.89
10	27.02	0.07	0.06	1 620.66	1 481.76	0.03	0	1 420.79	1 427.60
11	150.39	43.99	43.97	793.95	938.43	0.02	0	1 080.67	931.31
12	142.78	37.94	38.03	834.40	835.10	0	0	806.23	962.48
13	66.95	19.41	19.48	1 916.14	571.26	0	0.08	2 863.90	1 968.18
Avg.	107.55	27.10	27.17	1 350.01	1 236.27	$<$ **0.01**	**0.13**	1 350.39	1 242.75

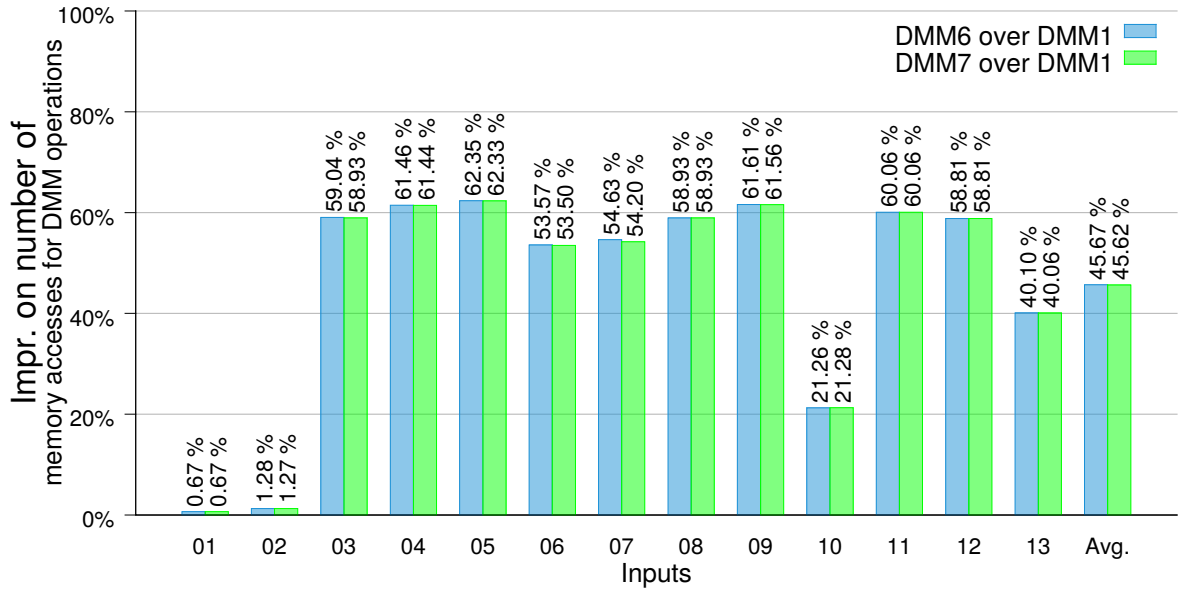


Figure 6.8.: Improvement on the number of memory accesses due to DMM when using DMM 6 or DMM 7 instead of DMM 1.

Table 6.6.: Difference between the total number of memory accesses caused by the dynamic memory managers and the optimum for each input. DMM 6 and DMM 7 minimize the number of memory accesses of the whole application.

Input	DMM1 %	DMM2 %	DMM3 %	DMM4 %	DMM5 %	DMM6 %	DMM7 %	DMM8 %	DMM9 %
01	0.01	< 0.01	< 0.01	40.41	39.26	0	< 0.01	40.10	35.82
02	0.01	< 0.01	< 0.01	20.38	20.20	0	< 0.01	20.63	20.36
03	4.11	0.76	0.73	20.38	20.74	0	0.03	15.46	16.18
04	9.10	1.84	1.95	42.34	41.53	0	< 0.01	29.76	29.24
05	9.06	2.43	2.41	46.09	47.11	0	0.07	35.56	38.13
06	4.05	1.01	1.02	30.07	30.19	0	0.06	20.81	21.88
07	13.87	4.03	4.14	63.55	63.61	0	0.11	68.00	70.74
08	16.53	4.85	4.85	65.94	65.99	0	< 0.01	65.20	65.41
09	8.93	2.40	2.42	45.54	45.53	0	0.03	36.19	31.98
10	0.54	< 0.01	< 0.01	32.22	29.46	< 0.01	0	28.25	28.39
11	7.04	2.04	2.08	37.47	44.36	0	0.02	50.99	44.03
12	6.75	1.91	1.88	39.39	39.42	0	0.12	38.01	45.30
13	5.48	1.59	1.60	156.88	46.77	0	0.01	234.48	161.15
Average	6.57	1.76	1.78	49.28	41.09	$< \mathbf{0.01}$	**0.03**	52.57	46.82

Reducing the number of memory accesses due to DMM is important, but the final goal is to reduce the total number of memory accesses of the application. Table 6.6 shows the difference in the total number of memory accesses when using each one of the managers compared to the optimal for each input case. The results are consistent and, again, DMMs 6 and 7 enable the biggest reduction in memory accesses, with a negligible average deviation to the optimal for each input case. In a similar way with the previous analysis, Figure 6.9 illustrates the improvement achieved by these two dynamic memory managers (up to 14.18 % for input 08 and 5.98 % on average).

The small difference between DMM 6 and DMM 7 is studied further below. For now, Figure 6.10 shows a closer comparison between them for each input case.

6.5.2. Reducing memory footprint

Using less memory is beneficial for an application running in an embedded system in many ways. For example, the system may use smaller memory modules, which will translate into smaller access times and lower energy consumption. The system manager may even be able to power down unused memory modules to further reduce energy consumption. Thus, reducing the memory footprint of the application is an important optimization goal.

In order to reduce the memory footprint of the application, the dynamic memory manager must ensure that the amount of memory wasted due to internal and/or external fragmentation is kept to a minimum. Therefore, each allocation must be satisfied with a block of the right size. With that purpose, DMM 6 builds lists of free blocks for the most popular sizes (40 B, 1460 B and 1500 B) to reduce the number of accesses needed to find the right block. DMM 7 adds some less frequently requested sizes (92 B and 132 B) and several intermediate ones (256 B, 512 B and 1024 B).

We performed several experiments to validate the previous assumptions based on the extracted software metadata. Table 6.7 and Figure 6.11 show the amount of memory required by

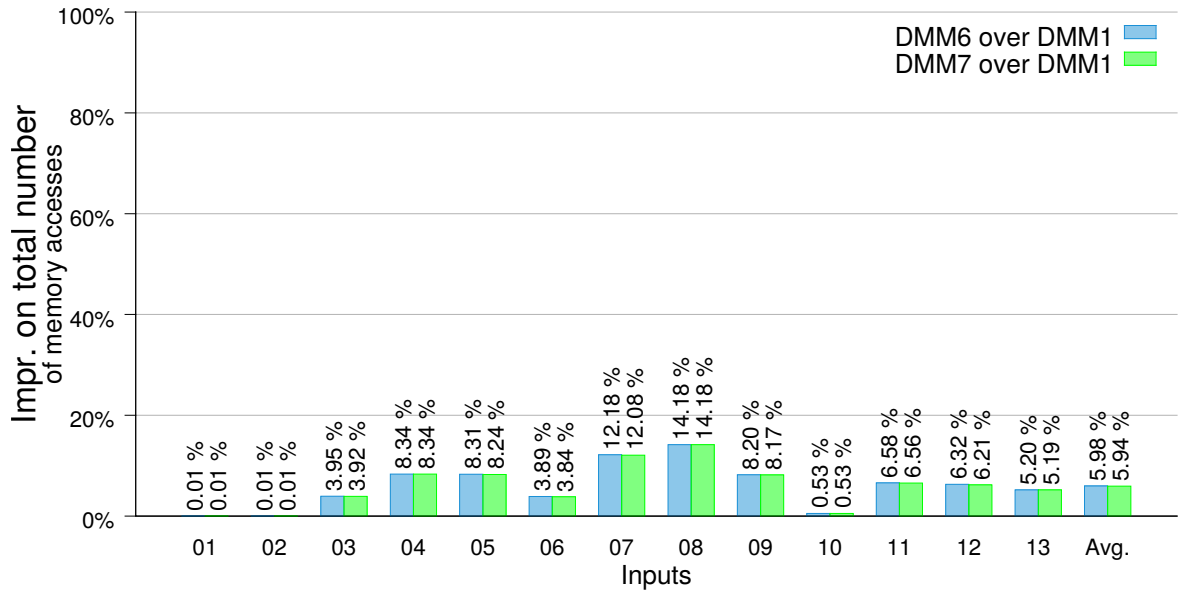


Figure 6.9.: Improvement on the total number of memory accesses due to the optimizations on the dynamic memory manager: Improvement of DMM 6 and DMM 7 over the reference memory manager (DMM 1).

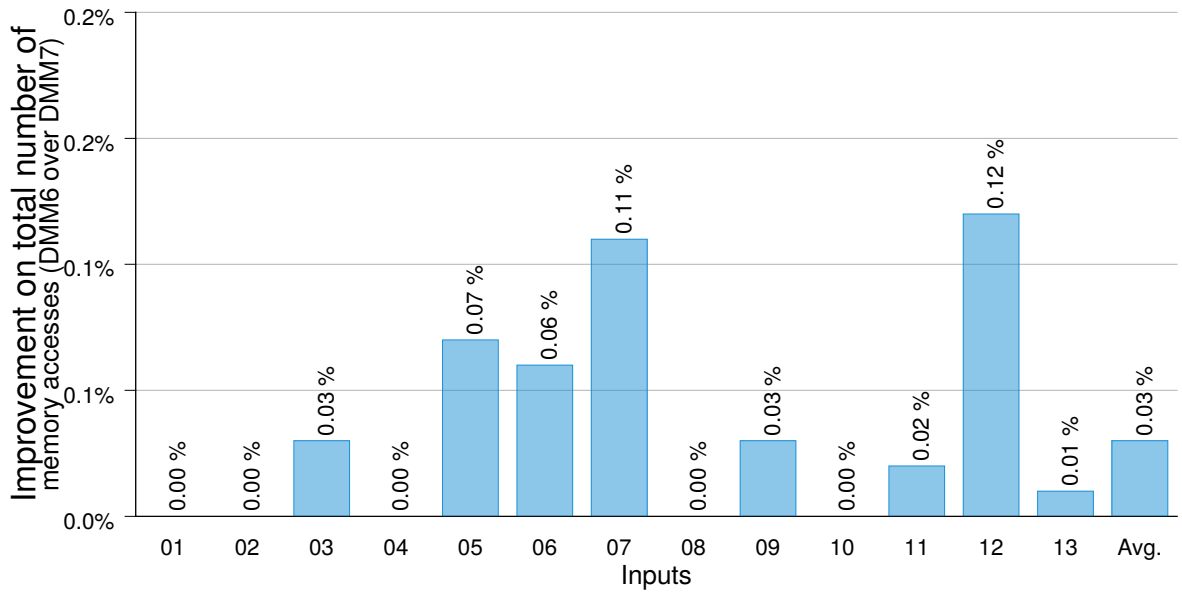


Figure 6.10.: Improvement on the total number of memory accesses due to the optimizations on the dynamic memory manager: Marginal improvement of DMM 6 over DMM 7 for the total number of memory accesses.

Table 6.7.: Total memory footprint (in KB) required by the different dynamic memory managers for each input case.

Input	DMM1	DMM2	DMM3	DMM4	DMM5	DMM6	DMM7	DMM8	DMM9
01	9 075.15	8 943.41	8 964.25	9 756.26	10 316.62	8 943.89	8 961.66	9 835.48	10 409.20
02	8 068.04	7 924.45	7 955.38	29 633.12	35 829.05	7 880.72	7 910.97	35 913.42	30 976.13
03	2 270.34	3 028.52	1 909.95	3 064.91	3 011.87	3 116.36	2 091.08	2 449.24	2 606.16
04	570.04	1 043.37	774.56	786.98	828.40	838.53	623.28	551.12	604.13
05	1 993.79	1 505.62	1 505.40	3 620.81	3 505.96	1 211.91	1 200.36	1 315.05	1 204.08
06	3 571.63	4 284.45	3 224.96	9 069.78	9 274.85	4 346.15	2 888.16	7 689.12	7 227.19
07	14 168.95	479.83	477.37	3 936.48	3 892.20	257.79	255.18	38 715.15	36 439.26
08	16 164.47	523.10	498.34	3 982.00	3 853.69	273.12	243.17	25 756.88	25 644.59
09	2 227.02	2 516.02	2 204.83	4 675.45	4 823.03	2 085.91	1 893.22	2 954.19	3 317.31
10	3 361.26	8 858.00	6 225.81	3 405.32	3 852.03	8 851.98	6 114.12	3 429.01	3 428.41
11	9 254.05	2 176.24	1 231.79	62 717.53	57 140.38	1 900.88	955.59	9 849.13	13 369.45
12	1 913.40	1 991.04	1 384.26	6 308.49	6 445.67	2 044.45	1 096.41	1 544.04	876.05
13	7 487.35	4 773.76	738.74	2 408.00	2 423.05	4 720.85	586.43	4 533.21	7 625.96

the application for each input dataset with each of the different dynamic memory managers. Table 6.8 presents the deviation of each DMM in respect to the optimum for each input case. Considering memory footprint, DMM 7 is the best memory manager for most of the input cases. However, the difference to a hypothetical perfect solution for all cases is now bigger with a maximum of 81.9 % (for input 10), although the average deviation is 10.0 %. On the contrary, the average difference of DMM 6 to the optimum is now considerably higher. The higher variation of the results obtained for memory footprint, compared to the results obtained for the number of memory accesses, is due to the fact that the memory footprint is more sensitive to variations in the number of blocks of each size that are allocated, even if the number of accesses to find the blocks remains similar.

Figure 6.12a shows again the difference in the memory footprint required by the application when using each of the dynamic memory managers in comparison with the optimum, but now averaged for all the input cases. Finally, Figure 6.12b shows a related measurement with an interesting twist: The average overhead in memory footprint that each dynamic memory manager imposes. The application requires a certain amount of memory, but the dynamic memory manager needs some extra memory for its own internal structures. Additionally, the effects of internal and external fragmentation increase the actual amount of memory needed to serve the needs of the application. The figure shows that for the best suitable memory manager, DMM 7, the overhead is only 40.26 % (compared to 1562.44 % of DMM 1, which is a general purpose memory manager). However, if the memory manager used is not fine-tuned to the allocation behavior of the application, the overhead can increase up to 2996.77 % (on average, when using DMM 8).

With the previous results, it seems logical to employ DMM 7 as the final memory manager for the application. However, in Section 6.6, the effect of both dynamic memory managers is analyzed independently when applying the optimizations on the transfer of blocks of dynamic data. Should there appear relevant differences between the behaviors of both memory managers, then one might be chosen when trying to reduce the number of memory accesses and the other when trying to reduce memory footprint. Otherwise, DMM 7 may be used for all cases.

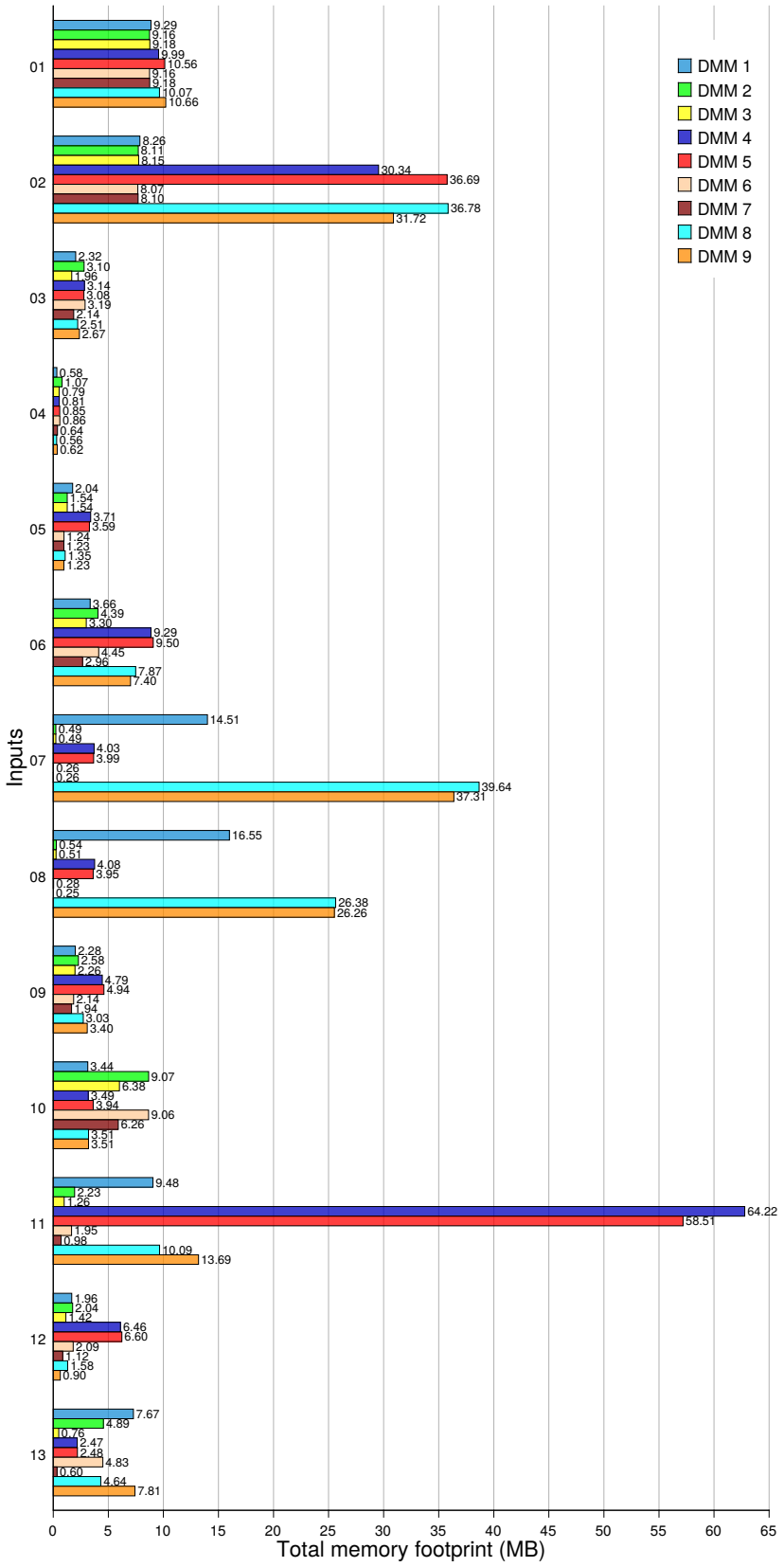
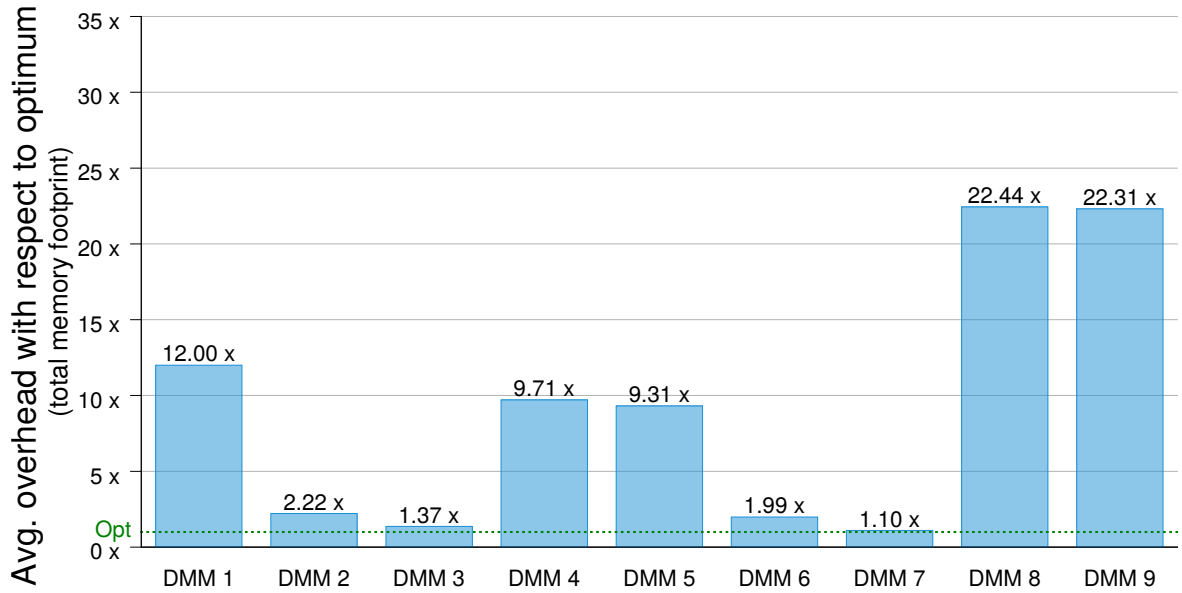
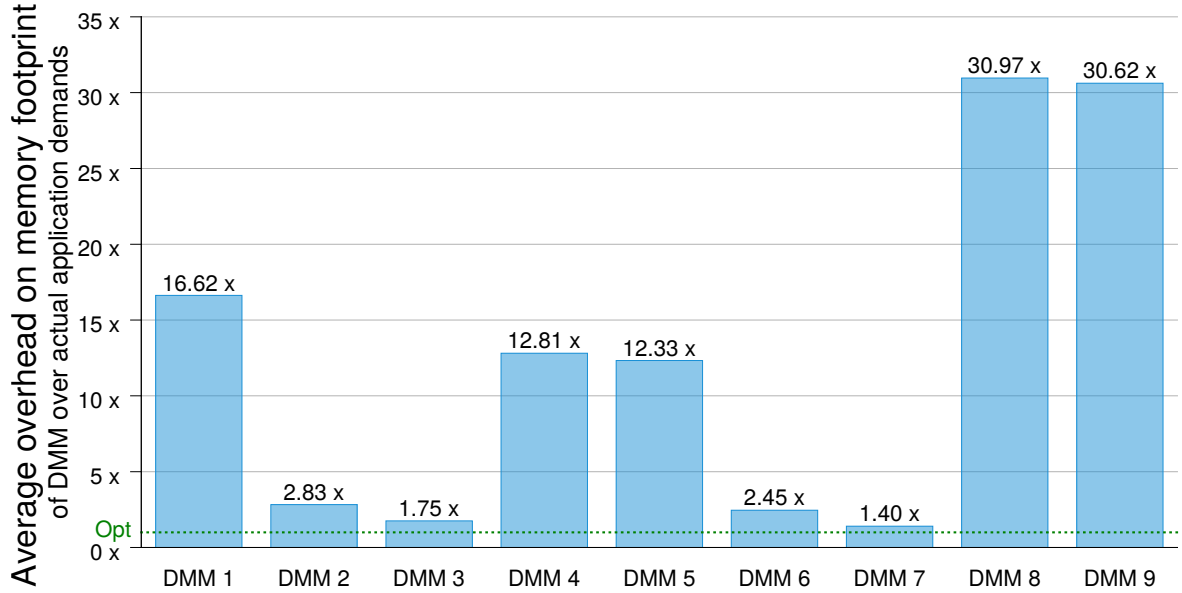


Figure 6.11.: Total memory footprint required by each of the dynamic memory managers for each input case.



(a) Average increase of total memory footprint using each memory manager in comparison with the optimum for each input. Although DMM 7 is not the best solution for all the input cases, it is the best overall one.



(b) Average overhead in terms of memory footprint of each dynamic memory manager.

Figure 6.12.: Analysis of memory footprint with each dynamic memory manager.

Table 6.8.: Difference to the optimum for each input case of the total memory footprint required by the different dynamic memory managers. DMM 7 minimizes, on average, the difference to the optimum.

Input	DMM1 %	DMM2 %	DMM3 %	DMM4 %	DMM5 %	DMM6 %	DMM7 %	DMM8 %	DMM9 %
01	1.5	0	0.2	9.1	15.3	0	0.2	10.0	16.4
02	2.4	0.6	0.9	276.0	354.6	< 0.1	0.4	355.7	293.1
03	18.9	58.6	0	60.5	57.7	63.2	9.5	28.2	36.4
04	3.4	89.3	40.5	42.8	50.3	52.1	13.1	0	9.6
05	66.1	25.4	25.4	201.6	192.1	1.0	0	9.5	0.3
06	23.7	48.3	11.7	214.0	221.1	50.5	0	166.2	150.2
07	5 452.4	88.0	87.1	1 442.6	1 425.3	1.0	0	15 071.5	14 179.6
08	6 547.4	115.1	104.9	1 537.5	1 484.8	12.3	0	10 492.1	10 445.9
09	17.6	32.9	16.5	147.0	154.8	10.2	0	56.0	75.2
10	0	163.5	85.2	1.3	14.6	163.3	81.9	2.0	2.0
11	868.4	127.7	28.9	6 463.2	5 879.6	98.9	0	930.7	1 299.1
12	118.4	127.3	58.0	620.1	635.8	133.4	25.1	76.2	0
13	1 176.8	714.0	26.0	310.6	313.2	705.0	0	673.0	1 200.4
Avg.	1 099.8	122.4	37.3	871.3	830.7	99.3	**10.0**	2 144	2 131.4

6.6. Dynamic memory block transfer optimization

The traditional application of the DMA module is to transfer data in the memory subsystem (or between it and the external devices), freeing up processor cycles. However, in embedded systems it is also common practice to use the DMA module to reduce the average latency to access data from main memory; for example, copying data to closer memories before the CPU actually needs them [DBD+06].

On the other hand, DRAM is usually the memory technology chosen to implement the main memory of embedded systems. These devices are internally organized in banks and rows, with the restriction that only one row can be active in each bank at any given time. Changing the active row in a DRAM bank has a non-negligible cost in terms of cycles and energy consumption. This type of organization favors sequential access patterns. However, when the DMA uses the main memory in parallel with the processor, accesses from both interleave in an undetermined way.

Therefore, two relevant optimization goals for applications running on embedded systems are the efficient scheduling of data transfers for blocks of dynamic data using the DMA module and the right interleaving of accesses from the processor and the DMA to avoid unnecessary row activations in the banks of the DRAM modules. This section evaluates some optimization techniques for the transfer of dynamic data blocks based on the information supplied by the software metadata.

The software metadata contains information on the number of block transfers that involve instances of dynamic data types, their length, direction (to or from main memory) and the thread that initiated them – mainly, the Block Transfer entity from Figure 5.2. This information facilitates improving the utilization of the DMA module for the driver application of this example. If the input case produces long series of sequential accesses (i.e., the system processes mainly long packets), they are good candidates to be executed by the DMA. On the contrary,

if the system has to process many small packets, the overhead of programming the DMA may be higher than the number of cycles required by the processor to transfer the data itself.

The scheduling of accesses to dynamic data types must also consider two additional circumstances. First, when a block transfer is executed by the DMA in parallel with the processor, the external DRAM modules receive two simultaneous streams of accesses that may force extra row activations. Second, the DMA module can benefit itself from the lower latency of the DRAM burst modes. Thus, the trade-off between ensuring that the DMA can issue efficient burst transfers and guaranteeing that the processor can access the memory in a bounded number of cycles needs to be evaluated.

Taking these considerations into account, we considered three different scheduling policies for this experiment. The first one executes all the accesses to dynamic memory in the processor. The second one uses the DMA module for blocks of more than 32 B (8 words), but the maximum number of cycles that the DMA engine may hold the bus during burst transactions is limited to eight words; once the DMA is granted access to the bus, it can transfer without interruptions at least as many bytes as the shortest transfer. Finally, the third configuration employs the DMA module for transfers of at least 32 B, but ensures that the DMA may access up to a full DRAM row in a single burst transaction to maximize efficiency. This last policy uses the techniques presented by Peón-Quirós et al. [PBM+07] and Bartzas et al. [BPM+08] – namely monitoring the type of packets processed by the system and the mean length of data transfers – to decide whether to use the DMA module or not. We refer to these policies as "No DMA," "DMA Bad" and "DMA Opt," respectively.

The results of this experiment reveal that the utilization of the DMA module can save a considerable amount of processor cycles (43 % on average when using the optimal DMA configuration with DMM 7), but only if the DMA is used appropriately. Otherwise, it may have a significantly negative impact on the energy consumption of the memory subsystem, memory average latency and processor cycles wasted waiting to access the memory.

Analysis of the improvements achieved

Figure 6.13 shows the effect of a good scheduling: Using the right configuration, an improvement of up to 33 % in the number of cycles that the processor spends accessing memory may be achieved in comparison with no using the DMA at all. There is also a small improvement in energy consumption.[6] Moreover, compared with a bad DMA configuration that does not limit sufficiently the interferences between the two elements, average improvements of 24 % for the number of DRAM row activations, 14 % for the number of processor cycles spent accessing memory and 9 % for energy consumption are possible. These values are calculated using the reference dynamic memory manager (DMM 1).

Figure 6.14 shows the impact that the optimizations performed on the dynamic memory manager have on the performance of the final system. First, in Figure 6.14a the best DMA configuration is used in combination with the dynamic memory managers selected in the previous step, DMM 6 and DMM 7. The obtained performance results are compared against the ones obtained with the reference manager (DMM 1). The achieved improvements are about

[6] This improvement refers to the energy consumption in the memory subsystem. It is small because the processor and the DMA access the DRAM concurrently and, as the graph shows, there is a slight increase of row activations. The penalty in energy consumption of these additional row activations masks the benefits obtained by using the DMA. However, the total energy consumption of the system may be reduced much more because first, the DMA is more efficient accessing the memory than the processor and, second, the number of cycles that the processor spends accessing the memory is reduced, potentially allowing it to finish other tasks sooner.

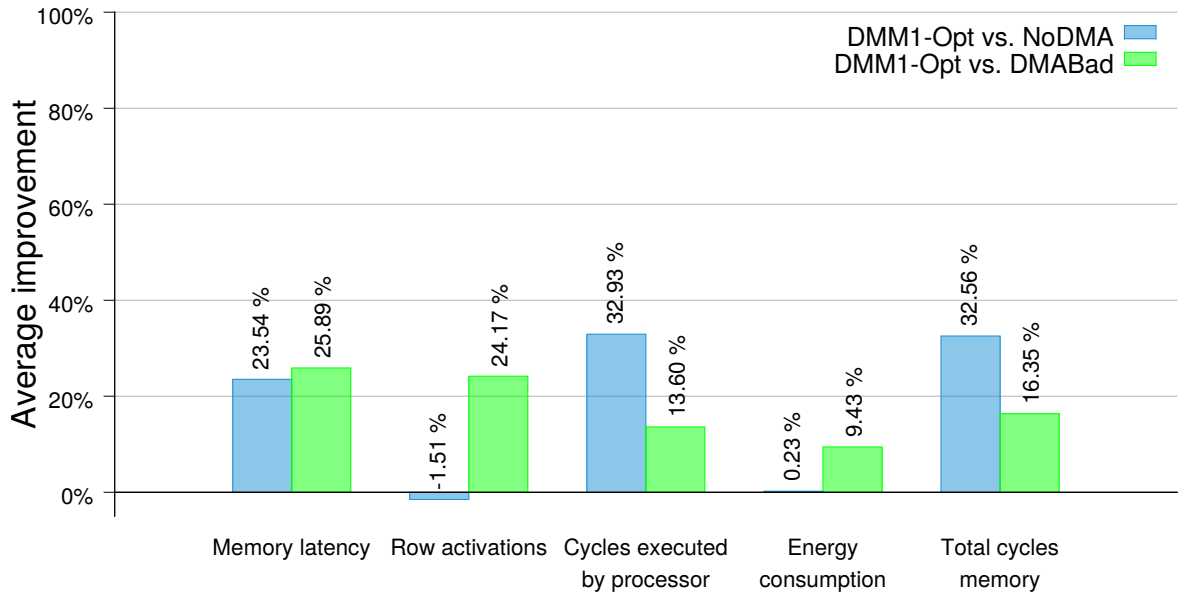
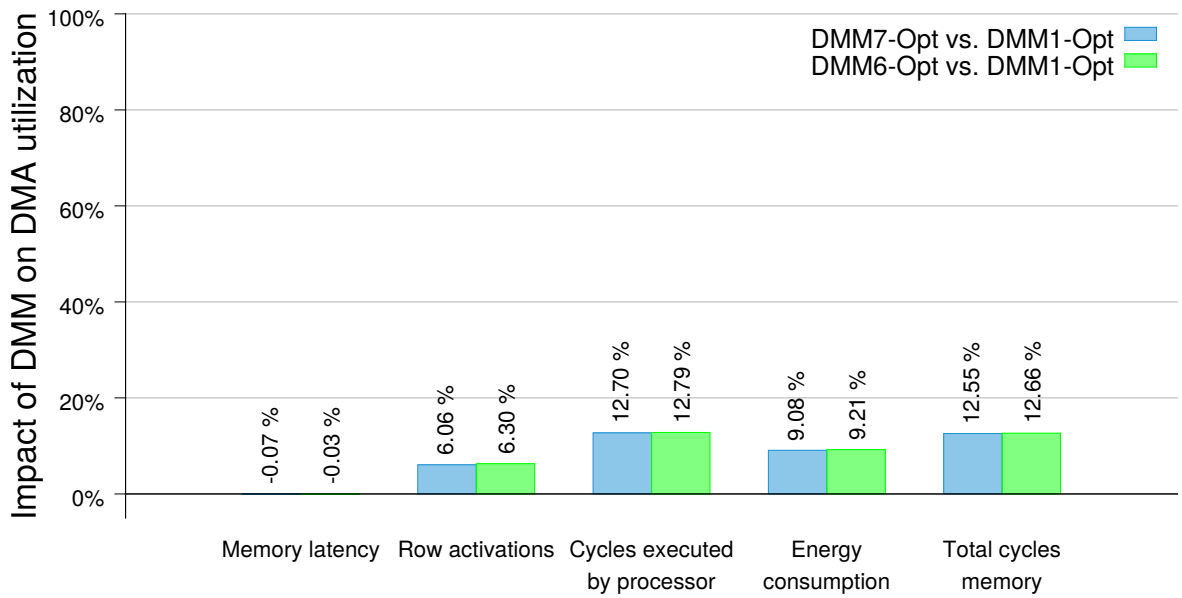


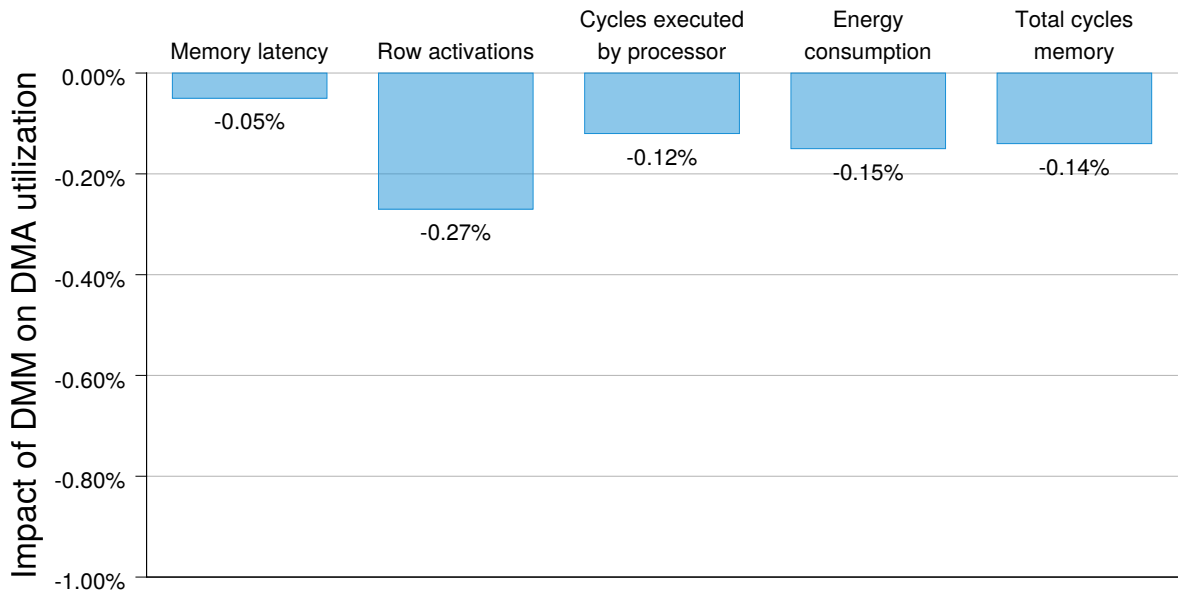
Figure 6.13.: Improvement achieved using a DMA module for block transfers of dynamic data. The leftmost bars (blue) show the improvements attained with the correct DMA configuration compared to not using the DMA. The rightmost bars (green) show the improvements of using the correct DMA configuration, in comparison with using a wrong one that does not limit the interference between DMA and processor. Specifically, a wrong DMA configuration may increase the number of DRAM row activations and hence, energy consumption.

6 % on the average number of DRAM row activations, 13 % on the number of cycles spent by the processor accessing the memory and 9 % in the energy consumption of the memory subsystem. Second, Figure 6.14b presents a direct comparison between DMM 6 and DMM 7. Just as a brief reminder, the outcome of the dynamic memory management optimization step was that DMM 7 is the best average solution when memory footprint is considered, and DMM 6 when considering the number of memory accesses. The difference between both of them when considering the number of memory accesses was negligible. Nevertheless, we kept around both dynamic memory managers just for the sake of analyzing their impact on the final configuration. The figure shows that both managers have a very similar effect on the behavior of the whole memory subsystem, with the bigger impact being lower than 0.3 % (for the number of DRAM row activations). Therefore, we can conclude that DMM 7 is the memory manager that should always be used in this system.

Finally, Figure 6.15 shows the overall improvements attained using DMM 7 with the optimal DMA configuration, in comparison with using the reference DMM 1 and each of the three DMA configurations analyzed: Up to 43 % of the processor cycles are now free to be used for any purpose other than accessing dynamic data from the main memory, and a mean reduction of 9 % in the energy consumption of the memory subsystem can also be obtained. Moreover, compared to a wrong configuration of the DMA module, a 29 % reduction on the number of DRAM row activations and 18 % on the energy spent in the memory modules is possible.



(a) Selecting an optimized dynamic memory manager has a significant impact on the performance of the DMA module, even when the best utilization policy is employed.



(b) Detail: Comparison between the effect of DMM 6 and DMM 7 on DMA performance. Both yield almost identical results.

Figure 6.14.: Impact of DMM selection on DMA performance.

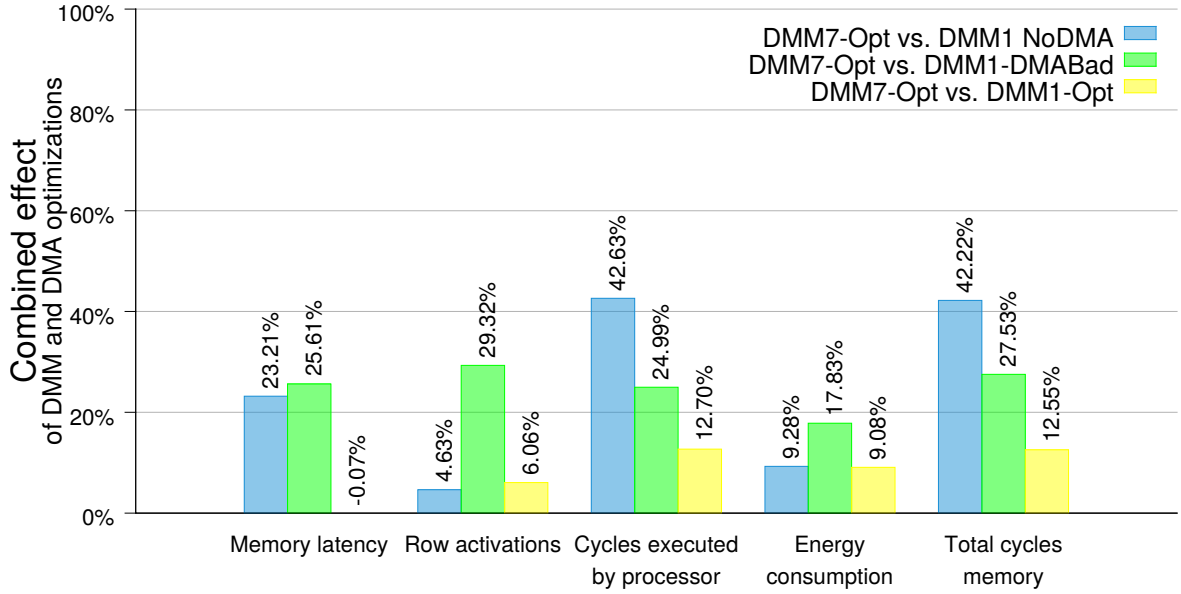


Figure 6.15.: Final combined effect of DMM and DMA optimizations.

Chapter 7

Conclusions and future work

"It is inconceivable, that inanimate brute matter, should, without the mediation of something else, which is not material, operate upon and affect other matter without mutual contact. That Gravity should be innate, inherent and essential to matter so that one body may act upon another at a distance thro' a vacuum without the mediation of anything else, by and through which their action and force may be conveyed, from one to another, is to me so great an absurdity that I believe no Man who has in philosophical matters a competent faculty of thinking can ever fall into it. Gravity must be caused by an agent acting constantly according to certain laws; but whether this agent be material or immaterial, I have left to the consideration of my readers."

— SIR ISAAC NEWTON, Mathematical Principle of Natural Philosophy
Translation by Andrew Motte (1729)

Once I have presented the methodology, tools and experiments done during this research work, in this chapter I draw its main conclusions and contributions, including the publications derived from it and, finally, I discuss some promising ideas for future research.

7.1. Conclusions

Thorough all this text I have tried to motivate the need for specific techniques for the placement of dynamic data objects in systems with heterogeneous memory organizations and the important benefits that they can bring. In particular, I have advocated a static and exclusive placement that avoids data movements, implemented through the dynamic memory manager. I have supported my claims with the formulation of a methodology and its implementation as a working tool that I use to analyze the important benefits that can be obtained in three extensive case studies. In summary, the main conclusions of the research reported in this work are:

- Applications that use dynamic memory and dynamic data structures typically have low access locality, which hinders the performance of cache memories. Specific techniques for dynamic-data placement that limit or avoid data movements can improve the performance and reduce the energy consumption of these applications, especially when they are executed on systems with heterogeneous memory organizations.

- Balancing between exclusive assignment of resources and resource exploitation is important. Otherwise, data movements would be saved at the price of wasted resources.
- The problem of dynamic data placement is complex. To tackle with that complexity, I propose a methodology that divides it in two phases: Grouping of DDTs with similar characteristics and mapping of those groups into actual memory resources.
- The grouping step is a trade-off that provisions dedicated space for the instances of the most accessed DDTs of the application, while limiting resource underuse. It combines DDTs whose instances present equivalent access characteristics, or that have complementary footprint demands so that when the space is needed for highly accessed instances most of the less accessed ones have already been destroyed and the space is again available.
- The second phase, mapping of the generated groups of DDTs into memory resources, is platform-dependent and computationally simpler because it is a particularization of the integer knapsack problem. Thus, this step may be delayed until run-time to automatically configure the system for various platform configurations or to achieve graceful performance degradation as it ages and some components start to fail.
- The dynamic memory manager (DMM) can be the means to implement the generated placement solutions. To carry out this new responsibility, the DMM requires extra information: The type of the data object involved in each allocation operation.
- The methodology requires an extensive characterization of the application, so a preliminary profiling phase is required. The same instrumentation can also supply the information needed by the dynamic memory manager; hence, the effort is shared for both purposes.
- The results of the experiments show that the specific placement of dynamic data objects produced by the methodology obtains clear improvements in performance and energy consumption in comparison with the utilization of traditional cache memories, without adding expensive HW or SW requirements.
- The reason of the improvements is not only that SRAMs are usually more efficient than caches built with the same technology, but also a reduced number of memory accesses.
- The importance of this work is not so much on the concrete algorithms that I have presented as on the necessity of considering data placement at all the abstraction levels, from the nodes of a linked list in a simple application to complex data repositories in dedicated rack-level memory resources.
- The utilization of a common software metadata reduces total design cost because the information generated during a single instrumentation, profiling and characterization phase is available for any subsequent optimization techniques. Owing to the lower entry-level barriers, it may also enable new optimizations.

7.2. Main contributions

The main contributions of this research work are:

- A placement methodology for dynamic data objects that avoids data movements across elements in the memory subsystem and exploits the characteristics of individual memory modules to improve accesses to dynamic data objects – or to avoid that dynamic data objects hinder the normal work of other techniques for other data objects.
- A mechanism to implement the data placement solutions generated with the methodology via the dynamic memory manager, including the modifications required to supply it with adequate additional information.
- The idea of grouping DDTs with similar characteristics as an intermediate point between blind assignment of resources and strict separation, hence enabling exclusive assignment of resources while improving resource utilization.
- The idea of splitting the placement problem into several phases: Preliminary characterization and analysis, grouping and mapping into resources. The first two are complex and should be performed during design time, but the last one may be delayed until run-time to improve system adaptability to resource degradation or different platform configurations.
- A methodology for the characterization of the dynamic behavior of embedded software applications and the construction of a central information repository that can be used by different optimization tools.
- A working tool, $\mathcal{DynAsT}$, to demonstrate the plausibility of the methodology and apply it.
- A working memory simulator that can be used to explore the performance and energy consumption of an application running on different heterogeneous memory organizations.

The research work that I have presented thorough this text has resulted in several conference and journal publications:

- Miguel Peón-Quirós, Alexandros Bartzas, Stylianos Mamagkakis, Francky Catthoor, José Manuel Mendías, and Dimitrios Soudris. Direct memory access optimization in wireless terminals for reduced memory latency and energy consumption. In *Proceedings of International Workshop on Power And Timing Modeling, Optimization and Simulation (PATMOS)*, volume 4644 of *Lecture Notes in Computer Science (LNCS)*, pages 373–383. Springer-Verlag Berlin Heidelberg, 2007. ISBN 978-3-540-74441-2

  In this first publication, we proposed a method to improve DRAM performance when it is simultaneously accessed by a processor and a DMA engine, offering also a glimpse of the relevant role of the memory subsystem in reducing energy consumption and improving performance in embedded systems.

- Alexandros Bartzas, Miguel Peón-Quirós, Stylianos Mamagkakis, Francky Catthoor, Dimitrios Soudris, and José Manuel Mendías. Enabling run-time memory data transfer

optimizations at the system level with automated extraction of embedded software metadata information. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 434–439, Seoul, Korea, 2008. IEEE Computer Society Press. ISBN 978-1-4244-1922-7

In our next publication, we started to experiment with a formal representation of the application's memory-access and data-transfer characteristics extracted through profiling, making an initial proposal for extracting, storing and processing software metadata.

- Alexandros Bartzas, Miguel Peón-Quirós, Stylianos Mamagkakis, Francky Catthoor, Dimitrios Soudris, and José Manuel Mendías. Direct memory access usage optimization in network applications for reduced memory latency and energy consumption. *Journal of Embedded Computing (JEC)*, 3:241–254, August 2009

  We continued improving our methods and experiments regarding DRAM data transfers optimization, which materialized in this publication where we proposed system-level optimizations to adapt DMA usage to run-time conditions. In this work, we also evaluated the use of system scenarios as the adaptation mechanism.

- Alexandros Bartzas, Miguel Peón-Quirós, Christophe Poucet, Christos Baloukas, Stylianos Mamagkakis, Francky Catthoor, Dimitrios Soudris, and José Manuel Mendías. Software metadata: Systematic characterization of the memory behaviour of dynamic applications. *Journal of Systems and Software (JSS)*, Volume 83 Issue 6, June 2010(83):1051–1075, 2010. Software Architecture and Mobility

  The line of work on software metadata led us to this publication where we presented a complete formalization of the methods and techniques required to characterize the dynamic-memory behavior of software applications and construct a representation that can be used by multiple optimization tools, thus saving overall optimization effort.

- Miguel Peón-Quirós, Alexandros Bartzas, Stylianos Mamagkakis, Francky Catthoor, José M. Mendías, and Dimitrios Soudris. Placement of linked dynamic data structures over heterogeneous memories in embedded systems. *ACM Transactions on Embedded Computing (TECS)*, 14(2):37:1–37:30, February 2015

  Our previous works on memory subsystem optimization built our understanding of the intricacies of heterogeneous memory organizations and the necessity for a careful data placement. More importantly, our work on optimizations to efficiently implement data movements across memory hierarchy elements via a DMA module uncovered the main motivator for this text: That for applications that create data objects dynamically and under specific circumstances, data movements may not be the best option to palliate the difference of speed between processing elements and memories.

  Other important milestones were our work on DRAM access optimizations, assimilating previous work on higher-level concepts such as dynamic data type and dynamic memory management optimization, the design of effective and systematic ways to characterize application behavior (regarding memory accesses) and the need to formulate comprehensive methodologies that can be easily applied to the development of real systems. The resulting work on dynamic data placement resulted in this publication [PBM+15] and the core contents for this text.

Finally, I would like to remark that the research presented in this text is part of a global project that spawns several institutions and generations of researchers, led by Professors Francky Catthoor (IMEC, Belgium), Dimitrios Soudris (DUTH and currently NTUA, Greece) and José Manuel Mendías (UCM, Madrid). Part of the research results of this decades-long work conducted by several authors have been recently collected in a book [AMP+15].

7.3. Future work

— To boldly go where no one has gone before...

I hope that this text motivates well enough the importance of a careful placement of dynamic data objects, particularly as memory hierarchies become more complex. However, what I would really like is that it opens more questions than the ones it answers – as should be the case with any worthwhile research. Further than performing more experiments on more applications and different memory subsystem designs, in the following paragraphs I discuss briefly some of the topics that I consider worth of future research, either by me or by other people.

In first place, I enumerate future research on each of the steps of the methodology for placement of dynamic data objects. Second, I study the applicability of this idea to other computing environments besides embedded systems. Finally, I present future possibilities for the study and exploitation of software metadata.

7.3.1. Methodology and algorithms

The methodology that I have presented in Chapters 2 and 4 is based on the heuristic of splitting the placement problem in two steps, grouping and mapping. However, future research could evaluate the feasibility of performing both steps at the same time, even with a perfect solver (possibly based on estimators for partial solutions), the computational complexity of that approach and its possible improvements.

7.3.1.1. Profiling

The profiling methods that I have presented here have two main drawbacks. The first one is the necessity for manual instrumentation of the source code. Even with the reduced overhead of the exception-based mechanism, the designer still has to modify the declaration of every dynamic class. Then, these methods are quite slow, hindering the analysis of time-sensitive applications.

One approach that could be interesting is the modification of the compiler so that it inserts automatically the instrumentation needed for memory access tracking and provides type information to the DMM API functions. Memory access instrumentation would be used only during profiling, but type information would be inserted also in the final code. As an example, LLVM [LA04] is a compiler infrastructure specifically built to ease its modification and currently used in multiple commercial and research products.[1] For example, type information is explicit for the compiler during a `new` operation; for `free` operations, it might be possible to obtain it using Run-Time Type Information (RTTI).

[1] Chris Lattner and Vikram Adve received the ACM Software System Award in 2012 for their work on LLVM `http://awards.acm.org/award_winners/lattner_5074762.cfm`.

Other possibility would be to completely avoid profiling and extract equivalent information using other analysis techniques. For example, the number of accesses in each code branch could be determined using static analysis techniques; the frequency of execution of each branch could be then calculated with a lightweight profiling (similar to the "scopes" presented in Chapter 5, Gprof or other methods used by profiling compilers). Such methods would be especially important to apply the placement methodology to other environments with more complex applications as presented later in Section 7.3.3.

Finally, the examples used in this text perform an initial profiling on a processor architecture that may not be the one used for execution. This is a very useful approach that can be used when the final platform is not yet available, but additional experiments should be conducted to analyze the deviations introduced by this approach. The most significant difference may be the availability of different numbers of general-purpose processor registers. With a reduced number of them, the compiler has to generate more accesses to memory. Although I believe that the difference is not significant because dynamic data objects are typically accessed through pointers – it is the pointers themselves which may be hold in registers – and traversals of data structures offer little opportunity for reuse, further experiments should be conducted to verify the impact of this decision. Particularly, because profiling on a workstation may be crucial in cases where the final platform does not have enough resources (storage, performance) to profile the application reasonably, or simply because a workstation may be much faster, or a cluster of them may be used to profile under a myriad of different conditions.

7.3.1.2. Grouping

As explained thorough this text, grouping is a mechanism introduced to improve resource exploitation while still being able to assign resources in exclusivity to the instances of some dynamic data types. Furthermore, the algorithm presented in Chapter 2 is based on a set of heuristics that may (and should) be improved in future work. Apart from evaluating a complete search of the solution space, in the following paragraphs I present several improvements that I consider worth of further examination.

The grouping algorithm can be extended to take into account the access pattern of the instances of each DDT: Some access patterns may be more suitable for specific types of memories. For instance, those DDTs with prominently sequential access patterns may be assigned to a DRAM, even if it is less efficient, whereas SRAM is reserved for other DDTs with more random access patterns that could hinder the row-oriented organization of DRAMs. In this regard, we conducted some preliminary experiments using the "selfishness" metric proposed by Marchal et al. [MGP+03, MCB+04], which gives a measure of how sequential the accesses to a data structure are. With it, DDTs with a high selfishness can only be joined to groups that contain DDTs with a similarly high one. During mapping, pools with a high selfishness are mapped preferably on DRAM memories. Although the results were promising, more research on the interactions with other considerations is needed before that work is ready for publication.

Regarding access pattern identification, we also conducted some preliminary tests in which DDTs whose objects are commonly accessed in tandem (e.g., dynamic vectors used in reduction operations) are marked as "incompatible," so that the grouping algorithm tries to avoid combining them and the mapping process avoids placing them in the same memory resources. For this to work, we augmented the attributes of DDTs, groups and pools to include a list of incompatible peers (in the sense of preference for not being placed together, not of a strict

incompatibility). These attributes can be used to avoid placing them in the same bank of a DRAM, thus reducing the number of row misses. During the elaboration of this text, such work was not fully functional and the experiments were still not sufficiently developed; thus, I used instead the mapping parameter `SpreadPoolsInDRAMBanks` to ensure that the DDTs were mapped into as many DRAM banks as possible. Therefore, I removed the portions of code required to implement this functionality in $\mathcal{DynAsT}$, leaving them for a more detailed presentation in future work.

The possibility of placing two DDTs in different memories so that they can be accessed in parallel is a more complex topic that may require multiple data ports in the processor or at least interleaving of operations with long access times (DRAMs may fall in this category for random accesses, but they can transfer data continuously while in burst mode). An interesting study, limited to static data objects, is presented by Soto et al. [SRS12]. This may constitute an interesting topic for future research in the context of dynamic data placement.

The most important issue that should be investigated is perhaps the possibility of differentiating among instances of a DDT that present very different FPBs. Through this text I have assumed that the instances of a DDT have all a similar number of accesses, but in some applications the situation may be different. An efficient answer would involve identification and specific placement of individual instances, or even migration of instances – maybe with mechanisms similar to those used by generational garbage collectors – once their specific characteristics are determined.

Another interesting research path is to develop the algorithms needed to solve the dynamic-data placement problem optimally and compare the solutions obtained with my methodology against the optimal for each case. Such work would require defining suitable estimators for partial solutions (dynamic programming) or executing the simulator for every solution generated (branch-and-bound). Appendix C offers more details on the complexities of the placement problem.

An interesting experiment might be the modification of the grouping and mapping algorithms to consider the combination of a cache memory plus a DRAM as a single entity, something such as a "cached-DRAM." The idea is that the properties of the combination would be different than the properties of the DRAM. Instead of the approach presented in this text (simply leaving cached DRAM areas for data objects that are not managed by the methodology) they would be seen as memories with special characteristics and automatically assigned by the methodology to the appropriate data objects, maybe opening the path to tackle as well the placement of more caching-amenable static data objects. The major obstacle for this approach would be that a simple statistic of typical hits and misses would not be enough: The methodology algorithms would need to know exactly how the cache reacts to the concrete (groups of) DDTs placed on its related memory and the possible interferences with other DDTs placed in the same DRAM cached area. Even more, multiple pools placed in the same cached-DRAM would interfere with each other. Therefore, this experiment might require also new algorithms to perform a simulation-guided full exploration of the design space.

Finally, a radically different approach that could be explored for the grouping of DDTs is the evaluation of the lifeness of each DDT and group as a digital signal. Instead of directly merging "behaviors" to find suitable combinations, the grouping step would analyze each "signal" applying techniques similar to the search of correlations used in the realm of digital signal processing such as explained by Proakis and Manolakis [PM98, Chap. 2] to identify similarities and differences in behavior. More specifically, instead of searching for high correlations,

the algorithm would look for low correlations, representing periods of disjoint resource usage.

7.3.1.3. Mapping

If, as suggested in Section 7.3.1.2, a mechanism to identify groups/pools that should be placed apart is added, then the mapping algorithm has to be modified to contemplate such restrictions. In particular, to respect the placement of pools in different DRAM banks. Once the restrictions are generated during the grouping step, the modifications to the mapping algorithm might seem straightforward. However, some new issues arise for consideration.

For example, what should be done when a memory resource has still some free space, but the pools in it are incompatible with the new one being considered? One option is to skip that free space and map the pool in the next available memory resource. The next pool would use the space that was left in the first resource and the remaining space in the second one. Or would it be better to map the next pools until the old resource is fully used and then map the pool that was kept on hold? That option could produce a dangerous effect of "priority inversion" if the new pool considered is also incompatible with the one that was being kept on hold.

A third option would be modifying the mapping algorithm to pick the next pool that is compatible with the pools already mapped in the current memory module and map as much of it as possible. When the resource is exhausted, the pool would be inserted back in the list of pools, with its size adjusted, so that the algorithm would reconsider again all the available pools for the new resource. That option would observe pool priorities, but possibly producing many splits over quite distinct resources for some pools.

All of those options would be very easy to implement in $\mathcal{DynAsT}$, but their effects should be carefully evaluated through additional experiments.

On a different topic, the most promising research path seems to be the execution of the mapping step at run-time. This approach promises very interesting possibilities for platform independence and system reliability via adaptation to resource degradation, making it a desirable research direction in the realm of embedded systems.

Although not strictly affecting the mapping phase, in Section 4.5.6 I analyzed the possibility of performing DMM design (pool formation) after the mapping step so that the actual properties of the memories assigned to each pool are known. The mapping step is agnostic with respect to the design of the DMMs in each pool; hence, that path could be pursued freely. However, one consideration is due: If executing the mapping step at run-time is deemed as interesting, then the pool formation step must also happen at run-time – strictly speaking, both would be executed at the time of loading the application in the same way than dynamic-library linking. I explore this situation below.

7.3.1.4. DMM

The design of dynamic memory managers has received a lot of attention during decades and thus, it is in a very mature state. However, one interesting research path could be fast and low-resource techniques for the design of DMMs. In other words, I would propose some meta-research on the techniques used to design the DMMs. The reason is that, traditionally, DMMs are built and evaluated during the design phase of the system. Consequently, even if reducing the time required to design a system is fundamental to reduce the time-to-market of embedded systems, time scales are completely different to what would be needed to perform DMM

design at run-time after the mapping stage. For example, some prior work involves searches in the design space, whereas a run-time DMM design step would have to run in subsecond times. This necessity may motivate the development of quick heuristics for approximating a good DMM design given some predefined parameters.

In the absence of such quick design mechanisms, other options would still be possible. One would be executing pool formation normally at design time, but producing one different design for each possible type of memory resource in which the pool may be placed. Given that the number of different memory types is relatively small, the overhead should not be prohibitive, particularly if DMM designs are shipped in the form of templates for assemblage by a factory object. At run-time, the factory would choose the recipe that best matches the memory resource actually assigned to the pool – the normal approach explained in this text corresponds with the case of having only one recipe for each pool.

However, before enrolling in such a research, a previous step would be to effectively determine if the advantages obtained designing the DMMs considering the characteristics of the actual memory resources are indeed significant, and under what circumstances.

7.3.1.5. Deployment

In Section 2.10 I propose the use of a library of modules to compose dynamic memory managers at run-time, instead of the mixin-based mechanism employed in previous work. However, an important consideration is that virtual calls to functions linked through a strategy design pattern imply a double indirection. Further research should be conducted to determine the impact of this factor, especially for embedded processors with simple (if at all) branch predictors. If the overhead were too high, a more involved "patching" mechanism, similar to the work of the system loader for dynamic linking, could be explored on the basis that the DMM library would not be a completely arbitrary piece of code (i.e., the extent of the changes is limited and the DMM objects are created by the system itself, not freely by the application).

7.3.1.6. Simulation

Accuracy. In Section 4.5.4 I explained that simulation is just an approximation to quickly analyze the characteristics of a system, but the results obtained may not be completely equal to those attained during execution on a real system. Therefore, additional work should be conducted to better assess the discrepancies between the results obtained with the simulator included in $\mathcal{DynAsT}$ and those obtained in a real platform. This is important because an accurate simulator is a very powerful tool to reduce costs during the design phase. For example, simulation allows the designer to explore different platform configurations fast and in a more flexible way than real execution, particularly if the physical platform is not yet available. In the following paragraphs I describe some of the most promising directions for future work in the simulator, including improving its accuracy or adding new capabilities.

Architectural simulation. $\mathcal{DynAsT}$'s simulator is based on memory traces. Although it allows analyzing the performance of the memory subsystem, some subtleties such as the timing between memory accesses are lost with this approach. An interesting future extension could be integrating the energy consumption and performance evaluation capabilities of the simulator with a full architectural simulation/emulation platform. In particular, integration with Gem5 [BBB+11] seems plausible and particularly promising: Gem5 can simulate several

processor architectures and run complete systems with their operating system and applications via two working modes, System-call Emulation (SE) and Full-System (FS). Integrating *DynAsT*'s memory simulator with Gem5 would enable the analysis of the effect of multiple applications executed simultaneously or the impact of different scheduling policies in single or multiprocessor environments. One of the most promising features of Gem5 seems to be the simulation of unlimited numbers of processors, which can enable the exploration of memory subsystems and placement techniques for current and future server configurations. Of course, simulation performance may become a bottleneck for such kind of experiments, an issue that should get special attention in such a hypothetical future research.

Bit-line transitions in energy calculations. A nice improvement that could be done to the simulator is to consider the effect of actual value transitions in the bit lines of the memory subsystem: If data values do not change with respect to the previous operation, no (dynamic) energy is consumed driving capacitative loads. Since the actual values of data exchanged with external memories may be different for each platform configuration, absolute energy consumption figures may also vary. The same considerations apply when calculating the energy consumed by the memory controller driving address and command lines.

Data values can be provided to the simulator modifying the profiling mechanism to include not only the address of each access, but also its value. At the expense of an increased log size, these values can be later supplied to the simulator. The simulation of SRAMs and DRAMs would require no further modifications because accesses are atomic and the simulator keeps the correspondence between addresses in the original execution and during simulation with the final placement decisions. However, cache memories would require more careful attention – probably storing actual data values in the simulated caches – because there is not a direct correlation between the dynamic data objects stored adjacently in a cache line during simulation and the original layout. An interesting remark is that integration with an architectural simulator such as proposed before would also provide the concrete data values exchanged with the external memories.

Analysis of inactivity periods. Taking into account the inactivity periods of the memories would enable, for example, the exploration of energy-saving techniques. The designer could evaluate the effects of varying degrees of aggressiveness in moving memory modules into low energy modes and the impact on performance of reactivating them with more or less frequency.

In this last regard, an innovative proposal would be the exploitation of high-level information from the DMM to detect when a memory module does not hold any alive instances. As the system knows which pools are mapped into a given memory resource, it can poll the corresponding DMMs to check if there are any instances alive. If the module is not being used, it can be completely powered down without risk of data loss.

Even more, techniques such as presented by Lattner and Adve [LA05] might be used to migrate entire pools according to usage statistics and increase the size of the inactivity periods. At the cost of some access overhead – that should be evaluated and included in the trade-off – the system could migrate the pools and seamlessly transition into different placement solutions as proposed in Section 4.5.3 with system scenarios. Alternatively, the presence of an MMU could also be used to directly migrate whole pools with page-granularity.

DMM during simulation. In the realm of dynamic memory manager simulation, the current implementation of the simulator uses idealized DMMs that create their data structures outside of the application data space and always use splitting and coalescing. This decision was taken to simplify the simulator implementation. In normal circumstances, DMMs build their data structures inside the pools themselves; thus, their own accesses may affect energy consumption in the platform to some degree. More research should be conducted to evaluate the impact of this decision or incorporate the actual DMMs during simulation.

However, it could actually be interesting to evaluate the effect of separating the DMM internal structures from the pool itself. Then, according to their number and pattern of accesses, they could be placed in different memory resources. Even if the memory footprint of the application grew, it might be possible to obtain interesting improvements due to more efficient DMM operations and the possibility of using more complex coalescing and splitting algorithms that are often avoided in DRAM memories to reduce the number of random accesses.

For example, in a 1 KB pool that serves exclusively requests of 24 B, the total number of blocks available for the application is 42. Depending on the space required by the internal DMM structures, their footprint might be 168 B (if 4 B per DMM block), 336 B (if 8 B per DMM block) or 504 B (if 12 B per DMM block). That space could be allocated from a tiny dedicated SRAM of 256 B or 512 B for very efficient DMM operations.

DRAM bank layout. As explained in Sections 3.5 and 3.6, I assume that address layout in DRAMs can be configured as "bank-row-column." This is possible in many systems, particularly in the case of FPGAs or ASICs, and is common in some DSPs. For general-purpose systems with caches, the usual layout tends to be "row-bank-column," while other options such as permutation-based interleaving [ZZZ00] have been proposed at different times.

The address layout used in this text enables the placement of complete pools into DRAM banks ensuring that objects do not cross banks. However, other schemes favor long sequences of accesses (streams) because the row in the next bank can be activated in the background while a row in the current bank is accessed. Thus, big data objects (e.g., of many KB or even several MB) can be more efficiently accessed – in comparison, with the proposed layout 1 out of 512-to-2048 accesses could suffer a full row-activation delay, depending on DRAM row sizes. Although I believe that the advantages obtained by the ability to place pools directly on banks far outnumber the possible inefficiencies encountered for long data transfers, future research might be conducted to evaluate the merits of these and other approaches.

DRAM row management. The simulator does not currently implement any policy for proactively closing rows and thus does not account for the potential energy savings. This is a topic worth future research that needs to be conducted at three different levels. First, policies for determining when to close a row. Second, DRAM banks consume less energy when no rows are open, but deeper "sleeping states" require more time before a row can be opened again. Third, the whole DRAM module can be pushed into energy-saving modes.

Regarding row policy, if the controller closes a row and it is accessed soon again, the new access incurs extra energy consumption and delay. As explained in Section 3.4.2, when DRAM accesses are sparse an interesting trade-off appears between closing a row and opening it again sometime later, or keeping it open all the time, because a bank with no open rows can enter a state with lower energy consumption. The extremes are usually known as "open-page" or "closed-page" policies. Some intermediate approaches try to identify groups of sequential

accesses and close the active row proactively during the last one [Dod06], a technique that is useful to at least partially hide precharge times since accesses to new rows will only wait for the row-activation time. The simulator does not implement any policy for proactively closing rows and thus does not currently account for the potential energy savings, but this capability can be easily implemented to conduct new research.

Additional energy savings may be obtained when none of the DRAM banks have open rows by carefully scheduling energy-saving modes for complete DRAM modules. The solutions designed with $\mathcal{DynAsT}$ may benefit from this possibility particularly because an efficient exploitation of SRAM memories may create long periods of time without any accesses to the DRAM; some special applications may even be run entirely on SRAMs. The simulator can be used to identify such conditions and explore new energy-saving schemes.

Pipelined accesses. Finally, the simulator may also be extended to implement pipelined accesses to SRAMs and caches, as multicycle accesses do currently produce stalls. In that way, more capable processor architectures could be evaluated.

7.3.1.7. Other areas

Extended aplicability. My methodology works particularly well when several DDTs alternate footprint requirements (high resource recycling) or some groups are very accessed with interleaved sporadic accesses to other ones since it protects the most accessed instances against eviction. However, even when all DDTs have similar liveness and receive significant numbers of accesses, the methodology may still be interesting if spatial locality is low because movement of non-reused data words is avoided. Although some of the capacity could be underexploited because of the lower chances to reuse space through grouping, avoiding continuous (unproductive) data movements may produce significant energy savings. This effect may also apply for energy consumption in data-center servers.

Improving cache performance via DMM. As I have explained thorough this text, DM tends often to hinder the performance of cache memories. However, I am preparing future research on two promising options to exploit high-level knowledge from the DMM and give hints to the cache memories that may help to improve their performance.

First, the DMM knows when a memory range contains valid – in the sense of "alive" – data, whereas the cache controller does not: Normally, cache lines are marked as "valid" when they contain data copied from a further memory, and "modified" when these data have changed and must be written back before being substituted. However, when a dynamic data object is destroyed, its associated cache lines do not need to be copied back to main memory anymore. Similarly, when a dynamic object is created on an address range that does not currently reside in the cache, there is no advantage on copying any words to the cache because those words in main memory do not contain valid data – only cache-line allocation is required. This knowledge, if exploited efficiently, has the potential to significantly reduce the number of data movements across levels in a memory hierarchy.

When a dynamic data object is destroyed, the corresponding cache lines (if any) may be marked as "invalid" to avoid copying them back to main memory in case of a future eviction. Indeed, in associative caches that line would be the first selected to store new data from main memory, saving the eviction of a potentially useful line in one of the other cache sets. All of this could result in interesting energy consumption and performance improvements.

Second, DDT knowledge obtained with $\mathcal{DynAsT}$ might be used to improve the performance of cache-based hierarchical memory organizations. Cache hierarchies tend to copy new data from main memory into all cache levels. When the new data are not going to be reused (e.g., during stream processing), the cache hierarchy suffers a phenomenon known as "cache pollution" that can severely reduce performance and increase energy consumption. Some processors support instructions to prefetch data directly into the L1, so that the other levels are not polluted and the contents of the L1 can be recovered faster.

It may be possible to design a more efficient approach by marking the pages used by each pool with a maximum "level of cacheability" (e.g., in a "Page Attribute Table"). When data are moved closer to the processor, they would be copied only up to the maximum allowed level, thus avoiding interference with more important data. Even better, they could be allowed to reside in closer levels, but only if they use free lines without forcing any evictions – situation whose likelihood might be increased with the idea proposed in the previous paragraphs. I believe that that approach may be more efficient than polluting the L1. The main drawback of this idea is that the processor architecture must allow data accesses to any of the cache levels. Nonetheless, it may be an interesting approach for ASIC or FPGA-based designs.

Static versus dynamic-data placement. In Section 4.5.5 I discuss the possibility of using $\mathcal{DynAsT}$ to manage also the static data objects of the application. This is an innovative approach that deserves consideration in future research.

Integration in the LLVM infrastructure. As mentioned earlier, exploring a possible integration of $\mathcal{DynAsT}$ with the infrastructure provided by LLVM may be interesting in order to automate some processes such as profiling and instrumentation. Indeed, the integration of the software metadata techniques may also be of interest.

7.3.2. Software metadata

The part of the software metadata methodology that would benefit the most from future research is the extraction of the raw information of the applications. Profiling works well for dynamic applications, but it requires instrumentation and sometimes lengthy execution with different inputs. In this section I present a couple of ideas that could be used instead of or in addition to profiling.

Future research may evaluate static-analysis techniques to explore and characterize the application code paths; a lightweight profiling mechanism would complement them assigning weights to the branches according to dynamic conditions. For example, symbolic execution seems a promising technique. Although its main drawback is code-path explosion, exponential on the number of branches, it might be possible to combine it with a light profiling mechanism, or perhaps concolic testing,[2] to identify the most common code paths in the application and order them by frequency of appearance or severity of their impact on system performance. Further analysis would then analyze the resource demands associated to each of those code paths and propose adequate optimizations.

Regarding the profiling techniques themselves, tools such as `clang` make it easier for researchers to implement experimental concepts in a real-world compiler. The possibility of using it to implement a compile-time instrumentation mechanism for profiling accesses to

[2] Concolic testing combines concrete and symbolic execution to cope with the code-path explosion of pure symbolic execution [BS08, CDE08, CS13].

dynamic memory objects seems alluring. Such a mechanism would introduce a library call after every access scheduled by the compiler, for any object known to be declared dynamically, independently of whether the object resides temporarily on a processor register or memory.

7.3.3. Applicability to other environments

This thesis began with the aim of improving the cost of accessing dynamically allocated objects in embedded systems with SRAMs. However, as other computing systems become also more complex and NUMA, more relevant becomes data placement also for them, especially because in many cases the sheer amount of data and the complexity of the access patterns makes the advantages of cache memories less clear. In this regard, an interesting experiment on the (un)suitability of complex cache hierarchies for scale-out workloads was presented by Ferdman et al. [FAK+12]. In spite of these observations, there is significant resistance to abandon the transparent mechanism of cache memories that so well has served us for many years. The following quotation summarizes the reasons why explicitly addressable memories have been relegated for a long time to the realm of embedded systems:

> "One idea that periodically arises is the use of programmer-controlled scratchpad or other high-speed memories [...]. Such ideas have never made the mainstream for several reasons: First, they break the memory model by introducing address spaces with different behavior. Second, unlike compiler-based or programmer-based cache optimizations (such as prefetching), memory transformations with scratchpads must completely handle the remapping from main memory address space to the scratchpad address space. This makes such transformations more difficult and limited in applicability. In GPUs [...], where local scratchpad memories are heavily used, the burden for managing them currently falls on the programmer." [HP11, p. 131]

We may imagine the trade-off between energy efficiency and ease of design as a continuous between these two extremes. Computer architects have been pulling towards the side of ease of design for many decades, with cache memories offering an almost uniform view of the memory subsystem. Although the power and memory walls are pressing problems, favoring energy efficiency to the point of making the design of new systems unfathomable is neither an option. The lack of tools that help to tackle the complexity of the designs is probably the reason that inspires the past reluctance to adopt new mechanisms that may complement the cache memory. In this text I show how we can move a bit towards the other extreme, so that we recover some energy efficiency with bounded impact on complexity.

The dynamic-memory based approach that I have proposed does not require special care by the programmers: Minimum changes to augment the DM API – which might be introduced automatically by the compiler in the future – are enough. Dynamic data objects are then accessed as usual, via pointers (references). The reason is that my method does not mimic caching mechanisms to prefetch blocks of data before processing. Nevertheless, the memory model has not been uniform for a long time. Consider, for example, the case of a dual-processor server with half its DRAM modules connected to the bus of each of the processors and an interconnection bus (e.g., AMD's HyperTransport) connecting both processors. The time it takes for each processor to access a memory word from a DRAM module depends on whether the module is connected to the processor local bus or it has to go through the interconnection bus; thus, the memory space is effectively non-uniform from a performance point

of view. Therefore, I believe the time to study explicit placement techniques as a technique to cope with the memory wall in more general types of computing systems has arrived.

Programmers can usually take advantage of the memory subsystem topology with specific functions to allocate memory local to a node (for example, with the use of GNU's libNUMA). But an automatic placement method that considers the characteristics of each element in this extended memory organization is still missing. I propose my methodology for dynamic-data placement as a starting point for this future work because it can be applied to any system in which the memory subsystem properties are a visible part of the programmer's model. As a natural choice, I further propose dynamic memory as the mechanism to implement data placement because it can adapt not only to the resources available when the execution starts, but also to the variations that happen during execution. Even more, the combination of dynamic memory and virtual memory may enable the reevaluation of placement decisions during execution.

To conclude the main body of this text, in the next paragraphs I explain why the methodology that I have proposed in this work is relevant for systems other than simple embedded devices and I outline interesting research options in those areas.

7.3.3.1. Scale-up systems

The traditional model for improving system performance is using faster processors or more processors in a single system. The main characteristic of these systems is that the processors form part of a single system, usually with a single operating system. Quite frequently, these systems support a shared memory space in which every memory address is visible for every processor. However, to limit the complexity of the designs and to improve bus performance, memory modules and processors are clustered. Although all the memory modules employ the same technology, the cost of each access depends on the distance between the processor demanding the data and the module containing them. Contention may also become a serious problem when several processors need to access the same memory module; thus the interconnection network becomes critical. This model of computation is suited for the resolution of big individual problems.

Solutions such as GNU's libNUMA or the Microsoft Windows NUMA memory management APIs provide the mechanism for application to allocate space on concrete nodes (where nodes represent groups of processors and the memory modules directly connected to them). However, these APIs offer just the mechanism to allocate the space. How that space is used is left for each application designer. Even worst, the default approach is to allocate memory in local resources and, if exhausted, from the closest ones. Disregarding how each data object is going to be used.

Data placement can help to improve performance in these systems by ensuring that the most accessed data objects are located in the closest memory resources. Most importantly, by improving the tools presented in this work, the process may be executed automatically, freeing the designers from the burden of data placement across complex systems.

Multiprocessing adds complexity to the problem, signaling clear ways for research: For data shared among tasks running on several processors, should placement select a memory that is close to all of them (even if that node is not any of the ones involved with the data) or a memory that is close to one of them although the rest may pay higher access costs?

Another area deserving future work is the extension of the techniques presented in this work, which focus on single-processor systems, to multiprocessor environments. In this re-

gard, the work presented by Berger et al. [BMBW00], the Hoard dynamic memory manager, seems as an appropriate starting point to study the particularities of these systems and the modifications that should be incorporated into the placement techniques. Many opportunities remain in this area because solutions such as Hoard focus on efficient techniques for managing the pool of free blocks in multiprocessor systems, but they do not consider the characteristics of the underlaying memory resources.

7.3.3.2. Scale-out systems

A more economical alternative to big supercomputers is the scale-out model, where instead of creating more complex (and expensive) computers, a set of simpler ones is interconnected through regular networking technologies (e.g., TCP/IP over 10 Gigabit Ethernet) to create a big coordinated system. These systems are appropriate for problems that can be split into more or less independent parts or for tackling with swarms of simpler tasks that probably depend on vast distributed data-repositories (e.g., NoSQL databases such as Cassandra). The scale-out model uses sophisticated algorithms to distribute the data aiming for improved performance and reliability against data losses.

Modern data centers are built around the scale-out model. Their use cases include swarms of simple works (e.g., search queries), hosting of virtual machines from different users (e.g., scalable cloud computing) and complex jobs over huge data collections that exceed the storage capacity of any single node (e.g., "big data" problems). The last use case has led to the concept of Warehouse-Scale Computers (WSC), where all the individual computers and the interconnection network are seen as a single big machine with particular characteristics.

In this new and exciting realm, I can foresee several situations where data placement solutions can benefit performance and, more importantly, reduce energy consumption. First, although applications are usually deployed in cloud services as complete virtual machines, the "OS-as-a-library" approach [MMR+13] may allow system programmers to build highly specialized solutions that exploit the underlaying memory resources. In a sense, these systems are an intermediate step between embedded systems and general NUMA systems where the complete set of applications is known at design time, keeping the same adaptability that is present in embedded systems. If the characteristics of the underlaying hardware are known to the designer, it can create adequate platform description files such as those used by $\mathcal{DynAsT}$ and apply full optimization techniques such as presented in this work.

Second, some organizations are exploring the possibility of attacking the three walls at the same time changing the model of powerful and energy-hungry processors for a model composed of a sea of processing elements of modest performance that operate at a lower clock frequency and are attached to an intricate web of storage elements. This model can be suitable for the problem of big numbers of simple queries, where the relevant factor is not so much the complexity of the computations but their latency and avoiding "long tails" (jobs that take significantly longer to complete than the average) [XMNB13]. Clearly, data placement becomes a major issue in this paradigm. Distributed shared memory may also be an interesting target for data placement optimization.

Finally, the most interesting future development is probably the redesign of the rack in datacenters that companies such as Intel and Facebook propose. The likely advent of silicon photonics may open the door for new computing models where memory resources are separated and shared by many computing nodes. In the extreme case, HP Labs have proposed an innovative project named "The Machine" [HP]. Applications should then choose carefully

which data should be placed on local memory resources or on the shared (and presumably much bigger) pool. In some respects, that model would mimic the memory subsystem of many embedded systems, with the local DRAM playing the same role with respect to the shared memory pool than the SRAM with respect to the main DRAM. Therefore, a careful data placement might grant similar performance improvements and reductions in energy consumption.

7.3.3.3. New horizons

The next years promise to be exciting if Storage-Class Memories (SCM) become a reality. Storage-Class Memories [FW08] promise persistence, low or moderate latency, high density and word addressing. The main differentiator with respect to, for example, flash-based SSDs or magnetic disks is the capability of accessing and updating individual memory words.

Two possibilities appear immediately to exploit these new technologies. First, Mnemosyne [VTS11] and NV-Heaps [CCA+11] propose using SCMs to blur the distinction between primary and secondary storage, introducing the concept of persistent heaps. Their authors pay special attention to the prevention of programming bugs, such as pointers in persistent storage that reference objects in volatile storage. This line of work aims to create a flatter model of memory and storage that can be very useful for big-scale applications such as distributed hash tables.

Once the data objects are separated into volatile and persistent, techniques similar to my methodology can be used to decide into which resource each dynamic data pool should be placed.

The second possibility is closer to the approach of this text and consists on using any available storage technologies in a system as primary storage, not with the aim of persistence, but to increase the amount of storage available for computation. Although some techniques enable direct computation on SSDs or magnetic disks, the word-addressability of SCMs offer the ideal means for this goal. Once again, dynamic data placement is crucial to decide which data objects should be placed on each component of the memory subsystem, according to the characteristics of both.

However, even without the advantages of SCMs, some things can already be done. As an example, let's consider the case of (flash) SSDs. At the time of writing this text, SSDs connected directly through the PCI-e bus (as opposed to the disk controller interface) are a reality. Although the original idea of SSDs was to improve I/O performance, that seems hardly to be the best we can do with them. For example, SDAlloc [BP11] offers a mechanism to use them as a repository of normal dynamic memory allocated with functions similar to `malloc()`. A careful management of dynamic memory and buffering allows SDAlloc to create the illusion of extended DRAM without blindly burning the flash storage. In comparison, placing the swap file of a regular non SSD-aware operating system on a flash disk may severely reduce its lifetime because of the mismatch between processor pages (4 KB) and flash blocks (usually 128 KB) – NAND flash devices can be read or written in pages of 512 B to 4 KB, but erasure can only be done at the block level.

An area that I deem interesting is the abstraction of all the storage elements in the system to provide working memory: Uniform algorithms, no need for serialization processes and no transitions between user and kernel code for file accesses may reduce software size and increase application performance. If word-addressability is possible, then the only decision that remains is data placement. For block-oriented storage (e.g., magnetic hard disks), some addi-

tions are needed. For example, to use a magnetic disk as primary (working) storage, a simple first approach would use several DRAM buffers as working copies of the disk sectors, exactly in the same way that DRAM modules use row buffers to access complete cell rows – indeed, at a similar granularity because magnetic disks have sectors of 512 B or 4 KB, whereas DRAM rows commonly range in size from 512 words to 2048 words. A more involved approach for purpose-specific machines would implement sector management in a special controller directly connected to the processor bus, so that the processor could use regular load/store instructions with sector management performed transparently by the controller, exactly as the memory controller hides DRAM-idiosyncrasies from the processor.

The ability to do data placement is the most relevant difference between this model and the traditional use of a swap file to increase the amount of memory available for applications: If swapping mechanisms are used without any precautions, it may be possible that physical memory pages get filled with a mixture of frequently and seldom accessed objects. In order to support application accesses, the operating system needs to move entire pages even if only a few bytes are going to be accessed – similar to the problem with cache memories that motivated this work.

A long-term expansion of the ideas for dynamic data placement presented in this work would exploit multiple technologies to improve system performance and reduce energy consumption. For example, (addressable) SRAM may be reserved for frequently accessed collections of small data objects, DRAM for objects frequently accessed and modified, flash storage for data accessed in streams and seldom written, and, finally, a magnetic disk used to store collections of big data objects that are infrequently accessed but where a high proportion of these accesses are updates. Data placement would be the key to separate data objects with different characteristics and exploit the strengths of each memory/storage technology.

At the end of the day, mechanisms such as SDAlloc provide the capability to allocate memory from different types of resources. Complementarily, future extensions to my methodology would allow choosing the appropriate memory elements to place each dynamic data object.

Appendix A

A gentle introduction to dynamic memory, linked data structures and their impact on access locality

This appendix analyzes the extent of the problem that lies at the core of this work: How common is that applications that use dynamic memory are not able to benefit from the improvements brought by cache memories (and equivalent techniques)? First, I explain briefly the most typical use cases for dynamic memory. Then, I explore different situations where the use of dynamically linked data structures (DDTs), perhaps the most significant use case of dynamic memory, can hinder data access locality. Finally, I also introduce some of the main causes for this loss of locality: Reservation of non-contiguous memory blocks for consecutive allocations, element insertion or removal and changes in the internal organization of the dynamic structures that shift their physical connections while preserving their logical ones.

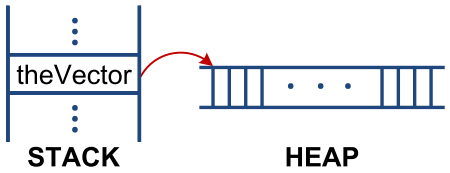
A.1. Use cases for dynamic memory

Dynamic memory (DM) has two main components: A range of reserved memory addresses and the algorithms needed to manage it. The algorithms perform their bookkeeping building data structures that are usually embedded in the address range itself. As in the rest of this work, I use the term *heap* to denote the address range, *dynamic memory manager* (DMM) for the algorithms and *pool* for the algorithms, their internal control data structures and the address range as a whole. With these definitions, let us consider now three common usage patterns of dynamic memory that applications can employ to organize their internal data objects.

1. Allocation of vectors whose size is known only at run-time. This is the most basic case and, apart from the allocation and deallocation calls themselves, the rest of the process is exactly the same as accessing a vector through a pointer in C or C++. In Java, this is the standard implementation for the allocation of arrays, either of primitive types or objects. The following code snippet shows how a vector can be created and destroyed dynamically, and a

possible layout in memory. The pointer to the starting address of the vector itself is usually located either in the stack or in the global data segment; in this example, it resides in the stack:

```
double * theVector = NULL;
int vectorSize = DetermineInputDataSize();

theVector = new double[vectorSize];

/*... Vector elements are used here ...*/

delete[] theVector;
```

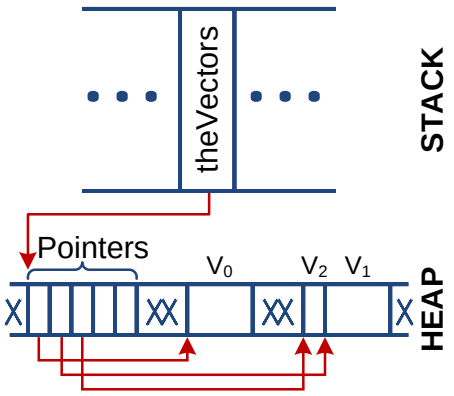


Vectors allocated in this way are amenable to optimizations introduced by cache memories almost as if they were statically allocated vectors, particularly if there are lots of sequential accesses to the vector once it is allocated. The main difference appears if the application creates and destroys multiple instances of small vectors. If a static vector existed, then all the associated memory positions could remain loaded in the corresponding cache lines and directly accessed every time the application used the vector. However, every new instance of a dynamically allocated vector can be created in a different memory position, even if only one is alive at a time, adding to the number of potential cache misses. This increases also the chances of an address collision with other data objects that forces evictions of cache lines, increasing the traffic between elements in the memory subsystem.

Several previous works focused on adapting techniques already developed for the management of static data (i.e., statically allocated during the design phase either in the global data segment or in the stack) to dynamically allocated vectors such as the ones explained here. For instance, [MCB+04] explored the possibility of allocating arrays (vectors) at run-time to exploit the characteristics of multi-banked DRAMs.

2. Creation of a collection of data objects with an unknown cardinality. The previous point shows the case when the size of a vector is unknown; however, the number of vectors required may also be unknown, for example, if they are created inside a loop with a data-dependent number of iterations. In those cases, instead of creating a vector of objects with a worst-case number of entries, the programmer can create a vector of pointers that acts as an index. The objects will be created and linked as needed. This technique, which can be generalized to any type of structures or objects, may reduce the memory footprint of the application significantly because it allocates space only for the objects that are currently needed, plus the size of the vector of pointers. The following code snippet illustrates the concept with a vector that holds pointers to other dynamic vectors of varying sizes that are in turn created and destroyed dynamically:

```
// vectorSizes[ii] contains the size of the
// vector at theVectors[ii]
int numVectors = 0;
int * vectorSizes = NULL;
double ** theVectors = NULL;

numVectors = CalcNumberOfInputVectors();
vectorSizes = new int[numVectors];
theVectors = new double * [numVectors];

for (int ii = 0; ii < numVectors; ++ ii)
{
      vectorSizes[ii] = GetVectorSize(ii);
      theVectors[ii] = new double[vectorSizes[ii]];
}

/*... Do something useful ...*/

/* Delete all the vectors */
for (int ii = 0; ii < numVectors; ++ ii)
      delete[] theVectors[ii];
delete[] theVectors;
delete[] vectorSizes;
```

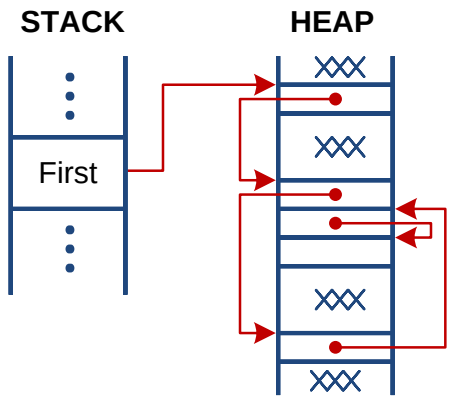


In this example, the number of elements in each of the dynamic vectors is known when the vectors are created: The vectors cannot grow or decrease once they are allocated until they are destroyed. However, the application can destroy an individual vector and create a new one with a different size, updating the pointer in the index.

3. Construction of dynamic data types (DDTs), which are usually complex structures of nodes "linked" through pointers, such as lists, queues, trees, tries, etc., and whose components are not necessarily in contiguous memory addresses. Many languages such as C++, Java or Python offer a standard library of ready-to-use DDTs that usually includes iterator-based or associative containers and provides a smooth method to group data objects, hence providing access to dynamic memory at a high-level of abstraction. A relevant property of DDTs is that the access time to the elements is potentially variable. For example, in a linked list the application has to traverse the $n-1$ first elements before getting access to the n^{th} one.

This case is the most flexible because it can deal both with unknown size and with unknown cardinality in the application data. The DDT itself is constructed as new objects are created; therefore, no worst-case provisions are needed. The following fragment of code shows a simplified example for the construction of a linked list. Each of the nodes may be physically allocated at any possible position in the heap:

```
struct TNodo {
      TNodo * next;
      int data;
};

TNodo * first = NULL;
TNodo * it = NULL, * aux = NULL;

first = new TNodo;
first->next = NULL;
first->data = 0;

/* Add elements */
it = first;
while (MoreElements()) {
      it->next = new TNodo;
      it = it->next;
      it->next = NULL;
      it->data = GetDataElement();
}

/*... Do something useful ...*/

/* Delete elements */
it = first;
while (it != NULL) {
      aux = it;
      it = it->next;
      delete aux;
}
```



This model can be composed as needed, for instance, building a vector of pointers to a previously unknown number of lists each with an indeterminate number of elements; or building lists of lists if the number of lists has to be adapted dynamically. The possible combinations are endless. Figure A.1 shows examples of different DDTs and their organizations. The nodes of the DDTs may contain the data values themselves, or they may store a pointer to another dynamic object. The second possibility requires additional memory to store the pointer to the object (plus the memory overhead introduced by the DMM to allocate it), but it offers advantages such as the possibility of creating variable-sized objects, creating empty nodes (e.g., through the use of `NULL` pointers) and separating the placement of the DDT nodes from the placement of the contained objects [CDL99, DAV+04]. This last transformation may improve data structure traversals because the nodes, which hold the pointers to the next ones, can be tightly packed into an efficient memory while the data elements, which may be much bigger and less frequently accessed, are stored in a different one.

As a final remark, although the concept of DDT is applied to the whole dynamic structure, the allocation and placement are executed for every instance of each node in the DDT.

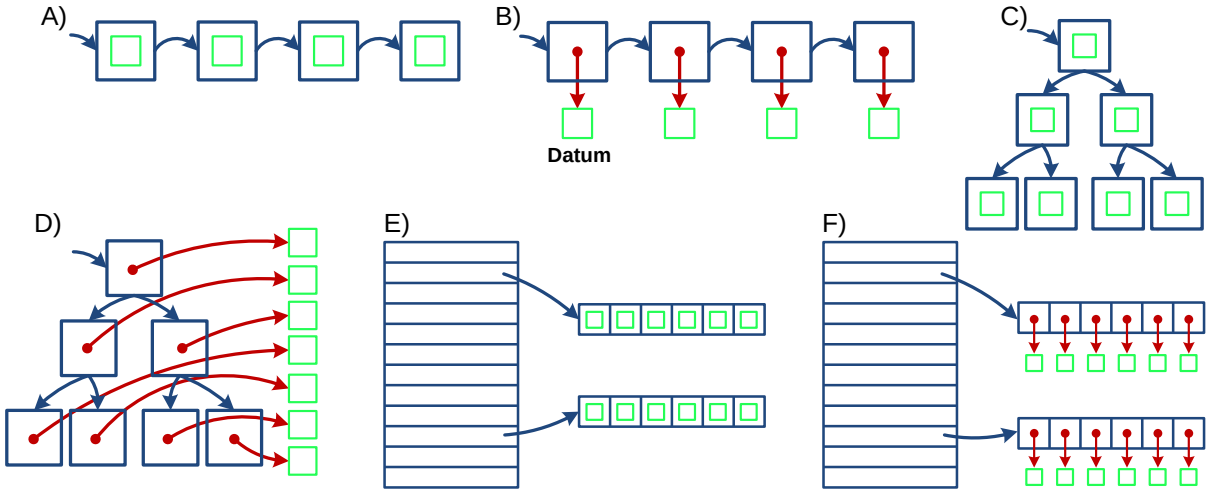


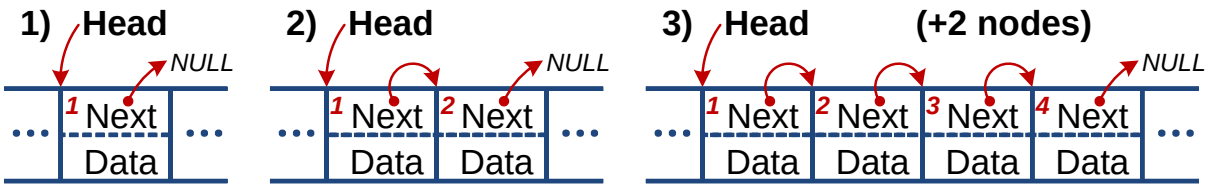
Figure A.1.: Several examples of DDTs: A) Linked list with data in the nodes. B) Linked list with pointers to external objects in the nodes. C) Tree with data in the nodes. D) Tree with pointers to external objects in the nodes. E) Open hash table with data in the nodes of each entry's list. F) Open hash table with pointers to external data in the nodes of each entry's list.

A.2. Impact of linked data structures on locality

This section presents three examples of increasing complexity that illustrate different situations that arise during the utilization of DDTs and their possible impact on cache memories. The first one uses a linked list to show the effect of element removal and the interactions with the state of the dynamic memory manager (DMM). The second example uses an AVL tree to explain that just the internal organization of the DDT can alter the locality of elements, even without deletions, and how different traversals affect also spatial locality. Finally, the third example justifies that the behavior observed with the AVL trees appears also when working with random data and explains the relation between tree nodes and cache lines.

Example A.2.1 A simple example with the list DDT

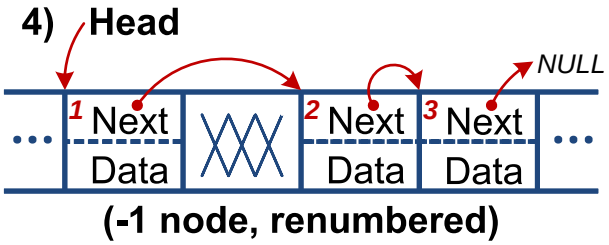
Let us consider an application that uses a linked list. We can assume for now that the DMM keeps a list of free blocks. Every time the application needs to insert a new node in the list, it issues an allocation request to the DMM. If the request can be served with any of the previously existing free blocks, then that block is assigned. Otherwise, the DMM gets a new block of memory from the system resources (e.g., using a call to `mmap` *or* `sbrk`*). In this simple scenario, the first nodes allocated by the application may get consecutive memory addresses (they are allocated consecutively from a single block of system resources):*



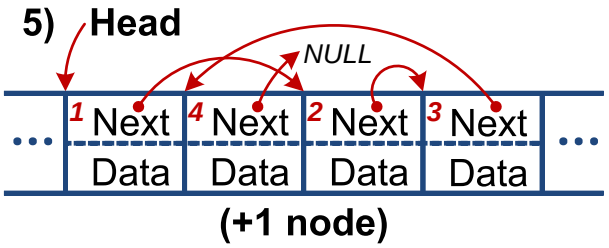
The first node of the list is created and its pointer to the next element ("Next") is initialized to `NULL` *to signal that this is also the last element of the list. The application keeps track of the first node in*

the list with a special pointer, "Head," that can be created as any regular global, stack or dynamically-allocated variable. When new elements are appended at the end of the list, the application traverses the list starting at the node pointed to by "Head" and stopping at the node whose "Next" field has the value `NULL`*. The new node is linked by updating the "Next" field of the formerly last node with the address of the new one. The invariant of this DDT says that the "Next" field of the last node has the value* `NULL`*.*

At a later point, the application does not need the second node any longer and thus, destroys it. The node is unlinked from the logical structure of the list and the released memory space becomes available to the DMM for assignment in later allocations. It becomes immediately noticeable that the first and (now) second nodes are no longer situated in consecutive memory addresses:



Next, the application receives a new data element and needs to append it at the end of the linked list. However, the DMM has now one suitable block in the list of free blocks. Therefore, it uses that block to satisfy the new memory request:



This simple example shows that although the logical structure of the DDT is preserved, a single element removal alters the spatial locality of the list nodes. After a number of operations, each consecutive node may be in a different cache line. This effect may lead to increased traffic between the cache and main memory and unwanted interactions among nodes of different DDTs. Furthermore, the spreading of logical nodes across memory addresses does not only depend on the operations performed on the DDT; it depends also on the previous state of the DMM and the interactions with operations performed on other DDTs.

The effects of list traversals on the performance of cache memories are an interesting topic. In the worst case, one data node will contain a pointer to the next node and a pointer to the data element corresponding to that node (or a single integer number). Assuming a 32-bit architecture with 64-byte cache lines, every cache line will have 64 bits of useful data for every 512 bits of data storage (or two 32-bit words for every sixteen words of storage). If the number of elements in a list becomes sufficiently large so that every node access requires fetching a new cache line from main memory during every traversal, then an 87.5 % of the data is transferred without benefit. A careful programmer or compiler can insert prefetch instructions to hide the time required for the data transfers. If the address of the next node can be calculated quickly and there is enough work to perform on every node, then the delay may be completely hidden. However, energy consumption is a different story. Every data transfer consumes a bit of energy, regardless of whether the data is later employed or not.

. . .

Some data structures depend on a correct organization of their internal nodes to guarantee predictable complexity orders. For example, search trees [BB96] require a certain balance between the left and right branches of every node in order to guarantee a search time in the order of $\mathcal{O}(log_2 n)$. However, the order in which the data elements are inserted and their actual values influence the internal organization of the dynamic data structure itself. Self-balanced structures such as AVL or red-black trees have been designed to palliate this problem. An AVL tree [AVL62] is a binary search tree in which the difference between the weights of the children is at most one for every node. AVL trees keep the weight balance recursively. If any action on the tree changes this condition for a node, then an operation known as a "rotation" [BB96] is performed to restore the equilibrium. These rotations introduce an additional cost; however, under the right circumstances, it is distributed among all the operations, yielding an effective *amortized cost* of $\mathcal{O}(log_2 n)$. How does it relate to the problem presented in this work? The internal reorganization of the nodes in the AVL tree changes the logical relations between them, but their placement (memory addresses) was already fixed by the DMM at allocation time.

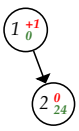
Example A.2.2 A more complex example with AVL trees
Consider now the case of an AVL tree with the following definition:

Offset	*Size*	*Field declaration*
0	4	*UINT32 key*
4	4	*TAVLNode * parent*
8	4	*TAVLNode * leftChild*
12	4	*TAVLNode * rightChild*
16	4	*TData * data*
20	1	*INT8 balance*

Assuming 32*-bit pointers and* 32*-bit padding for the last field, the size of the nodes is* 24 B*. We can study the construction of the tree as the integer numbers from* 1 *to* 12 *are inserted in order:* 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 *and* 12*. First, the number "*1*" is inserted in the empty tree:*

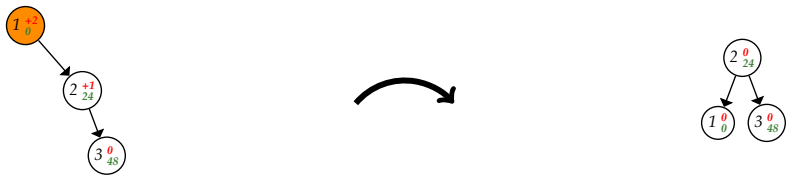
The black number (left) represents the value stored in the node. The red number (right, up) represents the balance factor: −1 *if the left child has a bigger weight,* +1 *if the right child has a bigger weight,* 0 *if both children have the same weight. An absolute value bigger than* 1 *means that the node is unbalanced and a rotation must be performed. Finally, the green number (right, bottom) represents the memory address of the node, as assigned by a hypothetical DMM manager.*

*When number "*2*" is inserted it goes to the right child, as is customary in binary search trees for values bigger than the value in the root node:*

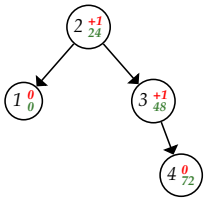


*The new node is balanced because both (null) children have equal weight. However, the root node is unbalanced towards the right child, although still inside the allowed margin. When number "*3*" is*

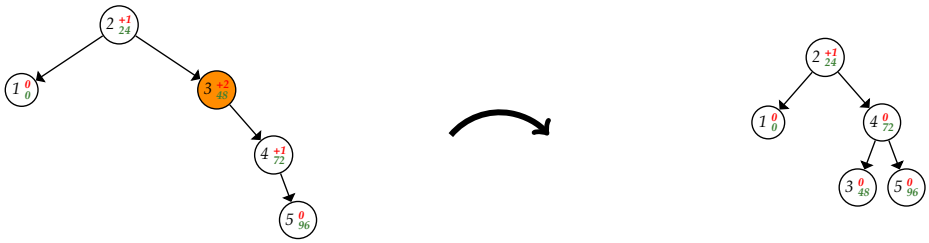
inserted, it goes to the right child of the root node because its value is bigger. Then, as it is also bigger than "2", it goes again to the right child. The root node becomes completely unbalanced (weight +2). In this case, a simple rotation towards the left is enough to restore the balance in the tree:



After the previous rotation, the balance factor of all the nodes is restored to 0. When the value "4" is added to the tree, it becomes again the rightmost child of the whole tree. Notice the new weight balances along the tree. The node that contains the value "3" has a right son and no left son, so it has a balance of +1. The node "2" has a right son with a depth of 2 and a left son with a depth of 1; therefore, this node has a balance of +1. The tree is still globally balanced:

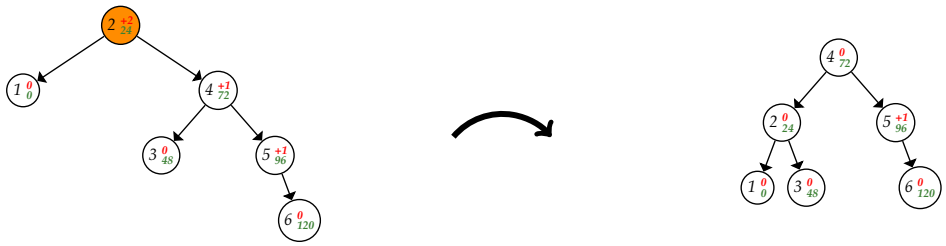


Adding the next value, "5", unbalances the tree again. This time, the node "3" has a right son of depth 2 and a left son of depth 0, so its own balance is +2. A simple rotation is again enough to fix the subtree:[1]

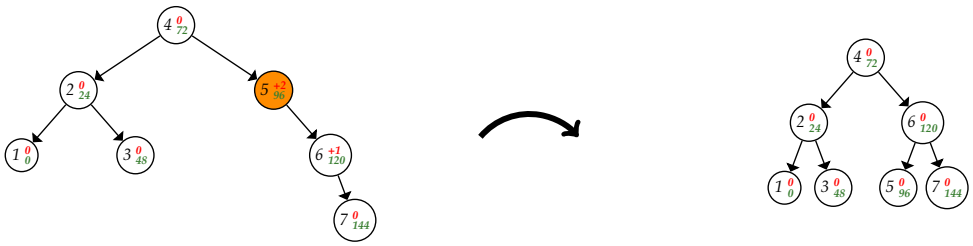


When "6" is added to the tree, the situation gets a bit more complex. The value is again added as a node in the rightmost part of the tree. This time the root of the tree itself becomes unbalanced: Its right child has a maximum depth of three levels while its left child has a depth of 1, giving a total balance of +2. However, a simple rotation as the ones performed before is not enough as it would leave "4" at the root, but with three children: "2", "3" and "5". A complex rotation *is needed in cases like this. After the rotation, node "4" becomes the new root and node "2" its left child. Additionally, the former left child of "4" becomes now the right child of "2". This is acceptable because binary search trees require only that the values of a node left children are smaller than its own value and the values of all right children, bigger. In the original tree node "3" was on the right part of "2", so it was bigger than it. In the new tree, it is also at the right side of "2", preserving the requirement. In respect to "4", which becomes the new root of the tree, both nodes, "2" and "3", are smaller than it, so they can be organized in any way as long as they are both in the left part of the tree. After this new type of rotation, the global tree becomes again balanced:*

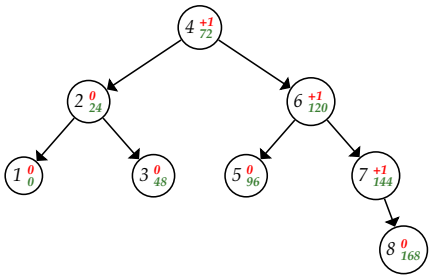
[1] It can be formally proven that the rotations described here restore the global balance of the tree without the need for more rotations in the upper levels of the tree. However, this proof is out of the scope of this work. Further references can be found in the literature, for instance in [AVL62, BB96, Wei95].



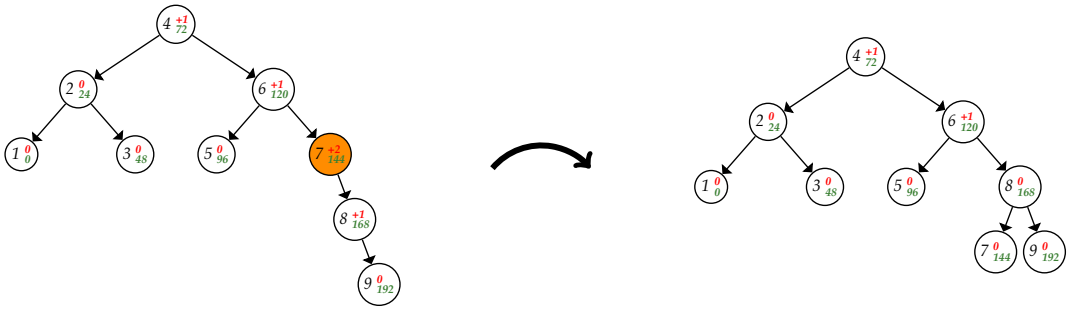
Adding the values from "7" to "11" requires a simple rotation, nothing, a simple rotation, a complex rotation and a simple rotation, respectively:



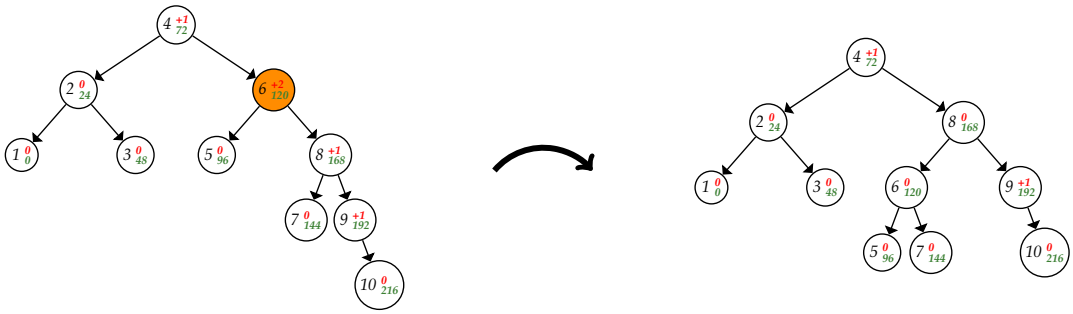
Adding value "7" and rotating.



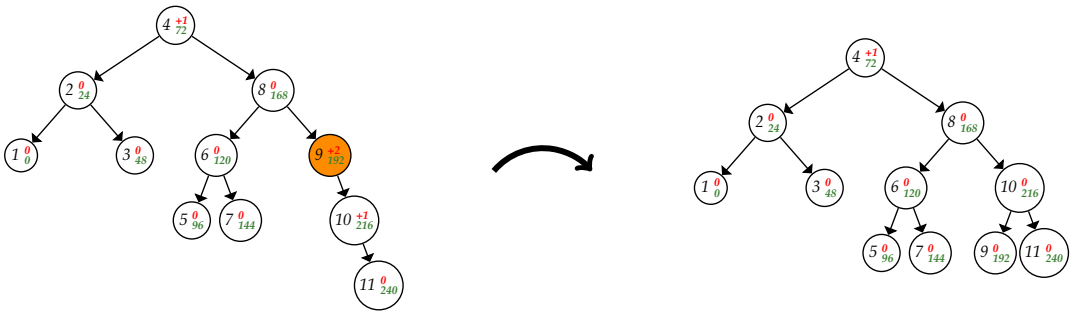
Adding value "8".



Adding node "9" and rotating.

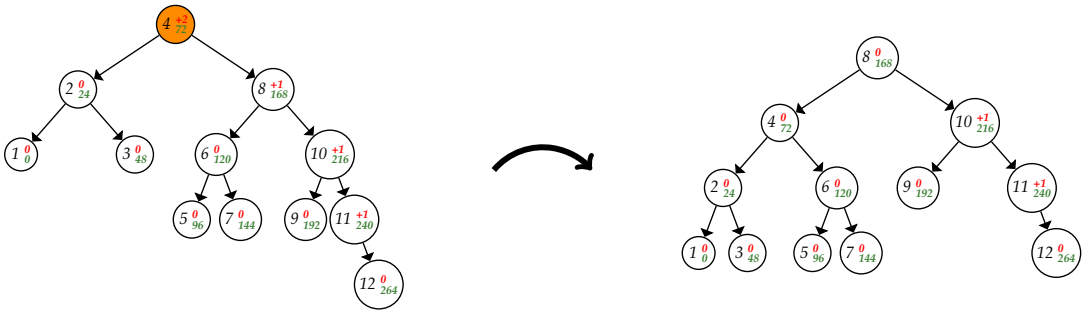


Adding node "10" requires a complex rotation.



Adding node "11" and rotating.

After the (complex) rotation required to add "12", the final configuration of the AVL tree becomes:



Adding node "12" and final configuration.

Assuming that the DMM assigns consecutive addresses to the nodes as they are created, the final layout of the tree in memory is:

1 0	2 24	3 48	4 72	5 96	6 120	7 144	8 168	9 192	10 216	11 240	12 264

The previous example gives a glimpse of the mismatch between memory addresses and logical links. We can explore it further if we analyze the memory accesses needed to perform different operations on the tree. First, consider a complete "in-order" traversal of the tree (e.g., to obtain the ordered list of elements). The following table lists the memory accesses performed by the application. Each cell, numbered from 1 to 23, corresponds to the visit to one node. The first line in a cell shows the node value, its address on memory and the step number. The next lines identify the memory accesses executed by the application while visiting that node: "L" if the application reads the pointer to the left child, "R" if it reads the pointer to the right one, "D" if the application accesses the data element at the node, "P" if the pointer to the parent node is used to go up one level in the tree and "K" if the application reads the key of the node. For each access, the table shows the offset from the start of the node and the corresponding absolute memory address. For example, the line "L: $+8 \rightarrow 176$" means that the application reads the pointer to the left node; as this pointer is at offset $+8$, the application accesses the memory word at position $168 + 8 = 176$.

8 168 (1) L: +8 → 176	4 72 (2) L: +8 → 80	2 24 (3) L: +8 → 32	1 0 (4) L: +8 → 8 D: +16 → 16 R: +12 → 12 P: +4 → 4	2 24 (5) D: +16 → 40 R: +12 → 36	3 48 (6) L: +8 → 56 D: +16 → 64 R: +12 → 60 P: +4 → 52
2 24 (7) P: +4 → 28	4 72 (8) D: +16 → 88 R: +12 → 84	6 120 (9) L: +8 → 128	5 96 (10) L: +8 → 104 D: +16 → 112 R: +12 → 108 P: +4 → 100	6 120 (11) D: +16 → 136 R: +12 → 132	7 144 (12) L: +8 → 152 D: +16 → 160 R: +12 → 156 P: +4 → 148
6 120 (13) P: +4 → 124	4 72 (14) P: +4 → 76	8 168 (15) D: +16 → 184 R: +12 → 180	10 216 (16) L: +8 → 224	9 192 (17) L: +8 → 200 D: +16 → 208 R: +12 → 204 P: +4 → 196	10 216 (18) D: +16 → 232 R: +12 → 228
11 240 (19) L: +8 → 248 D: +16 → 256 R: +12 → 252	12 264 (20) L: +8 → 272 D: +16 → 280 R: +12 → 276 P: +4 → 268	11 240 (21) P: +4 → 244	10 216 (22) P: +4 → 220	8 168 (23) P: +4 → 172	

Therefore, during the traversal, the application accesses the following memory positions: 176, 80, 32, 8, 16, 12, 4, 40, 36, 56, 64, 60, 52, 28, 88, 84, 128, 104, 112, 108, 100, 136, 132, 152, 160, 156, 148, 124, 76, 184, 180, 224, 200, 208, 204, 196, 232, 228, 248, 256, 252, 272, 280, 276, 268, 244, 220 and 172. This access pattern does not exhibit an easily recognizable form of spatial locality.

As the second tree operation, consider the retrieval of the data element corresponding to the key "5." The application visits the nodes "8," "4" and "6," accessing the memory addresses 168, 176, 72, 84, 120, 128, 96 and 112:

8 168 (1) K: +0 → 168 L: +8 → 176	4 72 (2) K: +0 → 72 R: +12 → 84	6 120 (3) K: +0 → 120 L: +8 → 128	5 96 (4) K: +0 → 96 D: +16 → 112

Finally, the insertion of the last node, "12," requires the following accesses, not counting the balance calculation in the branch up to "4" and the accesses required to perform the rotation at the root level:

4 72 (1) K: +0 → 72 R: +12 → 84	8 168 (2) K: +0 → 168 R: +12 → 180	10 216 (3) K: +0 → 216 R: +12 → 228	11 240 (4) K: +0 → 240 R: +12 → 252
12 264 (5) K: +0 → 264 P: +4 → 268 L: +8 → 272 R: +12 → 276 D: +16 → 280 Balance: +20 → 284			

For the accesses included in the table, the visited memory addresses are: 72, 84, 168, 180, 216, 228, 240, 252, 264, 268, 272, 276, 280 and 284.

Up to now, we have considered the memory accesses executed by the application only to get a sense of the effect of the intricacies of DDTs in their order. However, we can also do a few quick estimations of the cost associated with them using different memory elements. First, let us consider a simple system with a cache memory, its degree of associativity irrelevant as long as it has a capacity of at least 512 B. For the cache line size, we can explore the cases of 4 and 16 words per cache line. We can calculate the number of accesses to the main DRAM executed during the insertion of element "12," which involves 14 words of memory, starting from an empty cache condition:

- a) Lines of 16 words. Cache lines accessed: 1, 1, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 4 and 4.
  4 different lines accessed. 4 lines $\times$ 16 words $=$ 64 accesses to the DRAM.
  Overhead: $(64 - 14)$ / $14 = 3.57$.
- b) Lines of 4 words. Cache lines accessed: 4, 5, 10, 11, 13, 14, 15, 15, 16, 16, 17, 17, 17 and 17.
  9 different lines accessed. 9 lines $\times$ 4 words $=$ 36 accesses to the DRAM.
  Overhead: $(36 - 14)$ / $14 = 1.57$.

Second, consider a system with a scratchpad (SRAM) memory. As every word in the memory is independent, the processor accesses only those positions referenced by the application. Only 14 memory accesses are performed, in 14 memory cycles, with no energy or latency overheads (i.e., overhead is 1.0).

Finally, a system with only SDRAM would also execute only 14 accesses. Leaving aside the requirements to open the appropriate DRAM row, and assuming no row conflicts, an SDRAM would serve the accesses in 14 memory cycles. A DDR-SDRAM would be able to serve the accesses in only 11 memory cycles[2] because some of them would exploit its double data rate capabilities.

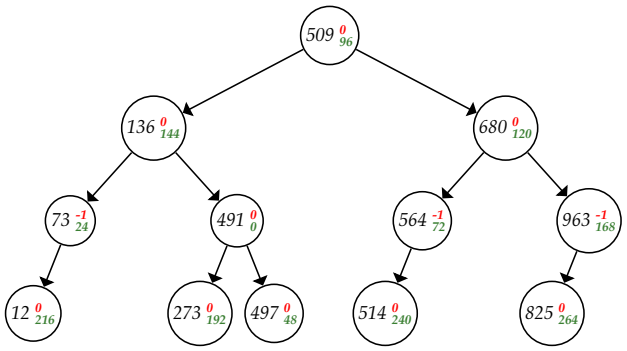
These calculations are oversimplified estimations. Nonetheless, the relevant factor is that the systems without cache memory do not waste energy in unneeded operations in cases such as this one. Cache performance would improve drastically with further high temporal locality traversals as many accesses would be served without accessing the external DRAM. However, the effect is not negligible when the number of nodes in the tree increases and the logical connections become more scattered over the memory space, with every operation accessing different subsets of nodes. Additionally, the mixed pattern of allocations and deallocations will cause over time a dispersion of the addresses assigned to related nodes. Even worse, the application will probably interleave accesses to the tree with accesses to other data structures, or the cache may be shared with other threads. The additional pressure over the cache will force more evictions and more transfers between levels, exacerbating the ill effects of accessing, transferring and storing data words that are not going to be used. Finally, cache efficiency suffers also from the fact that cache line size and object size do not necessarily match. Common options are padding (area and energy waste in storing and transferring filler words) or sharing cache lines with other objects (energy waste transferring words not used, possible false sharing phenomena in multiprocessor environments [HS12, p. 476]).

• • •

[2] An on-chip scratchpad (SRAM) or cache memory usually works at a higher frequency than an external DRAM chip. Therefore, in this discussion I refer to memory cycles in contraposition to processor cycles or real time.

The elements that we inserted in the AVL tree during the previous example were already ordered according to their keys. That forced many rotations in the tree. However, adding elements in random order produces a similar scattering of logical nodes in memory addresses because although less rotations are performed, randomly inserted keys go randomly towards left or right children. In the end, nodes that were created in contiguous memory addresses (assuming again a simplistic DMM model) become logically linked to nodes in very different addresses.

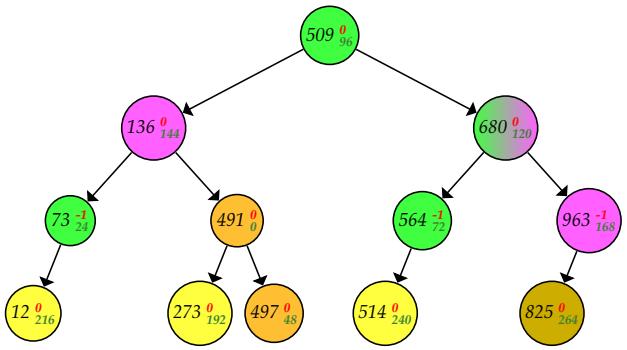
Example A.2.3 Another AVL example, with random data
Repeat the AVL experiment with twelve random numbers: 491, 73, 497, 564, 509, 680, 136, 963, 273, 12, 514 *and* 825. *This is the final configuration of the tree:*



And the layout of the tree nodes in memory is:

491 0	73 24	497 48	564 72	509 96	680 120
136 144	963 168	273 192	12 216	514 240	825 264

In order to get a better picture of the spreading of logical nodes over physical cache lines, we can color the previous graph assigning a different color to each cache line. The nodes arrive consecutively and are assigned successive memory addresses, thus using consecutive cache lines. However, the position that they occupy in the logical structure is very different and can potentially change along the DDT lifetime.



Colored tree for 12 *random numbers (*16 *words or* 64 *bytes per cache line).*

With the memory layout of Example A.2.3, a fetch of the data associated to the key value "273" requires visiting the nodes "509," "136" and "491" with the following access pattern:

509 96 (1) K: +0 → 96 L: +8 → 104	136 144 (2) K: +0 → 144 R: +12 → 156	491 0 (3) K: +0 → 0 L: +8 → 8	273 192 (4) K: +0 → 192 D: +16 → 208

According to this pattern, the application reads memory addresses in the following sequence: 96, 104, 144, 156, 0, 8, 192 and 208. Making similar assumptions to that in the previous example, we can calculate the number of accesses to the main DRAM for several cache configurations and reach similar conclusions:

- a) Lines of 16 words. Cache lines accessed: 1, 1, 2, 2, 0, 0, 3 and 3.
  4 different lines accessed. $4\,\text{lines} \times 16\,\text{words} = 64$ accesses to the DRAM.
  Overhead: $(64 - 8)\ /\ 8 = 7$.
- b) Lines of 4 words. Cache lines accessed: 6, 6, 9, 9, 0, 0, 12 and 13.
  5 different lines accessed. $5\,\text{lines} \times 4\,\text{words} = 20$ accesses to the DRAM.
  Overhead: $(20 - 8)\ /\ 8 = 1.5$.

A.3. In summary: DDTs can hinder cache memories

Cache memories rely on the exploitation of the locality properties of data accesses by means of prefetching and storing recently used data. However, the use of DDTs creates important issues because *logically-adjacent linked nodes are not necessarily stored in consecutive memory addresses* (the DMM may serve successive requests with unrelated memory blocks and nodes may be added and deleted at any position in a dynamically linked structure), breaking the spatial locality assumption, and, in some structures such as trees, *the path taken can be very different from one traversal to the next one*, thus hindering also the temporal locality.

These considerations support the thesis defended in this work: That embedded systems with energy or timing constraints whose software applications have a low data-access locality due to the use of DDTs, which is especially common in object-oriented languages, should be designed considering the utilization of explicitly addressable (i.e., non-transparent) memories rather than caches. In that scenario the placement problem becomes relevant.

Appendix B

Energy efficiency, Dennard scaling and the power wall

Technological and lithographic advances have allowed for an exponential increase on the number of transistors per chip for more than fifty years. This trend was predicted by Gordon Moore in 1965 [Moo65] and revised in 1975 [Moo75]: The number of transistors that could be integrated in a device with the lowest economic cost would double every year (two years in the revised version). Computer architects exploited these extra transistors (and those also from increasing wafer sizes) adding new capabilities to microprocessors (for instance, exploiting more ILP) with important performance gains. Therefore, performance has actually increased at an even higher rate than transistor density. Figure B.1 shows the evolution of computer performance for the last 60 years.

In light of this increasing integration capabilities, concerns about heat dissipation appeared early on. Moore himself pointed out in his 1965 work that power dissipation density should be constant: *"In fact, shrinking dimensions on an integrated structure makes it possible to operate the structure at higher speed for the same power per unit area."* A few years later, Dennard et al. [DGRB74] demonstrated this proposition for MOSFET devices, in what would be known as "Dennard scaling:" The power dissipation of transistors scales down linearly on par with reductions in their linear scale. In essence, a scaling factor of $1/k$ in linear dimensions leads to a reduction of voltage and current of $1/k$, a power dissipation reduction of $1/k^2$ and, therefore, a constant power density. As the authors point out (on page 265):

> "[...T]he power density remains constant. Thus, even if many more circuits are placed on a given integrated circuit chip, the cooling problem is essentially unchanged."

Thanks to Dennard scaling, every time the size of transistors was reduced, more transistors could be integrated into a single circuit while keeping a constant power requirement. This means that the energy efficiency of the circuits improved exponentially for several decades. Figure B.2 shows the evolution of computer performance per each kWh, that is, their energy efficiency. For all this time, computer architects could put to use an exponentially growing amount of transistors for essentially the same energy budget or, once reached a minimum threshold, design mobile devices with good enough capabilities and decreasing energy demands.

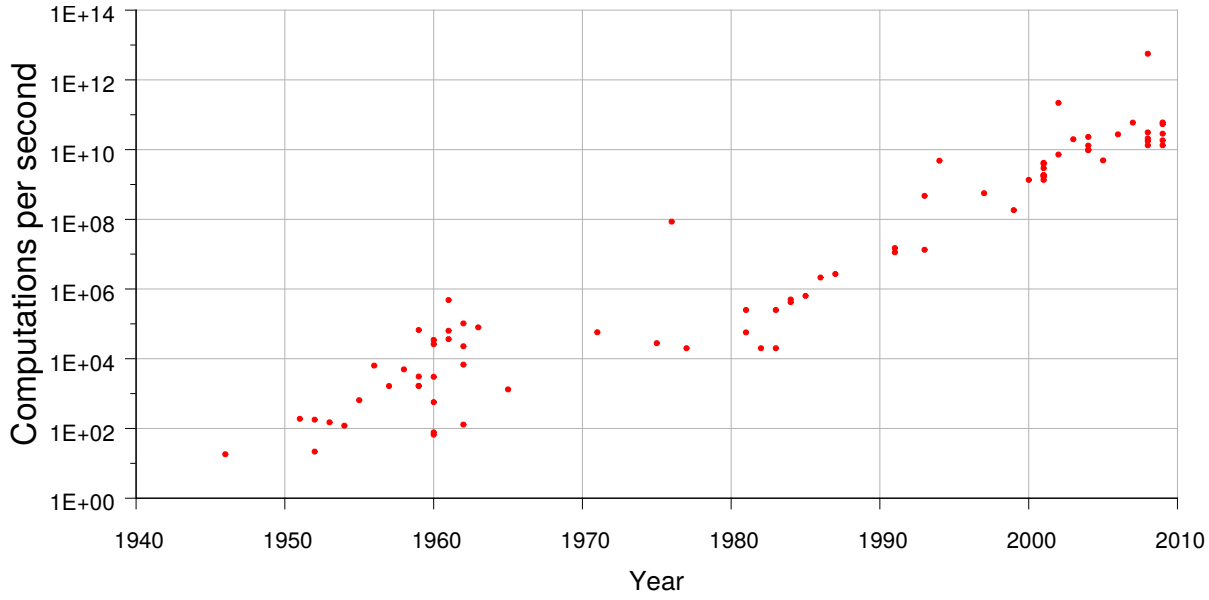


Figure B.1.: Evolution of computer performance during the last 60 years (logarithmic scale). This figure has been replotted with data from [KBSW11] "WEB Extra appendix," accessible at `http://doi.ieeecomputersociety.org/10.1109/MAHC.2010.28`. The authors calculated that, for personal computers (PCs) alone, performance has doubled every 1.52 years, corresponding to the popular interpretation of Moore's law for increases on performance.

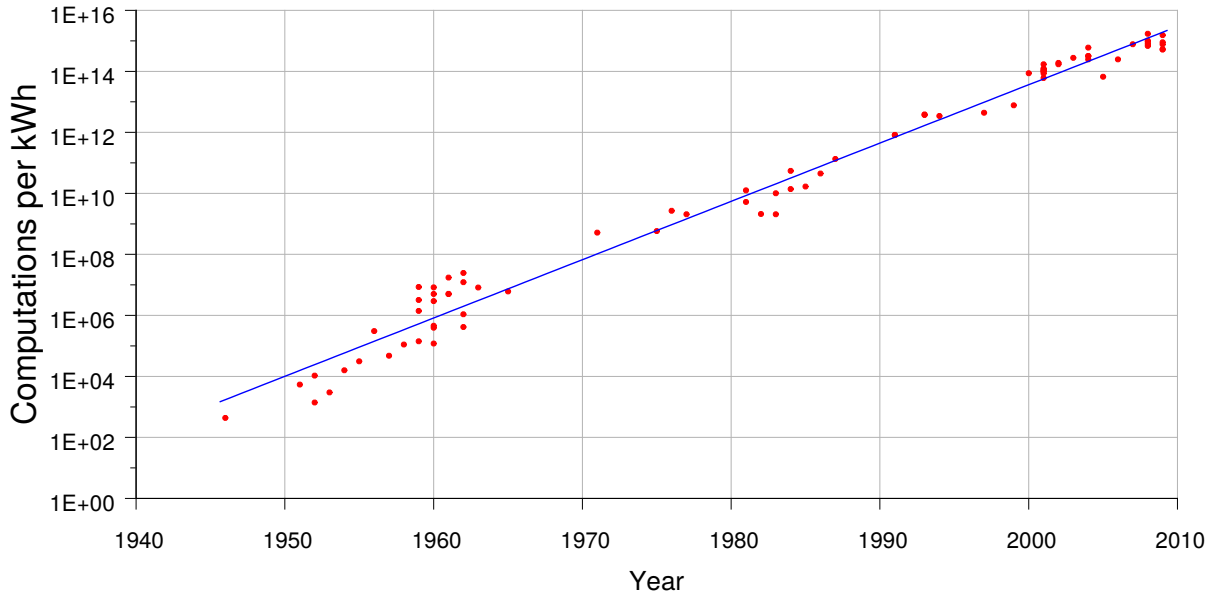


Figure B.2.: Evolution of the energy efficiency of computers during the last 60 years (logarithmic scale). This figure has been replotted with data from [KBSW11] "WEB Extra appendix," accessible at `http://doi.ieeecomputersociety.org/10.1109/MAHC.2010.28`. According to the authors of that work, the energy efficiency of computers has doubled every 1.57 years.

Dennard scaling hold true until about 2004, when it became really difficult to continue reducing the voltage along with linear dimensions. Many techniques have been used to palliate the continuous obstacles [Boh07], of which the main one is that standby power increases significantly as the threshold voltage is reduced. This problem was hinted by Dennard et al. in their original paper [DGRB74] as an issue with the scalability of subthreshold characteristics and further analyzed in a follow up twenty years later [DDS95]:

> "[...] Therefore, in general, for every 100 mV reduction in V_t the standby current will be increased by one order of magnitude. This exponential growth of the standby current tends to limit the threshold voltage reduction to about 0.3 V for room temperature operation of conventional CMOS circuits."

The consequences of the difficulties to scale down the threshold voltage are two-fold. First, reducing the power-supply voltage brings the electrical level for the logical "1" closer to that of the logical "0;" hence, it becomes more difficult to distinguish them reliably. Second, and more importantly, the energy density of the circuits is not (almost) constant anymore, which means that power dissipation becomes a much more pressing concern. The situation we face nowadays is therefore that we can integrate more transistors in a device than what we can afford to power at the same time [EBS+11, MH10]: This is the aforementioned "power wall." Computer architects can no longer focus on improving peak performance relying on technology improvements to keep energy consumption under control. Novel techniques are required to limit energy consumption while improving performance or functionality.

Appendix C

Data placement from a theoretical perspective

Computational complexity is an evolving theoretical field with important consequences on the practice of computing engineering. Interested readers can delve deeper into it with the classic textbook from Cormen et al. [CLRS01, Chap. 34–35]. In this appendix I offer a glimpse on the world of computational complexity, introducing the concepts that can help to understand the complexity of the data placement problem.

C.1. Notes on computational complexity

Problems that can be solved exactly with an algorithm that executes in polynomial time in the worst case are said to be in the $\mathcal{P}$ complexity class. Those problems are generally regarded as "solvable," although a problem with a complexity in the order of $\mathcal{O}(n^{100})$ is hardly easy to solve – anyways, typical examples are in the order of $\mathcal{O}(n^2)$ or $\mathcal{O}(n^3)$ at most, where n is the size of the problem. A very important property of $\mathcal{P}$ is that its algorithms can be composed and the result is still in $\mathcal{P}$.

Problems for which no exact algorithm working in polynomial time has ever been devised, but whose solutions – if given by, say, an oracle – can be verified in polynomial time by a deterministic machine are said to be in the $\mathcal{NP}$ class. An interesting remark is that not knowing any algorithm to solve a problem in polynomial time is not the same than being sure that such an algorithm does not exist – and can thus never be found. In fact, we know that $\mathcal{P} \subseteq \mathcal{NP}$, but the question of whether $\mathcal{P} = \mathcal{NP}$ is the holy grail of computational complexity: On May 24, 2000, the Clay Mathematics Institute of Cambridge announced a one million dollar prize for the person who can solve that riddle [CMIC00] as formalized by Cook [Coo00].

$\mathcal{NP}$-complete is a special class of problems in $\mathcal{NP}$. A problem p is said to be in $\mathcal{NP}$-complete if every other problem in $\mathcal{NP}$ can be polynomially reduced to it, that is, if there is a transformation working in polynomial time that adapts the inputs to the other problem into inputs to p and another transformation that converts the solution of p into a solution for the original problem. Therefore, we can informally say that $\mathcal{NP}$-complete contains the hardest problems in $\mathcal{NP}$.

> "[...] This class [$\mathcal{NP}$-complete] has the surprising property that if any

> $\mathcal{NP}$-complete problem can be solved in polynomial time, then every problem in $\mathcal{NP}$ has a polynomial-time solution, that is, $\mathcal{P} = \mathcal{NP}$." [CLRS01, Chap. 34.3]

Strictly speaking, $\mathcal{NP}$-complete contains decision problems, that is, problems that admit "yes" or "no" as an answer. However, we are commonly concerned with problems that require more complex solutions. A common technique is to define the decision problem related to a more general one, so that if the decision problem is shown to be in $\mathcal{NP}$-complete, the general problem is then said to be in the class of $\mathcal{NP}$-hard problems. This transformation can be done, for instance, changing a knapsack problem into a question such as "Can a subset of objects be selected to fill no more than v volume units and with a minimum of b benefit units?"

The difficulty in solving some problems has motivated the development of a complete theory of approximation algorithms. Although many problems are intractable in the worst case, many can be approximated within a determined bound with efficient algorithms. The approximation factor is usually stated as $\epsilon > 0$, so that a typical approximation algorithm will find a solution within a $(1 \pm \epsilon)$ factor of the optimal – the sign depending on whether the problem is a maximization or minimization one. A useful class of problems is FPTAS (fully polynomial-time approximation scheme), which is defined as the set of problems that can be approximated with an algorithm bounded in time both by $1/\epsilon$ and the problem size. However, an FPTAS cannot be found for the most complex instances of the knapsack family of problems unless $\mathcal{P} = \mathcal{NP}$ [Pis95, pp. 22–23, for all this paragraph]. The most complex ones are said to be hard even to approximate. For example:

> "The Multiple Knapsack Problem is $\mathcal{NP}$-hard in the strong sense, and thus any dynamic programming approach would result in strictly exponential time bounds. Most of the literature has thus been focused on branch-and-bound techniques [...]" [Pis95, p. 172]

More recent works, such as the one by Chekuri and Khanna [CK00], propose that the multiple knapsack problem is indeed the most complex special case of GAP that is not APX-hard – i.e., that is not "hard" even to approximate.

C.2. Data placement

The problem of placement for dynamic data objects on heterogeneous memory subsystems is complex to solve. Although not an expert on computational complexity myself, I believe that it is a generalization of the (minimization) general assignment problem (GAP). In this section I compare briefly both problems.

The efforts of theorists and practitioners in computational complexity have been long elicited by a family of problems of which perhaps the simplest is the 0/1 knapsack, in which a set of indivisible objects, each with its own intrinsic value, needs to be fit in a knapsack of limited capacity. The goal is to maximize the value of the chosen objects.[1] The problem can be complicated in several ways. For example, the multiple knapsack problem has several containers to fill, maximizing the aggregate value. Bin packing has the goal of packing all the

[1] Fractional or continuous knapsack, where the objects can be split at any point (e.g., liquids), is sometimes regarded as a different type of problem. However, that problem is also interesting because it can be solved exactly by a greedy algorithm in logarithmic time $\mathcal{O}(n \log n)$ using sorting or even in linear time $\mathcal{O}(n)$ using weighted medians. More importantly, it can be used to quickly obtain upper bounds in branch-and-bound schemes to solve the harder versions [Pis95, p. 18].

objects in a set in the minimum possible amount of containers. The multiple choice knapsack problem divides the objects in different classes from which a maximum number of items can be taken.

Similar problems include also instances of scheduling, where some tasks need to be scheduled on a number of processors (or orders assigned to production lines) to minimize the total execution time. The complexity of the problem increases if the processors have distinct characteristics so that the cost of each task depends on the processor that executes it. An interesting variation is the virtual machine (VM) colocation problem because the size of each VM can vary depending on the rest of machines that are assigned to the same physical server: Quite frequently pages from several VMs will have identical contents and the hypervisor will be able to serve them all with a single physical page [SSS11]. The particularity of this problem is that the set of previously selected objects affects the size or cost of the remaining ones. Although all the knapsack problems (except the continuous one) belong to the category of $\mathcal{NP}$-hard problems – they are "hard" to solve in the worst case – researchers have been able to devise techniques to solve or approximate many cases of practical interest in polynomial time, often in less than one second [Pis95, pp. 9–10].

The general assignment problem (GAP) raises the complexity level even more, because it has multiple containers and the cost and benefit of each object depends on the container into which it is assigned:

> "Instance: A pair (B, S) where B is a set of M bins and S is a set of N items. Each bin $c_j \in B$ has capacity $c(j)$, and for each item i and bin c_j we are given a size $s(i, j)$ and a profit $p(i, j)$." [CKR06]

In this work, I assume that tackling the problem of placement at the level of individual dynamic data objects is unfeasible and therefore I propose to approach it as the placement of DDTs. However, in this form the problem has still more degrees of freedom than other problems from the same family because the number and size of the containers is not fixed. It consists on assigning a set of DDTs to a set of memory resources, without exceeding the capacity of each resource and minimizing the total cost of the application accesses to the data objects. As in the GAP problem, multiple memory resources (containers) exist, each with a different capacity, and the cost of accessing each DDT is different according to the characteristics of each module. However, the size of the containers themselves can vary as well, adjusting to the combined, not added, size of the DDTs that they contain. As in the VM-colocation problem, the size of each DDT depends on the other DDTs (objects) that have been already selected in that resource.[2] Furthermore, for some memory resources, the very cost of accessing a DDT may depend on the other DDTs placed there. That is for instance the case of objects assigned to the same bank of a DRAM where accesses to each one can create row misses to access the others.

Data placement presents an additional difficulty. Hard instances of the previous problems are usually solved with branch-and-bound techniques. To prune the search space and avoid a full exploration, they require a mechanism to calculate an upper bound (for minimization) on the cost of the current partial solution. However, in the data placement problem assessing

[2] In this text I propose to group DDTs with similar characteristics to overcome the inefficiencies in resource exploitation of a static data placement. The size of a group depends on the specific objects in it, hence the similarity with the VM-colocation problem. Memory fragmentation inside the pools also contributes to this effect.

the cost of a partial solution can be quite difficult because it depends on the way that the application uses the placed data objects. Ideally, and assuming that the memory traces obtained during profiling are sufficiently representative, the simulator included in $\mathcal{DynAsT}$ could be used to calculate the exact cost of a complete placement solution. That approach is more difficult for partial solutions, though, as the cost of accesses to data objects belonging to DDTs not yet placed cannot be easily calculated. An option that would give a very coarse upper bound is to assume that all non-placed DDTs are placed in main memory.

Another issue is that the simulation of whole memory traces, although a fast process, may require a significant amount of time, especially if the estimation has to be calculated for many different nodes during the search process. One possibility could be using high-level estimators to produce a rough approximation of the cost of a solution. Those estimators would simply multiply the total number of accesses to the instances of each DDT by the cost of each access to the memory module where they are (tentatively) placed. For DDTs placed in SRAMs, the estimation should be pretty close to the real cost. For those placed on DRAMs, on the contrary, the estimation can deviate significantly from the real value as interactions between accesses to different DDTs in the banks of a DRAM can force an indeterminate number of row misses, with the corresponding increase in energy consumption and access time. In any case, I leave the exploration of such possibilities to further work.

Appendix D

Level-zero metadata file format

For reference purposes, Table D.1 presents the packet types used in the profiling format of Chapter 5 and their corresponding fields. The entries `AllocBegin` and `AllocEnd`, and `DeallocBegin` and `DeallocEnd`, are used in tandem to represent a single allocation or deallocation event in the application. They are explicitly separated to ease the analysis of the memory accesses performed during the (de)allocation process itself. The identifiers `seqId`, `varId`, `allocateId`, `scopeId` and `thredId` are the unique identifiers assigned by the programmer (or the instrumentation mechanism) to each element.

For sequences (vectors), `elementType` is the internal type of the elements in the sequence, obtained using the C++ `typeid` operator with run-time type information (RTT). `seqID` is the unique identifier assigned by the programmer during instrumentation. `instanceID` is a unique identifier created for each sequence instance. The difference between `seqID` and `instanceID` is that the first is assigned by the programmer and tags every dynamic data type variable or DDT, whereas the second is generated automatically and identifies different instances of the same sequence. In comparison, different instances of a dynamic variable have the same identifier and are distinguished by their memory address. The creation of a static sequence identifies it by its `instanceId`; a dynamically-allocated sequence (e.g., with `new` inside a loop) has also an `Alloc` entry. Finally, `elementSize` is the size in bytes of the elements contained in the sequence.

The sequences of the profiling library are implemented as arrays that grow as more space is needed (this does not affect the performance of the final application). Iterators are just pointers to elements in the array; thus, iterator operations record also the address pointed by the iterator and the number of elements affected.

Table D.1.: Structure of the profiling packet used for level-zero metadata extraction.

Log packet	Fields						
VectorConstruct	threadId	elementType	seqId	instanceId	elementSize		
VectorDestruct	threadId	elementType	seqId	instanceId	elementSize		
VectorResize	threadId	elementType	seqId	instanceId	elementSize		
IteratorNext	threadId	elementType	seqId	instanceId	elementSize	address	
IteratorPrevious	threadId	elementType	seqId	instanceId	elementSize	address	
IteratorAdd	threadId	elementType	seqId	instanceId	elementSize	address	offset
IteratorSub	threadId	elementType	seqId	instanceId	elementSize	address	offset
IteratorGet	threadId	elementType	seqId	instanceId	elementSize	address	
VectorGet	threadId	elementType	seqId	instanceId	elementSize	index	
VectorAdd	threadId	elementType	seqId	instanceId	elementSize	index	
VectorRemove	threadId	elementType	seqId	instanceId	elementSize	index	
VectorClear	threadId	elementType	seqId	instanceId	elementSize		
VarRead	threadId	varId	address	size			
VarWrite	threadId	varId	address	size			
AllocBegin	threadId	allocatedId	size				
AllocEnd	threadId	allocatedId	size	address			
DeallocBegin	threadId	allocatedId	address				
DeallocEnd	threadId	allocatedId	address				
ScopeBegin	threadId	scopeId					
ScopeEnd	threadId	scopeId					
ThreadBegin	threadId	oldThreadId					
ThreadEnd	threadId						

Appendix E

Full result tables for the experiments on dynamic data placement

The following tables present the results of the case studies conducted in Chapter 4 for all the platforms explored. Platform configuration names use the same keys as previously. For SRAM-based platforms, all the modules are enumerated. Thus, platform "SRAM: 512B, 1KB, 32KB, 8x512KB" represents a platform with a DRAM and eleven SRAMs: One of 512 B, one of 1 KB, one of 32 KB and eight of 512 KB for a total capacity of 4 228 608 B. Correspondingly, platform "Cache: L1=32KB(A2), L2=256KB(A16)" has a first-level 2-way associative cache of 32 KB and a second-level 16-way associative cache of 256 KB.

Cache memories may specify up to three parameters: Associativity, line-length (16 words by default) and replacement policy (LRU by default). Thus, "Cache 128KB(D)" represents a 128 KB direct-mapped cache (D) with lines of 16 words (W16) and LRU replacement policy. Correspondingly, "Cache: 256KB(A16,W4,Random)" represents a 256 KB 16-way associative cache (A16) with lines of 4 words (W4) and random replacement policy.

Table E.1.: Case study 1: Results for all platforms with Mobile SDRAM.

Platform	**Energy** mJ	**Energy** %	**Time** Cycles ($\times 10^6$)	**Time** %	**Page misses**	**DRAM accesses** ($\times 10^6$)	**Total accesses** ($\times 10^6$)
Only DRAM	360.01	100.0	2 132.5	100.0	15 557 047	87.1	87.1
Cache: L1=4KB(D)	766.51	212.9	2 613.0	122.5	4 482 846	270.7	618.3
Cache: L1=16KB(D)	706.36	196.2	2 309.9	108.3	3 432 408	242.1	562.1
Cache: L1=64KB(D)	601.21	167.0	1 631.1	76.5	27 758	185.2	450.6
Cache: L1=128KB(D)	708.86	196.9	1 626.1	76.3	1 384	184.8	449.8
Cache: L1=256KB(D)	725.84	201.6	1 722.1	80.8	1 384	184.8	449.8
Cache: L1=512KB(D)	774.80	215.2	1 722.1	80.8	1 384	184.8	449.8
Cache: L1=4KB(D), L2=256KB(D)	1 030.30	286.2	2 376.2	111.4	1 384	184.8	1 085.7
Cache: L1=16KB(D), L2=256KB(D)	1 005.83	279.4	2 283.9	107.1	1 384	184.8	996.1
Cache: L1=4KB(A4)	243.29	67.6	1 005.2	47.1	3 686 132	83.1	250.2
Cache: L1=16KB(A4)	172.51	47.9	736.4	34.5	2 771 338	57.5	199.9
Cache: L1=64KB(A4)	12.45	3.5	94.7	4.4	27 372	0.7	88.5
Cache: L1=128KB(A4)	18.12	5.0	91.9	4.3	13 726	0.5	88.1
Cache: L1=256KB(A4)	47.83	13.3	174.5	8.2	80	0.0	87.2
Cache: L1=512KB(A4)	100.80	28.0	174.5	8.2	74	0.0	87.2
Cache: L1=4KB(A4), L2=256KB(A4)	83.21	23.1	352.7	16.5	80	0.0	344.4
Cache: L1=16KB(A4), L2=256KB(A4)	60.85	16.9	271.6	12.7	80	0.0	265.7
Cache: L1=4KB(A8)	215.94	60.0	896.7	42.0	3 561 641	71.1	226.5
Cache: L1=16KB(A8)	168.92	46.9	713.5	33.5	2 694 757	55.3	195.6
Cache: L1=64KB(A8)	11.10	3.1	92.0	4.3	24 433	0.4	88.0
Cache: L1=128KB(A8)	13.67	3.8	87.3	4.1	81	0.0	87.2
Cache: L1=256KB(A8)	20.27	5.6	174.5	8.2	74	0.0	87.2
Cache: L1=512KB(A8)	51.59	14.3	174.5	8.2	74	0.0	87.2
Cache: L1=4KB(A8), L2=256KB(A8)	44.14	12.3	322.8	15.1	74	0.0	315.4
Cache: L1=16KB(A8), L2=256KB(A8)	34.87	9.7	266.8	12.5	74	0.0	261.1
Cache: L1=4KB(A16)	229.66	63.8	894.5	41.9	3 553 122	70.9	226.1
Cache: L1=16KB(A16)	181.21	50.3	713.7	33.5	2 680 351	55.4	195.8
Cache: L1=32KB(A16)	106.71	29.6	434.3	20.4	1 506 189	30.5	147.0
Cache: L1=32KB(A16,W4)	36.08	10.0	275.4	12.9	1 792 566	7.9	102.8
Cache: L1=64KB(A16)	15.94	4.4	88.3	4.1	5 312	0.1	87.3

Table E.1.: Case study 1: Results for all platforms with Mobile SDRAM. (Continued)

Platform	**Energy** mJ	**Energy** %	**Time** Cycles ($\times 10^6$)	**Time** %	**Page misses**	**DRAM accesses** ($\times 10^6$)	**Total accesses** ($\times 10^6$)
Cache: L1=64KB(A16,W4)	2.94	0.8	88.8	4.2	14 844	0.1	87.3
Cache: L1=128KB(A16)	18.01	5.0	87.3	4.1	74	0.0	87.2
Cache: L1=256KB(A16)	21.88	6.1	174.5	8.2	74	0.0	87.2
Cache: L1=256KB(A16,W8)	9.54	2.6	174.5	8.2	136	0.0	87.2
Cache: L1=256KB(A16,W4)	6.03	1.7	174.5	8.2	180	0.0	87.2
Cache: L1=512KB(A16)	30.13	8.4	174.5	8.2	74	0.0	87.2
Cache: L1=1MB(A16)	44.43	12.3	348.8	16.4	74	0.0	87.2
Cache: L1=2MB(A16)	92.42	25.7	348.8	16.4	74	0.0	87.2
Cache: L1=4MB(A16)	185.23	51.4	523.1	24.5	74	0.0	87.2
Cache: L1=4KB(A16), L2=256KB(A16)	64.47	17.9	322.2	15.1	74	0.0	314.8
Cache: L1=16KB(A16), L2=256KB(A16)	51.47	14.3	267.6	12.5	74	0.0	261.9
Cache: L1=32KB(A2), L2=256KB(A16)	52.60	14.6	339.6	15.9	74	0.0	331.7
Cache: L1=32KB(A16), L2=256KB(A16)	33.29	9.2	178.8	8.4	74	0.0	175.9
SRAM: 512B, 1KB, 32KB, 256KB	0.24	0.1	90.1	4.2	0	0.0	87.1
SRAM: 512B, 1KB, 32KB, 2x256KB	0.24	0.1	90.1	4.2	0	0.0	87.1
SRAM: 512B, 1KB, 32KB	10.62	2.9	132.9	6.2	349 784	3.0	87.1
SRAM: 512B, 1KB, 16KB, 32KB	6.39	1.8	112.3	5.3	149 096	1.9	87.1
SRAM: 512B, 1KB, 64KB	0.46	0.1	87.7	4.1	3 495	0.1	87.1
SRAM: 32KB	12.23	3.4	141.1	6.6	446 664	3.3	87.1
SRAM: 64KB	0.93	0.3	88.1	4.1	4 495	0.1	87.1
SRAM: 128KB	0.91	0.3	87.1	4.1	1	0.0	87.1
SRAM: 256KB	1.09	0.3	174.3	8.2	1	0.0	87.1
SRAM: 256KB, 256KB	1.09	0.3	174.3	8.2	1	0.0	87.1
SRAM: 512KB	2.16	0.6	174.3	8.2	1	0.0	87.1
SRAM: 512B, 4KB, 16KB, 32KB, 256KB	0.15	0.0	87.2	4.1	0	0.0	87.1
SRAM: 1MB	2.40	0.7	348.6	16.3	1	0.0	87.1
SRAM: 4x256KB	1.09	0.3	174.3	8.2	1	0.0	87.1
SRAM: 2x512KB	2.16	0.6	174.3	8.2	1	0.0	87.1
SRAM: 2MB	3.22	0.9	348.6	16.3	1	0.0	87.1
SRAM: 8x256KB	1.09	0.3	174.3	8.2	1	0.0	87.1

Table E.1.: Case study 1: Results for all platforms with Mobile SDRAM. (Continued)

Platform	**Energy** mJ	**Energy** %	**Time** Cycles ($\times 10^6$)	**Time** %	**Page misses**	**DRAM accesses** ($\times 10^6$)	**Total accesses** ($\times 10^6$)
SRAM: 4x512KB	2.16	0.6	174.3	8.2	1	0.0	87.1
SRAM: 4MB	6.70	1.9	522.8	24.5	1	0.0	87.1
SRAM: 16x256KB	1.09	0.3	174.3	8.2	1	0.0	87.1
SRAM: 8x512KB	2.16	0.6	174.3	8.2	1	0.0	87.1
SRAM: 4x1MB	2.40	0.7	348.6	16.3	1	0.0	87.1
SRAM: 512B, 1KB, 32KB, 256KB, 8x512KB	0.24	0.1	90.1	4.2	0	0.0	87.1
SRAM: 512B, 1KB, 32KB, 8x512KB	0.27	0.1	90.1	4.2	0	0.0	87.1
SRAM: 8x32KB	0.43	0.1	87.1	4.1	1	0.0	87.1
SRAM: 4x64KB	0.63	0.2	87.1	4.1	1	0.0	87.1
SRAM: 256B, 1MB	1.17	0.3	212.8	10.0	0	0.0	87.1
SRAM: 512B, 1KB, 32KB, 256KB Cache: 4KB(A4,W16)	0.24	0.1	90.1	4.2	0	0.0	87.1
SRAM: 512B, 1KB, 32KB, 256KB Cache: 4KB(A4,W8)	0.24	0.1	90.1	4.2	0	0.0	87.1
SRAM: 512B, 1KB, 32KB, 256KB Cache: 4KB(A4,W4)	0.24	0.1	90.1	4.2	0	0.0	87.1
SRAM: 512B, 1KB, 32KB, 256KB Cache: 4KB(D,W16)	0.24	0.1	90.1	4.2	0	0.0	87.1
SRAM: 512B, 1KB, 32KB, 256KB Cache: 4KB(D,W8)	0.24	0.1	90.1	4.2	0	0.0	87.1
SRAM: 512B, 1KB, 32KB, 256KB Cache: 4KB(D,W4)	0.24	0.1	90.1	4.2	0	0.0	87.1
SRAM: 512B, 1KB, 32KB, 256KB Cache: 16KB(A4)	0.24	0.1	90.1	4.2	0	0.0	87.1
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(A4)	0.24	0.1	90.1	4.2	0	0.0	87.1
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(A16,W16)	0.24	0.1	90.1	4.2	0	0.0	87.1
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(A16,W8)	0.24	0.1	90.1	4.2	0	0.0	87.1
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(A16,W4)	0.24	0.1	90.1	4.2	0	0.0	87.1
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(A16,W16,Random)	0.24	0.1	90.1	4.2	0	0.0	87.1
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(A16,W8,Random)	0.24	0.1	90.1	4.2	0	0.0	87.1
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(A16,W4,Random)	0.24	0.1	90.1	4.2	0	0.0	87.1
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(A4,W16,Random)	0.24	0.1	90.1	4.2	0	0.0	87.1
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(A4,W8,Random)	0.24	0.1	90.1	4.2	0	0.0	87.1
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(A4,W4,Random)	0.24	0.1	90.1	4.2	0	0.0	87.1
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(D,W16)	0.24	0.1	90.1	4.2	0	0.0	87.1
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(D,W8)	0.24	0.1	90.1	4.2	0	0.0	87.1
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(D,W4)	0.24	0.1	90.1	4.2	0	0.0	87.1

Table E.1.: Case study 1: Results for all platforms with Mobile SDRAM. (Continued)

Platform	**Energy** mJ	**Energy** %	**Time** Cycles ($\times 10^6$)	**Time** %	**Page misses**	**DRAM accesses** ($\times 10^6$)	**Total accesses** ($\times 10^6$)
SRAM: 512B, 1KB, 2KB, 4KB, 8KB, 16KB, 32KB, 64KB, 128KB	0.14	0.0	87.2	4.1	1	0.0	87.1
SRAM: 512B, 2x4KB, 2x32KB, 64KB, 128KB	0.16	0.0	87.1	4.1	0	0.0	87.1
SRAM: 512B, 1KB, 2KB, 3x4KB, 7x16KB, 4x32KB	0.13	0.0	87.2	4.1	1	0.0	87.1
SRAM: 512B, 1KB, 2KB, 3x4KB, 7x16KB, 2x32KB, 64KB	0.13	0.0	87.2	4.1	1	0.0	87.1
SRAM: 512B, 1KB, 2KB, 3x4KB, 3x16KB, 2x32KB, 2x64KB	0.13	0.0	87.2	4.1	1	0.0	87.1
SRAM: 512B, 1KB, 2KB, 3x4KB, 3x16KB, 2x32KB, 128KB	0.13	0.0	87.2	4.1	1	0.0	87.1
SRAM: 512B, 2x4KB, 3x32KB, 64KB, 128KB	0.16	0.0	87.1	4.1	0	0.0	87.1
SRAM: LowerBound(Energy↔1KB)	0.10	0.0	87.1	4.1	0	0.0	87.1
SRAM: LowerBound(Energy↔4KB)	0.14	0.0	87.1	4.1	0	0.0	87.1
Cache: LowerBound(D,W4,Energy↔4KB)	0.52	0.1	87.4	4.1	180	0.0	87.2
Cache: LowerBound(D,W8,Energy↔4KB)	1.25	0.3	87.4	4.1	136	0.0	87.2
Cache: LowerBound(D,W16,Energy↔4KB)	13.52	3.8	87.3	4.1	74	0.0	87.2
Cache: LowerBound(A4,W4,Energy↔4KB)	0.75	0.2	87.4	4.1	180	0.0	87.2
Cache: LowerBound(A4,W8,Energy↔4KB)	1.60	0.4	87.4	4.1	136	0.0	87.2
Cache: LowerBound(A4,W16,Energy↔4KB)	4.75	1.3	87.3	4.1	74	0.0	87.2
Cache: LowerBound(A8,W4,Energy↔4KB)	0.72	0.2	87.4	4.1	180	0.0	87.2
Cache: LowerBound(A8,W8,Energy↔4KB)	2.40	0.7	87.4	4.1	136	0.0	87.2
Cache: LowerBound(A8,W16,Energy↔4KB)	7.47	2.1	87.3	4.1	74	0.0	87.2
Cache: LowerBound(A16,W4,Energy↔4KB)	1.25	0.3	87.4	4.1	180	0.0	87.2
Cache: LowerBound(A16,W8,Energy↔4KB)	3.11	0.9	87.4	4.1	136	0.0	87.2
Cache: LowerBound(A16,W16,Energy↔4KB)	15.31	4.3	87.3	4.1	74	0.0	87.2

Table E.2.: Case study 1: Results for all platforms with LPDDR2 SDRAM.

Platform	**Energy** mJ	**Energy** %	**Time** Cycles ($\times 10^6$)	**Time** %	**Page misses**	**DRAM accesses** ($\times 10^6$)	**Total accesses** ($\times 10^6$)
Only DRAM	229.57	100.0	1 315.6	100.0	4 307 514	87.1	87.1
Cache: L1=4KB(D)	521.91	227.3	2 200.4	167.3	3 235 079	498.2	1 065.9
Cache: L1=16KB(D)	534.98	233.0	2 081.6	158.2	2 712 100	473.1	1 016.8
Cache: L1=64KB(D)	599.13	261.0	1 729.2	131.4	21 820	416.0	904.9
Cache: L1=128KB(D)	801.05	348.9	1 726.7	131.2	844	415.7	904.2
Cache: L1=256KB(D)	831.18	362.1	1 802.0	137.0	844	415.7	904.2
Cache: L1=512KB(D)	922.08	401.7	1 802.0	137.0	844	415.7	904.2
Cache: L1=4KB(D), L2=256KB(D)	1 380.19	601.2	2 781.9	211.5	844	415.7	1 994.4
Cache: L1=16KB(D), L2=256KB(D)	1 375.58	599.2	2 695.8	204.9	844	415.7	1 911.0
Cache: L1=4KB(A4)	149.82	65.3	815.5	62.0	3 134 118	148.6	378.9
Cache: L1=16KB(A4)	130.23	56.7	703.9	53.5	2 467 160	127.2	337.0
Cache: L1=64KB(A4)	77.63	33.8	365.1	27.7	75 374	69.7	224.3
Cache: L1=128KB(A4)	86.04	37.5	351.5	26.7	10 776	67.0	219.0
Cache: L1=256KB(A4)	140.58	61.2	435.2	33.1	780	66.7	218.4
Cache: L1=512KB(A4)	234.32	102.1	435.2	33.1	679	66.7	218.4
Cache: L1=4KB(A4), L2=256KB(A4)	213.02	92.8	739.7	56.2	778	52.3	633.9
Cache: L1=16KB(A4), L2=256KB(A4)	193.75	84.4	665.8	50.6	776	52.0	563.1
Cache: L1=4KB(A8)	94.35	41.1	544.1	41.4	3 173 735	80.5	245.0
Cache: L1=16KB(A8)	72.57	31.6	435.3	33.1	2 440 423	60.5	205.8
Cache: L1=64KB(A8)	10.91	4.8	92.2	7.0	32 037	1.0	89.0
Cache: L1=128KB(A8)	14.17	6.2	89.6	6.8	13 652	0.5	88.1
Cache: L1=256KB(A8)	20.50	8.9	175.4	13.3	5 076	0.3	87.6
Cache: L1=512KB(A8)	51.55	22.5	174.4	13.3	82	0.0	87.2
Cache: L1=4KB(A8), L2=256KB(A8)	48.95	21.3	353.6	26.9	1 644	0.1	345.2
Cache: L1=16KB(A8), L2=256KB(A8)	37.28	16.2	282.5	21.5	122	0.0	276.5
Cache: L1=4KB(A16)	109.03	47.5	542.2	41.2	3 171 416	80.1	244.1
Cache: L1=16KB(A16)	84.93	37.0	434.1	33.0	2 425 110	60.4	205.7
Cache: L1=32KB(A16)	49.30	21.5	258.5	19.6	1 160 469	29.9	145.8
Cache: L1=32KB(A16,W4)	34.85	15.2	323.6	24.6	2 050 099	13.8	113.0
Cache: L1=64KB(A16)	16.21	7.1	90.0	6.8	19 517	0.5	88.1

Table E.2.: Case study 1: Results for all platforms with LPDDR2 SDRAM. (Continued)

Platform	**Energy** mJ	**Energy** %	**Time** Cycles ($\times 10^6$)	**Time** %	**Page misses**	**DRAM accesses** ($\times 10^6$)	**Total accesses** ($\times 10^6$)
Cache: L1=64KB(A16,W4)	2.89	1.3	89.1	6.8	18 106	0.1	87.3
Cache: L1=128KB(A16)	17.97	7.8	87.2	6.6	48	0.0	87.2
Cache: L1=256KB(A16)	21.84	9.5	174.4	13.3	33	0.0	87.2
Cache: L1=256KB(A16,W8)	9.50	4.1	174.4	13.3	35	0.0	87.2
Cache: L1=256KB(A16,W4)	6.01	2.6	174.5	13.3	86	0.0	87.2
Cache: L1=512KB(A16)	30.09	13.1	174.4	13.3	33	0.0	87.2
Cache: L1=1MB(A16)	44.39	19.3	348.6	26.5	33	0.0	87.2
Cache: L1=2MB(A16)	92.38	40.2	348.6	26.5	33	0.0	87.2
Cache: L1=4MB(A16)	185.18	80.7	522.9	39.7	33	0.0	87.2
Cache: L1=4KB(A16), L2=256KB(A16)	70.73	30.8	352.2	26.8	33	0.0	344.1
Cache: L1=16KB(A16), L2=256KB(A16)	54.64	23.8	283.1	21.5	33	0.0	277.1
Cache: L1=32KB(A2), L2=256KB(A16)	94.59	41.2	582.4	44.3	33	0.0	567.2
Cache: L1=32KB(A16), L2=256KB(A16)	32.80	14.3	176.5	13.4	33	0.0	173.7
SRAM: 512B, 1KB, 32KB, 256KB	0.24	0.1	90.1	6.8	0	0.0	87.1
SRAM: 512B, 1KB, 32KB, 2x256KB	0.24	0.1	90.1	6.8	0	0.0	87.1
SRAM: 512B, 1KB, 32KB	7.35	3.2	117.6	8.9	12 861	3.0	87.1
SRAM: 512B, 1KB, 16KB, 32KB	4.65	2.0	106.0	8.1	2 761	1.9	87.1
SRAM: 512B, 1KB, 64KB	0.37	0.2	87.4	6.6	2 735	0.1	87.1
SRAM: 32KB	8.85	3.9	126.8	9.6	139 374	3.3	87.1
SRAM: 64KB	0.83	0.4	88.0	6.7	2 760	0.1	87.1
SRAM: 128KB	0.91	0.4	87.1	6.6	1	0.0	87.1
SRAM: 256KB	1.09	0.5	174.3	13.2	1	0.0	87.1
SRAM: 256KB, 256KB	1.09	0.5	174.3	13.2	1	0.0	87.1
SRAM: 512KB	2.16	0.9	174.3	13.2	1	0.0	87.1
SRAM: 512B, 4KB, 16KB, 32KB, 256KB	0.15	0.1	87.2	6.6	0	0.0	87.1
SRAM: 1MB	2.40	1.0	348.6	26.5	1	0.0	87.1
SRAM: 4x256KB	1.09	0.5	174.3	13.2	1	0.0	87.1
SRAM: 2x512KB	2.16	0.9	174.3	13.2	1	0.0	87.1
SRAM: 2MB	3.22	1.4	348.6	26.5	1	0.0	87.1
SRAM: 8x256KB	1.09	0.5	174.3	13.2	1	0.0	87.1

Table E.2.: Case study 1: Results for all platforms with LPDDR2 SDRAM. (Continued)

Platform	**Energy** mJ	**Energy** %	**Time** Cycles ($\times 10^6$)	**Time** %	**Page misses**	**DRAM accesses** ($\times 10^6$)	**Total accesses** ($\times 10^6$)
SRAM: 4x512KB	2.16	0.9	174.3	13.2	1	0.0	87.1
SRAM: 4MB	6.70	2.9	522.8	39.7	1	0.0	87.1
SRAM: 16x256KB	1.09	0.5	174.3	13.2	1	0.0	87.1
SRAM: 8x512KB	2.16	0.9	174.3	13.2	1	0.0	87.1
SRAM: 4x1MB	2.40	1.0	348.6	26.5	1	0.0	87.1
SRAM: 512B, 1KB, 32KB, 256KB, 8x512KB	0.24	0.1	90.1	6.8	0	0.0	87.1
SRAM: 512B, 1KB, 32KB, 8x512KB	0.27	0.1	90.1	6.8	0	0.0	87.1
SRAM: 8x32KB	0.43	0.2	87.1	6.6	1	0.0	87.1
SRAM: 4x64KB	0.63	0.3	87.1	6.6	1	0.0	87.1
SRAM: 256B, 1MB	1.17	0.5	212.8	16.2	0	0.0	87.1
SRAM: 512B, 1KB, 32KB, 256KB Cache: 4KB(A4,W16)	0.24	0.1	90.1	6.8	0	0.0	87.1
SRAM: 512B, 1KB, 32KB, 256KB Cache: 4KB(A4,W8)	0.24	0.1	90.1	6.8	0	0.0	87.1
SRAM: 512B, 1KB, 32KB, 256KB Cache: 4KB(A4,W4)	0.24	0.1	90.1	6.8	0	0.0	87.1
SRAM: 512B, 1KB, 32KB, 256KB Cache: 4KB(D,W16)	0.24	0.1	90.1	6.8	0	0.0	87.1
SRAM: 512B, 1KB, 32KB, 256KB Cache: 4KB(D,W8)	0.24	0.1	90.1	6.8	0	0.0	87.1
SRAM: 512B, 1KB, 32KB, 256KB Cache: 4KB(D,W4)	0.24	0.1	90.1	6.8	0	0.0	87.1
SRAM: 512B, 1KB, 32KB, 256KB Cache: 16KB(A4)	0.24	0.1	90.1	6.8	0	0.0	87.1
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(A4)	0.24	0.1	90.1	6.8	0	0.0	87.1
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(A16,W16)	0.24	0.1	90.1	6.8	0	0.0	87.1
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(A16,W8)	0.24	0.1	90.1	6.8	0	0.0	87.1
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(A16,W4)	0.24	0.1	90.1	6.8	0	0.0	87.1
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(A16,W16,Random)	0.24	0.1	90.1	6.8	0	0.0	87.1
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(A16,W8,Random)	0.24	0.1	90.1	6.8	0	0.0	87.1
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(A16,W4,Random)	0.24	0.1	90.1	6.8	0	0.0	87.1
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(A4,W16,Random)	0.24	0.1	90.1	6.8	0	0.0	87.1
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(A4,W8,Random)	0.24	0.1	90.1	6.8	0	0.0	87.1
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(A4,W4,Random)	0.24	0.1	90.1	6.8	0	0.0	87.1
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(D,W16)	0.24	0.1	90.1	6.8	0	0.0	87.1
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(D,W8)	0.24	0.1	90.1	6.8	0	0.0	87.1
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(D,W4)	0.24	0.1	90.1	6.8	0	0.0	87.1

Table E.2.: Case study 1: Results for all platforms with LPDDR2 SDRAM. (Continued)

Platform	**Energy** mJ	**Energy** %	**Time** Cycles ($\times 10^6$)	**Time** %	**Page misses**	**DRAM accesses** ($\times 10^6$)	**Total accesses** ($\times 10^6$)
SRAM: 512B, 1KB, 2KB, 4KB, 8KB, 16KB, 32KB, 64KB, 128KB	0.15	0.1	87.2	6.6	1	0.0	87.1
SRAM: 512B, 2x4KB, 2x32KB, 64KB, 128KB	0.16	0.1	87.1	6.6	0	0.0	87.1
SRAM: 512B, 1KB, 2KB, 3x4KB, 7x16KB, 4x32KB	0.14	0.1	87.2	6.6	1	0.0	87.1
SRAM: 512B, 1KB, 2KB, 3x4KB, 7x16KB, 2x32KB, 64KB	0.14	0.1	87.2	6.6	1	0.0	87.1
SRAM: 512B, 1KB, 2KB, 3x4KB, 3x16KB, 2x32KB, 2x64KB	0.14	0.1	87.2	6.6	1	0.0	87.1
SRAM: 512B, 1KB, 2KB, 3x4KB, 3x16KB, 2x32KB, 128KB	0.14	0.1	87.2	6.6	1	0.0	87.1
SRAM: 512B, 2x4KB, 3x32KB, 64KB, 128KB	0.16	0.1	87.1	6.6	0	0.0	87.1
SRAM: LowerBound(Energy↔1KB)	0.10	0.0	87.1	6.6	0	0.0	87.1
SRAM: LowerBound(Energy↔4KB)	0.14	0.1	87.1	6.6	0	0.0	87.1
Cache: LowerBound(D,W4,Energy↔4KB)	0.50	0.2	87.3	6.6	86	0.0	87.2
Cache: LowerBound(D,W8,Energy↔4KB)	1.22	0.5	87.3	6.6	35	0.0	87.2
Cache: LowerBound(D,W16,Energy↔4KB)	13.49	5.9	87.2	6.6	33	0.0	87.2
Cache: LowerBound(A4,W4,Energy↔4KB)	0.73	0.3	87.3	6.6	86	0.0	87.2
Cache: LowerBound(A4,W8,Energy↔4KB)	1.56	0.7	87.3	6.6	35	0.0	87.2
Cache: LowerBound(A4,W16,Energy↔4KB)	4.71	2.1	87.2	6.6	33	0.0	87.2
Cache: LowerBound(A8,W4,Energy↔4KB)	0.70	0.3	87.3	6.6	86	0.0	87.2
Cache: LowerBound(A8,W8,Energy↔4KB)	2.36	1.0	87.3	6.6	35	0.0	87.2
Cache: LowerBound(A8,W16,Energy↔4KB)	7.43	3.2	87.2	6.6	33	0.0	87.2
Cache: LowerBound(A16,W4,Energy↔4KB)	1.23	0.5	87.3	6.6	86	0.0	87.2
Cache: LowerBound(A16,W8,Energy↔4KB)	3.08	1.3	87.3	6.6	35	0.0	87.2
Cache: LowerBound(A16,W16,Energy↔4KB)	15.27	6.7	87.2	6.6	33	0.0	87.2

Table E.3.: Case study 2: Results for each input on all the platforms with Mobile SDRAM.

Platform	Energy mJ	Time Cycles ($\times 10^6$)	Page misses	DRAM accesses ($\times 10^6$)	Total accesses ($\times 10^6$)
INPUT 1					
Only DRAM	57.33	309.6	2 339 280	13.7	13.7
Cache: L1=64KB(A4)	14.45	60.2	178 113	4.2	22.0
Cache: L1=256KB(A16)	14.61	64.8	136 847	3.4	20.4
Cache: L1=256KB(A16,W4)	12.53	78.2	352 640	3.1	19.3
Cache: L1=512KB(A16)	6.83	32.0	5 606	0.5	14.6
Cache: L1=4MB(A16)	31.97	83.9	328	0.2	14.1
Cache: L1=16KB(A16), L2=256KB(A16)	15.62	62.1	135 701	3.4	31.0
Cache: L1=32KB(A2), L2=256KB(A16)	14.37	62.1	135 016	3.4	31.1
SRAM: 512B, 1KB, 32KB, 256KB	2.87	23.0	2 553	1.0	13.7
SRAM: 64KB	7.65	36.7	41 936	2.7	13.7
SRAM: 256KB	3.54	34.5	3 428	1.1	13.7
SRAM: 4MB	3.07	82.4	5	0.0	13.7
SRAM: 8x512KB	0.34	27.5	0	0.0	13.7
SRAM: 512B, 1KB, 32KB, 256KB, 8x512KB	0.09	16.8	0	0.0	13.7
SRAM: LowerBound(Energy↔4KB)	0.02	13.7	0	0.0	13.7
Cache: LowerBound(D,W16,Energy↔4KB)	2.92	15.3	328	0.2	14.1
INPUT 2					
Only DRAM	296.82	1 274.3	6 002 936	97.1	97.1
Cache: L1=64KB(A4)	227.17	824.5	826 778	76.5	245.7
Cache: L1=256KB(A16)	241.42	881.5	630 537	73.0	238.8
Cache: L1=256KB(A16,W4)	200.57	851.7	1 239 279	71.2	222.2
Cache: L1=512KB(A16)	255.61	874.4	591 129	72.4	237.6
Cache: L1=4MB(A16)	536.47	1 199.1	419 274	66.2	225.5
Cache: L1=16KB(A16), L2=256KB(A16)	270.20	957.7	630 159	73.0	398.2
Cache: L1=32KB(A2), L2=256KB(A16)	258.60	958.1	630 269	73.0	398.5
SRAM: 512B, 1KB, 32KB, 256KB	163.25	572.6	90 676	65.9	96.9
SRAM: 64KB	174.56	606.5	252 919	70.0	97.1
SRAM: 256KB	165.01	601.2	91 114	66.2	97.0
SRAM: 4MB	91.83	648.9	38 429	31.4	97.0
SRAM: 8x512KB	82.03	386.4	38 426	31.4	96.9
SRAM: 512B, 1KB, 32KB, 256KB, 8x512KB	75.10	345.6	35 301	29.0	96.9
SRAM: LowerBound(Energy↔4KB)	0.16	97.1	0	0.0	97.1
Cache: LowerBound(D,W16,Energy↔4KB)	23.27	116.3	2 307	2.1	101.2
INPUT 3					
Only DRAM	913.76	4 860.2	36 384 132	224.4	224.4
Cache: L1=64KB(A4)	230.74	975.6	3 170 944	66.9	355.4
Cache: L1=256KB(A16)	266.86	1 174.9	3 080 782	64.4	350.5
Cache: L1=256KB(A16,W4)	232.51	1 430.0	7 386 474	57.0	327.6
Cache: L1=512KB(A16)	89.96	454.0	5 855	0.5	225.5
Cache: L1=4MB(A16)	510.14	1 347.7	197	0.1	224.7
Cache: L1=16KB(A16), L2=256KB(A16)	281.87	1 123.3	3 076 138	64.4	512.7
Cache: L1=32KB(A2), L2=256KB(A16)	263.85	1 133.2	3 075 287	64.4	522.5
SRAM: 512B, 1KB, 32KB, 256KB	47.51	385.9	143 781	15.6	224.5
SRAM: 64KB	124.93	598.6	622 160	44.8	224.4
SRAM: 256KB	59.24	576.4	160 502	17.9	224.4

Table E.3.: Case study 2: Results for each input on all the platforms with Mobile SDRAM. (Continued)

Platform	**Energy** mJ	**Time** Cycles ($\times 10^6$)	**Page misses**	**DRAM accesses** ($\times 10^6$)	**Total accesses** ($\times 10^6$)
SRAM: 4MB	50.00	1 346.4	1	0.0	224.4
SRAM: 8x512KB	5.56	448.8	0	0.0	224.4
SRAM: 512B, 1KB, 32KB, 256KB, 8x512KB	1.43	274.8	0	0.0	224.5
SRAM: LowerBound(Energy↔4KB)	0.37	224.4	0	0.0	224.4
Cache: LowerBound(D,W16,Energy↔4KB)	40.52	225.7	197	0.1	224.7
INPUT 4					
Only DRAM	3 596.22	15 081.6	66 175 076	1 203.3	1 203.3
Cache: L1=64KB(A4)	2 839.10	10 232.3	8 167 449	960.6	3 067.2
Cache: L1=256KB(A16)	3 071.89	11 138.9	6 674 977	934.7	3 016.3
Cache: L1=256KB(A16,W4)	2 576.55	10 790.7	14 092 520	922.7	2 823.3
Cache: L1=512KB(A16)	3 254.79	11 066.8	6 291 635	928.7	3 004.5
Cache: L1=4MB(A16)	7 154.07	15 859.4	6 157 145	926.4	2 999.9
Cache: L1=16KB(A16), L2=256KB(A16)	3 428.79	12 085.1	6 684 100	934.7	4 990.7
Cache: L1=32KB(A2), L2=256KB(A16)	3 288.03	12 104.6	6 689 900	935.1	5 007.5
SRAM: 512B, 1KB, 32KB, 256KB	2 140.72	7 440.7	1 348 495	864.1	1 203.2
SRAM: 64KB	2 247.82	7 745.5	2 576 392	904.6	1 203.3
SRAM: 256KB	2 158.41	7 752.5	1 356 521	867.5	1 203.2
SRAM: 4MB	1 336.59	8 247.0	687 027	476.7	1 203.0
SRAM: 8x512KB	1 228.32	5 342.4	687 314	476.8	1 202.9
SRAM: 512B, 1KB, 32KB, 256KB, 8x512KB	1 144.40	4 870.4	638 028	447.5	1 202.9
SRAM: LowerBound(Energy↔4KB)	1.98	1 203.3	0	0.0	1 203.3
Cache: LowerBound(D,W16,Energy↔4KB)	221.49	1 223.9	3 066	2.3	1 207.7
INPUT 5					
Only DRAM	3 109.30	13 264.7	62 241 947	1 021.1	1 021.1
Cache: L1=64KB(A4)	2 456.75	8 921.6	9 562 375	828.5	2 629.4
Cache: L1=256KB(A16)	2 615.11	9 549.7	7 837 209	792.7	2 559.1
Cache: L1=256KB(A16,W4)	2 142.38	9 071.3	13 297 959	761.7	2 358.5
Cache: L1=512KB(A16)	2 761.11	9 457.1	7 395 845	784.7	2 543.4
Cache: L1=4MB(A16)	5 846.96	13 013.0	4 742 128	739.5	2 455.0
Cache: L1=16KB(A16), L2=256KB(A16)	2 927.58	10 394.9	7 835 034	792.7	4 273.9
Cache: L1=32KB(A2), L2=256KB(A16)	2 804.72	10 400.7	7 842 708	793.0	4 277.3
SRAM: 512B, 1KB, 32KB, 256KB	1 753.50	6 144.9	1 444 122	706.6	1 020.4
SRAM: 64KB	1 867.54	6 528.7	4 116 179	745.3	1 021.1
SRAM: 256KB	1 771.38	6 439.7	1 509 168	710.2	1 020.8
SRAM: 4MB	997.35	6 869.0	562 998	343.3	1 020.6
SRAM: 8x512KB	896.17	4 158.1	553 720	343.3	1 020.0
SRAM: 512B, 1KB, 32KB, 256KB, 8x512KB	817.96	3 720.2	500 609	316.0	1 019.9
SRAM: LowerBound(Energy↔4KB)	1.68	1 021.1	0	0.0	1 021.1
Cache: LowerBound(D,W16,Energy↔4KB)	188.14	1 039.2	2 682	2.0	1 025.0
INPUT 6					
Only DRAM	307.76	1 288.5	5 566 822	103.4	103.4
Cache: L1=64KB(A4)	244.94	882.6	706 031	82.9	264.2
Cache: L1=256KB(A16)	264.18	957.4	561 136	80.4	259.4
Cache: L1=256KB(A16,W4)	222.12	931.6	1 258 707	79.4	242.9
Cache: L1=512KB(A16)	280.24	952.6	541 728	80.0	258.5
Cache: L1=4MB(A16)	611.42	1 356.1	521 091	78.9	256.4

Table E.3.: Case study 2: Results for each input on all the platforms with Mobile SDRAM. (Continued)

Platform	Energy mJ	Time Cycles ($\times 10^6$)	Page misses	DRAM accesses ($\times 10^6$)	Total accesses ($\times 10^6$)
Cache: L1=16KB(A16), L2=256KB(A16)	295.03	1 039.5	561 707	80.4	429.8
Cache: L1=32KB(A2), L2=256KB(A16)	282.87	1 040.9	562 957	80.5	430.8
SRAM: 512B, 1KB, 32KB, 256KB	183.54	637.7	99 688	74.1	103.4
SRAM: 64KB	193.10	663.1	163 472	77.9	103.4
SRAM: 256KB	185.11	664.3	100 972	74.5	103.3
SRAM: 4MB	108.20	701.5	45 280	38.0	103.4
SRAM: 8x512KB	98.50	440.4	45 258	38.1	103.3
SRAM: 512B, 1KB, 32KB, 256KB, 8x512KB	90.91	399.1	41 864	35.4	103.3
SRAM: LowerBound(Energy↔4KB)	0.17	103.4	0	0.0	103.4
Cache: LowerBound(D,W16,Energy↔4KB)	24.40	122.7	2 921	2.1	107.5
INPUT 7					
Only DRAM	305.25	1 286.5	5 684 486	101.9	101.9
Cache: L1=64KB(A4)	239.21	863.1	712 734	80.9	258.8
Cache: L1=256KB(A16)	257.35	934.5	564 926	78.2	253.5
Cache: L1=256KB(A16,W4)	216.54	912.4	1 286 903	77.2	237.5
Cache: L1=512KB(A16)	273.39	930.9	550 899	77.9	252.9
Cache: L1=4MB(A16)	597.32	1 327.2	529 537	76.7	250.6
Cache: L1=16KB(A16), L2=256KB(A16)	287.44	1 013.9	565 916	78.2	420.1
Cache: L1=32KB(A2), L2=256KB(A16)	275.48	1 015.2	567 032	78.2	421.1
SRAM: 512B, 1KB, 32KB, 256KB	178.04	620.5	104 485	71.9	101.9
SRAM: 64KB	188.18	648.7	193 901	75.8	101.9
SRAM: 256KB	179.77	648.4	110 007	72.2	101.9
SRAM: 4MB	100.01	684.7	40 624	34.5	101.8
SRAM: 8x512KB	89.99	415.6	40 702	34.6	101.8
SRAM: 512B, 1KB, 32KB, 256KB, 8x512KB	82.04	372.9	37 084	31.7	101.8
SRAM: LowerBound(Energy↔4KB)	0.17	101.9	0	0.0	101.9
Cache: LowerBound(D,W16,Energy↔4KB)	23.77	120.0	2 720	2.0	105.7
INPUT 8					
Only DRAM	13.13	67.7	469 067	3.5	3.5
Cache: L1=64KB(A4)	5.62	21.4	35 527	1.8	7.0
Cache: L1=256KB(A16)	6.07	23.8	31 807	1.7	6.8
Cache: L1=256KB(A16,W4)	5.16	25.6	76 421	1.6	6.3
Cache: L1=512KB(A16)	6.20	22.5	27 309	1.6	6.5
Cache: L1=4MB(A16)	10.44	25.7	737	0.5	4.5
Cache: L1=16KB(A16), L2=256KB(A16)	6.69	24.7	31 772	1.7	10.8
Cache: L1=32KB(A2), L2=256KB(A16)	6.32	24.6	31 706	1.7	10.8
SRAM: 512B, 1KB, 32KB, 256KB	2.93	12.2	2 372	1.1	3.5
SRAM: 64KB	3.73	14.3	5 100	1.5	3.5
SRAM: 256KB	3.07	14.3	2 490	1.2	3.5
SRAM: 4MB	0.77	20.7	1	0.0	3.5
SRAM: 8x512KB	0.09	6.9	0	0.0	3.5
SRAM: 512B, 1KB, 32KB, 256KB, 8x512KB	0.04	5.0	0	0.0	3.5
SRAM: LowerBound(Energy↔4KB)	0.01	3.5	0	0.0	3.5
Cache: LowerBound(D,W16,Energy↔4KB)	2.13	8.4	737	0.5	4.5
INPUT 9					
Only DRAM	6.62	37.6	296 715	1.5	1.5

Table E.3.: Case study 2: Results for each input on all the platforms with Mobile SDRAM. (Continued)

Platform	**Energy** mJ	**Time** Cycles ($\times 10^6$)	**Page misses**	**DRAM accesses** ($\times 10^6$)	**Total accesses** ($\times 10^6$)
Cache: L1=64KB(A4)	1.53	6.2	13 078	0.5	2.4
Cache: L1=256KB(A16)	1.60	6.9	9 556	0.4	2.3
Cache: L1=256KB(A16,W4)	1.32	7.9	26 084	0.4	2.2
Cache: L1=512KB(A16)	1.13	4.7	1 534	0.2	1.9
Cache: L1=4MB(A16)	4.15	10.5	224	0.1	1.8
Cache: L1=16KB(A16), L2=256KB(A16)	1.70	6.6	9 527	0.4	3.4
Cache: L1=32KB(A2), L2=256KB(A16)	1.56	6.6	9 447	0.4	3.4
SRAM: 512B, 1KB, 32KB, 256KB	0.43	2.8	241	0.2	1.5
SRAM: 64KB	0.81	3.8	854	0.3	1.5
SRAM: 256KB	0.50	4.1	274	0.2	1.5
SRAM: 4MB	0.34	9.1	1	0.0	1.5
SRAM: 8x512KB	0.04	3.0	0	0.0	1.5
SRAM: 512B, 1KB, 32KB, 256KB, 8x512KB	0.01	1.9	0	0.0	1.5
SRAM: LowerBound(Energy↔4KB)	0.00	1.5	0	0.0	1.5
Cache: LowerBound(D,W16,Energy↔4KB)	0.68	2.9	224	0.1	1.8
INPUT 10					
Only DRAM	22.61	123.9	947 516	5.3	5.3
Cache: L1=64KB(A4)	4.03	17.8	57 414	1.1	7.5
Cache: L1=256KB(A16)	1.74	11.1	87	0.1	5.4
Cache: L1=256KB(A16,W4)	0.78	11.2	308	0.1	5.4
Cache: L1=512KB(A16)	2.24	11.1	87	0.1	5.4
Cache: L1=4MB(A16)	12.31	32.3	87	0.1	5.4
Cache: L1=16KB(A16), L2=256KB(A16)	1.74	8.3	87	0.1	7.9
Cache: L1=32KB(A2), L2=256KB(A16)	1.44	8.9	87	0.1	8.4
SRAM: 512B, 1KB, 32KB, 256KB	0.03	6.0	0	0.0	5.3
SRAM: 64KB	1.69	10.1	6 208	0.6	5.3
SRAM: 256KB	0.07	10.6	0	0.0	5.3
SRAM: 4MB	0.41	31.9	0	0.0	5.3
SRAM: 8x512KB	0.13	10.6	0	0.0	5.3
SRAM: 512B, 1KB, 32KB, 256KB, 8x512KB	0.03	6.0	0	0.0	5.3
SRAM: LowerBound(Energy↔4KB)	0.01	5.3	0	0.0	5.3
Cache: LowerBound(D,W16,Energy↔4KB)	1.09	5.8	87	0.1	5.4
INPUT 11					
Only DRAM	615.90	3 184.1	22 896 715	156.7	156.7
Cache: L1=64KB(A4)	160.87	680.2	2 233 479	46.6	248.1
Cache: L1=256KB(A16)	123.49	587.6	1 046 115	25.0	205.5
Cache: L1=256KB(A16,W4)	96.64	675.0	2 558 052	22.5	197.4
Cache: L1=512KB(A16)	86.25	393.5	167 461	8.0	172.3
Cache: L1=4MB(A16)	356.92	942.5	455	0.2	157.2
Cache: L1=16KB(A16), L2=256KB(A16)	140.30	579.1	1 039 219	24.9	347.8
Cache: L1=32KB(A2), L2=256KB(A16)	122.92	566.4	1 036 135	24.8	335.9
SRAM: 512B, 1KB, 32KB, 256KB	27.81	270.4	44 008	8.9	156.7
SRAM: 64KB	76.89	409.5	1 110 946	25.1	156.7
SRAM: 256KB	32.72	374.6	54 486	9.5	156.7
SRAM: 4MB	34.93	940.5	6	0.0	156.7
SRAM: 8x512KB	3.88	313.5	0	0.0	156.7

Table E.3.: Case study 2: Results for each input on all the platforms with Mobile SDRAM. (Continued)

Platform	Energy mJ	Time Cycles ($\times 10^6$)	Page misses	DRAM accesses ($\times 10^6$)	Total accesses ($\times 10^6$)
SRAM: 512B, 1KB, 32KB, 256KB, 8x512KB	1.11	213.6	0	0.0	156.7
SRAM: LowerBound(Energy↔4KB)	0.26	156.7	0	0.0	156.7
Cache: LowerBound(D,W16,Energy↔4KB)	28.65	158.8	455	0.2	157.2
INPUT 12					
Only DRAM	38.65	209.4	1 583 660	9.2	9.2
Cache: L1=64KB(A4)	8.85	37.4	112 295	2.5	14.2
Cache: L1=256KB(A16)	6.71	32.3	46 079	1.3	11.8
Cache: L1=256KB(A16,W4)	5.17	37.1	124 690	1.2	11.4
Cache: L1=512KB(A16)	4.58	21.5	3 697	0.3	9.8
Cache: L1=4MB(A16)	21.74	56.9	367	0.2	9.6
Cache: L1=16KB(A16), L2=256KB(A16)	7.12	29.3	45 737	1.3	17.7
Cache: L1=32KB(A2), L2=256KB(A16)	6.40	29.7	45 511	1.3	18.1
SRAM: 512B, 1KB, 32KB, 256KB	1.45	14.0	1 068	0.5	9.2
SRAM: 64KB	4.46	22.3	17 036	1.6	9.2
SRAM: 256KB	1.84	21.8	1 212	0.5	9.2
SRAM: 4MB	2.06	55.4	1	0.0	9.2
SRAM: 8x512KB	0.23	18.5	0	0.0	9.2
SRAM: 512B, 1KB, 32KB, 256KB, 8x512KB	0.06	11.1	0	0.0	9.2
SRAM: LowerBound(Energy↔4KB)	0.02	9.2	0	0.0	9.2
Cache: LowerBound(D,W16,Energy↔4KB)	2.11	10.7	367	0.2	9.6
INPUT 13					
Only DRAM	80.31	427.5	3 212 335	19.8	19.8
Cache: L1=64KB(A4)	21.43	89.5	274 836	6.3	32.1
Cache: L1=256KB(A16)	24.63	106.9	265 075	6.1	31.7
Cache: L1=256KB(A16,W4)	21.72	131.1	680 943	5.5	29.6
Cache: L1=512KB(A16)	12.92	56.3	34 869	1.7	23.1
Cache: L1=4MB(A16)	45.75	120.3	244	0.2	20.1
Cache: L1=16KB(A16), L2=256KB(A16)	26.21	103.5	264 447	6.1	47.1
Cache: L1=32KB(A2), L2=256KB(A16)	24.39	103.4	264 127	6.1	47.0
SRAM: 512B, 1KB, 32KB, 256KB	5.85	39.1	12 094	2.0	19.8
SRAM: 64KB	12.40	56.7	48 011	4.5	19.8
SRAM: 256KB	6.90	55.4	14 148	2.2	19.8
SRAM: 4MB	4.41	118.8	1	0.0	19.8
SRAM: 8x512KB	0.49	39.6	0	0.0	19.8
SRAM: 512B, 1KB, 32KB, 256KB, 8x512KB	0.14	24.8	0	0.0	19.8
SRAM: LowerBound(Energy↔4KB)	0.03	19.8	0	0.0	19.8
Cache: LowerBound(D,W16,Energy↔4KB)	4.00	21.3	244	0.2	20.1
INPUT 14					
Only DRAM	118.23	628.7	4 711 663	29.2	29.2
Cache: L1=64KB(A4)	30.90	129.4	397 407	9.1	46.9
Cache: L1=256KB(A16)	35.89	156.1	385 743	8.8	46.4
Cache: L1=256KB(A16,W4)	31.63	192.0	1 003 433	7.9	43.4
Cache: L1=512KB(A16)	17.42	77.1	21 119	2.0	33.0
Cache: L1=4MB(A16)	67.07	176.7	286	0.2	29.5
Cache: L1=16KB(A16), L2=256KB(A16)	37.92	149.8	384 605	8.8	68.0
Cache: L1=32KB(A2), L2=256KB(A16)	35.40	150.3	383 400	8.8	68.6

Table E.3.: Case study 2: Results for each input on all the platforms with Mobile SDRAM. (Continued)

Platform	**Energy** mJ	**Time** Cycles ($\times 10^6$)	**Page misses**	**DRAM accesses** ($\times 10^6$)	**Total accesses** ($\times 10^6$)
SRAM: 512B, 1KB, 32KB, 256KB	8.20	56.2	12 521	2.8	29.2
SRAM: 64KB	17.83	81.6	52 548	6.6	29.2
SRAM: 256KB	9.73	80.3	14 542	3.1	29.2
SRAM: 4MB	6.51	175.2	1	0.0	29.2
SRAM: 8x512KB	0.72	58.4	0	0.0	29.2
SRAM: 512B, 1KB, 32KB, 256KB, 8x512KB	0.20	36.5	0	0.0	29.2
SRAM: LowerBound(Energy↔4KB)	0.05	29.2	0	0.0	29.2
Cache: LowerBound(D,W16,Energy↔4KB)	5.68	30.7	286	0.2	29.5

Table E.4.: Case study 2: Results for each input on all the platforms with LPDDR2 SDRAM.

Platform	Energy mJ	Time Cycles ($\times 10^6$)	Page misses	DRAM accesses ($\times 10^6$)	Total accesses ($\times 10^6$)
INPUT 1					
Only DRAM	39.11	263.3	1 645 994	13.7	13.7
Cache: L1=64KB(A4)	6.60	35.9	122 043	4.2	22.0
Cache: L1=256KB(A16)	8.29	44.7	86 469	3.4	20.4
Cache: L1=256KB(A16,W4)	8.11	66.2	209 468	3.1	19.3
Cache: L1=512KB(A16)	6.08	29.4	3 772	0.5	14.7
Cache: L1=4MB(A16)	31.95	83.0	177	0.2	14.1
Cache: L1=16KB(A16), L2=256KB(A16)	9.32	42.1	85 785	3.4	31.0
Cache: L1=32KB(A2), L2=256KB(A16)	8.10	42.1	85 613	3.4	31.1
SRAM: 512B, 1KB, 32KB, 256KB	0.94	16.7	1 914	1.0	13.7
SRAM: 64KB	3.79	27.3	37 450	2.7	13.7
SRAM: 256KB	1.40	27.4	2 722	1.1	13.7
SRAM: 4MB	3.34	82.3	1	0.0	13.7
SRAM: 8x512KB	0.34	27.5	0	0.0	13.7
SRAM: 512B, 1KB, 32KB, 256KB, 8x512KB	0.09	16.8	0	0.0	13.7
SRAM: LowerBound(Energy↔4KB)	0.02	13.7	0	0.0	13.7
Cache: LowerBound(D,W16,Energy↔4KB)	2.67	14.3	177	0.2	14.1
INPUT 2					
Only DRAM	127.88	763.9	4 072 778	97.1	97.1
Cache: L1=64KB(A4)	88.92	387.3	393 054	76.5	245.7
Cache: L1=256KB(A16)	110.15	466.3	291 875	73.0	238.8
Cache: L1=256KB(A16,W4)	141.45	793.3	429 281	71.2	222.2
Cache: L1=512KB(A16)	125.42	462.3	255 642	72.4	237.6
Cache: L1=4MB(A16)	419.08	823.7	153 699	66.2	225.5
Cache: L1=16KB(A16), L2=256KB(A16)	139.18	542.5	293 284	73.0	398.2
Cache: L1=32KB(A2), L2=256KB(A16)	127.54	542.8	294 661	73.0	398.5
SRAM: 512B, 1KB, 32KB, 256KB	35.87	168.2	58 516	65.9	96.9
SRAM: 64KB	42.65	195.1	209 926	70.0	97.1
SRAM: 256KB	36.59	193.2	58 791	66.2	97.0
SRAM: 4MB	32.96	458.2	23 126	31.4	97.0
SRAM: 8x512KB	22.21	195.5	23 126	31.4	96.9
SRAM: 512B, 1KB, 32KB, 256KB, 8x512KB	19.64	168.6	21 155	29.0	96.9
SRAM: LowerBound(Energy↔4KB)	0.16	97.1	0	0.0	97.1
Cache: LowerBound(D,W16,Energy↔4KB)	19.81	104.6	1 106	2.1	101.2
INPUT 3					
Only DRAM	605.24	4 015.7	24 340 417	224.4	224.4
Cache: L1=64KB(A4)	105.32	574.0	1 994 636	66.9	355.4
Cache: L1=256KB(A16)	146.83	786.9	1 950 237	64.4	350.5
Cache: L1=256KB(A16,W4)	148.08	1 182.5	4 376 993	57.0	327.5
Cache: L1=512KB(A16)	90.57	451.1	5 944	0.6	225.5
Cache: L1=4MB(A16)	514.50	1 347.0	91	0.1	224.7
Cache: L1=16KB(A16), L2=256KB(A16)	161.62	735.2	1 949 923	64.4	512.7
Cache: L1=32KB(A2), L2=256KB(A16)	143.69	745.1	1 948 507	64.4	522.5
SRAM: 512B, 1KB, 32KB, 256KB	18.48	291.8	134 393	15.6	224.5
SRAM: 64KB	62.59	445.9	578 920	44.8	224.4
SRAM: 256KB	26.03	467.0	150 929	17.9	224.4

Table E.4.: Case study 2: Results for each input on all the platforms with LPDDR2 SDRAM. (Continued)

Platform	Energy mJ	Time Cycles ($\times 10^6$)	Page misses	DRAM accesses ($\times 10^6$)	Total accesses ($\times 10^6$)
SRAM: 4MB	54.60	1 346.4	1	0.0	224.4
SRAM: 8x512KB	5.56	448.8	0	0.0	224.4
SRAM: 512B, 1KB, 32KB, 256KB, 8x512KB	1.43	274.8	0	0.0	224.5
SRAM: LowerBound(Energy↔4KB)	0.37	224.4	0	0.0	224.4
Cache: LowerBound(D,W16,Energy↔4KB)	41.04	224.9	91	0.1	224.7
INPUT 4					
Only DRAM	1 464.20	8 661.8	45 090 915	1 203.3	1 203.3
Cache: L1=64KB(A4)	1 107.06	4 750.7	3 236 826	960.6	3 067.2
Cache: L1=256KB(A16)	1 393.90	5 832.3	2 699 700	934.7	3 016.3
Cache: L1=256KB(A16,W4)	1 819.93	10 079.0	3 937 140	922.7	2 823.3
Cache: L1=512KB(A16)	1 587.78	5 792.7	2 328 604	928.7	3 004.5
Cache: L1=4MB(A16)	5 506.42	10 579.8	1 828 905	926.4	2 999.9
Cache: L1=16KB(A16), L2=256KB(A16)	1 754.05	6 778.6	2 717 223	934.7	4 990.6
Cache: L1=32KB(A2), L2=256KB(A16)	1 612.77	6 797.3	2 738 120	935.1	5 007.6
SRAM: 512B, 1KB, 32KB, 256KB	461.00	2 105.9	927 464	864.1	1 203.2
SRAM: 64KB	526.60	2 353.2	2 119 329	904.6	1 203.3
SRAM: 256KB	476.82	2 410.5	933 808	867.5	1 203.2
SRAM: 4MB	439.73	5 361.2	454 693	476.7	1 203.0
SRAM: 8x512KB	317.04	2 440.3	454 969	476.8	1 202.9
SRAM: 512B, 1KB, 32KB, 256KB, 8x512KB	289.81	2 150.9	419 920	447.5	1 202.9
SRAM: LowerBound(Energy↔4KB)	1.98	1 203.3	0	0.0	1 203.3
Cache: LowerBound(D,W16,Energy↔4KB)	221.55	1 211.3	1 121	2.3	1 207.7
INPUT 5					
Only DRAM	1 259.43	7 708.2	43 711 953	1 021.1	1 021.1
Cache: L1=64KB(A4)	962.42	4 219.5	5 507 977	828.5	2 629.4
Cache: L1=256KB(A16)	1 190.83	5 065.4	4 593 843	792.7	2 559.1
Cache: L1=256KB(A16,W4)	1 516.71	8 521.3	6 307 385	761.6	2 358.5
Cache: L1=512KB(A16)	1 351.40	5 015.1	4 163 438	784.7	2 543.4
Cache: L1=4MB(A16)	4 534.41	8 845.2	2 367 205	738.5	2 453.1
Cache: L1=16KB(A16), L2=256KB(A16)	1 505.69	5 907.6	4 601 833	792.6	4 273.9
Cache: L1=32KB(A2), L2=256KB(A16)	1 382.32	5 912.4	4 620 119	793.0	4 277.3
SRAM: 512B, 1KB, 32KB, 256KB	372.08	1 754.6	1 099 416	706.6	1 020.4
SRAM: 64KB	387.38	1 868.7	3 678 781	745.3	1 021.1
SRAM: 256KB	382.49	2 023.5	1 163 034	710.2	1 020.8
SRAM: 4MB	357.99	4 803.5	395 469	343.3	1 020.6
SRAM: 8x512KB	247.32	2 091.9	386 144	343.3	1 020.0
SRAM: 512B, 1KB, 32KB, 256KB, 8x512KB	220.80	1 819.8	346 342	316.0	1 019.9
SRAM: LowerBound(Energy↔4KB)	1.68	1 021.1	0	0.0	1 021.1
Cache: LowerBound(D,W16,Energy↔4KB)	188.07	1 028.1	985	2.0	1 025.0
INPUT 6					
Only DRAM	124.23	728.7	3 699 812	103.4	103.4
Cache: L1=64KB(A4)	95.27	407.0	225 094	82.9	264.2
Cache: L1=256KB(A16)	119.71	499.4	185 776	80.4	259.4
Cache: L1=256KB(A16,W4)	156.41	864.7	272 724	79.4	242.9
Cache: L1=512KB(A16)	136.50	496.5	158 959	80.0	258.5
Cache: L1=4MB(A16)	470.87	904.9	120 520	78.9	256.4

Table E.4.: Case study 2: Results for each input on all the platforms with LPDDR2 SDRAM. (Continued)

Platform	Energy mJ	Time Cycles ($\times 10^6$)	Page misses	DRAM accesses ($\times 10^6$)	Total accesses ($\times 10^6$)
Cache: L1=16KB(A16), L2=256KB(A16)	150.84	581.5	187 518	80.4	429.8
Cache: L1=32KB(A2), L2=256KB(A16)	138.60	582.7	189 111	80.5	430.8
SRAM: 512B, 1KB, 32KB, 256KB	39.88	181.9	63 535	74.1	103.4
SRAM: 64KB	45.24	200.2	123 758	77.9	103.4
SRAM: 256KB	41.39	208.4	64 659	74.5	103.3
SRAM: 4MB	36.72	471.3	26 728	38.0	103.4
SRAM: 8x512KB	26.04	209.8	26 698	38.1	103.3
SRAM: 512B, 1KB, 32KB, 256KB, 8x512KB	24.10	186.9	24 603	35.4	103.3
SRAM: LowerBound(Energy↔4KB)	0.17	103.4	0	0.0	103.4
Cache: LowerBound(D,W16,Energy↔4KB)	20.95	110.9	1 064	2.1	107.5
INPUT 7					
Only DRAM	118.60	714.5	3 799 330	101.9	101.9
Cache: L1=64KB(A4)	93.15	398.9	233 562	80.9	258.8
Cache: L1=256KB(A16)	116.84	488.8	189 819	78.2	253.5
Cache: L1=256KB(A16,W4)	152.33	845.5	302 621	77.2	237.5
Cache: L1=512KB(A16)	133.41	486.5	165 938	77.9	252.9
Cache: L1=4MB(A16)	460.88	888.4	125 955	76.7	250.6
Cache: L1=16KB(A16), L2=256KB(A16)	147.17	568.2	190 915	78.2	420.1
Cache: L1=32KB(A2), L2=256KB(A16)	135.15	569.3	192 424	78.2	421.1
SRAM: 512B, 1KB, 32KB, 256KB	33.00	157.2	69 475	71.9	101.9
SRAM: 64KB	37.85	174.7	155 191	75.8	101.9
SRAM: 256KB	34.51	184.4	74 830	72.2	101.9
SRAM: 4MB	35.21	475.1	23 776	34.5	101.8
SRAM: 8x512KB	24.33	206.2	23 842	34.6	101.8
SRAM: 512B, 1KB, 32KB, 256KB, 8x512KB	21.83	181.1	21 605	31.7	101.8
SRAM: LowerBound(Energy↔4KB)	0.17	101.9	0	0.0	101.9
Cache: LowerBound(D,W16,Energy↔4KB)	20.55	108.9	994	2.0	105.7
INPUT 8					
Only DRAM	7.88	52.8	338 563	3.5	3.5
Cache: L1=64KB(A4)	2.29	10.7	13 381	1.8	7.0
Cache: L1=256KB(A16)	2.96	13.7	12 033	1.7	6.8
Cache: L1=256KB(A16,W4)	3.44	21.9	27 078	1.6	6.3
Cache: L1=512KB(A16)	3.30	13.1	8 927	1.6	6.5
Cache: L1=4MB(A16)	9.54	22.7	348	0.5	4.5
Cache: L1=16KB(A16), L2=256KB(A16)	3.58	14.6	12 157	1.7	10.8
Cache: L1=32KB(A2), L2=256KB(A16)	3.21	14.5	12 261	1.7	10.8
SRAM: 512B, 1KB, 32KB, 256KB	0.62	4.8	1 690	1.1	3.5
SRAM: 64KB	1.03	6.1	3 798	1.5	3.5
SRAM: 256KB	0.71	6.8	1 780	1.2	3.5
SRAM: 4MB	0.84	20.7	1	0.0	3.5
SRAM: 8x512KB	0.09	6.9	0	0.0	3.5
SRAM: 512B, 1KB, 32KB, 256KB, 8x512KB	0.04	5.0	0	0.0	3.5
SRAM: LowerBound(Energy↔4KB)	0.01	3.5	0	0.0	3.5
Cache: LowerBound(D,W16,Energy↔4KB)	1.17	5.4	348	0.5	4.5
INPUT 9					
Only DRAM	4.69	32.7	234 919	1.5	1.5

Table E.4.: Case study 2: Results for each input on all the platforms with LPDDR2 SDRAM. (Continued)

Platform	**Energy** mJ	**Time** Cycles ($\times 10^6$)	**Page misses**	**DRAM accesses** ($\times 10^6$)	**Total accesses** ($\times 10^6$)
Cache: L1=64KB(A4)	0.69	3.5	6 192	0.5	2.4
Cache: L1=256KB(A16)	0.90	4.6	3 555	0.4	2.3
Cache: L1=256KB(A16,W4)	0.86	6.6	9 123	0.4	2.2
Cache: L1=512KB(A16)	0.82	3.7	623	0.2	1.9
Cache: L1=4MB(A16)	3.92	9.6	104	0.1	1.8
Cache: L1=16KB(A16), L2=256KB(A16)	1.00	4.3	3 571	0.4	3.4
Cache: L1=32KB(A2), L2=256KB(A16)	0.86	4.3	3 740	0.4	3.4
SRAM: 512B, 1KB, 32KB, 256KB	0.13	1.9	150	0.2	1.5
SRAM: 64KB	0.30	2.3	594	0.3	1.5
SRAM: 256KB	0.17	3.0	176	0.2	1.5
SRAM: 4MB	0.37	9.1	1	0.0	1.5
SRAM: 8x512KB	0.04	3.0	0	0.0	1.5
SRAM: 512B, 1KB, 32KB, 256KB, 8x512KB	0.01	1.9	0	0.0	1.5
SRAM: LowerBound(Energy↔4KB)	0.00	1.5	0	0.0	1.5
Cache: LowerBound(D,W16,Energy↔4KB)	0.42	2.0	104	0.1	1.8
INPUT 10					
Only DRAM	15.99	107.8	678 328	5.3	5.3
Cache: L1=64KB(A4)	2.07	12.4	56 313	1.1	7.5
Cache: L1=256KB(A16)	1.68	10.8	55	0.1	5.4
Cache: L1=256KB(A16,W4)	0.75	11.1	116	0.1	5.4
Cache: L1=512KB(A16)	2.19	10.8	55	0.1	5.4
Cache: L1=4MB(A16)	12.32	32.1	55	0.1	5.4
Cache: L1=16KB(A16), L2=256KB(A16)	1.67	8.0	55	0.1	7.9
Cache: L1=32KB(A2), L2=256KB(A16)	1.37	8.6	55	0.1	8.4
SRAM: 512B, 1KB, 32KB, 256KB	0.03	6.0	0	0.0	5.3
SRAM: 64KB	1.20	9.6	5 138	0.6	5.3
SRAM: 256KB	0.07	10.6	0	0.0	5.3
SRAM: 4MB	0.41	31.9	0	0.0	5.3
SRAM: 8x512KB	0.13	10.6	0	0.0	5.3
SRAM: 512B, 1KB, 32KB, 256KB, 8x512KB	0.03	6.0	0	0.0	5.3
SRAM: LowerBound(Energy↔4KB)	0.01	5.3	0	0.0	5.3
Cache: LowerBound(D,W16,Energy↔4KB)	1.02	5.5	55	0.1	5.4
INPUT 11					
Only DRAM	403.66	2 638.4	16 167 711	156.7	156.7
Cache: L1=64KB(A4)	75.60	427.9	1 882 114	46.6	248.1
Cache: L1=256KB(A16)	78.12	443.1	753 322	24.9	205.5
Cache: L1=256KB(A16,W4)	66.79	599.2	1 729 674	22.5	197.4
Cache: L1=512KB(A16)	72.72	347.9	115 960	8.0	172.4
Cache: L1=4MB(A16)	359.74	941.3	283	0.2	157.2
Cache: L1=16KB(A16), L2=256KB(A16)	95.04	434.8	747 830	24.9	347.7
Cache: L1=32KB(A2), L2=256KB(A16)	77.71	422.4	745 618	24.8	335.9
SRAM: 512B, 1KB, 32KB, 256KB	10.46	212.9	38 569	8.9	156.7
SRAM: 64KB	41.24	325.1	1 048 531	25.1	156.7
SRAM: 256KB	14.65	313.8	48 733	9.5	156.7
SRAM: 4MB	38.14	940.4	4	0.0	156.7
SRAM: 8x512KB	3.88	313.5	0	0.0	156.7

Table E.4.: Case study 2: Results for each input on all the platforms with LPDDR2 SDRAM. (Continued)

Platform	Energy mJ	Time Cycles ($\times 10^6$)	Page misses	DRAM accesses ($\times 10^6$)	Total accesses ($\times 10^6$)
SRAM: 512B, 1KB, 32KB, 256KB, 8x512KB	1.11	213.6	0	0.0	156.7
SRAM: LowerBound(Energy↔4KB)	0.26	156.7	0	0.0	156.7
Cache: LowerBound(D,W16,Energy↔4KB)	28.79	157.5	283	0.2	157.2
INPUT 12					
Only DRAM	26.63	179.0	1 113 375	9.2	9.2
Cache: L1=64KB(A4)	4.17	23.4	87 095	2.5	14.2
Cache: L1=256KB(A16)	4.36	24.7	26 136	1.3	11.8
Cache: L1=256KB(A16,W4)	3.57	32.7	68 627	1.2	11.4
Cache: L1=512KB(A16)	4.10	19.8	2 425	0.3	9.9
Cache: L1=4MB(A16)	21.63	56.0	175	0.2	9.6
Cache: L1=16KB(A16), L2=256KB(A16)	4.77	21.7	25 715	1.3	17.7
Cache: L1=32KB(A2), L2=256KB(A16)	4.05	22.0	25 977	1.3	18.1
SRAM: 512B, 1KB, 32KB, 256KB	0.62	11.4	744	0.5	9.2
SRAM: 64KB	2.46	18.0	14 592	1.6	9.2
SRAM: 256KB	0.92	18.8	848	0.5	9.2
SRAM: 4MB	2.24	55.4	1	0.0	9.2
SRAM: 8x512KB	0.23	18.5	0	0.0	9.2
SRAM: 512B, 1KB, 32KB, 256KB, 8x512KB	0.06	11.1	0	0.0	9.2
SRAM: LowerBound(Energy↔4KB)	0.02	9.2	0	0.0	9.2
Cache: LowerBound(D,W16,Energy↔4KB)	1.85	9.8	175	0.2	9.6
INPUT 13					
Only DRAM	52.07	345.8	2 107 817	19.8	19.8
Cache: L1=64KB(A4)	9.57	50.7	150 255	6.3	32.1
Cache: L1=256KB(A16)	13.29	69.7	150 808	6.1	31.7
Cache: L1=256KB(A16,W4)	13.66	105.6	348 211	5.5	29.6
Cache: L1=512KB(A16)	9.97	46.4	15 106	1.7	23.1
Cache: L1=4MB(A16)	45.86	119.4	129	0.2	20.1
Cache: L1=16KB(A16), L2=256KB(A16)	14.86	66.3	150 402	6.1	47.1
Cache: L1=32KB(A2), L2=256KB(A16)	13.05	66.3	151 014	6.1	47.0
SRAM: 512B, 1KB, 32KB, 256KB	2.00	26.7	10 753	2.0	19.8
SRAM: 64KB	5.50	38.3	41 879	4.5	19.8
SRAM: 256KB	2.68	41.7	12 676	2.2	19.8
SRAM: 4MB	4.82	118.8	1	0.0	19.8
SRAM: 8x512KB	0.49	39.6	0	0.0	19.8
SRAM: 512B, 1KB, 32KB, 256KB, 8x512KB	0.14	24.8	0	0.0	19.8
SRAM: LowerBound(Energy↔4KB)	0.03	19.8	0	0.0	19.8
Cache: LowerBound(D,W16,Energy↔4KB)	3.77	20.4	129	0.2	20.1
INPUT 14					
Only DRAM	76.49	507.3	3 062 101	29.2	29.2
Cache: L1=64KB(A4)	13.82	73.3	207 230	9.1	46.9
Cache: L1=256KB(A16)	19.39	101.8	208 913	8.8	46.4
Cache: L1=256KB(A16,W4)	19.79	153.9	497 333	7.9	43.4
Cache: L1=512KB(A16)	14.06	65.9	10 129	2.0	33.0
Cache: L1=4MB(A16)	67.38	175.8	122	0.2	29.5
Cache: L1=16KB(A16), L2=256KB(A16)	21.42	95.5	207 372	8.8	68.0
Cache: L1=32KB(A2), L2=256KB(A16)	18.93	96.1	207 205	8.8	68.6

Table E.4.: Case study 2: Results for each input on all the platforms with LPDDR2 SDRAM. (Continued)

Platform	**Energy** mJ	**Time** Cycles ($\times 10^6$)	**Page misses**	**DRAM accesses** ($\times 10^6$)	**Total accesses** ($\times 10^6$)
SRAM: 512B, 1KB, 32KB, 256KB	2.87	39.0	11 173	2.8	29.2
SRAM: 64KB	8.02	55.8	46 058	6.6	29.2
SRAM: 256KB	3.86	61.2	13 012	3.1	29.2
SRAM: 4MB	7.10	175.2	1	0.0	29.2
SRAM: 8x512KB	0.72	58.4	0	0.0	29.2
SRAM: 512B, 1KB, 32KB, 256KB, 8x512KB	0.20	36.5	0	0.0	29.2
SRAM: LowerBound(Energy↔4KB)	0.05	29.2	0	0.0	29.2
Cache: LowerBound(D,W16,Energy↔4KB)	5.48	29.8	122	0.2	29.5

Table E.5.: Case study 3: Results for all platforms with Mobile SDRAM.

Platform	**Energy** mJ	**Energy** %	**Time** Cycles ($\times 10^6$)	**Time** %	**Page misses**	**DRAM accesses** ($\times 10^6$)	**Total accesses** ($\times 10^6$)
Only DRAM	59.11	100.0	379.1	100.0	3 461 858	12.7	12.7
Cache: L1=4KB(D)	410.26	694.1	1 425.6	376.0	3 801 420	145.1	296.2
Cache: L1=16KB(D)	367.01	620.9	1 243.6	328.0	3 356 339	126.0	259.0
Cache: L1=64KB(D)	340.32	575.8	1 042.0	274.9	2 728 252	105.8	219.7
Cache: L1=128KB(D)	353.56	598.2	944.3	249.1	2 391 125	96.2	201.0
Cache: L1=256KB(D)	330.53	559.2	876.8	231.3	2 089 781	87.9	184.8
Cache: L1=512KB(D)	318.03	538.0	794.7	209.6	1 793 201	79.9	169.2
Cache: L1=4KB(D), L2=256KB(D)	496.80	840.5	1 194.4	315.1	2 089 781	87.9	472.2
Cache: L1=16KB(D), L2=256KB(D)	484.55	819.8	1 156.0	304.9	2 089 781	87.9	434.9
Cache: L1=4KB(A4)	299.90	507.4	1 108.6	292.4	3 509 109	108.7	225.1
Cache: L1=16KB(A4)	220.73	373.4	822.3	216.9	2 752 180	79.2	167.4
Cache: L1=64KB(A4)	153.48	259.7	560.8	147.9	1 863 654	53.6	117.4
Cache: L1=128KB(A4)	125.96	213.1	446.5	117.8	1 466 055	42.5	95.7
Cache: L1=256KB(A4)	111.86	189.2	352.7	93.0	1 099 533	32.0	75.2
Cache: L1=512KB(A4)	102.34	173.1	255.4	67.4	766 835	22.5	56.6
Cache: L1=4KB(A4), L2=256KB(A4)	179.92	304.4	578.0	152.5	1 098 850	32.0	291.4
Cache: L1=16KB(A4), L2=256KB(A4)	161.01	272.4	516.7	136.3	1 099 520	32.0	231.8
Cache: L1=4KB(A8)	297.35	503.1	1 086.7	286.6	3 456 986	106.4	220.6
Cache: L1=16KB(A8)	218.29	369.3	807.0	212.9	2 697 160	77.7	164.5
Cache: L1=64KB(A8)	149.72	253.3	548.5	144.7	1 818 448	52.4	115.2
Cache: L1=128KB(A8)	120.77	204.3	434.8	114.7	1 419 009	41.4	93.6
Cache: L1=256KB(A8)	94.63	160.1	341.8	90.2	1 054 717	31.0	73.2
Cache: L1=512KB(A8)	80.14	135.6	248.2	65.5	736 719	21.8	55.3
Cache: L1=4KB(A8), L2=256KB(A8)	134.70	227.9	563.3	148.6	1 053 949	31.0	285.9
Cache: L1=16KB(A8), L2=256KB(A8)	123.44	208.8	503.1	132.7	1 051 906	31.0	227.8
Cache: L1=4KB(A16)	306.62	518.7	1 081.5	285.3	3 440 434	105.9	219.5
Cache: L1=16KB(A16)	224.27	379.4	800.7	211.2	2 667 833	77.2	163.4
Cache: L1=64KB(A16)	152.40	257.8	542.5	143.1	1 792 710	51.9	114.1
Cache: L1=128KB(A16)	122.41	207.1	430.9	113.6	1 402 439	41.0	92.9
Cache: L1=256KB(A16)	94.40	159.7	338.0	89.2	1 037 600	30.7	72.6

Table E.5.: Case study 3: Results for all platforms with Mobile SDRAM. (Continued)

Platform	**Energy** mJ	**Energy** %	**Time** Cycles ($\times 10^6$)	**Time** %	**Page misses**	**DRAM accesses** ($\times 10^6$)	**Total accesses** ($\times 10^6$)
Cache: L1=256KB(A16,W8)	48.56	82.2	215.9	57.0	1 084 670	14.9	41.2
Cache: L1=256KB(A16,W4)	31.13	52.7	172.9	45.6	1 236 687	7.9	27.4
Cache: L1=512KB(A16)	70.77	119.7	245.3	64.7	723 749	21.6	54.8
Cache: L1=1MB(A16)	52.31	88.5	192.8	50.8	459 263	13.9	39.7
Cache: L1=2MB(A16)	47.24	79.9	139.7	36.9	289 289	8.6	29.5
Cache: L1=4MB(A16)	39.88	67.5	100.2	26.4	70 012	2.2	17.1
Cache: L1=4KB(A16), L2=256KB(A16)	146.80	248.4	557.2	147.0	1 036 998	30.6	283.0
Cache: L1=16KB(A16), L2=256KB(A16)	132.05	223.4	498.2	131.4	1 035 023	30.6	226.1
Cache: L1=32KB(A2), L2=256KB(A16)	128.57	217.5	510.9	134.7	1 034 494	30.6	238.4
Cache: L1=32KB(A16), L2=256KB(A16)	125.59	212.5	470.3	124.1	1 033 567	30.6	199.3
SRAM: 512B, 1KB, 32KB, 256KB	15.86	26.8	98.6	26.0	660 519	4.3	12.7
SRAM: 512B, 1KB, 32KB, 2x256KB	11.89	20.1	73.6	19.4	407 489	3.4	12.7
SRAM: 128KB	24.46	41.4	159.7	42.1	1 305 038	6.0	12.7
SRAM: 256KB	16.81	28.4	106.6	28.1	718 707	4.5	12.7
SRAM: 256KB, 256KB	12.35	20.9	78.1	20.6	431 129	3.5	12.7
SRAM: 512KB	12.46	21.1	78.1	20.6	431 129	3.5	12.7
SRAM: 512B, 4KB, 16KB, 32KB, 256KB	15.38	26.0	95.2	25.1	625 902	4.2	12.7
SRAM: 1MB	9.32	15.8	77.0	20.3	222 050	2.6	12.7
SRAM: 4x256KB	8.68	14.7	56.9	15.0	222 060	2.6	12.7
SRAM: 2x512KB	8.80	14.9	56.9	15.0	222 050	2.6	12.7
SRAM: 2MB	6.16	10.4	63.3	16.7	84 781	1.6	12.7
SRAM: 8x256KB	5.36	9.1	41.2	10.9	85 319	1.6	12.7
SRAM: 4x512KB	5.49	9.3	41.2	10.9	84 867	1.6	12.7
SRAM: 4MB	3.59	6.1	77.7	20.5	13 029	0.3	12.7
SRAM: 16x256KB	1.59	2.7	28.1	7.4	13 266	0.3	12.7
SRAM: 8x512KB	1.74	2.9	28.1	7.4	13 124	0.3	12.7
SRAM: 4x1MB	2.37	4.0	52.9	14.0	13 124	0.3	12.7
SRAM: 512B, 1KB, 32KB, 256KB, 8x512KB	1.05	1.8	24.2	6.4	4 652	0.1	12.7
SRAM: 512B, 1KB, 32KB, 8x512KB	1.57	2.7	25.7	6.8	12 180	0.3	12.7
SRAM: 8x32KB	16.53	28.0	98.9	26.1	716 316	4.5	12.7

Table E.5.: Case study 3: Results for all platforms with Mobile SDRAM. (Continued)

Platform	**Energy** mJ	**Energy** %	**Time** Cycles ($\times 10^6$)	**Time** %	**Page misses**	**DRAM accesses** ($\times 10^6$)	**Total accesses** ($\times 10^6$)
SRAM: 4x64KB	16.55	28.0	99.0	26.1	716 729	4.5	12.7
SRAM: 256B, 1MB	9.16	15.5	72.3	19.1	222 508	2.6	12.7
SRAM: 512B, 1KB, 32KB, 256KB Cache: 4KB(A4,W16)	79.29	134.2	300.7	79.3	736 981	29.2	69.5
SRAM: 512B, 1KB, 32KB, 256KB Cache: 4KB(A4,W8)	46.27	78.3	202.0	53.3	857 677	15.9	42.8
SRAM: 512B, 1KB, 32KB, 256KB Cache: 4KB(A4,W4)	31.97	54.1	169.4	44.7	1 112 328	9.4	29.6
SRAM: 512B, 1KB, 32KB, 256KB Cache: 4KB(D,W16)	83.71	141.6	305.0	80.4	763 002	29.6	70.1
SRAM: 512B, 1KB, 32KB, 256KB Cache: 4KB(D,W8)	46.95	79.4	205.8	54.3	892 486	16.1	43.2
SRAM: 512B, 1KB, 32KB, 256KB Cache: 4KB(D,W4)	32.51	55.0	173.0	45.6	1 153 233	9.5	29.9
SRAM: 512B, 1KB, 32KB, 256KB Cache: 16KB(A4)	74.09	125.3	282.1	74.4	714 389	27.1	65.4
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(A4)	64.31	108.8	212.2	56.0	553 112	19.1	49.8
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(A16,W16)	56.78	96.1	210.1	55.4	541 324	18.9	49.5
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(A16,W8)	31.42	53.2	145.2	38.3	637 735	9.9	31.5
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(A16,W4)	22.10	37.4	128.0	33.8	842 465	5.8	23.4
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(A16,W16,Random)	57.68	97.6	212.1	56.0	519 691	19.3	50.2
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(A16,W8,Random)	30.98	52.4	141.1	37.2	557 840	9.9	31.5
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(A16,W4,Random)	20.24	34.2	113.8	30.0	643 288	5.6	22.9
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(A4,W16,Random)	64.96	109.9	213.0	56.2	526 155	19.3	50.3
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(A4,W8,Random)	32.12	54.3	142.5	37.6	573 069	10.0	31.7
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(A4,W4,Random)	20.52	34.7	116.2	30.6	673 627	5.6	23.0
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(D,W16)	76.50	129.4	220.2	58.1	584 267	19.8	51.2
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(D,W8)	35.79	60.6	154.2	40.7	695 350	10.5	32.8
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(D,W4)	23.88	40.4	136.5	36.0	920 827	6.2	24.2
SRAM: 512B, 1KB, 2KB, 4KB, 8KB, 16KB, 32KB, 64KB, 128KB	16.55	28.0	98.9	26.1	717 234	4.5	12.7
SRAM: 512B, 2x4KB, 2x32KB, 64KB, 128KB	16.32	27.6	97.5	25.7	703 220	4.5	12.7
SRAM: 512B, 1KB, 2KB, 3x4KB, 7x16KB, 4x32KB	16.56	28.0	99.3	26.2	720 558	4.5	12.7
SRAM: 512B, 1KB, 2KB, 3x4KB, 7x16KB, 2x32KB, 64KB	16.54	28.0	99.0	26.1	718 674	4.5	12.7
SRAM: 512B, 1KB, 2KB, 3x4KB, 3x16KB, 2x32KB, 2x64KB	16.54	28.0	98.9	26.1	717 234	4.5	12.7
SRAM: 512B, 1KB, 2KB, 3x4KB, 3x16KB, 2x32KB, 128KB	16.54	28.0	98.9	26.1	717 234	4.5	12.7
SRAM: 512B, 2x4KB, 3x32KB, 64KB, 128KB	15.51	26.2	91.9	24.2	649 729	4.3	12.7
SRAM: LowerBound(Energy↔1KB)	0.01	0.0	12.7	3.4	0	0.0	12.7

Table E.5.: Case study 3: Results for all platforms with Mobile SDRAM. (Continued)

Platform	**Energy** mJ	**Energy** %	**Time** Cycles ($\times 10^6$)	**Time** %	**Page misses**	**DRAM accesses** ($\times 10^6$)	**Total accesses** ($\times 10^6$)
SRAM: LowerBound(Energy↔4KB)	0.02	0.0	12.7	3.4	0	0.0	12.7
Cache: LowerBound(D,W4,Energy↔4KB)	3.31	5.6	22.8	6.0	3 051	1.2	15.0
Cache: LowerBound(D,W8,Energy↔4KB)	3.66	6.2	24.0	6.3	2 733	1.2	15.0
Cache: LowerBound(D,W16,Energy↔4KB)	5.47	9.3	23.1	6.1	2 097	1.2	15.0
Cache: LowerBound(A4,W4,Energy↔4KB)	3.34	5.7	22.8	6.0	3 051	1.2	15.0
Cache: LowerBound(A4,W8,Energy↔4KB)	3.72	6.3	24.0	6.3	2 733	1.2	15.0
Cache: LowerBound(A4,W16,Energy↔4KB)	4.07	6.9	23.1	6.1	2 097	1.2	15.0
Cache: LowerBound(A8,W4,Energy↔4KB)	3.34	5.7	22.8	6.0	3 051	1.2	15.0
Cache: LowerBound(A8,W8,Energy↔4KB)	3.85	6.5	24.0	6.3	2 733	1.2	15.0
Cache: LowerBound(A8,W16,Energy↔4KB)	4.50	7.6	23.1	6.1	2 097	1.2	15.0
Cache: LowerBound(A16,W4,Energy↔4KB)	3.43	5.8	22.8	6.0	3 051	1.2	15.0
Cache: LowerBound(A16,W8,Energy↔4KB)	3.96	6.7	24.0	6.3	2 733	1.2	15.0
Cache: LowerBound(A16,W16,Energy↔4KB)	5.76	9.7	23.1	6.1	2 097	1.2	15.0

Table E.6.: Case study 3: Results for all platforms with LPDDR2 SDRAM.

Platform	**Energy** mJ	**Energy** %	**Time** Cycles ($\times 10^6$)	**Time** %	**Page misses**	**DRAM accesses** ($\times 10^6$)	**Total accesses** ($\times 10^6$)
Only DRAM	44.63	100.0	324.6	100.0	2 946 215	12.7	12.7
Cache: L1=4KB(D)	158.74	355.7	718.8	221.4	3 245 129	145.1	296.2
Cache: L1=16KB(D)	148.68	333.1	634.7	195.5	2 981 439	126.0	259.0
Cache: L1=64KB(D)	157.11	352.0	533.4	164.3	2 454 551	105.8	219.7
Cache: L1=128KB(D)	187.05	419.1	483.2	148.9	2 158 956	96.2	201.0
Cache: L1=256KB(D)	177.97	398.8	448.1	138.0	1 882 348	87.9	184.8
Cache: L1=512KB(D)	179.31	401.7	404.7	124.7	1 595 920	79.9	169.2
Cache: L1=4KB(D), L2=256KB(D)	339.93	761.6	722.2	222.5	1 882 348	87.9	472.2
Cache: L1=16KB(D), L2=256KB(D)	327.54	733.9	683.8	210.7	1 882 348	87.9	434.9
Cache: L1=4KB(A4)	109.56	245.5	566.3	174.5	3 027 162	108.7	225.1
Cache: L1=16KB(A4)	82.07	183.9	428.2	131.9	2 494 090	79.2	167.4
Cache: L1=64KB(A4)	59.99	134.4	296.9	91.5	1 768 343	53.6	117.4
Cache: L1=128KB(A4)	51.99	116.5	238.0	73.3	1 409 325	42.5	95.7
Cache: L1=256KB(A4)	56.23	126.0	194.7	60.0	1 066 585	32.0	75.2
Cache: L1=512KB(A4)	63.28	141.8	144.4	44.5	747 341	22.5	56.6
Cache: L1=4KB(A4), L2=256KB(A4)	122.97	275.5	405.8	125.0	1 065 882	32.0	291.4
Cache: L1=16KB(A4), L2=256KB(A4)	103.81	232.6	344.4	106.1	1 066 972	32.0	231.8
Cache: L1=4KB(A8)	111.08	248.9	555.6	171.2	2 993 989	106.4	220.6
Cache: L1=16KB(A8)	82.28	184.4	420.7	129.6	2 453 131	77.7	164.5
Cache: L1=64KB(A8)	58.34	130.7	290.9	89.6	1 732 646	52.4	115.2
Cache: L1=128KB(A8)	48.77	109.3	232.1	71.5	1 369 892	41.4	93.6
Cache: L1=256KB(A8)	40.80	91.4	189.1	58.3	1 025 062	31.0	73.2
Cache: L1=512KB(A8)	42.27	94.7	140.6	43.3	718 517	21.8	55.3
Cache: L1=4KB(A8), L2=256KB(A8)	79.55	178.2	396.6	122.2	1 024 307	31.0	285.9
Cache: L1=16KB(A8), L2=256KB(A8)	68.14	152.7	336.5	103.7	1 022 444	31.0	227.8
Cache: L1=4KB(A16)	121.28	271.7	553.1	170.4	2 981 894	105.9	219.5
Cache: L1=16KB(A16)	89.30	200.1	417.5	128.6	2 430 123	77.2	163.4
Cache: L1=64KB(A16)	61.99	138.9	287.4	88.5	1 701 895	51.9	114.1
Cache: L1=128KB(A16)	51.06	114.4	230.1	70.9	1 354 138	41.0	92.9
Cache: L1=256KB(A16)	41.19	92.3	187.1	57.7	1 008 749	30.7	72.6

Table E.6.: Case study 3: Results for all platforms with LPDDR2 SDRAM. (Continued)

Platform	Energy mJ	Energy %	Time Cycles ($\times 10^6$)	Time %	Page misses	DRAM accesses ($\times 10^6$)	Total accesses ($\times 10^6$)
Cache: L1=256KB(A16,W8)	24.80	55.6	158.3	48.8	1 059 581	14.9	41.2
Cache: L1=256KB(A16,W4)	20.81	46.6	158.5	48.8	1 214 959	7.9	27.4
Cache: L1=512KB(A16)	33.38	74.8	139.1	42.9	705 762	21.6	54.8
Cache: L1=1MB(A16)	28.36	63.5	124.0	38.2	448 923	13.9	39.7
Cache: L1=2MB(A16)	32.39	72.6	97.1	29.9	284 591	8.6	29.5
Cache: L1=4MB(A16)	36.15	81.0	88.7	27.3	68 890	2.2	17.1
Cache: L1=4KB(A16), L2=256KB(A16)	92.26	206.7	392.4	120.9	1 008 296	30.6	283.0
Cache: L1=16KB(A16), L2=256KB(A16)	77.36	173.3	333.5	102.7	1 006 322	30.6	226.1
Cache: L1=32KB(A2), L2=256KB(A16)	73.91	165.6	346.1	106.6	1 005 548	30.6	238.4
Cache: L1=32KB(A16), L2=256KB(A16)	70.85	158.7	305.7	94.2	1 005 185	30.6	199.3
SRAM: 512B, 1KB, 32KB, 256KB	9.15	20.5	74.9	23.1	634 936	4.3	12.7
SRAM: 512B, 1KB, 32KB, 2x256KB	6.37	14.3	55.2	17.0	389 597	3.4	12.7
SRAM: 128KB	15.76	35.3	121.2	37.3	1 260 776	6.0	12.7
SRAM: 256KB	9.88	22.1	81.7	25.2	693 036	4.5	12.7
SRAM: 256KB, 256KB	6.72	15.1	59.3	18.3	413 048	3.5	12.7
SRAM: 512KB	6.84	15.3	59.3	18.3	413 048	3.5	12.7
SRAM: 512B, 4KB, 16KB, 32KB, 256KB	8.80	19.7	72.3	22.3	602 922	4.2	12.7
SRAM: 1MB	5.05	11.3	63.2	19.5	209 784	2.6	12.7
SRAM: 4x256KB	4.34	9.7	43.1	13.3	209 712	2.6	12.7
SRAM: 2x512KB	4.47	10.0	43.1	13.3	209 784	2.6	12.7
SRAM: 2MB	3.46	7.7	54.5	16.8	78 085	1.6	12.7
SRAM: 8x256KB	2.58	5.8	32.4	10.0	78 360	1.6	12.7
SRAM: 4x512KB	2.71	6.1	32.4	10.0	78 235	1.6	12.7
SRAM: 4MB	3.33	7.5	76.1	23.5	11 840	0.3	12.7
SRAM: 16x256KB	1.15	2.6	26.5	8.2	11 898	0.3	12.7
SRAM: 8x512KB	1.30	2.9	26.5	8.2	11 896	0.3	12.7
SRAM: 4x1MB	2.03	4.5	51.3	15.8	11 896	0.3	12.7
SRAM: 512B, 1KB, 32KB, 256KB, 8x512KB	0.93	2.1	23.6	7.3	4 044	0.1	12.7
SRAM: 512B, 1KB, 32KB, 8x512KB	1.17	2.6	24.2	7.5	10 910	0.3	12.7
SRAM: 8x32KB	9.57	21.5	73.7	22.7	690 671	4.5	12.7

Table E.6.: Case study 3: Results for all platforms with LPDDR2 SDRAM. (Continued)

Platform	**Energy** mJ	**Energy** %	**Time** Cycles ($\times 10^6$)	**Time** %	**Page misses**	**DRAM accesses** ($\times 10^6$)	**Total accesses** ($\times 10^6$)
SRAM: 4x64KB	9.60	21.5	73.8	22.7	692 177	4.5	12.7
SRAM: 256B, 1MB	4.88	10.9	58.5	18.0	209 854	2.6	12.7
SRAM: 512B, 1KB, 32KB, 256KB Cache: 4KB(A4,W16)	29.36	65.8	156.8	48.3	667 103	29.2	69.5
SRAM: 512B, 1KB, 32KB, 256KB Cache: 4KB(A4,W8)	21.75	48.7	138.7	42.7	724 532	15.9	42.8
SRAM: 512B, 1KB, 32KB, 256KB Cache: 4KB(A4,W4)	20.16	45.2	143.9	44.3	846 668	9.4	29.6
SRAM: 512B, 1KB, 32KB, 256KB Cache: 4KB(D,W16)	33.15	74.3	159.0	49.0	687 371	29.6	70.1
SRAM: 512B, 1KB, 32KB, 256KB Cache: 4KB(D,W8)	22.02	49.3	141.1	43.5	751 072	16.1	43.2
SRAM: 512B, 1KB, 32KB, 256KB Cache: 4KB(D,W4)	20.47	45.9	146.5	45.1	877 403	9.5	29.9
SRAM: 512B, 1KB, 32KB, 256KB Cache: 16KB(A4)	27.91	62.5	150.8	46.5	695 196	27.1	65.4
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(A4)	31.63	70.9	119.0	36.6	540 830	19.1	49.8
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(A16,W16)	24.42	54.7	117.8	36.3	529 178	18.9	49.5
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(A16,W8)	15.99	35.8	107.6	33.2	627 264	9.9	31.5
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(A16,W4)	14.91	33.4	118.2	36.4	834 361	5.8	23.4
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(A16,W16,Random)	24.67	55.3	117.0	36.0	505 498	19.3	50.2
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(A16,W8,Random)	15.63	35.0	102.0	31.4	545 677	9.9	31.5
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(A16,W4,Random)	13.61	30.5	103.3	31.8	633 125	5.6	22.9
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(A4,W16,Random)	31.83	71.3	117.6	36.2	512 085	19.3	50.3
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(A4,W8,Random)	16.62	37.2	103.3	31.8	560 475	10.0	31.6
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(A4,W4,Random)	13.76	30.8	105.7	32.6	663 181	5.6	23.0
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(D,W16)	42.50	95.2	123.1	37.9	571 177	19.8	51.2
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(D,W8)	19.27	43.2	113.7	35.0	683 437	10.5	32.8
SRAM: 512B, 1KB, 32KB, 256KB Cache: 256KB(D,W4)	16.08	36.0	125.5	38.7	911 085	6.2	24.2
SRAM: 512B, 1KB, 2KB, 4KB, 8KB, 16KB, 32KB, 64KB, 128KB	9.60	21.5	73.8	22.7	692 097	4.5	12.7
SRAM: 512B, 2x4KB, 2x32KB, 64KB, 128KB	9.42	21.1	72.5	22.3	676 669	4.5	12.7
SRAM: 512B, 1KB, 2KB, 3x4KB, 7x16KB, 4x32KB	9.59	21.5	73.9	22.8	693 860	4.5	12.7
SRAM: 512B, 1KB, 2KB, 3x4KB, 7x16KB, 2x32KB, 64KB	9.57	21.4	73.8	22.7	691 800	4.5	12.7
SRAM: 512B, 1KB, 2KB, 3x4KB, 3x16KB, 2x32KB, 2x64KB	9.58	21.5	73.8	22.7	692 097	4.5	12.7
SRAM: 512B, 1KB, 2KB, 3x4KB, 3x16KB, 2x32KB, 128KB	9.59	21.5	73.8	22.7	692 097	4.5	12.7
SRAM: 512B, 2x4KB, 3x32KB, 64KB, 128KB	8.82	19.8	68.0	21.0	624 024	4.3	12.7
SRAM: LowerBound(Energy↔1KB)	0.01	0.0	12.7	3.9	0	0.0	12.7

Table E.6.: Case study 3: Results for all platforms with LPDDR2 SDRAM. (Continued)

Platform	**Energy** mJ	**Energy** %	**Time** Cycles ($\times 10^6$)	**Time** %	**Page misses**	**DRAM accesses** ($\times 10^6$)	**Total accesses** ($\times 10^6$)
SRAM: LowerBound(Energy↔4KB)	0.02	0.0	12.7	3.9	0	0.0	12.7
Cache: LowerBound(D,W4,Energy↔4KB)	2.45	5.5	22.4	6.9	1 528	1.2	15.0
Cache: LowerBound(D,W8,Energy↔4KB)	1.87	4.2	18.7	5.8	1 368	1.2	15.0
Cache: LowerBound(D,W16,Energy↔4KB)	3.49	7.8	16.9	5.2	1 048	1.2	15.0
Cache: LowerBound(A4,W4,Energy↔4KB)	2.49	5.6	22.4	6.9	1 528	1.2	15.0
Cache: LowerBound(A4,W8,Energy↔4KB)	1.93	4.3	18.7	5.8	1 368	1.2	15.0
Cache: LowerBound(A4,W16,Energy↔4KB)	2.08	4.7	16.9	5.2	1 048	1.2	15.0
Cache: LowerBound(A8,W4,Energy↔4KB)	2.49	5.6	22.4	6.9	1 528	1.2	15.0
Cache: LowerBound(A8,W8,Energy↔4KB)	2.06	4.6	18.7	5.8	1 368	1.2	15.0
Cache: LowerBound(A8,W16,Energy↔4KB)	2.52	5.6	16.9	5.2	1 048	1.2	15.0
Cache: LowerBound(A16,W4,Energy↔4KB)	2.57	5.8	22.4	6.9	1 528	1.2	15.0
Cache: LowerBound(A16,W8,Energy↔4KB)	2.17	4.9	18.7	5.8	1 368	1.2	15.0
Cache: LowerBound(A16,W16,Energy↔4KB)	3.77	8.5	16.9	5.2	1 048	1.2	15.0

Appendix F

Example run of DynAsT

HERE I present a small example of *DynAsT* execution over a simple application that shows how it is instrumented, how the different steps of the methodology are performed with *DynAsT* and the output of the simulator.

F.1. Example application

The sample application is just a conceptual experiment with no real purpose. It has three data classes, V_1, V_2 and V_3. The first two are small data types of 500 B and 448 B, respectively, whereas the third one represents a class of big objects with a size of 1 MB.

The main code of the application creates an instance of V_3 that is alive during all the execution. This object is accessed in two bursts, with accesses to the other objects in between. One instance of V_1 and one of V_2 are created and destroyed consecutively, so that both objects are never alive at the same time (to enable an example of grouping). Both small objects receive many more accesses and thus, will be chosen by *DynAsT* for placement sooner.

F.1.1. Source code and instrumentation

The declaration of the three data classes is the only part of the application affected by the exceptions-based instrumentation used in this example:

```
#include "logged_allocated.hpp" // Instrumentation

class V1 : public logged_allocated<1>
{
public:
  unsigned int data[125]; // 500 bytes
};

class V2 : public logged_allocated<2>
{
public:
  unsigned int data[112]; // 448 bytes
};
```

```
class V3 : public logged_allocated<3>
{
public:
      unsigned int data[262144]; // 1 MB
};
```

The rest of the source code is straightforward, with the only exception that the application's entry point cannot be `main()`, as this is defined by the profiling library itself:

```
int MainCode(int , char **)
{
 V3 * v3 = new V3;
 for (int ii = 0; ii < 256*1024; ++ ii)
  v3->data[ii] = ii;

 V1 * v1 = new V1;
 for (int jj = 0; jj < 10; ++ jj) // Many accesses
  for (int ii = 0; ii < 125; ++ ii)
   v1->data[ii] = ii;
 delete v1;

 V2 * v2 = new V2;
 for (int jj = 0; jj < 10; ++ jj) // Many accesses
  for (int ii = 0; ii < 112; ++ ii)
   v2->data[ii] = ii;
 delete v2;

 volatile unsigned int aux = 0; // volatile prevents optimization
 for (int ii = 0; ii < 256*1024; ++ ii)
  aux += v3->data[ii];

 delete v3;
 return 0;
}
```

F.1.2. Instrumentation output after execution

The application is compiled normally and executed, creating the log file with the application memory allocations and data accesses. After execution, the profiling library outputs some useful information, such as the maximum application footprint, the number of allocations and deallocations (which should arguably be the same), and the number of reads and writes performed by the application on dynamic (instrumented) data objects:

```
$>SimpleTest.exe
Zeroing heap...DONE!
Starting address of heap: 4D30000

Maximum memory footprint: 1049076
Processed 526658 exceptions...
NumMallocs: 3 - NumFrees: 3
NumReads: 262144 - NumWrites: 264514
```

The log file for this example has a size of 2 633 368 B.

F.2. Processing with DynAsT

Once the log file is available, the designer can start working with *DynAsT* in two different ways. The whole process, from analysis to simulation (or simply mapping), can be executed at once. However, the log files tend to be quite big for realistic applications (in the order of several gigabytes) and so it may be desirable to execute the analysis just once and save the results in a binary file for quick loading in later executions of the tool. This is particularly interesting because the analysis is always performed in the same way, independently of the design decisions that might be made in later phases.

Grouping may also be time consuming in some cases and thus, the same trick can be used with it. Of course, if the designer wants to change some of the grouping parameters, this step must be executed again. In this example, I show how the work with *DynAsT* can be split into several steps through intermediate files.

F.2.1. Analysis

The analysis phase is easily executed with the following command line that instructs *DynAsT* to dump the results of the analysis to the file "SimpleTest.analysis.bin:"

```
$>dynast --InputLogFile log.bin --DoAnalysis
         --AnalysisResults SimpleTest.analysis.txt
         --DumpAnalysisFile SimpleTest.analysis.bin
         --PrintAnalyzerStatsOnFPB
```

The dump file has a size of just 498 B. The following is an excerpt of the information written by the analysis step in the file "SimpleTest.analysis.txt:"

```
Header is OK!
The log file is right and has 526664 packets.
Num of VAR_READ packets: 262144
Num of VAR_WRITE packets: 264514
Num of MALLOC_END packets: 3
Num of FREE_END packets: 3
Num of active blocks remaining: 0 (should be 0)
Number of distinct IDs: 3
Maximum memory footprint: 1049076
Max simultaneously active blocks: 2
Final memory footprint: 0

-----------------
ID STATISTICS (order FPB > ID > Size)
-----------------
ID: 1 SZ: 500     Read: 0      Writes: 1250   Created: 1 Dest: 1
      MaxAct: 1 MaxFoot: 500     FPB: 2.50 Selfish: 0.99 SeqAcc: 1240
ID: 2 SZ: 448     Read: 0      Writes: 1120   Created: 1 Dest: 1
      MaxAct: 1 MaxFoot: 448     FPB: 2.50 Selfish: 0.99 SeqAcc: 1110
ID: 3 SZ: 1048576 Read: 262144 Writes: 262144 Created: 1 Dest: 1
      MaxAct: 1 MaxFoot: 1048576 FPB: 0.50 Selfish: 1.00 SeqAcc: 524286
-----------------
```

DynAsT counts access operations when calculating the FPB, thus 1250 accesses on an object of 500 B yields an FPB of 2.50. The "Selfishness" and "SeqAcc" entries reflect DDT properties that may be exploited in the future.

F.2.2. Grouping

The grouping phase is executed with the following command line that instructs $\mathcal{DynAsT}$ to dump the results to the file "SimpleTest.grouping.bin:"

```
$>dynast --LoadAnalysisFile SimpleTest.analysis.bin
         --DoGrouping --GroupingResults SimpleTest.grouping.txt
         --DumpGroupingFile SimpleTest.grouping.bin
```

The grouping dump file has a size of 766 B. The following is an excerpt of the information written by the grouping step in the file "SimpleTest.grouping.txt:"

```
-----------------
GROUP STATISTICS
-----------------
Group 1 Read: 0 Writes: 2370 MaxFoot: 500 FPB: 4.74
      ExpRatio: 0.63
 ID: 1 SZ: 500 Read: 0 Writes: 1250 Created: 1 Dest: 1
       MaxAct: 1 MaxFoot: 500 FPB: 2.50
 ID: 2 SZ: 448 Read: 0 Writes: 1120 Created: 1 Dest: 1
       MaxAct: 1 MaxFoot: 448 FPB: 2.50
Group 2 Read: 262144 Writes: 262144 MaxFoot: 1048576 FPB: 0.50
      ExpRatio: 1.00
 ID: 3 SZ: 1048576 Read: 262144 Writes: 262144 Created: 1 Dest: 1
       MaxAct: 1 MaxFoot: 1048576 FPB: 0.50
-----------------
```

Thus, $\mathcal{DynAsT}$ has correctly identified that instances of v_1 and v_2 have disjoint life times and can be grouped together, whereas instances of v_3 should be placed apart.

F.2.3. Mapping

In its current version, $\mathcal{DynAsT}$ converts automatically groups into pools and proceeds with the mapping step, which is the first platform-dependent step in the methodology. For this example, I have chosen three simple platforms that later allow showing the simulator capabilities. The first two platforms contain a small SRAM of 512 B and 128 MB of Mobile SDRAM or 256 MB of LPDDR2 SDRAM, respectively. The third platform has a 4 KB direct-mapped cache with 128 MB of Mobile SDRAM. $\mathcal{DynAsT}$ is invoked with the following command lines for each of the cases, generating three mapping files of 258 B:

```
$>dynast --LoadGroupingFile SimpleTest.grouping.bin
         --DoMapping --MappingResults SimpleTest.mappingSRAM_SDR.txt
         --DumpMappingFile SimpleTest.mappingSRAM_SDR.bin
         --PlatformDescriptionFile Plat_SRAM_512_LPSDRAM_128MB.txt

$>dynast --LoadGroupingFile SimpleTest.grouping.bin
         --DoMapping --MappingResults SimpleTest.mappingSRAM_DDR2.txt
         --DumpMappingFile SimpleTest.mappingSRAM_DDR2.bin
         --PlatformDescriptionFile Plat_SRAM_512_LPDDR2S2_256MB.txt

$>dynast --LoadGroupingFile SimpleTest.grouping.bin
         --DoMapping --MappingResults SimpleTest.mappingCache_D.txt
         --DumpMappingFile SimpleTest.mappingCache_D.bin
         --PlatformDescriptionFile Plat_Cache(d)4KB_LPSDRAM_128MB.txt
```

For the SRAM-based platforms the mapping report is the same. In both cases, one pool is placed on the SRAM and the other is placed on the DRAM:

```
-----------------
MAPPING STATISTICS
-----------------
There are 2 pools.

Pool 1 -  Reads: 0 Writes: 2370 MaxFoot: 500 FPB: 4.7
The pool has 2 IDs:
(1, 500) (2, 448)
The pool is split over 1 memory blocks:
 Fragment 1: BlockID=0, Size=500, Address=0
------

Pool 2 -  Reads: 262144 Writes: 262144 MaxFoot: 1048576 FPB: 0.5
The pool has 1 IDs:
(3, 1048576)
The pool is split over 1 memory blocks:
 Fragment 1: BlockID=1, Size=1048576, Address=0
------
```

The mapping report for the cache-based platform is slightly different. Here, both pools are placed on the main DRAM because the cache is transparent. Thus, the second pool is placed at offset 500 of the DRAM (BlockID = 0):

```
-----------------
MAPPING STATISTICS
-----------------
There are 2 pools.

Pool 1 -  Reads: 0 Writes: 2370 MaxFoot: 500 FPB: 4.7
The pool has 2 IDs:
(1, 500) (2, 448)
The pool is split over 1 memory blocks:
 Fragment 1: BlockID=0, Size=500, Address=0
------

Pool 2 -  Reads: 262144 Writes: 262144 MaxFoot: 1048576 FPB: 0.5
The pool has 1 IDs:
(3, 1048576)
The pool is split over 1 memory blocks:
 Fragment 1: BlockID=0, Size=1048576, Address=500
------
```

F.2.4. Simulation

The simulation is executed for each of the platforms with the following command lines:

```
$>dynast --InputLogFile log.bin
         --LoadMappingFile SimpleTest.mappingSRAM_SDR.bin
         --DoSimulation --SimulationResults SimpleTest.simSRAM_SDR.txt
         --PlatformDescriptionFile Plat_SRAM_512_LPSDRAM128MB.txt

$>dynast --InputLogFile log.bin
         --LoadMappingFile SimpleTest.mappingSRAM_DDR2.bin
```

```
         --DoSimulation --SimulationResults SimpleTest.simSRAM_DDR2.txt
         --PlatformDescriptionFile Plat_SRAM_512_LPDDR2S2_256MB.txt

$>dynast --InputLogFile log.bin
         --LoadMappingFile SimpleTest.mappingCache_D.bin
         --DoSimulation --SimulationResults SimpleTest.simCache_D.txt
         --PlatformDescriptionFile Plat_Cache(d)4KB_LPSDRAM128MB.txt
```

F.2.4.1. Simulation for platform with SRAM and Mobile SDRAM

The following text shows the output of *DynAsT*'s simulator for the first platform. In essence, the output is divided in four sections. First, the simulator gives the estimation of energy consumption and latency for accesses to the instances of each of the application DDTs. This information may be interesting to identify and evaluate specific algorithmic optimizations. Second, the simulator presents information for every memory module in the platform. In this case, "MemBlock 0" corresponds to the SRAM and "MemBlock 1" to the Mobile SDRAM. Next, the information on the interconnections is shown; however, these examples use a simple organization with a single bus and assign a null energy and latency cost for every transaction. Finally, the simulator prints the calculated total values of energy consumption and cycles in the memory subsystem.

```
-------------
ID STATISTICS
-------------
ID: 1 SZ: 500 Energy: 0.81 nJ Cycles: 1250
ID: 2 SZ: 448 Energy: 0.73 nJ Cycles: 1120
ID: 3 SZ: 1048576 Energy: 1284460.76 nJ Cycles: 4227032
-------------

-----------------------
MEMORY BLOCK STATISTICS
-----------------------
MemBlock: 0
 Energy: 1.54 nJ Cycles: 2370 Reads: 0 Writes: 2370
MemBlock: 1
 Energy: 1284518.41 nJ Cycles: 4227032 Reads: 262144 Writes: 262144
 Page misses: 512
 Empty cycles: 2370 Largest empty slot: 2370 cycles NumEmptySlots: 1
 Avg. empty slot: 2370.00 cycles SlotsLongerThan1000 cycles: 1
 EnergyReads: 640933.28 nJ EnergyWrites: 641284.63 nJ
 EnergyActivations: 2242.86 nJ Background energy: 57.65 nJ
 DelayReads: 2101248 DelayWrites: 2101232 DelayActivations: 24552
Total number of memory accesses to all modules: 526658
-----------------------

---------------------------------
MEMORY INTERCONNECTION STATISTICS
---------------------------------
Interconnection: 1 Energy: 0.00 nJ Cycles: 0 Transfers: 526658
---------------------------------

TOTAL ENERGY CONSUMPTION: 1284519.95 nJ
TOTAL NUMBER OF CYCLES: 4229402
```

For the DRAMs, the simulator dumps extra information. For example, it identifies periods of inactivity and reports their number, average length and longest one – in this example, the simulator identifies a single inactivity period between the two bursts of accesses to v_3 while the application accesses v_1 and v_2, which are placed on the SRAM. The simulator also reports the energy consumed during read, write and row-change operations, and the background energy consumed by the DRAM being active but not accessed.

F.2.4.2. Simulation for platform with SRAM and LPDDR2 SDRAM

The output for the second platform is very similar to the previous one, as they both have an SRAM and a DRAM. However, the simulator generates additional information specific to LPDDR2 memories. In particular, the number of accesses that correspond to the second word of each double-data-rate transfer – which are performed whether the processor accesses a single word or two consecutive ones – and the number of transitions between the logical states are reported:

```
-----------------------
MEMORY BLOCK STATISTICS
-----------------------
MemBlock: 0
 Energy: 1.54 nJ Cycles: 2370 Reads: 0 Writes: 2370
MemBlock: 1
 Energy: 288391.46 nJ Cycles: 1071036 Reads: 262144 Writes: 262144
 Page misses: 256
 Empty cycles: 2370 Largest empty slot: 2370 cycles NumEmptySlots: 1
 Avg. empty slot: 2370.00 cycles SlotsLongerThan1000 cycles: 1
 EnergyReads: 142074.70 nJ EnergyWrites: 145333.96 nJ
 EnergyActivations: 917.04 nJ Background energy: 65.76 nJ
 DelayReads: 528376 DelayWrites: 530396 DelayActivations: 12264
 LPDDR2-S2 --> HiddenDDR: 262144 Idle2Read: 0 Idle2Write: 1
  Read2Read: 262143 Read2Write: 0 Write2Read: 1 Write2Write: 262143
  r2r_changerow: 127 r2r_samerow_hiddenddr: 131072
  r2r_samerow_seamlessburst: 130944 r2r_samerow_fulldelay: 0
  w2r_changerow: 1 w2r_samerow: 0
  r2w_changerow: 0 r2w_samerow: 0
  w2w_changerow: 127 w2w_samerow_hiddenddr: 131072
  w2w_samerow_seamlessburst: 130944 w2w_samerow_fulldelay: 0
  Extra tRAS delays: 0 Starting delays: 5588
Total number of memory accesses to all modules: 526658
-----------------------

TOTAL ENERGY CONSUMPTION: 288393.00 nJ
TOTAL NUMBER OF CYCLES: 1073406
```

F.2.4.3. Simulation for platform with cache and Mobile SDRAM

Finally, for the platform with a cache memory the simulator reports independently the energy consumption and latency for the caches and their associated DRAMs. The information about cache hits and misses is included so, although not used in this text, it could be employed in future analyses:

```
-----------------------
MEMORY BLOCK STATISTICS
-----------------------
MemBlock: 0
 Energy: 2239943.49 nJ Cycles: 8654168 Reads: 524432 Writes: 262288
 Page misses: 32786
 Empty cycles: 559444 Largest empty slot: 2263 cycles
 NumEmptySlots: 32777
 Avg. empty slot: 17.07 cycles SlotsLongerThan1000 cycles: 1
 EnergyReads: 1442014.23 nJ EnergyWrites: 640561.17 nJ
 EnergyActivations: 143759.99 nJ Background energy: 13608.10 nJ
 DelayReads: 4982160 DelayWrites: 2098304 DelayActivations: 1573704
CacheBlock: 0
 Energy: 205447.81 nJ Cycles: 559444 Reads: 508048 Writes: 788946
 Hits: 493881 Misses: 32777
Total number of memory accesses to all modules: 2083714
-----------------------

TOTAL ENERGY CONSUMPTION: 2445391.29 nJ
TOTAL NUMBER OF CYCLES: 9213612
```

An interesting observation derived from this excerpt is that the use of a cache memory changes completely how the DRAM is used. Now, instead of a single period of inactivity of 2370 (CPU) cycles, the DRAM sees 32 777 slots with an average length of 17.07 (CPU) cycles (i.e., they are probably too short to allow the DRAM to enter an energy-saving state). Future optimizations might exploit the better predictability of the solutions generated by the techniques presented in this text to transition the DRAM modules into energy-saving states more effectively.

Bibliography

[ABC+06] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical report, Electrical Engineering and Computer Sciences, University of California at Berkeley, 2006. 56 pp.

[ABP+07] David Atienza, Christos Baloukas, Lazaros Papadopoulos, Christophe Poucet, Stylianos Mamagkakis, Jose I. Hidalgo, Francky Catthoor, Dimitrios Soudris, and Juan Lanchares. Optimization of dynamic data structures in multimedia embedded systems using evolutionary computation. In *Proceedings of the International Workshop on Software & Compilers for Embedded Systems (SCOPES)*, pages 31–40, Nice, France, 2007. ACM Press.

[ABS01] Oren Avissar, Rajeev Barua, and Dave Stewart. Heterogeneous memory management for embedded systems. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 34–43, Atlanta, Georgia, USA, 2001. ACM Press. ISBN 1-58113-399-5.

[Acc10] Accellera Systems Initiative. IEEE standard for IP-XACT, standard structure for packaging, integrating, and reusing IP within tool flows, 2010. http://www.accellera.org/activities/committees/ip-xact.

[ACFM14] Jordi Arjona Aroca, Angelos Chatzipapas, Antonio Fernández Anta, and Vincenzo Mancuso. A measurement-based analysis of the energy consumption of data center servers. In *Proceedings of the International Conference on Future Energy Systems (e-Energy)*, pages 63–74, Cambridge, United Kingdom, 2014. ACM Press. ISBN 978-1-4503-2819-7.

[AGP+08] David Atienza, Pablo García del Valle, Giacomo Paci, Francesco Poletti, Luca Benini, Giovanni de Micheli, José M. Mendías, and Román Hermida. HW-SW emulation framework for temperature-aware design in MPSoCs. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 12(3):26:1–26:26, May 2008.

[AMC+04a] David Atienza, Stylianos Mamagkakis, Francky Catthoor, José Manuel Mendías, and Dimitrios Soudris. Dynamic memory management design methodology for reduced memory footprint in multimedia and wireless network applications. In *Proceedings of Design, Automation and Test in Europe (DATE)*, volume 1, pages 532–537, 2004.

[AMC+04b] David Atienza, Stylianos Mamagkakis, Francky Catthoor, José Manuel Mendías, and Dimitrios Soudris. Modular construction and power modelling of dynamic memory managers for embedded systems. In *Proceedings of International Workshop on Power And Timing Modeling, Optimization and Simulation (PATMOS)*, volume 3254 of *Lecture Notes in Computer Science (LNCS)*, pages 510–520. Springer-Verlag Berlin Heidelberg, 2004. ISBN 978-3-540-23095-3.

[AMM+06a] David Atienza, José M. Mendías, Stylianos Mamagkakis, Dimitrios Soudris, and Francky Catthoor. Systematic dynamic memory management design methodology for reduced memory footprint. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 11(2):465–489, apr 2006.

[AMM+06b] David Atienza, José M. Mendías, Stylianos Mamagkakis, Dimitrios Soudris, and Francky Catthoor. Systematic dynamic memory management design methodology for reduced memory footprint. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 11(2):465–489, April 2006.

[AMP00] Nawaaz Ahmed, Nikolay Mateev, and Keshav Pingali. Tiling imperfectly-nested loop nests. In *Proceedings of the IEEE/ACM International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, Dallas, Texas, United States, 2000. IEEE Computer Society Press. ISBN 0-7803-9802-5.

[AMP+04] David Atienza, Stylianos Mamagkakis, Miguel Peón-Quirós, Francky Catthoor, José Manuel Mendías, and Dimitrios Soudris. Power aware tuning of dynamic memory management for embedded real-time multimedia applications. In *XIX Conference on Design of Circuits and Integrated Systems (DCIS)*, pages 375–380, November 2004.

[AMP+15] David Atienza Alonso, Stylianos Mamagkakis, Christophe Poucet, Miguel Peón-Quirós, Alexandros Bartzas, Francky Catthoor, and Dimitrios Soudris. *Dynamic Memory Management for Embedded Systems*. Springer International Publishing Switzerland, 2015. ISBN 978-3-319-10571-0. 243 pp.

[APM+04] Mohammed Javed Absar, Francesco Poletti, Paul Marchal, Francky Catthoor, and Luca Benini. Fast and power-efficient dynamic data-layout with DMA-capable memories. In *Proceedings of the International Workshop on Power-Aware Real-Time Computing (PACS)*, 2004.

[ARM11] ARM. *Cortex-A15 Technical Reference Manual Rev. r2p0*. ARM, September 2011.

[Ati05] David Atienza. *Metodología multinivel de refinamiento del subsistema de memoria dinámica para los sistemas empotrados multimedia de altas prestaciones*. PhD thesis, Universidad Complutense de Madrid, Departamento de Arquitectura de Computadores y Automática, April 2005.

[AVL62] G. M. Adelson-Velskii and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259–1263, 1962.

[AXB+11] I. Anagnostopoulos, S. Xydis, A. Bartzas, Zhonghai Lu, D. Soudris, and A. Jantsch. Custom microcoded dynamic memory management for distributed on-chip memory organizations. *IEEE Embedded Systems Letters*, 3(2):66–69, June 2011.

[BB96] Gilles Brassard and T. Bratley. *Fundamentals of Algorithmics*, pages 227–230. Prentice Hall, 1st (Spanish) edition, 1996.

[BBB+11] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The Gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, August 2011.

[BBDT84] G. Bozman, W. Buco, T. P. Daly, and W. H. Tetzlaff. Analysis of free-storage algorithms—revisited. *IBM Systems Journal*, 23(1):44–64, 1984.

[BCH13] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. The datacenter as a computer: An introduction to the design of Warehouse-Scale machines, second edition. *Synthesis Lectures on Computer Architecture*, 8(3):1–154, 2013.

[BGN46] Arthur W. Burks, Herman H. Goldstine, and John von Neumann. Preliminary discussion of the logical design of an electronic computing instrument. pages 92–119, 1946.

[BH07] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, December 2007.

[BH09] Luiz André Barroso and Urs Hölzle. The datacenter as a computer: An introduction to the design of Warehouse-Scale machines. *Synthesis Lectures on Computer Architecture*, 4(1):1–108, 2009.

[BM00] Luca Benini and Giovanni de Micheli. System-level power optimization: Techniques and tools. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 5(2):115–192, 2000.

[BMBW00] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. *ACM SIGPLAN Notices*, 35(11):117–128, November 2000.

[BMP00] Luca Benini, Alberto Macii, and Massimo Poncino. A recursive algorithm for low-power memory partitioning. In *Proc. of ISLPD*, pages 78–83. ACM Press, 2000. ISBN 1-58113-190-9.

[BMP+06] Alexandros Bartzas, Stylianos Mamagkakis, Georgios Pouiklis, David Atienza, Francky Catthoor, Dimitrios Soudris, and Antonios Thanailakis. Dynamic data type refinement methodology for systematic performance-energy design exploration of network applications. In *Proceedings of Design, Automation and Test in Europe (DATE)*, pages 740–745. European Design and Automation Association, 2006. ISBN 3-9810801-0-6.

[Boh07] M. Bohr. A 30 year retrospective on Dennard's MOSFET scaling paper. *IEEE Solid-State Circuits Society Newsletter*, 12(1):11–13, 2007.

[BP11] Anirudh Badam and Vivek S. Pai. SSDAlloc: Hybrid SSD/RAM memory management made easy. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 211–224, Boston, MA, 2011. USENIX Association.

[BPM+08] Alexandros Bartzas, Miguel Peón-Quirós, Stylianos Mamagkakis, Francky Catthoor, Dimitrios Soudris, and José Manuel Mendías. Enabling run-time memory data transfer optimizations at the system level with automated extraction of embedded software metadata information. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 434–439, Seoul, Korea, 2008. IEEE Computer Society Press. ISBN 978-1-4244-1922-7.

[BPM+09] Alexandros Bartzas, Miguel Peón-Quirós, Stylianos Mamagkakis, Francky Catthoor, Dimitrios Soudris, and José Manuel Mendías. Direct memory access usage optimization in network applications for reduced memory latency and energy consumption. *Journal of Embedded Computing (JEC)*, 3:241–254, August 2009.

[BPP+10] Alexandros Bartzas, Miguel Peón-Quirós, Christophe Poucet, Christos Baloukas, Stylianos Mamagkakis, Francky Catthoor, Dimitrios Soudris, and José Manuel Mendías. Software metadata: Systematic characterization of the memory behaviour of dynamic applications. *Journal of Systems and Software (JSS)*, Volume 83 Issue 6, June 2010(83):1051–1075, 2010. Software Architecture and Mobility.

[BRA+09] Christos Baloukas, José L. Risco-Martín, David Atienza, Christophe Poucet, Lazaros Papadopoulos, Stylianos Mamagkakis, Dimitrios Soudris, J. Ignacio Hidalgo, Francky Catthoor, and Juan Lanchares. Optimization methodology of dynamic data structures based on genetic algorithms for multimedia embedded systems. *Journal of Systems and Software (JSS)*, 82(4):590–602, 2009. Special Issue: Selected papers from the 2008 IEEE Conference on Software Engineering Education and Training (CSEET'08).

[BS08] J. Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 443–446, September 2008.

[BSL+02] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. In *Proceedings of the International Symposium on Hardware/-Software Codesign (CODES)*, pages 73–78, Estes Park, Colorado, 2002. ACM Press. ISBN 1-58113-542-4.

[BZM01] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Composing high-performance memory allocators. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 114–124, Snowbird, Utah, United States, 2001. ACM Press. ISBN 1-58113-414-2.

[CCA+11] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 105–118, Newport Beach, California, USA, 2011. ACM Press. ISBN 978-1-4503-0266-1.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In

Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI), pages 209–224, San Diego, California, 2008. USENIX Association.

[CDK+02] F. Catthoor, K. Danckaert, C. Kulkarni, E. Brockmeyer, P. G. Kjeldsbert, T. Van Achteren, and T. Omnes. *Data access and storage management for embedded programmable processors*. Kluwer Academic Publishers, 2002.

[CDL99] Trishul M. Chilimbi, Bob Davidson, and James R. Larus. Cache-conscious structure definition. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 13–24, Atlanta, Georgia, United States, 1999. ACM Press. ISBN 1-58113-094-5.

[CH10] Aaron Carroll and Gernot Heiser. An analysis of power consumption in a smartphone. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, pages 21–21, Boston, MA, 2010. USENIX Association.

[CK00] Chandra Chekuri and Sanjeev Khanna. A PTAS for the multiple knapsack problem. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 213–222, 2000.

[CKR06] Reuven Cohen, Liran Katzir, and Danny Raz. An efficient approximation for the Generalized Assignment Problem. *Information Processing Letters*, 100(4):162–166, 2006.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press and McGraw-Hill Book Company, 2nd edition, 2001. ISBN 0-262-03293-7.

[CM95] Stephanie Coleman and Kathryn S. McKinley. Tile size selection using cache organization and data layout. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 279–290, La Jolla, California, USA, 1995. ACM Press. ISBN 0-89791-697-2.

[CMIC00] Clay Mathematics Institute (CMI). The Millennium Prize Problems, 2000. http://www.claymath.org/millennium-problems/millennium-prize-problems.

[Coo00] Stephen Cook. The P versus NP problem, 2000. http://www.claymath.org/sites/default/files/pvsnp.pdf.

[CS13] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Communications of the ACM (CACM)*, 56(2):82–90, February 2013.

[CWG+98] Francky Catthoor, Sven Wuytack, Eddy De Greef, Florin Balasa, Lode Nachtergaele, and Arnout Vandecappelle. *Custom Memory Management Methodology: Exploration of memory organisation for embedded multimedia system design*. Kluwer Academic Publishers, Boston, USA, 1998. ISBN 0-7923-8288-9.

[DAV+04] Edgard Daylight, David Atienza, Arnout Vandecappelle, Francky Catthoor, and José Manuel Mendías. Memory-access-aware data structure transformations for embedded software with dynamic data accesses. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, 12(3):269–280, 2004.

[DBD+06] Minas Dasygenis, Erik Brockmeyer, Bart Durinck, Francky Catthoor, Dimitrios Soudris, and Antonios Thanailakis. A combined DMA and application-specific prefetching approach for tackling the memory latency bottleneck. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, 14(3):279–291, 2006.

[DDS95] B. Davari, R. H. Dennard, and G. G. Shahidi. CMOS scaling for high performance and low power – The next ten years. *Proceedings of the IEEE*, 83(4):595–606, April 1995.

[DGRB74] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, and E Bassous. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[Dod06] James M. Dodd. Adaptive page management, July 2006. U.S. pat. 7,076,617 B2. Intel Corporation.

[Dre07] Ulrich Drepper. What every programmer should know about memory, 2007.

[DUB05] Ángel Domínguez, Sumesh Udayakumaran, and Rajeev Barua. Heap data allocation to scratch-pad memory in embedded systems. *Journal of Embedded Computing (JEC)*, 2005.

[EBS+11] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. *ACM SIGARCH Computer Architecture News*, 39(3):12, July 2011.

[EVB03] Lieven Eeckhout, Hans Vandierendonck, and Koen De Bosschere. Quantifying the impact of input data sets on program behavior and its applications. *Journal of Instruction-Level Parallelism*, 5:1–33, 2003.

[FAK+12] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 37–48, London, England, UK, 2012. ACM Press. ISBN 978-1-4503-0759-8.

[FGMS06] Lisa Fleischer, Michel X. Goemans, Vahab S. Mirrokni, and Maxim Sviridenko. Tight approximation algorithms for maximum general assignment problems. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 611–620, 2006. ISBN 0-89871-605-5.

[Fre60] Edward Fredkin. Trie memory. *Communications of the ACM (CACM)*, 3(9):490–499, September 1960.

[FW08] R. F. Freitas and W. W. Wilcke. Storage-class memory: The next storage system technology. *IBM Journal of Research and Development*, 52(4.5):439–447, July 2008.

[GAM+06] Pablo García del Valle, David Atienza, Iván Magán, Javier García Flores, Esther Andrés Pérez, José Manuel Mendías, Luca Benini, and Giovanni de Micheli. A complete multi-processor system-on-chip FPGA-based emulation framework. In *IFIP International Conference on Very Large Scale Integration*, pages 140–145, October 2006.

[GBC05] Stefan Valentin Gheorghita, Twan Basten, and Henk Corporaal. Intra-task scenario-aware voltage scheduling. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 177–184. ACM Press, 2005. ISBN 1-59593-149-X.

[GBD+05] Bert Geelen, Erik Brockmeyer, Bart Durinck, Gauthier Lafruit, and Rudy Lauwereins. Alleviating memory bottlenecks by software-controlled data transfers in a data-parallel wavelet transform on a multicore DSP. In *Proceedings of the IEEE BENELUX/DSP Valley Signal Processing Symposium (SPS-DARTS)*, pages 143–146, 2005.

[GCPT10] Rodrigo González-Alberquilla, Fernando Castro, Luis Piñuel, and Francisco Tirado. Stack filter: Reducing L1 data cache power consumption. *Journal of Systems Architecture (JSA)*, 56(12):685–695, 2010.

[Gen07] Universiteit Gent. Reconfigurable Embedded Systems for Use in scalable Multimedia Environments, 2007. http://www.elis.ugent.be/resume.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc, 1995. ISBN 0-201-63361-2.

[GKM82] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. *SIGPLAN Not. on compiler construction*, 17(6):120–126, 1982.

[GPH+09] Stefan Valentin Gheorghita, Martin Palkovic, Juan Hamers, Arnout Vandecappelle, Stylianos Mamagkakis, Twan Basten, Lieven Eeckhout, Henk Corporaal, Francky Catthoor, Frederik Vandeputte, and Koen De Bosschere. System-scenario-based design of dynamic embedded systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 14(1):1–45, 2009.

[GZ93] Dirk Grunwald and Benjamin Zorn. CustoMalloc: Efficient synthesized memory allocators. *Software: Practice and Experience*, 23:851–869, August 1993.

[HKA04] Tristan Henderson, David Kotz, and Ilya Abyzov. The changing usage of a mature campus-wide wireless network. In *Proceedings of the Annual International Conference on Mobile Computing and Networking (MobiCom)*, pages 187–201, Philadelphia, PA, USA, 2004. ACM Press. ISBN 1-58113-868-7.

[HLL+14] Ward Van Heddeghem, Sofie Lambert, Bart Lannoo, Didier Colle, Mario Pickavet, and Piet Demeester. Trends in worldwide ICT electricity consumption from 2007 to 2012. *Computer Communications*, 2014.

[HP] HP Labs. The Machine: A new kind of computer, Last fetched on July 2015. http://www.hpl.hp.com/research/systems-research/themachine.

[HP 08] CACTI 5.3, 2008. http://quid.hpl.hp.com:9081/cacti/.

[HP11] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Francisco, CA, USA, 5th edition, 2011.

[HS12] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 225 Wyman Street, Waltham, MA 02451, USA, 1st (revised) edition, 2012. ISBN 978-0-12-397337-5.

[IBS+10] François Ingelrest, Guillermo Barrenetxea, Gunnar Schaefer, Martin Vetterli, Olivier Couach, and Marc Parlange. SensorScope: Application-specific sensor network for environmental monitoring. *ACM Transactions on Sensor Networks (TOSN)*, 6(2):1–32, 2010.

[ISTI07] Information Society Technologies (IST). BEing on Time Saves energY, 2007. http://www.hitech-projects.com/euprojects/betsy.

[JED11a] JEDEC. *Failure mechanisms and models for semiconductor devices - JEP122G*. JEDEC Solid State Technology Association, October 2011.

[JED11b] JEDEC. *Low Power Double Data Rate 2 (LPDDR2) - JESD209-2E*. JEDEC Solid State Technology Association, April 2011.

[Jou90] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 364–373, Seattle, Washington, USA, 1990. ACM Press. ISBN 0-89791-366-3.

[JW94] N. P. Jouppi and S. J. E. Wilton. Tradeoffs in two-level on-chip caching. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 34–45, Chicago, Illinois, USA, 1994. IEEE Computer Society Press. ISBN 0-8186-5510-0.

[JW98] Mark S. Johnstone and Paul R. Wilson. The memory fragmentation problem: Solved? In *Proceedings of the International Symposium on Memory Management (ISMM)*, pages 26–36, Vancouver, British Columbia, Canada, 1998. ACM Press. ISBN 1-58113-114-3.

[KBSW11] Jonathan G. Koomey, Stephen Berard, Marla Sanchez, and Henry Wong. Implications of historical trends in the electrical efficiency of computing. *IEEE Annals of the History of Computing*, 33(3):46–54, 2011.

[KKC+04] Mahmut Kandemir, I. Kadayif, A. Choudhary, J. Ramanujam, and I. Kolcu. Compiler-directed scratchpad memory optimization for embedded multiprocessors. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, 12:281–287, 2004.

[KRI+01] Mahmut Kandemir, J. Ramanujam, J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. Dynamic management of scratch-pad memory space. In *Proceedings of the Design Automation Conference (DAC)*, pages 690–695, 2001. ISBN 1-58113-297-2.

[LA04] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 75–86. IEEE Computer Society Press, March 2004.

[LA05] Chris Lattner and Vikram Adve. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 129–142, Chicago, IL, USA, 2005. ACM Press. ISBN 1-59593-056-6.

[Lea96] Doug Lea. A memory allocator, 1996. http://g.oswego.edu/dl/html/malloc.html.

[LLL01] Amy W. Lim, Shih-Wei Liao, and Monica S. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP)*, pages 103–112, Snowbird, Utah, United States, 2001. ACM Press. ISBN 1-58113-346-4.

[LMK06] Wentong Li, S. Mohanty, and K. Kavi. A page-based hybrid (software-hardware) dynamic memory allocator. *IEEE Computer Architecture Letters (CAL)*, 5(2):13–13, 2006.

[Man04] Hugo De Man. Connecting e-dreams to deep-submicron realities. In *Proceedings of International Workshop on Power And Timing Modeling, Optimization and Simulation (PATMOS)*. Springer-Verlag Berlin Heidelberg, 2004.

[MAP+06] Stylianos Mamagkakis, David Atienza, Christophe Poucet, Francky Catthoor, and Dimitrios Soudris. Energy-efficient dynamic memory allocators at the middleware level of embedded systems. In *Proceedings of the ACM & IEEE International Conference on Embedded Software (EMSOFT)*, volume 1, pages 215–222, Seoul, Korea, 2006. ACM Press. ISBN 1-59593-542-8.

[MBP+07] Stylianos Mamagkakis, Alexandros Bartzas, Georgios Pouiklis, David Atienza, Francky Catthoor, Dimitrios Soudris, and Antonios Thanailakis. Systematic methodology for exploration of performance - energy trade-offs in network applications using dynamic data type refinement. *Journal of Systems Architecture (JSA)*, 53(7):417–436, 2007.

[MCB+04] Paul Marchal, Francky Catthoor, Davide Bruni, Luca Benini, José Ignacio Gómez, and Luis Piñuel. Integrated task scheduling and data assignment for SDRAMs in dynamic applications. *IEEE Design and Test of Computers*, 21(5):378–387, 2004.

[MDS08] Ross McIlroy, Peter Dickman, and Joe Sventek. Efficient dynamic heap allocation of scratch-pad memory. In *Proceedings of the International Symposium on Memory Management (ISMM)*, pages 31–40, Tucson, AZ, USA, 2008. ACM Press. ISBN 978-1-60558-134-7.

[MGP+03] Paul Marchal, José Ignacio Gómez, Luis Piñuel, Davide Bruni, Luca Benini, Francky Catthoor, and Henk Corporaal. SDRAM-energy-aware memory allocation for dynamic multi-media applications on multi-processor platforms. In *Proceedings of Design, Automation and Test in Europe (DATE)*, 2003.

[MH10] Trevor Mudge and Urs Hölzle. Challenges and opportunities for extremely energy-efficient processors. *IEEE Micro*, 30(4):20–24, 2010.

[Mic04] Microsoft Corporation. Low fragmentation heap in XP technologies, 2004. http://msdn.microsoft.com/library/default.asp?url=/library/ en-us/memory/base/low_fragmentation_heap.asp.

[MIC10] MICRON. *Mobile LPSDR SDRAM - MT48H32M32LF/LG Rev. D 1/11 EN*. Micron Technology, Inc, April 2010.

[MIC12] MICRON. *Mobile LPDDR2 SDRAM - MT42L64M32D1 Rev. N 3/12 EN*. Micron Technology, Inc, April 2012.

[Mic15] Microsoft Corporation. CRT debug heap details, Last fetched on April 2015. https://msdn.microsoft.com/en-us/library/974tc9t1.aspx.

[MMR+13] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 461–472, Houston, Texas, USA, 2013. ACM Press. ISBN 978-1-4503-1870-9.

[Moo65] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):4, 1965.

[Moo75] Gordon E. Moore. Progress in digital integrated electronics. In *International Electron Devices Meeting*, volume 21, pages 11–13, 1975.

[MPS71] Barry H. Margolin, Richard P. Parmelee, and Martin Schatzoff. Analysis of free-storage algorithms. *IBM Systems Journal*, 10(4):283–304, 1971.

[Net04] Nicholas Nethercote. *Dynamic Binary Analysis and Instrumentation or Building Tools is Easy*. PhD thesis, Trinity College, Cambridge, England, 2004.

[Obja] Object Management Group. Architecture-driven modernization (ADM): Knowledge discovery meta-model (KDM), Last fetched on November 2014. http:/adm.omg.org/.

[Objb] Object Management Group. Knowledge discovery meta-model (KDM), Last fetched on November 2014. http://www.omg.org/technology/kdm/.

[PAC06] Christophe Poucet, David Atienza, and Francky Catthoor. Template-based semi-automatic profiling of multimedia applications. In *Proceedings of the International Conference on Multimedia & Expo (ICME)*, pages 1061–1064. IEEE Computer Society Press, 2006.

[PBM+07] Miguel Peón-Quirós, Alexandros Bartzas, Stylianos Mamagkakis, Francky Catthoor, José Manuel Mendías, and Dimitrios Soudris. Direct memory access optimization in wireless terminals for reduced memory latency and energy consumption. In *Proceedings of International Workshop on Power And Timing Modeling, Optimization and Simulation (PATMOS)*, volume 4644 of *Lecture Notes in Computer Science (LNCS)*, pages 373–383. Springer-Verlag Berlin Heidelberg, 2007. ISBN 978-3-540-74441-2.

[PBM+15] Miguel Peón-Quirós, Alexandros Bartzas, Stylianos Mamagkakis, Francky Catthoor, José M. Mendías, and Dimitrios Soudris. Placement of linked dynamic data structures over heterogeneous memories in embedded systems. *ACM Transactions on Embedded Computing (TECS)*, 14(2):37:1–37:30, February 2015.

[PCB+05] L. Van Put, D. Chanet, B. De Bus, B. De Sutler, and K. De Bosschere. Diablo: a reliable, retargetable and extensible link-time rewriting framework. *Proc. of ISSPIT*, 0:7–12, 2005.

[PCD+01] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, and P. G. Kjeldsberg. Data and memory optimization techniques for embedded systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 6(2):149–206, 2001.

[PDN00] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. On-chip vs. off-chip memory: The data partitioning problem in embedded processor-based systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 5(3):682–704, 2000.

[Peó04] Miguel Peón-Quirós. *Estudio y optimización de la gestión de memoria dinámica en sistemas empotrados*. Proyecto de investigación de 3.^{er} ciclo, Universidad Complutense de Madrid, Departamento de Arquitectura de Computadores y Automática, Madrid, September 2004.

[Pis95] David Pisinger. *Algorithms for Knapsack Problems*. PhD thesis, University of Copenhagen, February 1995.

[Pis99] David Pisinger. An exact algorithm for large multiple knapsack problems. *European Journal of Operational Research*, 114(3):528–541, 1999.

[PM98] John G. Proakis and Dimitris G. Manolakis. *Tratamiento digital de señales. Principios, algoritmos y aplicaciones*. Prentice Hall, 3rd (Spanish) edition, 1998. ISBN 84-8322-000-8. 1048 pp.

[PMA+04] Francesco Poletti, Paul Marchal, David Atienza, Luca Benini, Francky Catthoor, and José Manuel Mendías. An integrated hardware/software approach for run-time scratchpad management. In *Proceedings of the Design Automation Conference (DAC)*, 2004.

[SGI06] SGI. Standard template library (STL), 2006. http://www.sgi.com/tech/stl/.

[SRS12] María Soto, André Rossi, and Marc Sevaux. A mathematical model and a metaheuristic approach for a memory allocation problem. *Journal of Heuristics*, 18(1): 149–167, February 2012.

[SSS11] Michael Sindelar, Ramesh Sitaraman, and Prashant Shenoy. Sharing-aware algorithms for virtual machine colocation. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 367–378, 2011.

[ST93] David B. Shmoys and Éva Tardos. An approximation algorithm for the generalized assignment problem. *Mathematical Programming*, 62:461–474, 1993.

[Sta94] Werner Staringer. Constructing applications from reusable components. *IEEE Softw.*, 11(5):61–68, 1994.

[STR07] STREP Project. Dynamic and distributed Adaptation of scalable multimedia coNtent in a context Aware Environment, IST-FP6-1-507113, 2007. http://danae.rd.francetelecom.com/index.php.

[Sub09] S. Subha. An exclusive cache model. In *International Conference on Information Technology: New Generations (ITNG)*, pages 1715–1716, Las Vegas, NV, USA, 2009. IEEE Computer Society Press. ISBN 978-1-4244-3770-2.

[SV96] M. Shreedhar and George Varghese. Efficient fair queueing using deficit round-robin. *IEEE/ACM Transactions on Networking*, 4(3):375–385, 1996.

[SWLM02] Stefan Steinke, Lars Wehmeyer, B. Lee, and Peter Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *Proceedings of Design, Automation and Test in Europe (DATE)*, page 409, 2002.

[UDB06] Sumesh Udayakumaran, Ángel Domínguez, and Rajeev Barua. Dynamic allocation for scratch-pad memory using compile-time decisions. *ACM Transactions on Embedded Computing (TECS)*, 5(2):472–511, 2006.

[Vo96] Kiem-Phong Vo. Vmalloc: A general and efficient memory allocator. *Software: Practice and Experience*, 26(3):357–374, 1996.

[VSM03] Manish Verma, Stefan Steinke, and Peter Marwedel. Data partitioning for maximal scratchpad usage. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 77–83, 2003. ISBN 0-7803-7660-9.

[VTS11] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight Persistent Memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 91–104, Newport Beach, California, USA, 2011. ACM Press. ISBN 978-1-4503-0266-1.

[VWM04] Manish Verma, Lars Wehmeyer, and Peter Marwedel. Cache-aware scratchpad allocation algorithm. In *Proceedings of Design, Automation and Test in Europe (DATE)*, 2004. ISBN 0-7695-2085-5-2.

[WDCM98] Sven Wuytack, Jean-Philippe Diguet, Francky Catthoor, and Hugo De Man. Formalized methodology for data reuse exploration for low-power hierarchical memory mappings. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, 6(4):529–537, December 1998.

[Wei95] Mark A. Weiss. *Estructuras de Datos y Algoritmos*. Addison-Wesley Iberoamericana, 1st (Spanish) edition, 1995.

[Wil65] M. V. Wilkes. Slave memories and dynamic storage allocation. *IEEE Transactions on Electronic Computers*, EC-14(2):270–271, April 1965.

[WJNB95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Proceedings of the International Workshop on Memory Management (IWMM)*, pages 1–116, London, UK, 1995. Springer-Verlag Berlin Heidelberg. ISBN 3-540-60368-9.

[WM95] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH Computer Architecture News*, 23(1):20–24, March 1995.

[WW88] Charles B. Weinstock and William A. Wulf. Quick Fit: An Efficient Algorithm for Heap Storage Allocation. *ACM SIGPLAN Notices*, 23(10):141–148, 1988.

[XMNB13] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. Bobtail: Avoiding long tails in the cloud. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 329–341. USENIX Association, 2013.

[ZDJ04] Ying Zheng, B. T. Davis, and M. Jordan. Performance evaluation of exclusive cache hierarchies. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 89–96, Washington, DC, USA, 2004. IEEE Computer Society Press. ISBN 0-7803-8385-0.

[ZZZ00] Zhao Zhang, Zhichun Zhu, and Xiaodong Zhang. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *Proceedings of the Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pages 32–41, Monterey, California, USA, 2000. ACM Press. ISBN 1-58113-196-8.

This text was started in Madrid long time ago,
its writing continued during several years
and was finally completed in Madrid, September 16, 2015.

And now, I'll rest on the grass for a while...