UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA
DEPARTAMENTO DE ARQUITECTURA DE COMPUTADORES Y AUTOMÁTICA

TESIS DOCTORAL

Mecanismos de gestión de escrituras en sistemas con tecnologías de memoria no volátiles
Write management mechanisms for systems with non-volatile memory technologies

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Roberto Alonso Rodríguez Rodríguez

DIRECTORES

Fernando Castro Rodríguez
Daniel Ángel Chaver Martínez

Madrid, 2017

Mecanismos de gestión de escrituras en sistemas con tecnologías de memoria no volátiles

(Write management mechanisms for systems with non-volatile memory technologies)

Roberto Alonso Rodríguez Rodríguez
DACYA ♠ UCM

Universidad Complutense de Madrid

Facultad de Informática

Departamento de Arquitectura de Computadores y Automática

Mecanismos de gestión de escrituras en sistemas con tecnologías de memoria no volátiles

(Write management mechanisms for systems with non-volatile memory technologies)

Memoria para optar al grado de doctor
presentada por

Roberto Alonso Rodríguez Rodríguez

Bajo la dirección de los doctores

Fernando Castro Rodríguez
Daniel Ángel Chaver Martínez

Madrid 2016

Tesis doctoral presentada por el doctorando Roberto Alonso Rodríguez Rodríguez en el Departamento de Arquitectura de Computadores y Automática de la Universidad Complutense de Madrid para la obtención del título de Doctor en Ingeniería Informática.

Terminada de escribir en Madrid el 31 de agosto de 2016.

Título:

Mecanismos de gestión de escrituras en sistemas con tecnologías de memoria no volátiles
(Write management mechanisms for systems with non-volatile memory technologies)

Doctorando:

Roberto Alonso Rodríguez Rodríguez (rrodriguezr@ucm.es)
Departamento de Arquitectura de Computadores y Automática, Universidad Complutense
Facultad de Informática, 28040 Madrid, España

Directores:

Fernando Castro Rodríguez (fcastror@ucm.es)
Daniel Ángel Chaver Martínez (dani02@ucm.es)

Esta tesis doctoral ha sido realizada dentro del grupo de investigación sobre Arquitectura y Tecnología de Sistemas Informáticos de la UCM, como parte de las actividades del proyecto de investigación: Arquitecturas y Tecnologías Emergentes, Eficiencia Energética Mediante Heterogeneidad (referencia TIN2012-32180).

ISBN-13: 978-84-617-4671-2

Agradecimientos

En unas horas estaré entregando la versión final de esta tesis en la que tanto esfuerzo e ilusión hemos puesto muchas personas. Los primeros que vienen a mi mente son Fernando y Dani, mis directores de tesis. Sus nombres aparecen al lado del mío en la portada y es ahí donde han estado durante estos años, a mi lado y poniendo toda su dedicación para que este trabajo fuese exitoso; gracias por darme siempre todo su apoyo sin el cual este trabajo no hubiera sido posible.

En este largo viaje hemos tenido la suerte de contar con la inestimable ayuda de mis companeros del grupo ArTeCS: agradecerle a Luis Piñuel su consejo y guía en momentos cruciales y a Juan Carlos Sáez por toda su colaboración sin la cual no hubiera sido posible llegar a este punto. Gracias a Manuel Prieto y Francisco Tirado por estar siempre pendientes y ocuparse de que tuviéramos todo lo necesario para poder llevar a cabo nuestro trabajo.

En octubre del 2014 llegaba a Zaragoza para realizar una colaboración con el grupo GAZ de la Universidad de Zaragoza. Agradecer a Pablo Ibáñez y Víctor Viñals por haberme acogido desde el primer momento y durante las dos estancias que realicé en su grupo; por su colaboración y por haberme dado la oportunidad de realizar el trabajo colaborativo con el que culmina esta tesis.

Gracias a Michael Huang por permitirme realizar una estancia de investigación en la universidad de Rochester, la cual me permitió ampliar mi perspectiva de la investigación y gracias a los chicos de su laboratorio por haberme hecho sentir en casa.

Me gustaría agradecer al profesor Narciso Martí Oliet, quien, como vicedecano de posgrado, día a día nos orienta en todos los trámites haciéndonos más ágiles con ellos.

Agradecerles a mis padres Elisa y Roberto; su cariño, cuidado, comprensión y apoyo durante toda mi vida y en especial durante estos años en los que me han demostrado el significado de la palabra generosidad. Sin duda son aquellos que más han sufrido mi ausencia; gracias por esperarme siempre todos los días al otro lado de esa fría pantallita.

De manera especial quisiera agradecer a Laura, quien ha estado conmigo desde el inicio de esta aventura, quien me ha enseñado el significado del trabajo duro y con su ejemplo me ha hecho querer ser mejor cada día, quien con su cariño y apoyo me ha ayudado a seguir cuando no parececía que hubiera luz al final del túnel. También quiero agradecer a Paula, Pepe, Adolfo, Beatriz y Sandrita por haberme acogido y darme una familia en Madrid.

Gracias a Rebe con quien la amistad ha transcendido el tiempo y el espacio y siempre ha estado más que dispuesta a ayudarme cuando he requerido de servicio técnico transcontinental.

Agradecer a Víctor M. Alfaro, Enrique Coen, Andres Díaz y Jorge Romero por no haber dudado en firmar un cheque en blanco que me ha dado la oportunidad de poder llegar hasta aquí.

Por último quiero agradecer a la Universidad de Costa Rica y al Ministerio de Ciencia y Tecnología de Costa Rica por el apoyo brindado para mi formación académica en el exterior.

Contents

List of Figures

List of Tables

Resumen

Desde el origen de los sistemas computacionales, el subsistema de memoria ha sido siempre uno de sus componentes fundamentales. Sin embargo, el diferente ritmo de evolución que han experimentado microprocesador y memoria se ha convertido en uno de los mayores desafíos que los diseñadores actuales deben abordar con el fin de desarrollar sistemas computacionales más potentes. A este problema, llamado brecha de memoria, se le suma la limitada escalabilidad y el elevado consumo de energía de las tecnologías de memoria convencionales (DRAM y SRAM), lo que ha llevado a considerar nuevas tecnologías de *memoria no volátil* (NVM por sus siglas en inglés) como posibles candidatas a reemplazar a las tecnologías convencionales. PCM y STT-RAM se postulan actualmente, entre las NVMs, como las mejores alternativas para ello.

Aunque PCM y STT-RAM poseen ventajas significativas sobre DRAM y SRAM, también adolecen de algunos inconvenientes que deben ser mitigados antes de que puedan ser utilizadas como tecnologías de memoria en la próxima generación de computadores. En particular, el elevado coste de las operaciones de escritura sobre ambos tipos de tecnología (tanto en términos de consumo de energía como en términos de latencia), así como la limitada durabilidad de las celdas de memoria PCM, que se vuelven inmutables tras una cantidad de escrituras sobre las mismas relativamente reducida, constituyen los principales inconvenientes de las tecnologías PCM y STT-RAM. En esta tesis se presentan dos propuestas con el objetivo de gestionar de forma eficiente las escrituras sobre estos tipos de memoria.

En la primera propuesta, enmarcada en un sistema con memoria principal implementada con tecnología PCM, se pretende reducir el número de escrituras a dicha memoria principal actuando a nivel del controlador de cache a través del algoritmo de reemplazo del nivel inmediatamente inferior en la jerarquía (el último nivel de cache, LLC por su siglas en inglés). Para ello, y como punto de partida, se han evaluado, aplicadas a la LLC, las políticas de reemplazo convencionales (orientadas a incrementar el rendimiento del sistema) en términos del número de escrituras a memoria principal que generan. Una vez determinado el algoritmo que produce una menor cantidad de escrituras a memoria principal, se han planteado una serie de modificaciones sobre dicho algoritmo encaminadas a

encontrar una política de reemplazo que obtenga un compromiso satisfactorio en el doble objetivo de minimizar el número de escrituras a memoria PCM (y por tanto aumentar su durabilidad y reducir el consumo de energía de la misma) y no penalizar el rendimiento. Los algoritmos propuestos se han codificado e integrado en el simulador arquitectónico *gem5*, de modo que se simula el entorno deseado en donde la memoria principal se modela de acuerdo a las características de una memoria PCM y el último nivel de cache opera con los algoritmos de reemplazo diseñados. Se evalúa el comportamiento de dichos algoritmos sobre distintos tipos de aplicaciones, tanto secuenciales como paralelas y también cargas multiprogramadas. Los resultados obtenidos revelan que, dependiendo del escenario evaluado (uni-procesador o multiprocesador) y comparado con usar el algoritmo de reemplazo LRU convencional, algunas de nuestras propuestas consiguen extender la vida útil de la memoria principal hasta un 45% y reducir el consumo de energía en la jerarquía de memoria hasta un 9%, todo ello sin apenas penalizar el rendimiento.

En la segunda de las propuestas, enmarcada en un entorno en el que el último nivel de cache se construye utilizando tecnología STT-RAM, se presenta un mecanismo para predecir escrituras innecesarias a este último nivel de cache, de modo que las escrituras que sean identificadas como ineficientes sean filtradas a la LLC y se realicen directamente en la memoria principal. Para ello, en primer lugar se exploró la localidad de reuso que exhibe el flujo de referencias que llega a la LLC, en contraposición a la localidad temporal que predomina en los niveles de cache más cercanos al procesador. Una vez comprobada y evaluada tal propiedad, se planteó la forma de explotarla utilizando un elemento capaz de detectar aquellos bloques que presentan reuso. Este detector de reuso se encarga de gestionar los contenidos de la LLC, de modo que los bloques para los que se detecta reuso son insertados en la LLC, mientras que aquellos para los que no se identifica reuso no son insertados en la LLC, reduciendo así el número de escrituras en dicho nivel y reduciendo también el consumo de energía en el mismo. Para evaluar esta técnica se procedió a codificar la inclusión del detector de reuso, así como las modificaciones necesarias en el protocolo de coherencia, utilizando el simulador arquitectónico *gem5*, donde también se modeló la LLC conforme a las características de las memorias STT-RAM. Posteriormente, se evaluó la propuesta usando aplicaciones secuenciales y cargas multiprogramadas en entornos multiprocesador. De acuerdo a los resultados obtenidos, esta técnica de gestión de contenidos, aplicada a una LLC implementada con tecnología STT-RAM, consigue, comparado con un sistema con una LLC

construida con tecnología STT-RAM en la que no se utiliza detector de reuso, reducciones de energía en la LLC compartida de un sistema multiprocesador de alrededor del 40%, reducciones adicionales en el consumo de energía de la memoria principal de más del 6% e incrementos en el rendimiento del sistema que oscilan en el rango 3-7% dependiendo de las características particulares de los distintos sistemas multiprocesador evaluados.

Como conclusiones, podemos destacar que es posible diseñar soluciones arquitectónicas que mitiguen las deficiencias de las NVMs y faciliten su recorrido en el camino emprendido hacia convertirse en las sustitutas naturales de las cada vez más agotadas tecnologías convencionales. Actuando a distintos niveles se ha demostrado que PCM y STT-RAM pueden ser alternativas más que eficientes al uso de DRAM y SRAM como tecnologías de memoria principal y último nivel de cache respectivamente.

Palabras Clave: PCM, Memoria de Cambio de Fase, Durabilidad, Celdas Defectuosas, Jerarquía de Memoria, Políticas de Reemplazo de Cache, Reducción de Escrituras, gem5, STT-RAM, Detector de Reuso, Localidad de Reuso, Ahorro de Energía, Rendimiento, Contadores de Monitorización de Rendimiento, PMCTrack, Monitorización de Cache, Intel CMT.

Abstract

Since the beginning of computer systems, the memory subsystem has always been one of their essential components. However, the different pace of change between microprocessor and memory has become one of the greatest challenges that current designers have to address in order to develop more powerful computer systems. This problem, called memory gap, is further compounded by the limited scalability and the high energy consumption of conventional memory technologies (DRAM and SRAM), which has leaded to consider new *non-volatile memory* (NVM) technologies as potential candidates to replace them. Among NVMs, PCM and STT-RAM are currently postulated as the best alternatives.

Although PCM and STT-RAM have significant advantages over DRAM and SRAM, they also suffer from some drawbacks that need to be mitigated before they can both be employed as memory technologies for the next computers generation. Notably, the slow and energy-hungry write operations on both technologies, and the limited endurance of PCM cells, which become unchangeable after performing a relatively reduced amount of writes on them, are the main constraints of PCM and STT-RAM technologies. This thesis presents two proposals aimed to efficiently manage the write operations on this kind of memories.

The first proposal, conceived for a system with a PCM-based main memory, is intended to reduce the number of writes to the main memory by operating at the cache controller level through the replacement policy used in the immediate-lower memory hierarchy level (the last-level cache, LLC). For this purpose, and as the starting point, the conventional LLC replacement policies (oriented to improve the system performance) have been evaluated in terms of the amount of writes generated to main memory. Once the algorithm reporting the lowest amount of writes to main memory has been identified, several changes are proposed aimed to find a replacement policy satisfying the twofold goal of minimizing the number of writes to PCM main memory (and hence reducing the corresponding energy consumption) and not penalizing the system performance. The proposed algorithms have been encoded and integrated in the *gem5* architectural simulator, so that the desired environment, where the main memory is modeled according to PCM memory features and the last-level cache operates with the designed replacement

policies, is simulated. The behavior of these algorithms when running different kind of applications, both sequential and parallel programs as well as multiprogrammed workloads, is evaluated. Experimental results show that, on average, compared with a conventional LRU algorithm, some of our proposals manage to extend the memory lifetime up to 20–45%, also reducing the energy consumption in the memory hierarchy by up to 9% and hardly degrading performance.

In the second proposal, conceived for a system with an STT-RAM last-level cache, a mechanism aimed to predict unnecessary writes to this last-level cache is presented, so that those writes identified as useless are filtered in the LLC and performed directly in the main memory. For this purpose, first it was explored the reuse locality that the stream of references arriving at the LLC exhibits, unlike the temporal locality that exhibits the stream of references arriving to the cache levels closer to the processor. Once verified and evaluated this feature, it was exploited by using an element able to detect those blocks exhibiting reuse. This reuse detector is in charge of managing the LLC contents, so that the blocks predicted to be non-dead blocks are inserted in the LLC while those predicted to have not reuse bypass the LLC, hence reducing the amount of writes to this level and also the corresponding energy consumption. For the evaluation of this approach, the inclusion of the reuse detector (as well as the required modifications in order to adapt the coherence mechanism) was encoded using the *gem5* architectural simulator, where also the LLC was modeled according to STT-RAM memory features. Then the proposal was evaluated using sequential applications and multiprogrammed workloads in a multiprocessor environment. Experimental results reveal that this content management technique, applied to an STT-RAM LLC and compared to an STT-RAM LLC baseline where no reuse detector is employed, reports energy reductions in the shared LLC of a multiprocessor system of around 40%, an additional energy reduction of more than 6% in the main memory, and improves performance by 3-7% depending on the specific features of the different multiprocessor systems evaluated.

In conclusion, we must highlight that it is possible to design architectural solutions that mitigate the shortcomings of NVMs and facilitate their route to become the natural replacement of the exhausted conventional technologies. By addressing this issue at different levels, it has been shown that PCM and STT-RAM may be efficient alternatives to the usage of DRAM and SRAM as technologies of the main memory and the last-level cache, respectively.

Keywords: PCM, Phase Change Memory, Endurance, Failing Cells, Memory Hierarchy, Cache Replacement Policies, gem5, STT-RAM, Reuse Detector, Reuse Locality, Write Reduction, Energy Savings, Performance, Performance Monitoring Counters, PMCTrack, Cache Monitoring, Intel CMT.

1

Introduction

The different evolution experienced by microprocessors and memory subsystems has become one of the greatest challenges that current designers have to deal with in order to develop more powerful computer systems. Moreover, a large memory capacity is currently required given the huge amount of information that people need to store in the computer systems. In order to efficiently manage the memory capacity, a *memory hierarchy* is usually employed to create the illusion of a perfect memory, i.e. a memory with response time near to zero, low cost and unlimited storage capacity.

1.1 Memory hierarchy

In order to establish a memory hierarchy, several memory levels are used, where each level can employ different manufacturing technologies. The physical registers constitute the memory level closest to the microprocessor, offering the lowest response time at the expense of a very limited capacity (only a few registers are available). Then, several levels of cache memory are included, usually built employing SRAM technology (Subsection 1.2.2). Given that the cache memories and the processor are based on the same primary technology (CMOS) and are usually placed within the same integrated circuit, the cache constitutes a significantly fast level among the memory levels. However, given the low integration capacity of the SRAM technology and especially its high cost, the cache memories also provide a reduced storage capacity.

Moving further away from the processor, the next memory level within the hierarchy is the main memory, which is traditionally manufactured using DRAM technology (Subsection 1.2.1). The main memory provides higher storage capacity but also higher latency than the cache levels. It is also the last-level in the memory hierarchy managed by hardware.

Finally, a secondary memory managed by the operating system (OS) serves as a backup for the main memory. This level is usually manufactured with magnetic or flash technologies (Subsection 1.3.3.1) and, regardless of the type of technology used, it provides the highest storage capacity and latency in the memory hierarchy. Figure 1.1 illustrates the diagram of a typical memory hierarchy[1], where also external memory devices, which follow the same trends illustrated, are shown at the bottom of the pyramid.

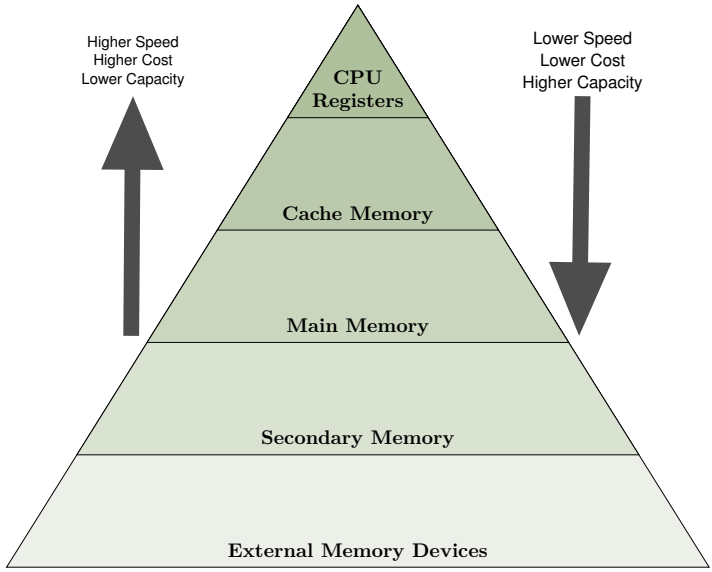


Figure 1.1: Typical Memory Hierarchy.

The control of the data traffic through the different levels of the memory hierarchy is carried out using different algorithms aimed to make the memory system efficient by exploiting the specific characteristics of the executed programs (spatial and temporal locality) or the specific memory system implementation (number of cache levels, replacement policy, coherency protocol, etc.). The cache controller and the memory controller are in charge of the cache memory and the main memory management, respectively. Conversely, the OS is responsible of managing the data traffic between the main memory and the secondary memory.

These algorithms are focused on different aspects. For example, a *write policy* for

[1]The memory level closest to the processor is referred to as the *lowest level* within the hierarchy. Likewise, the level farthest to the processor is called the *highest level*.

updating the information across the different memory levels must be established. Typically, a write-back policy (where data are only modified between adjacent levels of the hierarchy once the blocks the data belong to are evicted from the corresponding level) or a write-through policy (i.e. when data are modified in a level of the hierarchy, they are also immediately updated in the next higher level) is used. Or, as another example, a *replacement policy* is employed for selecting the block to evict when a replacement in a memory level is required. The most typical replacement policy is Least Recently Used (LRU), where the replaced block is the one that was referenced the furthest in the past.

1.2 Conventional memory technologies

In this section we explore the main features of the memory technologies currently employed in most computer systems. Notably, we detail DRAM and SRAM and we analyze the major constraints that both kind of technologies suffer from.

1.2.1 Dynamic Random-Access Memory (DRAM)

Traditionally, DRAM has been the technology used for implementing the main memory in most computer systems. Currently, DRAM has scaled down to 20nm and has a lifespan of 10^{15} write cycles [1], [2], [3]. Figure 1.2 shows the structure of a DRAM memory array and the DRAM cell [4].

As the zoom of Figure 1.2 illustrates, the DRAM cell is mainly based on a field effect transistor (FET) and a capacitor which contains the logical datum (one if it is charged and zero if it is discharged). The memory array is made up of many of these cells, organized in rows and columns. The DRAM controller reads/writes an individual storage cell by means of a row address and a column address. Special circuits called sense amplifiers (SA) are used to detect the value stored at the capacitor when this gets connected to its associated bitline. They are necessary for detecting and amplifying the tiny logical signals represented by the electric charges in the cells.

Because of leakage current in the DRAM cell, the data need to be refreshed from time to time, which requires extra circuitry. The major leakage paths in a DRAM cell stem from reverse junction leakage from the storage node, and gate induced

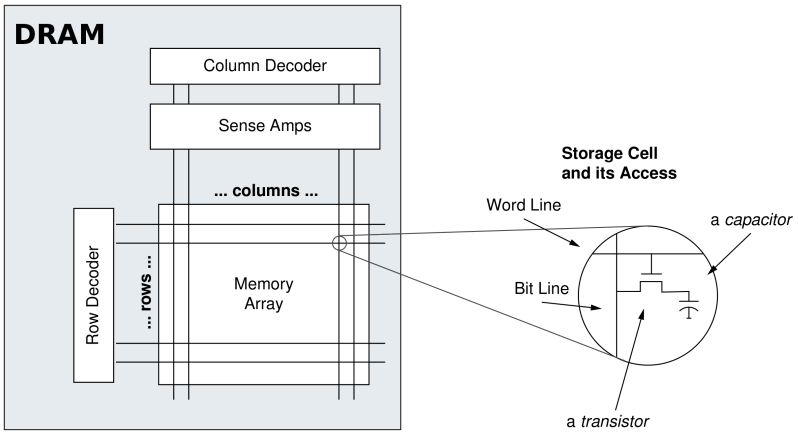


Figure 1.2: Structure of DRAM.

drain leakage current. Moreover, the reading operation is destructive, so the data must be refreshed after the reading process.

In the DRAM cells the capacitors must be large enough not only to store enough charge, but also to mitigate the amount of leakage currents. Also, the transistors must be able to exercise full control in the channel between the drain and source. For these reasons, it is not possible to implement DRAM memories below 20nm [1], [5]. In addition to the challenge of DRAM integration, other obstacles are arising related to this technology, such as a high cost and a high energy consumption. Thus, in the case of high performance servers, the highest cost and energy consumption are derived from the main memory [6].

1.2.2 Static Random-Access Memory (SRAM)

SRAM has traditionally been the technology used for implementing the cache memory levels in computer systems. As shown in Figure 1.3(a), the SRAM cell is made up by two control transistors and a bistable based on two inverters with a feedback loop. The read is performed by turning on the control transistors –by means of the word line (WL)–, so that the stored value is read through the bit lines (BL). For writing a value in the cell, the control transistors have to be turned on and the value to store must be written to the BL lines. The control transistors must be larger than the inverter transistors, so that when the new datum to store

is different than the data currently stored in the cell, the control transistors can provide more current than the inverter transistors and the voltage swaps to the new value. Note that unlike DRAM, where the information is represented by the amount of charge in the capacitors, in SRAM cells it is represented based on voltage levels.

As shown in Figure 1.3(b), the SRAM cell (usually called 6T cell) is actually made up of six transistors. Note that DRAM cells just need one transistor and a capacitor to store a logical value, hence reporting higher integration capacity. However, unlike the case of DRAM cells, the stored data do not need to be refreshed periodically, given that the stored data are not lost. The major problem with SRAM technology is that leakage increases dramatically as the transistor size decreases. Because cache memories generally occupy a chip area similar or even bigger than the cores, their contribution to the total energy consumption is quite significant.

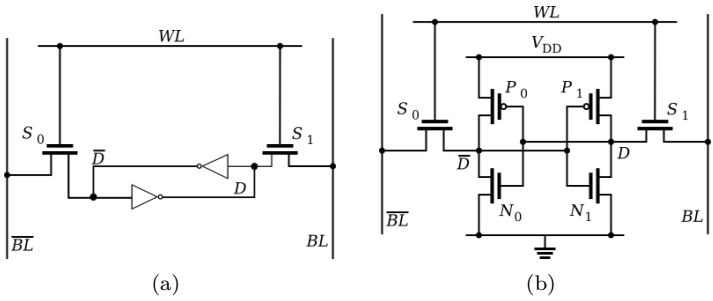


Figure 1.3: (a) Schematic of an SRAM cell. (b) SRAM 6T cell.

1.2.3 Constraints of conventional memory technologies

As stated earlier, the different pace of change between the microprocessor and the memory subsystem is one of the greatest challenges designers have to address. This problem, called memory gap [7], [8], lies in that, over the last decades, the performance improvement of the processor is significantly higher than the improvement achieved in the memory systems. Figure 1.4, adapted from [8], illustrates the concept of memory gap. Notably, this figure represents how both the processor and the memory system have evolved with respect to the performance they delivered in 1978. Note that in the case of the memory, the performance is measured

in terms of the DRAM access latency. According to the data depicted (dotted red line), the processor has roughly followed a 25% improvement per year from 1978 to 1986, a 52% improvement from 1986 until 2000, and a 20% improvement from 2000 to 2010. The dashed red line from 2005 to 2010 indicates that there is almost no change in processor performance on a per-core basis in these years. Overall, the processor performance has improved (with respect to the processor performance in 1978) more than four orders of magnitude. Table 1.1 identifies the real commercial processors corresponding to data shown in Figure 1.4 (continuous blue line). Conversely, the memory performance has hardly improved one order of magnitude with respect to the memory performance exhibited in 1978.

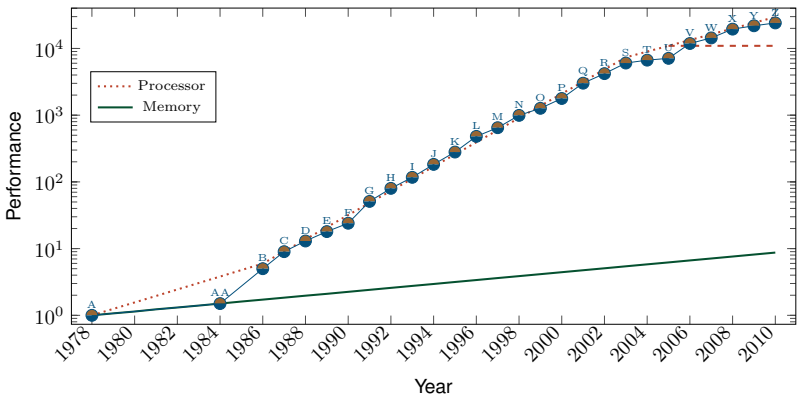


Figure 1.4: Evolution of processor and memory performance.

Furthermore, among the main conventional memory technologies (DRAM and SRAM) there is also some imbalance in their properties that further accentuates this gap, since there is not a memory technology that clearly evolves well in both performance and capacity. Figures 1.5 and 1.6 show the evolution along the last decades of SRAM and DRAM in terms of performance and capacity with respect to the values they exhibited in 1980.

As shown in these figures, performance is increasing at a higher rate for SRAM than DRAM, whereas in the case of capacity, the trend is the opposite. Moreover, the memory gap problem has recently increased because both DRAM and SRAM technologies have serious scalability issues that significantly restrict their development. For theses reasons, during the lastest years there has been a lot of research

Year	Processor	Performance	Symbol
1978	AX-11/780, 5 MHz	1	A
1984	VAX-11/785, 7.5MHz	1.5	AA
1986	VAX 8700, 22 MHz	5	B
1987	Sun-4/260, 16.7 MHz	9	C
1988	MIPS M/120, 16.7 MHz	13	D
1989	MIPS M2000, 25 MHz	18	E
1990	IBM RS6000/540, 30 MHz	24	F
1991	HP 9000/750, 66 MHz	51	G
1992	Digital 3000 AXP/500, 150 MHz	80	H
1993	IBM POWERstation 100, 150 MHz	117	I
1994	Digital Alphastation 4/266, 266 MHz	183	J
1995	Digital Alphastation 5/300, 300 MHz	280	K
1996	Digital Alphastation 5/500, 500 MHz	481	L
1997	AlphaServer 4000 5/600, 600 MHz 21164	649	M
1998	Digital AlphaServer 8400 6/575, 575 MHz 21264	993	N
1999	Professional Workstation XP1000, 667 MHz 21264A	1267	O
2000	Intel VC820 motherboard, 1.0 GHz Pentium III processor	1779	P
2001	IBM Power4, 1.3 GHz	3016	Q
2002	Intel D850EMVR motherboard (3.06 GHz, Pentium 4 processor with Hyper-Threading Technology)	4195	R
2003	Intel Xeon EE 3.2 GHz	6043	S
2004	AMD Athlon 2.6 GHz	6681	T
2005	AMD Athlon 64, 2,8 GHz	7108	U
2006	Intel Core 2 Extreme 2 cores, 2.9 GHz	11865	V
2007	Intel Core Duo Extreme 2 cores, 3.0 GHz	14387	W
2008	Intel Core i7 Extreme 4 cores 3.2 GHz (boost to 3.5 GHz)	19484	X
2009	Intel Xeon 4 cores 3.3 GHz (boost to 3.6 GHz)	21871	Y
2010	Intel Xeon 6 cores, 3.3 GHz (boost to 3.6 GHz)	24129	Z

Table 1.1: Evolution of commercial processors performance [8].

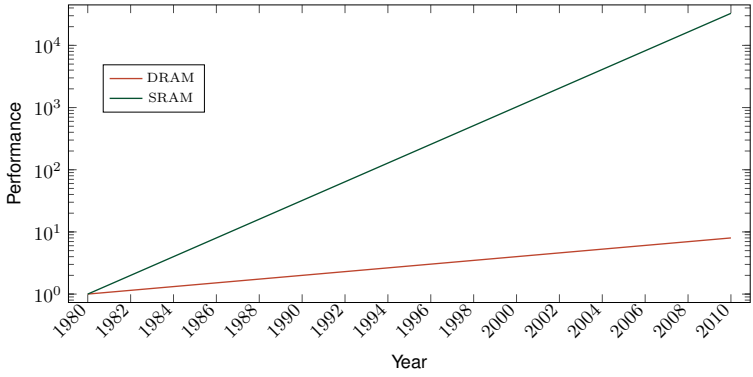


Figure 1.5: SRAM and DRAM performance evolution.

on new memory technologies oriented to deal with these issues. Notably, there is a whole set of new memory technologies that have recently experienced a significant level of progress in order to become real contenders to replace conventional memory technologies in a near future. In the next section we explore them.

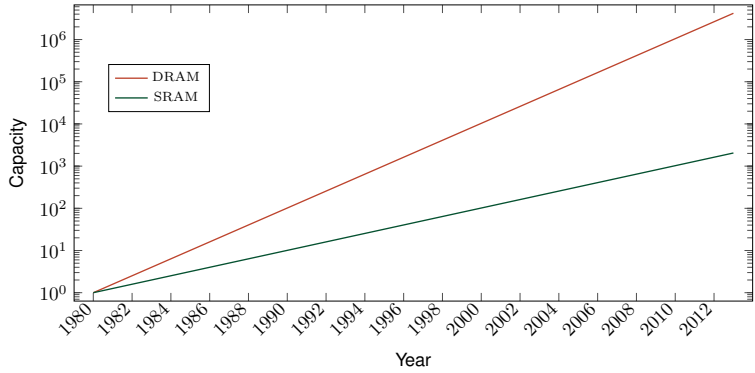


Figure 1.6: SRAM and DRAM capacity evolution.

1.3 Emerging memory technologies

Non-Volatile Memories (NVMs) are a set of memory technologies that have been explored by researchers for many years. These technologies, unlike DRAM and SRAM, are non-volatile, i.e. the stored data are not erased when the energy supply is cut off. Moreover, these technologies exhibit a high energy efficiency and thus can solve the problem of high energy consumption of DRAM and SRAM. Next we recap the main features and the operation of some of the main technologies belonging to the NVM set.

1.3.1 Magnetic RAM (MRAM) and Spin-Transfer Torque RAM (STT-RAM)

The concept of magnetic RAM (MRAM) originates from the ferrite cores memory developed in the 1950s [9]. Although many similar approaches were proposed for decades, such as domains for data storage[10], magnetostriction for moving the domains walls in magnetic wire [11], employing different material alloys to reduce energy and increase bit density [12], etc., none of these mechanisms had wide acceptance because the performance, energy consumption and cost of DRAM always overcame them.

The discovery of giant magneto resistance (GMR) [13], [14] put back the interest in MRAMs. GMR is a quantum effect observed in structures called magnetic

tunnel junction (MTJ). A MTJ is composed of a non-magnetic but conductive material which is placed between two ferromagnetic material layers, e.g. Fe/Cr/Fe. The electrical resistance changes significantly depending on the magnetic field direction of the magnetic layers (parallel or antiparallel). Henceforth, when we talk about MRAM, we will be referring to structures that make use of the GMR effect. Figure 1.7 shows the structure of an MRAM cell [2] formed by an MTJ. The MTJ is formed by a fixed layer and a free layer: if the two ferromagnetic layers have different directions, the MTJ resistance is high, indicating a "1" state, whereas if the two layers have the same direction, the MTJ resistance is low, indicating a "0" state. The inputs for the MRAM cell are the following: one bit line (BL), one word line (WL) and one write word line (WWL). The WL is turned on for reading the cell value, and current through the MTJ is sensed at the BL. For programming the cell, the WL is turned off and the free layer orientation is set based on the magnetic field generated by the current flowing through the BL and WWL. This structure has the disadvantage of requiring a high current density (in the order of 10 MA/cm^2) to program it.

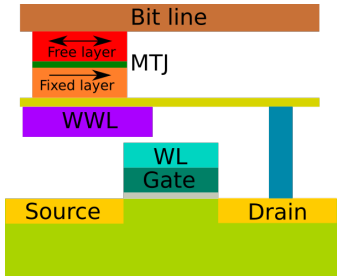
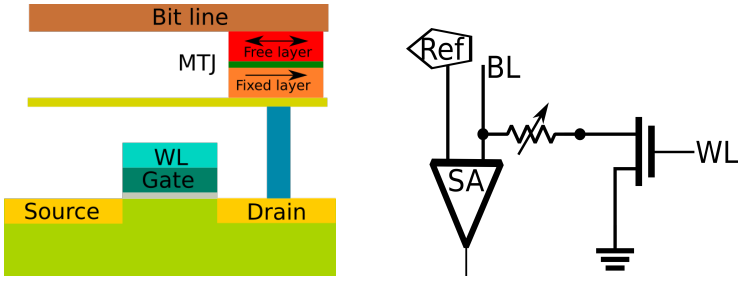


Figure 1.7: MRAM cell structure.

The spin-transfer torque (STT) effect [15] occurs when a spin-polarized current of electrons flows through a magnetic layer, so it transfers spin angular momentum to that layer, which results in a torque on the magnetization. If the torque is strong enough, it can set the magnetic direction of that layer.

In the case of an MTJ, the fixed layer is used to generate the spin-polarized current; only the electrons whose spin is aligned with its magnetization direction pass to the free layer. If the STT effect is sufficiently strong, the torque causes the magnetization of the free layer to be set to the desired direction.



(a) (b)

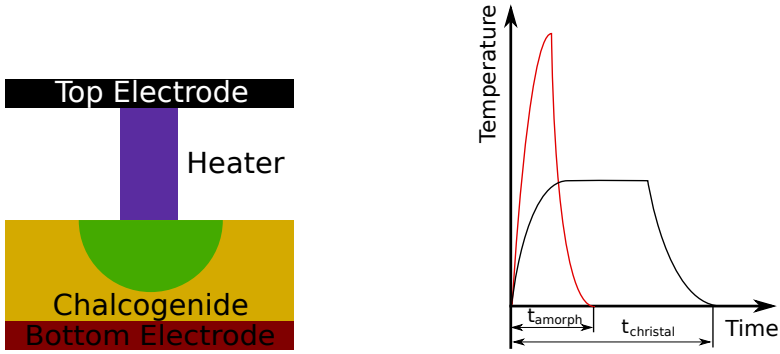
Figure 1.8: (a) STT-RAM cell structure. (b) STT-RAM equivalent circuit.

Spin-Transfer Torque memories (STT-RAM) make use of STT effect to program the memory cell. Figure 1.8(a) shows an STT-RAM cell, where the WWL is not included, as opposed to an MRAM cell. A read operation to an MTJ is performed by applying a small voltage difference between the two electrodes of the cell (BL and WL), and then sensing the current flow. A write operation is performed by applying a large voltage difference between the two electrodes for a given duration, called write pulse width (see Figure 1.8(b), where the STT-RAM cell is represented as a variable resistor). The current density is reduced to 1-5 MA/cm^2. STT-RAM has read and write speeds of 20ns and a lifespan of 10^{12} write cycles. All these features make STT-RAM a promising candidate to become a universal memory.

1.3.2 Phase Change Memory (PCM)

A PCM cell (Figure 1.9(a)) consists of two enveloping electrodes, a thin layer of chalcogenide material (e.g. $Ge_2Sb_2Te_5$, GST in short) and a heating element [2]. This heater is just a material that produces Joule heat when an electrical current is driven through, warming the chalcogenous material. In order to write a logical value in the cell, the heating element is employed to apply electrical pulses to the chalcogenide, which changes the properties of the material resulting in two different physical states: amorphous (high electrical resistance) and crystalline (low resistivity). Notably, if a fast high-intensity current pulse is applied, the material goes over 600°C and melts. Then, it is cooled down quickly, going to an amorphous state (RESET process), in which resistance is high. If the pulse is long and with a low intensity, the material goes through an annealing process allowing the molecules to re-crystallize, lowering the electrical resistance (SET process).

Thus, the chalcogenide switches easily, rapidly and in a reliable way between both states. Figure 1.9(b) shows graphically the heat pulses used to set and reset a PCM cell.



(a) (b)

Figure 1.9: (a) PCM cell. A heating element (purple) is attached to a chalcogenous material (yellow/green), and enclosed between the two electrodes. The bit of material attached to the heater forms the programmable volume (green), *i.e.* the part of material that will experiment the phase change. (b) Heat pulses used to set and reset.

Regarding writing to a PCM cell it is worth to note that after a certain number of writes (around 10^8), the heating element is detached from the cell as a consequence of the continuous expansions/contractions derived from the writing process, leaving the cell in a stuck at failure state. From that moment on, although the cell is still readable, the stored valued cannot be changed anymore. The process for reading the stored value just consists in applying a low current to the cell in order to measure the associated resistance. Table 1.2 shows the two basic operations on a PCM cell and the relationship with the logical and physical states.

Operation	Logical Value	Impedance	Material State
Set	1	Low	Crystalline
Reset	0	High	Amorphous

Table 1.2: States of a PCM cell.

PCM achieves an integration density higher than DRAM. So far, 3nmx20nm pro-

totypes have been manufactured. Also, a PCM cell can store multiple bits and it is a non-volatile memory. PCM is currently presented as the best option to replace DRAM as main memory technology.

1.3.3 Other NVM technologies

1.3.3.1 Flash memory

The cell of flash memory [2] is composed by a transistor with an electrically programmable threshold voltage. This is achieved by introducing an additional conducting layer in the gate stack of the conventional field effect transistor called the floating gate. The memory cell is programmed by applying bias voltages to the control gate and the drain, which causes tunnelling and trapping of charges in the floating gate, resulting in a shift of the effective threshold voltage of the device. This technology has been widely accepted as secondary and external memory. However, due to its high latency and low endurance (which worsens when the integration density is further increased), flash memory is not a good candidate for implementing main memory or cache.

1.3.3.2 Racetrack or Domain Wall Memory (DWM)

In a racetrack memory, information is stored on an U-shaped nanowire as a pattern of magnetic regions with different polarities (bits are stored like in serial tapes). The U-shaped magnetic nanowire is an array of keys, which are arranged vertically like trees in a forest. The nanowires have regions with different magnetic polarities, and the boundaries between the regions represent 1s or 0s, depending on the polarities of the regions on either side. A separate nanowire, perpendicular to the U-shaped "racetrack", writes data by changing the polarity of the magnetic regions. A second device at the base of the track reads the data. Data can be written and read in less than a nano-second [16]. Because of the reduced number of write and read heads, it is necessary to shift the array and locate the desired bit over an access head. Racetrack is a promising candidate to become a universal memory but, unlike PCM or STT-RAM technologies, it is not mature enough yet.

1.3.3.3 Ferro-Electric RAM (FeRAM)

The cell of a Ferro-Electric RAM [17], [18] is composed by a ferro-electric material enclosed between two metallic electrodes, forming a capacitor with the ferroelectric material in between instead of a dielectric material. The ferroelectric material is able to store a remnant polarization induced by the electric field formed by the two electrodes that compose the capacitor. Thus, the cell has two stable states associated to the direction of the remnant polarization which depends on the direction of the current through the electrodes. This cell behaves similarly to DRAM, in the sense that the read operation is destructive and a FET is necessary for isolating the cells. The main benefit of this technology is that programming pulses can be shorter than 50 ns and that FeRAM endurance is around 10^{12} cycles, both for reading and writing. Its main drawbacks are that reading is destructive and that it is not highly scalable, due to the need of including a capacitor of at least 20nmx20nm (however, this drawback could be overcame with three-dimensional circuitry), and also that it requires temperatures up to 700°C for manufacturing the ferroelectric element.

1.3.3.4 Resistive RAM (ReRAM)

Resistive RAM cell [2] (ReRAM) is implemented by placing an insulating material between two metal electrodes. Although usually this system would not lead electrons from one electrode to another, a filament conduction path between the two electrodes can be created by applying a high voltage. This path could be created by induced defects in the insulating material such as metal defect migration. Applying other voltage can eliminate the filament (reset the cell) and generate a new high impedance path between the two electrodes. This technology is not mature enough yet because there are a lot of materials that can be used as potential storage cells and researchers still continue to analyze the mechanism that generates the switching effect and the scalability that could be reached.

1.4 Motivation and objectives

In response to the scalability and energy consumption constraints that both DRAM and SRAM technologies exhibit, the NVMs arise as candidates to replace them for the next generation of computer systems. Among the different NVMs, the PCM

technology reveals as the prime contender to replace DRAM as main memory technology, while STT-RAM postulates as the optimal candidate to replace SRAM in the cache levels of the memory hierarchy. Although both PCM and STT-RAM significantly improve the integration capacity and energy consumption of DRAM and SRAM respectively, they also exhibit some drawbacks, such as the slow and energy-hungry write operations on both technologies, that need to be mitigated before they become true design alternatives.

Notably, PCM cells show a serious constraint related to the endurance [2], [6], [19], [20], [21], because they only allow around 10^8 cycles of writing and erasing (due to the detachment of the heating element from the cell explained earlier), while DRAM is estimated to exceed the 10^{15} cycles. This implies that a main memory built using PCM technology would exhibit a lifetime clearly lower than DRAM, constituting an important restriction in the potential adoption of the PCM technology. Moreover, although STT-RAM consumes around an order of magnitude less static power than SRAM, the STT-RAM write latency and write energy consumption are significantly worse than those of SRAM.

For these reasons, the main goal of this thesis is to explore architectural techniques that enable to exploit the various advantages of PCM and STT-RAM as main memory and cache technologies respectively and, at the same time, to significantly mitigate the endurance (only in PCM) and write operation constraints. For this purpose, we first address the endurance problem of PCM technology by proposing some new replacement algorithms for the last-level cache (LLC) with the goal of cutting down the write traffic to the PCM main memory, hence extending its lifetime and decreasing the corresponding energy consumption, without degrading system performance. The rationale behind that is that conventional LLC replacement algorithms just pursue to increment the hit rate and therefore the system performance, but are absolutely-unaware of the amount of writes to the main memory that they imply, so they do not take into consideration at all the corresponding impact on the memory endurance. Then, we address the write energy consumption and write latency problems of STT-RAM LLCs by using a management scheme that selects the contents of the cache aiming to exploit a special kind of locality, referred to as reuse locality, instead of temporal locality. Specifically, the proposal includes a hardware element (called reuse detector) for detecting blocks that do not exhibit reuse, in order to avoid their insertion in the LLC, hence reducing the number of write operations (also improving performance)

and the energy consumption in the STT-RAM.

1.5 Outline

The organization of this document is as follows:

- Chapter 2: recaps the related work in the research field this thesis belongs to.
- Chapter 3: details and evaluates the proposal conceived to mitigate the endurance and write energy consumption constraints of a PCM-based main memory by dealing with the LLC replacement policy.
- Chapter 4: details and evaluates the proposal conceived to efficiently address the write energy consumption of STT-RAM technology by managing the contents of an STT-RAM LLC using a reuse detector.
- Chapter 5: summarizes the conclusions derived from this thesis and also gives some hints on how this work could be continued.

2

Related Work

As described in the previous chapter, in this thesis we explore some architectural techniques for mitigating the write energy consumption and write latency problems that both PCM and STT-RAM suffer from and also the endurance problem of PCM technology. In order to address the memory lifetime constraint that PCM exhibits, in Chapter 3 we introduce new replacement algorithms for the last-level cache oriented to reduce the amount of writes generated to the PCM-based main memory system. Thus, in this chapter, and as a previous step to get a better understanding, we first do a thorough review of the conventional cache replacement policies proposed in the literature (Section 2.1), and then we analyze some replacement algorithms and other architectural techniques specifically designed for improving the management of a PCM memory (Section 2.2). Also, in Chapter 4 we deal with STT-RAM constraints by employing a cache management scheme for improving the content selection mechanisms of an STT-RAM-based LLC, in order to reduce the number of write operations and the energy consumption in this cache level. Therefore, we also include in this chapter (Section 2.3) the review of some proposals aimed to improve the management of STT-RAM memories.

2.1 Conventional cache replacement policies

Many cache replacement algorithms have been proposed in the literature in the latest decades. In this section we focus on the explanation of those cache replacement policies more closely related to our work, but that were not conceived specifically for NVMs-based systems.

Prior to that, we briefly review some key concepts related with the management of the replacements in the cache: When a new block arrives at the cache and the frame (in a direct-mapped cache) or the set (in a set-associative cache) where it maps to is full, the cache replacement policy must decide which block to replace. In

general, as Bélády established in [22], the best decision in terms of performance is to choose the block that will not be referenced again for the longest time. Precisely knowing the future in advance is not possible, but, in computer systems, we usually employ the past behavior for obtaining hints about the future behavior. As such, the different cache replacement policies proposed in the literature for improving performance try to predict which one is the block that will not be referenced again for the longest time, based on the analysis of past information. This information, which constitutes the *replacement state* of a block, is gathered at different points of the block residency in cache, specifically, at *insertion* (i.e. when the block initially arrives at the cache), and at *promotion* (i.e. every time the block experiences a hit, either for reading or for writing it). Then, when a block has to be selected for *victimization*, the replacement policy makes the decision based the *replacement state* of each block.

We can divide the replacement policy into 3 sub-policies:

- *Insertion sub-policy:* This sub-policy determines the initial *replacement state* to assign to a block when it is filled into the cache.
- *Promotion sub-policy:* This sub-policy determines how to update the *replacement state* of a block when it experiences a hit.
- *Victimization sub-policy:* When a block has to be evicted for accommodating a new one, this sub-policy chooses the victim block by comparing the *replacement states* of the candidate blocks.

The access pattern to the various cache levels in the hierarchy is different. For example, in the lowest level, a strong temporal locality is observed for most applications, which leads to designing replacement policies that seek to exploit such locality. In these levels, the Least Recently Used (LRU) policy (or any approximation of LRU that allows to reduce the hardware devoted for implementing it) is the most frequently employed policy, given that it provides satisfactory results with a reasonable hardware overhead. Conversely, when a block reaches the LLC, the temporal locality has already been almost totally exploited by the lower levels, so the replacement policy must exploit other features. In this work we will focus on the highest cache level, so we will next describe some recent replacement algorithms applied to the LLC for improving performance, as well as the classic LRU

and one extended approximation of this algorithm (Not Recently Used, or NRU). We defer to Subsection 2.2.1 the description of LLC replacement policies focused on reducing the amount of writes generated in an NVM-based scenario.

2.1.1 Least Recently Used (LRU)

The LRU policy constitutes the baseline algorithm any new proposal compares to, given that it is implemented in most commercial systems under different simplified versions. LRU [8] arranges blocks using a *recency stack*, in which the block that occupies the LRU position is the furthest referenced block in the past, while the one at the MRU (Most Recently Used) position is the nearest referenced block in the past.

- *Insertion sub-policy:* a new block is inserted into the *recency stack* as the MRU block, whereas all the remaining blocks are moved one step closer to the LRU position.

- *Promotion sub-policy:* a block experiencing a hit is moved to the MRU position inside the *recency stack*, moving the remaining blocks (those previously located between the block that experiences the hit and the block occupying the MRU position) one step closer to the LRU position.

- *Victimization sub-policy:* the block occupying the LRU position inside the *recency stack* is selected for eviction, under the philosophy that, due to temporal locality, it is also the block that will not be required again for the longest time.

While LRU provides good performance for workloads with high data locality, LRU penalizes performance in applications where data are only reused in the distant future. For example, LRU would deliver poor performance under a situation where the application working set is larger than the available cache size, or when a burst of references to non-temporal data discards the active working set from the cache. In both scenarios, LRU inefficiently utilizes the cache since newly inserted blocks have no temporal locality after insertion [23].

2.1.2 Not Recently Used (NRU)

Given that, due to its high cost, it is unfeasible to implement the LRU policy in systems with high associativity caches, several approximations have been proposed in the literature and are usually employed in real systems. One such algorithm is the NRU policy [24], [25].

This algorithm uses an extra bit per cache block. This bit is used for predicting the reuse behavior of the block. As such, if the block is predicted to be reused in the *immediate* future, the bit is set to 0, whereas if the block is predicted to be reused in the *distant* future, the bit is set to 1. In detail, the algorithm works as follows:

1. When a new block is inserted into the cache or a block already present in the cache experiences a hit, the corresponding NRU bit is set to 0 to indicate an *immediate* prediction.

2. When a block has to be selected for victimization, a block with the corresponding NRU bit showing a *distant* prediction is chosen. Note that if more than one block meet this criteria, a random block is selected for victimization among those with *distant* prediction.

3. If there is no block with the NRU bit set to 1, the NRU bit of all blocks is shifted and one random candidate is selected for eviction.

The NRU policy may provide a limited performance in some cases, especially when used in the LLC. For example, an access pattern where the blocks will not be referenced again (*streaming* access pattern) could pollute the cache under this policy, assigning these blocks a wrong *immediate* prediction when they are inserted, and evicting other blocks that will be requested in the near future. In fact, an LRU policy would also be harmful in this scenario. Therefore, based on this and other similar observations, other replacement policies have been proposed in the last years for the LLC, where, as explained before, temporal locality exploitation is not the main objective. In the following subsections, some of these policies (the ones more closely related to our work) are described in detail.

2.1.3 Probabilistic escape LIFO (peLIFO)

The peLIFO policy [26] builds on a LIFO (Last In First Out) replacement policy [27], in which, making use of a *fill stack*, the last block entering the cache is the candidate for replacement. With this scheme, some blocks will remain at the bottom part of the stack, being able to exploit *long-term* reuses. In peLIFO the bottom part of the *fill stack* is reserved for *long-term* reuses as well. However, unlike LIFO, peLIFO selects dynamically intermediate stack positions (called Escape Points) for replacement, that guarantee that *short-term* reuses are also fulfilled.

As most applications go through different stages along execution, these Escape Points must be recalculated periodically. Specifically, at the end of every stage, three Escape Points are selected for the next period, based on a set of statistics collected during the previous stage. In every replacement, one among the three Escape Points is dynamically selected via a Set Dueling mechanism [28], according to hit rate information. Notably, if none of these three Escape Points demonstrates a satisfactory behavior, the LRU block is chosen. Figure 2.1 illustrates the difference between LIFO and peLIFO. Notably, in the situation shown in the figure where the block b_9 must be accommodated in the cache, in the case of the LIFO policy the block b_1 (the last entered block) would be evicted while in the case of a peLIFO policy the b_3, b_5 and b_7 blocks as well as the LRU block would be the candidates, being the set-dueling mechanism in charge of determining the final victim.

2.1.4 Static, Bimodal and Dynamic Re-Reference Interval Prediction (SRRIP, BRRIP and DRRIP, respectively)

These techniques were proposed in [23], and currently constitute the state-of-the-art cache replacement policies. Therefore, most new proposed approaches compare to this algorithm as well as to the classic LRU policy. In addition, our best proposed algorithms in terms of reduction of writes to main memory (detailed in Chapter 3) are based on these policies. For these reasons, in this section we will explain SRRIP, BRRIP and DRRIP in great detail.

The recency stack is thought of as a Re-Reference Interval Prediction (RRIP) Stack that represents the order in which blocks are predicted to be re-referenced

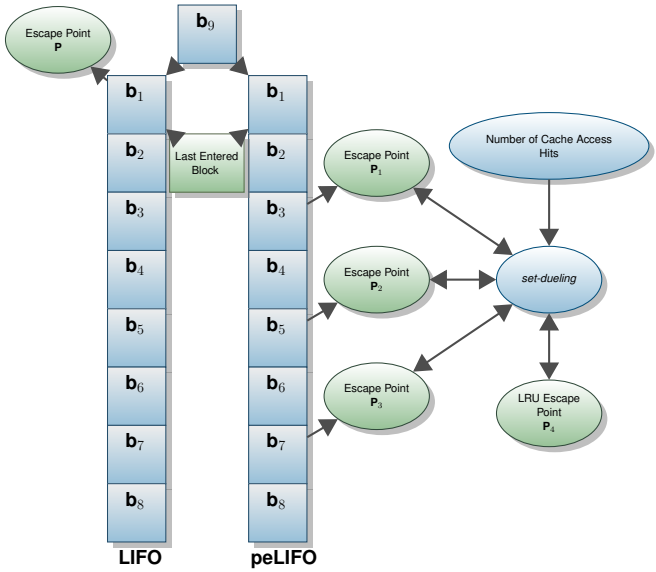


Figure 2.1: LIFO (left) vs peLIFO (right). LIFO has just one fixed escape point (the top of the stack), whereas peLIFO selects dynamically 3 escape points (plus a fourth fixed escape point, the LRU block).

(note that the LRU recency stack could also be viewed this way). The block at the head of the RRIP Stack is predicted as *near-immediate* (i.e. the block will be re-referenced sometime soon) while the block at the tail of the RRIP Stack is predicted as *distant* (i.e. the block will be re-referenced in the distant future).

For implementing the RRIP Stack, each block has an associated *replacement state* that represents the prediction of how far in the future it will be referenced again (Re-Reference Interval Prediction, or RRIP). This *replacement state* is codified with M bits, that represent 2^M different RRPVs (Re-Reference Prediction Values). A block with RRPV=0 is predicted to be referenced again soon (*near-immediate* RRIP), whereas a block with RRPV=2^M-1 is predicted to be referenced again far in the future (*distant* RRIP).[1]

Based on this *replacement state* information, the *insertion*, *promotion* and *victimization* sub-policies of SRRIP (Static RRIP) operate as follows:

[1]Note that in the LRU policy different blocks have always different *replacement states*, but in the RRIP approaches different blocks could have the same *replacement state*.

- *Insertion sub-policy:* on cache fills SRRIP assigns to the new block an intermediate prediction *replacement state* of RRPV=2^M-2 (denoted as *long* RRIP).
- *Promotion sub-policy:* on the re-reference of a block, there are two different options: HP (Hit Priority) that sets the RRPV of the block to zero, and FP (Frequency Priority) that decrements it by one. Figure 2.2 illustrates, in a set of 4-way associative cache, an example of both promotion schemes for the block b_1 using M=2.
- *Victimization sub-policy:* for eviction, SRRIP selects one block with a *distant* RRIP (RRPV=2^M-1) [2]. If there is not such a block, SRRIP increments the RRPV of all the blocks in the cache set (there is one RRIP stack per cache set in the case of an associative cache) and repeats the search. Figure 2.3 illustrates the victimization process when using M=2.

In some cases, for example when the re-reference interval of all the blocks is larger than the available cache size, SRRIP utilizes the cache inefficiently. In such scenarios, SRRIP generates cache *thrashing* and results in no cache hits at all [23]. To avoid this situation, the authors propose BRRIP (Bimodal RRIP), that modifies the *insertion sub-policy* of SRRIP as follows:

- *Insertion sub-policy of BRRIP:* it inserts majority of cache blocks with a *distant* RRIP (RRPV=2^M-1) and infrequently inserts new blocks with a *long* RRIP (RRPV=2^M-2).

This policy helps to preserve some of the working set in cache, improving performance under the scenario described in the previous paragraph. However, for non-thrashing access patterns, always using BRRIP can significantly hurt cache performance. In order to be robust across all kind of cache access patterns, the authors also propose a third policy, called DRRIP (Dynamic RRIP), which follows an *insertion sub-policy* that combines both SRRIP and BRRIP:

- *Insertion sub-policy of DRRIP:* it includes a Set Dueling mechanism [28] that identifies which insertion policy among SRRIP and BRRIP –based on

[2] The victim selection policy breaks ties by always starting the victim search from a fixed location (see example of Figure 2.5).

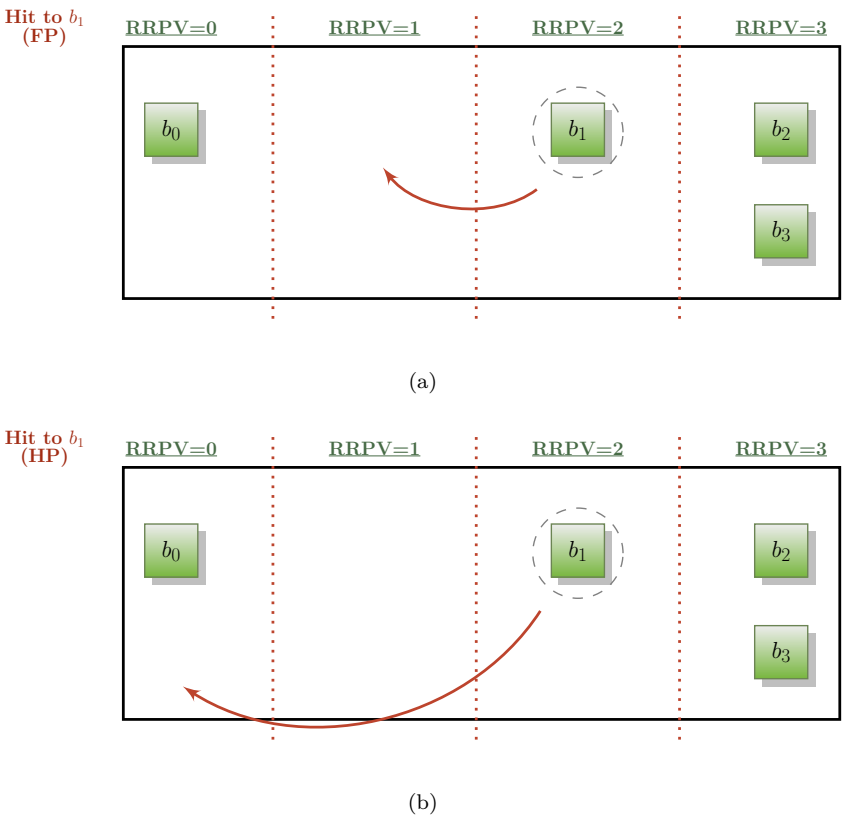


Figure 2.2: SRRIP promotion schemes: (a) SRRIP-FP, (b) SRRIP-HP.

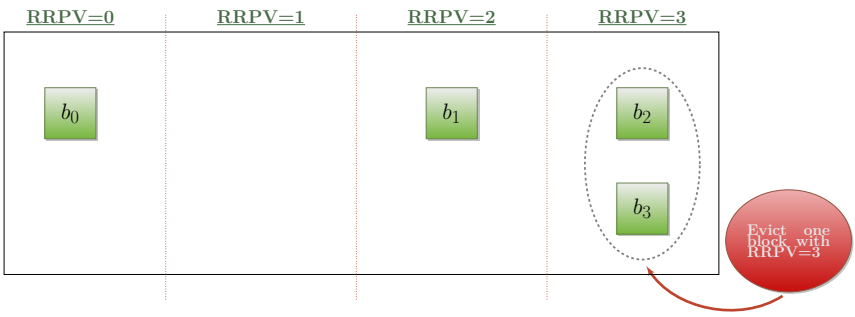


Figure 2.3: SRRIP Victimization.

the current miss rates reported– is best suited for the application (Figure 2.4).

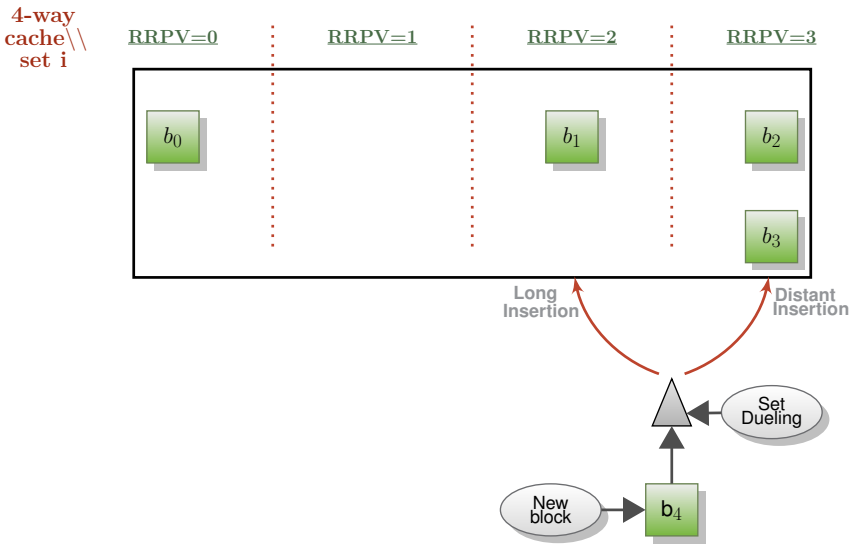


Figure 2.4: DRRIP Insertion.

To illustrate the operation of various of the policies explained so far (specifically, Bélády, LRU, NRU and SRRIP-HP), Figure 2.5 shows the behavior of these policies –in a 4-way associative cache– for the access pattern shown in equation 2.1. This access pattern starts with several consecutive requests to the block a_1 (n times, in this example we suppose $n = 2$) followed by a sequence of references to blocks from c_1 to c_5, which constitutes a *streaming* pattern since they are not referenced again, and finally block a_1 is referenced again n times (twice in this example). Initially the cache is assumed to be empty. Note that in the case of NRU and SRRIP-HP policies, each block has a subscript that indicates the value of the NRU bit and the RRPV respectively.

$$\{a_1^n, (c_1, c_2, c_3, c_4, c_5), a_1^n\} \qquad \text{with } n = 2 \tag{2.1}$$

According to Figure 2.5, SRRIP-HP is the policy exhibiting a behavior closer to the ideal one (Bélády). In the first seven references (two access to the a_1 block and the five references from c_1 to c_5), all the four evaluated policies behave in the same fashion: six compulsory misses since the six different referenced blocks must be brought to the cache on their first references, and one hit (the second reference to block a_1). After these first seven references, both LRU and NRU experiences a

Ref	Bélády	LRU	NRU	SRRIP-HP
a_1	a_1	*MRU* *LRU* a_1	$a_1{}_0$	$a_1{}_2$
a_1	a_1	*MRU* *LRU* a_1	$a_1{}_0$	$a_1{}_0$
c_1	a_1 c_1	*MRU* *LRU* c_1 a_1	$a_1{}_0$ $c_1{}_0$	$a_1{}_0$ $c_1{}_2$
c_2	a_1 c_1 c_2	*MRU* *LRU* c_2 c_1 a_1	$a_1{}_0$ $c_1{}_0$ $c_2{}_0$	$a_1{}_0$ $c_1{}_2$ $c_2{}_2$
c_3	a_1 c_1 c_2 c_3	*MRU* *LRU* c_3 c_2 c_1 a_1	$a_1{}_0$ $c_1{}_0$ $c_2{}_0$ $c_3{}_0$	$a_1{}_0$ $c_1{}_2$ $c_2{}_2$ $c_3{}_2$
c_4	a_1 c_4 c_2 c_3	*MRU* *LRU* c_4 c_3 c_2 c_1	$c_4{}_0$ $c_1{}_1$ $c_2{}_1$ $c_3{}_1$	$a_1{}_1$ $c_4{}_2$ $c_2{}_3$ $c_3{}_3$
c_5	a_1 c_4 c_5 c_3	*MRU* *LRU* c_5 c_4 c_3 c_2	$c_4{}_0$ $c_5{}_0$ $c_2{}_1$ $c_3{}_1$	$a_1{}_1$ $c_4{}_2$ $c_5{}_2$ $c_3{}_3$
a_1	a_1 c_4 c_5 c_3	*MRU* *LRU* a_1 c_5 c_4 c_3	$c_4{}_0$ $c_5{}_0$ $a_1{}_0$ $c_3{}_1$	$a_1{}_0$ $c_4{}_2$ $c_5{}_2$ $c_3{}_3$
a_1	a_1 c_4 c_5 c_3	*MRU* *LRU* a_1 c_5 c_4 c_3	$c_4{}_0$ $c_5{}_0$ $a_1{}_0$ $c_3{}_1$	$a_1{}_0$ $c_4{}_2$ $c_5{}_2$ $c_3{}_3$

Figure 2.5: Behavior of LRU, NRU and SRRIP-HP policies in a 4-way cache for the access pattern of equation 2.1.

miss upon the next access to block a_1 (after the reference to block c_5). However, SRRIP-HP (and also Bélády) is able to experience a hit in this case, so that after the seven first access of the pattern, only SRRIP-HP and Bélády are able to report hits for the two subsequent accesses to block a_1. Note that in NRU and SRRIP-HP policies, the victim selection policy breaks ties by always starting the victim search from a fixed location (the left in this example). Please also note that in the case of the Bélády policy, since when a c_i block must be evicted all these blocks exhibit the same *replacement state* (given that neither of which will be referenced again), we just select the victim block randomly.

2.1.5 Signature-based Hit Predictor (SHiP)

This proposal [29] is not a policy per se, but a predictor for the insertion of new blocks that can be used in conjunction with any other policy. SHiP associates each cache reference with a distinct signature, learns dynamically the re-reference interval of each one, and employs this information in the *insertion sub-policy*. The proposal also incorporates a Signature History Counter Table (SHCT) of saturating counters to learn the re-reference behavior of each signature. Every time a block is re-referenced (hit), the corresponding counter is incremented. If a block is never re-referenced during its residency in cache, the corresponding counter in the SHCT is decremented when the block leaves the cache. The authors evaluate SHiP in conjunction with an SRRIP replacement policy: when a new block enters the cache, SHiP assigns it a *distant* RRPV if the associated counter is zero; otherwise a *long* RRPV is assigned.

2.1.6 Probabilistic Replacement Policy (PRP)

The Pobabilistic Replacement policy [30] is based on the argument that lower cache levels have already filtered the references with short reuse distances (temporal locality), leaving a reference stream dominated by moderate and long reuse distances. Key aspect in PRP is to victimize the block with minimum estimated probability of being used again (hit probability) instead of the block with maximum expected reuse distance (Denning et al. [31] define the maximum expected reuse distance as $1/\lambda_i$, where λ_i is the stationary probability of accessing to a block i).

PRP adds some additional metadata to cache lines for determining the hit probability: for each line L, it stores the frequency $N_L(t)$ with which the reuse distance t is observed, and the last access timestamp (M_L). Also, the count of accesses to each set of the LLC, denoted as M, is stored. PRP tracks the evicted lines so that upon eviction, $N_L(t)$ values are stored in a separate DRAM area, and fetched when the line enters the cache again.

To minimize space, PRP uses a logarithmic spacing of reuse-distance histogram bins focused on the range where the hit rate varies with the reuse distance. They group reuse distances ($N_L(t)$) into H histogram bins (H = 6 is used). Bin 0 records

reuse distances in the interval [1, W], where W is the number of ways of the cache. Bins from i=1 to H-2 record reuse distances that fall in the interval [${W_\alpha}^{i-1}$, ${W_\alpha}^{i}$], where α is a constant ($\alpha = 2$ is used). The last bin (i = H-1) records reuse distances in the range [${W_\alpha}^{H-2}$, ∞[.

For determining the hit probability of line L (${P_L}^{hit}$), PRP uses the following formula:

$$ {P_L}^{hit} = \frac{\sum_{t<T_L} N_L(t) P^{hit}(t)}{\sum_{t<T_L} N_L(t)} \tag{2.2} $$

where T_L is the age of line L (calculated as $M - M_L$), $N_L(t)$ is the frequency of occurrence of reuse distance t for line L, and $P^{hit}(t)$ is the probability that a line of reuse distance t will hit in the cache. This latter distribution is not dependent on the line (they use a fixed cache distribution, which is the average hit rate of the optimal policy in the selected reuse bins using the training input set).

To select a victim, an array of hit probability calculators computes ${P_L}^{hit}$ for each candidate line L_i using the corresponding age T_{L_i} and the reuse distribution $N_{L_i}(t)$. The candidate with the lowest probability of hit is evicted. The reuse distribution for the incoming line is initialized with the reuse profile $N_L(t)$ that was stored alongside the page table translations in the TLB and brought to the LLC alongside the memory request that initiated the LLC access.

2.2 Architecting PCM for main memory

Phase Change Memory (PCM) is a promising candidate to substitute DRAM as the technology used for implementing main memory [2], [32], [33], [34], [35], [36], [37], since it provides several benefits such as the following: its non-volatile nature allows significant leakage power savings, its integration level is higher than that of DRAM and it is capable of storing multiple bits in one cell. Table 2.1 shows the key features of a 16MB main memory implemented with DRAM and PCM 90 nm technology[3] according to CACTI 6.5 [38] and NVSim [39]. As shown, the

[3]We employ an old transistor technology and a reduced main memory size for the comparison due to current NVSim constraints. However, the goal here is just to provide an insightful and illustrative idea about the differences between both technologies.

PCM implementation exhibits smaller die footprint and better efficiency in read operations than a DRAM main memory. More importantly, compared to DRAM, a PCM main memory reduces static power consumption in more than one order of magnitude. Nevertheless, PCM also exhibits two major drawbacks, both related with write operations: the cost of performing a write to a PCM cell is higher than in a DRAM cell (in terms of both latency and energy) and it also suffers from an endurance issue, as a PCM cell degrades on every write and thus it can only be written for a limited amount of cycles before breaking. In this section, we summarize several architectural studies that seek to mitigate these drawbacks of PCM in order to make it viable as a main memory technology candidate.

Parameter	DRAM	PCM	Ratio DRAM/PCM
Area (mm^2)	53.61	13.79	3.89
Read Latency (ns)	20.74	20.15	1.02
Write Latency* (ns)	20.74	151.3/42.58	0.14/0.49
Read Energy (nJ)	3.44	0.23	14,95
Write Energy (nJ)	3.37	6.57	0.51
Leakage Power (mW)	388.57	25.72	15.11

Table 2.1: Area, latency and energy consumption for 90 nm DRAM and PCM 16MB main memories (*note that for write latency in PCM the different Set/Reset times are shown).

2.2.1 Write-aware cache replacement policies

As explained in the previous section, cache replacement algorithms are usually designed only for performance improvement. Nevertheless, the high latency and energy involved in write operations to PCM cells, as well as the limited endurance of this technology, have added a new objective to these algorithms in the last-level cache: reducing the amount of writes generated to main memory. As such, several cache replacement policies have appeared in the last years aimed to address this goal. Next we recap some of the most outstanding proposals.

2.2.1.1 CLean-Preferred victim selection (CLP)

The CLean-Preferred victim selection policy [40] aims to maintain dirty blocks (blocks that have been modified with respect to the memory copy) in the cache

to increase the probability that writes were coalesced. This policy implements a modified LRU that gives preference to clean blocks when choosing a victim, since they do not imply writing to main memory as the memory copy is already updated. The authors propose a family of clean-preferred replacement policies, called *N-Chance*. The N parameter reflects how much preference is given to clean blocks. The algorithm selects as victim the LRU clean block among the N least recently used ones. If such a block does not exist, the LRU block is used. Given that this policy is based on a conventional LRU, its implementation complexity would be the same (or even higher) as that of LRU. Other CLP approaches based on efficient implementations of LRU (such as Tree-LRU) would also be possible, but they would come at the expense of some performance degradation. Figure 2.6 shows an example of the CLP behavior for different N-Chance values in a 8-way set within an associative cache. In this figure an extra bit is allocated per each block inside the cache (b_0, b_1, ..., b_7), referred to as dirty bit (DB), in order to indicate if the block has been modified (dirty block, DB=1) or not (clean block, DB=0). When N=4, CLP looks for a clean block among the 4 least recently used ones (b_7, b_6, b_5 and b_4). As this search is not successful, the LRU block (b_7) is evicted. Note that analogous situations occur when N is lower than 4. Conversely, if N=5, the search for a clean block is extended to the block b_3, which is a clean block and therefore it is selected as the victim in this case (and also if N>5).

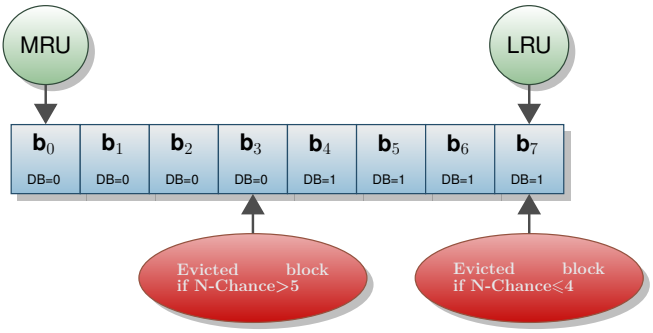


Figure 2.6: CLP algorithm behavior for different N-Chance values.

2.2.1.2 Read-Write Aware (RWA) and Improved RWA (I-RWA)

These two policies [41] are based on SRRIP-HP and they both employ RRPV values of $log_2 assoc$ bits per cache block, being *assoc* the associativity of the cache

(specifically, the authors employ an 8-way associative LLC). RWA modifies neither the *victimization* nor the *promotion* sub-policy with respect to SRRIP-HP, but the *insertion* sub-policy is modified as follows: when a read/store misses in the LLC, the RRPV of the inserted block is set to *assoc*-2; when a writeback to the LLC misses, the RRPV of the inserted block is set to 0 (recall that SRRIP-HP treats writebacks and read/store misses the same, setting the RRPV of the inserted block to 2^M-2 in both cases, being M the number of bits used to codify the *replacement state*). Conversely, I-RWA distinguishes between single and multiple-use dirty lines, trying to protect multiple-use lines. The SRRIP-HP *victimization* sub-policy is not modified, but the *insertion* sub-policy is changed so that any read/store miss sets the RRPV of the filled block to *assoc*-2 while a writeback miss sets it to *assoc*-3. Besides, the *promotion* sub-policy is changed so that a read/store hit sets the RRPV of the block to a medium value (3) –instead of 0 as done in SRRIP-HP– while a writeback hit sets it to 0. Both techniques pursue the same goal of keeping dirty blocks inside the cache in order to decrease the amount of writes to the main memory that they incur when evicted.

2.2.1.3 Adaptive and Combined Wear-out-Aware Replacement algorithms (AC-WAR)

Adaptive and Combined Wear-Out-Aware Replacement Algorithms (AC-WAR) [42] aims to reduce the number of bit-flips in the writes to main memory by redesigning the cache replacement policy of the different levels of the cache hierarchy. These proposals are based on several observations. First, the authors experiment with a cache where the cache block is divided into several sub-blocks, allocating a dirty bit per sub-block. Based on their evaluation, they determine that there is a high correlation between dirty sub-blocks and bit-flips. Second, according to their experiments, they determine that, in a 3-level cache hierarchy, the LLC is more sensitive to cache replacement policy modifications than L2. Based on these observations, they propose a different cache replacement policy for each level.

For the n-way set-associative L2 cache, they propose to evict the Least Modified block First (LMF), but avoiding evicting the w (with $w<n$) Most Recently Used (MRU) blocks in the set, i.e. the LRU policy is used in part of the set. The value of w can be changed dynamically, using a set-dueling mechanism, for adapting it to the different program phases.

As for the LLC, when a block is evicted it must also be invalidated in the private cache levels since an inclusive policy is employed. In addition to the dirty bit per sub-block, they include an *offset value* which determines if the block is present or not in any private cache. Based on this information, the cache replacement policy for the LLC first evicts non-shared blocks using the LMF policy, and, when non-shared blocks exist in this level, the LMF is applied over shared blocks.

2.2.1.4 Writeback-aware cache partitioning

Zhou et al. [43] also propose to reduce the number of LLC-to-memory writebacks at the LLC controller level. Specifically, they propose a writeback-aware cache partitioning (WCP) and a write-queue balancing (WQP) replacement policy, both for a multi-core environment. WQP is a replacement policy, based on [44], aimed to distribute the writebacks among the available write-queues. The key idea is to reduce the delays due to LLC writebacks by avoiding an overuse of the write-queues. When a block must be victimized, the WQP algorithm starts the search for a clean block or a block mapped to a light used write-queue from the LRU position to the MRU one. The WCP approach employs a set of counters to estimate the hit rate and the avoidable writebacks, so that the cache space allocation for the different cores is dynamically determined. For a N-way associative cache, WCP adds N hit counters and N avoidable writeback counters per way and core. Each read hit increments the corresponding hit counter and each writeback increments the corresponding avoidable writeback counter. Based on the values of these counters, WCP tries to select a valid cache partition that maximizes a weighted sum of the number of read hits and the number of writebacks.

2.2.1.5 Least-Dirty-First (LDF)

Yoo et al. [45] propose the Least Dirty First (LDF) cache replacement policy, which works on top of any other replacement algorithm (they employ NRU and SRRIP as example of this). The authors use 4KB cache blocks in the third cache level (the LLC) and a cache line of 256B in the other two cache levels, so that the LLC cache block includes 16 cache lines which share the same tag and also the bits that determine the *replacement state*, e.g. the NRU bit and the RRPV in the case of NRU and SRRIP replacement policies respectively. Each line inside the cache block is associated with its own dirty bit, which is set when a dirty line is

evicted from L2. The key idea of LDF (similarly to LMF in [42]) is that the more dirty lines a cache block includes, the more PCM writes the block will generate when it were evicted. Thus, the policy victimizes the block with the least number of dirty lines among those blocks less likely to be referenced again (according to the *replacement state* derived from the policy the LDF works on top of).

2.2.2 Other proposals

In this section we summarize other techniques which, although not so closely related to our proposal described in Chapter 3 (in the sense that they do not address the PCM write operation constraints by proposing new cache replacement policies), they also are oriented to make the PCM technology feasible for implementing the main memory.

2.2.2.1 Eliminating redundant bit writes

Several techniques have been proposed in this area [2], [19], [20], [46]. In a conventional DRAM write access, all bits in the corresponding row must be updated. However, a great portion of these writes are redundant (i.e. many bits after the write remain unchanged). Hence, taking advantage of the fact that PCM reads are much faster and less power consuming than writes (as we earlier illustrated in Table 2.1), every write is preceded by a read and a bitwise comparison, writing only those bits differing from the original values. Figure 2.7 illustrates an example on how redundant bit writes can be eliminated. Notably, the figure reproduces the situation in which a fictitious (just 8 bits) dirty cache block is evicted from the LLC and therefore the main memory (denoted as MM in the figure) must be updated. Before writing, each bit of the LLC block is read and compared to the value stored in the corresponding location of the main memory where the write must be performed. In the example shown, only three bits of the LLC block (those marked with arrows) differ from the corresponding values in the main memory, so only the corresponding three writes are effectively performed, avoiding the other five unnecessary writes, which translates into energy savings in the main memory and also into a lower degradation of PCM cells.

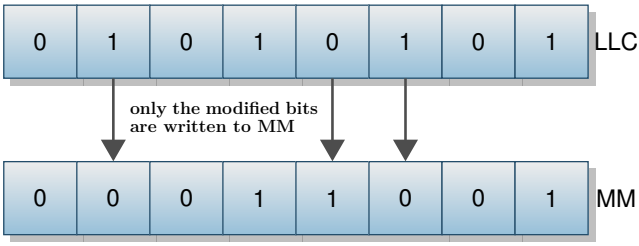


Figure 2.7: Elimination of redundant bit writes.

2.2.2.2 Flip-N-write

In [47], the authors propose a technique for reducing the amount of writes to a PCM-based memory. Similarly to the previous proposal, before performing a write to PCM the data to write are compared to the data stored in the row, but this approach also compares the bitwise inverse of the data to write to the data stored in the row, writing the one (data to write or the bitwise inverse) that involves less bit flips. For this purpose, the memory row incorporates an extra bit (flip bit, F) to indicate whether the stored data is the original one (F=0) or the opposite (F=1). Figure 2.8 recreates the same situation as that of the previous technique, where the same fictitious (just 8 bits) dirty cache block is evicted from the LLC and hence the main memory (denoted again as MM) must be updated. In the figure the original data to be written (on the top left corner) and the corresponding bitwise inverse which results from shifting each bit in the original data (top right corner) are shown. Both data (original data to write and the corresponding bitwise inverse) are compared to the stored values in the main memory. In the case of the comparison to the original data, only three bits (highlighted in red) differ from the values in the main memory, while in the case of the comparison to the bitwise inverse there are obviously five bits (also highlighted in red) differing form the MM data. As the original data involves a number of bit changes (bit-flips) not higher than N/2 (where N is the word width, 8 in our example), the original data is written to main memory and the extra bit in the memory row (called flip bit) is set to 0 to indicate that the data has not been flipped.

2.2.2.3 Wear leveling

The wear leveling techniques [2], [46] try to even out the amount of writes performed to a PCM memory among the various PCM cells. It has been observed that

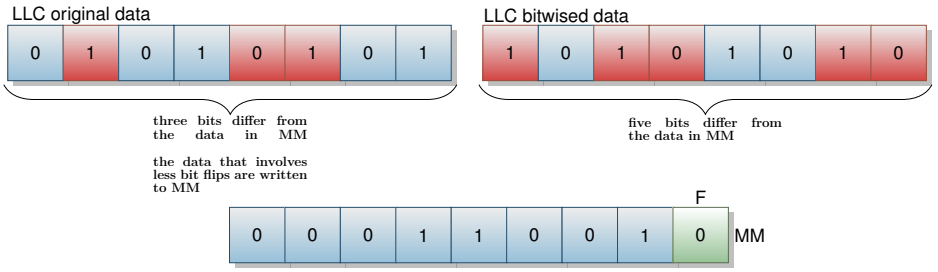


Figure 2.8: Flip-N-Write.

memory updates are highly concentrated over a reduced number of memory cells, which creates an extremely unbalanced write distribution and leads these cells to early failures while the rest remain almost untouched. A proposal to mitigate this problem is to apply periodic row shifts to even out the writes to all cells on the row. At a coarser granularity, another approach aims to periodically swap highly written memory pages with lowly written ones.

2.2.2.4 Hybrid memory

The underlying idea of this research area is to combine PCM with other technologies which are not so sensitive to the number of writes performed. Thus, many proposals combine a large PCM storage with a fast and small DRAM memory which acts both as a cache for the main memory and as an interface between the PCM main memory and the processor system [6]. In [48], this combination is studied in a Digital Signal Processor (DSP), whereas in [49] the scenario is analogous but a bit different: Chip Multiprocessors (CMPs) with local Scratch Pad Memories (SPM) and PCM as main memory.

Many of these proposals also study content management techniques for this hybrid configuration. For example, Jang et al. [50] proposed to split the DRAM buffer into two regions: the Aggresive Fetching Superblock Buffer (AFSB) and the Selective Filtering Buffer (SFB). A page fetched into the DRAM buffer is initially stored in the AFSB. Pages in the AFSB are divided into subpages, each with a size equal to the cache block. Every subpage has an access counter, and the average value of all the counters is used as a threshold for classifying subpages into high-access and low-access subpages. When a miss occurs in the DRAM buffer, the page is requested to the PCM memory and stored into the AFSB, and the

page with the lowest average access count is evicted from the AFSB but the most accessed subpages are inserted into the SFB to allow them a second chance to avoid the eviction.

In a similar direction, Parker et al. [51] proposed an algorithm for managing the DRAM buffer similarly to wear leveling approaches. They propose to divide the DRAM buffer in two layers: the first layer manages both clean and dirty blocks with an LRU policy; the second layer manages clean blocks using an LRU policy but for dirty blocks it implements a write frequency list. When the DRAM buffer is full, the LRU clean block from the second layer is evicted. If no clean blocks exist, the dirty block with the lowest write count is evicted. Finally, if all dirty blocks exhibit the same write count, the LRU policy is used for victimize a dirty block.

To conclude this body of work, Choi et al. [52] divide the DRAM buffer in three regions: the Adaptive Filtering Buffer (AFB), the Aggressive Streaming Buffer (ASB) and the Write Buffer (WB). Most pages are stored in the ASB, which seeks to improve spatial locality by prefetching a superblock (i.e. a set of pages) each time. When a page in the ASB experiences a hit, it is moved to AFB or WB depending on if the access was a read or a write respectively. The AFB exploits temporal locality by managing the recently referenced pages in an LRU list, whereas the WB is in charge of delaying the eviction of dirty pages to PCM main memory. When no space left in the WB, a page is evicted to the AFB using a FIFO replacement policy, whereas when no space left in the AFB, a page is evicted to the PCM following an LRU replacement policy. Finally, in the case of a miss in the ASB, the superblock is prefetched from the PCM main memory and a page is evicted (if necessary) using a FIFO replacement policy.

A different approach is proposed in [53], where the authors, instead of employ a DRAM as a different memory level, combine DRAM and PCM in a large and flat memory, allocating those pages that are most frequently written to the DRAM structure, and also migrating pages between the two different parts when necessary. Notably, DRAM and PCM memory addresses spaces are shown as the same main memory addresses space, and the OS differentiates between both kind of memories via physical addresses. The memory controller is in charge of translating the addresses come from the different cores and the OS periodically updates the pages mapping based on the memory controller information.

Finally, Lee et al. [54] present a new *write-aware* replacement policy, called Clock-DWF, that we include in this section since it is specifically intended for swapping efficiently memory pages between PCM and DRAM in a hybrid architecture. The authors propose a new memory management policy, based on the well-known CLOCK algorithm [55], that predicts if memory pages will receive future write references soon or not, and depending on that prediction maps the pages either to DRAM or to PCM.

2.2.2.5 Error resilience

Resistive memories are much less susceptible to transient faults than DRAM. However, the conventional correction codes used in DRAM are designed to handle transient faults with no effective lifetime limits, and these codes applied to resistive memories would wear out faster than the cells they are designed to repair [56]. That is why multiple solutions have been proposed for detecting and correcting errors in PCM memories. The goal of these techniques is to provide a fast and durable working memory with low hardware complexity, focusing on solving the reliability problem mostly in hardware and hiding it from software layers as much as possible. Next we recap some of the more noteworthy proposals in this field:

- Dynamically replicated memory [57]: In this technique, faulty pages (those where an inner block wears out) are paired for creating a healthy one, with the only restriction for pairing a couple of pages that there was no faulty bytes in the same position.
- Error-correcting pointers [58]: The authors propose to perform the error management at line level (instead of at page level) by using error-correction metadata storage to point to worn bits and replace them. Specifically, each line has associated n pointers (they can tolerate up to n bits of hard faults), which allow to encode the location of the failed cells and also the corresponding right values.
- Alternate Data Retry [59] (ADR) and Stuck-At-Fault Error Recovery [60] (SAFER): ADR is a technique useful for 1 bit faults (recall that when a failure occurs, the cell can still be read but it is stuck at some value). ADR inspects the stuck-at fault by reading the cell after each write, and if a stuck-at is detected the technique writes the inverse of the datum (that

unsuccessfully tried to wrote previously), and an extra bit indicates that the datum stored has been inverted. The SAFER proposal extends the ADR approach to multi-bit faults. It divides faulty blocks into different partitions containing only one failed bit each, and then the ADR technique is applied to the various partitions. Again, extra bits are needed to indicate if data are inverted or not (as in ADR) and also in this case to indicate the partition size.

- Fine-grained Remapping with Error-Correcting Code and Embedded-Pointers [61] (FREE-p): Given that if a page contains a faulty line it should be discarded and therefore the entire memory will wear out so quickly, the FREE-p proposal employs a fine-grained remapping so that the faulty lines from a page are remapped to working lines from another page. Also, as when a line fails not all the bits are damaged but most of them remain healthy, FREE-p uses these working bits from a faulty line to store the address of the line where it remaps. Thus, the page can still be used normally. FREE-p just adds an extra bit to each line, indicating if the data stored correspond to effective data or to an address for a redirection.

- Pay-as-you-go [62] (PAYG): This technique allocates one error correction entry per line to correct 1-bit fault. In order to correct lines with more faults, PAYG uses a pool of global correction entries to provide additional entries for this kind of blocks. The main difference with the previous proposals lies in that this mechanism works at hardware level and hence it is OS-independent.

- Coset coding [63]: The authors provide multiple encodings for data words, and propose to employ, when data must be written, the more convenient encoding in order to minimize wear and to tolerate errors.

- Zombie memory [64]: Similarly to FREE-p, Zombie maps faulty lines to working lines, but this technique uses the working lines from the discarded pages pool. The authors propose both ZombieXOR and ZombieMLC approaches. The former is well suited for single-level cells (see next subsection) and only tolerates stuck-at faults. The latter is designed specifically for multilevel cells in order to tolerate both stuck-at faults and drift[4].

[4]The resistance drift is the phenomenon that makes the resistance of a PCM cell increases over time. Such drifting was not a problem in single-level PCM cells because the rate of resistance drift is proportional to the initial resistance of the cell and is nearly-zero for the crystalline state. However, researchers found that the resistance of cells at the the intermediate states written in

2.2.2.6 SLC and MLC

PCM cells have so far been shown to store just one of two possible values, that is, a single bit. This is the traditional kind of memory cells, called *single-level cell* (SLC). However, a single PCM cell can be put in four (or more) distinct physical states [65], [66], while sustaining reasonable latency and write endurance limitations. An especially designed programming algorithm allows the PCM cell to be put in intermediate states in terms of resistivity, by exposing the cell to a set of accurately designed electrical pulses. Such a design is called *multilevel cell* (MLC), and, in the case of four different physical states, two bits are stored in a single MLC, effectively doubling the storage capacity of the memory at the same area cost [67]. Therefore, in this sense MLCs clearly constitute a further step in the PCM chances of becoming a feasible technology for the main memory.

However, it is worth noting that in order to put the PCM cell in the correct state when it is used as an MLC, it is often necessary to use iterative writes [68], [69]. This technique stems from the fact that different PCM cells have different responses to electrical pulses, and this physical behavior is unpredictable. It is therefore impossible to program a PCM cell to a desired state with a single electrical programming pulse without some probability of errors.

2.2.2.7 Data compression

In order to address the endurance problem of PCM by maintaining the effective storage capacity as memory cells fail, various compression techniques at the memory controller level have been proposed. Pekhimenko et al. [70] propose a new approach to compress pages, which they called Linearly Compressed Pages (LCP). The key idea of LCP is to compress all of the cache lines within a given page to the same size. Doing so simplifies the computation of the physical address of the cache line, because the page offset is simply the product of the index of the cache line and the compressed cache line size (i.e. it can be calculated using a simple shift operation). Based on this idea, a target compressed cache line size is determined for each page. Cache lines that cannot be compressed to the target size of the corresponding page are called exceptions. All exceptions, along with the metadata required to locate them, are stored separately in the same compressed

the multilevel PCM cells can cross the state boundary and lead to undesirable errors due to state changes.

page. If a page would require more space in the compressed form than in the uncompressed form, then this page is not compressed. The page table is extended to indicate in which form the page is stored.

In the same direction, Dgien et al. [71] propose a compression-based architecture integrated into the NVM module, which relies on a compression-decompression engine (CDE) and a data comparator that work together to reduce the number of bit writes that occur during each write access to memory. This abstracts all data manipulations from the memory controller on the processor, allowing the processor to communicate seamlessly with the NVM module. The proposed CDE implements the frequent pattern compression (FPC) algorithm [72] (which uses a static pattern table that is capable of matching a wide range of values without the need of an application profiling or runtime modifications) to compress the incoming data. Notably, during a write operation, the new data bits are compared with the currently stored bits to only write the modified bits. On a read access, the data value is decompressed if it was stored in compressed form. Further, the additional space saved by compression is used to perform wear-leveling, such that the compressed data value is written to opposite sides of the word in the NVM array. This helps in achieving wear-leveling by uniformly spreading the writes to NVM. The write-minimization and wear-leveling achieved by this technique leads to improved lifetime and reduction in write latency/energy.

2.2.2.8 Quantifying wasted write energy in the memory hierarchy

Shelor et al. [73] characterize the amount and type of wasted writes through the memory hierarchy and quantify the potential energy savings that can be obtained from removing them. Their results reveal that a significant amount of wasted writes occur at every level of the memory hierarchy and that a significant part (around 15%) of the total memory subsystem energy could be saved by eliminating all these wasted writes. Bock et al. [74] introduce the concept of useless writebacks, which occur when a dirty cache line that belongs to a dead memory region is evicted from the cache. Such writebacks could safely be avoided to improve endurance and energy consumption. Also, they develop algorithms to measure the number of useless writebacks to PCM for three different types of memory regions (heap, global and stack) and they determine that a technique based on useless writebacks is a promising candidate for reducing energy consumption and improving endurance

of a PCM-based main memory.

2.3 Architecting STT-RAM for the last-level cache

As stated in Chapter 1, various emerging technologies are currently considered to replace SRAM as the building-block for LLCs, being STT-RAM the best placed to overcome SRAM constraints, such as energy consumption and low integration capacity.

The key aspects of employing STT-RAM instead of SRAM as the last-level cache technology are illustrated in Table 2.2, that recaps some important features of a 1MB LLC for both kind of implementations. These data corresponds to caches implemented with 32 nm technology according to CACTI 6.5 [38] and NVSim [39]. As shown, an STT-RAM cache exhibits smaller die footprint and better efficiency in read operation than an SRAM cache. More importantly, an STT-RAM cache consumes more than an order of magnitude less static power compared to SRAM. Conversely, the STT-RAM cache exhibits a significant drawback that needs to be mitigated: a poor write efficiency both in terms of latency and energy consumption.

Parameter	SRAM	STT-RAM	Ratio SRAM/STT-RAM
Area (mm^2)	12.14	0.62	19.71
Read Latency (ns)	4.51	3.37	1.34
Write Latency (ns)	4.51	14.71	0.3
Read Energy (nJ)	0.29	0.23	1.26
Write Energy (nJ)	0.29	0.34	0.87
Leakage Power (mW)	215.57	12.78	16.87

Table 2.2: Area, latency and energy consumption for 32 nm SRAM and STT-RAM 1MB caches.

It is worth noting that some authors have even proposed the use of STT-RAM for main memory. Thus, Kultursay et al. [75] evaluate the behavior of an STT-RAM-based main memory and show that the performance and energy can be significantly improved by using partial write and row buffer write bypass. Notably, this proposal achieves energy savings in the main memory compared to employing DRAM technology. Although there are some other proposals to use STT-RAM as main memory, this is not the usual scenario. Thus, in the next subsections

we recap several works where STT-RAM is used in cache memories, proposing techniques for improving the efficiency of an STT-RAM-based LLC. We classify these proposals in two broad categories: those mainly focused on reducing the number of writes performed to the STT-RAM LLC in order to decrease its energy consumption and those mainly oriented to improve the system performance.

2.3.1 Reducing the amount of writes to an STT-RAM LLC

In this subsection, we start by describing in detail a recent approach for reducing the amount of writes to an STT-RAM-based LLC, which constitutes the most similar approach to our proposed techniques in this area (described in Chapter 4), and therefore constitutes the baseline to which we will compare our mechanisms: Ang et al. [76] propose *Dead Write Prediction Assisted STT-RAM Cache Architecture* (DASCA) to predict and bypass dead writes (writes to data in last-level caches not referenced again during the lifetime of corresponding cache blocks) for write energy reduction. In this work dead writes are classified into three categories: dead-on-arrival fills, dead-value fills and closing writes, as a theoretical model for redundant write elimination. On top of that they also present a dead write predictor based on a state-of-the-art dead block predictor [77]. Thus, DASCA bypasses a write operation to the LLC only if it is predicted not to incur extra cache misses. This proposal employs a PC-based predictor that correlates dead blocks with addresses of memory instructions (signatures). Notably, it samples a few cache sets and keeps track of PC information only for those sets. Predictions are made via a predictor table, made up of saturating counters similar to those used in a bimodal branch predictor, being the counters indexed by the signatures stored in the sampler entries. Different signatures are used depending on the kind of dead write predicted according to the aforementioned dead writes classification.

Chang et al. [78] analyze the cache coherence protocols impact on the number of writes to a LLC based on STT-RAM, showing that the protocols with a owned state (MOESI and MOSI) reduce the number of writes to the LLC.

Zhang et al. [79] propose *Statistics based cache Bypassing method for Asymetric-access Caches* (SBAC), an approach conceived for a cache hierarchy formed by an SRAM L1 cache and STT-RAM-based L2 and L3 caches. The main goal is to reduce the cache energy consumption bypassing some L2 write operations. The bypassing technique employed makes use of statistics of data locality from the

whole cache instead of a single cache block signature. In a more recent work, Kim et al. propose [80] Inclusive Bypass Tag Cache (IBTC), where the SBAC approach is extended to an inclusive memory hierarchy environment.

Wang et al. [81] propose an obstruction-aware cache management policy called OAP. OAP periodically monitors the STT-RAM cache in a multi-core system in order to detect LLC-obstruction processes (those processes that they do not only experience a performance loss themselves, but they also negatively affect the performance of other processes running simultaneously on the system) and also manage the cache accesses from different processes, so that when an LLC-obstruction is detected the memory references corresponding to the conflicting process are forwarded to the next cache level or main memory as appropriate.

Rasquinha et al. [82] propose two techniques to reduce the number of writes to a last-level (L2) STT-RAM cache and also save energy. The first one adds a small cache between L1 and L2 –called write-cache (WC)– which is mutually exclusive with L2 and stores only the dirty lines evicted from L1. On a cache access, both L2 and WC are accessed in parallel. The write misses are allocated to WC and the load misses are allocated to L2. WC reduces the number of L2 writes by absorbing most of the L1 writebacks.

Yazdanshenas et al. [83] propose a coding scheme for STT-RAM last-level cache based on the concept of value locality. They reduce switching probability in cache by swapping common patterns with limited weight codes to make writes less often as well as more uniform.

Jung et al. [84] rely on the observation that, on average, a large fraction of bytes and words written to the L2 cache are only zero-valued data. Based on this, this technique adds additional "all-zero-dat" flags in the tag arrays at the granularity of a single byte and a single word. Before any cache write, the data value is checked. If the all-zero bytes or words are detected, the corresponding flags are set and only the non-zero bytes or words are written. During a cache read operation, only the non-zero bytes or words are read and then the actual data are constructed by combining the information from the all-zero flags.

Park et al. [85] logically divides the STT-RAM cache line into multiple partial lines. In L1 cache, a history bit is kept for each partial line to track which partial lines have changed. Using this information, when a dirty L1 block is written to

last-level cache, only those partial lines which have been changed are written.

Mao et al. [86] propose techniques for mitigating the write pressure caused due to prefetching in an STT-RAM-based LLC. One of these techniques prioritizes different types of LLC requests such as load, store, prefetch, or writeback, etc., based on their criticality. The critical requests are assigned a high priority and hence, they are served earlier. In multi-core systems, the excessive requests generated from a cache-intensive program may block those generated from a cache-unintensive program which may lead to its starvation. To address this, they propose another technique which prioritizes the requests from a cache-unintensive program, so that they are served promptly.

2.3.2 Improving performance of an STT-RAM LLC

Jog et al. [87] propose a cache revive technique to calculate retention time. Some cache blocks retain data even after completion of retention time. The retention time is chosen so that it will minimize the number of unrefreshed cache blocks.

Guo et al. [88] propose the use of STT-RAM to design combinational logic, register files and on-chip storage (I/D L1 caches, TLBs and L2 cache). Also, to hide the write latency of STT-RAM, they propose sub-bank buffering which allows the writes to complete locally within each sub-bank, while the reads from other locations within the array can complete unobstructed. They show that by carefully designing the pipeline, the STT-RAM based design can significantly reduce the leakage power, while also maintaining the performance level close to the CMOS design.

Sun et al. [89] propose an STT-RAM cache design for lower level caches where different cache ways are designed with different retention periods. For example, in a 16-way cache, way 0 is designed with a fast STT-RAM design with low retention period and the remaining 15 ways are designed with a slow STT-RAM design which has higher retention period. Their technique uses hardware to detect whether a block is read or write-intensive. The write-intensive blocks are primarily allocated to way 0, while the read intensive blocks are allocated to the other ways. Also, to avoid refreshing dying blocks in way 0, their technique uses data migration to move such blocks to banks with higher retention period.

Sun et al. [90] propose a write-buffer design to address the long write latency of last-level (L2) STT-RAM cache. The L2 may receive a request from both L1 and the write buffer. Since read latency of STT-RAM is smaller than the write latency and also reads are performance-critical, the buffer uses a read-preemptive management policy, which ensures that a read request receives higher priority than a write request. The authors also propose a hybrid SRAM and STT-RAM cache design which aims to move the most write-intensive blocks to SRAM.

3

LLC replacement policies for improving PCM endurance

The current trend of increasing the number of cores in a single chip allows various threads or applications to execute simultaneously, which increases the demand on the main memory system to retain the working set of all the concurrently executing streams. This leads to the requirement of a larger main memory capacity in order to maintain the expected performance growth. However, the increase in memory size makes its leakage current to grow proportionally, and as a result, its energy consumption has become a major portion of the overall energy consumption in the system. Moreover, although DRAM has been the prevalent building block for main memories during many years, scaling constraints have been observed when DRAM is used with small feature sizes. Consequently, current research is focused on exploring new technologies for designing alternative memory systems in response to these energy and scaling constraints observed in DRAM technology. Among these technologies, as was introduced in Section 1.3.2, Phase Change Memory (PCM) is clearly the prime contender.

PCM is a low-cost and non-volatile memory technology that almost removes the static power consumption and provides higher density and therefore much higher capacity within the same budget than DRAM. Nevertheless, several obstacles restrict the adoption of PCM as main memory for the next computer generation: long write access latency, high write power and limited endurance.

In this chapter, we deal with the PCM endurance problem at the cache controller level by focusing on the LLC replacement policy. Conventional policies make their replacement decisions with the only objective of increasing the hit rate in the cache. Our goal is to redesign these policies so that they report a satisfactory trade-off between memory lifetime and performance.

The rest of the chapter is organized as follows: Section 3.1 presents the algorithms proposed to increase the lifetime of PCM-based systems. Sections 3.2 and 3.3 detail the experimental framework used and the results, respectively. Finally, Section 3.4 concludes.

3.1 Proposed policies

In this section we present our cache replacement policies proposals aimed to reduce the amount of writes to a PCM main memory and thus to increase the PCM lifetime. Subsection 3.1.1 shows the principles our policies are based on and Subsection 3.1.2 explains our proposals.

3.1.1 Rationale

As explained in the previous chapters, it becomes essential to restrict the number of writes performed to main memory in order to improve the PCM lifetime. Writes can reach memory via two channels: from the upper level in the hierarchy (the disk) for loading the code or data, or from the lower level in the hierarchy (the LLC) for updating those blocks that have been modified by the processor. This work focuses on the second type of writes.

The observed LLC-to-memory write pattern is very dependent on the memory updating policy employed: if a *write-through* policy is used, every time a block is modified in the LLC it is also updated in the main memory level; conversely, if a *write-back* policy is employed, a block is updated in memory only when it leaves the LLC in a dirty state. In this proposal we will employ the latter policy, which is the most frequently used one in real systems due to its better tolerance to high memory latency and lower bandwidth requirements than the former policy. Moreover, the *write-back* policy implies a significantly lower amount of writes to memory, which makes it even more adequate in our scenario.

In a *write-back* policy, a block can be modified several times in cache before eviction. Our aim is to coalesce as many modifications to a block as possible in the LLC. For this purpose we focus on the cache replacement policy, which mainly determines the lifetime of the blocks inside the cache. When a cache miss implies the eviction of a block, the replacement algorithm decides which block must

be replaced/victimized. Conventional policies –conceived for systems with several cache levels backed up by a DRAM main memory– make their decisions with the only goal of increasing the cache hit rate and hence system performance. However, in our PCM scenario, the replacement policy for the LLC should not only aim to improve performance but also to reduce the number of writes to memory that it implies.

For developing such write-aware replacement policy, we should pay attention to the following general considerations:

1. First, a clean block leaving the LLC can simply be discarded, generating no writeback at all. Therefore, a *write-aware* replacement policy should give a higher priority to the eviction of clean blocks over dirty ones.

2. Second, among dirty blocks, we should distinguish whether they will be modified again in the future or not. A block that will be modified later again should stay in cache in order to merge future modifications with the previous ones into a single writeback. Conversely, a block that will never be modified again will not be able to reduce the amount of writebacks even though staying in the LLC. Consequently, our policy should give a higher priority to the eviction of the latter blocks.

3. Finally, among dirty blocks that will be modified in the future, we should consider the following two aspects:

    - First, based on Belady's conclusions [22], the evicted block should be the one that will be modified again furthest in the future, given that this is the block with the lowest probability of merging the future modifications with the preceding ones into the same writeback.
    - Second, based on the fact that a dirty block may have from only one to all its words modified, the evicted block should be the one with the least amount of modified words, since this is the block with the lowest probability of overwriting a dirty word in the next write to the block.

The first issue is quite easy to accomplish, since the dirtiness information of blocks is available in the cache. However, the two remaining points are more complex to achieve, as they require extra hardware and some knowledge about the future.

Besides, the third issue takes into account two conflicting aspects. Hence, as done in other similar situations, and in particular in conventional cache replacement policies, the solution in this case will be to build a prediction based on the previous behavior/state of the blocks.

Obviously, we have to take into account that an appropriate block replacement policy is essential to guarantee a high hit rate in the cache, and if we only pay attention to writeback reduction we can severely impact performance. Therefore, we must look for a satisfactory trade-off between performance and write reduction. Our proposed *write-aware* replacement policies for the LLC are presented below.

3.1.2 Implementation

Before delving into our *write-aware* LLC replacement policies, we first analyze the behavior of some of the conventional algorithms described in Section 2.1 (i.e. *performance-oriented* cache replacement policies), in order to inspect how they impact the amount of writes to main memory. Notably, we evaluate LRU, SRRIP, DRRIP, SHiP (in combination with SRRIP, as done in the original paper [29]), peLIFO and RANDOM[1] policies employing the SPEC CPU2006 benchmark suite [91] and the experimental framework detailed in Section 3.2. Figure 3.1 illustrates the amount of writes to memory normalized to the LRU baseline. Note that the figure shows results per benchmark as well as the geometric mean obtained.

Given that all assessed algorithms report quite poor results (between 5% and 32% more writes to memory than LRU), we propose and evaluate several modifications looking for a reduction in the amount of writes to memory without penalizing performance. Thus, we have studied many policies based on the conventional ones by modifying them in several ways. DRRIP-based policies proved to achieve the best trade-off between writeback reduction and performance. In fact, according to Figure 3.1, DRRIP ranks only second to LRU as the policy that on average delivers the lowest number of writes to main memory, although DRRIP outperforms LRU for most applications. Moreover, DRRIP, as we detail in the evaluation section, is the policy that, especially in the multi-core scenario, delivers a higher performance. Therefore, hereafter we stick to DRRIP-based policies.

[1] Although the RANDOM policy has not been described in Section 2.1, this approach, that simply relies on randomly choosing the block to victimize, also reports satisfactory performance numbers under some scenarios.

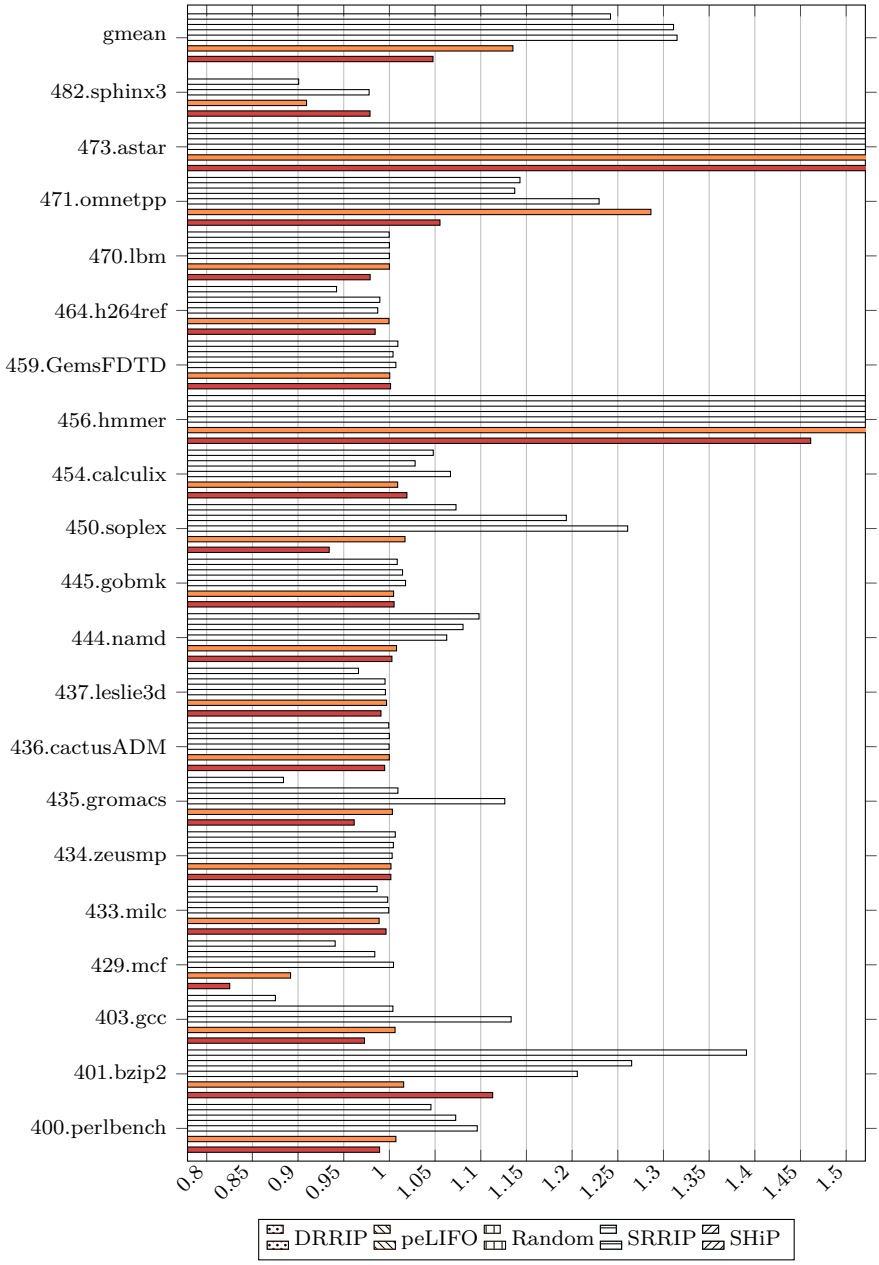


Figure 3.1: Writes to main memory normalized to LRU for performance-oriented policies: SPEC CPU2006 suite. *hmmer* and *astar* applications report numbers ranging between 7 and 16x for most policies.

Below we detail those changes to DRRIP *insertion*, *promotion* and *victimization* sub-policies that demonstrated to be efficient enough. We should highlight that all these proposals can easily stack on top of any other technique applied at the main memory level for the same or for a different purpose. Note that we always use 2-bit DRRIP (RRPV ranges between 0 and 3). Recall that a detailed description of DRRIP foundation and operation was provided in Section 2.1.4.

1. *Changes to the insertion sub-policy of DRRIP:* the only change that reveals as satisfactory enough deals with the set-dueling mechanism, which decides the policy to employ in each insertion (SRRIP or BRRIP). This mechanism makes the decision by comparing the number of misses that each policy generates in a few *dedicated sets*. Specifically, for the insertion, some cache sets always follow SRRIP while other cache sets always follow BRRIP; the remaining sets (*follower sets*) use the policy determined by the mechanism, so that the block is inserted according to the scheme reporting the lowest amount of misses at that moment. Our proposal is to change the metric employed to make the decision: instead of using the number of misses (which is reasonable for a performance-oriented policy), we compare the number of writebacks to memory that both SRRIP and BRRIP generated in the dedicated sets, inserting the block according to the policy currently exhibiting the lowest number of writebacks. We will refer to this change as SD (from Set-Dueling). Figure 3.2 illustrates the changes proposed to the set-dueling mechanism. Notably, in the figure, an incoming block b_4 must be filled into a set of 4-way associative LLC, so a set-dueling mechanism determines if the block is inserted with *distant* or *long* RRPV relying on the number of writebacks generated by SRRIP and BRRIP in the sampled sets.

2. *Changes to the promotion sub-policy of DRRIP:* we make the following three proposals:

   - PL (Promotion Low-aggressiveness): as explained before, clean blocks leaving the LLC are not harmful at all for PCM, whereas dirty blocks cause a writeback to main memory when evicted from the LLC. Consequently, we propose to promote more aggressively (i.e. to a state that makes the block to be replaced further in the future) a dirty block than a clean one. Specifically, PL promotes clean blocks using the FP option (decrementing the RRPV), while dirty blocks are promoted using the

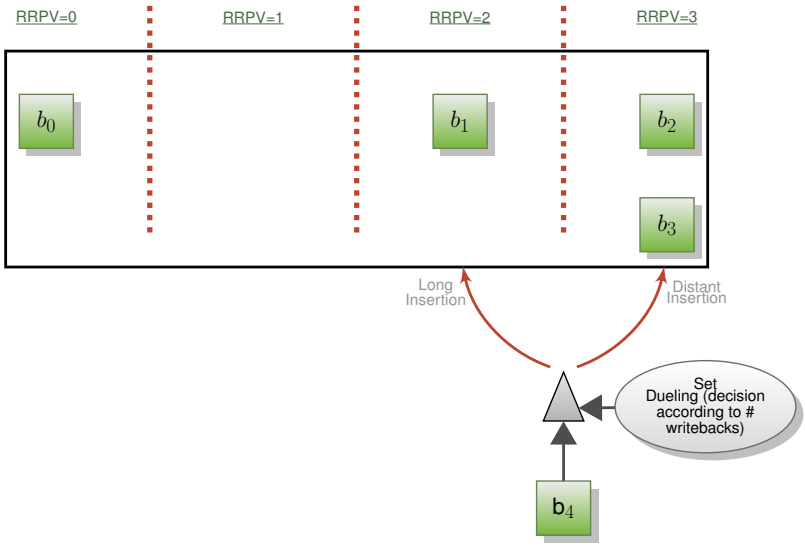


Figure 3.2: Changes to the insertion sub-policy of DRRIP.

HP option (setting the RRPV to 0). Note that a more aggressive approach would be not to promote clean blocks at all; however, according to our experiments, this would lead to an excessive performance drop, due to not exploiting whatsoever the temporal locality of clean blocks. As shown in Figure 3.3, where we have included the dirty bit of the block next to the block identifier, clean blocks (dirty bit = 0) are promoted with an FP policy, whereas as shown in Figure 3.4, dirty blocks (dirty bit = 1) are promoted with an HP policy.

- PM (Promotion Medium-aggressiveness) and PH (Promotion High- aggressiveness): based on both the second and third general considerations previously detailed in Section 3.1.1 and the temporal locality principle, we propose to promote a block that experiences a write hit[2] with a very aggressive policy under the intuition that, if a block is written, it will probably be written again soon. Notably, in both PM and PH, blocks experiencing a write hit always promote under the HP option. However, regardless of the dirtiness state of the block, whereas PM promotes a block that experiences a read hit using the FP option in order to not

[2]Note that, unlike L1, LLC reads and writes do not correspond directly to program loads and stores respectively. In our system, the LLC is written to only when a dirty block is evicted from L2, which can take place both due to a load or a store in the processor. Conversely, LLC is read when L2 misses, which again can occur both due to a load or a store.

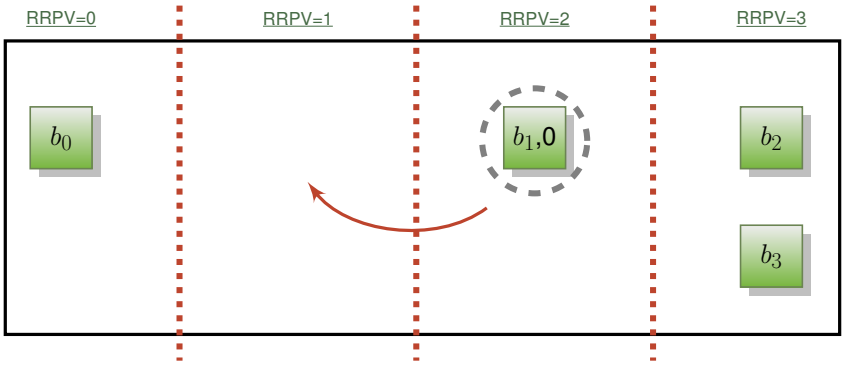


Figure 3.3: Changes to the promotion sub-policy of DRRIP. Promotion Low-aggressiveness: the clean block b_1 is promoted with FP policy.

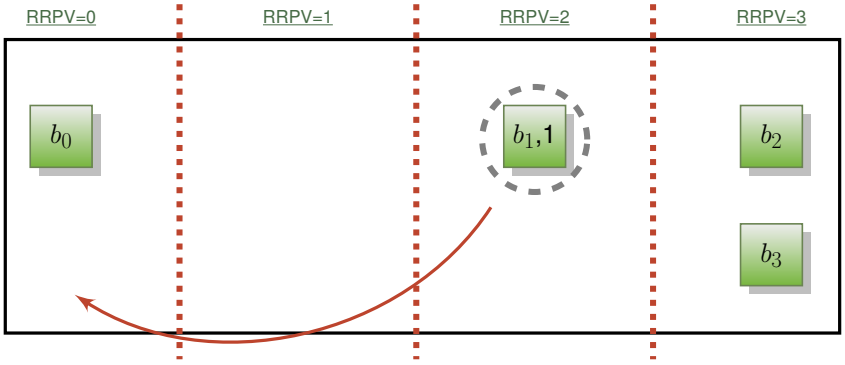


Figure 3.4: Changes to the promotion sub-policy of DRRIP. Promotion Low-aggressiveness: the dirty block b_1 are promoted with HP policy.

impact performance in excess, PH does not promote a block at all under a read hit, for giving even more protection to writes at the cost of some performance degradation. Thus, in PM, as shown in Figure 3.5, read hits promote the block with an FP policy whereas, as shown in Figure 3.7, write hits promote the block with an HP policy. In PH, as shown in Figure 3.6, read hits keep the RRPV unchanged, whereas as shown in Figure 3.7, write hits promote the block with an HP policy.

3. *Changes to the victimization sub-policy of DRRIP:* as for block eviction, the

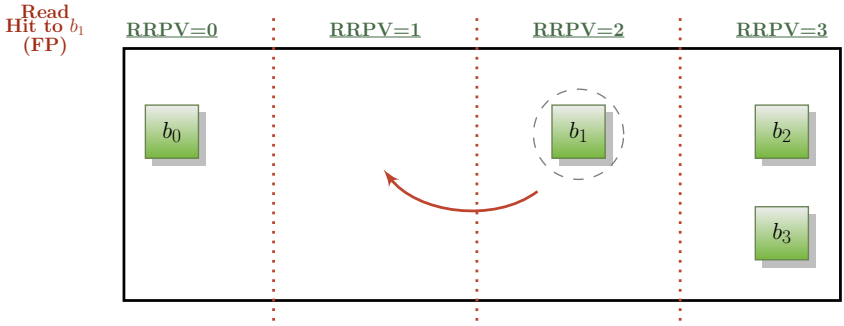


Figure 3.5: Changes to the promotion sub-policy of DRRIP. Promotion Medium-aggressiveness: a read to block b_1 promotes it with FP policy.

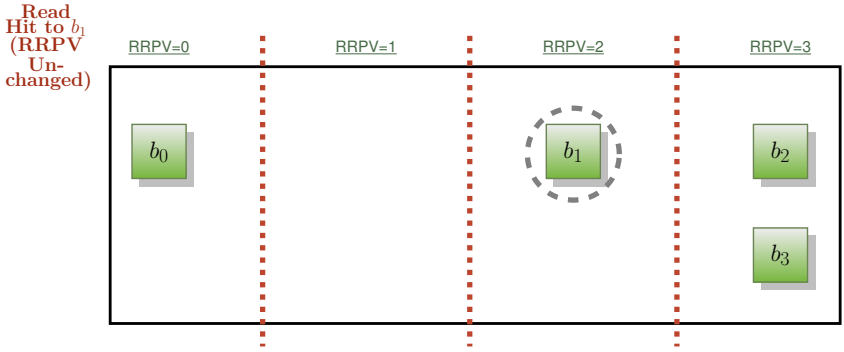


Figure 3.6: Changes to the promotion sub-policy of DRRIP. Promotion High-aggressiveness: a read to block b_1 does not promotes it at all.

only extra issue (with respect to DRRIP) to which we will pay attention is the dirtiness of the blocks. According to the first general consideration previously described in Section 3.1.1, we propose the following three modifications:

- VL (Victimization Low-aggressiveness): at first, only clean blocks with a *distant* RRIP are considered for replacement. Note that, as done in original DRRIP, our three victimization sub-policies break ties by always starting the victim search from a fixed location (the left in our studies). In the event that the policy is unable to find such a block, a dirty block with a *distant* RRIP is victimized. As in the original DRRIP policy, if no blocks with a *distant* RRIP exist, VL increments the RRPV of all the blocks and repeats the same process. For the sake

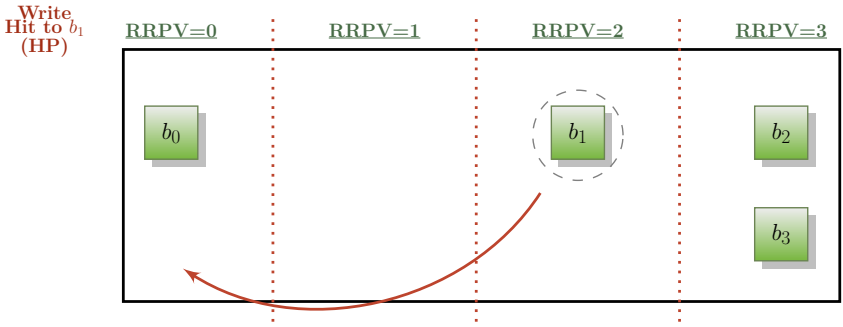


Figure 3.7: Changes to the promotion sub-policy of DRRIP. Promotion Medium and High-aggressiveness, a write to block b_1 promotes it with HP policy.

of clarity, Figure 3.8 illustrates the flow chart of VL policy, whereas Figure 3.9 shows an example of the VL operation where the block b_2 (the clean block with a distant RRPV) is evicted.

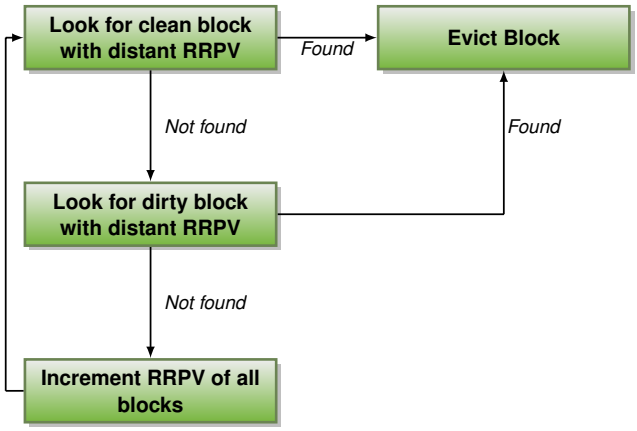


Figure 3.8: VL Flow Chart.

We next recap the working operation flow of VL to determine the victim block:

1. Find the first clean block with RRPV=3.
2. If not found, find the first dirty block with RPRPV=3.
3. If not found, the RRPV of all the blocks in the set is incremented and the whole process starts again.

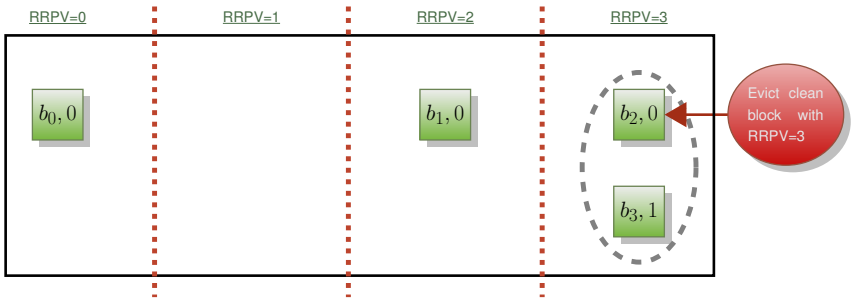


Figure 3.9: Changes to the victimization sub-policy of DRRIP. Victimization Low-aggressiveness: among the blocks with *distant* RRIP a clean block (b_2) is evicted.

- VM (Victimization Medium-aggressiveness): we can view this policy as a two-stage process. At the first stage, only clean blocks are considered for replacement, so that the clean block with the highest RRPV is victimized. Notably, when no clean block with a *distant* RRIP is found, the policy augments the RRPV of clean blocks. In the event that VM is unable to find clean blocks in this first stage, a second stage takes place, in which a conventional search is performed, including all the blocks in the set. Figure 3.10 shows the flow chart of VM policy, and Figure 3.11 illustrates an example of the VM operation, where the block b_1 (the clean block with the highest RRPV) is evicted after incrementing its RRPV.

  Next we summarize the VM operation to search for the victim block:

  1. Find the first clean block with RRPV=3.
  2. If not found, the RRPV of clean blocks in the set is incremented and the process starts again.
  3. If no clean blocks exist, find the first dirty block with a *distant* RRIP.
  4. If not found, the RRPV of all the blocks in the set is incremented and the process restarts again from the (c) point.

- VH (Victimization High-aggressiveness): this is also a two-stage process. At the first stage, as in the case of VM, only clean blocks are considered for replacement, so that the clean block with the highest RRPV is victimized. Note that, as opposed to the VM policy, the RRPV of clean blocks remains unchanged in this first stage, whatever

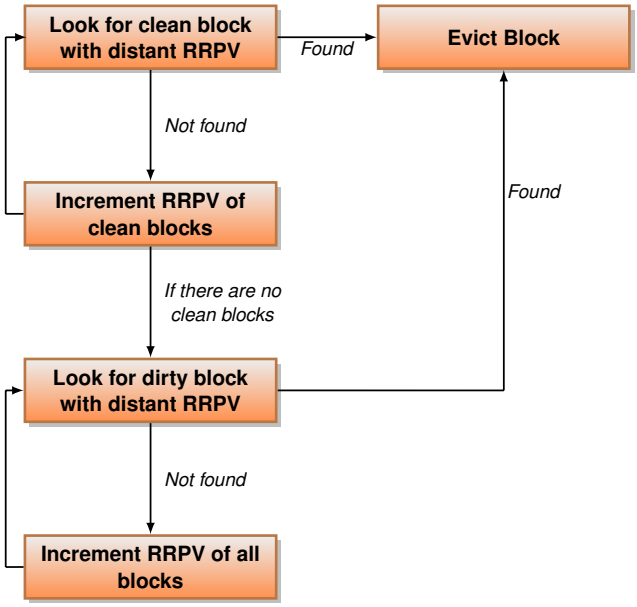


Figure 3.10: VM Flow Chart.

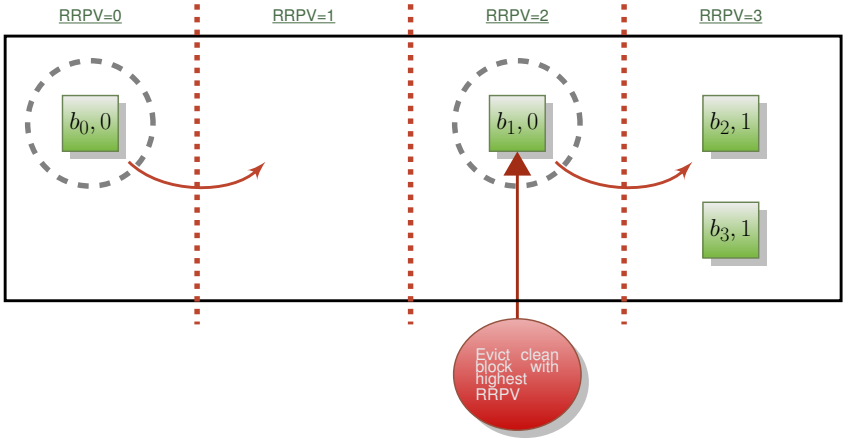


Figure 3.11: Changes to the victimization sub-policy of DRRIP. Victimization Medium-aggressiveness: since there is not a clean block with *distant* RRIP the RRPV of all clean block is increased.

the combination of RRPVs may be. When no clean blocks exist in the set, a second stage is carried out, in which a conventional search is performed, considering all the blocks in the set. Figure 3.12 illustrates the flow chart of VH policy, and Figure 3.13 shows an example of the VH

operation where the block b_1 (the clean block with the highest RRPV) is evicted without incrementing its RRPV.

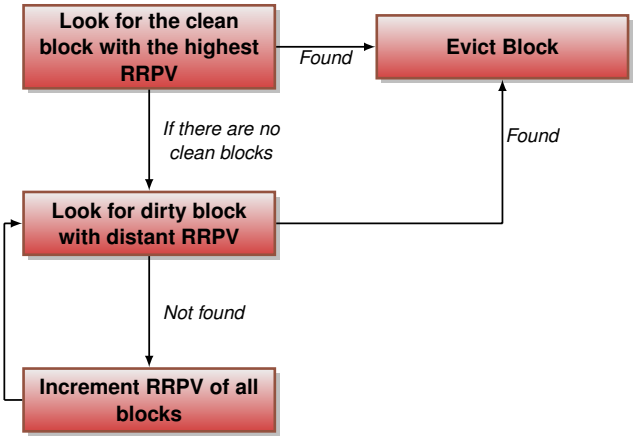


Figure 3.12: VH Flow Chart.

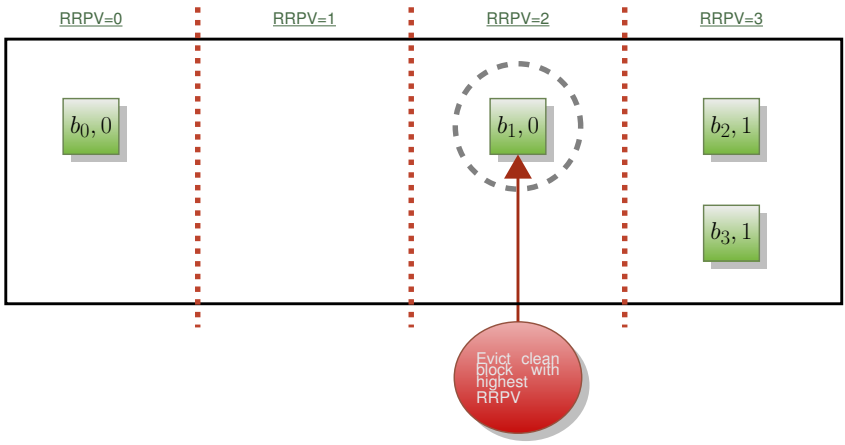


Figure 3.13: Changes to the victimization sub-policy of DRRIP. Victimization High-aggressiveness: since there is not a clean block with *distant* RRIP, the clean block with the highest RRPV is evicted.

Summarizing the operation of VH to select the victim:

1. Find the first clean block with RRPV=3.
2. If not found, find the first clean block with RRPV=2.

3. If not found, find the first clean block with RRPV=1.
4. If not found, find the first clean block with RRPV=0.
5. If no clean blocks exist, find the first dirty block with a *distant* RRIP.
6. If not found, the RRPV of all the blocks in the set is incremented and the process restarts again from the (e) point.

As a result of all the possible combinations of the 7 proposed sub-policies (SD, PL, PM, PH, VL, VM and VH), we introduce a full set of LLC replacement policies, denoted by the the names of the sub-policies they consist of. Thus, for instance, the policy referred to as PL-VH-SD, combines the PL change proposed to the promotion sub-policy of DRRIP, the VH change proposed to the victimization sub-policy of DRRIP and the only change proposed to the original insertion sub-policy of DRRIP (denoted as SD). Although we have evaluated all of them, in the evaluation section, for the sake of simplicity, we only show results for those policies providing a satisfactory trade-off among PCM lifetime and performance (see Section 3.3.1.1).

In order to further clarify our proposals, Figure 3.14 shows an illustrative example of the operation of three of our policies (PL-VL-SD, PM-VM-SD, PH-VH-SD) and original DRRIP for a sequence of five references to the last-level cache. For simplicity, we make several assumptions: first, we consider a 4-ways LLC in which all accesses are mapped to the same set. Second, a write-back updating policy is used. Third, in the sub-insertion policy, the SD mechanism is supposed to always select SRRIP (i.e. blocks inserted with RRPV=2). Finally, HP is employed as the promotion sub-policy for DRRIP. Note also that the second row shows the initial state for each way of the involved set, and rows from the fourth to the eighth show the contents of the set after each access.

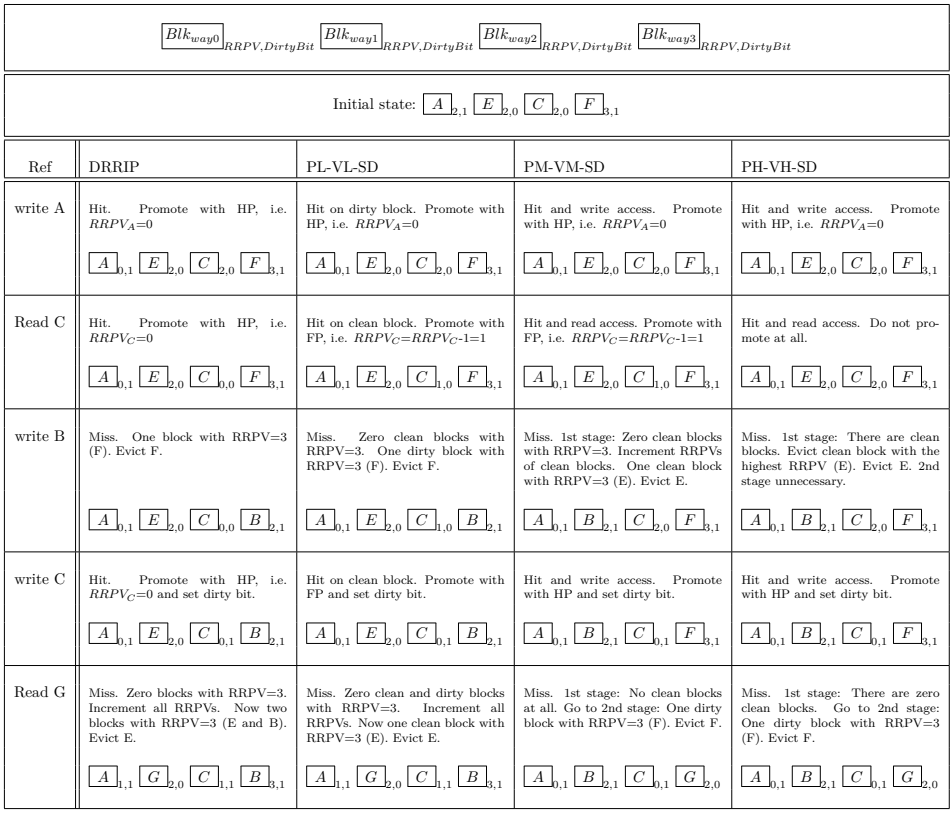


Figure 3.14: Example of different policies operation: original DRRIP and three of our proposals.

3.2 Experimental framework

This section is divided into four parts. In Section 3.2.1 we describe the main aspects related with the architectural simulator employed, the memory hierarchy simulated and the different benchmarks used for the the evaluation of our proposals. In Section 3.2.2 we detail the endurance model we use in order to translate the number of writes avoided to the PCM main memory into endurance improvement numbers. Section 3.2.3 describes a statistical simulator we employ to better visualize the lifetime extension results we obtained and finally, in Section 3.2.4, we detail the energy model we employ in order to quantify the energy savings in the memory hierarchy derived from the implementation of our proposals.

3.2.1 General aspects

For our main experiments we use *gem5* [92]. We use the classic memory model provided by this simulator and we modify it by including a new cache level (L3) and encoding the proposed cache replacement policies. We simulate both a single and a multi-core scenario, using the simulator in its Syscall Emulation mode (SE) or Full System mode (FS) respectively. For the sake of a better accuracy in both execution modes, an O3 processor type (detailed mode of simulation) was used.

The cache hierarchy models after that of an processor operating at 2GHZ, formed by three cache levels, being all the levels non-inclusive/non-exclusive. In the case of the multi-core scenario we model 2 and 4-core CMPs, with private L1 and L2 (both using a classical LRU algorithm as replacement policy), and a shared L3 (the cache level where we implement our proposed policies). Figure 3.15 illustrates the experimental system used in both single-core and multi-core scenarios.

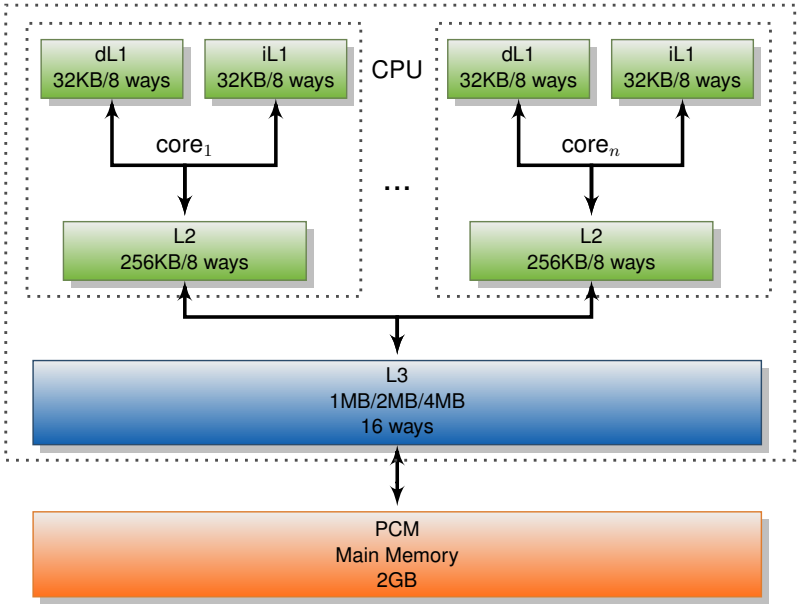


Figure 3.15: Simulated system, for all caches a 64 bytes/block is used.

For modeling the PCM main memory we use DRAMSIM2 [93]. The integration between DRAMSIM2 and *gem5* is done using the patch developed for *gem5* [94]. We adapt the read and write latencies according to the PCM target (see Table 3.4

Mixes	Applications	Mixes	Applications
MIX0	lbm, mcf, milc, soplex	MIX1	lbm, mcf, milc, GemsFDTD
MIX2	mcf, milc, soplex, GemsFDTD	MIX3	bzip2, zeusmp, cactus, leslie3d
MIX4	zeusmp, cactus, leslie3d, gobmk	MIX5	cactus, leslie3d, gobmk, calculix
MIX6	perlbench, gcc, gromacs, namd	MIX7	hmmer, h264ref, omnetpp, astar
MIX8	lbm, mcf, gromacs, sphinx3	MIX9	mcf, milc, perlbench, h264ref
MIX10	cactus, hmmer, h264ref, lbm	MIX11	mcf, cactus, hmmer, h264ref

Table 3.1: SPEC 2006 multiprogrammed mixes.

in Section 3.2.4). We consider a PCM main memory with 1 channel with 2 ranks of 8 banks each. Each bank has an 8-entry read queue and a 32-entry write queue for pending requests. When a writeback of a cache line from the LLC occurs, a write request is sent to the PCM, which is queued at a write queue. The application progresses without delay if the write queue has available entries since the writebacks are not on the critical path. Otherwise, the application stalls. The latencies and buffer sizes were selected according to the system used in [43].

Since we target both uniprocessor and multi-processor architectures, our experiments make use of the SPEC CPU2006 [91] and the PARSEC [95] benchmark suites. When using the former suite in a single-core scenario (L3 1MB size) we employ *train* inputs –we also tried with *reference* inputs, obtaining very similar results but at the expense of very long simulation times– and we simulate 1 billion instructions from the checkpoint determined using PinPoints [96]. Note that results from 9 out of 29 benchmarks are not considered in the evaluation section due to experimental framework constraints. We also report results of 12 multiprogrammed mixes using SPEC CPU2006 programs (Table 3.1) in a 4-CMP system with a 4MB L3. In this case, we fast forward 100M instructions, warm up caches for 200M instructions and then report results for 1B instructions per core. Finally, when the PARSEC benchmarks are employed, we use *large* inputs, which exhibit memory footprints much more similar to those of *train* inputs for SPEC applications than the *small* ones, and each simulation fast forwards to the predefined checkpoint at the code region of interest (ROI), warms-up for 100 million instructions, and then simulates 1 billion instructions for all threads or up to ROI completion (whichever comes first).

For selecting the multiprogrammed mixes from Table 3.1, we employ the following methodology: we execute each benchmark alone, using an L3 of 1MB and an LRU replacement policy for all cache levels, and we measure the amount of LLC-

to-memory writebacks that it generates. We then obtain for each benchmark the *writebacks to main memory per 1000 instructions* ratio (WPKI). Based on these values, we include each benchmark into the *high*, *medium* or *low* category. Specifically, the *high* category includes benchmarks with a WPKI higher than 3, the *medium* those with a WPKI satisfying $1 < \text{WPKI} < 3$ and finally, in the *low* category we include the programs with a WPKI lower than 1. Table 3.2 shows this classification. Based on this classification, and as we will further detail in Section 3.3.2.1, we build some mixes made up of applications with high WPKI, some with medium WPKI, some with low WPKI, and some combining applications from different WPKI categories.

High	Medium	Low
lbm, mcf, milc, soplex, gemsFDTD	bzip2, zeusmp, cactus, gobmk, calculix, leslie3d	perlbench, gcc, gromacs, hmmer, h264ref, omnetpp namd, astar, sphinx3

Table 3.2: Benchmark characterization according to the number of writes to main memory per Kinstruction (WPKI).

We should also highlight here that *gem5* supports both x86 and Alpha ISAs. However, given that we found restrictions when using the FS mode for x86, we only simulated the PARSEC suite compiled for the Alpha ISA. In the case of the SPEC CPU2006 suite, we run the simulations with the benchmarks compiled for both architectures, but, given that we obtained results in the same ballpark, we only report here those corresponding to the x86 architecture.

3.2.2 Endurance model

The policies proposed in Section 3.1.2 are conceived with the aim of extending the lifetime/endurance of a PCM main memory without sacrificing too much performance. For achieving such a goal, we modified conventional performance-oriented replacement policies trying to reduce the number of LLC-to-memory writebacks. Note that the lifetime extension of the device is closely related to the reduction of writes, assuming the presence of wear leveling mechanisms (both at page level and inside each cache block) to prevent hot memory locations to wear memory unevenly, as [19], [97], [98], [99], [100]. This claim holds in the reasoning that if we write to memory, e.g. half often, then each cell is worn half as much, which

translates directly into twice (1/0.5 = 2) the memory lifetime.

There is a caveat though: any conventional PCM-based memory would implement the redundant bit writes technique [2], [19], [46], based on which only those bits that have changed need to be updated. Therefore, we cannot just look at the amount of writebacks to memory we reduce, because that could artificially make our results look better than they are, but we need to take into account how many bits are affected by each writeback. Given the size of the task, we have slightly relaxed the model as follows:

- Whenever a cache block is written to main memory, we account for how many of the 64-bit words within the block have been modified. Unmodified bits do not wear off PCM cells.
- We assume that, when a word is modified, half of its bits are flipped and the other half preserves their value. This assumption is based on the fact that we use both *integer* and *FP* applications: on the one hand, control variables have a small variability, on the other hand, pointers to the heap and FP-numbers have much larger variability. Therefore, we take 0.5 as a general, broad approximation.

Based on this reasoning, we can derive the following equations, which we will employ in the evaluation section for calculating endurance results.

First, we need to account for how many bits in the block are affected by each writeback. In other words, the probability of each bit to flip, denoted as BFP (*Bit Flip Probability*). This is needed because reducing the number of writes to main memory could have the side effect of increasing the dirtiness of a block and increasing the probability of bits being flipped.

$$BFP = \frac{RMBW * BW * \sum_{n=1}^{WB} n * NWMM_n}{NWMM * BB} \tag{3.1}$$

where:

– RMBW = Ratio of Modified Bits per Word (recall that our relaxed model assumes RMBW=0.5).

- BW = Bits per Word (in our scenario, BW=64).
- WB = Words per Block (in our scenario, WB=8).
- n = Number of words modified within a written-back block (in our scenario, n can be an integer from 1 to 8).
- NWMM = Total Number of Writebacks to Main Memory.
- $NWMM_n$ = Number of Writebacks to Main Memory with n words modified.
- BB = Bits per Block (in our scenario, BB=512).

Based on the fact that memory wear is proportional to NWMM corrected with BFP, and that memory endurance is inversely proportional to memory wear, we can derive the wear reduction and endurance extension (denoted as WR and EE respectively) achieved by a *policy* with respect to *lru* when running a particular application (note that we are using the relation WB=BB/BW):

$$\frac{1}{WR_{policy/lru}} = EE_{policy/lru} = \frac{1}{\frac{BFP_{policy}}{BFP_{lru}} * \frac{NWMM_{policy}}{NWMM_{lru}}} = \frac{1}{\frac{\sum_{n=1}^{WB} \frac{n}{WB} * NWMM_{n,policy}}{\sum_{n=1}^{WB} \frac{n}{WB} * NWMM_{n,lru}}} \tag{3.2}$$

Note that this equation is perfectly coherent with the reasoning from Section 3.1 and the assumptions of our model: it weights each write to main memory with the percentage of dirty words that the written-back block contains. Note also that, due to the normalization vs LRU, the EE obtained in Equation 3.2 is independent of the constant factors we assumed above (RMBW, BB, BW and WB).

3.2.3 Memory endurance simulator

Although *gem5* yields the values required to estimate the memory endurance extensions according to the model detailed in Section 3.2.2, we also employ a statistical simulator in order to plot the available memory (*#alive_pages*) under each evaluated policy as time passes. Doing faithful, cycle accurate simulation is unfeasible due to the amount of time PCM requires to experience stuck-at faults.

Therefore, we have developed an in-house Monte Carlo simulator following the description in [58]. Table 3.3 recaps the main parameters employed in this simulator.

Each proposal is simulated by creating a number of memory pages *#starting_pages*. Each bit inside every page is created with a lifetime randomly distributed according to a Gaussian distribution $N(\mu = 10^8, \sigma = 2.0 * 10^7)$. The simulator uses as input parameters for each proposal the following (provided by the *gem5* simulator): the average BFP, the total number of writes and the average write reduction vs LRU. Initially, the wear rate w is calculated as a function of BFP and the number of pages in the system, as Equation 3.3 shows. The BFP expresses how much each bit is worn per-write, and the part involving the number of pages expresses how much extra wear alive pages have to absorb on behalf of those faulty pages that have been discarded from the system.

$$w = BFP * \frac{\#starting\_pages}{\#alive\_pages} \tag{3.3}$$

Then, according to the characterization obtained from *gem5*, we start simulating writes to the cells. At some point in time t a cell will wear out, the page containing it is discarded, the simulator updates the *#alive_pages* and the wear rate accordingly, and the simulation proceeds until all pages are discarded. As a result of this simulation, a curve is generated that shows the amount of memory available (expressed in percentage of available pages) as a function of the number of writes. In the evaluation section we will show this kind of curves when quantifying the memory endurance achieved by the different proposals evaluated.

Parameter	Value
Page size	4KB
Row size	64 Bytes
Chips per rank	8
Bit lines per chip	$x8$
Lifetime distribution	$N(\mu = 10^8, \sigma = 2.0 * 10^7)$
Pages	2000

Table 3.3: Main parameters of the memory endurance simulator.

3.2.4 Energy model

Finally, we should mention that in the evaluation section we will show results about energy consumption in the memory hierarchy, following a model that includes both dynamic and static contributions. The static component is calculated using CACTI 6.5 [38], [101], which reports a leakage number for each cache level. In the case of the main memory, as it is built with PCM technology, the static power is considered as negligible in our experimental environment based on [43]. Thus, adding the power contributions of each cache level, we obtain the total static power consumed in the memory hierarchy. Finally, we multiply that number by the execution time of the program to obtain the total static energy consumed in the execution of that application.

In the case of the dynamic component, we again use CACTI 6.5 for determining the dynamic energy consumption per access to each cache level, whereas for computing the dynamic energy consumption associated with the accesses to main memory we follow [43], employing 1J/GB and 6J/GB per PCM read and PCM write respectively (Table 3.4 includes data regarding latencies and energy consumption per memory hierarchy level). Then, the equation employed to determine the dynamic energy consumption in the memory hierarchy is:

$$\begin{aligned} \textit{Dynamic Energy} = \sum_{i=1}^{n} (&RHL_i * REL_i + WHL_i * WEL_i + \\ &(RML_i + WML_i) * (TEL_i + WEL_i)) + \\ &RPCM * REPCM + WPCM * WEPCM \end{aligned} \tag{3.4}$$

where n is the amount of cache levels, RHL_i and WHL_i denote, respectively, the number of read and write hits in cache level i, RML_i and WML_i denote, respectively, the number of read and write misses in cache level i, RPCM and WPCM correspond to the amount of reads and writes to PCM, respectively, REPCM and WEPCM denote the energy consumption per read and write to PCM respectively and finally REL_i, WEL_i and TEL_i correspond to the energy consumption of a read, a write and a tag array consult, respectively, in cache level i.

It is worth noting that in the energy consumption analysis of the next section

Level	Latencies (cycles)	Energy (Read/Write/Tag)(nJ)	Leakage (mW/bank)
L1	1	0.22/0.21/0.0017	4.02
L2	10	0.53/0.54/0.0055	11.65
L3 1MB	30	2.26/2.50/0.019	38.61
L3 2MB	30	1.98/2.17/0.035	115.60
L3 4MB	30	2.72/2.98/0.067	202.53
PCM read	100	59.61	0
PCM write	700	357.63	0

Table 3.4: Latencies and energy consumption.

we do not account for the implementation overhead of the replacement policies, which, according to the analysis from Section 3.3.3.2, constitutes a reasonable approximation. In any case, including this overhead in the energy study would be beneficial to our interests, as it will be detailed in the aforementioned section.

3.3 Evaluation

This section is divided into three parts. Sections 3.3.1 and 3.3.2 assess the effectiveness of our proposed LLC replacement policies and of other *write-aware* policies in cutting the write memory traffic and extending the memory lifetime in single-core and multi-core environments respectively. Finally we also report some additional results in Section 3.3.3. Note that, throughout this section, when providing results about writes, we refer to LLC-to-memory writebacks (recall that we do not deal with writes from the disk to memory) at a *block-level*.

3.3.1 *Write-aware* policies in a single-core scenario

In this section we deeply analyze the behavior of our proposed LLC replacement policies and other algorithms in a PCM-based system within a single-core scenario (Figure 3.15, just one CPU) with a 1MB L3 cache. First, we explore the isolated contribution of each of our proposals and, based on this evaluation, we justify the decision of reporting results from just some of our policies, which we consider as representative, along this section. Then, we report data about the number of writes to memory and endurance that each evaluated proposal involves as well as the performance delivered. Besides, being energy consumption one of the main

motivations for adopting PCM as main memory technology, we also include results about the involved energy consumption in the memory hierarchy according to the model detailed in Section 3.2.4. Finally, we expose a brief discussion about the trade-off between the memory endurance and the performance that the evaluated policies deliver.

3.3.1.1 Contribution of each proposed change

The various modifications we have proposed over original DRRIP impact the write memory traffic differently. Figure 3.16 quantifies this impact for each change isolated and also when we combined them. First, we observe that modifying the *insertion* sub-policy by changing the criteria that rules the Set Dueling mechanism (SD label) reports low reductions (around 1%) in the amount of writes to memory. Second, the impact of our *promotion* sub-policies is also limited. The low, medium and high-aggressiveness versions (PL, PM and PH respectively) only manage to cut the write traffic to memory between 0.5 and 1.5% with respect to DRRIP. Third, the *victimization* sub-policies clearly report the major benefits. Notably, VL, VM and VH are able to reduce the amount of writes generated by original DRRIP by around 10, 24 and 32% respectively. Therefore, we arrange our proposals into *high-aggressiveness*, *medium-aggressiveness* and *low-aggressiveness* categories depending on whether they include the VH, VM or VL label respectively. It is worth to note that when different sub-policies are combined, in some cases the write reduction achieved is higher than the one that would result by simply adding their isolated contributions.

Next we explain the criteria followed to choose the most representative policies across the board (highlighted in Figure 3.16 with green bars and north-west lines). First, we discarded all policies that do not reduce the write traffic to memory with respect to conventional LRU. We also ruled out most of the algorithms that achieve modest write reduction at the expense of performance drops. Finally, among those policies reporting similar results in both amount of writes to memory and performance, we pick one of them within each group. As a result, we choose just one algorithm from each class of victimization sub-policies (i.e. VH, VM and VL) combined with one of the three possible promotion sub-policies (the best performing one in each case, either PH, PM or PL) and with the SD insertion sub-policy. Notably, we choose PM-VH-SD, PM-VM-SD and PL-VL-SD.

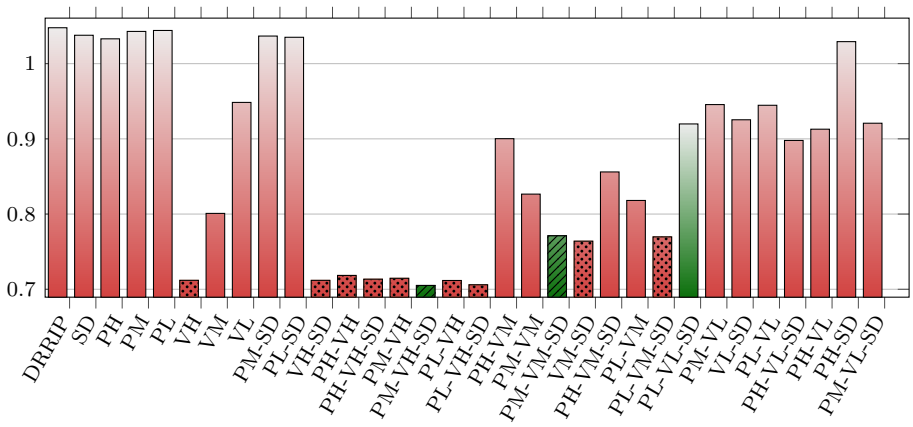


Figure 3.16: Amount of writes to main memory normalized to LRU: contribution of each proposed policy.

From now on, in all the following figures and for each evaluated policy, we report the geometric mean of the normalized metrics (write reduction, endurance, performance and energy consumption in the memory hierarchy) with respect to LRU considering both all the benchmarks (we labeled them as *All*) and only the most memory-intensive programs (we labeled them as *memory-intensive*). The rationale behind including this group of applications with high memory footprints is twofold: first, to reveal potential biased values in the *All* numbers due to programs in which the amount of writes is so low that minimal write reduction in absolute values lead to high percentage numbers, polluting the *All* number, and second, to stress the benefits derived from our techniques over those applications that, performing high amounts of writes to memory, are more harmful to PCM lifetime. In order to define this second group of applications, we sort all the programs according to the WPKI values exhibited in the baseline policy, and, starting from the benchmark with the highest WPKI, we pick applications until the accumulated WPKI of the selected programs reaches at least the 75% of the total WPKI obtained when considering all applications. In the experimental setting we employ in this section, the *memory-intensive* group is conformed by *lbm*, *mcf*, *milc* and *soplex*.

Note that in this scenario we also report the geometric mean of the evaluated metrics when all the applications except the *sphinx3* program are considered (we label this group as *All w/o sphinx3*). The reason for this is that this application

exhibits a special behavior that may lead the evaluated metrics to be somehow biased. Notably, while for the rest of applications the evaluated policies are able to reduce the amount of writes to memory by a factor ranging from 1X to 2X, for *sphinx3* many policies under evaluation are able to cut the write traffic to memory in an unusually large fashion (up to a hundredth part, 100X).

3.3.1.2 Write filtering and endurance

Figure 3.17 illustrates the number of writes to main memory generated by different proposals: from left to right we show results of some *performance-oriented* policies (DRRIP, peLIFO and SHiP), some previously proposed *write-aware* approaches (RWA, IRWA and CLP) described in Section 2.2.1, and some of our proposed policies, ordered by decreasing aggressiveness.

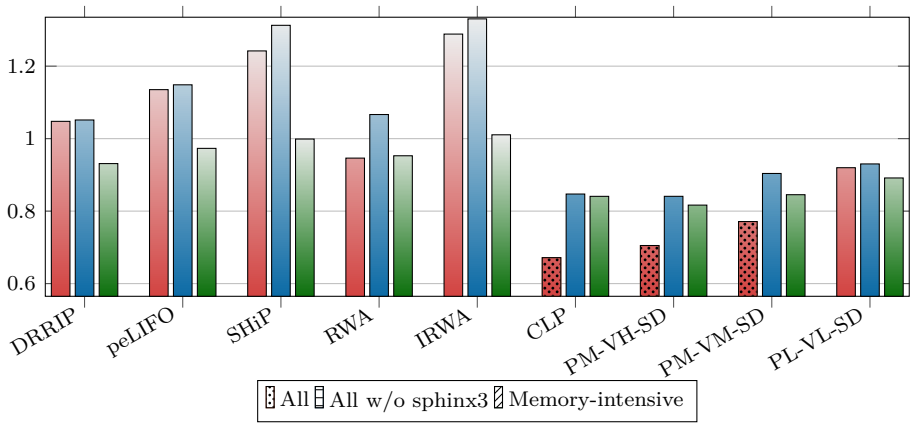


Figure 3.17: Amount of writes to main memory normalized to LRU: SPEC CPU2006 suite.

First, as shown, all our proposals significantly outperform original DRRIP. Second, they also exhibit higher ability in cutting the write traffic to main memory than other *write-aware* policies (such as RWA and IRWA) and similar ability as CLP. The rationale behind the deficient behavior of both RWA and IRWA is twofold: 1) they are based on SRRIP instead of DRRIP, which generates a higher number of writes as Figure 3.1 illustrates, and 2) they do not modify the SRRIP victimization sub-policy, that, as demonstrated in Section 3.3.1.1, constitutes the sub-policy with the highest impact. CLP reduces the amount of writes by around 33% and

15.3% when *sphinx3* is considered or not respectively, while the best-performing of our proposals (PM-VH-SD) reaches 30% and 16% respectively. Third, the three *performance-oriented* policies augment the number of writes compared to LRU. Fourth, zooming into the *memory-intensive* programs, we observe that, although the results follow a similar tendency, the differences among policies get reduced[3]. Besides, our most aggressive policy even improves CLP for these applications.

Next, according to the model detailed in Section 3.2.2, we show how the achieved reductions in writes to memory translate into extensions of the PCM lifetime for each evaluated proposal. Notably, Table 3.5 illustrates the memory lifetime improvement (percentages) with respect to LRU.

Policies / Benchs	All/All w/o sphinx3	Memory-intensive
DRRIP	-5.9/-6.3	2.0
peLIFO	-11.5/-12.2	0.9
SHiP	-20.9/-24.9	-3.7
RWA	5.4/-7.2	3.6
IRWA	-23.1/-25.1	-4.2
CLP	49.1/17.6	11.9
PM-VH-SD	41.4/19.0	16.1
PM-VM-SD	36.3/17.1	11.6
PL-VL-SD	7.4/6.5	5.8

Table 3.5: Memory lifetime improvements (percentages) with respect to LRU: SPEC CPU2006 suite.

As shown, the trends observed in the values of memory lifetime extension closely follow those observed for the write reduction numbers when considering all applications. This is due to the fact that the ratios between the BFP of each evaluated policy and that of LRU, which modulates the contribution of the write reduction factor to the memory endurance extension number as shown in Equation 3.2, are very similar across the board. Notably, the average number of dirty words per writeback when considering all applications ranges between 5.95 in SHiP and 5.62 in PM-VM-SD, exhibiting LRU a value of 5.86, which implies BFPs ratios in the range 1.02-0.95. This makes the write reduction number the major contribution to the memory endurance improvement obtained. However, those policies exhibiting a BFP value lower than LRU obtain an additional memory lifetime improvement.

[3]The percentages of write reduction exhibited by most *write-aware* policies decrease for the *memory-intensive* programs (as analyzed in Section 3.3.3.1, this occurs even for an optimal policy).

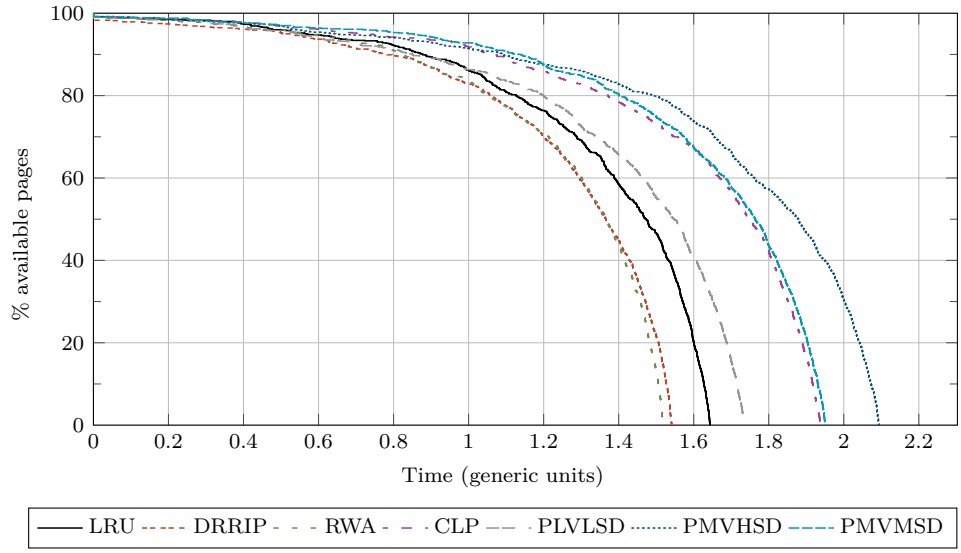
Thus, PM-VM-SD, while reducing 6% less memory writes than CLP when considering all applications except *sphinx3*, is able to almost match its endurance extension (17.1% vs 17.6%) since the geomean of the ratios between BFP and the BFP of LRU is 0.94 in our proposal while it is just 1.0 in CLP. When considering the *memory-intensive* programs, the ratios obtained between the BFPs of the evaluated policies and that of LRU are higher than 1 across the board (up to 1.06 in the case of CLP). This implies that the write reduction achieved translates into lower memory endurance extensions. Note that for these programs, the average number of dirty words per writeback is moderately lower, ranging from 3.9 in LRU to 4.17 in CLP.

In order to visually observe the memory lifetime improvements achieved, we use the memory endurance simulator detailed in Section 3.2.3, which reports the amount of memory available (expressed in percentage of available pages) as a function of the number of writes. We change the variable in the x-axis from number of writes to time (expressed in generic units). Figure 3.18 illustrates the curves obtained when considering both all the applications except *sphinx3* and the *memory-intensive* programs. We observe how in both cases our PM-VH-SD policy reports the highest amount of memory available for a given time. For instance, when considering all applications except *sphinx3*, after 1.8 billions of instructions executed, PM-VH-SD maintains around 58% of pages surviving while CLP just maintains around 42%. Note that, for the sake of clarity, we omit some proposals (peLIFO, SHiP and IRWA) that do not provide satisfactory results according to Table 3.5.

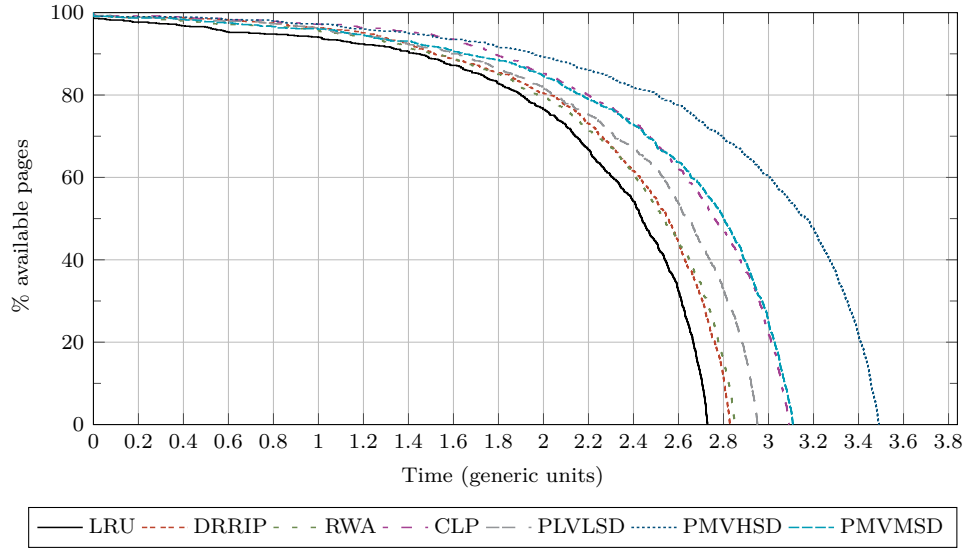
Note also that the time reached for the *memory-intensive* programs until the whole memory fails is higher than the one obtained for all the applications, since, although the amount of writes performed is higher, the average number of dirty words that exhibit the writebacks in this kind of applications is significantly lower than when considering all applications.

3.3.1.3 Performance

Although our prime goal is to extend the PCM lifetime, it is clear that a proposal meeting this requirement could not be adopted if it comes at the expense of a significant performance drop. In order to further evaluate the benefits that each policy reports, Figure 3.19 shows the performance (execution time) delivered.



(a)



(b)

Figure 3.18: Available memory vs time for SPEC CPU2006 suite: (a) All, (b) Memory Intensive.

First, we observe that most of our DRRIP-based proposals almost match the execution time of original DRRIP and even some of our algorithms –those including

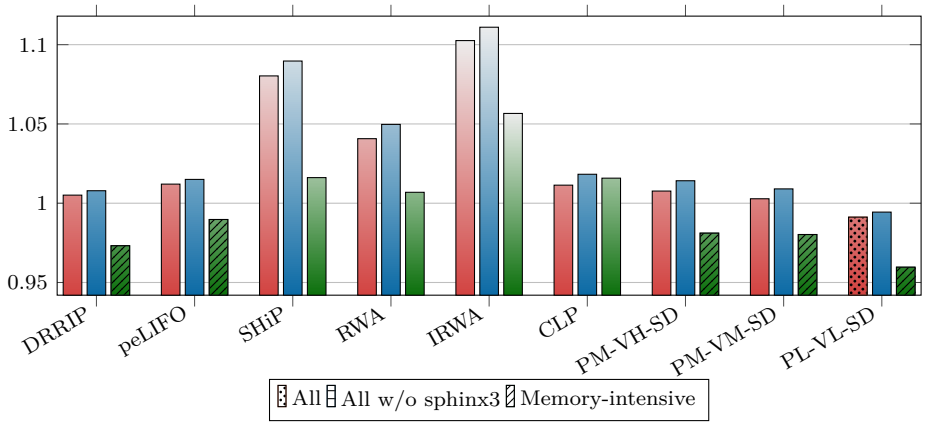


Figure 3.19: Execution Time normalized to LRU: SPEC CPU2006 suite.

a medium-low *victimization* sub-policy (VM or VL)– manage to outperform it. Second, all our proposals outperform the other *write-aware* policies (especially RWA and IRWA, which significantly degrade performance, 4.1 and 10.2% over LRU respectively).

Third, the two most aggressive policies in reducing the amount of writes to memory (CLP and PM-VH-SD) do provide a moderately satisfactory trade-off with performance since they both penalize it around 1-2%, depending if we include the *sphinx3* program or not. Intuitively, we would expect that this performance penalty was higher. Therefore, it is worth to note the key factors that may affect the performance delivered by the various *write-aware* approaches. Note that two opposing factors turn up:

1) Our proposed replacement algorithms are mainly focused on reducing the amount of dirty blocks evicted, being partially unaware of the performance impact, so it would be expected to lead to an increase on the number of read misses, hence augmenting the amount of accesses to main memory and hurting performance. However, our results reveal that for the PM-VH-SD policy –our algorithm achieving the highest reduction of writes to memory– the amount of LLC read misses in 8 out of 20 benchmarks decreases with respect to LRU. In fact, in the original DRRIP algorithm (the policy our schemes are based on) more than half of the benchmarks exhibit a lower amount of LLC read misses than LRU. The rationale behind this decrease in the amount of LLC read misses is that the temporal locality has been almost totally filtered at the LLC by the lower cache levels, thus

a policy focusing only on temporal locality exploitation, like LRU, may provide poor results at this level of the hierarchy for some applications. Overall, the total amount of read misses in L3 considering all applications (and hence the events of read queue filling) has slightly increased in our proposals compared to LRU. This makes the net contribution on performance essentially negligible for our proposals.

2) Conversely, reducing the number of writes to memory reduces the pressure over the write queues, which leads to performance improvements (as explained in Section 3.2, once the corresponding write queue is full, the application stalls when a writeback from the LLC occurs). As for the write queues filling, our results illustrate that in the LRU baseline only in 8 out of 20 programs some write queue fills up at least once. For these 8 benchmarks, PM-VH-SD is able to significantly reduce the chances of filling up write queues and therefore mostly cancel the performance drops associated with the higher LLC read misses observed in 6 of these programs. For the remaining two applications (*soplex* and *h264ref*), as the amount of LLC read misses is also lower than that of the baseline, performance improvements of 16 and 5% respectively are delivered.

Fourth, among the *performance-oriented* policies, DRRIP and peLIFO provide satisfactory results (as expected) while SHiP surprisingly exhibits the second worst number across the board (largely due to the performance drop delivered in *hmmer*, *astar*, *bzip2* and *soplex* applications).

Finally, zooming into the *memory-intensive* programs, we observe that all our policies manage to improve significantly the performance of the other *write-aware* proposals and also that of some *performance-oriented* algorithms (note that DRRIP performs especially well for this kind of applications). Notably, our most aggressive algorithm, providing very similar write reduction as CLP, manages to outperform it by more than 3%. Moreover, for these applications (and also considering all benchmarks) our low-aggressiveness approach even reports the best numbers across the board, exhibiting also the lowest amount of LLC read misses among all evaluated policies.

3.3.1.4 Memory energy consumption

As for the energy consumption on the memory hierarchy for the different evaluated policies, our experimental results reveal that our proposals outperform all

the other policies, reporting energy savings ranging between 4% (medium and low-aggressiveness algorithms) and 2% (high-aggressiveness policy) with respect to LRU. Only DRRIP and CLP are able to also report energy savings with respect to LRU. The other two *write-aware* policies significantly augment the energy consumption. Considering only the *memory-intensive* programs, the energy savings are greater across the board, being again our three proposals (and also CLP) those reporting the highest values (in the range from 9 to 8%). We should highlight that these savings mainly come from a lower number of writes to main memory, which are highly energy-consuming in the PCM technology. However, note also that for the SPEC applications the most aggressive policies in cutting the write traffic to memory are not the best ones in terms of energy consumption, since they penalize the execution time with respect to less aggressive schemes, leading the static energy to grow. Thus, the energy savings for these most aggressive policies are partially canceled.

3.3.1.5 Putting it all together

Although considering a policy as the best one depends on the particular requirements of the system and the user, here we try to extract some insights about the trade-offs reported based on the different metrics evaluated. We consider that our high-aggressiveness algorithm provides satisfactory trade-offs between the memory endurance extension and performance. Notably, it exhibits a high number in memory lifetime extension (around 19% when the *sphinx3* application is not considered) without significantly degrading performance (around 1.4% penalty), reducing also the memory energy consumption around 2%. Our medium-aggressiveness policy also reports satisfactory trade-offs, improving memory endurance in a slightly more modest fashion (17%) but maintaining the system performance largely unchanged with respect to original DRRIP and LRU, and also reporting around 4% energy savings. Our least aggressive algorithm –although it contributes less to PCM lifetime extension (around 6.5%)– exhibits the best performance and energy numbers (1% and 4% improvement respectively for all benchmarks and 4% and 8% improvement for memory intensive benchmarks). As for the other *write-aware* policies, while RWA and IRWA clearly report poor trade-offs, CLP exhibits similar numbers to our most aggressive proposal when considering all the applications except *sphinx3* (although CLP achieves a lower endurance extension and delivers lower performance), but performs significantly worse when we focus on the

memory-intensive programs (our PM-VH-SD is able to improve memory lifetime by 4% more than CLP while also exhibits a performance value more than 3% better than CLP). Finally, the *performance-oriented* policies –although exhibiting satisfactory performance numbers, except SHiP– fail in improving the memory endurance.

3.3.2 *Write-aware* policies in a multi-core scenario

In this section we extend the prior study about the behavior of different cache replacement policies to a multi-core scenario. We just move to this new and more realistic setting without any change in our proposed algorithms. We evaluate the same policies as in the single-core environment for both multiprogrammed workloads (Section 3.3.2.1) and multithreaded applications (Section 3.3.2.2). For this purpose, we measure again the number of writes to main memory, endurance, performance and energy consumption in the memory hierarchy that each policy involves, normalized to LRU. However, due to the inherent non-determinism that all simulators exhibit (especially in multi-core environments, where the number of instructions executed across different policies are not stable owing to the random interleaving among memory accesses of different threads) and for the sake of higher accuracy, we opted to employ in this scenario the geometric mean of the metrics above mentioned (except for performance where we employ the Instruction Throughput metric) divided by the total number of instructions executed. Note that, conversely, in the single-core scenario both kind of metrics match, since all the benchmarks execute the same amount of instructions (1B) in all the runs. Finally, in Section 3.3.2.3 we analyze the trade-off between memory endurance and performance that the evaluated policies exhibit.

3.3.2.1 Multiprogrammed workloads

For the evaluation of our proposals in a 4-CMP system with a 4MB shared L3 (Figure 3.15), we employ 12 mixes made up of applications from SPEC CPU2006 chosen accordingly to the WPKI categories illustrated in Table 3.2. We randomly compose 3 mixes made up of applications with high values of WPKI (mixes 0, 1 and 2, referred to as the *memory-intensive* ones), 3 made up of programs exhibiting a medium WPKI ratio (mixes 3, 4 and 5) and 2 composed by benchmarks with low WPKI values (mixes 6 and 7). We also evaluate four extra mixes merging

applications with WPKI corresponding to different categories (mixes 8 to 11). The detailed mixes are illustrated in Table 3.1. Besides, like in the single-core configuration, we report data considering both all mixes and only the *memory-intensive* workloads.

3.3.2.1.1 Write filtering and endurance Figure 3.20 shows the number of writes to memory *per instruction* that each evaluated policy performs normalized to LRU. As shown, considering all mixes, our most aggressive proposal exhibits the best behavior. Notably, PM-VH-SD achieves a write reduction of around 19%. The rest of our techniques manage to cut the write traffic to memory in the range 14-12%. The other *write-aware policies* (except CLP that reduces writes by around 16%) as well as the *performance-oriented* ones behave significantly worse. Indeed some of them even augment the amount of writes with respect to the baseline.

When we just consider the 3 *memory-intensive* mixes, our proposals with medium and low-aggressiveness are able to further increase the write reduction capability, mainly due to the numbers that they exhibit for the *mcf* application (better than those of all the other policies evaluated), which appears in these 3 mixes. Besides, our three techniques –and also SHiP and peLIFO– achieve the best numbers across the board (reductions ranging from 22% to 15%), significantly outperforming the other *write-aware* evaluated policies. Even 2 of our proposals exhibit an amount of writes more than 10% lower than that of CLP, the best-performing of the other *write-aware* techniques.

Table 3.6 illustrates the memory lifetime improvements with respect to LRU for the different evaluated policies.

Policies / Benchs	All	Memory-intensive
DRRIP	-5.0	4.7
peLIFO	-4.9	6.0
SHiP	-16.7	4.3
RWA	-1.6	0.7
IRWA	-17.1	-1.1
CLP	16.8	7.5
PM-VH-SD	19.9	12.9
PM-VM-SD	7.8	14.4
PL-VL-SD	4.1	13.1

Table 3.6: Memory lifetime improvements (percentages) with respect to LRU: multiprogrammed workloads.

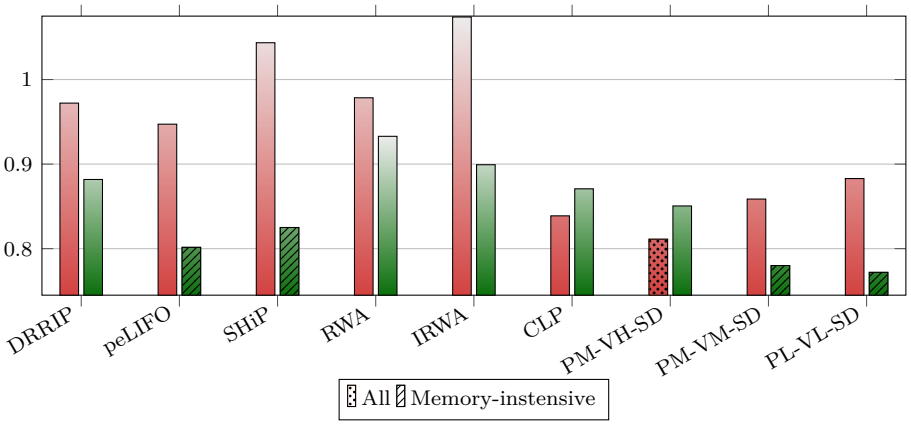
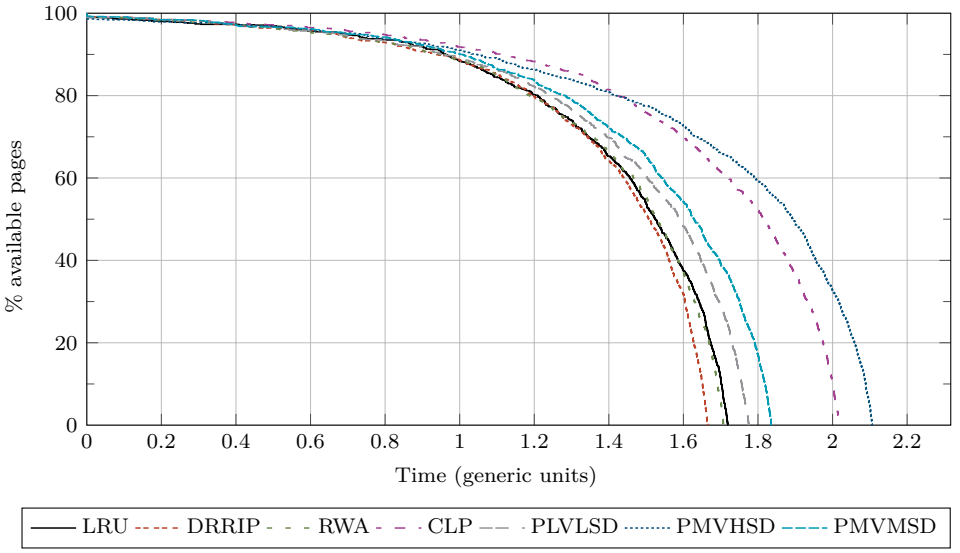


Figure 3.20: Writes to main memory per instruction normalized to LRU: multi-programmed workloads.

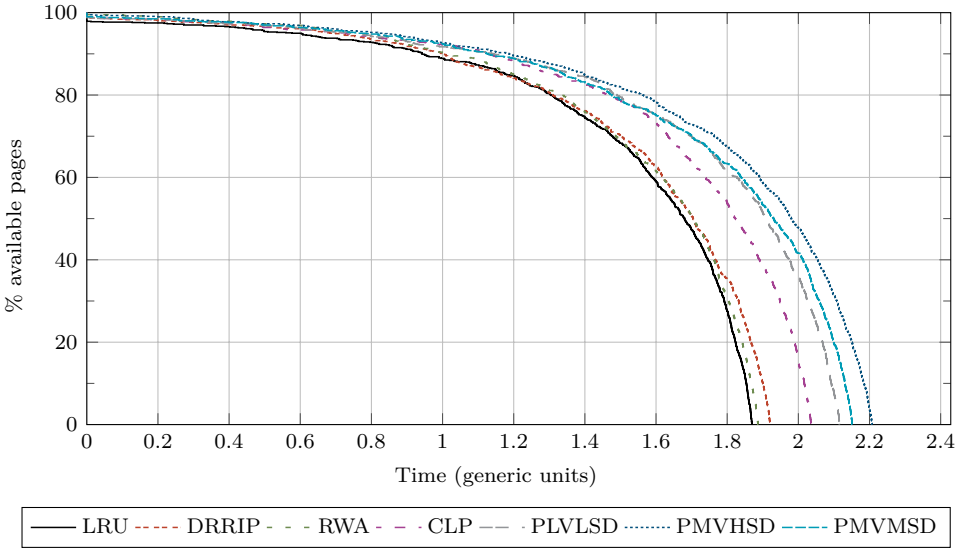
As in the single-core scenario, the general trend observed is similar to that of write reduction numbers. However, in this case we observe a higher variability in the ratios of BFPs when considering all applications (from 1.02 in CLP to 1.15 in SHiP), which mitigates the memory endurance extensions. These numbers are even higher when considering just the *memory-intensive* mixes, ranging from 1.04 in PM-VH-SD to 1.18 in peLIFO. The average number of dirty words per writeback are slightly lower than in the single-core scenario, oscillating between 5.31 in CLP and 5.98 in SHiP.

Figure 3.21 shows the memory available when using each proposal over the time, where we observe that, according to data of Table 3.6, PM-VH-SD exhibits the best behavior when considering all mixes and our three proposals are the best-performing across the board when considering just the *memory-intensive* workloads. Note also that in this scenario, unlike in the single-core environment, we do not observe the effect of a significant higher amount of programs execution for the *memory-intensive* workloads until the entire memory fails. This is due to the fact that the average number of dirty words per writeback for these workloads is just slightly lower than that when considering all the mixes, ranging between 4.73 and 5.57 among the evaluated policies.

3.3.2.1.2 Performance In order to evaluate the performance delivered by each proposal when executing multiprogrammed workloads, we analyze the *Instruction Throughput* (IT) and the *Weighted Speedup* (WS) metrics. The IT metric is defined



(a)



(b)

Figure 3.21: Available memory vs time for multiprogrammed workloads: (a) All, (b) Memory Intensive.

as the sum of all the number of instructions committed per cycle in the entire chip ($\sum_{i=1}^{n} IPC_i$, being n the number of applications/threads), while the WS is defined

as the slowdown experienced by each application in a mix, compared to its run under the same configuration when no other application is running on other cores ($\sum_{i=1}^{n}(IPC_i^{shared}/IPC_i^{alone})$). Figure 3.22 illustrates the IT that each evaluated policy delivers normalized to LRU. Note that, unlike in the case of execution time (or CPI) metrics, now values higher than 1 imply performance improvements with respect to LRU.

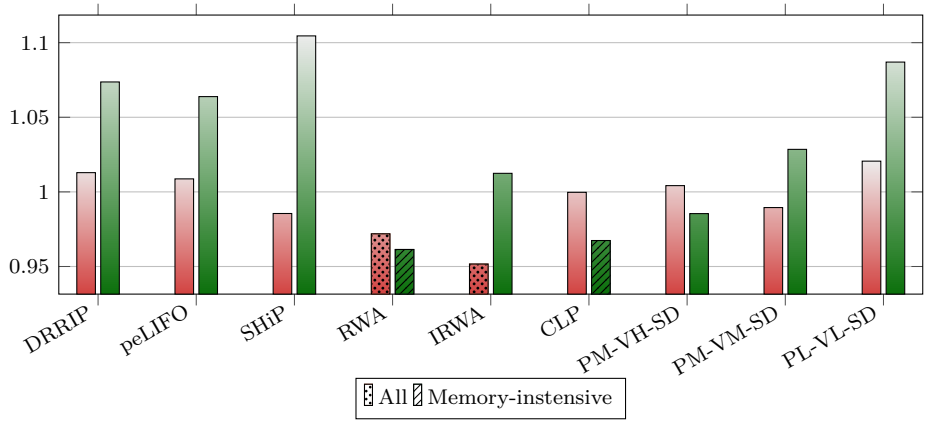


Figure 3.22: Instruction throughput normalized to LRU: multiprogrammed workloads.

First, we observe that although our medium-aggressiveness policy suffers a performance drop of around 1%, our high-aggressiveness algorithm slightly improves LRU performance and our low-aggressiveness scheme outperforms LRU (and also all the other algorithms evaluated) by more than 2%. Second, all our techniques behave better than RWA and IRWA *write-aware* policies, while CLP performs slightly worse than our most aggressive proposal. Third, zooming into the *performance-oriented* policies, the figure reveals that DRRIP and peLIFO slightly improve the LRU performance, while SHiP suffers a moderate penalty due to the high performance drop exhibited in mixes 7, 10 and 11, which include the *hmmer* application, for which, as stated in Section 3.3.1.3 SHiP performs especially poor.

Fourth, for the *memory-intensive* mixes, we observe that all evaluated policies, except the most aggressive ones (CLP and PM-VH-SD) and RWA, improve their IT with respect to considering all the mixes. Note also that original DRRIP performs significantly well for these mixes. Our low-aggressiveness proposal reports performance numbers that improve LRU by more than 8%, still outperforming all the

other techniques –except SHiP–. All our proposals also outperform RWA, while IRWA is able to report for these mixes an IT improvement over LRU of around 1%. Finally, we should also highlight that all our techniques clearly outperform CLP –even up to more than 10%– for these *memory-intensive* workloads.

For the sake of simplicity and since in our context the WS does not constitute a metric as significant as the IT, we do not show the WS results obtained. Anyway, these results follow an analogous trend as those obtained when we evaluate the *instruction throughput*.

3.3.2.1.3 Memory energy consumption As for the energy savings *per instruction*, our results reveal that our three evaluated proposals and CLP achieve the best results with numbers around 9% (our most aggressive policy) and 7% (CLP and our medium and low-aggressiveness schemes) with respect to LRU. The results from the other *write-aware* and *performance-oriented* policies are much more modest even augmenting the energy consumption as in the case of RWA, IRWA and SHiP. For the *memory-intensive* mixes, our PM-VH-SD policy reports energy savings in the memory hierarchy of around 5% whereas our medium and low-aggressiveness schemes are around 12 and 15% respectively. The best of the other *write-aware* techniques (IRWA) hardly reaches 7% while CLP is around just 3%.

3.3.2.2 Multithreaded applications

In this section we inspect the behavior of our proposals when using the parallel applications from the PARSEC suite running in the system depicted in Figure 3.15 with a shared L3 cache. We explore both a 2-CMP system where L3 is 1MB/16-Way and a 4-CMP system with a 2MB/16-Way L3[4]. Again, we report data considering both all applications and only the *memory-intensive* programs. Following the same criteria defined in Section 3.3.1.1, this group of benchmarks includes *vips*, *facesim*, *dedup*, *ferret* and *fuidanimate* for the 2-CMP system while in the 4-CMP it consists of *vips*, *facesim*, *dedup*, *ferret* and *canneal*. Note that for the PARSEC applications we use the same algorithms as when employing the

[4]Note that with PARSEC applications we do not follow the rule 1MB per core as we did with SPEC CPU2006 benchmarks. The reason is that with the PARSEC suite, the amount of writes to main memory is significantly lower than that observed with the SPEC suite, and hence we opted to employ the rule 0.5MB/core in the multi-core scenario when running PARSEC applications

SPEC benchmark suite.

3.3.2.2.1 Write filtering and endurance Figure 3.23 shows the number of writes to memory *per instruction* that each policy generates. We can observe that in both CMP systems all our proposals significantly outperform the *performance-oriented* policies (that roughly match and even exceed the number of writes of LRU) as well as the IRWA *write-aware* algorithm. Notably, our most aggressive policy reduces the write traffic to memory with respect to LRU by around 31 and 28% in the 2-CMP and 4-CMP evaluated systems respectively, improving CLP numbers by around 2 and 5% respectively. Moreover, when considering the *memory-intensive* programs, our best-performing policy is able to even involve amounts of writes 8% and 7% lower than those of CLP in the 2-CMP and 4-CMP systems respectively. These amounts of writes leads to the memory lifetime improvements with respect to LRU shown in Table 3.7.

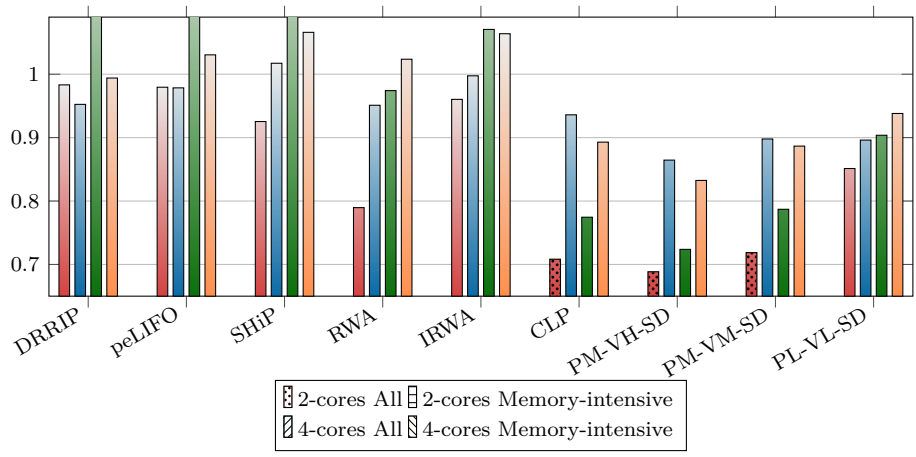


Figure 3.23: Writes to main memory per instruction normalized to LRU: PARSEC suite.

As in the case of the single-core scenario, in the 2-CMP system the trends observed in the memory lifetime improvements when considering all applications closely follow those of the write reduction numbers since the BFP ratios just ranges between 0.99 and 1.02. For the *memory-intensive* programs, these values are lower than 1 for all the policies across the board (except RWA), which further extends the memory lifetime, with PM-VM-SD and PL-VL-SD exhibiting the best numbers (0.95 and 0.96 respectively), just outperformed by SHiP with 0.93. Thus, SHiP,

Policies / Benchs	2-CMP		4-CMP	
	All	**Mem-intensive**	**All**	**Mem-intensive**
DRRIP	3.2	9.1	-8.7	11.2
peLIFO	2.5	4.7	-15.2	-6.8
SHiP	9.3	5.1	-23.9	-2.3
RWA	24.9	5.0	-2.5	-4.5
IRWA	4.2	2.7	-10.4	-4.6
CLP	38.5	6.9	35.1	24.2
PM-VH-SD	44.1	18.3	40.7	30.7
PM-VM-SD	39.3	17.1	29.0	22.9
PL-VL-SD	18.3	16.3	16.2	21.8

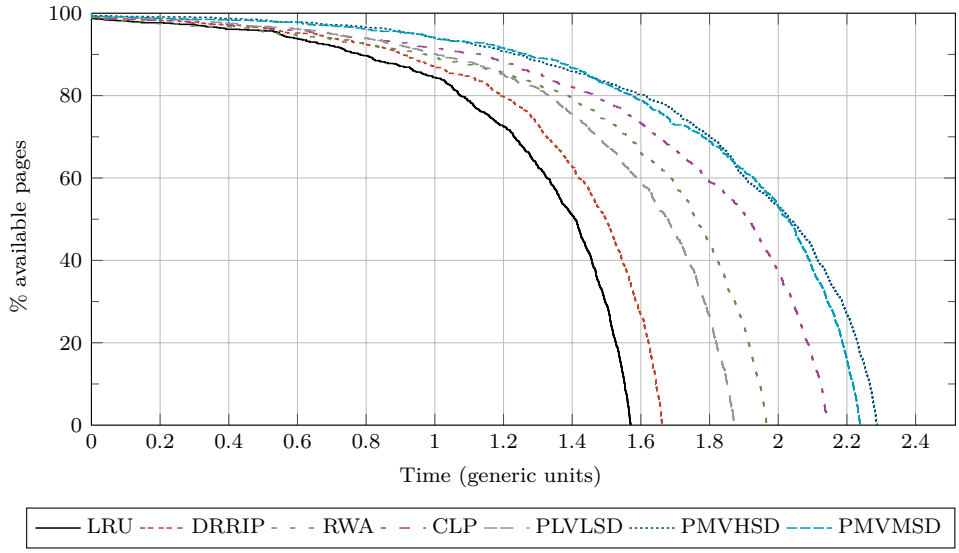
Table 3.7: Memory lifetime improvements (percentages) with respect to LRU: PARSEC suite.

even augmenting the amount of writes to memory with respect to LRU by 1.7% is able to extend the memory lifetime by 5.1% compared to the baseline. The average of dirty words per writeback ranges between 5.5 and 5.7 among the evaluated policies in this scenario.

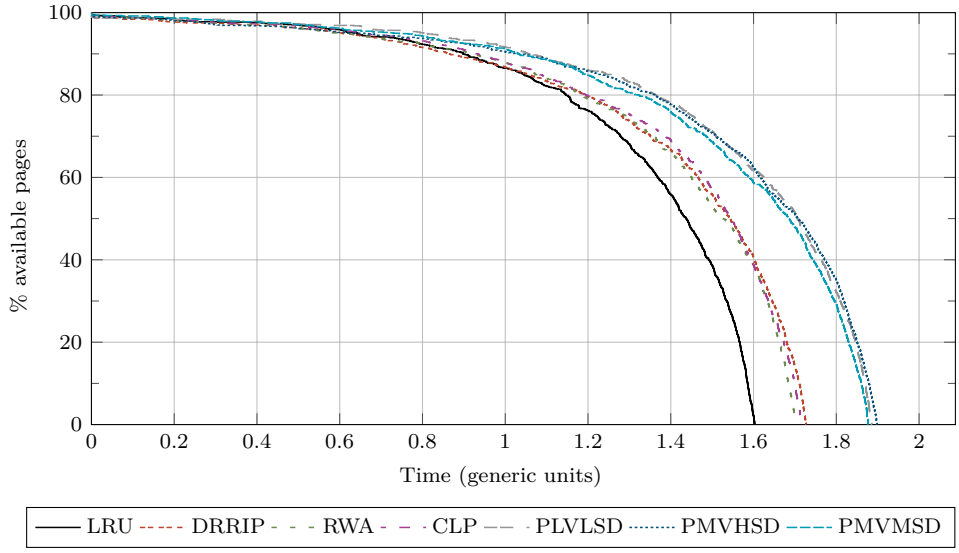
The 4-CMP system is the scenario in where our policies benefit the most from the BFP ratios, which considering all applications range between 0.95 and 0.98 and considering just the *memory-intensive* programs range among 0.87 and 0.91. Notably, PL-VL-SD, reducing writes with respect to LRU by 9.6 and 6.2% for *All* and *memory-intensive* groups respectively, manages to improve the memory lifetime by 16.2 and 21.8% respectively. The average of dirty words per writeback ranges between 5.0 and 5.5 in this scenario.

Figures 3.24 and 3.25 show the memory available over time for each evaluated proposal in the 2-CMP and 4-CMP systems respectively, where we observe that in all the cases PM-VH-SD exhibits the best behavior, significantly outperforming CLP. Furthermore, our three proposals behave especially well for the *memory-intensive* programs in both scenarios.

Note also here how, for our medium and high-aggressiveness policies and also CLP, the program executions number reached for *All* applications in the 2-CMP system before the entire memory fails is higher than that of the *memory-intensive* programs, since the difference between the memory lifetime improvements obtained by these two groups of benchmarks is the highest across all the scenarios evaluated, and, although the average of dirty words is slightly lower for *memory-intensive*



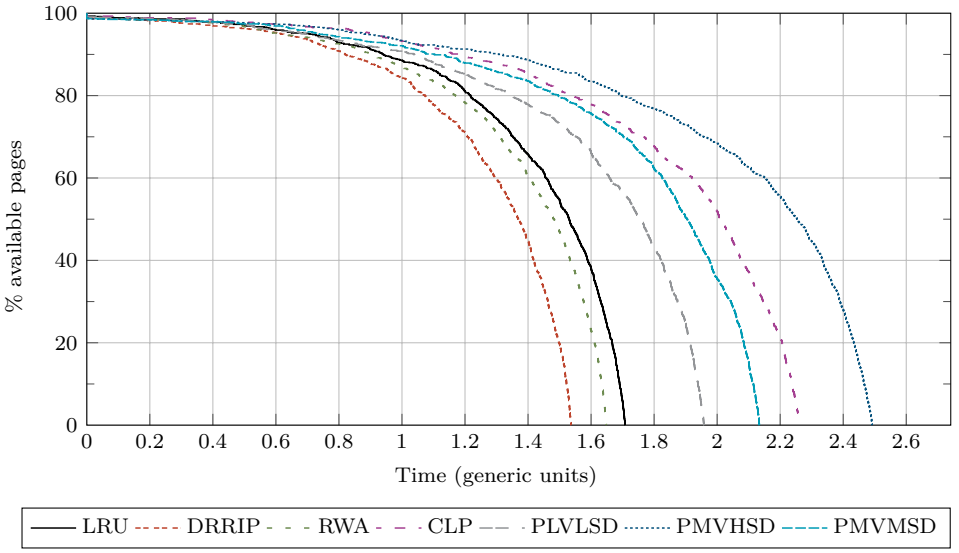
(a)



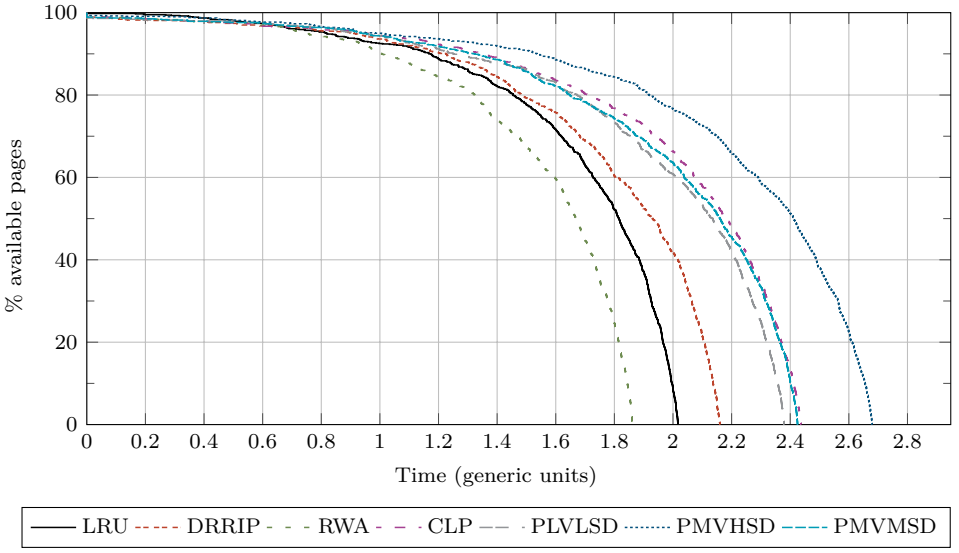
(b)

Figure 3.24: Available memory vs time for PARSEC suite in a 2-CMP: (a) All, (b) Memory Intensive.

applications, it is not enough to compensate the lower write reduction obtained in these applications.



(a)



(b)

Figure 3.25: Available memory vs time for PARSEC suite in a 4-CMP: (a) All, (b) Memory Intensive.

3.3.2.2.2 Performance Figure 3.26 shows the geometric mean of the *cycles per instruction* (CPI) reported by all evaluated policies.

First, we observe that our proposals behave slightly better in the 2-CMP system than in the 4-CMP one. Indeed, in the 2-CMP system most of our schemes even outperform original DRRIP.

Second, RWA performs worse than all of our proposals while IRWA outperforms most of them in both scenarios at the expense of reporting no memory lifetime extension or even punishing lifetime. CLP, the other *write-aware* policy under evaluation, reports worse numbers than all our proposals in both 2 and 4-CMP systems (up to 5 % performance degradation with respect to LRU in the 4-CMP system while our PM-VH-SD is able to improve LRU performance). Third, as expected, the *performance-oriented* policies –except peLIFO in the 4-CMP system– deliver satisfactory performance numbers.

Fourth, when considering the *memory-intensive* programs, in both CMP systems our algorithms that report higher write reduction than CLP are also able to exhibit better performance numbers.

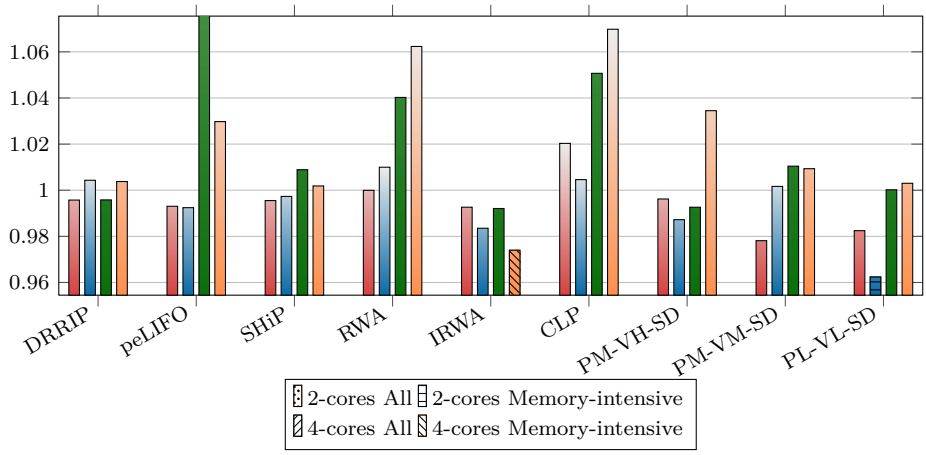


Figure 3.26: CPI normalized to LRU: PARSEC suite.

3.3.2.2.3 Memory energy consumption As for the energy consumption *per instruction* reported by the evaluated policies in the memory hierarchy, in the 2-CMP scenario the energy savings achieved are moderate across the board, being the best-performing policy our low-aggressiveness PL-VL-SD, with reductions by around 5% compared to LRU. In the 4-CMP system our proposals obtain more modest numbers, up to 2.5%. In both scenarios they all outperform the other

evaluated policies unless DRRIP in the 4-CMP system, which also reduces the energy consumption by around 2.5%.

For the *memory-intensive* programs, in the 2-CMP system our proposals obtain numbers between 8 and 6.5% while CLP reports energy savings of just 2.3% with respect to LRU. In the 4-CMP, our proposals report energy savings in the range 5-4.5%, while CLP reduces the energy consumption by 1%.

3.3.2.3 Putting it all together

Analyzing the trade-off between memory endurance and performance exhibited by the evaluated proposals in the multi-core scenario, we observe that, when running multiprogrammed workloads, PM-VH-SD is able to report the highest memory lifetime improvement (19.9%) and the highest reduction in the energy consumption in the memory hierarchy (more than 9%) without degrading performance. We must also highlight the behavior of our low-aggressiveness PL-VL-SD policy, which clearly reports the best trade-off across the illustrated proposals[5] when considering only the *memory-intensive* programs. In such scenario, it achieves the second-highest memory lifetime improvement (13.1%, just slightly outperformed by PM-VM-SD), the second-highest throughput value (8.7% improvement with respect to LRU, just outperformed by SHiP) and the highest energy savings (more than 15%). In the same scenario CLP reports 7.5% of endurance extension, 3.3% of throughput degradation and just 3.3% of memory energy savings.

In the 2-CMP system running parallel applications, our most aggressive policy, extending the memory lifetime up to 44.1%, provides also satisfactory performance numbers (around 1% improvement over LRU). Our medium-aggressiveness algorithm, that improves the memory endurance by around 39%, experiences also a performance improvement of around 2%. Finally, our least aggressive policy, although extending memory endurance by a moderate 18%, manages to also improve performance by around 2%. They also are able to deliver energy savings of 3, 4 and 5% respectively. Thus, for our purpose of extending the memory lifetime without penalizing performance, PM-VH-SD seems to be the best option in this scenario.

[5]In the multiprogrammed workload scenario, other of our low-aggressiveness schemes report better numbers than the chosen algorithm. Notably, PM-VL-SD, delivering the same performance and energy numbers as PL-VL-SD, is able to reach 8.1% and 15.6% memory endurance extension for *All* and *memory-intensive* programs respectively vs 4.1 and 13.1% obtained by PL-VL-SD respectively.

Note that CLP improves memory lifetime by 6% less than our proposal and also behaves worse (2.5%) than PM-VH-SD in terms of performance. Moreover, for *memory-intensive* programs, our three proposals significantly outperform CLP in memory lifetime while also deliver higher performance.

In the 4-CMP, PM-VH-SD clearly provides again the best trade-off across the board (40% memory lifetime extension, performance improvement of around 1% and energy savings of 1.5%). Note that CLP improves endurance by 35%, at the expense of a performance drop of around 5%. Finally, it is worth to note that the same trends, even augmented in favour of our proposals, are maintained when only the *memory-intensive* programs are considered.

Overall, we consider that in all the multi-core systems analyzed we always may find at least one of our policies that is able to clearly deliver a better trade-off (higher lifetime extension and also higher performance) than the other *write-aware* proposals and also *performance-oriented* algorithms. Notably, most *performance-oriented* policies decrease the memory lifetime with respect to LRU and even exhibit worse performance numbers than some of our proposals, while the RWA and IRWA *write-aware* policies clearly exhibit poor trade-offs. Finally, for CLP, which also reports significant extensions in memory endurance, note that we have performed a direct comparison with our proposals along the evaluation section to demonstrate the better trade-offs achieved by our policies.

3.3.3 Additional analysis

In this final section, we extend our study with some additional experiments. Specifically, we start by comparing some *performance* and *write-aware* oriented policies with an optimal one. Then, we estimate the implementation overhead of our proposals. Finally, we evaluate the sensitivity to the LLC size of the different *write-aware* policies as well as DRRIP.

3.3.3.1 Comparison to an optimal policy

In order to find out how far we are from the maximum feasible write reduction, we compare the *performance-oriented* policies which report the lowest amount of writes –DRRIP and LRU–, our most aggressive *write-aware* algorithm evaluated in the previous sections (PM-VH-SD) and also CLP, with a straightforward optimal

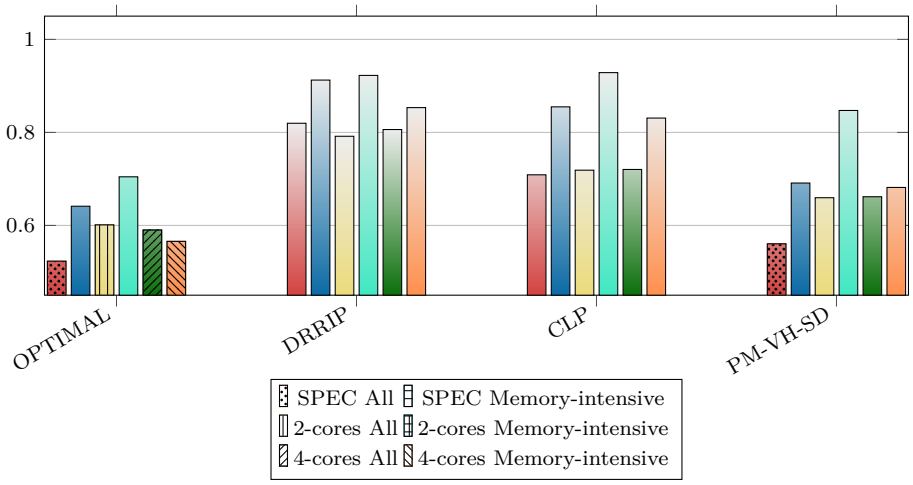


Figure 3.27: Amount of writes to main memory normalized to LRU: optimal policy and ours.

policy (in terms of writes to main memory), that operates as follows: For a given cache set, we allocate an array with an amount of entries matching the associativity (n) of the LLC, and traverse the trace of writebacks from L2 to L3. We first fill the array with the first n different blocks that are written back. Then, for every new block written back from L2 to L3 and not contained in the array, we do the following: (1) we analyze the trace onward to find the block in the array that will be re-written the furthest in the future; (2) we replace the found block with the incoming one (or we just bypass the new block if it is re-written further than all the other blocks in the array); and (3) we increment a counter. When the entire trace has been processed, the counter stores the total number of writes to main memory generated in this set under an optimal replacement policy.

Given that the simulation of the optimal policy is a time-consuming process, we opted to restrict the analysis to a pool of sets (32) conveniently scattered across the cache. Obviously, the other policies are also evaluated in the same cache sets.

Figure 3.27 illustrates the amount of writes to memory that each evaluated policy implies normalized to LRU for both SPEC (single-core) and PARSEC benchmarks (2 and 4-CMP systems). Although the analysis is limited due to the reduced number of sets considered, we can infer a couple of relevant qualitative conclusions. First, our most aggressive policy is moderately close to the maximum theoretical

reduction. Notably, PM-VH-SD is the algorithm closest to the optimal policy in all the scenarios evaluated for both *All* and *memory-intensive* benchmark sets. Second, for both SPEC applications and parallel programs in the 2-CMP system, the maximum write reduction feasible for the *memory-intensive* programs is lower than that when considering all the benchmarks (in the 4-CMP system these numbers for both groups of benchmarks remain laregely unchanged). Overall we can extract the conclusion that, although the optimal policy is unfeasible and our proposals are close to the optimal numbers, it seems that it still remains opened some avenue for improvement in order to even further reduce the current gap with this optimal policy and even improving the performance delivered.

3.3.3.2 Implementation overhead

A new policy may induce two types of overhead, specifically, extra hardware (both extra storage needed for book keeping and extra logic needed for implementing the new algorithm) and impact on the critical path. We should start mentioning that, as many authors have previously pointed out, the updating of the replacement policy state is completely off the critical path [23], [26], which makes the critical path delay remain unaltered by our proposed changes. However, for the sake of completeness, we will prove in this section, by means of deeply analyzing the algorithm complexity of our proposals, that the delay generated by our changes, as well as the extra logic they involve, are both negligible compared to DRRIP. Besides, we will analyze the extra storage involved by our policies. To complete the section, we will also analyze the overhead of DRRIP, LRU, CLP, RWA and IRWA.

3.3.3.2.1 Algorithm complexity Let us analyze the algorithm complexity of each sub-policy for the different replacement policies, which can provide us an idea of the logic that they involve.

- Insertion sub-policy: With respect to DRRIP, our policies only modify the *set-dueling evaluating metric*, so it comes "for free".

  Moreover, comparing DRRIP with LRU, note that, whereas the former policy only requires to update the RRPV of the new block, the latter one needs to update the position in the Recency Stack of all the blocks in the set.

- Promotion sub-policy: Our policies incorporate cleanness/dirtiness (PL) or

read/write (PM and PH) information to the promotion process, something extremely simple. Besides, all this information is already present in the block or comes along with the access itself.

Furthermore, compared to LRU, DRRIP promotion is much easier, since, like in the insertion sub-policy, DRRIP just updates the RRPV of the promoted block, whereas LRU promotion updates also the blocks between the promoted one and the MRU position.

- Victimization sub-policy: As established in [23], original DRRIP searches for the first block with the highest RRPV by replicating the Find First One (FFO) logic, requiring four FFO circuits that operate in parallel to find pointers to the first "0", "1", "2", and "3" RRPV registers. Then, a priority MUX chooses the output of the appropriate FFO circuit as the victim. When a block with *distant* RRIP is not found, DRRIP also requires additional logic to age the RRPV registers, using a state machine for this purpose. Our policies could either duplicate the number of FFO circuits for distinguishing between clean and dirty blocks, leaving the delay intact vs DRRIP, or access the four FFO circuits sequentially, first for clean blocks and then for dirty blocks (if needed), leaving the logic overhead almost untouched with respect to DRRIP. Besides, VM entails some changes on the aging logic for being able to increment only clean blocks during the first stage of this sub-policy.

  Now let us move to the comparison among LRU and DRRIP. Similarly to DRRIP, the former policy requires a search for finding the LRU block. However, LRU needs no aging logic, making its complexity slightly lower.

To sum up, the time delay and extra logic that our policies introduce with respect to DRRIP can be considered negligible, whereas they are higher in LRU than in DRRIP.

3.3.3.2.2 Extra storage Original DRRIP just requires 2 bits per cache block for storing the state associated to the replacement policy. In our policies, no extra bits need to be added to those required by DRRIP.

Furthermore, as stated by the authors in [23], DRRIP implies less hardware overhead than LRU. Notably, the number of bits required per cache set, assuming an N-way associative cache, is $N * logN$ in LRU and $2 * N$ in DRRIP, which, for

example, in our setting (LLC associativity = 16), would lead LRU to need twice as much storage as DRRIP.

Finally, note that the implementation complexity of CLP, given that this policy is based on a conventional LRU, would be the same (or even higher) as that of LRU. Other approaches in which CLP is based on efficient implementations of LRU (such as Tree-LRU) would be possible, but this would come at the expense of some performance degradation. RWA and IRWA, like in our case, are DRRIP-based policies so they entail a negligible implementation overhead compared to original DRRIP. Indeed, the overhead is even smaller than in our policies due to the fact that RWA and IRWA preserve the victimization sub-policy unchanged with respect to DRRIP.

3.3.3.3 Sensitivity to LLC size

In order to further evaluate the impact of our proposals, we inspect the reduction in the number of writes to memory[6] and also the performance delivered when larger sizes of LLC are considered. To analyze our proposals operation we scale the problem by augmenting the LLC size without changing the number of cores. We choose a 4-CMP system and explore multiprogrammed workloads (using SPEC CPU2006 applications) with LLC sizes of 4, 8 and 16 MB and parallel applications (PARSEC) employing 2, 4, 8 and 16 MB LLC sizes. We just evaluate LRU, DRRIP, CLP and our three chosen policies (PM-VH-SD, PM-VM-SD and PL-VL-SD). Figures 3.28 and 3.29 show the amount of writes per instruction and the performance (CPI) respectively when using multiprogrammed workloads, whereas Figures 3.30 and 3.31 illustrate the same information (using the throughput metric instead of CPI to measure performance) when multithreaded programs are employed.

For the evaluated mixes we observe that the same trends are still valid when we increase the LLC size. First, our PM-VH-SD policy reports again the highest write reduction across the board when considering all the mixes, slightly outperforming CLP, while also delivers a higher throughput than CLP. Second, for the *memory-intensive* workloads (labeled as M.I.), although the differences get reduced with respect to a smaller LLC size, most of our proposals still outperform CLP in both

[6]For the sake of simplicity we omit endurance results, that follow a similar trend to write reduction as previously demonstrated.

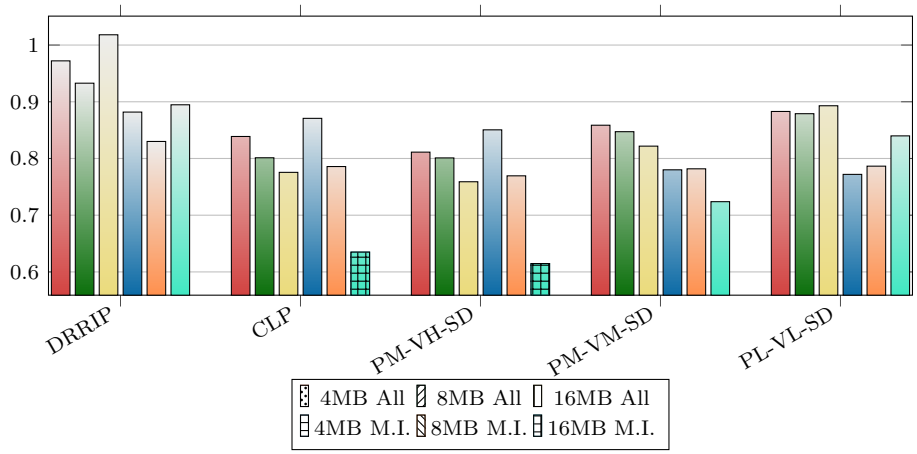


Figure 3.28: Writes to main memory per instruction for different LLC sizes normalized to their respective LRU: multiprogrammed workload.

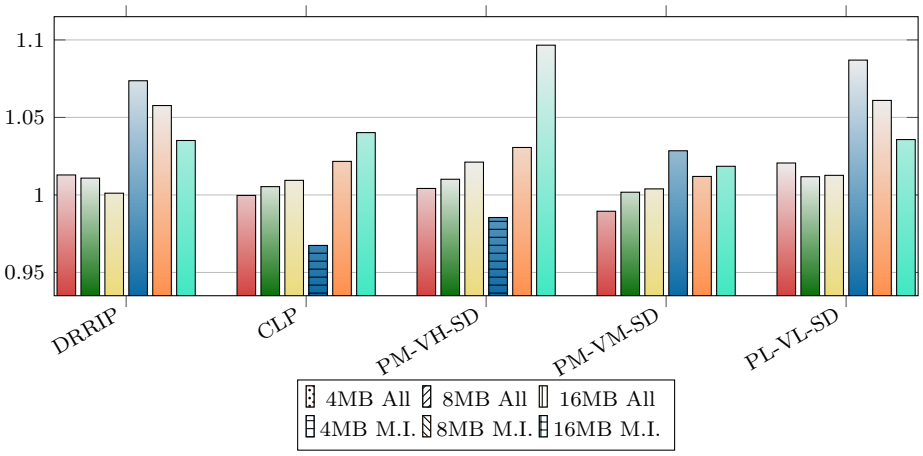


Figure 3.29: Instruction Throughput for different LLC sizes normalized to their respective LRU: multiprogrammed workloads.

write reduction capability and throughput. Thus, for this kind of applications, our PL-VL-SD policy, matching the CLP number regarding the amount of writes to memory in the 8MB scenario, is able to deliver a throughput 4% higher than CLP. Conversely, in the 16MB scenario, CLP is just outperformed by our PM-VH-SD policy.

For multithreaded applications, as shown, PM-VH-SD always reports the best numbers in cutting the write traffic to memory across the board for the four LLC

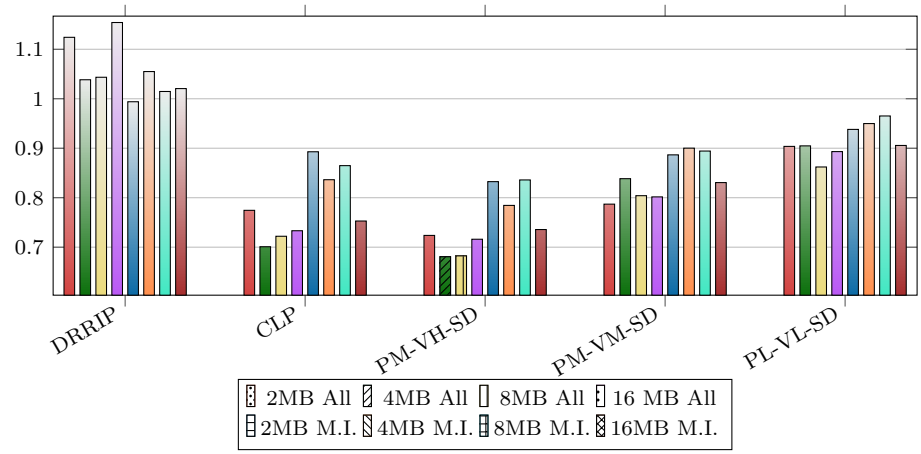


Figure 3.30: Writes to main memory per instruction normalized to LRU for different LLC sizes: PARSEC suite.

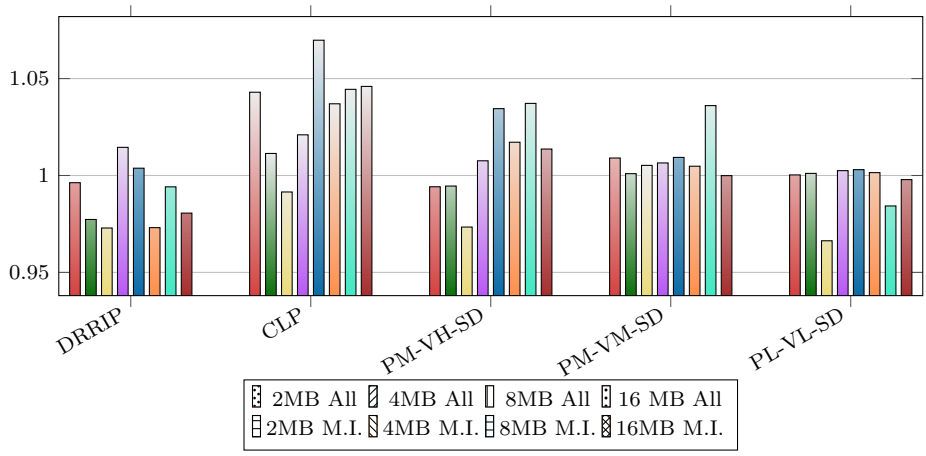


Figure 3.31: CPI normalized to LRU for different LLC sizes: PARSEC suite.

sizes analyzed and for both *All* and *memory-intensive*[7] groups of benchmarks, moderately outperforming CLP. Furthermore, our PM-VH-SD scheme, being the technique that provides the highest reduction in amount of writes to memory, also behaves correctly in terms of performance, significantly outperforming CLP (up to 5%) in all the scenarios evaluated. Finally, note that our medium and low-aggressiveness schemes analyzed also exhibit better performance numbers than

[7]Note that the group of memory-intensive applications, chosen according to the criteria explained in Section 3.3.1.1, is not made up of exactly the same programs for all the LLC sizes considered.

CLP for all the LLC sizes and groups of benchmarks under evaluation.

3.4 Conclusions

In this chapter we have addressed the endurance constraint of the phase-change memories by redesigning the last-level cache replacement policy. First, we have evaluated the operation of classical replacement algorithms in terms of writes to main memory and then we have proposed new policies with the main goal of minimizing this number of writes to main memory in order to extend the PCM lifetime. The foundation behind these proposals is to merge as many modifications to a block as possible in a single writeback, while maintaining the system performance.

According to our results, the conclusions are triple. First, as the conventional *performance-oriented* replacement algorithms are absolutely unaware of the number of writes performed to memory, they entail low PCM lifetime values, which suggests that they must be adapted to a potential future scenario where PCM-based systems prevail. Second, combining efficiently the proposed changes to the insertion, promotion and victimization sub-policies leads to algorithms that significantly cut the write traffic to memory and hence increase its lifetime, with low impact on performance. Specifically, our most aggressive policy delivers the best trade-off between endurance and performance in both the single and the multi-core scenarios, reporting memory lifetime improvements in the range 20-45% at the cost of a very small impact on performance. Besides, the write traffic reduction it achieves is not far from optimal. Concerning our medium and low-aggressiveness algorithms, they also report satisfactory results in both scenarios evaluated, managing to moderately improve the PCM lifetime and also reduce the energy consumption while delivering satisfactory performance results. Third, as demonstrated in Section 3.3, most previously proposed *write-aware* policies (such as RWA and IRWA) clearly fail to achieve satisfactory trade-offs.

4

Reuse Detector for STT-RAM LLCs

In the last years chip multiprocessors have become majority on many off-the-shelf systems, such as high performance servers, desktop systems, mobile devices and embedded systems. In all of them the designers usually include a multilevel memory hierarchy, where the shared last-level cache (LLC) plays an important role in terms of cost, performance and energy consumption. As for the cost, the LLC generally occupies a chip area similar or even bigger than the cores. Regarding performance and energy consumption, the LLC is the last resource before accessing the main memory, which delivers higher energy consumption and lower performance as it is located outside the chip.

As detailed in Section 1.2.2, the technology currently employed in building LLCs is mainly SRAM (and also, to a lesser extent, embedded DRAM). However, they both reveal as power-hungry, especially for the large sizes required as the number of cores increases. One way of mitigating this problem is to employ emerging non-volatile memory technologies. Among them, Spin-Transfer Torque RAM (STT-RAM) is clearly the prime contender. STT-RAM removes almost all the static power consumption and, compared to SRAM, provides higher density and therefore much higher capacity within the same budget. Moreover, it delivers higher read efficiency in terms of latency and energy. Nevertheless, some obstacles restrict the adoption of STT-RAM as last-level cache for the next generation of CMPs: its write operation is slower and requires more energy than in an SRAM cache. These constraints may lead to a performance drop and even to almost cancel the energy savings derived from the minimal static power consumption of STT-RAM.

In addition, previous research states that conventional shared LLC designs are inefficient since they waste most storage space [76], [77]. This is due to the fact that LLC management policies often lead to store *dead blocks*, i.e. blocks that will

not be referenced again before their eviction. This is mainly because the cache levels closer to the processor exploit most of the *temporal locality*, which therefore becomes largely filtered before accessing the LLC. With the goal of avoiding this effect and hence increasing the hit rate, various mechanisms that modify the LLC insertion and replacement policies have been proposed recently.

This chapter addresses the shortcomings aforementioned by focusing on improving the efficiency, in terms of both performance and energy, of a non-inclusive and non-exclusive STT-RAM LLC in a chip multiprocessor system. Notably, we present a new mechanism of content selection for last-level caches that benefits from the *reuse locality* that LLC references exhibit [102], [103]. This locality lies in the following principle: when a block is referenced twice in the last-level cache (i.e. it is reused), this block has a good chance of being referenced again in the near future. Our approach pursues to insert in the LLC only those blocks that exhibit reuse at that level. For this purpose, we propose to include a hardware resource –referred to as *Reuse Detector*– between the LLC and the private cache levels which determines for each block evicted from the private cache levels if it has been reused or not at the LLC. If the block is determined to having been reused, it is inserted (or it updates) in the LLC. Otherwise, the block *bypasses* the LLC and is sent directly to main memory.

4.1 Proposed design

In this section we first motivate the need of a new LLC management scheme (Section 4.1.1) by describing the main limitations of conventional management and also recalling those of SRAM technology. Then we describe in Section 4.1.2 the baseline system we start from in designing our mechanism of content selection for LLCs. In Section 4.1.3 we describe in detail the proposed design built on top of that baseline system. Finally we briefly discuss the main differences between our proposal and the DASCA scheme, which is the closest approach to our work and the state-of-the-art STT-RAM LLC management scheme [76].

4.1.1 Rationale

As detailed in Section 2.3 (Table 2.2), key aspects of employing STT-RAM instead of SRAM as last-level cache technology are that an STT-RAM cache ex-

hibits smaller die footprint and better efficiency in read operation than an SRAM cache. Also, an STT-RAM cache consumes more than an order of magnitude less static power compared to SRAM. However, STT-RAM caches exhibit a significant drawback that needs to be mitigated: the poor write performance both in terms of latency and energy consumption.

Moreover, regardless of implementation technology, last-level caches usually suffer from the same problem: they keep data assuming that recently referenced lines are likely to appear in the near future (*temporal locality*). Nevertheless, various recent studies point out that the reference stream entered in the LLC does not usually exhibit temporal locality. Notably, in [102] the authors observe that this reference stream exhibits *reuse locality* instead of temporal locality. Essentially, that term describes the property that the second reference to a line is a good indicator of forthcoming reuse and also that recently reused lines are more valuable than other lines reused a long time ago.

The studies carried out in [102], [103], [104] demonstrate, considering a large amount of multiprogrammed workloads and different multiprocessor configurations, two important aspects: first, most lines in the LLC are dead (they will not receive any further hits during their lifetime in the LLC) and second, most LLC hits come from a small subset of lines. We have performed our own analysis about the behavior of the blocks evicted from the LLC, in terms of the amount of accesses they receive before eviction, in the scenario depicted in Figure 4.3 but, as a starting point, employing just one core (detailed configuration is shown in Table 4.1). Figure 4.1 illustrates this behavior, grouping the blocks into three different categories: no reuse, just one reuse or more than one reuse (multiple reuse).

As shown, our experimental results confirm that most lines in the LLC are dead. Notably, around 70% of the blocks do not receive any further access since the moment they enter the LLC. Only around 5% of the blocks just receives one further hit (i.e. one reuse) and around 25% exhibit more than one reuse.

Consequently, getting blocks with just one use (the block fill, so no reuse) to bypass the LLC when they are evicted from the previous level caches, and just storing blocks with reuse (at least two LLC accesses), should allow to hold the small fraction of blocks with multiple reuses, increasing the LLC hit rate and improving

Figure 4.1: Breakdown of blocks replaced from the LLC according to the number of accesses they receive before eviction.

system performance.

Furthermore, Figure 4.2 shows that most LLC hits are to blocks having multiple reuses, which together with the aforementioned fact that most blocks inserted in the LLC do not experience any reuse, highly justify the idea of a content selector based on reuse detection between private caches and LLC.

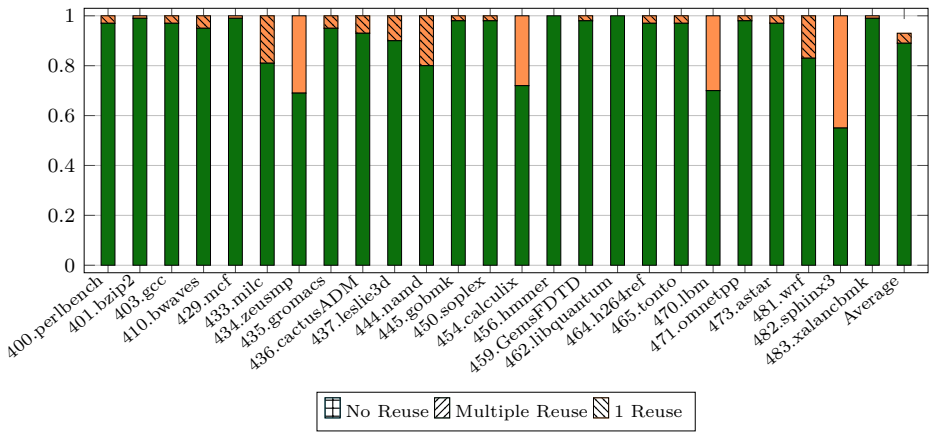


Figure 4.2: Breakdown of block hits at the LLC according to the number of accesses they have received before read.

4.1.2 Baseline system

The memory hierarchy used in the baseline multi-core system includes two private levels (L1 and L2) and a last-level cache shared among all the cores. All caches are write-back, write-allocate and employ LRU as replacement policy. L1 and L2 are inclusive while the LLC is non inclusive.

The baseline management of this memory hierarchy is as follows: When a block is requested to Main Memory (MM), it is copied to the private cache levels of the requester core, but not to the LLC. During its lifetime at the private levels of the core, the block can be requested by other cores, in which case a copy will be sent from a private L2 cache to the L1-L2 caches of the requesting core, as dictated by the directory-based coherency mechanism (please refer to Section 4.2 for more details on the coherency mechanism).

When a block is evicted from an L2 cache, the LLC is checked: In case the block is not present there (either because it has not been inserted yet or because it has already been inserted and evicted by the LLC replacement mechanism), it is inserted in the LLC; otherwise, if the block is already in the LLC, the block is updated or just discarded, depending on whether the block is dirty or clean respectively. Thus, in our hierarchy, LLC insertions never come from MM but from an L2, in a similar way to an exclusive policy. Note however that our mechanism differs from an exclusive policy in that, as a result of a hit in the LLC, the block is copied to the private cache levels of the requester core, but maintained in the LLC.

4.1.3 The Reuse Detector

As explained earlier, several works have demonstrated that a notable percentage of the blocks inserted/updated in the LLC are in fact useless, as they are dead-blocks [76], [77]. These useless blocks are harmful, as they might evict other blocks which could potentially be useful in the future, and moreover, they increase the amount of writes to the LLC, which in the context of NVMs is far from convenient, as explained in previous chapters.

In this work we leverage the technique for reducing the amount of dead-blocks inserted/updated in the LLC [104] to improve the efficiency of an STT-RAM LLC.

In [104], the authors present a proposal that, in an exclusive memory hierarchy, reduces the amount of blocks inserted in a conventional LLC by around 90%. We apply this technique to a different context, i.e., to a non-inclusive STT-RAM LLC design within a memory hierarchy where L1-L2 are inclusive. The exclusion policy employed in [104] implies that, upon a LLC hit, the block is copied to the private cache levels and removed from the LLC. In our case, the block is inserted in the LLC at the end of its usage in the private caches and remains in the LLC until eviction. For our purpose, we include an intermediate element between each private L2 cache, Last-Private-Level cache (LPL), and the shared LLC (Figure 4.3). A block evicted from the private L2 caches is targeted to the corresponding element, which we denote as Reuse Detector (RD), instead of accessing directly to the LLC as it would do in the baseline system. The RD decides whether to send the block to the LLC or not (i.e. to bypass the shared last-level cache), by means of a prediction about the future usefulness of the block. We must highlight that, being the RD out of the block request path to the LLC, it does not impact the LLC hit or miss latencies.

For accomplishing the RD prediction, we apply Albericio's concept of reuse locality [102], [103]. As such, if the evicted block from the L2 has never been requested to the shared LLC since the time it entered the cache hierarchy (i.e. it has never been reused at the LLC), the block is predicted as a dead-block and thus it bypasses the LLC, directly updating MM (if the block is dirty) or being just discarded (if it is clean). Otherwise, if the block has been reused (i.e. it has been requested to the LLC at least once since the time it entered the cache hierarchy) and thus it is predicted as a non-dead-block, it is inserted/updated in the LLC.

The RD consists of a FIFO buffer and some management logic. The buffer stores the addresses of the blocks evicted by the corresponding last-private-level cache in order to maintain their reuse state. Moreover, an extra bit, called *reuse bit*, is added to each cache line in the private levels. This bit distinguishes if the block was inserted at the private cache levels from the shared LLC or from another private cache level (*reuse* bit is 1), or main memory (*reuse* bit is 0). In the following sections, we analyze in detail the Reuse Detector operation and implementation.

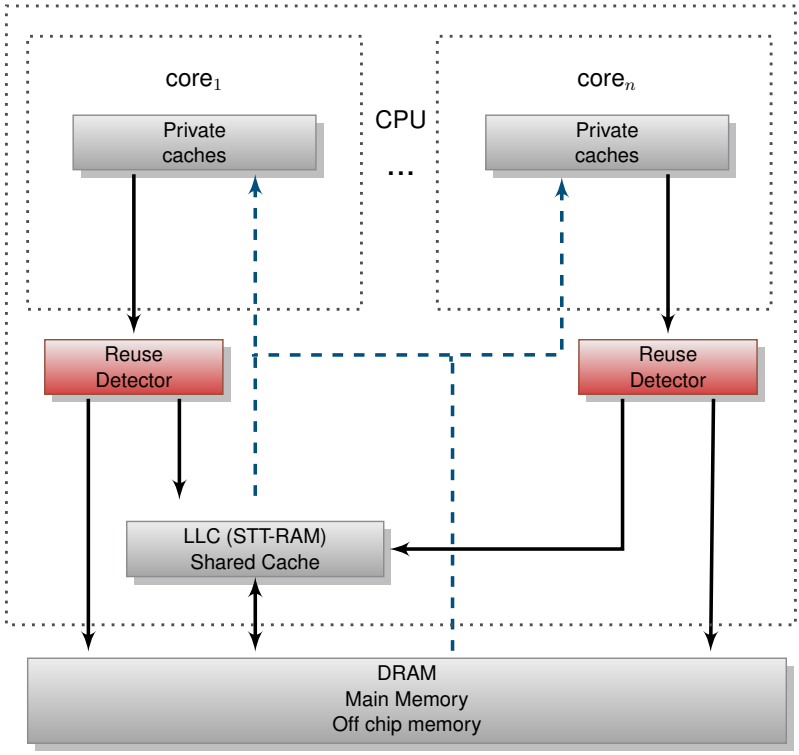


Figure 4.3: Placement of a Reuse Detector between each private L2 level and the STT-RAM LLC.

4.1.3.1 Reuse Detector operation

As we said in a previous section, our proposal aims to reduce the amount of writes to the STT-RAM LLC and to improve the management of LLC blocks, which translate into performance improvement and energy consumption reduction on the system.

Figure 4.4 shows a flow diagram of a block request from a core to its associated private caches. If the request hits in L1 or L2 the reuse bit is untouched, and the block is copied in L1 if it was not there (inclusive policy). Otherwise, the request is forwarded to the LLC. If the access hits in the LLC, the block is provided to the core and copied in the private levels with the reuse bit set. If the access misses in the LLC but the coherency mechanism informs that the block is present in another private cache, the block is provided by that cache. In this case, the access

is recognized as a reuse, so the reuse bits are also set. Finally, if no copy of the block is present in the cache hierarchy, it is requested to MM and copied in L1-L2 with the reuse bit unset.

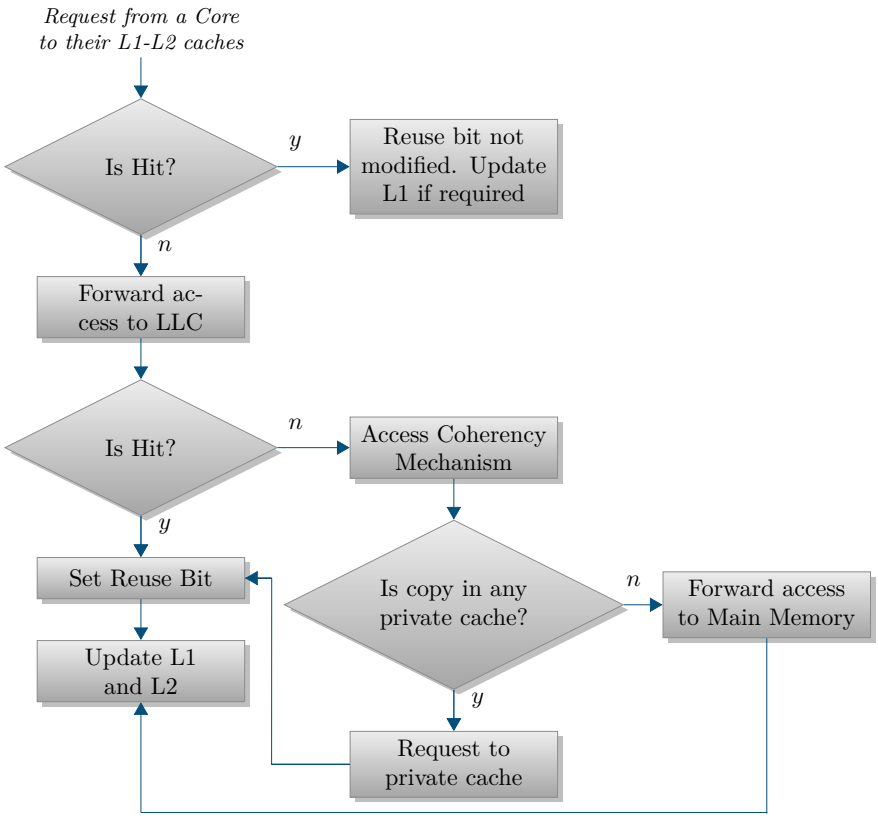


Figure 4.4: Block request and reuse bit management.

Figure 4.5 shows a flow diagram of a block eviction from an L2 cache (if required, the corresponding L1 cache is invalidated). When a block is evicted from a last-private-level cache, its reuse bit is checked. If the reuse bit is set, it means that the block was inserted into the private caches either from the LLC or from another private cache after checking the LLC and the coherency mechanism. In any case, the block is considered as having been reused, and it should be inserted in the LLC (if not present yet) or just updated (if the block is dirty but it is already present in the LLC). Note that if the block is clean and already present in the LLC, it can just be discarded. If the reuse bit is unset (i.e. the block was brought into the

private caches directly for main memory) but the block's tag is found in the RD buffer, the block is also considered as having been reused, and thus it is handled as in the previous situation. Finally, if the reuse bit is unset and its tag is not present in the RD buffer, it means that the block is considered as not having been reused yet. In this case, the tag is inserted in the RD, and, based again on Albericio's observations [102], [103], the block should bypass the LLC, as it is predicted as a dead-block, and it should be sent to MM (if the block is dirty) or just discarded (if it is clean). Note that in all cases the coherency mechanism must be updated.

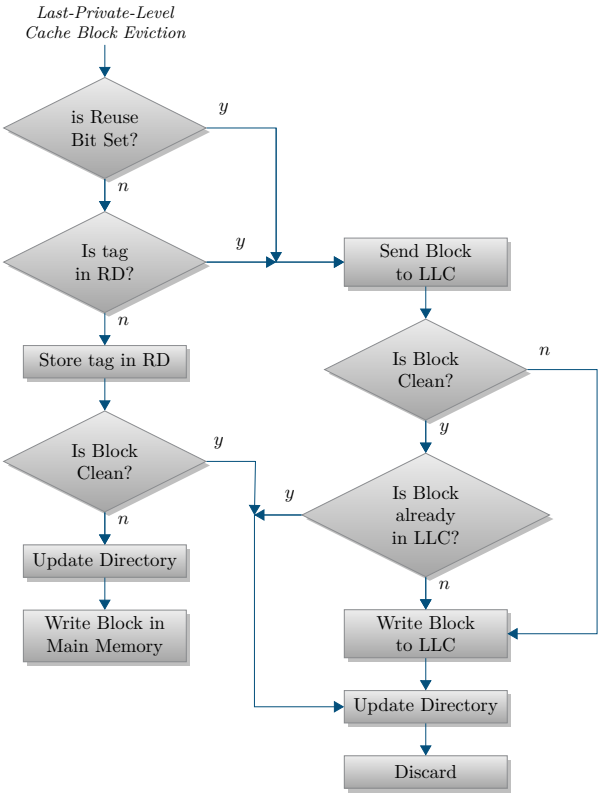


Figure 4.5: Block eviction from a last-private-level cache and LLC insertion.

4.1.3.2 Example

For the sake of completeness, in this subsection we provide a straightforward example illustrating the flow of five memory blocks (A, B, C, D and E) through the

different cache levels under a given access pattern. In this example, we consider a dual-core system ($Core_0$ and $Core_1$) with private first level caches ($L1_0$ and $L1_1$), a shared second level cache (LLC), and the corresponding Reuse Detectors between both cache levels. In the example we assume a simplified configuration where: 1) direct-mapped L1s, 2-way set associative RDs and a 4-way set associative LLC are considered; 2) all memory blocks map to the same L1 frame and to the same RD and LLC set; and 3) initially, all caches and RDs are empty. Next, we detail the access sequence of our example and show the contents of the memory hierarchy after each access in Figure 4.6. Note that we specify as a subindex the dirty bit followed by the reuse bit ($X_{d,r}$) for each block X in the private cache levels, and only the dirty bit (X_d) for each block X in the LLC.

1. $Core_0$ requests a word within block A for reading: The access misses in $L1_0$, it is forwarded to the LLC, and given that the access to LLC also misses and the block is not present in any other private cache, it is forwarded to MM. According to Figure 4.4, block A is copied to $L1_0$ with its reuse bit unset, and the requested word is provided to $Core_0$.

2. $Core_1$ requests a word within block A for reading: The access misses in $L1_1$ and LLC. However, the coherency mechanism informs that the block is at $L1_0$, so the request is forwarded to that cache. According to Figure 4.4, the block is copied to $L1_1$ and both reuse bits are set, as we recognize this access as an LLC reuse.

3. $Core_1$ requests a word within block B for reading: The access misses in $L1_1$ and LLC, and the block is not present in any other private cache, so the request is forwarded to MM. According to Figure 4.4, block B is copied to $L1_1$ (replacing block A) with its reuse bit unset, and the requested word is provided to $Core_1$. According to Figure 4.5, given that block A has its reuse bit set, it is inserted into the LLC.

4. $Core_1$ requests a word within block C for reading: Block C is inserted in $L1_1$ and block B is replaced. As the reuse bit of block B was unset and its tag was not in RD_1, according to Figure 4.5 the tag is stored in RD_1 and, given that the block is clean, it is not inserted in the LLC but just discarded.

5. $Core_1$ requests a word within block B for reading: This access is handled analogously to the previous access.

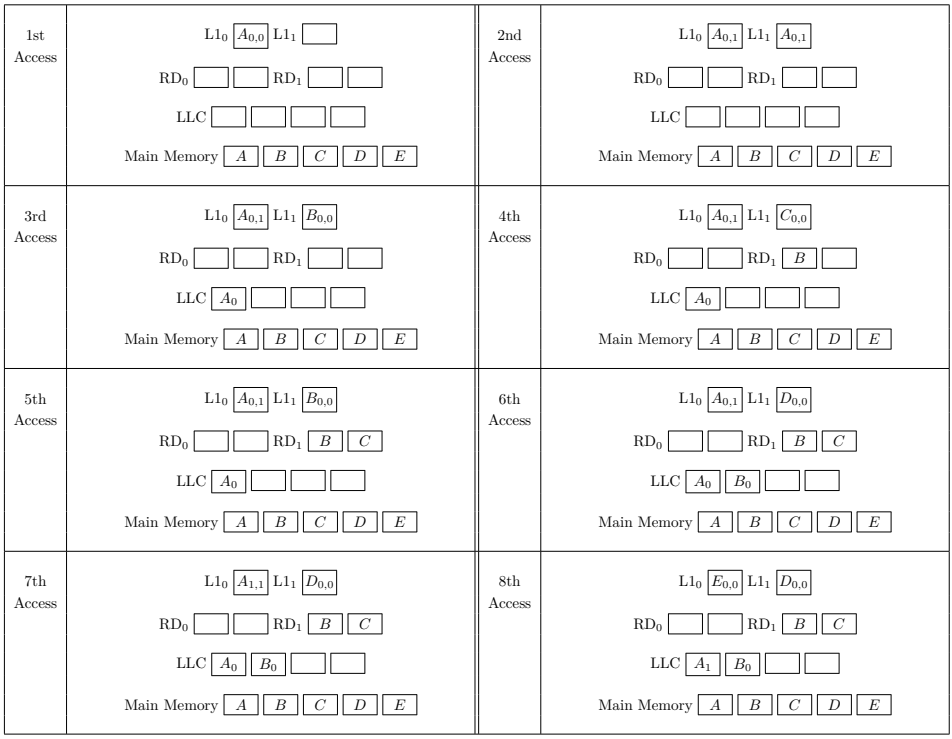


Figure 4.6: Example of the Reuse Detector operation.

6. $Core_1$ requests a word within block D for reading: Block D is inserted in $L1_1$ and block B is replaced. As the reuse bit of block B was unset but its tag was present in RD_1, according to Figure 4.5 block B is inserted in the LLC.

7. $Core_0$ writes to a word within block A: The access hits in $L1_0$. The dirty bit for the block is set.

8. $Core_0$ requests a word within block E for reading: Block E is inserted in $L1_0$ and block A is replaced. As the dirty bit of block A is set and A is already present in the LLC, the block is updated at this level.

4.1.3.3 Implementation details

Although a typical set-associative design could be used for the RD implementation, where a whole block tag, a validity bit and some information related with the replacement policy is included for each line, as in [104], we use two techniques aimed at reducing the required space: sectoring and compression. A sector is a set of consecutive memory blocks aligned to the sector size. Storing sector tags in the RD allows to merge in a single line of the RD the information related with several blocks. Note that for each entry it is necessary to store a presence bit. For example, with a sector comprising 4-blocks, each line is made up of a tag derived from the sector address, a validity bit, some bits storing the replacement state and 4 presence bits.

The compression of the tags is achieved based on the following process: let t be the amount of bits of the full tag and c the amount of bits of the compressed tag, being t>c. We first divide the full tag into several pieces, each of size c (the last piece is filled with *0s* if necessary). Then, we *xor* all the pieces, obtaining the compressed tag. Note that each compressed tag is shared among various sectors, thus false positives are possible when non-reused blocks are delivered to the LLC. This situation does not cause a functional problem, but it may degrade system performance, so the value of c must be chosen carefully.

As for the storage overhead of the RD implementation, i.e. the total amount of extra bits required compared to the baseline, we need the following hardware: Each RD has 1024 sets and 16 ways (our simulations reveal that this value provides similar performance to that of higher associativity values), and a sector size of 2 blocks. Each RD entry requires 14 bits (10 for the compressed tag, 2 for the block presence, 1 for the replacement policy and 1 validity bit) as Figure 4.7 illustrates. Given that the amount of entries in the RD is 8K, the total extra storage required per core is 14 KB, which represents a negligible 1.3% of an 1MB last-level cache.

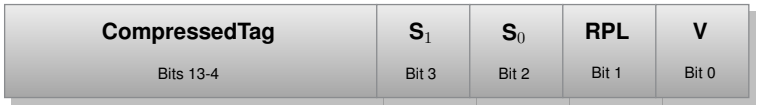


Figure 4.7: Reuse Detector entry.

Finally, as RD replacement policy we use a 1-bit FIFO. Again based on our sim-

ulations, this scheme delivers a similar performance as other policies that would require more storage. In a FIFO policy, age information is updated only when a new address is inserted, and not during subsequent hits. This approach is fully consistent with the main RD buffer goal of detecting the first reuse of a block.

4.1.4 Reuse Detector vs DASCA

In this section we briefly discuss the main differences between the RD approach and the DASCA scheme (explained in Section 2.3.1). As for the operation of both approaches, note that the DASCA mechanism tries to predict dead writes based on a PC-based predictor. For this purpose, the PC signature of each block that accesses the LLC must be recorded. Conversely, the RD scheme tries to predict dead-blocks based on their reuse state. Our prediction is based on the addresses of the accessed data instead of the instruction addresses used in DASCA. Also, in our approach we directly store the mentioned addresses while in the DASCA scheme the authors employ a PC-signatures table which is trained by an auxiliary cache that works in parallel with the conventional cache.

4.2 Experimental framework

For our experiments we use the *gem5* simulator [92] and we employ the *ruby* memory model, specifically the MOESI_CMP-directory coherence policy provided by the simulator, this protocol supports multiple chips and 2-level caches (the 3-level cache support had to be added), also it is a non-inclusive (neither strictly inclusive nor exclusive) hierarchy. It is worth noting that we focus on a MOESI policy since protocols with owned state (e.g. MOESI and MOSI) are able to reduce the number of writes to the LLC, as demonstrated by Chang et al. [78]. We modify the coherence protocol, encoding the proposed reuse detector. We simulate both a single and a multi-core scenario. For the sake of a better accuracy in both scenarios, an O3 processor type (detailed mode of simulation) was used.

The main features of both the processor and the memory hierarchy are shown in Table 4.1. The network used is a crossbar modeled with Garnet [105], a detailed interconnection network model inside gem5. As explained above, for the evaluation of our proposed RDs we implement them in the cache hierarchy modifying the coherence protocol. For modeling the DRAM main memory we use DRAMSIM2 [93].

We adapt the LLC read and write latencies according to the STT-RAM target. Both latencies and energy consumption values are obtained from NVSim [39] for a 1MB (1 bank) cache and are illustrated in Table 4.2. For scaling the LLC to larger sizes, we multiply the leakage power by the number of cores.

Parameter	Value
Architecture	x86
CPUS	1/4/8, 2GHz
Pipeline	8 Fetch, 8 Decode, 8 Rename, 8 Issue/Execute/Writeback, 8 Commit
Registers	Integer (256), Floating Point (256)
Buffers	Reorder Buffer (192), Instruction Queue (64)
Branch Predictor	TournamentBP
Functional Units	IntALU=6, IntMulDiv=2, FPALU=4, FPMultDiv=2, SIMD-Unit=4, RdWrPort=4, IprPort=1
Private Cache L1 D/I	32 KB, 8 ways, LRU repl., Block Size 64B, Access Latency 2 cycles, SRAM
Private Cache L2 D/I	256 KB, 16 ways, LRU repl., Block Size 64B, Access Latency 5 cycles, SRAM
Interconnection	Crossbar network, modeled using Garnet, latency 3 cycles
Shared Cache L3	1 bank/1MB/core, 16 ways, LRU repl., Block Size 64B, R/W Latency 6/17 cycles, STT-RAM
DRAM	2 Ranks, 8 Banks, 4kB Page Size, DDR3 1066MHz
DRAM Bus	2 channels with a 8 bus of 8 bits

Table 4.1: CPU and memory hierarchy specification.

Parameter	**Value**
Hit Latency (ns)	5.61
Miss Latency (ns)	1.75
Write Latency (ns)	16.5
Hit Dynamic Energy (nJ)	0.32
Miss Dynamic Energy (nJ)	0.19
Write Dynamic Energy (nJ)	1.31
Leakage Power (mW)	3.09

Table 4.2: Latencies and energy consumption of an 1MB (1 bank) STT-RAM cache using 22nm technology.

Our experiments make use of the SPEC CPU2006 benchmark suite [91]. When we evaluate our proposal in a single-core scenario (LLC 1MB size) we employ *reference* inputs and simulate 1 billion instructions from the checkpoint determined using PinPoints [96]. Note that results from 4 out of 29 benchmarks are not considered in the evaluation section due to experimental framework constraints. We also report results of 28 multiprogrammed mixes employing SPEC CPU2006 programs in both 4 and 8-CMP systems with 4 and 8MB shared LLC sizes, respectively. In both cases, we fast forward 100M instructions, warm up caches for 200M instructions and then report results for at least 500M instructions per core.

For selecting the aforementioned multiprogrammed mixes, we employ an analogous methodology to the one used in the previous chapter, but, in this case, based on

the amount of writes to LLC instead of the amount of writes to main memory: we execute each benchmark alone, using an LLC of 1MB and without any reuse detector, and we measure the amount of LLC writes that it generates. We then obtain for each benchmark the *number of writes to LLC per 1000 instructions* ratio (WPKI). Based on these values, we include each benchmark into the *high*, *medium* or *low* category. Specifically, the *high* category includes benchmarks with a WPKI higher than 8, the *medium* those with a WPKI satisfying $1 < WPKI < 8$ and finally, in the *low* category we include the programs with a WPKI lower than 1. Table 4.3 shows this classification. Based on this classification, and as we will further detail in Section 4.3, we build some mixes made up of programs with high WPKI, some with medium WPKI, some with low WPKI, and some combining applications from different WPKI categories trying to fill most of the combinations high-medium, high-low, medium-low and high-medium-low. Tables 4.4 and 4.5 show the built mixes for the 4-core and 8-core CMP system, respectively.

High	Medium	Low
lbm, mcf, libquantum, bwaves, milc, cactusADM, zeusmp, lelie3d	bzip2, soplex, gcc, wrf, astar, hmmer, xalancbmk, gobmk, perlbench	gromacs, calculix, h264ref, tonto, omnetpp, namd, sphinx3, GemsFDTD

Table 4.3: Benchmark characterization according to the number of LLC writes per Kinstruction (WPKI).

4.2.1 Energy model

The DRAM energy is obtained directly from the simulator. For computing the LLC energy we employ a model that includes both dynamic and static contributions. The static component is calculated using NVSim [39], which reports the leakage number for 1MB LLC. Thus, we multiply that number by the execution time and the number of cores to obtain the total static energy. In the case of the dynamic component, we again use NVSim for determining the dynamic energy consumption per access to the LLC. Then, we compute the dynamic energy consumption as follows:

$$Dynamic\ Energy = H_{LLC} * HE_{LLC} + W_{LLC} * WE_{LLC} + M_{LLC} * ME_{LLC}$$

Mixes	Applications
H0	cactusADM, libquantum, mcf, leslie3d
H1	cactusADM, bwaves, leslie3d, milc
H2	mcf, milc, zeusmp, libquantum
H3	lbm, leslie3d, bwaves, cactusADM
M0	xalancbmk, gobmk, soplex, bzip2
M1	wrf, xalancbmk, gcc, perlbench
M2	gcc, soplex, hmmer, gobmk
M3	perlbench, gobmk, wrf, astar
L0	GemsFDTD, sphinx3, namd, calculix
L1	omnetpp, gromacs, tonto, h264ref
L2	omnetpp, GemsFDTD, sphinx3, calculix
L3	tonto, namd, gromacs, h264ref
HM0	zeusmp, milc, astar, gcc
HM1	lbm, libquantum, gobmk, leslie3d
HM2	zeusmp, gobmk, gcc, milc
HM3	mcf, soplex, xalancbmk, bwaves
HL0	cactusADM, leslie3d, bwaves, omnetpp
HL1	lbm, omnetpp, bwaves, libquantum
HL2	namd, gromacs, lbm, leslie3d
HL3	GemsFDTD, leslie3d, omnetpp, milc
ML0	sphinx3, tonto, gromacs, perlbench
ML1	hmmer, wrf, h264ref, gromacs
ML2	GemsFDTD, perlbench, wrf, namd
ML3	xalancbmk, soplex, sphinx3, tonto
HML0	hmmer, h264ref, omnetpp, mcf
HML1	milc, GemsFDTD, hmmer, h264ref
HML2	GemsFDTD, milc, bzip2, wrf
HML3	GemsFDTD, leslie3d, xalancbmk, bwaves

Table 4.4: SPEC 2006 multiprogrammed mixes for the 4-core CMP.

where H_{LLC}, W_{LLC} and M_{LLC} denote the number of hits, writes and misses in the LLC respectively, and HE_{LLC}, WE_{LLC} and ME_{LLC} correspond to the energy consumption of a hit, a write and a miss in the LLC respectively.

Mixes	Applications
H0	mcf, milc, zeusmp, cactusADM, leslie3d, lbm, bwaves, libquantum
H1	cactusADM, libquantum, mcf, leslie3d, cactusADM, bwaves, leslie3d, milc
H2	cactusADM, bwaves, leslie3d, milc, mcf, milc, zeusmp, libquantum
H3	mcf, milc, zeusmp, libquantum, lbm, leslie3d, bwaves, cactusADM
M0	xalancbmk, gobmk, soplex, astar, gcc, perlbench, hmmer, wrf
M1	xalancbmk, perlbench, soplex, bzip2, wrf, xalancbmk, gcc, perlbench
M2	soplex, xalancbmk, gcc, perlbench, gobmk, soplex, hmmer, gobmk
M3	perlbench, 445.gobmk, 481.wrf, 473.astar, 481.wrf, 450.soplex, 456.hmmer, 445.gobmk
L0	GemsFDTD, sphinx3, namd, calculix, h264ref, tonto, gromacs, omnetpp
L1	GemsFDTD, sphinx3, namd, calculix, omnetpp, gromacs, tonto, h264ref
L2	omnetpp, gromacs, tonto, h264ref, tonto, namd, gromacs, h264ref
L3	omnetpp, GemsFDTD, sphinx3, calculix, tonto, namd, gromacs, h264ref
HM0	cactusADM, leslie3d, libquantum, mcf, xalancbmk, gobmk, soplex, bzip2
HM1	zeusmp, milc, lbm, mcf, wrf, hmmer, perlbench, gcc
HM2	libquantum, bwaves, milc, cactusADM, soplex, astar, gobmk, perlbench
HM3	leslie3d, lbm, cactusADM, zeusmp, gobmk, soplex, gcc, wrf
HL0	cactusADM, leslie3d, libquantum, mcf, GemsFDTD, sphinx3, namd, calculix
HL1	zeusmp, milc, lbm, mcf, h264ref, tonto, gromacs, omnetpp
HL2	libquantum, bwaves, milc, lbm, GemsFDTD, sphinx3, h264ref, tonto
HL3	leslie3d, lbm, cactusADM, zeusmp, namd, calculix, gromacs, omnetpp
ML0	xalancbmk, gobmk, soplex, bzip2, GemsFDTD, sphinx3, namd, calculix
ML1	astar, perlbench, hmmer, wrf, h264ref, tonto, gromacs, omnetpp
ML2	xalancbmk, gobmk, gcc, perlbench, GemsFDTD, sphinx3, gromacs, omnetpp
ML3	soplex, bzip2, hmmer, wrf, h264ref, tonto, namd, calculix
HML0	cactusADM, leslie3d, bwaves, xalancbmk, gobmk, soplex, GemsFDTD, sphinx3
HML1	libquantum, mcf, lbm, h264ref, astar, perlbench, namd, calculix
HML2	milc, zeusmp, cactusADM, hmmer, wrf, xalancbmk, h264ref, tonto
HML3	mcf, milc, leslie3d, namd, soplex, perlbench, gromacs, omnetpp

Table 4.5: SPEC 2006 multiprogrammed mixes for the 8-core CMP.

4.3 Evaluation

This section evaluates how well RD and DASCA behave when managing an STT-RAM LLC, both in terms of performance and energy consumption of LLC and main memory. Single, four and eight core systems are examined in Sections 4.3.1, 4.3.2, and 4.3.3, respectively.

4.3.1 Evaluation in a single-core scenario

First, we show the number of writes to the LLC that each evaluated proposal involves as well as the performance delivered. Then, we focus on the involved energy consumption in both the STT-RAM and the main memory according to the model detailed in Section 4.2. Finally, we discuss the obtained results. All the

graphs shown in this section report individual data for each benchmark, adding at the right end the geometric mean of all data (labeled as *GMEAN*) and the geometric mean of the eight most write-intensive benchmarks, according to Table 4.3 (labeled as *HIGH*).

4.3.1.1 Write filtering

Figure 4.8 illustrates the number of writes to the STT-RAM LLC generated by the DASCA scheme and our proposal (using a RD of 8K entries) normalized to a baseline system without any write filtering scheme.

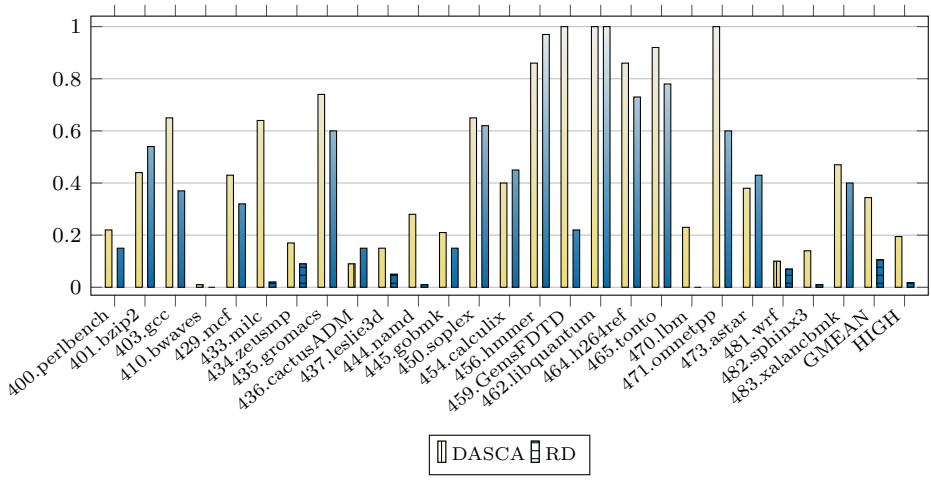


Figure 4.8: Number of writes to the STT-RAM LLC normalized to the baseline: SPEC CPU2006 suite.

As shown, our proposal significantly outperforms DASCA. Notably, in 20 out of 25 benchmarks evaluated the Reuse Detector exhibits higher ability in cutting the write traffic to the STT-RAM LLC. Overall, the block bypassing decisions commanded by RD reduce the number of LLC writes with respect to the baseline system around 90% whereas DASCA just achieves a 65% reduction. In addition, if we zoom just in the 8 programs with the highest WPKI numbers (those labeled as *high* in Table 4.3), RD reduces the number of LLC writes by around 98% with respect to the baseline, while DASCA cuts the write traffic by 80%.

4.3.1.2 Performance

Apart from the goal of decreasing the STT-RAM LLC energy consumption (quantified later in this section), it is clear that energy efficiency should not come at the expense of a performance drop. Thus, to further evaluate the benefits of RD, Figure 4.9 shows the performance (IPC) delivered.

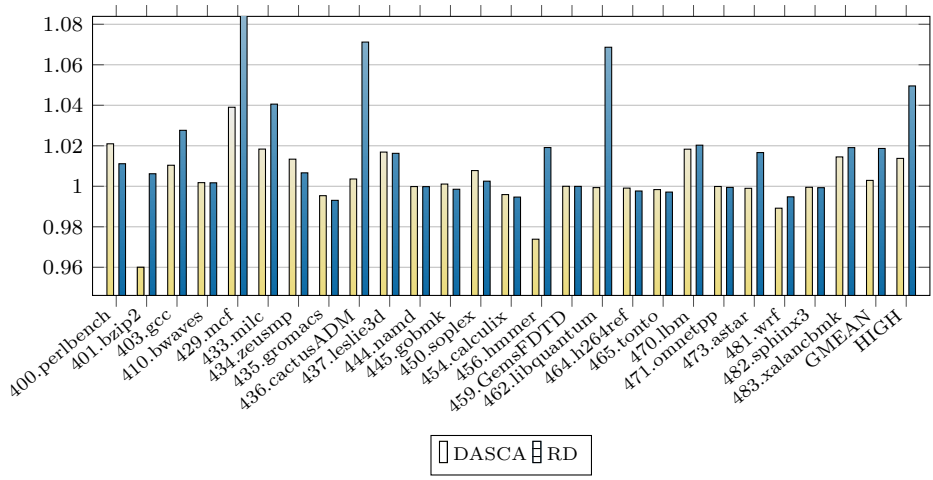


Figure 4.9: Performance (Instructions per Cycle) normalized to the baseline: SPEC CPU2006 suite.

Overall, our scheme performs moderately better than DASCA: RD delivers 1.9% performance improvement compared to the baseline while DASCA just improves IPC by 0.3%. If we focus on the write-intensive applications RD clearly outperforms DASCA, achieving performance improvements of 5% and 1.4%, respectively. This reveals, as we will confirm later in the multi-core environment, that our approach works especially well for those applications for which the amount of writes to the LLC is higher, both in terms of write reduction and performance improvement.

4.3.1.3 Energy savings

Figure 4.10 shows the total energy savings (adding both the dynamic and the static components) in the LLC. Overall, our proposal reports 41% energy reduction compared to the baseline, while DASCA reports 36%. Considering only the

write-intensive programs, the numbers are 65% and 54%, respectively. If we split the total energy savings with respect to the baseline into the dynamic and static parts, our proposal achieves a 60% reduction in the dynamic part considering all the applications (75% for the *high* programs), while DASCA obtains 50% (64% for the *high* benchmarks). As for the static part, RD is able to obtain 2% energy savings (around 5% for the *high programs*), while DASCA just achieves 0.3% (1.4% for the write-intensive applications). Note that avoiding LLC writes reduces dynamic energy, whereas improving performance translates into static energy savings. It is also worth noting that, as Figure 4.11 illustrates, the dynamic energy consumption in the LLC of the baseline system is, for most of the applications evaluated, significantly higher than the static contribution.

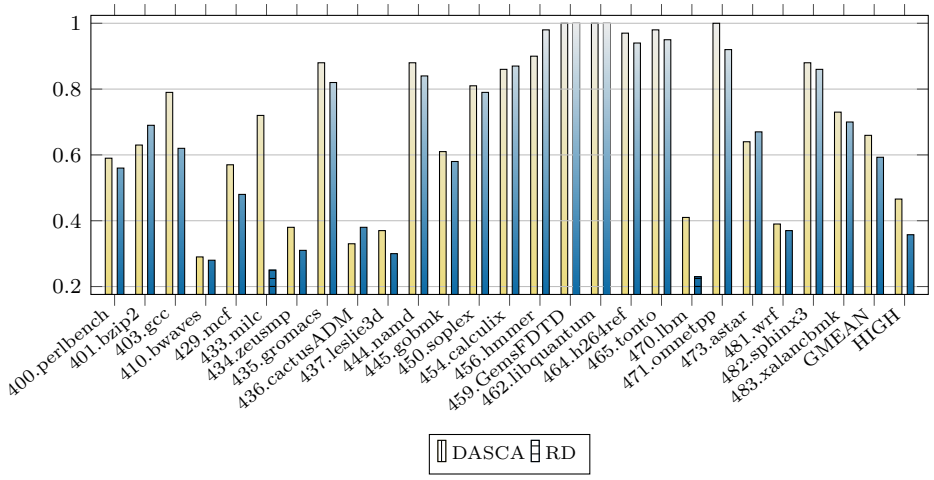


Figure 4.10: Energy consumption in the LLC normalized to the baseline: SPEC CPU2006 suite.

Finally, we have also explored the impact on the energy consumption in the DRAM main memory. In Figure 4.12, we show the results for all the applications. As expected, the DRAM energy reduction follows the trend of performance improvement. Overall, our proposal manages to reduce the DRAM energy consumption by 2.1% (5.3% for the write-intensive programs) with respect to the baseline, while DASCA just improves the memory energy consumption by 0.2% (1.4% for the *high* applications).

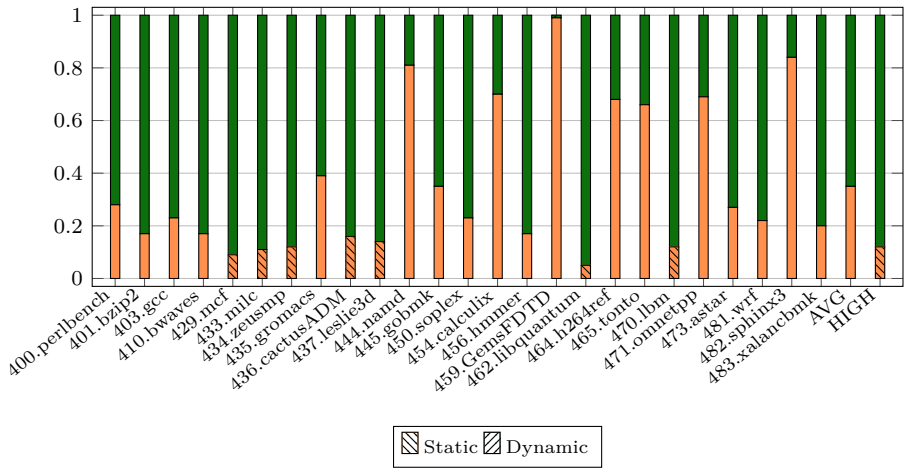


Figure 4.11: Breakdown of energy consumption in the LLC into the static and dynamic contributions for the baseline in the single-core system.

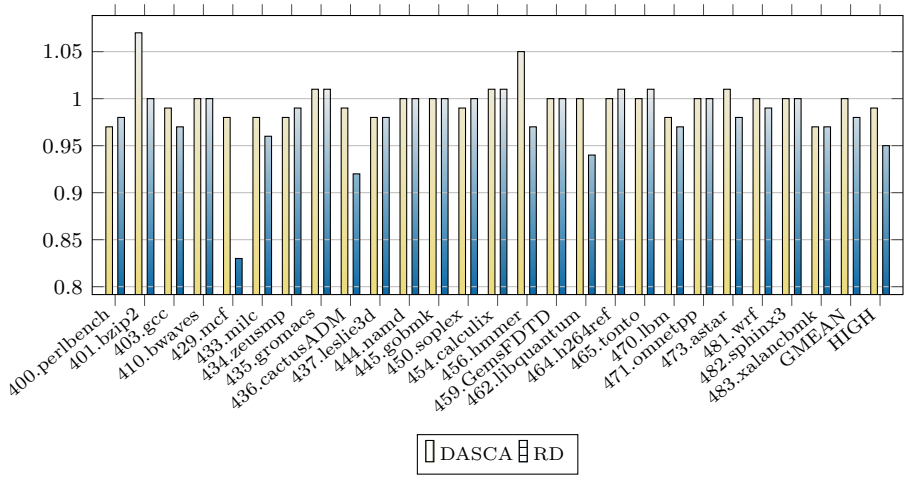


Figure 4.12: Energy consumption in the DRAM normalized to the baseline: SPEC CPU2006 suite.

4.3.1.4 Discussion

If we zoom into specific benchmarks, there are some special cases that deserve further detail to get a deeper insight. Note that globally, the relative trend shown for each benchmark in the amount of writes to the LLC between our approach and DASCA, is quite stable in the energy consumption differences, although modulated

with the relative performance numbers. However, there are some few exceptions such as *namd*, *GemsFDTD* or *omnetpp*, where RD is able to reduce the amount of LLC writes significantly more than DASCA but the energy savings and also the performance improvements obtained by both techniques are almost the same (and quite low compared to the baseline) in all these three cases. The reason is that these programs are three of the four benchmarks that exhibit the lowest values of WPKI, so, although the write reductions that RD achieves in relative terms compared to DASCA is significant for these applications, the corresponding reduction in absolute values are very modest, and therefore the impact on the energy is almost negligible.

Also, in other applications such as *mcf*, *cactusADM* or *hmmer*, our approach is able to report IPC numbers significantly higher than those of DASCA, while both techniques exhibit quite similar write reduction capabilities. In order to explain that, first note that there are many different aspects involved in the system performance delivered. Among others, one key aspect is that reducing the amount of writes to the LLC is not sufficient in itself to guarantee performance improvements: although the main goals when bypassing blocks from the LLC to main memory are both to save energy and improve performance by increasing the hit rate in the LLC, obviously the bypassing may fail in the sense that a bypassed block could be referenced again soon, leading to a LLC miss and even a performance drop with respect to the case where bypassing is not carried out. Thus, for all these three benchmarks, the experimental data reveal that with our proposal the amount of hits in the LLC clearly outperforms both the baseline and the DASCA mechanism. Notably, the amount of LLC hits experienced in the *cactusADM* and *mcf* programs are 7.23x and 2x the values obtained in the baseline, while DASCA obtains 1.89x and 0.89x, respectively. Also, the amount of misses in the LLC is lower than in the baseline and DASCA, with values ranging between 0.77-0.87x those obtained in the baseline. Considering all the evaluated benchmarks, RD is able to augment the amount of hits around 7% with respect to the baseline (40% if we only consider the write-intensive applications) while DASCA experiences no increment when considering all the benchmarks and 23% for the *high* applications.

At a first glance, the behavior of the *libquantum* application may seem somehow strange: Neither RD nor DASCA are able to significantly reduce the amount of writes to the LLC; however this benchmark running under RD reports a performance improvement of 7% with respect to the baseline, while performance remains

largely unchanged under DASCA. In addition, and as one would expect since the number of bypasses is low, the number of hits in the LLC is practically the same in the three policies. The reason to explain the performance improvement lies in the LLC bank contention due to the write activity: this application is by far the most stalled one due to write contention. Thus, although the write reduction is very limited with our scheme, it is enough to reduce stalls with respect to the baseline by around 8%, which in absolute numbers implies a huge number of this kind of situations avoided, which leads to the performance improvement obtained.

Conversely, although other benchmarks such as *gromacs*, *calculix* or *wrf* exhibit moderate LLC write reduction with RD and DASCA, they all perform worse than in the baseline. For these three programs the amount of hits experienced in the LLC is, in RD and DASCA, lower than in the baseline, which suggests that the bypassing performed is not efficient for these benchmarks. Recall that the energy savings achieved in the LLC as a consequence of the reduction in the number of writes performed in this cache level may be partially offset with the performance drop derived from the increment in the amount of LLC misses, as in these three programs occurs. Note also that, although the write operations are outside the critical path, the performance improvement derived from avoiding the long write operations may be mitigated if bank contention exists between the writes effectively performed.

4.3.2 Evaluation in a 4-core CMP system

In this section we extend the single-core analysis performed in the previous section to a more up-to-date environment: a multi-core scenario where the LLC is shared among different cores. For this purpose, we measure again the number of writes to the shared LLC (4MB), the performance and the energy consumption in both the STT-RAM LLC and the DRAM main memory for RD and DASCA, and report results normalized to the baseline. However, as done in the Chapter 3, we employ in this scenario the geometric mean of the number of writes and energy consumption (per application) *divided by the total number of instructions executed*.

We employ 28 mixes made up of applications from the SPEC CPU2006 suite chosen accordingly to the WPKI categories illustrated in Table 4.3. First, we randomly compose three groups of 4 mixes each made up of applications belonging to just one WPKI category (mixes referred to as Hi, Mi and Li for high, medium and

low WPKI respectively). Then, we build 16 more mixes merging applications with WPKI corresponding to different categories and trying to construct them in a balanced and homogeneous fashion. Again, the workload name encodes the WPKI categories of the applications. For example, HL2 is the third mix we build consisting of some applications with high WPKI and some applications with low WPKI. The detailed mixes are illustrated in Table 4.4. Most graphs in this section report results considering all the mixes (GMEAN), just the 4 H*i* most write-intensive mixes (HIGH), the 4 H*i* and the 4 HM*i* mixes together (H+HM), the 4 H*i*, the 4 HM*i* and the 4 HML*i* mixes together (H+HM+HML) and all the mixes including a high program (SomeH).

4.3.2.1 Write filtering

Figure 4.13 illustrates the number of writes to the STT-RAM LLC generated when using DASCA and an 8K-entry RD per core normalized to a baseline STT-RAM without any write reduction mechanism.

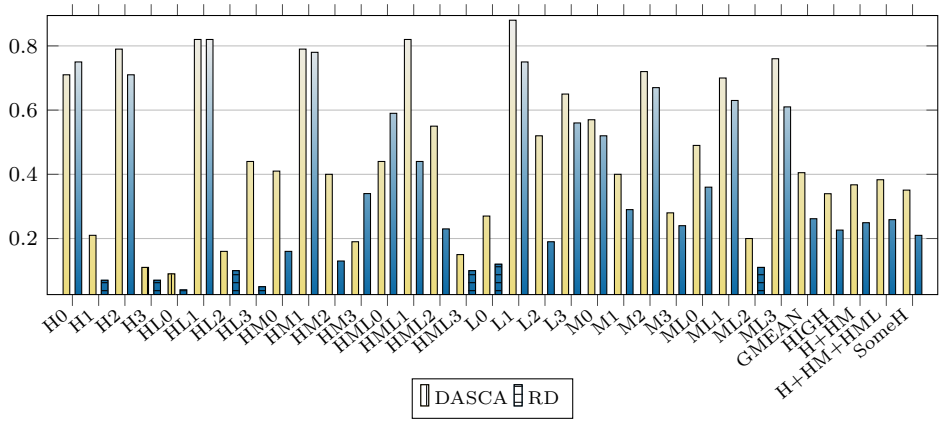


Figure 4.13: Number of writes to the STT-RAM LLC normalized to the baseline in a 4-core CMP system.

The experimental results reveal that RD exhibits a significantly greater ability to decrease the amount of writes to the LLC than DASCA. Notably, in 25 out of the 28 mixes evaluated RD outperforms DASCA. Overall, the number of writes in the baseline system gets reduced to 26% by using RD, in contrast with DASCA where it gets reduced to just 40%. As for the write-intensive mixes (HIGH) the

RD and DASCA make around 22% and 34% of the writes the baseline performs, respectively.

4.3.2.2 Performance

In order to evaluate performance when executing multiprogrammed workloads, we analyze, as done in the Chapter 3, the *Instruction Throughput* (IT) metric. The IT metric is defined as the sum of all instructions committed per cycle in the entire chip ($\sum_{i=1}^{n} IPC_i$, being n the number of threads). Figure 4.14 illustrates the IT that each evaluated policy delivers normalized to the baseline.

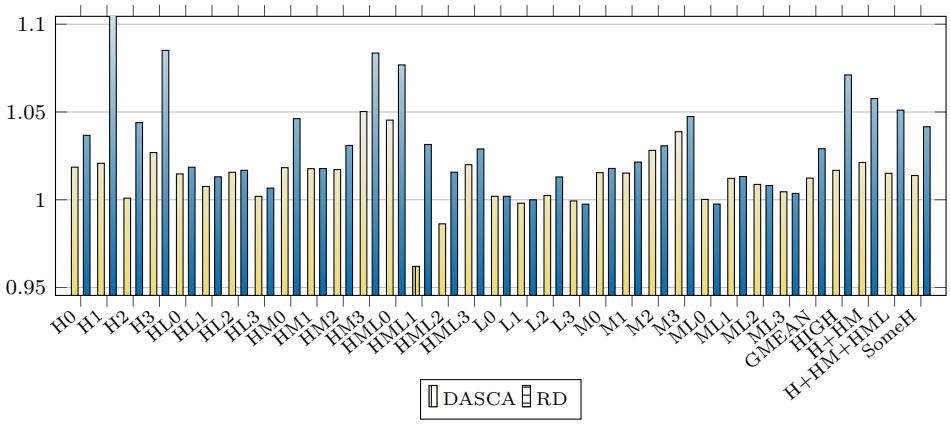


Figure 4.14: Instruction throughput normalized to the baseline in the 4-core CMP system.

As shown, RD moderately outperforms DASCA. This is a key contribution of RD, since our approach, managing to reduce the amount of writes to the shared LLC to a greater extent than DASCA, is also able to deliver higher performance (which also allows to report higher energy savings in both the LLC and the main memory as shown later). The data reveal that, overall, RD improves performance by around 3% compared to the baseline, while DASCA just improves it by around 1.2%. Moreover, we can observe that, in almost all of the 28 mixes evaluated (except mainly those mixes made up of benchmarks with a reduced WPKI, those labeled as *low*, where the performance of both techniques essentially matches that of the baseline), our technique performs better. Zooming into particular mixes, the results reveal that RD performs especially better than DASCA in those mixes

made up of write-intensive applications. Thus, our approach reports a performance improvement of more than 7% when considering just the H_i mixes while DASCA just reports 1.7% IT improvement with respect to the baseline. Also, RD delivers significantly higher performance than DASCA and the baseline for those mixes which contain any application with high WPKI.

4.3.2.3 Energy savings

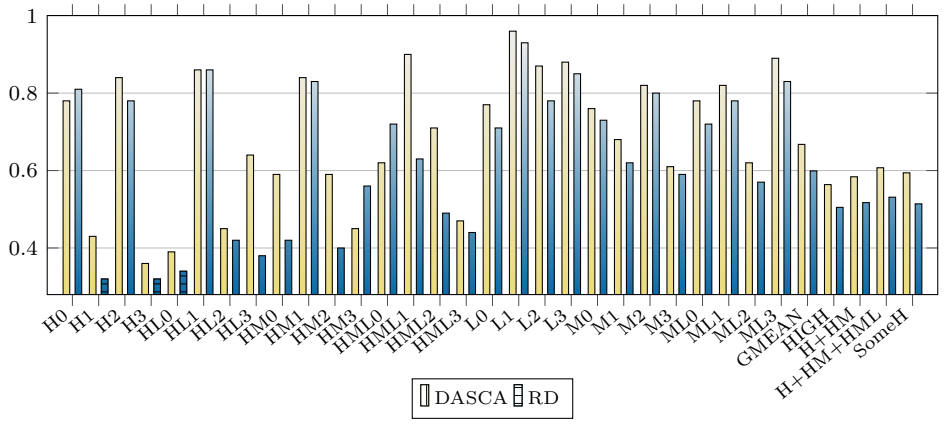


Figure 4.15: Energy consumption in the LLC normalized to the baseline in the 4-core CMP system.

Figure 4.15 illustrates the energy savings in the LLC. As in the single-core scenario, the graph follows a similar relative trend between our approach and DASCA to the one observed in the write reduction numbers (Figure 4.13), just slightly modulated with the performance numbers since, as shown in Figure 4.16, the dynamic contribution to the energy consumption in the LLC is higher than the static part (except in the mixes made up of applications with low WPKI only), so that the ability to reduce the amount of writes to the LLC (dynamic contribution) impacts the total energy consumption more than the ability to improve performance, which mainly affects the static contribution. Overall, our proposal reports around 40% energy reduction in the STT-RAM LLC compared to the baseline while DASCA reduces it by around 33%. If we zoom into the write-intensive mixes, RD and DASCA are able to save around 50% and 43% of LLC energy consumption, respectively. If we break the LLC energy numbers down into the static and dynamic contributions, our results reveal that, overall, RD is able to reduce –considering all mixes– the

static energy part by around 2.7% with respect to the baseline (around 6% for the write-intensive mixes) while DASCA reduces the static contribution by 1.2% (1.7% for the *high* mixes). In addition, our approach reports dynamic energy savings of around 56% (61% for the *high* mixes) while DASCA numbers are 46% (53% for the *high* mixes).

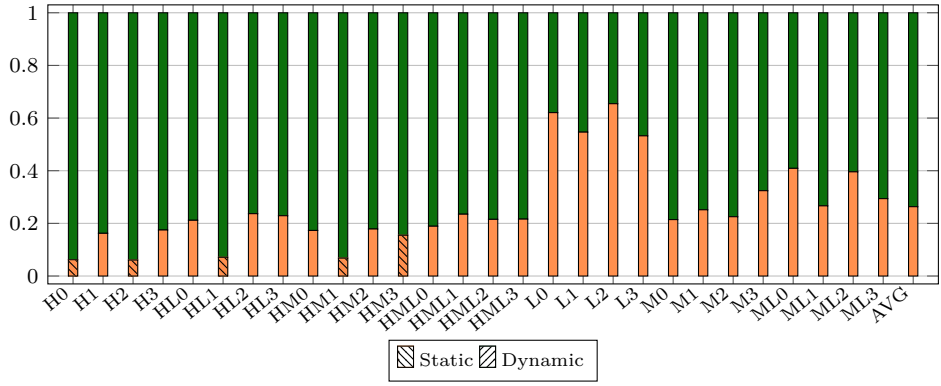


Figure 4.16: Breakdown of energy consumption in the LLC into the static and dynamic contributions for the baseline in the 4-core CMP system.

Also, we explore the energy savings obtained in the DRAM main memory, where the leakage contribution has far greater significance than in the STT-RAM LLC, so that the trends obtained essentially follow those of the IT graph, but inverted (higher performance translates into lower DRAM energy consumption). Figure 4.17 illustrates that RD manages to additionally reduce the energy consumption of the main memory by around 6.3% on average compared to the baseline (8.3% for the write-intensive mixes), while DASCA barely reaches a 4% energy reduction (around 2% for the *high* mixes), mainly due to the higher performance improvement that our proposal exhibits.

4.3.2.4 Discussion

For the sake of clarity, we next explain where the performance improvements of our technique come from. First, as Figure 4.13 illustrated earlier, the write reductions to the LLC that RD achieves are greater than those of DASCA. Second, and more importantly, as Figure 4.14 reveals, the bypasses dictated by RD translate into a higher performance than those of DASCA. As in the single-core scenario, the

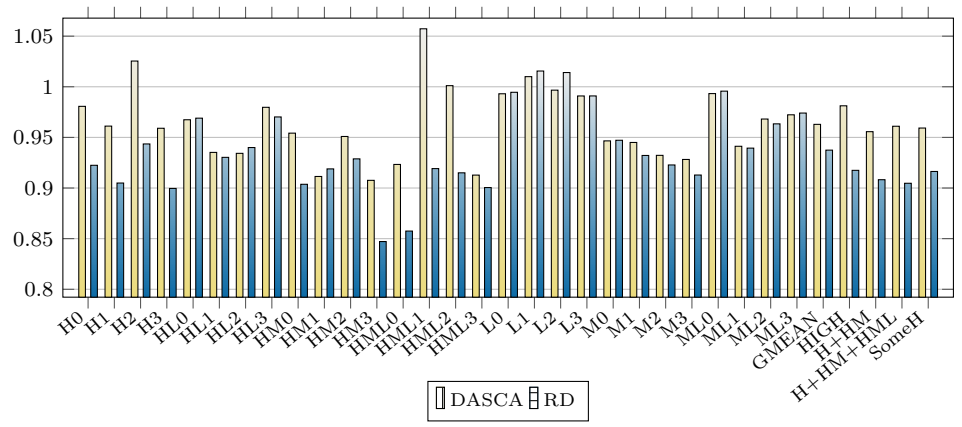


Figure 4.17: Energy consumption in the DRAM normalized to the baseline in the 4-core CMP system.

rationale behind that is related with the hit rate experimented in the LLC with both schemes. Figure 4.18 illustrates the amount of hits in the LLC per kilo instruction that each mix experiences normalized to the baseline.

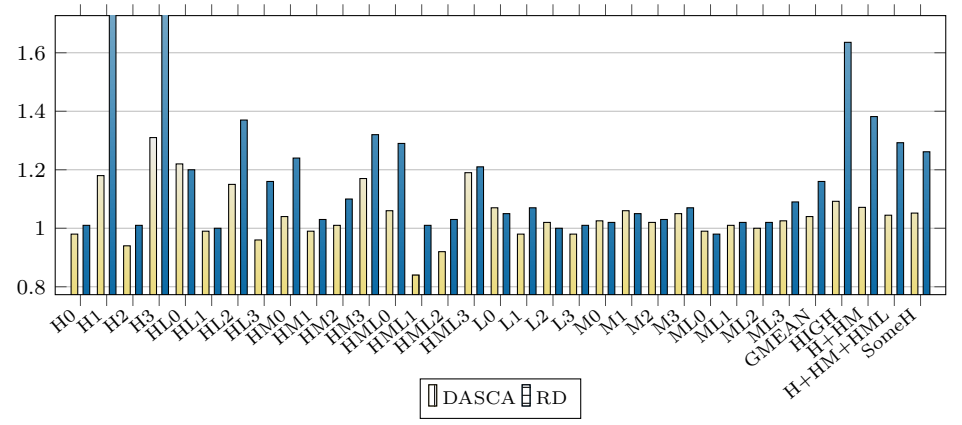


Figure 4.18: Amount of LLC hits per kilo instruction normalized to the baseline in the 4-core CMP system.

The results reveal that in most of the mixes evaluated the amount of hits in the LLC is higher under our approach than using DASCA. Again, this is especially evident for the case of the mixes including write-intensive applications such as H1, H3 and HL2 where the number of hits is 2.87x, 2.45x and 1.37x those of the baseline, respectively. This is the key to explain our performance improvements:

the efficient management of the LLC contents by exploiting the reuse locality. In addition, there are other factors that also contribute to the throughput gain such as less write operations to the LLC, less main memory accesses, and increased row buffer hit rates. In order to perform a deeper comparison between RD and DASCA, Table 4.6 recaps the average values of different metrics involved in the performance delivered by both techniques, normalized to those of the baseline. As shown, our scheme improves DASCA and the baseline (especially in the data from write-intensive mixes) in all the metrics considered.

Metrics / Policies	LLC Misses	LLC Hits	Row buffer Read Hit Rate	DRAM reads	DRAM Writes	Bank contention in LLC
DASCA (All/High)	1.00/1.05	1.04/1.09	1.03/1.00	1.00/1.05	1.03/1.06	0.37/0.15
RD (All/High)	0.95/0.93	1.16/1.63	1.04/1.01	0.97/0.98	0.95/0.94	0.21/0.08

Table 4.6: Average (geomean) values of different metrics normalized to the baseline in the 4-core CMP system.

As in the single-core scenario, next we zoom into particular mixes that need further detail to get a better understanding. First, in some mixes such as H0, HM3 or HML0, we can observe that the DASCA scheme is able to reduce the amount of writes to the shared LLC and also the energy consumption in the STT-RAM more than our scheme does (Figures 4.13 and 4.15). Conversely, the RD manages to deliver a higher throughput than DASCA (Figure 4.14). However, these performance improvements our approach achieves are not enough to offset the higher energy savings in the LLC that the DASCA scheme reports for these mixes as a consequence of the lower number of writes to the STT-RAM.

Second, data for mix L2 reveal that the RD is able to reduce the amount of writes to the LLC much more than DASCA with respect to the baseline (81% vs. 48%). However, this great difference translates into just 22% of energy savings in RD vs. 13% of DASCA. As shown, the difference between both policies has been significantly reduced due to the low contribution of the dynamic energy to the total energy consumption in the LLC that this mix exhibits, as Figure 4.16 illustrates.

4.3.2.5 Sensitivity to Reuse Detector size

The RD size is a key design aspect of our proposal. In order to evaluate its impact we show in Figure 4.19 the amount of writes to the LLC, the Instruction Throughput and the energy consumption in both the LLC and the main memory for different RD sizes per core, namely 8K, 16K, 32K and 64K entries.

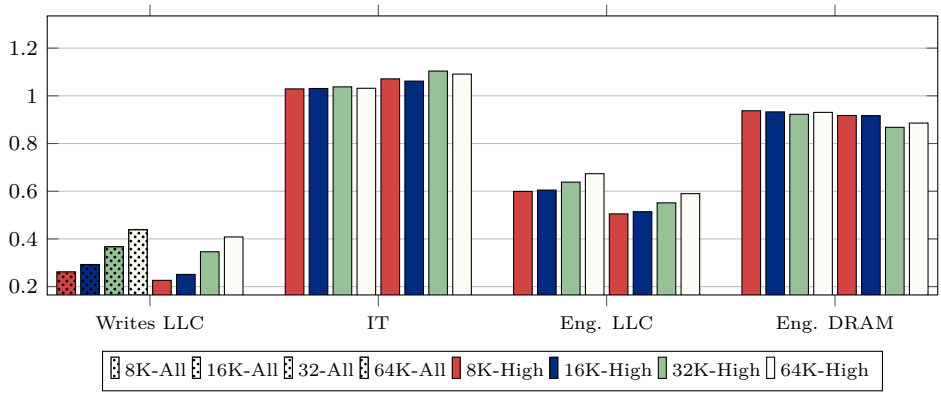


Figure 4.19: Writes to LLC, IT and energy consumption in both LLC and main memory normalized to the baseline for different RD sizes per core in the 4-core CMP system.

As shown, the major impact is observed on the capability to reduce the number of writes in the LLC, ranging from an average reduction of 74% with respect to the baseline when an 8K-entry RD per core is employed (78% for the write-intensive mixes) to a reduction of around 56% for a 64K-entry RD per core (60% for the *high mixes*). Note that maybe these data might appear contradictory at first sight. However, they are not: As the size of RD increases, it also augments the probability that a block finds its tag in the RD, so the probability of bypassing decreases, leading to a minor reduction of writes to the LLC. We can also observe a moderate impact on the average energy consumed in the LLC, with values in the range 60-67% as the size of RD gets increased: again, note that these numbers follow a similar trend to those exhibited by the amount of writes. Finally, the impact on the performance and the energy consumption of the main memory is highly reduced, falling the average IT variations into a small range of 1% (4% for the write-intensive mixes) and the average DRAM energy variations into a range of 1.5% (5% for the write-intensive mixes).

4.3.2.6 Overhead analysis

In Section 4.1.3.3 we outlined that an 8K-entry RD for a 1MB LLC requires an extra storage of 14KB, which represents a 1.37% overhead with respect to the LLC size. In this section we previously noted that for the 4-CMP system under evaluation (4MB shared LLC) we employ an 8K-entry RD per core. The reason is that we are maintaining for each evaluated system the same overhead (1.37%) with respect the LLC size. Therefore, in the 8-CMP evaluated later, we also employ an 8K-entry RD per core. Hence, the total extra storage (overhead) of RD is 56KB and 112KB for the 4-CMP and 8-CMP systems respectively, representing in all cases a 1.37% overhead with respect to the LLC size.

4.3.2.7 RD in a two-level cache hierarchy

We have evaluated the operation of our proposal in a three-level cache hierarchy since most current multi-core processors employ this configuration. Furthermore, two private levels are more likely to filter more temporal locality than just one private level. However, for a fair comparison, we have also evaluated our proposal and the DASCA scheme in a configuration with just two cache levels. Notably, we reproduce the same configuration (4-CMP) used by the authors in [76] when presenting the DASCA technique (32 KB IL1 and DL1 as private caches and a 1MB per core shared L2 cache). Table 4.7 illustrates the main results.

Policies \ Metrics	Writes LLC	Instr. Throughput	Energy consumpt. LLC	Energy consumpt. DRAM	Hits LLC
DASCA (All/High)	0.67/0.79	1.01/1.03	0.81/0.85	0.97/0.96	0.98/1.01
RD (All/High)	0.55/0.75	1.03/1.04	0.74/0.82	0.95/0.95	0.98/1.00

Table 4.7: Average (geomean) values of different metrics normalized to the baseline in the 4-core CMP system with two cache levels.

As shown, RD maintains a higher capability than DASCA (around 12-14% higher) in reducing the amount of writes to the LLC. However, as expected, the amount of writes avoided (and also the hits experienced in the LLC) is significantly lower than the one exhibited in an scenario with 3 cache levels. Recall that this is due to the fact that, with two cache levels only, most temporal locality has not been filtered by the first level, so the last-level should not only seek to exploit

reuse locality. Also, as a consequence of this lower capability in cutting the write traffic to the LLC, the energy savings achieved in the shared L2 are significantly lower than those obtained with three cache levels, although RD still reports better numbers than DASCA. Finally, RD again improves the Instruction Throughput to a greater extent than DASCA, and consequently also delivers higher energy savings in the main memory. Note that we have also evaluated 28 mixes in this configuration following the same criteria explained earlier, but they are not exactly the same as in the three-level cache hierarchy experiments since the WPKI values that the benchmarks exhibit do not match those of the three-levels configuration and therefore some programs changed the category (high, medium or low) in which they were classified.

4.3.3 Evaluation in an 8-core CMP system

In this section we illustrate and analyze the main results obtained when using RD and DASCA in an 8-core CMP system with an 8MB shared LLC. Like in the previous section, in this scenario we create 28 mixes following the same criteria as in a 4-CMP system. The mixes evaluated are shown in Table 4.5. In this section we will not zoom into the details as we did in the previous section (Section 4.3.2), but will only describe the average results and main trends. Given that a detailed analysis of the 8-core system would show similar results as the 4-core scenario, in this section we will not zoom into details but will only describe the average results and main trends.

4.3.3.1 Write filtering

Figure 4.20 illustrates the number of writes to the STT-RAM LLC generated with DASCA and with RD (assuming an 8K-entry RD per core). Both schemes are normalized to a baseline STT-RAM without any content selection mechanism.

Similarly to the results for the 4-core scenario, the experimental results reveal that RD just performs 41% of the writes in the baseline scheme, whereas DASCA produces 52% of the writes that the baseline did. For the write-intensive mixes, RD and DASCA reduce the amount of writes compared to the baseline in 44% and 35% respectively.

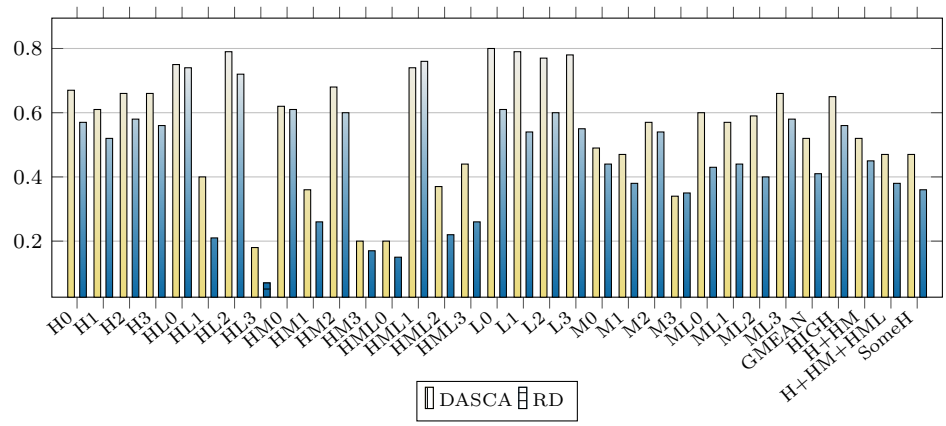


Figure 4.20: Number of writes to the STT-RAM LLC normalized to the baseline in the 8-core CMP system.

4.3.3.2 Performance

As we did in Section 4.3.2.2, we employ the *Instruction Throughput* (IT) to evaluate the performance when executing multiprogrammed workloads. Figure 4.21 illustrates the IT that each evaluated policy delivers normalized to the baseline.

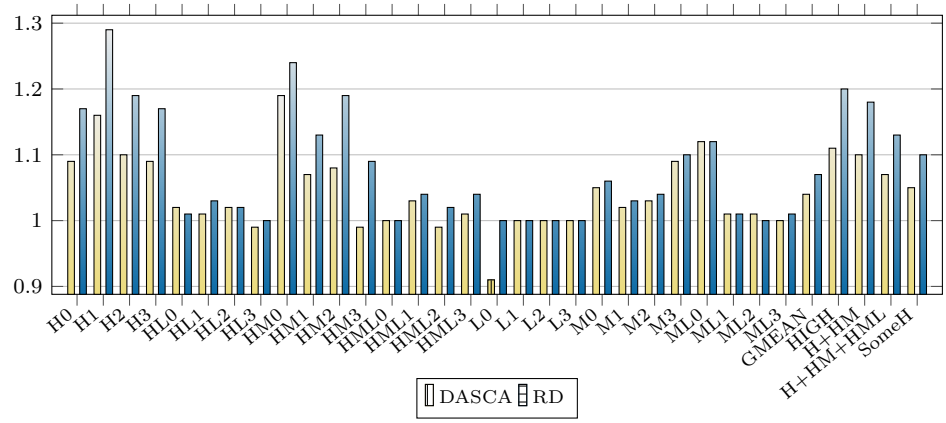


Figure 4.21: Instruction throughput normalized to the baseline in the 8-core CMP system.

Similarly to the results obtained for a 4-core CMP system, RD outperforms DASCA in the 8-core scenario. Moreover, in the 8-core scenario, higher performance improvements are achieved in both schemes over the baseline. The results reveal

that RD improves performance by around 7% compared to the baseline, while DASCA improves it by around 4%. As for write-intensive mixes, RD improves the baseline by 20% and DASCA by 11%. As shown in Figure 4.21, RD significantly overcomes DASCA and the baseline scheme in those mixes which contain any application with high WPKI.

4.3.3.3 Energy savings

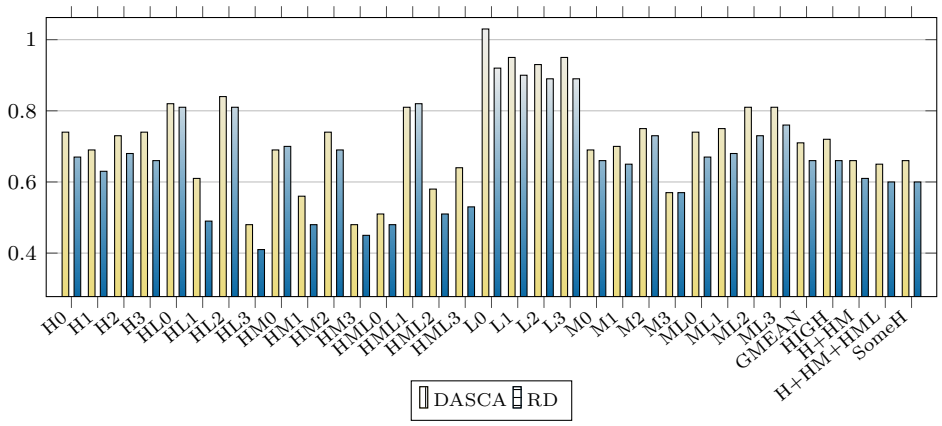


Figure 4.22: Energy consumption in the LLC normalized to the baseline in the 8-core CMP system.

Figure 4.22 illustrates the energy savings in the shared LLC. In general, the results in the 8-core scenario follow the trend observed for the 4-core environment. Specifically, RD reports around 34% energy reduction in the STT-RAM LLC compared to the baseline while DASCA reduces energy by around 29%. In the case of write-intensive mixes, both RD and DASCA reduce the LLC energy consumption by 34% and 28%, respectively. Analyzing the static and dynamic contributions on the LLC energy consumption, overall, RD is able to reduce –for all mixes– the static energy part by around 6% with respect to the baseline (around 15% for the write-intensive mixes) while DASCA reduces the static contribution by 4% (10% for the *high* mixes). In addition, our approach reports dynamic energy savings of around 45% (36% for the *high* mixes) while DASCA numbers are 38% (30% for the *high* mixes). Note that mixes made up of applications with low WPKI exhibit the lowest energy savings across the board. This is consistent with the modest write reduction they report and especially with the high contribution of the static

part to the total LLC energy consumption that they exhibit, as Figure 4.23 shows.

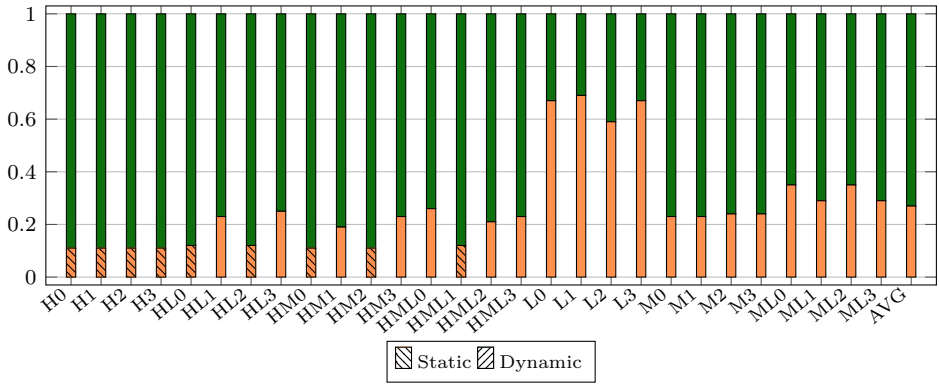


Figure 4.23: Breakdown of energy consumption in the LLC into the static and dynamic contributions for the baseline in the 8-core CMP system.

Figure 4.24 illustrates the energy savings obtained in the DRAM main memory, where it is shown that RD reduces the energy consumption of the main memory by around 6% on average compared to the baseline (3% for the write-intensive mixes), while DASCA reaches a 3% energy reduction and actually wastes more energy, around 6%, for the *high* mixes. This energy waste may look surprising, given that DASCA is able to reduce the number of writes with respect to the baseline in more than 35% and to deliver a performance improvement higher than 10%. However, this can be explained by the fact that DASCA suffers a very significant increase in the amount of LLC misses, which translates into high values of DRAM accesses (as shown in Table 4.8 below).

4.3.3.4 Discussion

As in the 4-core configuration, in this section we explain the reasons for the higher performance improvement achieved in our technique (RD) against DASCA in the 8-core scenario.

As we already reasoned in the previous section, the better performance of RD is due to several factors, being the most important one the high efficiency achieved from the reuse locality exploitation. For demonstrating that fact, Figure 4.25 shows the amount of hits in the LLC per kilo instruction that each mix experiments

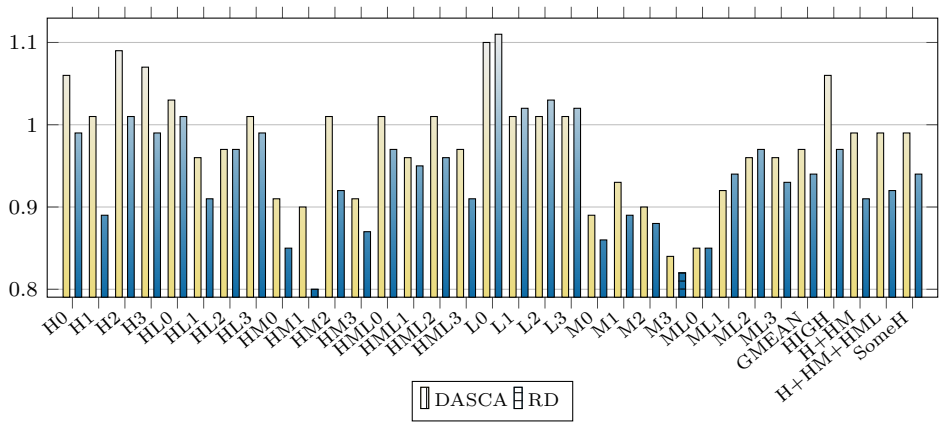


Figure 4.24: Energy consumption in the DRAM normalized to the baseline in the 8-core CMP system.

normalized to the baseline. As the figure shows, our approach achieves in most mixes a higher or much higher number of hits than DASCA, which confirms that RD uses a more efficient policy than DASCA.

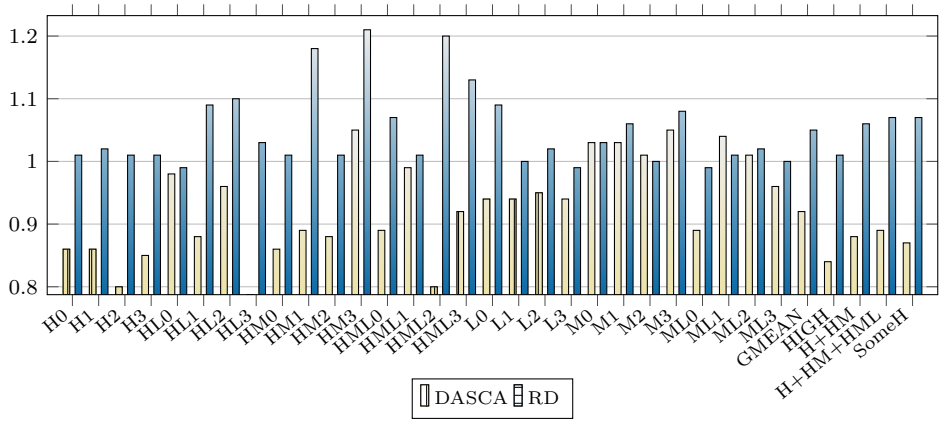


Figure 4.25: Amount of LLC hits per kilo instruction normalized to the baseline in the 8-core CMP system

In addition to the hit rate improvement, there are other metrics that also justify achieving a better performance, such as LLC misses, DRAM reads and writes, row buffer read hit rate and bank contention in the LLC. All these metrics are shown in Table 4.8, for both RD and DASCA and also for both all and write-intensive mixes. Note that the RD beats DASCA in all the metrics considered.

Policies \ Metrics	LLC Misses	LLC Hits	Row buffer Read Hit Rate	DRAM reads	DRAM Writes	Bank contention in LLC
DASCA (All/High)	1.07/1.30	0.92/0.84	1.00/0.99	1.07/1.30	1.08/1.21	0.32/0.13
RD (All/High)	0.98/0.96	1.05/1.01	1.02/1.04	1.00/1.06	1.02/1.05	0.19/0.07

Table 4.8: Average (geomean) values of different metrics normalized to the baseline in the 8-core CMP system.

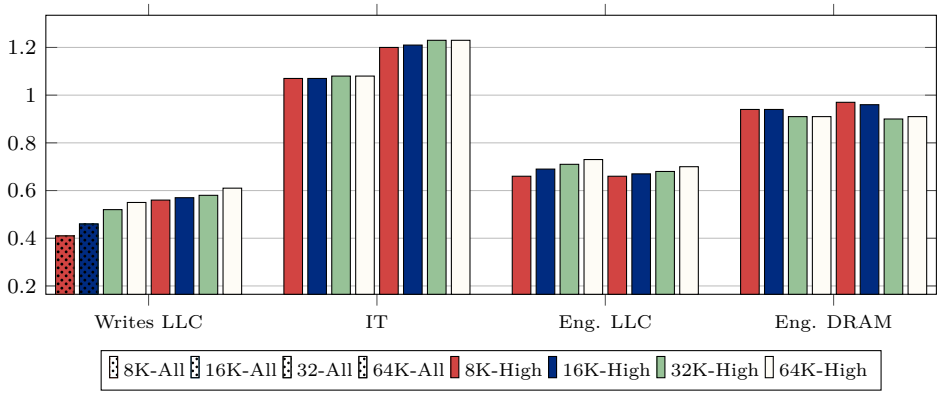


Figure 4.26: Writes to LLC, IT and energy consumption in both LLC and main memory normalized to the baseline for different RD sizes per core in the 8-core CMP system.

4.3.3.5 Sensitivity to Reuse Detector size

Given that the RD size is a determining factor in our proposal, and as done in the 4-CMP system, in Figure 4.26 we show the amount of writes to the LLC, the Instruction Throughput, and the energy consumption in both the LLC and the main memory for different RD sizes per core, namely 8K, 16K, 32K and 64K entries.

The trends are very similar to those observed in the 4-core scenario. Notably, a high impact is observed on the capability to reduce the number of writes in the LLC, especially for the All mixes, whereas a moderate (or even negligible in some cases) impact is seen on the average energy consumed in that cache level or main memory and performance of the overall system.

4.4 Generation of Miss-Rate Curves (MRCs)

Along this thesis we have acquired a vast experience about the operation of the memory system available in current computers, the constraints that it presents and the potential enhancements that future systems may include (from better organization policies to different technologies). Using all this knowledge, and learning some new concepts about the Linux kernel internals and about a recent feature introduced in Intel processors known as Cache Monitoring Technology (CMT), in this section we develop a methodology for building the Miss-Rate Curve (MRC) of an application. The MRC reports an application's cache occupancy on a given cache level (usually the shared LLC in a multi-core scenario) vs. a certain related performance metric, such as the number of Misses Per Kilo Instructions (MPKI).

Several mechanisms have been proposed for building these curves [106], [107], [108], [109], but they all pose different limitations, such as requiring hardware support or relying on code instrumentation. For example Qureshi et al. [106] propose to estimate the distance from the LRU position of the accessed tag to the top of the LRU stack (stack distance) and, based on the stack distance, speculate about the access result (miss or hit) in function of the cache size. However, for that purpose, they need to add counters to each position of the LRU stack for some sets of the LLC, something not available on nowadays designs. Or, as another example, in [110], [111] the authors propose to instrument all memory accesses and record them in a trace log. Then, from this trace log, it is possible to estimate the stack distance. Note that this approach is very limited due to the overhead that such instrumentation adds to execution.

Our proposed methodology is inspired by several previous works:

1. PMCTrack [112], [113], [114], an OS-oriented PMC (Performance Monitoring Counter) tool for the Linux kernel, that simplifies the collection of PMC application data from both the userspace and at the OS-level.

2. Intel's Cache Monitoring Technology (CMT) [115], [116], a new feature introduced in the Intel Xeon E5 2600 v3 product family. This feature allows an operating system or a Hypervisor/Virtual Machine Monitor (VMM) to determine the current last-level cache (LLC) usage of the various applications running on the platform. It currently monitors the L3 cache, which is

the LLC in most server platforms. Intel processors expose the CMT facilities to the system software via a set of Model Specific Registers (MSRs) [117].

Overall, our technique works as follows: By using PMCTrack we periodically gather the MPKI and by making use of Intel's CMT we collect the LLC occupancy of the co-running applications, thus obtaining different discrete MRC points. Note, however, that when several applications share a cache, they may reach an equilibrium state in the distribution of the cache. In that case, in order to obtain points in the whole range of cache sizes, we slow down co-runner applications by applying duty-cycle modulation techniques to the cores where they run. This allows other applications to increase their occupancy, which in turn, makes it possible for us to explore different MPKI values for the whole cache size range. Then, when enough points have been collected, we apply regression analysis to obtain the whole MRCs for the applications.

Figure 4.27 illustrates two examples of curves obtained with this technique, employing the experimental framework depicted in Table 4.9. Notably, the MRC points obtained by using our technique are represented as blue dots in the figure and the continuous red line is derived from a regression analysis of the MRC points. The MRC for the *lbm* application shows a steep MPKI fall for small cache occupancy values and then it saturates from a certain cache size point on. The MRC for the *omnetpp* program, in contrast, shows a linear MPKI drop for the whole range of cache sizes.

Platform	Superserver SYS-6018R-MTR Supermicro
Processor Model(s)	Intel Xeon E5-2695 v3 @ 2.3GHz
Core count	14
Last-Level Cache	35MB (L3)
Main memory	32GB DDR4 @ 2133 MHz

Table 4.9: Features of the evaluated platform

Finally, note that MRCs can be used for different purposes, such as to improve the distribution of a shared cache among threads [106], [107] or to adapt the cache size to reduce energy [118]. For example, in [107] the authors propose a complete methodology for estimating the MRCs in a multiprogrammed environment and, based on that methodology, partitioning the shared cache among threads. In our case, we strongly believe that MRCs could be used to develop new approaches on top of those presented in Chapters 3 and 4.

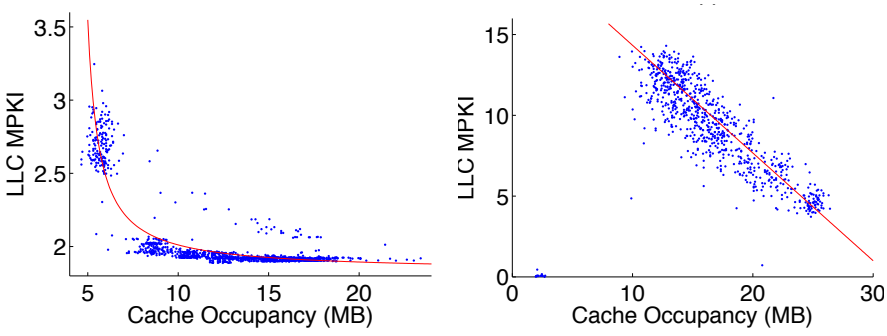


Figure 4.27: MRCs for *lbm* (a) and *omnetpp* (b) applications

4.5 Conclusions

In this chapter we have addressed the main constraints of conventional SRAM last-level caches: power-hungry operation and inefficient management. In order to overcome these drawbacks we propose to employ an STT-RAM shared LLC where its contents are selected according to a *Reuse Detector* which exploits the reuse locality of the stream of references arriving at the LLC. The Reuse Detector is a hardware component that tracks block reuse and determines, according to its predicted future utility, if they must be inserted in the LLC or bypassed to the main memory.

The Reuse Detector succeeds in managing the STT-RAM LLC contents in two complementary ways. First, it is able to bypass to main memory a significant fraction of the blocks evicted from the last-private-level cache instead of inserting them in the LLC, thus decreasing the amount of the energy-hungry writes to be performed in the STT-RAM. Second, it increases significantly the LLC hit rate, which leads to moderate performance improvements. In addition, the energy consumption in the main memory is also reduced. This way, our approach is able to outperform other strategies also oriented to decrease the energy consumption in STT-RAM LLCs, such as the DASCA scheme. Although DASCA exhibits slightly lower ability to cut the write operations to the LLCs, this technique, which predicts if a block will not be reused again instead of predicting if a block is going to be reused as ours, achieves lower accuracy in the prediction, hence also significantly lower hit rates at this cache level and therefore much lower performance

improvements.

Overall RD reports on average energy reductions of 40% (quad-core) and 35% (eight-core) in the LLC, an additional 6.5% (in both quad and eight-core) energy reduction in the main memory, and improves performance by 3% (quad-core) and 7% (eight-core) compared to an STT-RAM LLC baseline.

5

Conclusions and major contributions

Since the beginning of the computing era, the memory system has been one the most important topics in computer architecture research. This is because, as explained extensively along this thesis, one of the major constraints for the improvement of computer performance is the well-known memory gap. Specifically, since the very first conception of memory hierarchy, more levels have been progressively added or technological improvements have been applied to each single level. For example, switching from single-core to multi-core processors brought along with it the inclusion of a Shared Last-Level Cache, which allows to have a common area of data and simplifies coherency management. As for the technological improvements, SRAM and DRAM integration has progressively increased along the last decades, reducing the access latency and the energy consumption.

However, computer system requirements of the current era, such as portability, high performance, low power consumption or reliability, demand deep qualitative changes in all system components. Particularly, the search for a perfect memory (i.e. one with an immediate response, infinite capacity and no cost), seems to lead to the adoption of Non-Volatile Memory (NVM) technologies as storage systems for the future computers, given that these technologies entail an important reduction on energy consumption and at the same time a high performance improvement. In fact, although some NVM technologies were discovered 50 years ago and they have been an important research topic for many years, there has recently been an important rise in the research area concerning these technologies.

Flash is an example of a NVM technology that has been widely used in external storage (flash drive, SD, compact flash, etc.) and recently even as a secondary storage with the development and popularization of SSD (Solid State Disk). Thus, the classic non-volatile storage (magnetic disks) has been replaced with another

non-volatile technology (flash technology) which is clearly faster. The use of SSD as secondary storage led to an increase of system performance, mainly because the difference between DRAM and secondary storage was reduced from six to three orders of magnitude. Integration of SSDs is very simple, given that these devices use the same interface and algorithms as the classic secondary storage. In order to achieve this performance improvement, DRAM-write-caches must be added to the SSDs for addressing the high write latency of flash technology, which is mainly caused by the complex algorithms used for error correction. These algorithms are necessary to ensure the reliability of data stored in a technology with such low endurance as flash.

Many efforts have been made to use NVM technologies in the levels closer to the processor. Part of them have been collected in the related work of this thesis, and we have also added new proposals to the state of the art to meet the same objectives. It is important to highlight that using NVM technologies in the lower levels of the hierarchy poses many challenges, as we have discussed along this thesis. The techniques that we have proposed address some of these challenges, such as low endurance (in PCM techonology) or high dynamic energy consumption of both PCM and STT-RAM, and do not target others, such as the fact that exposing the non-volatile characteristic to software can imply an evolution in the current paradigm which exists in the memory hierarchy so far. As such, we could see scenarios where: the operating system would always be latent, programming languages would have their data structures persistent, data from interpreted programming languages and virtual machines (Java, Python, Virtual Box, VMware, etc.) would always be available, etc. This entails great advantages but also new challenges, that go from taking advantage of non-volatibility feature to the involvement of this non-volability in the memory management, encryption of data in the non-volatile levels and, in the case of hybrid hierarchies, handling the management of the combination of non-volatile and volatile technologies.

In this thesis we have explored architectural techniques for aiding NVMs in being adopted as the main technology for implementing cache or main memories in the forthcoming generations of computers. The result of this work can be divided in two main parts.

Firstly, the DRAM, which is typically used for implementing main memory, suffers important limitations such as a low integration capacity or a high energy consump-

tion. PCM has been postulated in the latest years as one of the most promising candidates for replacing DRAM as main memory, given that this NVM technology highly overcomes the drawbacks of DRAM. However, PCM has its own limitations, being the most important one its low endurance, mainly due to the degradation suffered by PCM cells when they are written to. In this thesis, we assume a main memory implemented with PCM technology and propose architectural techniques for reducing the endurance problem. For that purpose, we change the last-level cache (LLC) replacement policy, introducing a new goal in its design: Usually, the LLC replacement algorithms are only *performance-aware*, as performance has traditionally been the main objective when implementing computing systems. Our techniques modify the LLC replacement policy by introducing a second goal in the algorithms: reducing the amount of writes. Thus, the objective in our proposed algorithms is to obtain a convenient trade-off among both (sometimes conflicting) objectives. Our proposals allow to reduce significantly the number of writes to main memory, and, in the case of a main memory based on PCM technology, to increse its endurance up to 45%.

The main contributions of the first part of this thesis are included in the following publications:

- Roberto Rodríguez, Rodrigo González-Alberquilla, Fernando Castro, Daniel Chaver, Luis Piñuel, and Francisco Tirado, "Optimización de políticas de reemplazo cache para entornos PCM", in *Proceedings of the XXIII Jornadas de Paralelismo*, ser. JP-2012, Elche, Spain, 2012, pp. 461–466
- Roberto Rodríguez-Rodríguez, Fernando Castro, Daniel Chaver, Luis Piñuel, and Francisco Tirado, "Reducing writes in phase-change memory environments by using efficient cache replacement policies", in *Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22*, 2013, pp. 93–96
- Roberto Rodríguez, Fernando Castro, Daniel Chaver, Luis Piñuel, and Francisco Tirado, "Increasing the Endurance of Phase-Change Memories with Cache Replacement Policies", in *Proceedings of the XXIV Jornadas de Paralelismo*, ser. JP-2013, Madrid, Spain, 2013, pp. 18–23
- Roberto Rodríguez, Fernando Castro, Daniel Chaver, Luis Piñuel, and Francisco Tirado, "Extending endurance of Phase Change Memories", in *Non-*

Volatile Memories Workshop, San Diego, USA, Sep. 2014

- R. Rodríguez-Rodríguez, F. Castro, D. Chaver, R. Gonzalez-Alberquilla, L. Piñuel, and F. Tirado, "Write-aware replacement policies for PCM-based systems", *The Computer Journal*, vol. 58, no. 9, pp. 2000–2025, 2015. DOI: doi:10.1093/comjnl/bxu104. [Online]. Available: http://dx.doi.org/10.1093/comjnl/bxu104

In the second part of this thesis, we have focused our efforts on improving the efficiency of a system based on a LLC implemented with STT-RAM technology. Cache memories have traditionally been implemented using SRAM technology. The high static energy consumption exhibited by SRAM has put it in the spotlight for being replaced by a new memory technology. STT-RAM, another NVM, is a promising candidate for that, due to its negligible leakage, small size and low read latency. However, similarly to PCM, this NVM also has its own drawbacks: high write latency and energy consumption.

Because temporal locality is highly exploited in the private levels of the cache hierarchy, the accesses to the LLC do not exhibit such property, as almost all temporal locality is filtered by the private levels. Thus, using this property to manage the LLC contents causes that most of the blocks allocated to this cache level are dead on arrival. In this thesis we propose to use a content selector as a filter to classify the blocks that will be allocated in the LLC. Although the temporal locality is filtered by the private levels, the reuse locality (i.e. the property that a block that is reused in the LLC will probably be reused again in the near future) is a good choice to predict the behavior of the LLC blocks, so we use a Reuse Detector (RD) as a content selector for the LLC. Our RD reports on average energy reductions up to 40% in the LLC, an additional 6.5% energy reduction in the main memory, and improves performance up to 7% compared to an STT-RAM LLC without RD.

As a last contribution included in this second part of the thesis, we have presented a novel method for building the Miss-Rate Curve (MRC) of an application in a real system. For building those curves, we use PMCTrack, a novel tool that permits, among other features, to interact with the Performance Monitoring Counters available in the current processors. MRCs can be used for multiple purposes in the context of cache efficient techniques, and could thus be used for aiding the

techniques presented in this thesis to reach their objectives.

The main contributions of the second part of this thesis are contained in the following publications:

- Roberto Rodríguez-Rodríguez, Javier Díaz, Fernando Castro, Pablo Ibáñez, Daniel Chaver, Victor Viñals, Juan Carlos Saez, Manuel Prieto-Matias, Luis Piñuel, Teresa Monreal, and Jose María Llabería, "Reuse detector: Improving the management of STT-RAM SLLCs", Submitted to The Computer Journal, June 17th, 2016
- Juan Carlos Saez, Jorge Casas, Abel Serrano, Roberto Rodríguez-Rodríguez, Fernando Castro, Daniel Chaver, and Manuel Prieto-Matías, "An OS-Oriented Performance Monitoring Tool for Multicore Systems", in *Euro-Par 2015: Parallel Processing Workshops - Euro-Par 2015 International Workshops, Vienna, Austria, August 24-25, Revised Selected Papers*, 2015, pp. 697–709. DOI: 10.1007/978-3-319-27308-2_56. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-27308-2_56
- Juan Carlos Saez, Adrian Pousa, Roberto Rodríguez-Rodríguez, Fernando Castro, and Manuel Prieto-Matias, "PMCTrack: Delivering performance monitoring counter support to the OS scheduler", *The Computer Journal*, 2016, in press. DOI: 10.1093/comjnl/bxw065. [Online]. Available: http://dx.doi.org/10.1093/comjnl/bxw065

Adoption of NVMs as the technology used in the different cache levels or in main memory still poses many challenges. Thus, as a future work, we plan to improve the techniques proposed in this thesis and to envision new ones, with the aim of further reducing energy consumption, extending durability, improving performance, etc. Likewise, we plan to deep in generating MRCs. Specifically, we plan to use another feature recently introduced on Intel processors and related to CMT, namely Cache Allocation Technology (CAT) [125]. This feature allows the OS to dynamically specify the amount of cache space into which an application can fill. The system software could periodically vary the per-application maximum LLC allocation to monitor the MPKI for different cache size configurations. This would allow us to obtain MRC points over the whole cache size range for the co-running applications.

Bibliography

[1] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger, "Architecting phase change memory as a scalable dram alternative", *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3, p. 2, Jun. 2009, ISSN: 01635964. DOI: 10.1145/1555815.1555758.

[2] Moinuddin K Qureshi, Sudhanva Gurumurthi, and Bipin Rajendran, "Phase change memory: From devices to systems", *Synthesis Lectures on Computer Architecture*, vol. 6, no. 4, pp. 1–134, 2011.

[3] Mark Neisser and Stefan Wurm, "ITRS lithography roadmap: 2015 challenges", *Advanced Optical Technologies*, vol. 4, no. 4, pp. 235–240, 2015.

[4] Bruce L. Jacob, Spencer W. Ng, and David T. Wang, *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2008, ISBN: 978-0-12-379751-3.

[5] Semiconductor Industry Association, "International Technology Roadmap for Semiconductors (ITRS)", *Semiconductor industry association*, 2010. [Online]. Available: http://maltiel-consulting.com/ITRS%5C_2010%5C_Updates%5C_maltiel%5C_semiconductor%5C_consulting.html%5C#ITRS2010.

[6] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers, "Scalable high performance main memory system using PCM technology", *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3, pp. 24–33, Jun. 2009, ISSN: 01635964.

[7] Maurice V Wilkes, "The memory gap and the future of high performance memories", *ACM SIGARCH Computer Architecture News*, 2001. [Online]. Available: http://dl.acm.org/citation.cfm?id=373576.

[8] JL Hennessy and DA Patterson, *Computer architecture: a quantitative approach*. Morgan K. Pub, 2011, ISBN: 978-0123838728.

[9] Emerson W Pugh, "Ferrite core memories that shaped an industry", *Magnetics, IEEE Transactions on*, vol. 20, no. 5, pp. 1499–1502, 1984.

[10] K Iida, M Saito, and K Furukawa, "An 8 MBYTE magnetic bubble memory", *Magnetics, IEEE Transactions on*, vol. 15, no. 6, pp. 1892–1894, 1979.

[11] Andrew H Bobeck, "New Concept in Large-Size Memory Arrays—the Twistor", *Journal of Applied Physics*, vol. 29, no. 3, pp. 485–486, 1958.

[12] GA Fedde, "Design of a 1.5-Million-Bit Plated-Wire Memory", *Journal of Applied Physics*, vol. 37, no. 3, pp. 1373–1375, 1966.

[13] Luc Piraux, JM George, JF Despres, C Leroy, E Ferain, R Legras, K Ounadjela, and A Fert, "Giant magnetoresistance in magnetic multilayered nanowires", *Applied Physics Letters*, vol. 65, no. 19, pp. 2484–2486, 1994.

[14] Peter A Grünberg, "Nobel lecture: From spin waves to giant magnetoresistance and beyond", *Reviews of Modern Physics*, vol. 80, no. 4, p. 1531, 2008.

[15] John C Slonczewski, "Current-driven excitation of magnetic multilayers", *Journal of Magnetism and Magnetic Materials*, vol. 159, no. 1, pp. L1–L7, 1996.

[16] Jagan Singh Meena, Simon Min Sze, Umesh Chand, and Tseung-Yuen Tseng, "Overview of emerging nonvolatile memory technologies", *Nanoscale research letters*, vol. 9, no. 1, pp. 1–33, 2014.

[17] Charles F Pulvari, "An electrostatically induced permanent memory", *Journal of Applied Physics*, vol. 22, no. 8, pp. 1039–1044, 1951.

[18] Dudley Allen Buck, "Ferroelectrics for Digital Information Storage and Switching", DTIC Document, Tech. Rep., 1952.

[19] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang, "A durable and energy efficient main memory using phase change memory technology", *Acm sigarch computer architecture news*, vol. 37, no. 3, pp. 14–23, Jun. 2009, ISSN: 01635964.

[20] Wangyuan Zhang and Tao Li, "Characterizing and mitigating the impact of process variations on phase change based memory systems", *Proceedings of the Micro-42*, pp. 2–13, 2009. DOI: 10.1145/1669112.1669116.

[21] MK Qureshi and MM Franceschini, "Morphable memory system: a robust architecture for exploiting multi-level phase change memories", *Computer Architecture*, 2010.

[22] Laszlo A. Belady, "A study of replacement algorithms for virtual-storage computer", *Ibm systems journal*, vol. 5, no. 2, pp. 78–101, 1966.

[23] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, and Joel S. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)", in *37th International Symposium on Computer Architecture (ISCA 2010), June 19-23, Saint-Malo, France*, 2010, pp. 60–71.

[24] HP, "Inside the Intel Itanium 2 Processor", HP Technical White Paper, Tech. Rep. July, 2002.

[25] Sun Microsystems, "UltraSPARC T2 Supplement to the UltraSPARC architecture", Sun Microsystems, Santa Clara, Tech. Rep., 2007. DOI: DraftD1.4.3.

[26] Mainak Chaudhuri, "Pseudo-LIFO: the foundation of a new family of replacement policies for last-level caches", in *42st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42 2009), December 12-16, 2009, New York, New York, USA*, 2009, pp. 401–412.

[27] CS Kim, "LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies", *IEEE Transactions on Computers*, vol. 50, no. 12, pp. 1352–1361, 2001.

[28] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel S. Emer, "Adaptive insertion policies for high performance caching", in *34th International Symposium on Computer Architecture (ISCA 2007), June 9-13, San Diego, California, USA*, 2007, pp. 381–391.

[29] Carole-Jean Wu, Aamer Jaleel, William Hasenplaugh, Margaret Martonosi, Simon C. Steely, and Joel S. Emer, "SHiP: signature-based hit predictor for high performance caching", in *44rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2011, 3-7 December, Porto Alegre, Brazil*, 2011, pp. 430–441.

[30] Subhasis Das, Tor M Aamodt, and William J Dally, "Reuse distance-based probabilistic cache replacement", *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 4, p. 33, 2016.

[31] Thomas J. Watson IBM Research Center. Research Division, PJ Denning, YC Chen, and GS Shedler, *A model for program behavior under demand paging*. 1968.

[32] Geoffrey W Burr, Bülent N Kurdi, J Campbell Scott, Chung Hon Lam, Kailash Gopalakrishnan, and Rohit S Shenoy, "Overview of candidate device technologies for storage-class memory", *IBM Journal of Research and Development*, vol. 52, no. 4.5, pp. 449–464, 2008.

[33] Simone Raoux, Geoffrey W Burr, Matthew J Breitwisch, Charles T Rettner, Yi-Chou Chen, Robert M Shelby, Martin Salinga, Daniel Krebs, S-H Chen, Hsiang-Lan Lung, *et al.*, "Phase-change random access memory: A scalable technology", *IBM Journal of Research and Development*, vol. 52, no. 4.5, pp. 465–479, 2008.

[34] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger, "Architecting phase change memory as a scalable DRAM alternative", *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3, pp. 2–13, 2009.

[35] Moinuddin K Qureshi, Vijayalakshmi Srinivasan, and Jude A Rivers, "Scalable high performance main memory system using phase-change memory technology", *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3, pp. 24–33, 2009.

[36] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang, "A durable and energy efficient main memory using phase change memory technology", in *ACM SIGARCH computer architecture news*, ACM, vol. 37, 2009, pp. 14–23.

[37] Benjamin C Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger, "Phase-change technology and the future of main memory", *IEEE Micro*, no. 1, pp. 143–143, 2010.

[38] HP Labs, *CACTI An integrated cache and memory access time, cycle time, area, leakage, and dynamic power model*, `http://www.hpl.hp.com/research/cacti/`, Accessed: 2014-04-30.

[39] Xiangyu Dong, Cong Xu, Norm Jouppi, and Yuan Xie, "NVSim: A circuit-level performance, energy, and area model for emerging nonvolatile memory", *IEEE Transaction on Compter-Aided Design of Integrated Circuits and Systems*, vol. 31, pp. 994–1007, Jul. 2012.

[40] Alexandre Peixoto Ferreira, Miao Zhou, Santiago Bock, Bruce R. Childers, Rami G. Melhem, and Daniel Mossé, "Increasing PCM main memory lifetime", in *Design, Automation and Test in Europe, DATE 2010, Dresden, Germany, March 8-12*, IEEE, 2010, pp. 914–919.

[41] Xi Zhang, Qian Hu, Dongsheng Wang, Chongmin Li, and Haixia Wang, "A read-write aware replacement policy for phase change memory", in *Advanced Parallel Processing Technologies*, Springer, 2011, pp. 31–45.

[42] Pablo Abad, Pablo Prieto, Valentin Puente, and Jose-Angel Gregorio, "ACWAR: Architecting the Cache Hierarchy to Improve the Lifetime of a Non-Volatile Endurance-Limited Main Memory", *Parallel and Distributed Systems, IEEE Transactions on*, vol. 27, no. 1, pp. 66–77, 2016.

[43] Miao Zhou, Yu Du, Bruce Childers, Rami Melhem, and Daniel Mossé, "Writeback-aware partitioning and replacement for last-level caches in phase change main memory systems", *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 4, pp. 1–21, Jan. 2012, ISSN: 15443566.

[44] Moinuddin K. Qureshi and Yale N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches", in *39th annual ieee/acm international symposium on microarchitecture (micro-39 2006), 9-13 december 2006, orlando, florida, usa*, IEEE Computer Society, 2006, pp. 423–432.

[45] S Yoo, Edward Lee, and Hyokyung Bahn, "LDF (less dirty first): dirtiness-aware cache replacement policy for PCM main memory", *Electronics Letters*, vol. 49, no. 25, p. 1607, 2013.

[46] Benjamin C Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger, "Phase-Change Technology and the Future of Main Memory", *IEEE Micro*, vol. 30, no. 1, pp. 131–141, 2010.

[47] Sangyeun Cho and Hyunjin Lee, "Flip-n-write: A simple deterministic technique to improve pram write performance, energy and endurance", in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42 2009), December 12-16, New York, New York, USA*, IEEE, 2009, pp. 347–357.

[48] Jingtong Hu, Chun Jason Xue, Wei-Che Tseng, Yi He, Meikang Qiu, and Edwin Hsing-Mean Sha, "Reducing write activities on non-volatile memories in embedded CMPs via data migration and recomputation", in *Proceedings of the 47th Design Automation Conference, DAC 2010, Anaheim, California, USA, July 13-18,*, 2010, pp. 350–355.

[49] Tiantian Liu, Yingchao Zhao, Chun Jason Xue, and Minming Li, "Power-aware variable partitioning for DSPs with hybrid PRAM and DRAM main memory", in *Proceedings of the 48th Design Automation Conference, DAC 2011, San Diego, California, USA, June 5-10,*, 2011, pp. 405–410.

[50] Sung-In Jang, Cheong-Ghil Kim, and Shin-Dug Kim, "An Efficient DRAM Converter for Non-Volatile Based Main Memory", in *IT Convergence and Security 2012*, Springer, 2013, pp. 401–407.

[51] Sung Kyu Park, Min Kyu Maeng, Ki-Woong Park, and Kyu Ho Park, "Adaptive wear-leveling algorithm for PRAM main memory with a DRAM buffer", *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 4, p. 88, 2014.

[52] In-Sung Choi, Sung-In Jang, Chang-Hoon Oh, Charles C Weems, and Shin-Dug Kim, "A dynamic adaptive converter and management for PRAM-based main memory", *Microprocessors and Microsystems*, vol. 37, no. 6, pp. 554–561, 2013.

[53] Luiz E Ramos, Eugene Gorbatov, and Ricardo Bianchini, "Page placement in hybrid memory systems", in *Proceedings of the 25th International Conference on Supercomputing, 2011, Tucson, AZ, USA, May 31 - June 04*, ACM, 2011, pp. 85–95.

[54] Soyoon Lee, Hyokyung Bahn, and S Noh, "CLOCK-DWF: A Write-History-Aware Page Replacement Algorithm for Hybrid PCM and DRAM Memory Architectures", *IEEE Transactions on Computers*, vol. 63, no. 9, pp. 2187–2200, 2013.

[55] Fernando J Corbato, "A Paging Experiment with the Multics System", in *In Honor of P.M. Morse*, MIT Press, 1969, pp. 217–228.

[56] Karin Strauss and Doug Burger, "What the Future Holds for Solid-State Memory", *Computer*, vol. 47, no. 1, pp. 24–31, 2014.

[57] Engin Ipek, Jeremy Condit, E Nightingale, Doug Burger, and Thomas Moscibroda, "Dynamically replicated memory: Building resilient systems from unreliable nanoscale memories", in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2010)*, 2010.

[58] Stuart E. Schechter, Gabriel H. Loh, Karin Strauss, and Doug Burger, "Use ECP, not ECC, for hard failures in resistive memories", *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 141–152, 2010.

[59] John J. Shedletsky *et al.*, "Error correction by alternate-data retry", *IEEE Trans. Computers*, vol. 27, no. 2, pp. 106–112, 1978.

[60] Nak Hee Seong, Dong Hyuk Woo, Vijayalakshmi Srinivasan, Jude A Rivers, and Hsien-Hsin S Lee, "SAFER: Stuck-at-fault error recovery for memories", in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Computer Society, 2010, pp. 115–124.

[61] Doe Hyun Yoon, Naveen Muralimanohar, Jichuan Chang, Parthasarathy Ranganathan, Norman P Jouppi, and Mattan Erez, "FREE-p: Protecting non-volatile memory against both hard and soft errors", in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, IEEE, 2011, pp. 466–477.

[62] Moinuddin K Qureshi, "Pay-as-you-go: Low-overhead hard-error correction for phase change memories", in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ACM, 2011, pp. 318–328.

[63] Adam N Jacobvitz, Robert Calderbank, and Daniel J Sorin, "Coset coding to extend the lifetime of memory", in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, IEEE, 2013, pp. 222–233.

[64] Rodolfo Azevedo, John D Davis, Karin Strauss, Parikshit Gopalan, Mark Manasse, and Sergey Yekhanin, "Zombie memory: Extending memory lifetime by reviving dead blocks", in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ACM, 2013, pp. 452–463.

[65] Ferdinando Bedeschi, Rich Fackenthal, Claudio Resta, Enzo Michele Donze, Meenatchi Jagasivamani, Egidio Cassiodoro Buda, Fabio Pellizzer, David W Chow, Alessandro Cabrini, G Calvi, *et al.*, "A bipolar-selected phase change memory featuring multi-level cell storage", *IEEE Journal of Solid-State Circuits*, vol. 44, no. 1, pp. 217–227, 2009.

[66] Xiangyu Dong, Yuan Xie, Naveen Muralimanohar, and Norman P Jouppi, "Hybrid checkpointing using emerging nonvolatile memories for future exascale systems", *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 2, p. 6, 2011.

[67] Jun-Tin Lin, Yi-Bo Liao, Meng-Hsueh Chiang, I-Hsuan Chiu, Chia-Long Lin, Wei-Chou Hsu, Pei-Chia Chiang, Shyh-Shyuan Sheu, Yen-Ya Hsu, Wen-Hsing Liu, *et al.*, "Design optimization in write speed of multi-level cell application for phase change memory", in *IEEE International Conference of Electron Devices and Solid-State Circuits, 2009. EDSSC 2009*, IEEE, 2009, pp. 525–528.

[68] T Nirschl, JB Philipp, TD Happ, GW Burr, B Rajendran, MH Lee, A Schrott, M Yang, M Breitwisch, CF Chen, *et al.*, "Write strategies for 2 and 4-bit multi-level phase-change memory", in *Electron Devices Meeting, 2007. IEDM 2007. IEEE International*, IEEE, 2007, pp. 461–464.

[69] Moinuddin K Qureshi, Michele M Franceschini, and Luis Alfonso Lastras-Montaño, "Improving read performance of phase change memories via write cancellation and write pausing", in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, IEEE, 2010, pp. 1–11.

[70] Gennady Pekhimenko, Vivek Seshadri, Yoongu Kim, Hongyi Xin, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, and Todd C Mowry, "Linearly compressed pages: A low-complexity, low-latency main memory compression framework", in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ACM, 2013, pp. 172–184.

[71] David B Dgien, Poovaiah M Palangappa, Nathan A Hunter, Jiayin Li, and Kartik Mohanram, "Compression architecture for bit-write reduction in non-volatile memory technologies", in *Nanoscale Architectures (NANOARCH), 2014 IEEE/ACM International Symposium on*, IEEE, 2014, pp. 51–56.

[72] Alaa R Alameldeen and David A Wood, "Frequent pattern compression: A significance-based compression scheme for L2 caches", *Dept. Comp. Scie., Univ. Wisconsin-Madison, Tech. Rep*, vol. 1500, 2004.

[73] Charles Shelor, Jim Buchanan, Krishna Kavi, and Ron Cytron, "Quantifying wasted write energy in the memory hierarchy",

[74] Santiago Bock, Bruce Childers, Rami Melhem, Daniel Mossé, and Youtao Zhang, "Analyzing the impact of useless write-backs on the endurance and energy consumption of PCM main memory", in *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, IEEE, 2011, pp. 56–65.

[75] Emre Kultursay, Mahmut Kandemir, Anand Sivasubramaniam, and Onur Mutlu, "Evaluating STT-RAM as an energy-efficient main memory alternative", in *Performance Analysis of Systems and Software (ISPASS), IEEE International Symposium on*, IEEE, 2013, pp. 256–267.

[76] Junwhan Ahn, Sungjoo Yoo, and Kiyoung Choi, "DASCA: Dead Write Prediction Assisted STT-RAM Cache Architecture", in *High Performance Computer Architecture (HPCA), IEEE 20th International Symposium on*, IEEE, 2014, pp. 25–36.

[77] Samira Manabi Khan, Yingying Tian, and Daniel A. Jimenez, "Sampling Dead Block Prediction for Last-Level Caches", in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '43, Washington, DC, USA: IEEE Computer Society, 2010, pp. 175–186, ISBN: 978-0-7695-4299-7.

[78] Mu-Tien Chang, Shih-Lien Lu, and Bruce Jacob, "Impact of Cache Coherence Protocols on the Power Consumption of STT-RAM-Based LLC",

[79] Chao Zhang, Guangyu Sun, Peng Li, Tao Wang, Dimin Niu, and Yiran Chen, "SBAC: a statistics based cache bypassing method for asymmetric-access caches", in *Proceedings of the 2014 international symposium on Low power electronics and design*, ACM, 2014, pp. 345–350.

[80] Min Kyu Kim, Ju Hee Choi, Jong Wook Kwak, Seong Tae Jhang, and Chu Shik Jhon, "Bypassing method for STT-RAM based inclusive last-level cache", in *Proceedings of the 2015 Conference on research in adaptive and convergent systems*, ACM, 2015, pp. 424–429.

[81] Jue Wang, Xiangyu Dong, and Yuan Xie, "OAP: an obstruction-aware cache management policy for STT-RAM last-level caches", in *Proceedings of the Conference on Design, Automation and Test in Europe*, EDA Consortium, 2013, pp. 847–852.

[82] Mitchelle Rasquinha, "An energy efficient cache design using spin torque transfer (STT) RAM", Master of Science in the School of Electrical and Computer Engineering, Georgia Institute of Technology, 2011. [Online]. Available: `https://smartech.gatech.edu/bitstream/handle/1853/42715/Rasquinha_Mitchelle_Ml_201112_mast.pdf`.

[83] Sadegh Yazdanshenas, Marzieh Ranjbar Pirbasti, Mahdi Fazeli, and Ahmad Patooghy, "Coding last level STT-RAM cache for high endurance and low power", *IEEE computer architecture letters*, vol. 13, no. 2, pp. 73–76, 2014.

[84] Jinwook Jung, Yohei Nakata, Masahiko Yoshimoto, and Hiroshi Kawaguchi, "Energy-efficient Spin-Transfer Torque RAM cache exploiting additional all-zero-data flags", in *Quality Electronic Design (ISQED), 2013 14th International Symposium on*, IEEE, 2013, pp. 216–222.

[85] Sang Phill Park, Sumeet Gupta, Niladri Mojumder, Anand Raghunathan, and Kaushik Roy, "Future cache design using STT MRAMs for improved energy efficiency: devices, circuits and architecture", in *Proceedings of the 49th Annual Design Automation Conference*, ACM, 2012, pp. 492–497.

[86] Mengjie Mao, Hai Helen Li, Alex K Jones, and Yiran Chen, "Coordinating prefetching and STT-RAM based last-level cache management for multicore systems", in *Proceedings of the 23rd ACM international conference on Great lakes symposium on VLSI*, ACM, 2013, pp. 55–60.

[87] Adwait Jog, Asit K Mishra, Cong Xu, Yuan Xie, Vijaykrishnan Narayanan, Ravishankar Iyer, and Chita R Das, "Cache revive: architecting volatile STT-RAM caches for enhanced performance in CMPs", in *Proceedings of the 49th Annual Design Automation Conference*, ACM, 2012, pp. 243–252.

[88] Xiaochen Guo, Engin Ipek, and Tolga Soyata, "Resistive computation: avoiding the power wall with low-leakage, STT-MRAM based computing", in *ACM SIGARCH Computer Architecture News*, ACM, vol. 38, 2010, pp. 371–382.

[89] Zhenyu Sun, Xiuyuan Bi, Hai Helen Li, Weng-Fai Wong, Zhong-Liang Ong, Xiaochun Zhu, and Wenqing Wu, "Multi retention level STT-RAM cache designs with a dynamic refresh scheme", in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ACM, 2011, pp. 329–338.

[90] Guangyu Sun, Xiangyu Dong, Yuan Xie, Jian Li, and Yiran Chen, "A novel architecture of the 3D stacked MRAM L2 cache for CMPs", in *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, IEEE, 2009, pp. 239–249.

[91] Standard Performance Evaluation Corporation, *Benchmarks SPEC CPU 2006*, `http://www.spec.org/cpu2006/`, Accessed: 2014-04-30.

[92] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, *et al.*, "The gem5 simulator", *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, Aug. 2011, ISSN: 01635964.

[93] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A Cycle Accurate Memory System Simulator", *Computer Architecture Letters*, vol. 10, no. 1, pp. 16–19, Jan. 2011, ISSN: 1556-6056. DOI: `10.1109/L-CA.2011.4`.

[94] Rio Xiangyu Dong, *Gem5+DRAMsim2 integration patch*, `http://www.cse.psu.edu/~xydong/software.html`, Accessed: 2014-04-30.

[95] Christian Bienia, "Benchmarking Modern Multiprocessors", PhD thesis, Princeton University, Jan. 2011.

[96] Harish Patil, Robert S. Cohn, Mark Charney, Rajiv Kapoor, Andrew Sun, and Anand Karunanidhi, "Pinpointing Representative Portions of Large Intel® Itanium® Programs with Dynamic Instrumentation", in *37th Annual International Symposium on Microarchitecture (MICRO-37 2004), 4-8 December 2004, Portland, OR, USA*, IEEE Computer Society, 2004, pp. 81–92, ISBN: 0-7695-2126-6.

[97] Amir Ban, *Wear leveling of static areas in flash memory*, US Patent 6,732,221, May 2004.

[98] Taeho Kgil, David Roberts, and Trevor N. Mudge, "Improving NAND Flash Based Disk Caches", in *35th International Symposium on Computer Architecture (ISCA 2008), June 21-25, Beijing, China*, IEEE, 2008, pp. 327–338.

[99] Eran Gal and Sivan Toledo, "Algorithms and data structures for flash memories", *ACM Comput. Surv.*, vol. 37, no. 2, pp. 138–163, 2005.

[100] Avraham Ben-Aroya and Sivan Toledo, "Competitive analysis of flash memory algorithms", *ACM Transactions on Algorithms*, vol. 7, no. 2, pp. 23–37, 2011.

[101] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi, "CACTI 6.0: A tool to understand large caches", *University of Utah and Hewlett Packard Laboratories, Tech. Rep*, vol. 1, pp. 1–20, 2009.

[102] Jorge Albericio, Pablo Ibáñez, Víctor Viñals, and Jose María Llabería, "Exploiting Reuse Locality on Inclusive Shared Last-level Caches", *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, 38:1–38:19, Jan. 2013, ISSN: 1544-3566.

[103] Jorge Albericio, Pablo Ibáñez, Víctor Viñals, and José M. Llabería, "The Reuse Cache: Downsizing the Shared Last-level Cache", in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46, Davis, California: ACM, 2013, pp. 310–321, ISBN: 978-1-4503-2638-4.

[104] Javier Díaz, Teresa Monreal, Víctor Viñals, Pablo Ibáñez, and José M. Llabería, "Selección de contenidos basada en reuso para caches compartidas en exclusión", in *Proceedings of the XXVI Jornadas de Paralelismo*, ser. JP-2015, Cordoba, Spain, 2015, pp. 433–442, ISBN: 9788416017522.

[105] Niket Agarwal, Tushar Krishna, Li-Shiuan Peh, and Niraj K Jha, "GARNET: A detailed on-chip network model inside a full-system simulator", in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, IEEE, 2009, pp. 33–42.

[106] Moinuddin K. Qureshi and Yale N. Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches", in *Proceedings of MICRO 06*, Orlando, FL: IEEE Computer Society Washington, DC, USA, Sep. 2006, pp. 423–432.

[107] David K. Tam, Reza Azimi, Livio B. Soares, and Michael Stumm, "RapidMRC: Approximating L2 Miss Rate Curves on Commodity Systems for Online Optimizations", in *Proceedings of ASPLOS 09*, Washington, DC: ACM, New York, Jul. 2009, pp. 121–132, ISBN: 978-1-60558-406-5.

[108] Erik Berg and Erik Hagersten, "Statcache: A probabilistic approach to efficient and accurate data locality analysis", in *Proceedings of ISPASS 04*, Austin, TX: IEEE Computer Society, Washington, DC, USA, Oct. 2004, pp. 20–27.

[109] Fei Guo and Yan Solihin, "An Analytical Model for Cache Replacement Policy Performance", in *Proceedings of SIGMETRICS 06*, Saint Malo, France: ACM, New York, 26-30 June 2006, pp. 228–239.

[110] Julian Seward and Nicholas Nethercote, "Using Valgrind to Detect Undefined Value Errors with Bit-Precision", in *USENIX Annual Technical Conference, General Track*, 2005, pp. 17–30.

[111] Qin Zhao, Rodric Rabbah, Saman Amarasinghe, Larry Rudolph, and Weng-Fai Wong, "Ubiquitous memory introspection", in *Proceedings of the International Symposium on Code Generation and Optimization*, IEEE Computer Society, 2007, pp. 299–311.

[112] Juan Carlos Saez, *PMCTrack's Official Website*, http://pmctrack.dacya.ucm.es/, Accessed: 2016-06-15.

[113] Juan Carlos Saez, Jorge Casas, Abel Serrano, Roberto Rodríguez-Rodríguez, Fernando Castro, Daniel Chaver, and Manuel Prieto-Matías, "An OS-Oriented Performance Monitoring Tool for Multicore Systems", in *Euro-Par 2015: Parallel Processing Workshops - Euro-Par 2015 International Workshops, Vienna, Austria, August 24-25, Revised Selected Papers*, 2015, pp. 697–709. DOI: 10.1007/978-3-319-27308-2_56. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-27308-2_56.

[114] Juan Carlos Saez, Adrian Pousa, Roberto Rodríguez-Rodríguez, Fernando Castro, and Manuel Prieto-Matias, "PMCTrack: Delivering performance monitoring counter support to the OS scheduler", *The Computer Journal*, 2016, in press. DOI: 10.1093/comjnl/bxw065. [Online]. Available: http://dx.doi.org/10.1093/comjnl/bxw065.

[115] K. Nguyen, *Intel's Cache Monitoring Technology Software-Visible Interfaces*, https://software.intel.com/en-us/blogs/2014/12/11/intel-s-cache-monitoring-technology-software\-visible-interfaces, Accessed: 2015-02-10, 2014.

[116] ——, *Benefits of Intel Cache Monitoring Technology in the Intel Xeon Processor E5 v3 Family*, https://software.intel.com/en-us/blogs/2014/06/18/benefit-of-cache-monitoring, Accessed: 2015-02-10, 2014.

[117] *Intel® 64 and ia-32 architectures developer's manual: Vol. 3b*, http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.html, 2014.

[118] Rathijit Sen and David A. Wood, "Reuse-based Online Models for Caches", in *Proceedings of sigmetrics 13*, Pittsburgh, PA: ACM, New York, 17-21 June 2013, pp. 279–292.

[119] Roberto Rodríguez, Rodrigo González-Alberquilla, Fernando Castro, Daniel Chaver, Luis Piñuel, and Francisco Tirado, "Optimización de políticas de reemplazo cache para entornos PCM", in *Proceedings of the XXIII Jornadas de Paralelismo*, ser. JP-2012, Elche, Spain, 2012, pp. 461–466.

[120] Roberto Rodríguez-Rodríguez, Fernando Castro, Daniel Chaver, Luis Piñuel, and Francisco Tirado, "Reducing writes in phase-change memory environments by using efficient cache replacement policies", in *Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22*, 2013, pp. 93–96.

[121] Roberto Rodríguez, Fernando Castro, Daniel Chaver, Luis Piñuel, and Francisco Tirado, "Increasing the Endurance of Phase-Change Memories with Cache Replacement Policies", in *Proceedings of the XXIV Jornadas de Paralelismo*, ser. JP-2013, Madrid, Spain, 2013, pp. 18–23.

[122] ——, "Extending endurance of Phase Change Memories", in *Non-Volatile Memories Workshop*, San Diego, USA, Sep. 2014.

[123] R. Rodríguez-Rodríguez, F. Castro, D. Chaver, R. Gonzalez-Alberquilla, L. Piñuel, and F. Tirado, "Write-aware replacement policies for PCM-based systems", *The Computer Journal*, vol. 58, no. 9, pp. 2000–2025, 2015. DOI: doi:10.1093/comjnl/bxu104. [Online]. Available: http://dx.doi.org/10.1093/comjnl/bxu104.

[124] Roberto Rodríguez-Rodríguez, Javier Díaz, Fernando Castro, Pablo Ibáñez, Daniel Chaver, Victor Viñals, Juan Carlos Saez, Manuel Prieto-Matias, Luis Piñuel, Teresa Monreal, and Jose María Llabería, "Reuse detector: Improving the management of STT-RAM SLLCs", Submitted to The Computer Journal, June 17th, 2016.

[125] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual Volumes 3A and 3B: System Programming Guide*, http://www.intel.com/products/processor/manuals, Accessed: 2015-01-15.

Alphabetical Index

V

W

Z

Since the beginning of computer systems memory subsystem has been one of their mainstays. But the different evolution rate between microprocessor and memory has become one of the greatest challenges that current designers have to overcome in order to develop more powerful computer systems. In addition to this problem, called memory gap, it is the fact that the scalability and energy consumption of conventional technologies (SRAM and DRAM) is very limited currently, leading to consider new Non-Volatile Memory technologies (NVM) as possible candidates for the replacement of conventional ones. PCM and STT-RAM are currently postulated as the best alternative for it.

NVMs have significant advantages over SRAM and DRAM, but also some drawbacks that need to be mitigated before they can be used as memory technologies for the next computers generation. In particular, the cost in energy and latency of the writes of these technologies and the endurance of PCM (the life time limited by the number of write cycles that can be performed on each cell) are their main drawbacks. This thesis presents two proposals: first a behavioral analysis, in terms of number of writes to main memory, of conventional cache replacement policies and also new proposals aimed at reducing the number of writes to main memory are exposed thought for increase the life time of the memory when it is developed using PCM technology. Likewise, a second proposal to avoid useless writes to last level of cache (LLC) is presented, this is aimed to reduce the energy consumption on a system where the LLC is based on STT-RAM technology.

ISBN-13: 978-84-617-4671-2

MECANISMOS DE GESTIÓN DE ESCRITURAS EN SISTEMAS CON TECNOLOGÍAS DE MEMORIA NO VOLÁTILES
WRITE MANAGEMENT MECHANISMS FOR SYSTEMS WITH NON-VOLATILE MEMORY TECHNOLOGIES

UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA

MECANISMOS DE GESTIÓN DE ESCRITURAS EN SISTEMAS CON TECNOLOGÍAS DE MEMORIA NO VOLÁTILES

(WRITE MANAGEMENT MECHANISMS FOR SYSTEMS WITH NON-VOLATILE MEMORY TECHNOLOGIES)

Roberto Alonso Rodríguez Rodríguez

Bajo la dirección de los doctores:

Fernando Castro Rodríguez

Daniel Ángel Chaver Martínez

Madrid 2016