UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA

Departamento de Arquitectura de Computadores y Automática

TESIS DOCTORAL

Caracterización y optimización térmica de sistemas en chip mediante emulación con Fugas

Thermal characterization and optimization of systems-on-chip through FPGA-based emulation

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Pablo García del Valle

Directores

David Atienza Alonso
José Manuel Mendías Cuadros

Madrid, 2012

Caracterización y optimización térmica de sistemas en chip mediante emulación con FPGAs

Thermal Characterization and Optimization of Systems-on-Chip through FPGA-Based Emulation

Tesis Doctoral

Pablo García del Valle

Departamento de Arquitectura de Computadores y Automática
Facultad de Informática
Universidad Complutense de Madrid

2012

Caracterización y optimización térmica de sistemas en chip mediante emulación con FPGAs

Thermal Characterization and Optimization of Systems-on-Chip through FPGA-Based Emulation

Tesis presentada por
Pablo García del Valle

Departamento de Arquitectura de Computadores y Automática
Facultad de Informática
Universidad Complutense de Madrid

2012

Caracterización y optimización térmica de sistemas en chip mediante emulación con FPGAs

Memoria presentada por Pablo García del Valle para optar al grado de Doctor por la Universidad Complutense de Madrid, realizada bajo la dirección de D. David Atienza Alonso y D. José Manuel Mendías Cuadros (Departamento de Arquitectura de Computadores y Automática, Universidad Complutense de Madrid).

Madrid, Febrero de 2012.

Thermal Characterization and Optimization of Systems-on-Chip through FPGA-Based Emulation

Dissertation presented by Pablo García del Valle to the Complutense University of Madrid in order to apply for the Doctoral degree. This work has been supervised by Mr. David Atienza Alonso and Mr. José Manuel Mendías Cuadros (Computers Architecture and Automation Department, Complutense University of Madrid).

Madrid, February 2012.

Este trabajo ha sido posible gracias a la Comisión Interministerial de Ciencia y Tecnologia, por las ayudas recibidas a través de los proyectos CICYT TIN2005/5619 y CICYT TIN2008/00508, y de la beca de investigación FPU AP2005-0073.

A mis padres,
a Rocío.

Acknowledgements

A todos aquellos que,
de una manera u otra,
han hecho posible que
este trabajo
vea la luz.

Gracias.

En primer lugar, he de agradecer especialmente el esfuerzo realizado por mis dos directores de tesis, D. David Atienza Alonso y D. José Manuel Mendías Cuadros, de quienes siempre he recibido la orientación necesaria a lo largo de estos años. Lo mucho que he aprendido de vosotros, tanto a nivel profesional como personal, ha sido, es, y será, impagable. De verdad.

I would also like to thank Prof. Giovanni De Micheli; I have learnt a lot and really enjoyed my research stays at LSI. Thanks for your continuous guidance, help and support.

Por otro lado, quiero agradecer al Prof. Román Hermida el interés que ha demostrado y el seguimiento que ha hecho de mi trabajo. A él debo mi iniciación en esta singladura, que hoy atraca en buen puerto. Recuerdo cuando, en su clase de Arquitectura de Computadores, allá por 2004, hizo que me picara el gusanillo de la investigación, hablándonos de ciertas becas de colaboración que el departamento ofrecía...

Deseo también expresar mis agradecimientos al Prof. Francisco Tirado, investigador principal de los proyectos en los que he participado, y al Departamento de Arquitectura de Computadores, sin los cuáles no habría contado ni con el apoyo ni con la infraestructura necesarios para realizar mi labor investigadora.

Sin duda, le debo mucho a José Luis Ayala, quien me ha guiado en la etapa final de mi tesis. A pesar de estar muy ocupado, siempre ha tenido un momento para ayudarme, ya fuera con discusiones científicas, o con sus broncas focalizadoras. Resulta increíble que no haya perdido la fé en mí. Gracias José.

El hecho de realizar esta tesis ha tenido algunos efectos colaterales: Por un lado, me ha exigido mucha dedicación; es por ello que quiero agradecer su apoyo a todos esos amigos que "siguen ahí", a pesar de no haberles podido dedicar el tiempo que se merecían, en especial a Javi y Raquel. Por otro lado, me ha dado la oportunidad de conocer a gente fantástica con la que he compartido momentos inolvidables. Quiero daros las gracias a todos: A Alberto, sin duda, el mejor tipo que conozco; A Miguel, que ya dejó de sorprenderme; A Fran, Abhi, Ahmed, Anto, Antonio, Carlitos, Cuesta, Dixon, Emilio, Fabrizio, Federico, Guada, Guillermo, Íñigo, JC, Joaquín, Josele, Juanan, Katza, Laura, Lanne, Lucas, Marcos, Milagros, Mohe, Morfino, Motos, Naser, Poletti, RoFamily, Shashi, Srini, Suiza, y Urbón. A Marga y a sus monjas.

A special acknowledgement goes to Haykel Ben Jamaa, a.k.a. "el Tunecino", for "not showing me the light". You are worth all the camels in this Galaxy, man !

A todos aquellos que me han complicado la existencia porque, lejos de perjudicarme, lo que han hecho es darme lecciones magistrales sobre la vida; un verdadero ejemplo de altruismo, ¡sí señor!. Gracias.

Finalmente, nada de esto habría sido posible sin la educación, el respeto y la honestidad que me han inculcado mis padres, José y Aurora. No me cabe la menor duda de que, aunque esta tesis os suene más a chino que a cristiano, estaréis más orgullosos que yo de ella. A partir de ahora podré dedicar más tiempo a corresponderos. Gracias por todo.

Y... ¡cómo no!, unas cosas terminan y otras comienzan... quiero dedicar esta tesis a la pequeña Lucía y a sus padres, Aurora y José.

A Rocío, que durante este tiempo siempre ha estado a mi lado. Día a día, me has enseñado qué es la vida, y me has animado a seguir adelante. Tengo mucho que aprender de tí. Gracias por tu apoyo incondicional.

Index

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays, the consumer electronics market is dominated by state-of-the-art handhelds like tablets, GPS navigation systems, smartphones, or digital cameras. These systems are complex to design as they must execute multiple applications, most of them related to the boom in the multimedia sector (e.g.: real-time video processing, 3D games, or wireless communications), while meeting additional design constraints, such as low energy consumption, reduced implementation size and, of course, a short time-to-market.

From the point of view of the architecture designers, in addition to the challenge of selecting the right system components to meet all these design constraints, new problems, mainly technology-related, have appeared, that complicate even more the design process of state-of-the-art chips.

As technology scales down the sizes of transistors, the system integration complexity also increases: Current gadgets that offer the computing power of personal computers designed 5 years ago, but now shrinked into portable devices, burn a substantial amount of power in a very small area, which results into a high on-chip power density. The logic density of this kind of designs, coupled with very demanding SW applications can lead to the generation of hotspots [SSS+04] that compromise the chip reliability. In fact, temperature and reliability issues, are already a major concern in latest technology nodes [SABR05; RS99]. In the past, thermal problems were solved by improving the packaging solution, but now, designing a chip for the worst-case scenario often makes the final product prohibitibely expensive, and sometimes not even possible to manufacture (due to space constraints in the embedded system, for example). In this context, new design constraints need to be taken into account during the design phase of the embedded system.

In order to discover new methodologies and techniques to tackle the thermal issues, mechanisms to efficiently evaluate complete designs in terms of energy consumption, temperature, performance and other key metrics, are needed. Specially, tools able to accurately model these parameters, before the manufacturing of the chip, while running real-life applications are pri-

mordial for designers; not only to design and optimize the HW system, but also to test and elaborate complex, hibrid (hardware and software) run-time power/thermal/reliability management strategies.

With this purpose in mind, in this thesis, I introduce a new framework that offers an integrated flow for the fast exploration of multiple HW and SW implementation alternatives, with accurate estimations of performance, power, temperature, and reliability, to help designers tune the system architecture at an early stage of the design process.

1.1. Embedded Systems

When we mention the word *processors*, many people think intuitively of general purpose processors (GPPs). Those acting as servers, workstations, or personal computers, manufactured by named brands like Intel and that are spread worldwide solving a wide range of problems. However, there are other types of processors much more present in our daily lifes: *embedded processors* and *microcontrollers*. They are found in dedicated *embedded systems*, with a more or less specific function, and with clear limitations and requisites.

Attending to their characteristics, we can divide the microprocessor market into GPPs, embedded processors, and microcontrollers (MicroController Units, or MCUs). Table 1.1 summarizes their main characteristics.

Table 1.1: Microprocessor types and characteristics.

TYPE	EXAMPLES	CHARACTERISTICS	USE
General purpose processors	Pentium, Alpha, SPARC.	Complex OOSS: UNIX, NT. General purpose SW. Volume production. Optimized for versatility.	Workstations, PC's.
Embedded processors	ARM, Hitachi SH7000, Microblaze, NEC V800, PowerPC 405.	Real-time (minimal) OOSS. Executing light applications. Large volume production. Optimized for size, power consumption, reliability, etc.	Cell phones, consumer electronics.
uControllers	Motorola 68HCxx family, Microchip PICs.	No OOSS. Tiny tasks (data adquisition). Huge volume production. Optimized for cost.	Automotive, household electrical appliances.

As Figure 1.1 shows, the niche market that each of these microprocessor-based solutions offers, comes determined by the cost-performance trade-offs.

Although the MCUs have the lowest cost, their volume production is large and, thus, they generate important revenues. Figure 1.2 depicts the sales, in millions of dollars, of the MCU market. A big share belongs to the under-32 bit microcontrollers (these simple microcontrollers are still extremelly

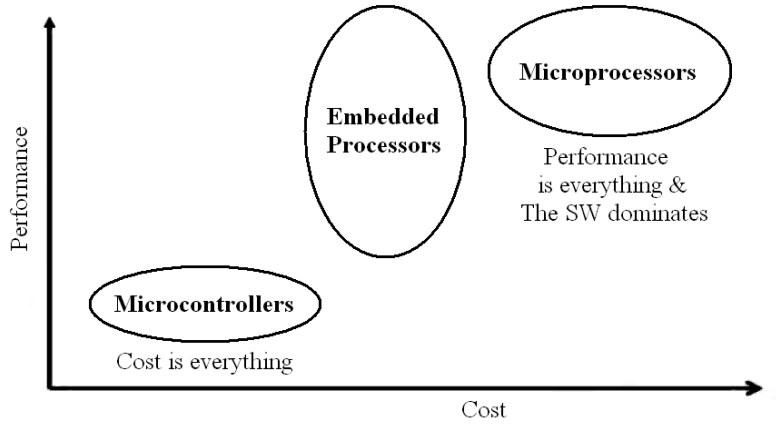


Figure 1.1: Cost-performance trade-offs of microprocessor-based solutions.

useful for tasks where we need very little performance at the lowest cost). Similarly, the market volume of embedded processors greatly surpasses that of the GPPs.

An estimation says that each household contains between 40 and 50 embedded microprocessors, on average. There are microcontrollers in the microwave, in the washing machine, in the hair drier, in the dishwasher... and not only that, but also inside audio and video devices, such as the DVD and CD players. They are also present in vehicules in really high quantities: a car has, on average, a dozen of embedded microprocessors and, to give precise examples, the BMW series 7 has 63 embedded microprocessors [BDT03]. Most of the electronic devices that surround us have one or more embedded processors dedicated to acomplish the different tasks.

After this brief introduction, I can now more formally define an embedded system, and enumerate its main characteristics: *An embedded system is that whose control is based on a general purpose microprocessor/microcontroller, and dedicated to perform a task, or set of specific tasks.* In the last years, this field has experienced a spectacular rise. The systems have evolved, from simple control devices, designed specifically to perform one task or a small set of specific tasks, into more complex systems, running applications similar to those found on desktop computers, but with strong requirements, mainly power related, to satisfy. In fact, nowadays, the most important features required from embedded systems are similar to those present in high performance systems:

- *Reliability and security:* These requirements are, generally, much more restrictive in embedded systems than for any other of computer-based systems. When, for example, a scientific computing program fails, it is enough aborting the execution, solving the error, and relaunching the

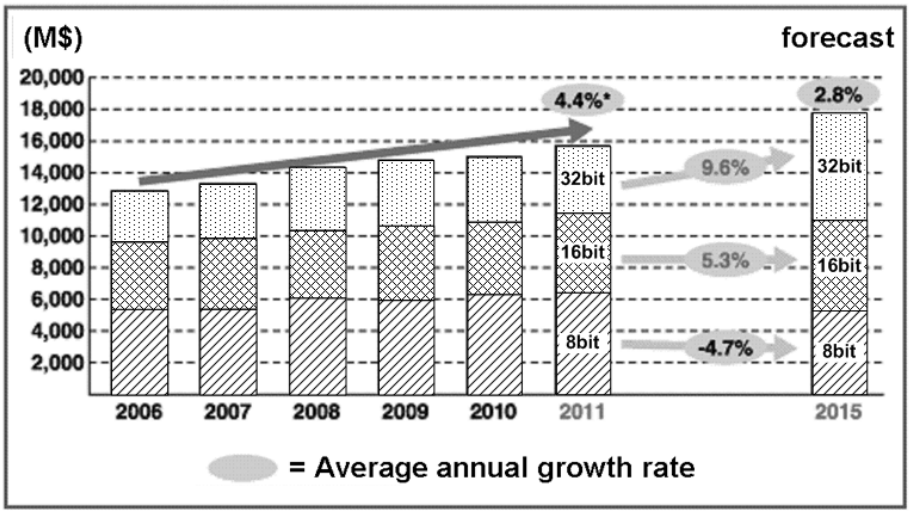


Figure 1.2: Microcontroller market.

program. However, the control system of a nuclear plant must never permit the reactor to go out of control, since this situation would cause terrible consequences. We find another example in the ejection seats inside fighter aircrafts.

- *Interaction with physical devices:* Embedded systems must interact with the environment using different kinds of devices that are normally not conventional: data adquisition boards, A/D and D/A converters, PWM, serial and parallel inputs and outputs, sensors, etc...

- *Reactivity and real-time:* Some components of the embedded systems must react continuously and simultaneously to the environment changes, and they must compute results in real-time. It is of notorious importance the high degree of concurrency, determinism and predictability requested from this kind of systems.

- *Robustness:* It is frequent the case where these systems are placed in movable parts, or can be transported, exposed to vibrations, and even impacts. The correct behaviour must be garanteed, even under bad temperature, humidity, and/or dirtiness conditions. Error handling can be done through HW, if the device supports it, or via SW, but, in this case, the system is not so robust.

- *Low power:* The need to operate with batteries, or in poorly ventilated environments, mandates the reduction in the power consumption, in order to mitigate the power dissipation that generates the overheating of the electronic components.

- *Reduced price:* Specially important if our product is intended for mass production, or we try to release a commercial version of the system.
- *Small dimensions:* Do not only depend on the size of the device itself, but also on the available space around the controlled/monitorized system. It is directly related to the power consumption.
- *Special design flow:* Designing this kind of systems implies developing together a set of HW and SW components. HW offers performance, while SW offers flexibility.
- *Flexibility:* Extremely sensitive to the market and technology factors, the systems must be able to evolve with the market in a flexible way and in a limited time.

The aforementioned metrics and characteristics are tipically inter-related or, even worse, compete one against each other; improving one of them usually implies the degradation of the others. For this reason, the designer must be familiar with a variety of technologies and HW/SW techniques, with the goal to find the best implementation for a given application and constraints. The validation of the final system through the appropriate selection of different case studies is mandatory. Performance is always important, but it normally comes as a secondary feature.

Apart from the microprocessor, an embedded system is made of additional components. The most significant ones are the memory (with optional Memory Management Unit), storage elements, the input/output devices (sensors, actuators), and the debugging ports. All of them can be observed in Figure 1.3.

1.1.1. High performance embedded systems: SoCs and MPSoCs

In the recent years, new application demands have popped up in the embedded market, specially in the consumer electronics field, that cannot be satisfied with the classic HW or SW systems. The new *High Performance Embedded Systems* must provide services such as videoconference, recording and reproduction of music and video, 3D games and so on, that imply satisfying a common set of design/implementation constraints that distinguish them from the other, more general, computing systems.

On their way towards competitiveness, designers increased chip integration to reduce manufacturing costs and to enable smaller systems. New techniques, together with the improvements in technology, led to the construction of devices comprising a number of chips in a single package; the systems in package (SiPs) [DYIM07]. This miniaturizing trend continued in the embedded market, until all the elements fit inside a single ship, greatly reducing the

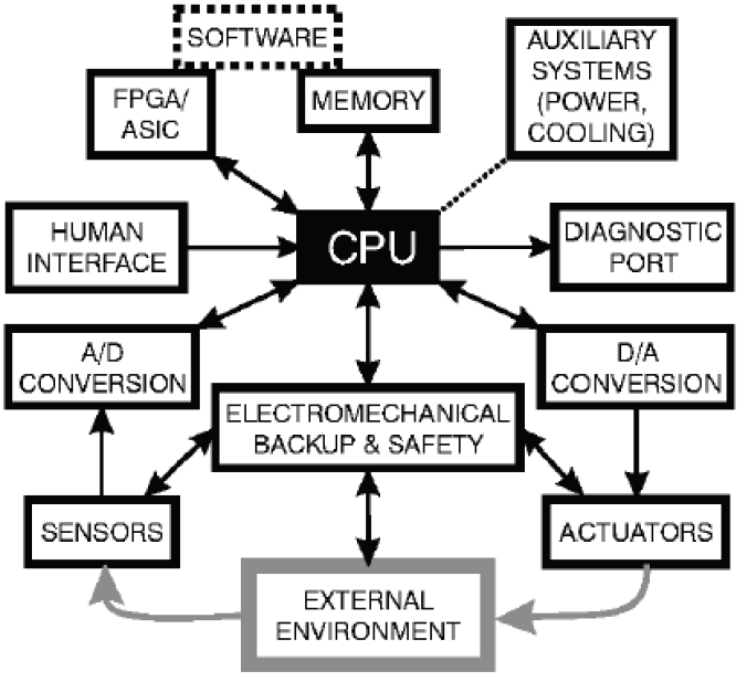


Figure 1.3: Typical components in an embedded system.

communication delays and, thus, the execution times. The systems-on-chip concept was born.

System-on-a-chip or system-on-chip (SoC) [Cla06] refers to integrating all components of a computer or other electronic system into a single integrated circuit (chip); It may contain digital, analog, mixed-signal, and often radio-frequency functions, all in a single chip substrate.

Both SoCs and SiPs coexist today. In large volumes, SoC is more cost effective than SiP since it increases the yield of the fabrication and because its packaging is simpler. However, for some applications, it is still not possible or too expensive to integrate all the functionality into one integrated circuit (IC), resulting in a SiP implementation.

The previously mentioned microcontrollers typically have just a few KBytes of RAM, and very often are single-chip-systems; whereas the term SoC is typically used with more powerful processors, capable of running full featured operating systems (Windows or Linux), which need external memory chips (flash, RAM) to be useful, and which are connected to various external peripherals; e.g.: the ARM from ARM Holdings, the SH RISC from Hitachi, the PowerPC from IBM and Motorola, the Am29K from AMD, and the MIPS from Silicon Graphics. A natural extension to SoCs are the MPSoCs:

A Multi-Processor System-on-Chip (MPSoC) [JTW05] is a system-on-a-chip (SoC) that contains multiple processors (i.e.: multi-core), usually tar-

geted for embedded applications. In addition, they typically contain several, usually heterogeneous, processing elements with specific functionalities, reflecting the need of the expected application domain. MPSoC architectures are really efficient in meeting the performance needs of multimedia applications, telecommunication architectures, network security and other application domains, while limiting the power consumption through the use of specialised processing elements and architecture.

1.1.2. The HW-SW codesign

In a high performance embedded system, we distinguish two fundamental components that must work together to satisfy the system specifications:

1. *HW component:* Designing a state-of-the-art dedicated system starting from scratch, and trying to redesign and optimize globally all the necessary modules, is an extremely complex task; Thus, the only valid alternative is to create the global system by using composition and reuse of existing components designed independently.

2. *SW component:* In new embedded systems, the SW component is essential, for it will determine the success in the market of the final product, as well as the final cost of the new system. The SW must be capable of using efficiently the optimizations that the HW offers, to enhance the performance of the embedded applications and reduce the power consumption.

The traditional design techniques (i.e., independent HW and SW design) for classical embedded systems are now being challenged when heterogeneous models and applications are getting integrated to create complex MPSoCs. In HW-SW codesign [dMG97], designers decide the location of the HW and SW components, and how to intercommunicate them efficiently to reach the specified funtionality, satisfying the development time constraints, cost, and power consumption for a given set of performance goals and technology.

When they are developed independently, there is little opportunity to optimize both HW and SW together. Moreover, it is also difficult to reason about a complete system (i.e. simulation, verification). As observed in Figure 1.4, the HW and SW design methodologies are now merged into one single design flow so that the partition into HW/SW elements is not fixed, and can be adjusted to trade-off features. One basic approach is to identify and implement SW parts which consume high computing resources (usually time) in HW [EH92]. The dual approach seeks to identify complex system parts which are good candidates to be implemented in SW [GD92].

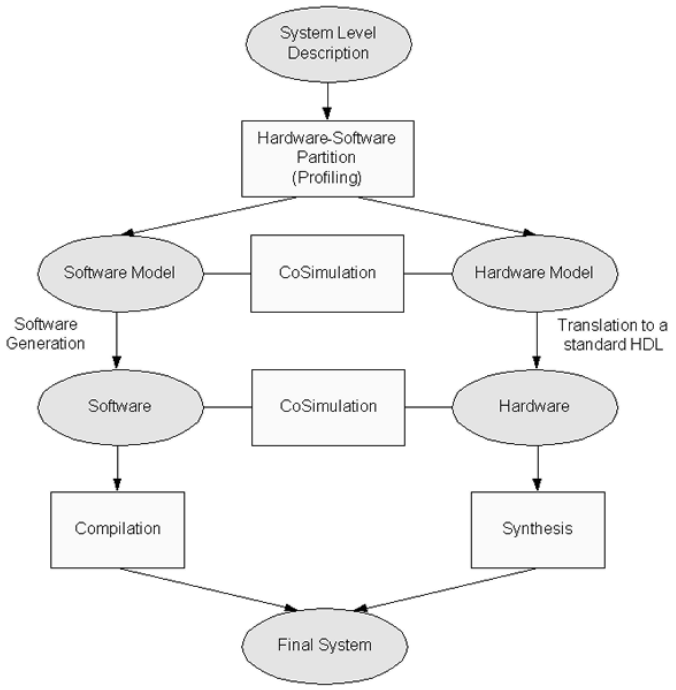


Figure 1.4: Codesign flow.

1.1.3. Intellectual Property Cores

In electronic design, a semiconductor intellectual property core, IP core [dOFdLM+03] or IP block, is a reusable unit of logic, cell, or chip layout design that is the intellectual property of one party. IP cores may be licensed to another party or can be owned and used by a single party alone. The licensing and use of IP cores in chip design came into common practice in the 1990s. The microprocessor cores of ARM Holdings are recognized as some of the first widely licensed IP cores.

Figure 1.5 shows a subsystem entirely made of IP cores: from the UART to the bus arbiter, they are all independent elements, already verified by third parties, and interconnected through standard interfaces.

Although I did not mention the name, I introduced before the IP cores as the building blocks in SoCs designs. An IP core can be described as being for chip design what a library is for computer programming, or a discrete integrated circuit component is for printed circuit board design. Attending to the way they are deployed, we can differentiate two types of IP cores:

1. *Soft cores:* IP cores are typically offered as synthesizable RTL, in a HW description language such as Verilog or VHDL, or as generic gate-level netlists (to avoid modifications). Both allow a synthesis, placement and route design flow.

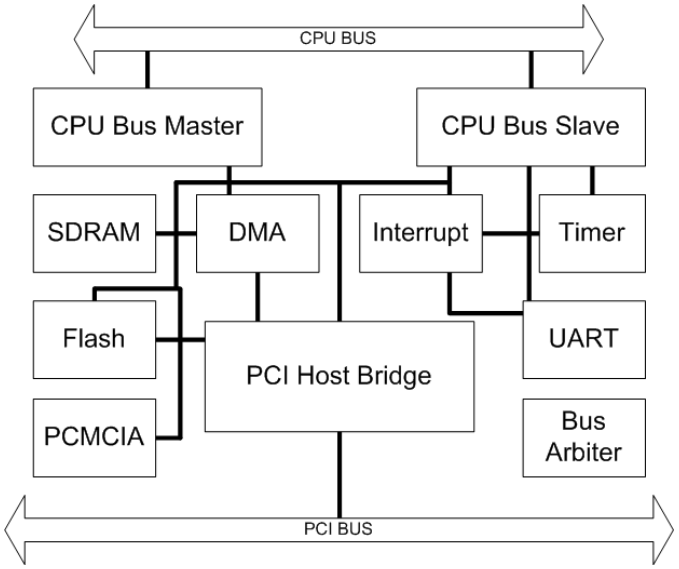


Figure 1.5: Subsystem made of IP cores.

2. *Hard cores:* Such cores are also called “hard macros”, because the core’s application function cannot be meaningfully modified by chip designers. Transistor layouts must obey the process design rules of the target foundry and, hence, hard cores delivered for one foundry process cannot be easily ported to a different process or foundry.

As I will show in the following section, Xilinx, for example, is shipping chips with a fixed hard core *PowerPC 440 block*, to handle the most complex and memory-intensive computing applications, to which soft cores can be added at will (from a components list), to perform specific tasks.

1.1.4. Field-Programmable Gate Arrays

The Field-Programmable Gate Arrays (FPGA) are integrated circuits designed to be configured by the customer or designer after manufacturing; hence, "field-programmable". In fact, they can be reprogrammed multiple times: they feature a reconfigurable architecture, consisting of an array of logic blocks and an interconnection network. The functionality and the interconnection of the logic blocks can be modified by means of programmable configuration bits. The FPGA configuration is generally specified using a HW description language (HDL), similar to that used for an application-specific integrated circuit (ASIC).

In the last years, we are witnessing a large growth in the amount of research being addressed worldwide into field-programmable logic and its related technologies. As the electronic world shifts to mobile devices [KPPR00;

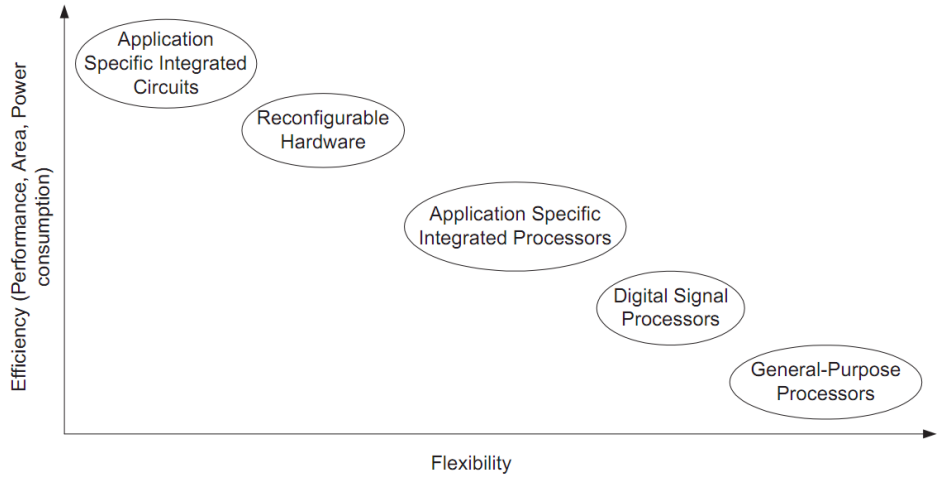


Figure 1.6: Comparison of computing platforms.

Rab00; PG03], reconfigurable systems emerge as a new paradigm for satisfying the simultaneous demand for application performance and flexibility [WVC02]. Reconfigurable computing systems [GK89] represent an intermediate approach between general-purpose and application-specific systems.

For GPPs, the same HW can be used for executing a large class of applications; however, it is this broad application domain which limits the efficiency that can be achieved. ASICs, on the other hand, are optimally designed to execute a specific application and, hence, each ASIC has superior performance when it executes its task but, since it has a fixed functionality, any post-design optimizations and upgrades in features and algorithms are not permitted. Reconfigurable systems potentially achieve a similar performance to that of customized HW, while maintaining a similar flexibility to that of general purpose machines.

Figure 1.6 shows a graphic comparison of different computing platform types in terms of efficiency (performance, area and power consumption) versus flexibility. Reconfigurable computing represents an important implementation alternative since it fills the gap between ASICs and microprocessors.

In modern FPGAs, the availability of an increasingly large number of transistors [biba], provides the silicon capacity to implement full MPSoCs containing several host processors (hard and soft cores), complex memory systems, and custom IP peripherals, combining a wide range of complex functions in a single die [PG03].

New design methodologies, like IP-cores-based design [RSV97], allow simple system creation. Users can create complex MPSoCs in a matter of minutes by simply instantiating different IP components included in preexisting libraries. Thanks to the standarization of system interconnects, the designer

can select the components, and interconnect them in a plug-and-play fashion. If needed, new components can be easily created, thanks to the existence of tools to validate and debug custom-made cores. For microprocessor-based designs, FPGA manufacturers provide tools supporting the codesign, where system SW can be developped and debugged at the same time as the HW.

1.2. State-of-the-art in MPSoC design

The fast evolution of process technology is reducing more and more the time-to-market and price [JTW05], which does not permit anymore complete redesigns of complex multi-core systems on a per-product basis. We all know how fast a product can become "the coolest device of the moment" like, for example, videoconference phones or pocket PCs. If a company does not have the product ready for "that moment" but after a delay of several months, when the product finally reaches the market it will surely present important losses with respect to the initial expectations. Surveys have demonstrated that the losses in the total gain of a product are more affected by a late appearance in the market, rather than by an increase in the final cost. In this scenario, *Multi-Processor Systems-on-Chip (MPSoCs)* have been proposed as a promising solution, since they integrate in one single-chip different complex components (IP-Cores) that have been already verified in previous designs (normally by third parties).

In this context, there is a need for design methodologies and implementation tools that permit the development of new high performance multimedia embedded systems in a very short time while ensuring design correctness. They must support the fast and flexible creation of prototypes, incorporating the last trends appeared; specially, the implementation under new constraints like the power minimization. Currently, there is a big effort aimed at automating the whole design flow for embedded systems.

Overall, designing MPSoCs is a complex task. Even if we fix the IP cores to be used, the exploration space is still huge. Designers must decide multiple HW details, from high level aspects (the frequency of the system, the location of the cores, or the interconnect), to low-level physical ones (the routing of the clock network, the technology used at the foundry, and so on). On top of this, comes the SW: whether the system will run bare-C applications, or a full featured OS, are decisions that must be accounted for at design time.

1.2.1. Power, temperature and reliability problems

One of the main design challenges in MPSoC design is the fast exploration of multiple HW and SW implementation alternatives with accurate estimations of performance, energy and power to tune the architecture at an early stage of the design process, because the aforementioned decisions

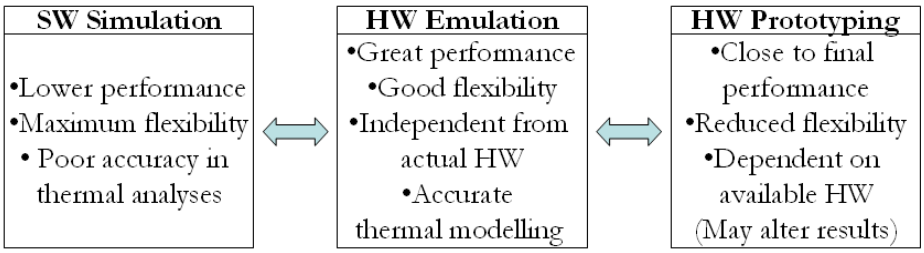


Figure 1.7: Different alternatives for MPSoC design space exploration.

(cf. previous section) will not only affect the final performance of the system; there are also other implications, such as the physical size of the chip, the power consumption, or the temperature and reliability of the components [Cla06]. Several tools and frameworks have been developed aimed at guiding designers in the exploration of the MPSoC design space.

Regarding thermal modeling, [SSS+04] presents a thermal/power model for super-scalar architectures that predicts the temperature variations in the different components of a processor. It shows the subsequent increased leakage power and reduced performance. [SLD+03] has investigated the impact of temperature and voltage variations across the die of an embedded core. Their results show that the temperature varies around 13.6 degrees across the die. Also, in [LBGB00], the temperature of FPGAs used as reconfigurable computers is measured using ring-oscillators, which can dynamically be inserted, moved or eliminated. This empirical measurement method is interesting, yet it is only applicable to FPGAs as target devices.

Overall, these works clearly prove the importance of the hotspots in high-performance and reconfigurable embedded systems, and the need for temperature-aware design and tools to support it. Moreover, it is clear that performance, power, temperature, reliability, etc. issues have to be addressed at design-time to reach the market on time.

In the next two sections, I describe the most important tools and frameworks available to designers for MPSoC exploration. I have categorized them into SW simulators and HW emulators. HW prototyping, despite being also a valid and useful approach, is not considered in my study, since is too close to the final implementation, and I am only concerned about methodologies that can be applied early in the MPSoC design cycle. Figure 1.7 compares the three alternatives.

1.2.1.1. Design space exploration through SW simulators

From the SW viewpoint, solutions have been suggested at different abstraction levels, enabling trade-offs between simulation speed and accuracy:

First, fast analytical models have been proposed to prune very distinct

design options using high level languages (e.g., C or C++) [BWS+03]. Also, full-system simulators, like Symics [MCE+02], have been developed for embedded SW debugging and can reach megahertz speeds, but they are not able to capture accurately performance and power effects, that depend on the cycle-accurate behaviour of the HW.

Second, transaction-level modeling in SystemC, both at the academia [PPB02] and industry [CoW04; ARM02], has enabled more accuracy in system-level simulation at the cost of sacrificing simulation speed (circa 100-200 KHz). Such speeds render unfeasible the testing of large systems due to the too long simulation times. Moreover, in most cases, these simulators are only limited to a number of proprietary interfaces (e.g., AMBA [ARM04a] or Lisatek [CoW04]).

Finally, important research has been done to obtain cycle-accurate frameworks in SystemC or Hardware Description Languages (HDL). For instance, companies have developed cycle-accurate simulators using post-synthesis libraries from HW vendors [Gra03; Syn03]. However, their simulation speeds (10 to 50 KHz) are unsuitable for complex MPSoC exploration. In the academic context, the MPARM SystemC framework [BBB+05] is a complete simulator for system-exploration since it includes cycle-accurate cores, complex memory hierarchies (e.g., caches, scratch-pads) and interconnects, like AMBA or Networks-on-Chip (NoC). It can extract reliable energy and performance figures, but its major shortcoming is again its simulation speed (120 KHz in a P-IV at 2.8 GHz).

Coming from either academic or industrial partners, a great variety of MPSoC simulators populate the market. Advanced SW tools can be added to them to evaluate in detail thermal pressure in on-chip components based on run-time power consumption and floorplanning information of the final MPSoCs [SSS+04]. Nevertheless, although these combined SW environments achieve accurate estimations of the studied system with thermal analysis, they are very limited in performance (circa 10-100 KHz) due to signal management overhead. Thus, such environments cannot be used to analyze MPSoC solutions with complex embedded applications and realistic inputs to cover the variations in data loads at run-time. On the other hand, higher abstraction level simulators attain faster simulation speeds, but at the cost of a significant loss of accuracy. Hence, they are not suitable for fine-grained architectural tuning or thermal modeling.

1.2.1.2. Design space exploration through HW emulators

One of the main disadvantages of using cycle-accurate SW simulators to study MPSoCs is the big performance drop that appears as we increase the number of processors in the system, due to the huge number of signals that need to be handled and kept synchronized during the simulation. Higher abstraction level simulators provide faster simulations, but the accuracy during

the evaluation of thermal effects is limited. An alternative to architectural simulators is HW emulation. The nature of the HW is parallel; thus, it allows the study of complex multi-processor environments without significant speed loss with respect to the mono-processor case. As the counterpart, HW is not so flexible as SW.

In industry, one of the most complete sets of statistics is provided by Palladium II [Cad05], which can accommodate very complex systems (i.e., up to 256 Mgate). However, its main disadvantages are its operation frequency (circa 1.6 MHz) and cost (around $1 million). Then, ASIC Integrator [ARM04a] is much faster for MPSoC architectural exploration. Nevertheless, its major drawback is the limitation to up to five ARM-based cores and only AMBA interconnects. The same limitation of proprietary cores for exploration occurs with the Heron SoC Emulator [Eng04]. Other relevant industrial emulation approaches are System Explore [Apt03] and Zebu-XL [EE05], both based on multi-FPGA emulation in the order of MHz. They can be used to validate intellectual property blocks, but are not flexible enough for fast MPSoC design exploration or detailed statistics extraction.

In the academic world, a relatively complete emulation platform for exploring MPSoC alternatives is TC4SOC [NBT+05]. It uses a proprietary 32-bit VLIW core and enables exploration of interconnects by using an FPGA to reconfigure the *Network Interfaces (NIs)*. However, it does not enable detailed extraction of statistics and performing thermal modeling at the other two architectural levels, namely memory hierarchy and processing cores. Another interesting approach that uses FPGAs to speed up a SW simulation (co-verification) is described in [NHK+04]. In this case, the FPGA part is synchronized, in a cycle-by-cycle basis, with the SW part (implemented in C/C++ and running in a PC) by using an array of shared registers located in the FPGA that can be accessed from the PC. This work shows a final emulation speed of 1 MHz, outlining the potential benefits of combined HW-SW frameworks. Finally, the RAMP (Research Accelerator for Multi-Processors) [AAC+05] project is another example that also exploits a hybrid HW-SW infrastructure.

1.2.2. Power, thermal, and reliability management techniques

Using the existing frameworks and tools to study the behaviour of MPSoCs, many design solutions have been proposed to tackle the problems of high on-chip power consumption and temperatures, and lack of reliability, using both architectural adaptation and profiling-based techniques:

In [SSS+04], it is proposed the use of formal feedback control theory as a way to implement adaptive techniques in the processor architecture. In [ZADM09; ZADMB10], a predictive frame-based Dynamic Thermal Management (DTM) algorithm, targeted at multimedia applications, is presen-

ted; it uses profiling to predict the theoretical highest performance within a thermally-safe HW configuration for the remaining frames of a certain type. Also, [BM01] performs extensive studies on empirical DTM techniques (i.e., clock frequency scaling, DVS, DFS, fetch-toggling, throttling, and speculation control) when the power consumption of a processor crosses a predetermined threshold (24W). Its results show that frequency scaling and DFS can be very inefficient if their invocation time is not set appropriately. At the OS level, [RS99] stops scheduling hot tasks when the temperature reaches a critical value. In this way, the CPU spends more time in low-power states, and the temperature can be either locally or globally decreased.

Recent studies have demonstrated that an intelligent placement of cores can reduce the thermal gradients inside the chip. This leads to interesting research lines for future MPSoCs, like power-aware synthesis and temperature-aware placement [CW97; CS03; GS05]. In this case, the temperature issues are addressed at design-time to ensure that circuit blocks are placed in such a way that they even out the thermal profile; therefore, improving the system robustness and reliability. Alternatively, by adding run-time techniques (SW or HW based) for limiting the maximum allowable power or temperature dynamically, we can reduce the packaging cost as well as extend the chip lifespan. A significant bottleneck of all the run-time dynamic methods is the performance impact associated with stalling or slowing down the processor [SSS+04]. Multi-processor chips bring new opportunities for system optimizations. For example, advanced temperature-aware job allocation and task migration techniques have been proposed (e.g. [DM06], [CRW07]) to reduce thermal hot spots and temperature variations dynamically at low cost.

Overall, in any MPSoC, designers need from exhaustive system profiling to discover the best tradeoff: performance vs peak temperature or cost. Moreover, since each design is different, the goals are not always the same; sometimes, there is a need for performance at no matter what cost while, in another situation, the designer may be looking for the cheapest chip, the higest power-efficiency or the most reliable design.

1.3. Motivation and goals of this thesis

After this introduction, it is clear that MPSoC designers are in great need of tools that help them ease the design process. One of their main design challenges is the fast exploration of multiple HW and SW implementation alternatives with accurate estimations of performance, energy, power, temperature, and reliability to tune the MPSoC architecture at an early stage of the design process.

In the previous sections, I cited many frameworks that are already available for designers, and I classified them into SW simulators and HW emulation frameworks:

SW simulators are very accurate, but typically inappropriate to perform long thermal simulations due to their limited performance. Thus, such environments cannot be used to analyze complex MPSoC solutions. Higher abstraction level simulators (e.g., at the transactional level) provide faster simulations, but the accuracy during the evaluation of thermal effects is limited, so they are not suitable for fine-grained architectural tuning or thermal modeling.

On the other hand, we have MPSoC HW emulation frameworks. The available ones are usually very expensive for embedded design, not flexible enough for MPSoC architecture exploration and, typically, offer proprietary baseline architectures, not permitting internal changes. Therefore, thermal effects can only be verified in the last phases of the design process, when the final components have been already developed.

The main idea behind this research work is to create a new design flow that will reduce the complexity of the MPSoC development cycle. To this end, it will be introduced my new HW-SW FPGA-based emulation framework, abreviated as the Emulation Platform (EP) from now on, which allows designers to explore a wide range of design alternatives of complete MPSoC systems at cycle-accurate level, while characterizing their behaviour/power/temperature/reliability at a very fast speed with respect to MPSoC architectural simulators.

The EP is a hybrid framework that consists of two elements: On one side there is a FPGA, where the MPSoC under development is mapped, instrumented, and profiled; On the other side, there is a PC, that receives the statistics coming from the emulation, and uses them to estimate the power/thermal/reliability profile of the final chip.

One of the most important features of the EP is that the framework will be conceived from the beginning to be versatile and flexible, so that it could be adapted to the new market demands by adding new state-of-the-art features. In fact, in Chapter 3, I will exemplify this important characteristic by incorporating *a posteriori* to the platform a novel approach for fast transient thermal modeling, and the support to analyse 3D MPSoCs with active (liquid) cooling solutions.

1.3.1. Thesis structure

The rest of this thesis is organized as follows:

In Chapter 2, I describe in detail the HW part of the EP, that runs onto the FPGA. First, I explain the type of MPSoCs that can be instantiated, and the different components that can be used. Next, I show the mechanism added around the system under study in order to perform detailed profiling of the execution. Basically, additional HW components are included to monitor the MPSoC and extract information that is then sent from the FPGA to a

host PC for ulterior analysis.

Chapter 3 describes how this information is processed in the PC. From the simplest option, that consists on logging down all the statistics, and present a report once the emulation is finished, to more advanced mechanisms like, for example, estimating the reliability of the system, and returning this information to the FPGA so that the emulated MPSoC can elaborate a balancing policy to extend the lifespan of its components. The SW developed for the PC estimates power, temperatures, and reliability numbers of the final MPSoC based on the data received from the FPGA. Through the different sections, I detail the process of how this input is converted into output using advanced mathematical models.

The HW running on the FPGA, explained in Chapter 2, and the SW models that run on the host PC, explained in Chapter 3, are put together in Chapter 4, that describes the platform integration: how to instantiate and interconnect all the components and perform an emulation. It describes the emulation flow that allows designers to speed-up the design cycle of MPSoCs, the design considerations that arise when putting the different parts together, and the HW and SW elements necessary to setup an EP instance.

After describing the platform in detail, I illustrate the benefits of my EP through examples and experiments. Chapter 5 presents three case studies aimed at showing the practical use of the EP to evaluate the impact that different HW-SW design alternatives have into the performance, power, thermal, and reliability profile of the final chip. I show how the tool can be used to choose the right floorplan, the best package, or decide if it is worh implementing DFS support in a new MPSoC (Experiment 1). In cases when the chip is already manufactured, designers can use the EP, for instance, to develop a reliability enhancement policy aimed at extending the lifespan of a processor by simply changing the way the compiler allocates the HW registers (Experiment 2), or to ellaborate system-level thermal management policies like, for example, a Multi-Processor Operating System that performs task migration and task scheduling to effectively regulate the temperature (Experiment 3).

Finally, in Chapter 6, I synthesize the conclusions derived from this research, and the contributions to the state-of-the-art in MPSoC development. For completeness, I also propose some enhancements to the EP, and present several application fields (open research lines) that will benefit from this work.

Appendix A includes a Spanish summary of this dissertation, in compliance with the regulations of the *Universidad Complutense de Madrid*.

Chapter 2

The HW Emulation Platform

In the introduction, I described one of the main design challenges of MPSoC designers: the fast exploration of multiple hardware (HW) and software (SW) implementation alternatives with accurate estimations of performance, energy and power to tune the MPSoC architecture at an early stage of the design process. It was introduced, as well, my HW/SW Field-Programmable Gate Array (FPGA) -based emulation framework: the Emulation Platform (EP), whose structure is depicted in Figure 2.1, and comprises the following three components:

1. **The Emulated System:** This is the MPSoC being optimized; the system under observation. Typically, this design is tuned to meet the design constrains.

2. **The Emulation Engine:** It is the HW architecture that hosts the *Emulated System*. It is in charge of stimulating it, and extracting run-time statistics from three key architectural levels: processing cores, memory subsystem and interconnection elements, while real-life applications are executed on the MPSoC. It is also connected to the host computer for data interchange. The idea is similar to that of the SW architecture simulators, where we have the simulator itself (e.g.: the Simplescalar [ALE02]) and, then, we have to fit inside the SoC architecture to simulate.

3. **The SW Libraries for Estimation:** Running in a general purpose desktop computer, they calculate power, temperature, reliability figures, etc. based on the statistics received at run-time from the *Emulation Engine*.

In the normal operation flow with the EP, the user downloads the complete framework (both the *Emulated System* and the *Emulation Engine*) to the FPGA. Then, a start command is issued, and the emulation starts. The

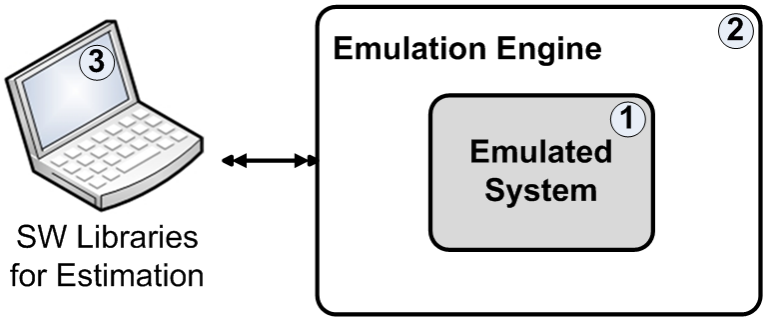


Figure 2.1: High-level view of the Emulation Platform.

statistics generated are sent through a communications port to a host computer, that logs them down, and uses them as the input to the *SW Libraries for Estimation*, that calculate power, temperatures, reliability numbers, etc. of the final MPSoC. The emulation process is autonomous; i.e.: the FPGA is automatically synchronized with the SW in the PC, so that they interchange data continously in a bidirectional way. In addition to this, the user can interact with the system at any point: a set of control commands can be issued to the EP from the host computer, through a separate channel. At the other side, the *Emulation Engine* processes the orders, and proceeds accordingly.

Regarding the synchronization FPGA-computer, the emulation is divided into *Emulation Steps*: it runs for a fixed amount of cycles; then, it pauses and performs the information exchange (upload/download); once finished, it resumes for the next *Emulation Step*. Full details are provided in Section 2.2.1. During the whole process, the host computer provides visual feedback of the emulation evolution in real-time.

This chapter describes the HW components of the EP; that is, the elements of the tool that reside inside the FPGA, while the SW components, i.e., the *SW Libraries for Estimation*, are explained later on, in Chapter 3.

In the following sections, I describe first the *Emulated System*, explaining the different types of cores that can be instantiated. Next, I detail the *Emulation Engine*, namely, the architecture of my emulator.

2.1. The Emulated System

The baseline architecture of an MPSoC typically contains these three elements:

1. Processing cores like, for example: PowerPC, Microblaze, ARM, or VLIW cores.

2. A memory architecture: instruction and data memories, L1 and L2 caches, scratchpads, and main memories (private or shared between processors).

3. Interconnect mechanisms to communicate the system elements: multi-level buses, crossbars, or NoCs.

Figure 2.2 shows an example of such architecture. It is a gaming platform designed at ARM. In the block diagram we can observe a couple of Cortex-A9, as the main processors, both containing the NEON coprocessor, designed to accelerate the signal processing operations. Through an AMBA AXI bus, they also have access to two Mali multimedia accelerators, several on-chip memories (flash, ROM...), and input/output interfaces (USB, memory cards, audio, debug, camera, the SDRAM external memory...). There are additional ARM processors (Cortex M0, ARM968...) to handle special operations, like the touchscreen input, the high definition audio, and the bluetooth and wifi communications.

In the EP, any element of the *Emulated System* is finally translated to a netlist, and mapped onto the underlying FPGA; therefore, the accepted input formats to specify them range from netlists, directly, to other HDL languages offering higher levels of abstraction, like Verilog, VHDL or Synthesizable SystemC. Attending to the decription of the components, I classify them into fully specified or modeled, and proprietary or public. However, before explaining this concepts, I must emphasize the difference between prototyping and emulation.

2.1.1. Prototyping vs. Emulation

In integrated circuit design, *hardware emulation* is the process of imitating the behaviour of one or more pieces of hardware (typically a system under design) with another piece of hardware, typically a special purpose emulation system. On the other hand, *hardware prototyping* is the process of obtaining an actual circuit with a design very close to the final one. While HW emulation may include modeled components, at an early stage of the design cycle, HW prototyping, however, requires the final components to be available, and it is typically made at the last stages of the design cycle.

Let us suppose that we use in our MPSoC a module available in a components library provided by a second party. This module is a mature product (already debugged, verified, etc.) that has been implemented in several chips. It has been designed in VHDL, and the license agreement specifies that the whole source code is available to us. Then, we can directly instantiate it into the *Emulated System* so that it will be mapped into the FPGA. This is a case of prototyping. It must be noted that, although the behaviour of the module will be identical to the silicon version, some parameters, like the maximum working frequency, will most likely differ.

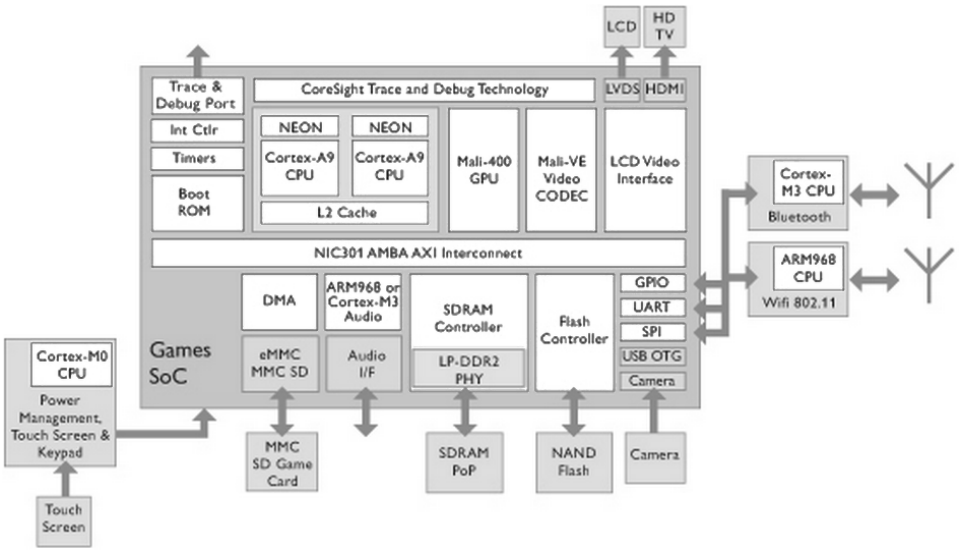


Figure 2.2: The typical ARM gaming platform, which is an example of heterogeneous MPSoC architecture.

In a different situation, maybe the final component is not yet implemented. We can think, for example, of an square root calculator. While another party is implementing it, we can create a model that will behave in the same way but, instead of actually performing the calculations, will fetch the results from a pre-created lookup table. The internals of the module will differ from the final one. However, from the point of view of the interaction with the rest of the *Emulated System*, it will be the same. Notice that, although the result is inmediatly available in the table, we can model the desired latency by using idle wait states. This is a case of emulation. As opposed to prototyping, the components do not have to be fully specified; we can work with models. By using models of the missing components, we can debug the whole system with live data.

To summarize, HW prototyping deals with designs, and HW emulation with models. In my platform, both designs and models can be instantiated; i.e., when the prototype of a memory, core, bus, etc. is not ready, I can use its model instead. The key to achieve this is the use in the EP of a module called the *Virtual Platform Clock Manager*, that helps us in the task of hiding extra latencies, allowing designers to model memories and other modules that are not yet available. The mechanism is explained with the example of a memory controller in Section 2.2.1.

2.1.2. MPSoC Components

The previous dissertation showed the differences between prototyping and emulating a system. This definition naturally creates a way to classify the components of an emulated MPSoC, attending to how they are specified:

1. **Fully Specified Components:** These are the final components that will be included in the manufactured chip. When used in the emulation, they provide the highest accuracy for the statistics. They are normally taken from IP cores libraries, or designed ad-hoc for a specific MPSoC.

2. **Modeled Components:** Also named *Virtual Components*, are modules that only live inside the emulation. I use them when the real component is not yet implemented, or in situations when it cannot be included in the platform. Eventually, in the final implementation, they will be replaced by a final component, that can be a synthesizable element, a hard core already implemented in silicon, or even another chip containing the functionality that was previously modeled during the emulation.

A mix of the two flavours is possible: we can have a partially specified component, where part of it is fully specified, and the other part is still modeled. Also, we can start by modeling a component that is not yet available and, later on, in advanced stages of the verification cycle, replace its model by the real component in the EP. Finally, we can also use models if we do not have interest in accurately studying the module, since a model is, normally, faster to synthesize, and occupies less resources in the FPGA.

2.1.2.1. An example of Modeled Components

A sensor is a device that measures a physical quantity and converts it into a signal which can be read by an observer or by an instrument. If an MPSoC needs to know the temperature conditions, for example, in HW prototyping we would attach a temperature sensor to our system so that we can directly access the data. With emulation, everything is more flexible: we do not need the sensor, neither we are restricted to the real measurements from the ambient. We can recreate (emulate) our own conditions.

With this idea in mind, I have implemented sensors as modeled components. The final MPSoC, implemented in silicon, will contain real sensors to adquire external data from the real world. For instance, it can get the light conditions, temperature, or fabric degradation, provided it has access to the appropriate sensor. Since we are emulating the MPSoC, my sensors can be read from the MPSoC in they same way as the real ones. However, the information they return is injected by the *Emulation Engine*, so we can recreate a context for the emulation. If we have a temperature sensor, for example,

we can send to the FPGA a data trace containing the thermal conditions we want to model. Whenever the sensor is accessed from the *Emulated System*, it will return the next value of the trace. In this way, the content of the sensors is another input parameter that we can set in our emulation.

Another example of a modeled component is a new memory that has not been manufactured yet. Imagine that this new memory is to be twice as fast as the fastest memory that we have on the market now. We can still model it in the EP using a standard memory. Two different approaches are possible:

1. Scaling down the frequency of the whole system to be the half, so that the speed ratio would be the same as the system running full speed with the new memory.
2. We can still clock the system at its original frequency, and hide the extra cycles whenever there is a memory access, by keeping track of the elapsed cycles.

As I explain in Section 2.2.1, the solution adopted in the EP is the second one. It has the benefit of only stopping the system when it is strictly required (when this special memory is accessed); thus, it does not cut to half the emulation speed. In the worst case, if the memory is accessed every cycle, the second solution still has benefits: Imagine the case when this special memory needs a frequency 2 times slower than the standard one, and another module needs a frequency 3 times slower. We can wait 3 cycles until both elements are ready. Had we taken solution one, it would have required to slow down the frequency to the least common multiple: 6.

2.2. The Emulation Engine

The *Emulation Engine* (cf. Figure 2.3) is made of the following elements:

1. **The Virtual Platform Clock Manager (VPCM):** Generates and keeps synchronized the different clock domains of the *Emulated System*.
2. **The Statistics Extraction Subsystem:** Extracts the information from the *Emulated System* in a non-intrusive way (i.e., transparently connected).
3. **The Communications Manager:** Handles the bidirectional packet-based communication FPGA - computer.
4. **The Emulation Engine Director:** Controls the whole system, orchestrating the emulation: controlling the *Emulated System*, directing the statistics extraction, and synchronizing the FPGA and the host computer.

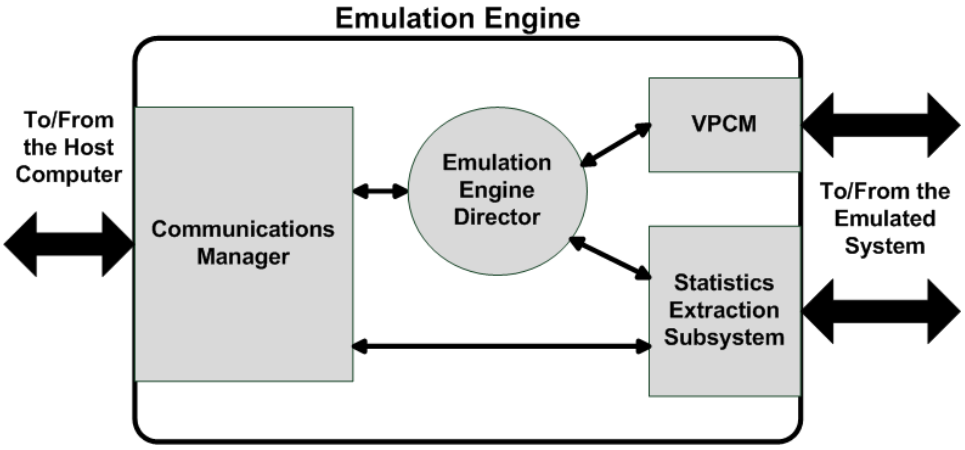


Figure 2.3: Parts of the Emulation Engine.

2.2.1. The Virtual Platform Clock Manager

Cycle-accurate simulators are normally implemented as event-triggered engines, where a clock event triggers a cascade of signal updates that goes on until all the signals become stable. The simulator then awaits ready to simulate the next clock cycle. A similar idea is the basis of the EP, where the emulation only advances every time the rising edge of a special clock signal reaches a component of the *Emulated System*. It, then, generates some signal transitions that, as opposed to the simulator, occur in parallel. I have denominated this special clock *The Virtual Clock* (VC); as opposed to the regular clock, also known as the system clock, real clock or just "the clock".

An *Emulated System* is composed of one or several VC domains (see Figure 2.4), each of them clocked by a different VC. A VC domain is, then, a set of components that share a common clock; It can contain just one element (a memory, a processor, a core...), or a complete system (processors + buses + cores). The VPCM is the element used to generate the multiple VCs. Each VC is controlled according to the needs. More exactly, a VC can be stopped, resumed, and scaled at any moment, controlling the evolution of the emulation. To simplify the concept, a VC can be seen as a normal clock that can be inhibited during some cycles, so that the *Emulated System* "receives clock cycles under demand".

From the previous section, we know that the modeled components are key in the EP. Some of them try to model the behaviour of components that are not yet available. In the case of a HW multiplier, for example, we could use iterative additions to achieve the same result. The difference with respect to the final module is that it may need extra cycles to complete the operations. In this situation, the modeled HW multiplier sends a signal to the VPCM, so that the VC of the rest of the components is inhibited until the result

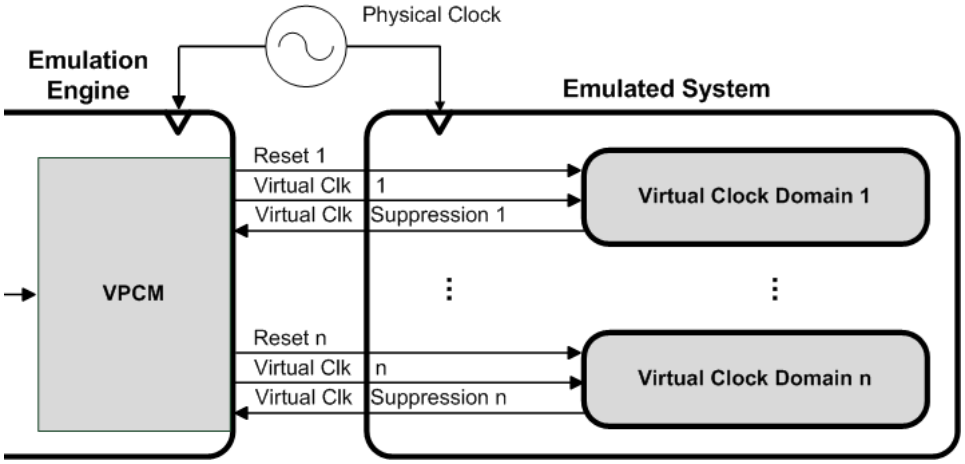


Figure 2.4: Detail of the clock management system.

is ready. Without the VCs, we should be talking about HW prototyping, instead of emulation.

The VPCM generates as output the *Virtual Clk* signals shown in Figure 2.4. Observe that it also has the capacity to reset the *Emulated System* rising the *Reset* line. The VPCM receives two different types of input signals: First, the physical clock generated by an oscillator. Second, one signal from each VC domain (*Virtual Clk Suppression 1..n*) used to request a VC inhibition period if any module is not able to return the requested value on time. This *Virtual Clk Suppression 1..n* signal may not exist in the case when a domain contains only fully specified components. However, that domain will still be clocked by a VC, because the VPCM must be able to stop it, to wait for other domains.

Thanks to the use of multiple VC domains, the emulation of MPSoCs can be done for different physical features than those of available HW components. Once the respective *Virtual Clk Suppression 1..n* signal is high, the corresponding *Virtual Clk* signal of the affected domain/s is frozen. Then, the stopped domains preserve their current internal state until they are resumed by the VPCM, when the module that caused the clock inhibition is ready; for example, a memory controller informs that the information requested is available in the accessed memory.

Following with the example of the memory, this mechanism allows us to implement the corresponding memory resources either in internal FPGA memory (optimal performance) or with external memories (bigger size), while balancing emulation performance and use of resources. For instance, if the desired latency of main memories are 10 cycles, but the available type of memory modules in the FPGA are slower (e.g., use of DDR instead of SRAMs),

the VPCM can stop the clock of the processors involved at run-time; thus, it can hide the additional clock cycles required by the memory. The modeled components contain some extra logic to generate the VC suppression signal. Internally, they keep track of the elapsed time and compare it with the user-defined latencies.

Regarding the timing of the emulation, it is discretized into what I have called "Emulation Steps": the *Emulated System* runs for a fixed amount of cycles, then, it is paused so that the information interchange (both upload and download) FPGA-computer takes place. When ready, the emulation is resumed, and the next Emulation Step starts. The number of cycles per Emulation Step can be configured by the user. It depends on the amount of information we can store in the FPGA (the size of the buffers), and the required update frequency of the estimation models that run on the host computer:

The VPCM keeps track of the number of cycles elapsed and, once it reaches the predefined number, it freezes the generation of the VCs, and signals the *Emulation Engine Director*. This module, then, empties the FPGA buffers, sending the data to the host computer. After that, it signals back the VPCM so that it resumes the VC generation. When using the closed-loop SW estimation models, there is an additional intermediate step, that involves receiving data in the FPGA from the computer (e.g.: estimated temperatures for the sensors).

2.2.2. The Statistics Extraction Subsystem

The Statistics Extraction Subsystem extracts information from the *Emulated System*. The main feature pursued in its design is its transparent inclusion in the basic MPSoC architecture to be evaluated, and with negligible performance penalty in the overall emulation process. With this purpose, I have implemented HW Sniffers that monitor internal signals of the system cores and the external pinout of each device included in the emulated MPSoC. In Figure 2.5 we can see many of these devices (marked as *Sniffer 1..4*) attached to the corresponding monitored cores (in a stripped pattern).

All the sniffers are attached to a bus, The Statistics Bus, designed with a simple arbitration policy that maximizes the bandwidth to collect the data from the buffers (i.e.: logged data in the internal memory of the sniffers) without incurring into extra delays or penalties. It also enables to access the sniffers for control purposes; for example, they accept commands (either from the user or from the *Emulation Engine Director*) like enabling/disabling the collection, resetting the statistics, or changing internal parameters. For a complete list of commands and actions, refer to Section 2.2.4.

In Figure 2.6, we can appreciate both the *HW Sniffers* and the *Statistics Bus*, and how they are connected. We can also see the third element that

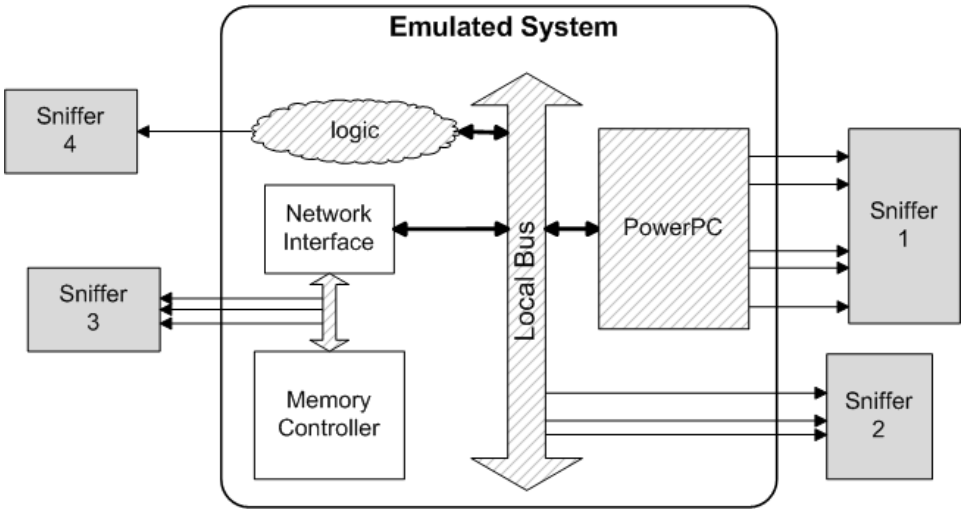


Figure 2.5: Emulated System with associated sniffers.

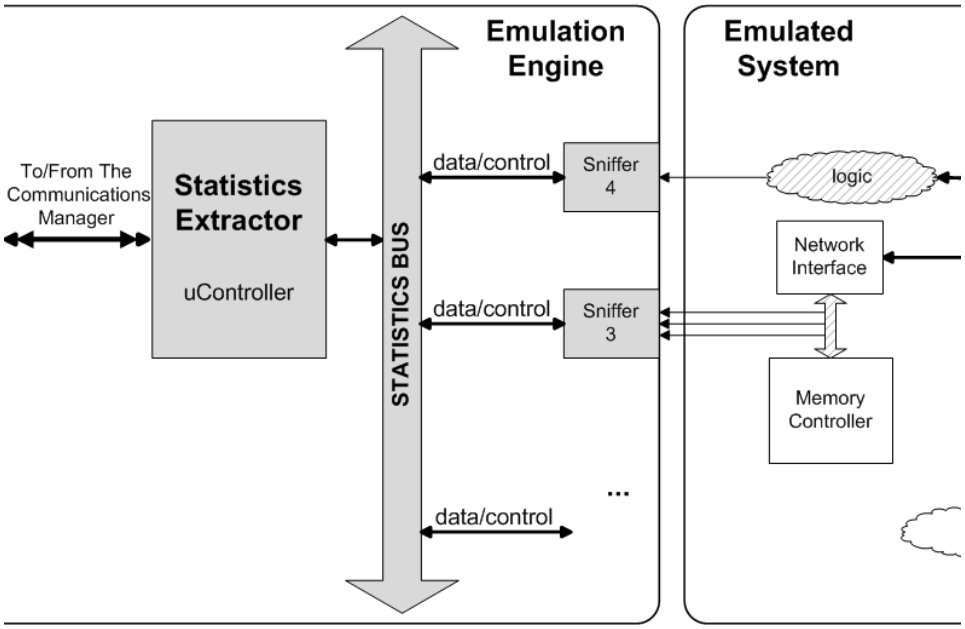


Figure 2.6: Schema of the Statistics Extraction Subsystem.

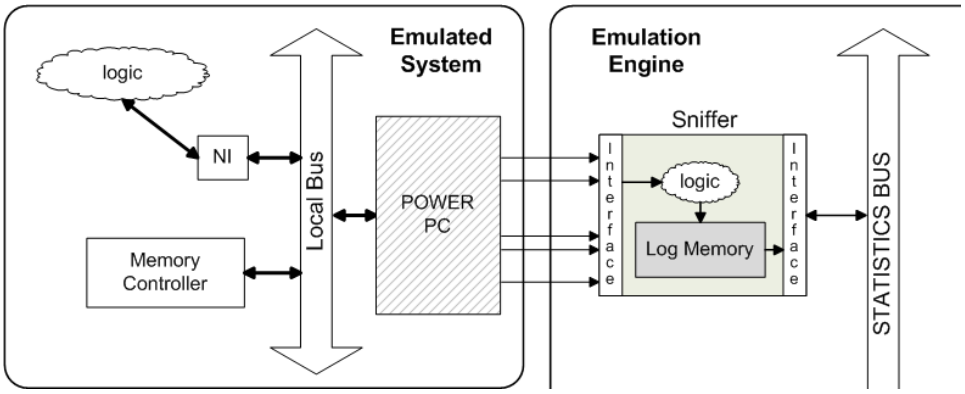


Figure 2.7: Details of the structure and connection of a template sniffer.

completes the Statistics Extraction Subsystem: The Statistics Extractor, a microcontroller in charge of interfacing the sniffers (through the *Statistics Bus*) for information (data and control) interchange. The *Statistics Extractor* has also a direct connection to the Communications Manager (see link in the left part of Figure 2.6) so that it can access the outside world (i.e.: outside the FPGA). This enables the extraction of statistics to the host PC, and the reception of information (data and control) from it.

2.2.2.1. HW Sniffers

The *HW Sniffers* are elements that transparently extract the statistics from each component of the *Emulated System*; that is, without interfering, neither modifying, the normal behaviour of the core under study. From a design point of view, all the sniffers in the platform share a common structure (cf. Figure 2.7): They have a dedicated interface to capture internal signals from the module they are monitoring, logic that converts this signal activity into meaningful statistics, a small local memory (buffer) to store the statistics, and a connection to my custom *Statistics Bus*, that allows the extraction of the logged data.

To create a new sniffer, the designer first needs to define what to monitor in the component under observation. Templates of sniffers are provided that cover the most common situations. The templates are VHDL files containing the interface to the *Statistics Bus*; i.e., the arbitration and communication logic is already there so they *understand* the custom protocol, and, on the other side (cf. Figure 2.7), the interface with the monitored module; i.e., sample of code that monitors a signal, and carries out some processing on it. The designer should put his signals there, instead. Depending on the type of sniffer we want to instantiate, the referred processing will be just storing the value of the signal, counting the number of transitions, checking

for protocol violations, etc. Later, in this section, I provide more detailed examples. Thus, these templates should be used as a skeleton that has to be customized depending on the nature of the monitored module.

According to how much information is available from a module, we can face the following situations:

1. **Full VHDL description:** The complete code of the core is available. This is the case, for example, of user-created cores, or when we use licensable cores with source code access. Being the most favourable case, we can dig as much as we want into the module internals and monitor every present signal.

2. **Partial access to the core:** When using modules developed by third parties, normally, we only have access to part of the code, or not even that, if the core ships as a netlist. However, the core designer often provides an interface to access the internal state, or monitor the events that occur inside. This characteristic, intended for debugging, profiling, or synchronization, can be reused for sniffing purposes.

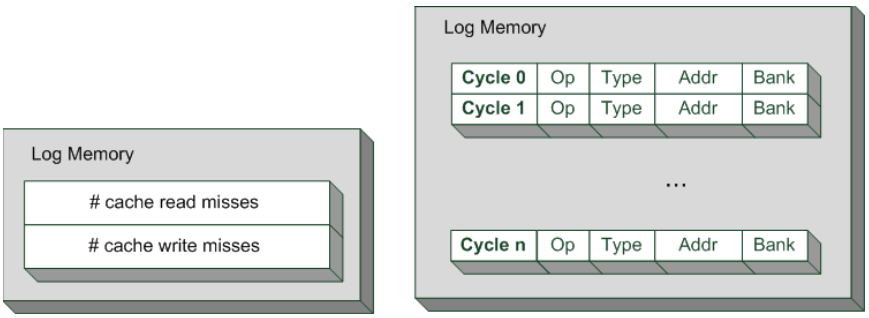
3. **Black box model:** In some cases, there is no possibility to access the module (e.g.: the whole core may be encrypted, or provided as a silicon block). Thus, very little information can be extrated by snooping the external ports or connections.

Currently, I provide five different templates of sniffers that cover the most common situations:

1. **Event-logging Sniffer:** Exhaustively logs all the events, selected by the designer, that occur in the platform. Logging detailed events means storing a message such as: "In cycle 24, there was a byte read request to address 0x8000, of bank 2 of the memory controller".

2. **Event-counting Sniffer:** Counting events is creating a summary that specifies, for example: "this core was accessed 750 times", or "the memory controller registered 320 reads and 470 writes during this emulation". They can also account for cache misses, bus transactions, etc. They generate more concise results than event-logging sniffers, and what typically designers demand from cycle-accurate simulators to test their systems.

3. **Protocol-checker Sniffer:** More intended for debugging, they are normally used to verify that the operations occur according to the specification. A bus deadlock detector, for example, will sit on the bus sniffing all the transactions and emitting error messages whenever a module enters a deadlock situation.



(a) Event-counting sniffer for a cache memory.

(b) Event-logging sniffer for a multibank SRAM.

Figure 2.8: Examples of the stored information inside the sniffers.

4. **Resource-utilization Sniffer:** Monitors the state of a link, module, etc. and reports an estimation of how saturated it is. For example, instantiated inside a network-on-chip, can monitor a router, and report whenever a channel utilization goes beyond 80 %.

5. **Post-processor Sniffer:** It is a special type of sniffer, that is always attached to an event-logging sniffer. It processes the stored data and converts them into different information. Using the example presented above, the one with the event-logging sniffer that logs down detailed memory accesses, a post-processor sniffer could attach to it, and infer the pattern of how the memory banks are accessed.

The most relevant sniffers in the EP are the event-logging and the event-counting sniffers. They store the necessary information to enable the power consumption, temperature, and reliability estimations of the emulated MPSoCs. Figure 2.8 shows examples of the kind of information that these sniffers can store.

The experimental results carried out with real-life MPSoC designs (cf. Chapter 5) indicate that, practically, an unlimited number of event-counting and event-logging sniffers can be added to the design without deteriorating at all the emulation speed. This is one of the advantages of using HW emulation, where the HW modules work in parallel, in contrast to HW cycle-accurate simulation systems, that sequentially simulate each of the modules and have the overhead of synchronizing them all.

The overhead in FPGA area is also quite small. For example, the amount of resources used by one event-logging sniffer is 14 slices in a Virtex II pro FPGA, which represents only the 0.1 %. For an event-counting sniffer is about 0.2 % (31 slices). However, in this case, the limiting factor will be the available onboard BRAM used for the buffers. The average size of the buffers

is 1KB, and there are 2MB of BRAM in this FPGA model.

Sniffer Examples

As an example to the reader, in this section I introduce the most significant sniffers implemented during the development of the platform.

- **Example of event-counting sniffer with partial access to the core:** For temperature monitoring, for example, *HW Sniffers* can measure the time that each processor spends in active/stalled/idle mode at run-time. When studying the PowerPC processors embedded in the Xilinx FPGAs, we have to take into account that they are physically implemented in silicon. Following the scheme in Figure 2.7, we see that many signals from the PowerPC core are sent to the sniffer.

  The PowerPC documentation [Xil10c], describes two sets of signals intended for execution trace and debugging (enumerated in Figures 2.9 and 2.10, respectively). By inspecting the debug signal *c405dbgstopack* and the two trace signals *c405trcevenexecutionstatus* and *c405trcoddexecutionstatus*, it is possible to determine the state of the processor at any cycle; thus, this case falls into the category *partial access to the core*. Since it is an event-counting sniffer, I include some logic to do the precise calculations, and store into the log memory a report with the number of cycles the processor spent in each state. To simplify things, for the particular case when the number of activity states is three: active/stalled/idle, the log memory will be reduced to just three registers. The first one will contain the number of cycles the processor spent in the active mode, and the numbers for the stalled and idle states will be stored in the second and third registers.

  In another scenario, the cores instantiated can be protected or encrypted, what means that we should get the information by sniffing from outside the core. On the other hand, when the full VHDL code of the module is available to us, we can exhaustively monitor every signal transition. As an example, in a complex core like the Leon3 [Gaib], we can precisely know which floating point units are active, and what registers are being accessed at every cycle.

- **Example of event-logging and event-counting sniffers with full VHDL description of the core:** When dealing with memories, we normally monitor the memory controller, so that we can observe the number and type of accesses (read/write, line/word...). In some cases, by sniffing the local bus interface, where the memory controller is connected, we can get most of the data. Figure 2.11, for example, shows

Signal	I/O Type	If Unused	Function
DBGC405EXTBUSHOLDACK	I	0	Indicates the bus controller has given control of the bus to an external master.
DBGC405DEBUGHALT	I	0	Indicates the external debug logic is placing the processor in debug halt mode.
DBGC405UNCONDDEBUGEVENT	I	0	Indicates the external debug logic is causing an unconditional debug event.
C405DBGWBFULL	O	No Connect	Indicates the PowerPC 405 writeback pipeline stage is full.
C405DBGWBIAR[0:29]	O	No Connect	The address of the current instruction in the PowerPC 405 writeback pipeline stage.
C405DBGWBCOMPLETE	O	No Connect	Indicates the current instruction in the PowerPC 405 writeback pipeline stage is completing.
C405DBGMSRWE	O	No Connect	Indicates the value of MSR[WE].
C405DBGSTOPACK	O	No Connect	Indicates the PowerPC 405 is in debug halt mode.
C405DBGLOADDATAONAPUDBUS	O	No Connect	Virtex-4 FX only. Valid load data transferred between the APU controller and PowerPC 405 core.

Figure 2.9: List of the PowerPC debug signals.

Signal	I/O Type	If Unused	Function
C405TRCTRIGGEREVENTOUT	O	Wrap to Trigger Event In	Indicates a trigger event occurred.
C405TRCTRIGGEREVENTTYPE[0:10]	O	No Connect	Specifies which debug event caused the trigger event.
C405TRCCYCLE	O	No Connect	Specifies the trace cycle.
C405TRCEVENEXECUTIONSTATUS[0:1]	O	No Connect	Specifies the execution status collected during the first of two processor cycles.
C405TRCODDEXECUTIONSTATUS[0:1]	O	No Connect	Specifies the execution status collected during the second of two processor cycles.
C405TRCTRACESTATUS[0:3]	O	No Connect	Specifies the trace status.
TRCC405TRIGGEREVENTIN	I	Wrap to Trigger Event Out	Indicates a trigger event occurred and that trace status is to be generated.
TRCC405TRACEDISABLE	I	0	Disables trace collection and broadcast.

Figure 2.10: List of the PowerPC trace signals.

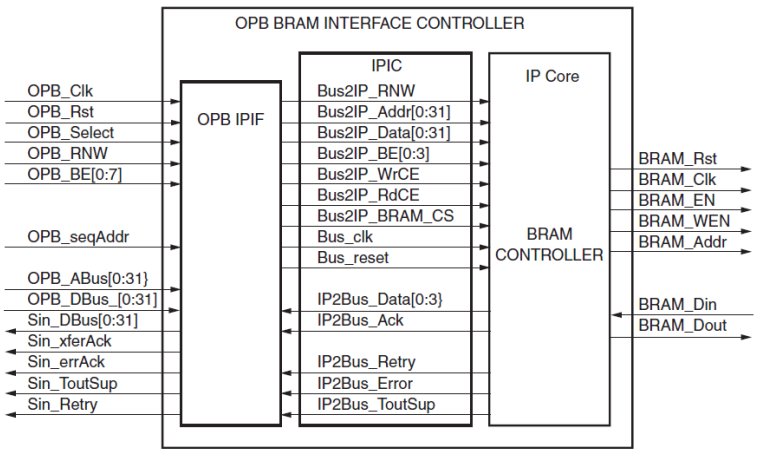


Figure 2.11: The OPB BRAM controller.

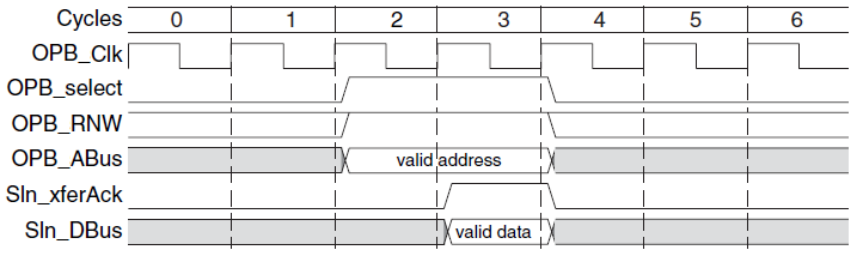


Figure 2.12: Temporization of an OPB read data transfer.

the schema of an OPB BRAM memory controller [Xil05], where we can appreciate the modules it contains, as well as the interface signals. The controller works as a slave on the OPB side, and a master on the BRAM side. We are interested on the OPB side. OPB stands for On-chip Peripheral Bus, and is one of the standard buses (created by IBM) available in Xilinx tools [Xil10a]. OPB IPIF is the interface adaptor between the IP and the OPB. The available signals are depicted in the figure; from them, we can infer if the access was a read or a write (signal OPB_RNW), the accessed address (signal $OPB\_ABus[0:31]$), etc. Figure 2.12 shows a read data transfer. The controller is accessed whenever signal OPB_Select goes high and there is a valid address in OPB_ABus (i.e., it falls within the module address range). In the example, signal OPB_RNW is high, indicating that it is a read access. One cycle later, signal $SIn\_xferAck$ goes high to indicate that the data is already available in the SIn_DBus signal.

While bus-sniffing is enough in the case of a scratchpad memory, for

complex memory hierarchies more information is needed; specially when we want to use an event-logging sniffer. In this case, we must monitor inside the memory controller to snoop the hits, misses, line replacements, or initialization states... In the simplest case, our sniffer will tell us the number of read and write accesses to the memory, and two registers will suffice (event-counting). If we want a complete report (event-logging), we will include extra elements to monitor detailed events, like the cycle number in which a cache line was replaced, or the data cache was flushed, etc.

- **Example of sniffers for interconnects:** At the interconnect level (buses or NoCs), the monitored values can vary from the number of bus transactions to the number of signals transitions. For packet-based interconnects, we typically calculate the number of packets interchanged, the average latency, or the packet sizes. The log memory inside the sniffer will vary according to the decisions taken.

- **Example of post-processing sniffer:** The sniffers included in the EP to estimate the power burnt in the cores. Chip manufacturers have estimations of how much power their cores dissipate in their different states. Roughly, the idea used to estimate the power dissipated in the *Emulated System* is to accurately monitor the states the cores are in, and add the equivalent power they would be burning in such state according to the manufacturer specifications. This technique is extensively used in SW cycle-accurate simulators that provide power and energy numbers. We find many examples in the literature. In Chapter 3, I explain in detail the basis, and how this technique was utilized to create the power model that is included in the *SW Libraries for Estimation*. My first implementation approach did all the calculations on the computer, post-processing the data received from the FPGA. Later on, once verified, and due to the simplicity of the process, I decided to integrate the power calculation into the HW. I did this by creating a post-processing sniffer (I will call it Lookup Sniffer), see Figure 2.13, that works together with an event-counting sniffer. In this case, the power consumption of the monitored module is directly calculated inside the sniffer itself. Since it is a function of the switching activity, the running frequency, and the technology used to build the component, a lookup table (that depends on the specified technology) is instantiated into each post-procesing sniffer at synthesis time. The running frequency and the switching activity are obtained in the associated event-counting sniffer and, from that information, every cycle, the post-processing sniffer will index the power table, and will add the resulting value to the accumulated power budget, stored into the log memory.

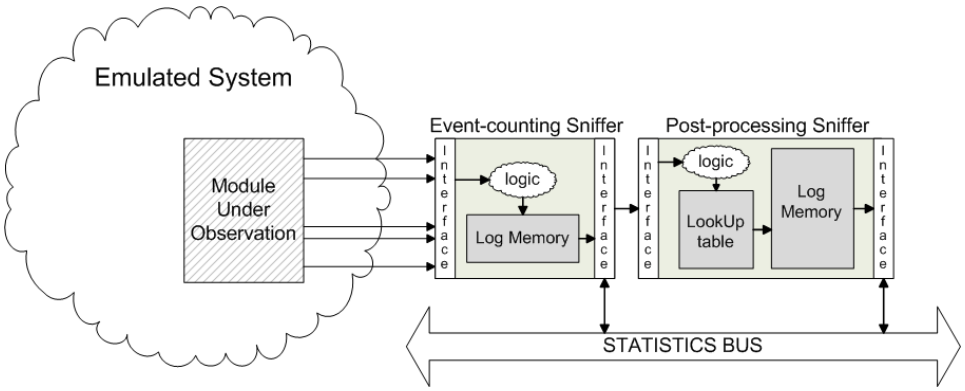


Figure 2.13: The Lookup Sniffer, an example of post-processing sniffer.

The main disadvantage of using this extra sniffer is the need for available BRAM to build the lookup tables. Thus, it is up to the designer choosing to do this task in SW (host PC) or HW (post-processing sniffer). If enough BRAM is available, it is recommended to instantiate the sniffer, since it will benefit the performance (as explained, the power numbers will be calculated inside the FPGA, relieving the host computer from this task). Nevertheless, only a small increase in performance was observed, since the FPGA is running at 100 MHz, but the computer runs at 3.0 GHz, compensating somehow the advantages of performing the operation in HW.

Another approach is that the sniffer just stores the number of cycles that the core was active, and at what frequency it was running. The final power budget is then calculated by multiplying the corresponding number of cycles by the power consumption constants. Multiplying by the emulated time we get energy values. While this alternative may save some memory space, it requires the instantiation of a HW multiplier, an expensive resource that cannot be included in every sniffer.

2.2.2.2. The Statistics Bus

In Figure 2.14 we can see, all the components of the Statistics Extraction Subsystem, whose main purpose is to extract statistics from the *Emulated System*. To this end, as I explained in Section 2.2.2.1, I created the *HW Sniffers*, that can monitor the inspected modules, and log down statistics. This information, however, needs to be sent to the host computer; otherwise, the limited buffers of the sniffers would saturate. Here, it comes into the picture the *Statistics Extractor*, a module in charge of periodically emptying the buffers, sending these data outside the FPGA. As depicted in Figure 2.14, the *Statistics Extractor* has access to the Communications Manager through

a dedicated link (arrow on its bottom), and to all the system sniffers thanks to the *Statistics Bus* (on its right).

The *Statistics Bus* connects together all the elements of the Statistics Extraction Subsystem, allowing the *Statistics Extractor* to access the system sniffers for data (statistics) retrieval; information that is then preprocessed by the Communications Manager, to form packets, and sent outside the FPGA to the external computer. The host computer processes this information and, in some cases, it generates new data to be sent back to the FPGA (e.g.: temperatures, commands). It follows the reverse path and, once inside the FPGA, it is delivered to its destination (sniffers or sensors) through the *Statistics Bus*.

The *Statistics Bus* is a 32-bit bus designed with a simple arbitration policy: the bus slaves are ordered by priority, with round robin between the elements with the same priority; In this way, there are no dynamic calculations that may require extra cycles. It was designed starting from the OPB specification [Xil10a], part of IBM's Coreconnect solutions. Borrowing ideas from the Wishbone, and the AMBA APB buses, the signaling mechanism was greatly simplified, ripping off the support for advanced error detection, split transactions (bus parking), and complex arbitration policies.

2.2.2.3. The Statistics Extractor

With all the sniffers connected to the *Statistics Bus*, a control module is in charge of accessing them, extracting the statistics, and forwarding them to the Communications Manager: the *Statistics Extractor*. It works as a DMA controller, moving information (both data and control commands) from the different elements attached to the bus (see Figure 2.14), to the Communications Manager, and viceversa.

Figure 2.14 depicts a special type of modules (a couple of them, on the bottom-right corner, painted in dark grey): the sensors. Although they are part of the *Emulated System*, as I explained in Section 2.1.2.1, being modeled components implies that we need to provide them the data traces that they present as "sensor reads". The *Statistics Bus* is the perfect medium for this task, since the data, that enter the FPGA through the Communications Manager, can be forwarded by the *Statistics Extractor* into the sensors, following the opposite path to that of the statistics; Thus, in the EP, the modeled sensors have an interface to the *Statistics Bus* and the *Statistics Extractor* has extended functionality to handle the data addressed to them.

2.2.3. The Communications Manager

This element enables a bidirectional link between the FPGA and the host computer. A schematic view of the connection can be observed in Figure 2.15. This link serves two purposes: First, it enables the data interchange between

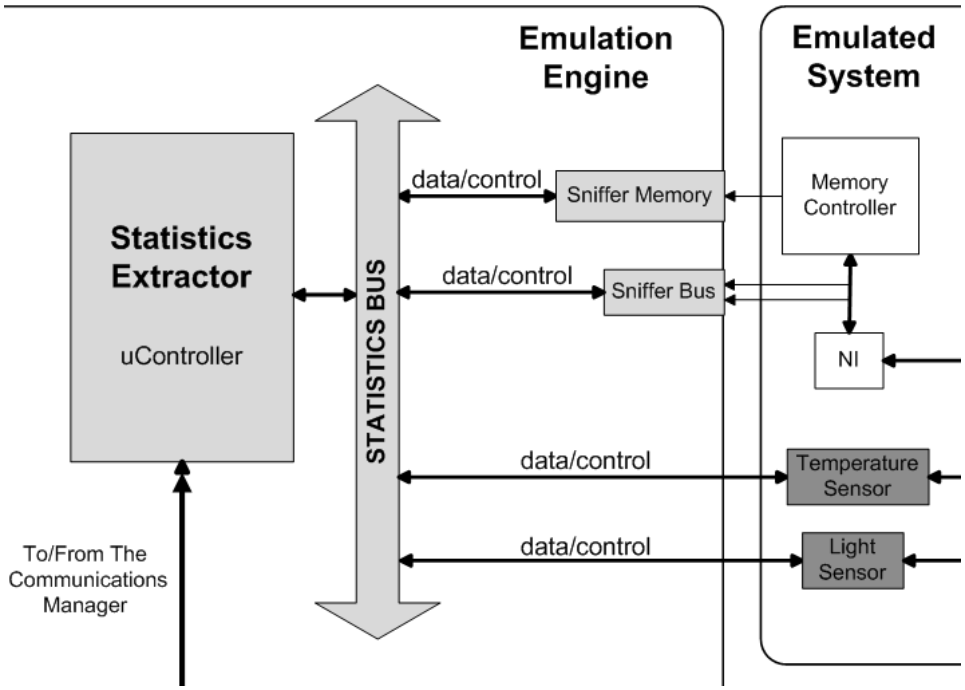


Figure 2.14: The complete Statistics Extraction Subsystem (with sensors).

the *Emulation Engine* and the *SW Libraries for Estimation*; second, it makes possible to control the EP from the computer. The only implementation requirement is the existence of a medium that physically communicates the FPGA and the computer. It can be a serial port, a JTAG connection, a PCI slot, an Ethernet connection, or a dedicated slot, to cite some examples; in fact, it can be a combination of connections. The bandwidth and lag of the communication will vary according to the type of connection used.

Regarding the different control actions that we can issue to the EP, I have divided them into four main categories:

1. Download a new *Emulated System* to the platform.

2. Control the evolution of the emulation: start, stop, pause, resume, reset and set the Emulation Step.

3. Manage the Statistics Extraction Subsystem: enable/disable the collection of data, initialize, reset, retrieve the statistics and feed data into the sensors.

4. Debug the processors (on-chip debugging): change the code, start, stop, trace execution and inspect internal registers.

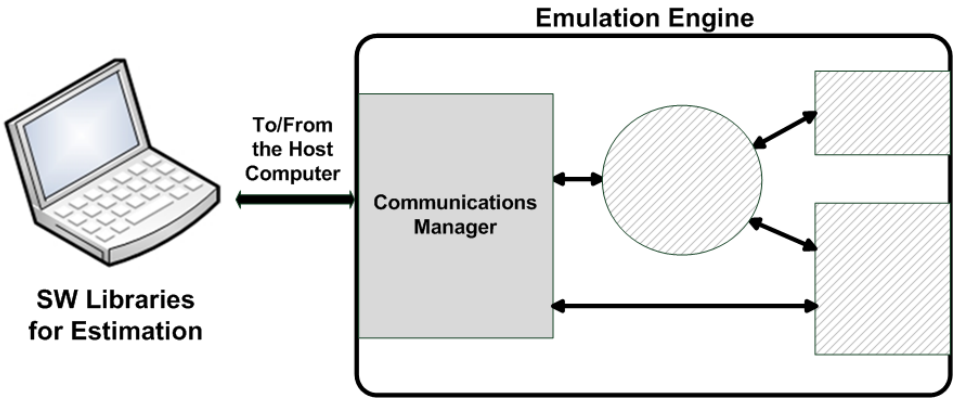


Figure 2.15: Bidirectional communication FPGA-computer.

My preferred method to implement the Communications Manager (for data and control) was a JTAG + Ethernet solution. To perform tasks 1 and 4, I used the JTAG connection, through the API provided by Xilinx. To this end, I developped some scripts to automate the processes and ease the user interaction. Tasks 2 and 3 were first implemented with a serial port. A simple and cheap solution, fast enough for this kind of flow control, since the volume of information interchanged is very little. Later on, since I added an Ethernet connection for the data interchange with the *SW Libraries for Estimation*, I decided to embed this control commands into the Ethernet frames to simplify the connections and reduce the number of cables. In order to deal with the particular characteristics of this packet-based communication system, I implemented a dedicated module called the *Network Dispatcher*, that is described in the next section.

2.2.3.1. The Network Dispatcher

As explained, my preferred implementation of the Communications Manager uses a standard Ethernet connection. The main advantage of this solution is that a crossed Ethernet cable (with rj45 connectors) is enough to connect both elements (FPGA and computer), being cheap and easy to interface with any standard PC host computer. The *Network Dispatcher* handles the low level details of the communication. Thus, from the point of view of a module that wants to exchange information, it only has to place it in an intermediate buffer, and signal the *Network Dispatcher*; similarly, the module will be signaled when new data arrive, so it can directly retrieve them from the reception buffer.

The implementation of the *Network Dispatcher* relies on the EthernetLite component, provided by Xilinx. Its operation is controlled from a Microblaze, an embedded microcontroller that also has access to a BRAM block

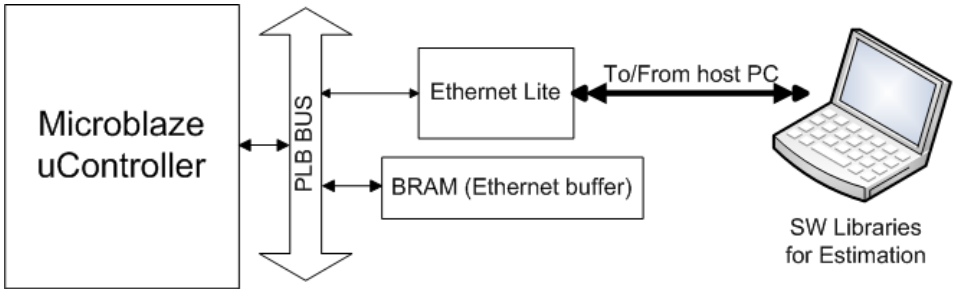


Figure 2.16: Structure of the Network Dispatcher.

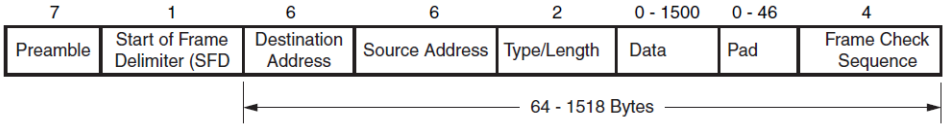


Figure 2.17: Format of an Ethernet data frame.

for buffering data. All the elements are interconnected through a PLB bus [Xil10b]. A schematic view can be observed in Figure 2.16.

Regarding the transit of packets, since the Ethernet standard limits the maximum size of a datagram, when data is sent from the FPGA to the host computer, they are transparently processed by my *Network Dispatcher* that automatically splits the data into packets. In the same way, when packets are received from the host PC, the *Network Dispatcher* transparently reassembles them in an intermediate buffer so that the control processor can deliver it to the final destination. Figure 2.17 shows the structure of an Ethernet frame, where *Destination Address* and *Source Address* is filled with the host computer and the FPGA MAC addresses, respectively, for the packets sent from the FPGA (outgoing packets), while the incoming packets swap these values. Observe that the data field may vary from 0 to 1,500 bytes in length.

The structure of the packets inside the data field of the Ethernet frame follows my own custom format. I call these packets *EP packets*, to differentiate them from the Ethernet frames. Figure 2.18 shows this encapsulation.

Inside an *EP packet*, the meaning of the data is implicit, and given by their position inside the packet. Figure 2.19 details the structure of the two types of *EP packets*, data and control: As shown, they do not have header; instead, the data to be transmitted directly start in the first place, followed by a termination tail, that contains flow control bits, error detection information, and the control commands. More in detail: A simple Ack-based control flow is embedded into the packets (*Sequence Number* field) that checks the sequence number of the last packet received. The *Control* field contains com-

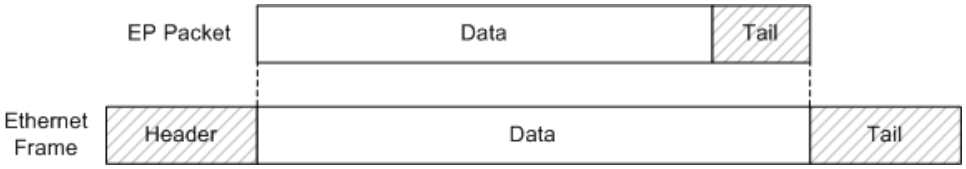


Figure 2.18: *EP packet* encapsulation.

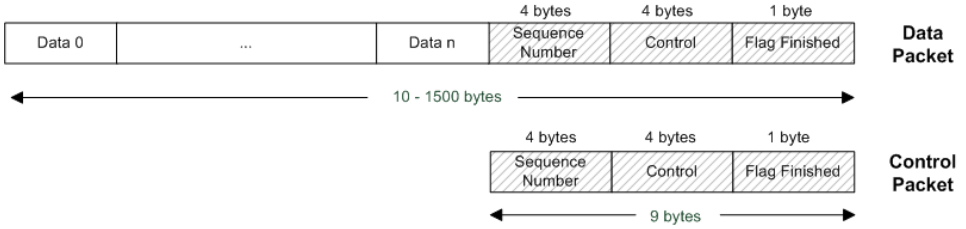


Figure 2.19: The two types of *EP packets*: data and control.

mands to be executed, or information of the state of the emulation. The last byte of the packets contains a flag (*Flag Finished* field) to indicate when a packet is the last of a fragmented datagram, and also to signal the end of the emulation.

As depicted in Figure 2.19, the lenght of the tail is always fixed (9 bytes) and present for the two possible types of *EP packets* (data and control). Note that a control packet is a particular case of data packet with no data fields. Control information (the tail) is always transmitted. Together with the data, all the information can not surpass the 1,500 bytes limit of the Ethernet container; bigger packets will be fragmented. Figure 2.20 shows two practical examples:

The first example, the outgoing packet, contains 1,491 bytes of statistics, the maximum that fits into one *EP packet*: 1,491 data bytes + 9 control bytes = 1,500 bytes. The second example represents the case for bigger packets. In particular, it is an incoming packet received in the FPGA containing temperatures. The size of the payload is bigger than 1,491 bytes; thus, it has been split into two *EP packets* (that will be, later on, encapsulated into two Ethernet frames).

Regarding the data packaging, as shown in both the statistics and temperature *EP packets* of Figure 2.20, the first data field is concatenated to the second one, the second one to the third one, and so on... This is possible since both the sender and the receiver of the packet, i.e., the *Emulation Engine* and the host computer, have the required information to decode it; that is, the packet structures are defined by the system designer before starting the emulation. They depend, of course, on the number of cells the floorplan has

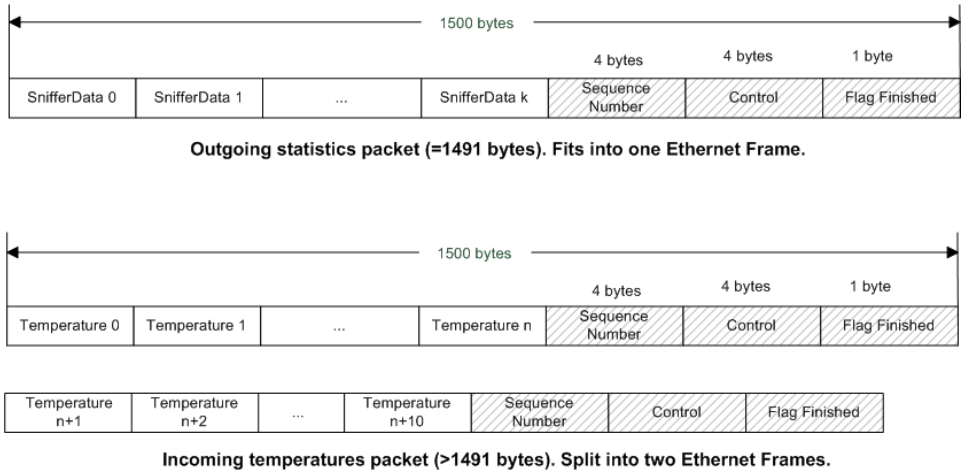


Figure 2.20: Two examples of *EP packet*: with and without fragmentation.

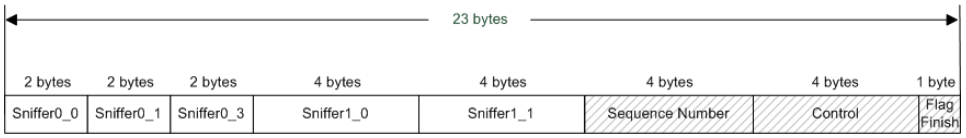


Figure 2.21: Example of *EP packet* containing the statistics from two sniffers: the first one with three 16-bit registers and the second one with two 32-bit registers.

been divided into (see Chapter 3), the number of sniffers instantiated, the type of the information extracted, etc. For a statistics packet, we can decide, for example, to send the statistic data from the processor X (Sniffer0) in the first place, followed by the statistic data from the memory Y (Sniffer1). If the first one contains three 16-bit registers, and the second one two 32-bit registers, the resulting packet would be like the one in Figure 2.21. Similarly, the structure of the incoming (temperatures) packet should be defined.

Additionally, I have implemented the possibility to wrap the *EP packets* into standard TCP/IP frames, with the whole IPv4 header. It allows for the broadcasting, inside a network, of the data collected. However, as explained, since I am using a point-to-point connection, I can directly send *Medium Access Control (MAC)* packets. This option is normally prefered, for it removes the extra overhead introduced by the IP layer. Figure 2.22 shows this new encapsulation scheme including the IP header, whose fields are detailed in Figure 2.23.

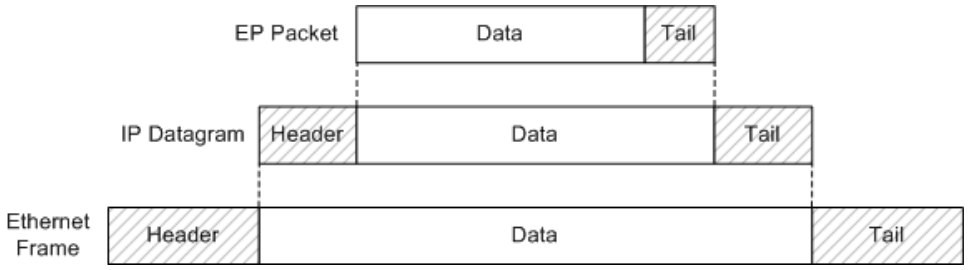


Figure 2.22: Frame encapsulation with the IP layer included.

4-bit	8-bit	16-bit	32-bit
Ver.	Header Length	Type of Service	Total Length
Identification		Flags	Offset
Time To Live	Protocol	Checksum	
Source Address			
Destination Address			
Options and Padding			

Figure 2.23: IP datagram header structure.

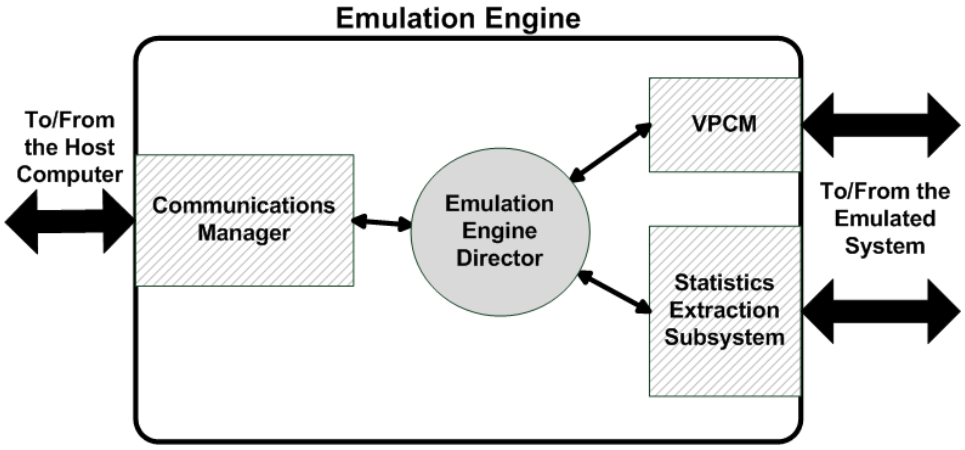


Figure 2.24: The Emulation Engine Director, coordinator of the Emulation Engine.

2.2.4. The Emulation Engine Director

Through the previous sections, I have explained the function of the Statistics Extraction Subsystem, the Communications Manager and the VPCM. In the normal operation flow, the VPCM clocks the *Emulated System* while, at the same time, statistics are being extracted by the Statistics Extraction Subsystem, and processed by the Communications Manager to generate packets and send them to the host computer. From the PC, I can also receive information that needs to be fed back into the *Emulated System*.

In this scenario, with multiple modules exchanging information, it is necessary the inclusion of a new element, that I have called the *Emulation Engine Director*, that links together the Statistics Extraction Subsystem, the Communications Manager and the VPCM. These three elements appear in a striped pattern in Figure 2.24, connected to the *Emulation Engine Director*, that sits in the middle of the picture.

At run-time, the *Emulation Engine Director* continously receives events, and must generate a response that requires to coordinate one or more of the *Emulation Engine* components. Table 2.1 shows the control commands that can be issued to the EP: Some of them allow us to control the general evolution of the emulation, like *start, stop, reset, resume* and *set emulation step*, while some others are specific to manage the Statistics Extraction Subsystem, like *enable/disable/reset* the collection of data. This last three commands can be issued either globally, or on a per-sniffer basis; i.e., we can enable, disable or reset the statistics of one specific sniffer, or all of them.

We classify the events that the *Emulation Engine Director* receives according to the source that originates them:

Table 2.1: Emulation control commands.

COMMAND	DESCRIPTION
Start	Starts the emulation
Stop	Pauses the emulation
Reset	Restarts the emulation from the beginning
Resume	Continues with the emulation
Set emulation step <n>	Specifies the number of cycles as <n>
Enable <i>	The sniffer <i> will log down all the statistics
Disable <i>	The sniffer <i> stops gathering statistics
Retrieve <i>	The statistics from the sniffer <i> are sent to the host computer
Initialize <i>	Resets the buffer of sniffer <i>
Feed sensor <i>	Puts the given data into a sensor

1. External events: At any point of the emulation, from the host computer, the EP user can issue any of the control commands in Table 2.1, with the purpose to experiment with the platform, debug it, or verify a specific part of the system.

2. Internal events: The members of the *Emulation Engine* signal events that require the intervention of the *Emulation Engine Director*, like the saturation of the FPGA-PC connection, or when a bottleneck appears in the Statistics Extraction Subsystem during the download/upload of data (e.g.: the extracted statistics or the estimated temperatures). The expiration of the Emulation Step, for example, is also considered an internal event, since no user interaction occurs.

Whenever an event arrives, the *Emulation Engine Director* must react accordingly. For example, it must stop the emulation in case of congestion of the Communications Manager; this operacion implies instructing the VPCM to stop the Virtual Clock of all or part of the components in the emulated MPSoC, and report the pause to the host computer, wich requires the Communications Manager. Table 2.2 describes all the actions and responses, along with the components involved.

At this point, the specification of the different components of the *Emulation Engine* (cf. Figure 2.3 for a high level view of the system) is complete. In the next section, I describe the implementation details.

2.2.5. The Complete Emulation Engine implementation

This section describes my particular implementation of the *Emulation Engine*, including some refinements aimed at optimizing the mapping into an FPGA. For example, conceptually, both the *Network Dispatcher* (Figure 2.16) and the Statistics Extraction Subsystem (Figure 2.14) are indepen-

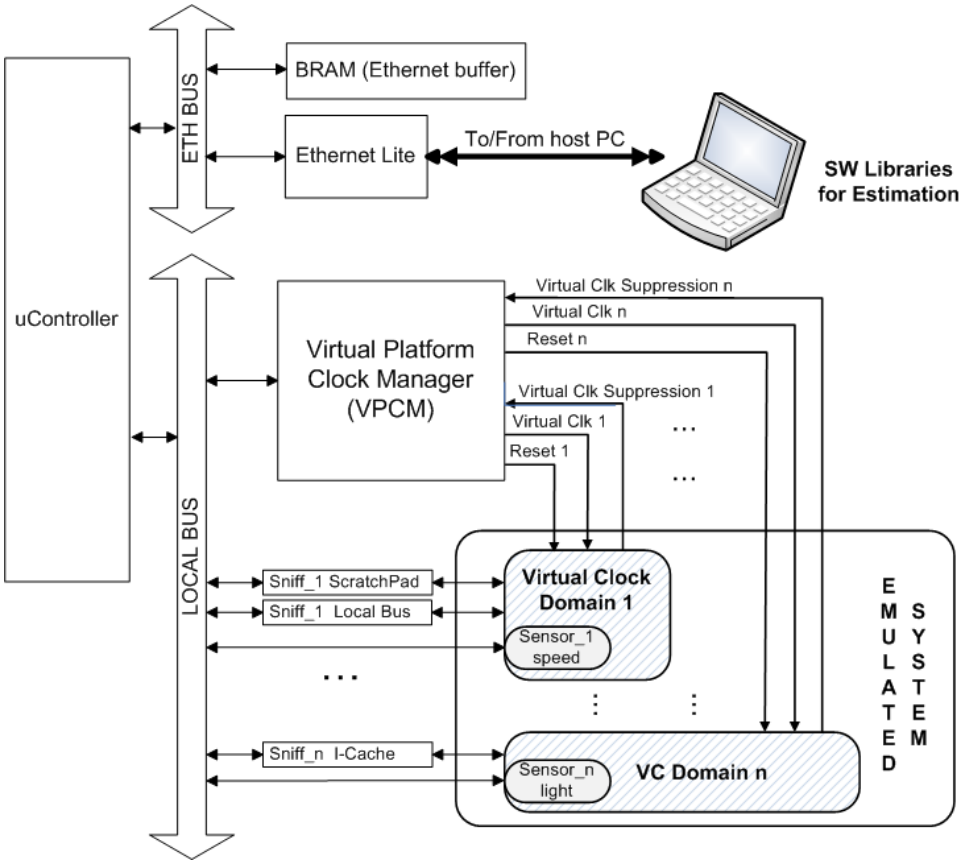


Figure 2.25: Implementation details of the Emulation Engine.

dent entities. However, from the implementation point of view, they both consist of a uController that coordinates the operation of some modules; therefore, we can merge both subsystems into one, saving one uController. This is possible because the utilization of the processor is very low: As stated in the previous sections, the uControllers used, typically perform synchronization tasks (exchange of simple commands/signals) or data-moving operations, that can be offloaded to a DMA controller.

After a thorough refinement of the system, the final implementation (Figure 2.25) contains only one uController, that directs the statistics extraction, controls the VPCM, and manages the communications with the host PC. I have, on the other hand, split the system bus into two different buses (shown in the figure as "LOCAL BUS" and "ETH BUS"), to separate the Ethernet traffic so that the packets can be processed concurrently, while I am, for example, retrieving statistics.

Algorithm 1 shows the pseudocode of the application that runs on the uController. The Main Program initializes the emulation parameters with the call to *startEmulation()* and, then, enters the main loop (*while not flag_emulation_completed do*) that performs the periodic extraction of statistics: it waits until the Emulation Step is completed, retrieves the statistics from all the sniffers with *collectStatistics()*, and sends them to the host computer with *sendEthernetPacket(stat)*. After that, the emulation iterates for another Emulation Step, and the same set of operations occur.

Asynchronous events like the reception of information from the computer (commands for the sniffers, data for the sensors, or general commands to control the emulation) are handled by the associated interrupt handlers. Internal events, like the saturation of the Ethernet buffer, also trigger an interrupt, so that the uController sends the appropriate commands (to the VPCM) to freeze and resume the emulation, or (to the Communications Manager) to report the situation to the host computer. Table 2.2 lists the possible events together with the action they trigger.

Algorithm 1: Main Program

```
startEmulation()
while not flag_emulation_completed do
  wait_until(emulation_step_completed)
  for i = 1 to NUM_SNIFFERS do
    stat = collectStatistics(i)
    sendEthernetPacket(stat)
  end for
end while
stopEmulation()
exit()
```

Table 2.2: Emulation events and corresponding actions.

EVENT	TARGET COMPONENT	ACTION
start_emulation	VPCM statistics	Generate the VC Activate sniffers log
stop_emulation	VPCM	Stop the VC generation
reset_emulation	VPCM VPCM	Stop the VC generation Reset the *Emulated System*
resume_emulation	VPCM	Resume the VC generation
set_emulation_step	VPCM	Set the number of cycles for the Emulation Step
enable_statistics	statistics	Activates the logging of statistics
disable_statistics	statistics	Stops the logging of statistics
retrieve_statistics	statistics communications	Extract statistics from target sniffers Send statistics to host computer
reset_statistics	statistics	Initialize sniffer buffers containing the statistics
feed_sensor	statistics	Put received data into the target sensors
emulation_step_expired	VPCM communications	Stop the VC Signal the computer
ethernet_buffer_full	VPCM communications	Stop the VC Signal the computer
ethernet_buffer_emptied	VPCM	Resume the VC generation
error_incorrect_data	communications	Signal the computer
error_communication_lost	communications	Signal the computer

For short, I use "VPCM" for the Virtual Platform Clock Manager, "statistics" for the Statistics Extraction Subsystem, and "communication" for the Communications Manager.

2.3. Conclusions

This chapter has been dedicated to describe in detail the HW part of the EP, namely, the part that is mapped onto the FPGA, composed by the *Emulated System* and the *Emulation Engine*.

Regarding the *Emulated System*, I have described the type of systems that can be instantiated, the different components that can be used (either fully specified or virtual ones), and I have emphasized the difference between HW prototyping and HW emulation. Next, I have explained in detail the internals of the *Emulation Engine*, with the different elements that form its architecture: the VPCM, the Statistics Extraction Subsystem, the Communications Manager, and the *Emulation Engine Director*; dedicating special attention to the *HW Sniffers*, the key component of the emulation. To conclude, I have shown an overview of the final system implementation.

In the next chapter, I describe the *SW Libraries for Estimation* that run on the host PC, and interact with the FPGA, calculating different values of interest.

Chapter 3

The SW Estimation Models

In the previous chapter, I described in detail the HW components of the platform; mapped into the FPGA, the *Emulated System* runs normally while, at the same time, the *Emulation Engine* controls the emulation, extracts statistics, and sends them to the host computer (see Figure 2.1). This chapter focuses on how this information is processed in the PC: From the simplest option, that consists on logging down all the information and present a report once the emulation is finished, to more advanced mechanisms like, for example, estimating the reliability of the system, and returning this information to the FPGA so that the *Emulated System* can elaborate a balancing policy to extend the life span of its components.

A set of configurable SW libraries, implemented in C++, runs on a general purpose computer and is in charge of the data manipulation. As input, they receive the run-time statistics from the *Emulated System*. As output, they calculate power, temperatures, reliability numbers, etc. of the final MPSoC. Through the following sections, I detail the process of how this input is converted into output using advanced mathematical models.

Regarding the way that these libraries interact with the FPGA, in the flow, the emulation runs for a predefined number of cycles (*Emulation Step*) and, then, the gathered statistics are retrieved from the FPGA buffers, and sent to the host computer. After that, the emulation is resumed for the next *Emulation Step*. The buffers, thus, must be dimensioned according to the size of the *Emulation Step*, since the *Emulation Engine* has to regularly empty them to avoid overflows. If we are, for example, logging down the number of read accesses to a memory, and we decide to use a 32-bit register to store them, a quick calculation tells us the maximum number of accesses (one per cycle, in a single-ported memory) per *Emulation Step* that we can store (2^{32}) and, therefore, the maximum size (in number of cycles) of the *Emulation Step*. Both values (the size of the buffers and the size of the *Emulation Step*) are user-configurable, and the designer is responsible for assigning correct values.

At this point, I should emphasize the fact that we are *emulating* a system (see Section 2.1.1); that is, the system whose behaviour we are evaluating (*Emulated System*) is mapped into the FPGA so that we can get statistics from it much faster than using a SW architectural simulator. However, the FPGA is not the target device. Therefore, the different values we estimate (power, temperature, reliability...) belong to the final implementation of the system: a silicon chip manufactured with a specific process technology, following the VLSI fabrication flow.

3.1. System statistics

The starting point for all the subsequent calculations made in this chapter are the *System Statistics*. I have given this name to all the information collected from the *Emulated System* at run-time, that is identical to that of the final chip, whose behaviour is being emulated. In the case of the SW simulators, it is clear: if we simulate the behaviour of an MPSoC architecture, the voltage of the *Simulated System* is not the voltage of the Pentium Core that is running the simulation. Similarly, when emulating the system in an FPGA, the voltage measured by the sniffer will be read from the voltage regulator present in the *Emulated System*, that is not the real voltage at wich the FPGA is operating, because the voltage regulators are modeled components, see Section 2.1.2.

The *System Statistics* comprise the current frequency and voltages of the system, as well as the *Activity Statistics*: an exhaustive log of all interesting events that occur in the platform, collected at run-time by the *HW Sniffers* that monitor the signals of the system cores every cycle (see Chapter 2, Section 2.2.2.1 for details). Examples of such *Activity Statistics* are provided in the next section (Section 3.2).

These *System Statistics* are extracted to a host computer through a communications port. The raw data that arrive from the FPGA uses a custom format that has the form of a series of numbers concatenated together, without separators. The meaning is implicit, and given by the position of the number inside the statistics packet. In this way, we minimize the amount of data interchanged. We may receive, for example, a string of numbers where the first one (the first 64 bits) represents the number of accesses to memory X, the second one the transactions in bus Y, and so on. All the numbers will come concatenated into one single string that contains all the detailed information. The data format is configured, before starting the emulation, at both ends of the communication (the FPGA and the *SW Libraries for Estimation*), and depends on the characteristics of the *Emulated System* (number of cores, number of sniffers, size and shape of the floorplan, etc.). In Section 2.2.3, I provided details and examples of the format of these packets.

By using the adecuate scripts, we can process the data that arrive to the

PC and generate accurate reports to track the different events that occurred during the emulation: cache misses, bus transactions, memory accesses, core states, resource-utilization reports, etc. This is the kind of information used as input by the *SW Libraries for Estimation*. In the following sections, I show how to use this information to calculate different system figures.

3.2. Power estimation

Chip manufacturers characterize the power consumption of the different elements of an MPSoC. Depending on the characteristics of the IP core, they may provide average power consumption, min/max values (based on the core activity), or detailed power states (sleep/active modes). These values depend on parameters, such as the implementation technology, the running frequency, the voltage, or the current temperature, so they normally come indicated in tables that we can index with the actual parameters. If we put this together with the fact that, in the Emulation Platform (EP), thanks to the sniffers, we can exhaustively log all the events that occur, from switching activity to high-level events (e.g. cache misses, bus transactions, memory accesses), generating power numbers from these data is pretty straight forward. Thus, I developed a C++ library that estimates the power burnt in the *Emulated System* performing the aforementioned calculations; it is called the *Power Estimation Model*.

Figure 3.1 describes the interface of the *Power Estimation Model*: As inputs, it receives the *System Statistics* (from either a predefined trace or from the FPGA), along with the temperature of each element under observation (coming from a predefined trace, or from the *Thermal Model* output; see Section 3.3); As output, the model calculates the power consumption of each system element. In order to do all the calculations, first, the user needs to configure the *Power Estimation Model*, providing some information about the *Emulated System*. I distinguish, then, two types of input parameters; Figure 3.1 depicts, on the left side of the *Power Estimation Model* (the square box), the parameters specified at compile time, whereas the ones specified at run-time come from the upper part. I use this format throughout the series of figures that describe the interfaces of the SW models.

The *Power Estimation Model* needs then the following information:

1. At compile time: The definition of all the components of the system; expressed as the power and leakage tables characterizing them (technology-dependent).

2. At run-time: The current temperatures of the different system elements, as well as the *System Statistics*: frequency and voltage of the elements of the system, plus the *Activity Statistics*, indicating the sta-

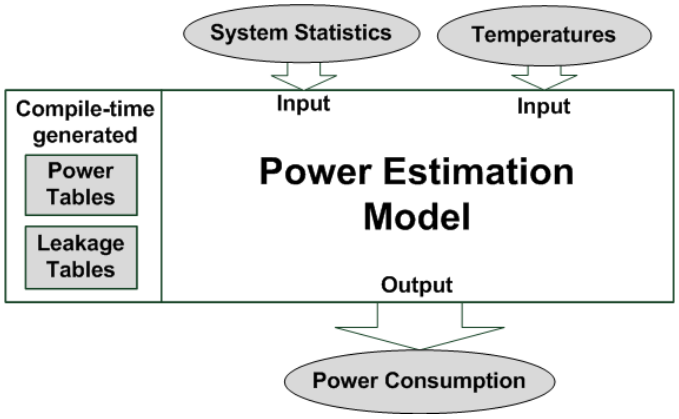


Figure 3.1: Interface of the *Power Estimation Model*.

tes of the cores (number of accesses to the resources, bus congestion, etc.).

I next present the details of the library and, for clarification, I illustrate with a design example the whole process to estimate the power of an *Emulated System*. For each aspect, first I describe the general case and, then, proceed to focus on the particular example.

First of all, I define what I have called the *components* of the system. I use this word to refer to the multiple pieces in which I divide the *Emulated System*. Each of the components is an independent entity defined by a set of properties: temperature, frequency, voltage, and *Activity Statistics*, as well as its power consumption. Figure 3.2, depicts the floorplan of a multi-processor system with four ARM11 processing cores and NoC-based interconnect. We differentiate twenty-nine components, grouped into five types of components: the ARM11 cores, the caches (data or instructions), the memories (private or shared), the network interfaces, and the switches. Note that the ARM11 core is quite a big element on itself, so we could have increased the resolution of the system, dividing each ARM11 core into smaller components, such as the integer register file, ALU unit, and so on. In such a situation, we could calculate the power consumption of the different parts of the ARM11 cores; as the counterpart, we would also need to provide more information (statistics), since the ARM11 components would be now divided into many different components.

The size of the components used during an emulation is fixed by the user and, sometimes, limited by the amount of information that the manufacturer provides about the current components used in the design. Something that helps to soften this inconvenience is the fact that the EP does not specify any fixed size for the components employed, nor does it set a global constrain on the size ratios among components; they can have any size. This is

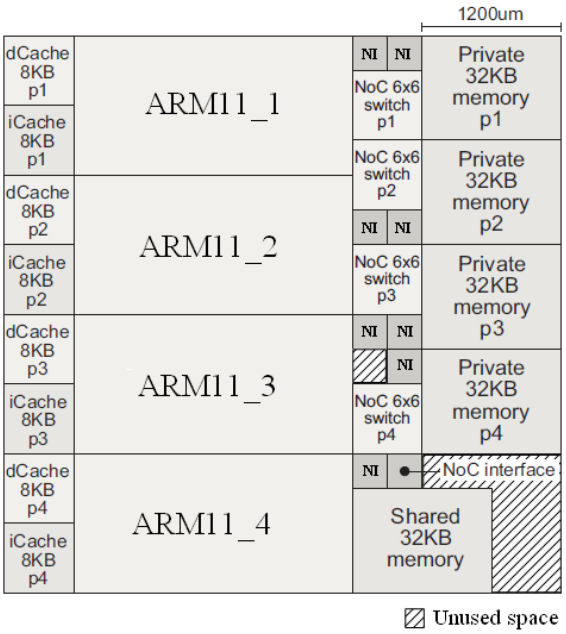


Figure 3.2: MPSoC floorplan with 4 ARM11 cores, several memories, and a NoC-based interconnect containing switches and NoC interfaces (NIs).

specially advantageous, for instance, in the case where we only want to study a part of the chip, for we do not need to model all the components with the highest level of accuracy. In order to keep the example simple, I stick to the components represented in the Figure 3.2.

Before starting the emulation, we must characterize all the different types of monitored components in the system, using the information obtained from the manufacturer (datasheet), third parties, or profiling tools. Following with the example of the ARM11-based system, Table 3.1 outlines the power consumption of the components present in the evaluated MPSoC; It indicates the maximum power numbers (peak power) for each component as worst case, but the effective power can normally be lower, depending on the workload (activities of processors and memories), and can be given as an input by the designer for his particular design. The values have been derived from industrial power models for a 0.13 μm technology, assuming the temperature remains stable around 333 Kelvin.

For the sake of simplicity, two assumptions have been made in this particular example: First, a stable temperature during the emulation process is assumed, which takes out one dimension of the table. The power consumption depends on the system temperature; thus, in the general case, when the thermal variance is bigger, the temperature is an extra parameter that affects the calculations and, therefore, the corresponding table would have one

Table 3.1: Power consumption of the components of the MPSoC example from Figure 3.2, implemented with a 0.13 μm bulk CMOS technology, and working at 333 Kelvin.

MPSoC Component	Max. Power (at 100 MHz)		Max. Power (at 500 MHz)	
	ON	OFF	ON	OFF
RISC 32-ARM11	300mW	80mW	1.5W	140mW
Cache 8kB (ARM11)	142mW	0	710mW	0
Memory 32kB	55mW	0	275mW	0
NoC switch (6x6-32b)	56mW	0	257mW	0
NoC network interface	23mW	0	128mW	0

more dimension, to account for different temperature conditions. Second, the voltage scales with the frequency, which removes another dimension of the table. Thus, although I do not show it explicitly, the voltage effects are taken into account, since the power reduction obtained with the frequency scaling is thanks to voltage scaling. When the voltage does not scale jointly with the frequency, another dimension is required in the table. Together, these two simplifications allow us to greatly reduce the complexity of the example: Instead of having a table with five dimensions (the type of component, and its state, temperature, frequency, and voltage), we reduced it to have only three (the type of component, its state, and its frequency-voltage).

Additionally, in the power calculations we must take into account leakage. The leakage current contributes with a percentage to the total power consumption of a system, according to the characteristics of the emulated circuit. It can be specified at run-time, for this parameter may vary depending on the type of component, its temperature, voltage, frequency, and/or activity, or may be assumed constant through the entire emulation. In any case, for generalization, I calculate it using a *Leakage Table* that, indexed with the type of component and its run-time parameters, returns the percentage of leakage to use in the current context.

In this particular example, I set the leakage to be 5 % of the total power consumption, for each component and working conditions. This figure actually corresponds to the indications of the *International Technology Roadmap for Semiconductors (ITRS)* [biba] for low-standby power systems in 0.13 μ with supply voltage of 1.2-1.3V. Therefore, the *Leakage Table* contains the same value (5 %) in all entries.

In order to apply the values from the *Power Table* (Table 3.1) and the *Leakage Table* to obtain the power consumption of our system, we abstract the elements of the *Emulated System* as state machines that, at a given cycle, are in a determined "power state". I next present some practical cases to help understand the procedure:

- **The core can be either active or idle:** In the simplest case, the core only consumes power in the cycles when it is ON. In a more general case, we are given two values: the power consumption when it is ON (active), and the power consumption when it is OFF (idling).
- **The core performs accesses/operations:** The memories, for instance, burn a different amount of power if accessed for reading or for writting. The buses also consume different if they perform a single operation, or a burst transaction. Therefore, we differentiate power states such as: initiating-transaction, performing-transaction, finishing-transaction... in the case of the buses, and tag-read, line-read, line-written, word-written... in the case of the memories.
- **The core executes instructions:** A processor power profile, for example, may depend on the type of instruction it is executing at a given cycle. The core could, thus, be in the state: executing-add-intruction, executing-nop-instruction, and so on.

The accuracy of the power estimations depends on how we model our system elements; For a given processor, for instance, we could use two models: one that sees the processor as a two-state machine that is either in the WORKING or IDLE state, or another one that sees up to thirty two different states, depending on the instruction being executed. Generally, the second one will offer more accuracy. In the same fashion, the smaller the elements (more granularity), the greater the accuracy: Intuitively, at certain instant, saying "my processor is ON" is less detailed than saying: "the register file of my processor is ON, while the ALU unit is OFF"; Note that, in both cases, the more accuracy we want, the more data we need to provide to the power model as inputs (either we gather more statistics for the same component, or we gather statistics for more components).

In our example of Figure 3.2, I decide, for instance, to monitor only the ARM11 cores and the local cache memories (instructions and data). The processors can run at 100 or 500 Mhz and be into two possible states: WORKING or IDLE. The memories always run at a fixed frequency (100 Mhz) and only consume power when being active.

At this point, the components to be monitored are fully specified; which means that we know the power states they can be in, and the power they consume in each of them. Putting this information together with the statistics we get from the EP at run-time (our *Emulated System* knows, thanks to the sniffers, the running frequency of each of the cores, and if they are active or not), we calculate the power consumed in the emulated circuit. In order to do these calculations, at compile time, several lookup tables must be generated, one per type of component, characterizing their power consumption. Such tables, are multidimensional arrays with four dimensions: state, temperature, frequency, and voltage; even though, in simple cases, one

Table 3.2: Power table for the ARM11 core.

	100 MHz	500 MHz
WORKING	300mW	1.5W
IDLE	80mW	140mW

Table 3.3: Power table for the cache memory.

ON	142mW
OFF	0mW

or more dimensions could be missing. In our example, for instance, before starting the emulation, we create two power tables: Table 3.2, associated to the processors, that can be indexed with the current frequency, and state of the core, and Table 3.3, indexed only with the memory state.

At run-time, the actual parameters index these tables, thus they are translated into a power number.

Inside the EP, the sniffer associated to the ARM11 core will be able to determine when it is active or idle and, will know, as well, its current frequency (fixed during each *Emulation Step*). This information is logged down and sent to the host computer, where it is fed into the *Power Estimation Model* that contains a simple power table with two values: the ARM11 power consumption per cycle at 100MHz, and at 500MHz. From that information, every cycle, the program indexes the power table, and adds the resulting value to the accumulated power burnt. In the case of the cache memories, the addition is only performed in the cycles they were ON.

Algorithm 2 represents, in pseudocode, the operation of estimating the power consumption for a component of the MPSoC in one *Emulation Step*. In this implementation of my model, the power consumption is calculated incrementally, in small *Emulation Steps*. During each *Emulation Step*, the temperature is assumed constant (as well as the frequency and voltage). This is specially useful when using together the power and thermal models, since the power consumed and the temperature of the system depend one on another. Performing the calculations in small steps, we can generate a discrete function that represents the evolution of the power consumption along time; i.e., time in the x axis, and power in the y axis.

The inputs of the algorithm specified at compile time (i.e., the Power and Leakage Tables) depend on the *Emulated System*. Thus, the notation *TABLE componentType1...Table[temperature, frequency, voltage, state]* denotes a static table with all the required information, that returns the Power/-Leakage of an element when indexed with the four parameters. As run-time inputs; i.e., the parameters of the function, in addition to a component reference, the algorithm receives two vectors, containing the gathered statistics (*systemstatistics*) and the temperatures (*systemtemperatures*) of the system,

Algorithm 2: estimateComponentPower(component, systemstatistics, systemtemperatures)

```
Constants:
  TABLE compType1PowerTable[temperature, frequency, voltage, state]
  TABLE compType1LeakageTable[temperature, frequency, voltage, state]
  TABLE compType2PowerTable[temperature, frequency, voltage, state]
  TABLE compType2LeakageTable[temperature, frequency, voltage, state]
  ...
  TABLE compTypeiPowerTable[temperature, frequency, voltage, state]
  TABLE compTypeiLeakageTable[temperature, frequency, voltage, state]
Program:
  temp ← systemtemperatures[component.id]
  freq ← systemstatistics[component.id].frequency
  volt ← systemstatistics[component.id].voltage
  activitystat ← systemstatistics[component.id].activity
  power ← 0
  switch component.type do
    case ComponentType1
      for each state in activitystat do
          partialpower ← componentType1PowerTable[temp, freq, volt, state] *
          activitystat[state].numcycles
          partialleakage ← compType1LeakageTable[temp, freq, volt, state]
          power ← power + partialpower * (1 + partialleakage)
      end for
    case ComponentType2
      for each state in activitystat do
          partialpower ← compType2PowerTable[temp, freq, volt, state] *
          activitystat[state].numcycles
          partialleakage ← compType2LeakageTable[temp, freq, volt, state]
          power ← power + partialpower * (1 + partialleakage)
      end for
    case ComponentType3
      ...
  endsw
  return power
```

for the current *Emulation Step*.

Regarding the algorithm itself, first, for clarification, we put the temperature, frequency, voltage, and *Activity Statistics* of the component into temporary variables. Then, we differentiate cases depending on the type of the evaluated component, in order to use its particular power and leakage tables.

The power consumption of the component is calculated as a linear combination of the contribution of each of the states it was through; i.e., time spent in each state, multiplied by the power consumption in such state. The calculation is simple (*switch* block inside Algorithm): for each state the component was in, the power table is indexed with the component's temperature, frequency, voltage, and its state, to get a value that is then multiplied by the time (in number of cycles) spent in that particular state. This partial sum will be corrected to account for the leakage (a percentage obtained using the corresponding *componentTypeiLeakageTable* table), and accumulated to yield the total power consumption of that component during the current *Emulation Step*.

In order to profile the complete power consumption of the whole *Emulated System* during an emulation, for each *Emulation Step* we must obtain the *System Statistics* and temperatures and call Algorithm 2 as many times as the number of components the system is divided in. Algorithm 3 shows the aspect of the main program, where we assume the existence of a function called *getEmulationData* that updates the variables that contain the *System Statistics* and temperatures to reflect those of the latest *Emulation Step*. Such values may come directly from the emulation, or from a pre-recorded trace.

Algorithm 3: obtainSystemPowerProfile()

```
for each Emulation Step do
    getEmulationData(systemstatistics, systemtemperatures)
    for each component in the Emulated System do
        estimateComponentPower(component, systemstatistics,
        systemtemperatures)
    end for
end for
```

Algorithm 4 is the particularization of Algorithm 2 for the ARM11 MPSoC, where we only model two types of elements: the ARM11 core and the cache memories. The statistics (activitystatistics) contain, for each processor, the number of cycles it was ON and IDLE, and the frequency of operation. The memories always run at a fixed frequency and only consume power when being active; thus, we just need to know the power per cycle consumption when they are ON. The resulting tables, *ARM11PowerTable* and *CacheMe-*

Algorithm 4: estimateComponentPowerExample(component, systemstatistics, systemtemperatures)

```
Constants:
  TABLE ARM11PowerTable[frequency, state]
  TABLE ARM11LeakageTable[]
  TABLE CacheMemoryPowerTable[state]
  TABLE CacheMemoryLeakageTable[]
Program:
  freq ← systemstatistics[component.id].frequency
  activitystat ← systemstatistics[component.id].activity
  power ← 0
  switch component.type do
    case ARM11
      for each state in activitystat do
          partialpower ← ARM11PowerTable[freq, state] *
          activitystat[state].numcycles
          partialleakage ← ARM11LeakageTable[]
          power ← power + partialpower * (1 + partialleakage)
      end for
    case CacheMemory
      for each state in activitystat do
          partialpower ← CacheMemoryPowerTable[state] *
          activitystat[state].numcycles
          partialleakage ← CacheMemoryLeakageTable[]
          power ← power + partialpower * (1 + partialleakage)
      end for
  endsw
  return power
```

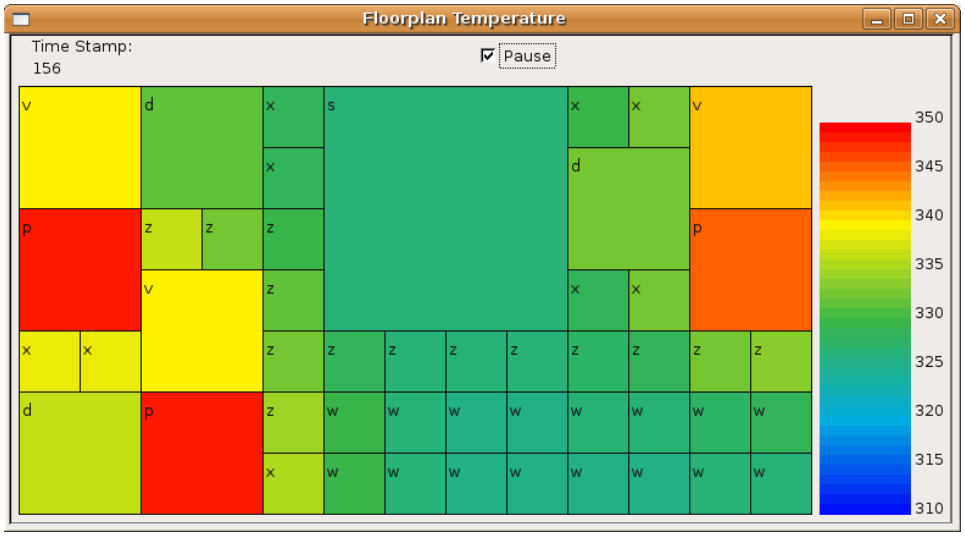


Figure 3.3: Thermal map generated with the thermal library.

moryPowerTable, are depicted in Tables 3.2 and 3.3, respectively.

As I mentioned in Section 2.2.2.1, since this particular estimation is really simple for the case when the temperature is stable, we can embed this functionality into a post-processing sniffer. The lookup table (PowerTable + LeakageTable) is then instantiated into the post-procesing sniffers at synthesis time (depending on the specified technology). As in the previous case, the event-counting sniffers that monitor the cores log down when their associated cores are active or not, and their current frequencies. However, the post-processing sniffers are now who receive this information, and index the power and leakage tables, storing power values into their internal log memory, so that the computer directly receives power numbers.

3.3. 2D thermal modeling

In the previous section, the procedure of translating statistics into power numbers was shown. Now, from the power numbers I will calculate temperatures. The idea is to characterize the thermal behaviour of the system so that, for any particular moment, we can provide a detailed thermal map, like the one depicted in Figure 3.3, where we can clearly appreciate the hotspots of the *Emulated System* under observation. Calculating temperatures is slightly more complicated than calculating power, for it depends on spatial characteristics; e.g., the location of a particular element in the floorplan.

In order to perform all the temperature calculations, the thermal library needs to know:

1. At compile time: The size and placement of all the components of the

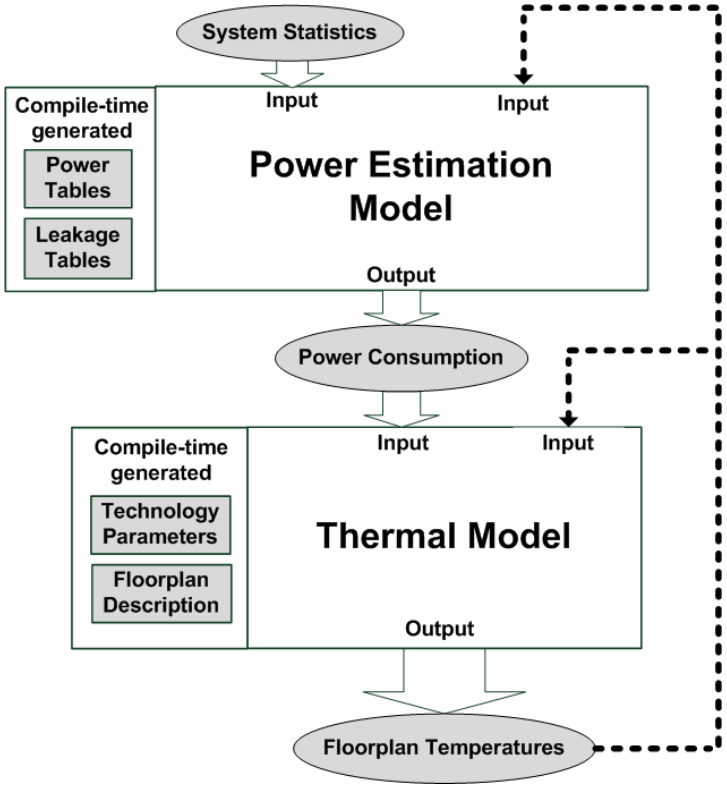


Figure 3.4: Interface of the *Thermal Model*.

system (i.e., floorplan layout), technology and packaging information.

2. At run-time: The power consumption of the system elements, that depends on the frequency, voltage, temperature, and activity.

Figure 3.4 shows the inputs and outputs of the *Thermal Model*. It estimates temperatures based on the power consumption but, at the same time, the power consumption depends on the current temperature (mainly due to the leakage current); observe the feedback loop depicted in the figure as a dashed arrow.

In order to accurately model thermal on-chip effects, a closed-loop system like the one described is mandatory, where both the power and thermal models work together, and depend one on another. For this reason, similarly to the power model, the temperatures are calculated in small *Emulation Steps*; i.e., the emulation time is discretized, so that a call to the *Thermal Model* returns the temperature at moment i. Since the temperature at moment i+1 depends on the temperature at moment i, the calculated temperature is fed back again as input, for the next iteration.

The main implication of this closed loop is that, opposed to the power model, where the *Emulation Step* is only constrained by the size of the buffers

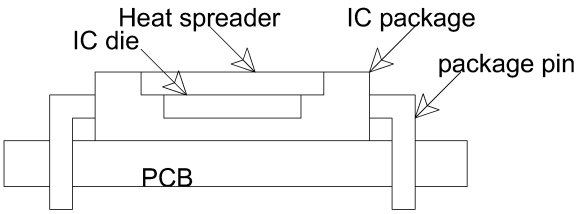


Figure 3.5: Chip packaging structure.

of the sniffers (we have to regularly empty them to avoid overflows), in the *Thermal Model*, if the *Emulation Step* is too big, the temperatures will not converge. Since the calculated temperatures are assumed constant during each *Emulation Step*, the *Emulation Step* size determines the accuracy of the estimations.

Regarging the library structure, like the rest of the EP, it has been implemented in a modular way, so that the different elements are independent and can be plugged/unplugged as required. This feature makes possible to use third-party thermal libraries (instead of this *Thermal Model*) to estimate the on-chip temperatures, as long as the interface (see Figure 3.4) remains unchanged. As an example, I performed a set of emulation experiments replacing my library with the well-known Hotspot v3.0 thermal library [SSS+04].

I dedicate the following sections to describe in detail the *Thermal Model*.

3.3.1. The SW thermal library

The second component of the *SW Libraries for Estimation* is the *Thermal Model*. It is a SW library written from scratch, to be able to evaluate the thermal behaviour in devices modeled at different levels of abstraction (i.e., gate level, RTL level and architectural level). It enables thermal exploration of silicon bulk chip systems.

I use the chip depicted in Figure 3.5 as an example through this section. It is composed by a silicon die wrapped into a package, and placed on a *Printed Circuit Board (PCB)*. On top of the IC die there is the heat spreader. The heat flow starts from the bottom surface of the die and goes up through the silicon, passes through the heat spreader and ends at the environment interface, where the heat is spread by natural convection with the ambient. There is no heat transfer from the IC package to the PCB, since it is considered an adiabatic material. New elements can be added to the model, like a heat sink, or removed, like the heat spreader, for instance, that does not exist in some mobile devices. The building materials are part of the configuration of the *Thermal Model*, so they can be easily changed; e.g., the IC package may vary, ranging from a low-cost to high-cost packaging solution.

The phenomena of heat conductance is modeled in physics using par-

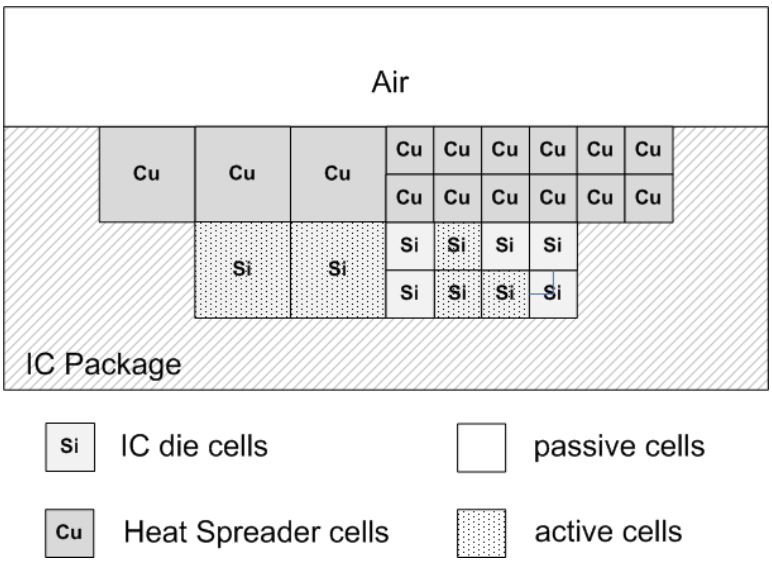


Figure 3.6: Simplified 2D view of a chip divided in regular cells of two sizes.

tial differential equations (PDE) [Daw10]. In particular, the heat diffusion inside a material is calculated based on the density, the specific heat, the thermal conductivity, and the heat transfer coefficient of the material, and it is governed by a PDE that depends on the instantaneous power density of the heat sources and the temperature T of each of the particles, specified by their location in the 3D space. Although the resulting equation describes very accurately the temperature of every point of the chip at a given time, it is too expensive, in terms of computation, to be used in the EP that aims at calculating its temperature evolution in real-time; for this reason, I use a simpler, equivalent model, to analyze the heat flow, instead. Similar to [SSS+04; SLD+03; HBA03], I exploit the well-known analogy between electrical circuits and thermal models, by which, the way heat propagates through materials is similar to the way current propagates through an RC electric circuit. Thus, with electrical currents playing the role of heat, I decompose the silicon die and heat spreader into elementary cells (or cubes) which have a cubic shape, and use an equivalent RC model for computing the temperature of each cell, and calculate how it propagates to the sorrounding neighbouring cells.

These *cells*, in which the system is divided, are different from the system components, elements, or components previously mentioned in the Power Model. For this reason, I first explain the *Thermal Model* without mentioning the system components, that are at a higher level (functional, instead of physical).

Figure 3.6 shows a 2D view of an IC die made of silicon (dark grey) attached to a heat spreader made of copper (light grey); no interface materials

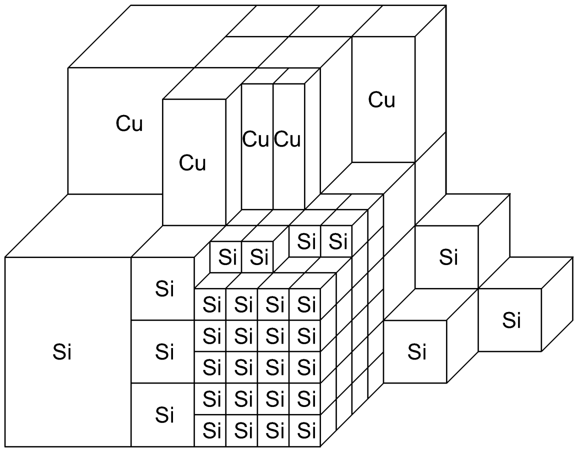


Figure 3.7: 3D view of a chip divided in regular cells of different sizes.

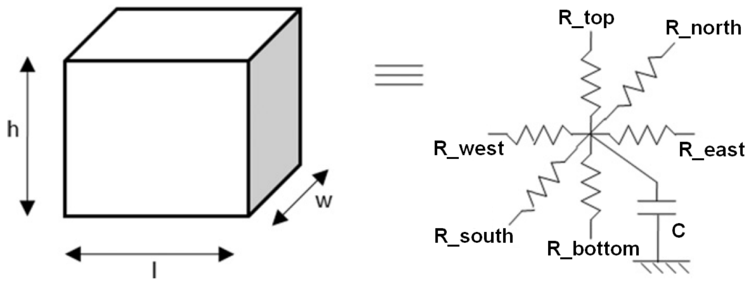


Figure 3.8: Equivalent RC circuit for a passive cell.

are modeled between die and spreader. The system contains cells of two different sizes: the small ones (8 for the IC die, and 12 for the spreader), that we take as the reference, of 1×1 units, and the big ones (2 for the IC die, and 3 for the heat spreader), of 2×2. In real life, a chip has three dimensions, what means that the small cells, for instance, should measure 1×1×1 units; similarly, the other cells would extend through this third dimension. Figure 3.7 illustrates this, showing a 3D view of a chip divided into many different sized cells.

In order to create an equivalent RC thermal model, I associate with each cell a thermal capacitance and six thermal resistances (see Figure 3.8). The capacitance (C) represents the heat storage inside the cell, four resistances are used for modeling the horizontal thermal spreading (R_{north}, R_{south}, R_{east} and R_{west}), and the other two (R_{top} and R_{bottom}) are used for the vertical heat diffusion.

The generation of heat is due to the activity of the functional units inside the chip; this is the point where the power and thermal models meet: In my thermal model, some cells of the IC die (those with a dotted pat-

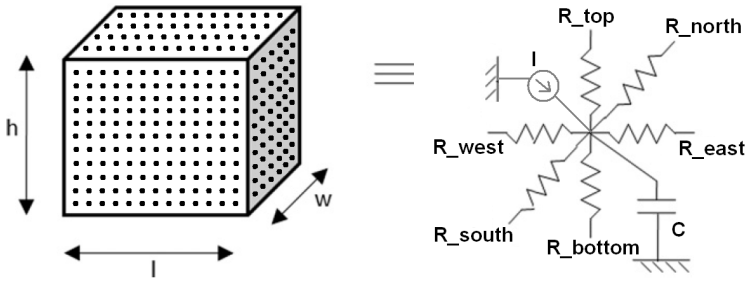


Figure 3.9: Equivalent RC circuit for an active cell.

tern in the example of Figure 3.6) are considered to contain funtional units (components). The power density of these funtional units is calculated by the *Power Model*, and input to the *Thermal Model* by adding an equivalent current source to these *active cells*, as opposed to the *passive cells*, that only spread the heat. Figures 3.8 and 3.9 depict the equivalent electrical circuits used, for the passive and *active cells*, respectively. There are no restrictions on the location of the *active cells*; it is the user, during the floorplan design stage, who decides the role of each cell. For the type of chips modeled in this thesis, the heat is always generated in the lower layers, those ones corresponding to the IC die cells. They are mostly made of silicon (containing the logic), and some metal (for the interconnection). If part of the silicon is unused, the cells in that region will be *passive cells*.

Inside a thermal cell, the conductance of each resistor (g) and the capacitance (c) are calculated as follows:

$$g_{top/bottom} = k_{th} \cdot \frac{l \cdot w}{(h)} \tag{3.1}$$

$$g_{north/south} = k_{th} \cdot \frac{l \cdot h}{(h)} \tag{3.2}$$

$$g_{east/west} = k_{th} \cdot \frac{w \cdot h}{(h)} \tag{3.3}$$

$$c = sc_{th} \cdot (l \cdot w \cdot h) \tag{3.4}$$

where w, h and l are the width, height and length that indicate the dimensions of the cell. The subscripts top, east, south, etc., indicate the direction of conduction, and k_{th} and sc_{th} are the thermal conductivity and the specific heat capacity per volume unit of the material, respectively. These equations are directly inserted into the code of the *Thermal Model*. As an example, Algorithm 5 illustrates the calculation of the capacitance of the cells: The parameters that define a cell are selfcontained into a variable called

cell, of type record. The type of cell (the material), for instance, is stored into the field *type*; thus, we use *cell.type* to index the array *capPerUnit*, that contains the thermal capacitance per unit volume of all the materials present in the system (cf. comments shown in Algorithm 5), and multipy the obtained value by the dimensions of the cell.

Algorithm 5: calculateCellsCapacitance()

```
for all the cells do
    cell.cap ← capPerUnit[cell.type] * cell.l * cell.h * cell.w
    {populates cell.cap with the thermal capacitance of the cell.}
    {l, h and w are the length, height and width of the cell.}
end for
```

For the *active cells*, the "heat" injected by the current source corresponds to the power density of the architectural component covering the cell (e.g., a memory decoder, a processor, etc.) multiplied by the surface area of the cell. This calculation yields Watts, the same units output from my power model. However, unless we are in the cases where the size of the thermal cells corresponds 1 to 1 to the size of the architectural components, we must convert the outputs from the power model from power to power density (using the size of the components), and back to power (using the size of the cells).

Figure 3.10 represents the complete RC circuit that models the chip shown in Figure 3.6. Since this is a 2D simplification, each cell shows only four resistors (top, bottom, west and east); the complete version would be similar to Figure 3.7, were each cell would contain also the north and south resistances to model the heat propagation along the third dimension, exactly like the cells shown in Figures 3.8 and 3.9.

Then, in Figure 3.10, I model the removal through air convection of the heat from the cells on the top surface by connecting an extra resistance (dotted, in the figure) in series with all the resistances R_{TOP} of the cells of the top layer of the heat spreader. Regarding the heat difussion from the cells to the package materials, initially, the first implementation of the *Thermal Model* assumed a simplistic behaviour of the package: It was considered "an entity that helped reducing the power density of the *active cells*"; thus, it was modeled substracting a fixed amount of Watts from the border cells of the IC in contact with it. Currently, the package is modeled as a material, characterized with its own thermal conductivity and capacitance; thus, heat diffussion occurs from the IC to the package, both laterally (outwards), and vertically (downwards). I model this by increasing the value of the border resistances to account for the difference of conductance from the IC to the package materials. Figure 3.10 shows the weighted resistances in bold.

The temperature of a cell depends on two factors: First, the power burnt

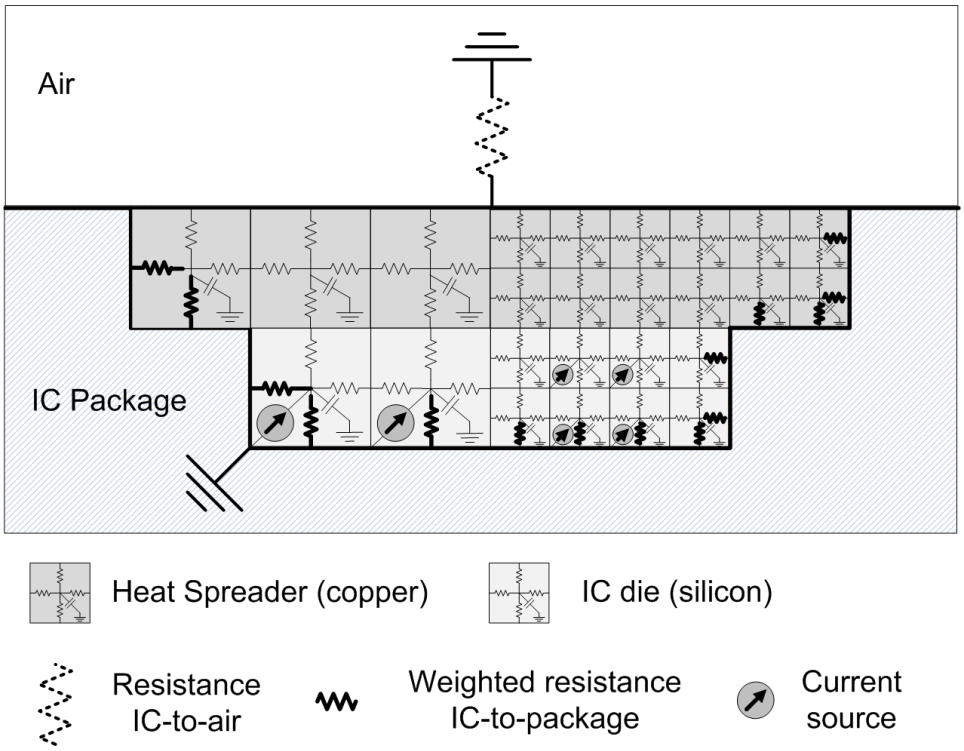


Figure 3.10: Simplified 2D view of the equivalent RC circuit for the whole chip.

Table 3.4: Thermal properties of materials.

Silicon thermal conductivity	295-0.491 TW/mK
Silicon specific heat	$1.659 \times 10^6 J/m^3K$
SiO2 thermal conductivity	$1.38\ W/mK$
SiO2 specific heat	$4.180 \times 10^6 J/m^3K$
Copper electrical resistivity	1.68×10^{-8} (1+0.0039$\triangle$T)Ωm, $\triangle$T = T-293.15K

inside it, determined by its activity (thus, null in the case of the *passive cells*) and, second, the heat diffusion that occurs towards/from the sorrounding cells (the neighbours). Section 3.2 explained how to calculate the power burnt inside a cell. In order to calculate the heat diffusion, we must analyze the behaviour of the resulting RC circuit shown in Figure 3.10; it can be described, using a set of first-order differential equations via nodal analysis [VS83], as follows:

$$G \cdot X(t) + C \cdot \dot{X}(t) = B \cdot U(t), \tag{3.5}$$

where X(t) is the vector of cell temperatures of the circuit at time t, G and C are the conductance and capacitance matrices of the circuit, U(t) is the vector of input heat (current) sources, and B is a selection matrix. G and C present a sparse block-tridiagonal and diagonal structure, respectively, due to the characteristics and definition of the thermal problem (see [ASC10] for details).

In Equation 3.5, G and U(t) are functions of the cell temperatures, X(t), making the behaviour of the circuit non-linear; this is because of the temperature-dependent thermal conductivity of the silicon and the temperature-dependent electrical resistance of the copper (used in the interconnections) [HLZW05]. In this work, a first-order dependence of these parameters on temperatures around 300 K is assumed. Some of these parameters are presented in Table 3.4.

The temperatures in the *Thermal Model* are updated in small *Emulation Steps*, which corresponds to calculating the steady state (its properties are unchanging in time) response of the circuit described by Equation 3.5, where the input currents are DC sources. Equation 3.6 shows this particular case:

$$G \cdot X = B \cdot U \tag{3.6}$$

The above set of equations is normally solved by inversion of the matrix G, using the sparse LU decomposition method [DD97]. However, in this case, the resulting equations, Eq. 3.7, are non-linear; thus, I use the Forward Euler 1st order method instead, that works directly with Equation 3.6. I refer to publication [ASC10] for the low level details of the algorithm. Basically, it makes a guess on the initial value of the matrix X, solves the equations (i.e.,

Algorithm 6: calculateSteadyStateTemperatures()

Define:
$X^r \leftarrow$ vector of cell temperatures during the *rth* iteration,
$G^r \leftarrow$ conductance matrix during the *rth* iteration,
$U^r \leftarrow$ input vector during the *rth* iteration.
Program:
$r \leftarrow 0$
// Generate an initial-guess for X^0:
$X^0 \leftarrow$ initialguess
Calculate G^0 and U^0 using the generated guess X^0
loop
$X^{r+1} \leftarrow (G^r)^{-1} \cdot B \cdot U^r$
if $\left|\left|X^{r+1} - X^r\right|\right| >$ maxErrorAllowed **then**
exit with error
else
$r \leftarrow r + 1$
Calculate G^{r+1} and U^{r+1} using the updated temperatures X^{r+1}
end if
end loop

propagates the heat) for the next "time instant", and calculates the error. If it is less than a predetermined error criterion, it means that the temperatures converged, and the process can be iterated.

$$X = G^{-1} \cdot B \cdot U \tag{3.7}$$

The detailed description of this iterative algorithm is presented in Algorithm 6: At the beginning, the model estimates the initial temperature conditions, X^r (that equals X^0, when $r = 0$), determining the values of matrices G and U. Then, it enters the main loop where, in each iteration, it calculates a small temperature evolution (X^{r+1}), and updates the values of matrices G and U. If the new temperatures (X^{r+1}) are not close enough to the old ones (X^r), i.e., the temperatures do not converge, the algorithm terminates and must be executed again with a different initial guess. In most of the test cases, 5-6 iterations were found to be sufficient to reach convergence within an error of 10^{-6}.

The thermal library can be configured in multiple ways to evaluate the thermal behaviour of different alternatives for each final MPSoC chip. For instance, its space resolution for thermal accuracy is configurable (i.e., number of temperature cells in a fixed area) as well as many other packaging parameters (e.g., quality of heat sink, thermal capacitance of the different materials that compose the chip, etc.). Together, they all define the final values of the resistances and capacitances inside the cells.

By varying the cell size and number of cells, we can trade-off simulation speed of the thermal library with its accuracy; the coarser the cells become, the less cells we need to simulate, but the less accurate the temperature estimates become. Figure 3.3 shows a floorplan thermal map generated with the thermal library, where we can appreciate some of the implications of what I have called "the cell-resolution of the model": Observe the division of the floorplan into a set of cells, and that the temperature is constant within a cell.

The emulation process is divided into *Emulation Steps* or *slots*, as a way to discretize the time; for each *Emulation Step*, we retrieve the *System Statistics*, calculate the power consumed, and the increment in the temperatures. Therefore, in order to analyze the heat diffusion phenomena, that is continous in time and space, the EP works with discrete time (it has been divided into *Emulation Steps*) and discrete space (it has been divided into cubic cells).

Algorithm 7 describes the structure of the *Thermal Model*, as it is implemented in the EP:

First, the different parameters of the emulation are initialized: The system floorplan is loaded (*Load Floorplan*), including the dimensions, characteristics (type) and placement (neighbours) of the different cells. With this information, additional properties of each cell are computed: its resistances ($cell.r_{north}$, $cell.r_{south}$, ...) and capacitance (cell.cap), derived from the technology used and the physical dimensions of the cell by directly applying the ecuations 3.1, 3.2, 3.3 and 3.4 (*calculateCellsResistances()* and *calculateCellsCapacitance()*).

Next, the initial temperatures (cell.temp) are loaded from a file, so we can start the emulation from neutral conditions (the system is switched off, at ambient temperature), from a recreated thermal-stressing situation, or even from the final conditions of a previous emulation.

The emulation time is initialized to zero, and the flag *emulationfinished* is set to false. To conclude the initialization, the program signals the FPGA to start the emulation (*Initialize Emulation*). At this point, the preparation is finished, and the emulation begins.

Once inside the main loop, the function *runEmulationStep()* lets the emulation run for the number of cycles specified as "Emulation Step" and, then, with a call to *retrieveStatistics()*, the *System Statistics* are retrieved, along with the flag that indicates if the end of the emulation arrived (*emulationfinished*). Next, the power consumed is estimated (*updatePower()*) in each cell during the current emulation slot (refer to Algorithm 2 from Section 3.2). With that information, the new temperatures (*updateTemperatures()*) are calculated, and the emulation can resume with the next *Emulation Step*.

Algorithm 8 details the process of calculating the temperatures of the cells after a given emulation step: First, we discriminate if it is an active

Algorithm 7: Thermal Model

```
Load Floorplan
calculateCellsResistances()
calculateCellsCapacitance()
loadInitialTemp()
time ← 0
emulationfinished ← false
Initialize Emulation
while NOT emulationfinished do
  runEmulationStep()
  retrieveStatistics(systemstatistics, emulationfinished)
  updatePower() {starts the power model.}
  updateTemperatures() {starts the thermal model.}
end while
```

Algorithm 8: updateTemperatures()

```
for all the cells do
  if cell.isActive then
    cell.partialtemp ← cell.temp + cell.cap * cell.power
  else
    cell.partialtemp ← cell.temp
  end if
end for
calculateSteadyStateTemperatures()
for all the cells do
  cell.temp ← cell.newtemp
end for
```

cell. Due to its own contribution, the cell temperature can only remain the same, or increase; it is thanks to the contribution of the neighbours that the temperature can decrease (some heat may be transfered to them). The next step of the algorithm is to correct the partial temperatures calculated by adding the effect of the heat diffusion among neighbours. It is represented as the function call: *calculateSteadyStateTemperatures()* and, internally, consists on solving the system of equations presented in equation 3.7 (applying algorithm 6).

The updated temperatures are stored in a temporary field (newtemp) until the calculations are completed for all the cells. At that point, we commit the changes (temp=newtemp), and continue to the next *Emulation Step*, where these calculated values will be the new input of the power and thermal models. Once the emulation finishes, we have a detailed log of the evolution of the system temperature along time, that could be used, for example, to

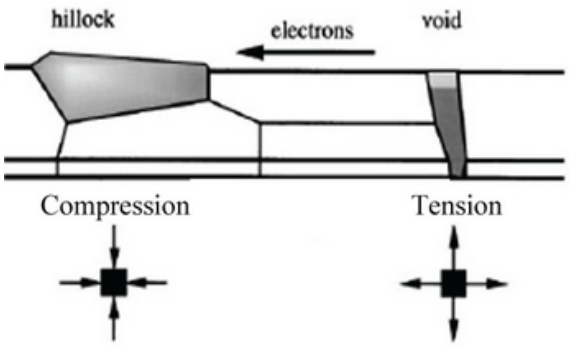


Figure 3.11: **Electromigration:** when atomic flux into a region is greater than the flux leaving it, the matter accumulates in the form of a hillock or a whisker. If the flux leaving the region is greater than the flux entering, the depletion of matter ultimately leads to a void.

estimate the reliability of the system (see Section 3.4).

3.4. Reliability modeling

In the previous section, I detailed the development of the thermal estimation library that allows us to explore on-chip temperatures. In this section, I explain how, with some additions, I enhanced the framework with the ability to perform reliability analysis of MPSoCs. In this case, however, I borrowed an existing model for calculating reliability figures; thus, my job was to port the code and do some minor modifications to adapt it to the platform. For this reason, I will not give the low level details of the implementation, since the model itself can not be considered as my contribution. Instead, I briefly mention the fundamentals.

Chip manufacturers provide the estimated MTTF of their chips. This estimation is calculated statically, without taking into account any chip activity. However, the dynamic behaviour experienced by highly stressed chips may eventually modify the estimated MTTF, and must be taken into account. The analysis of the influence of the temperature changes on the reliability of CMOS systems is investigated through the use of several mathematical models that include this dependency [SABR05]. The effects included in my experimental work have been selected by their strong impact on the Mean Time To Failure (MTTF), namely, Electromigration (EM), Time-Dependent Dielectric Breakdown (TDDB), Stress Migration (SM), and Thermal Cycling (TC).

- **EM:** Appears due to the momenta exchange between the electrons and the aluminium ions in long metal lines. The induced mechanical stress

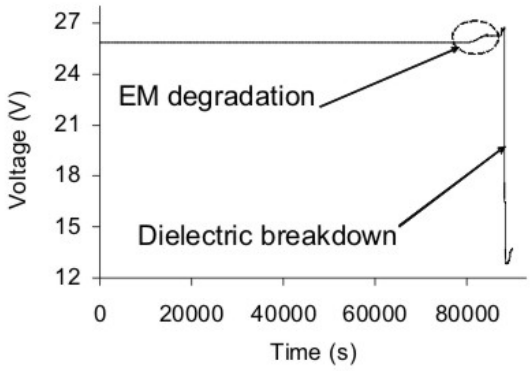


Figure 3.12: **Dielectric breakdown:** A 1.5mm long parallel Cu line structure stressed at 3mA and 200°C: the phenomenon of EM starts to appear followed by a sudden dielectric breakdown.

may eventually cause fractures and shorts (see Figure 3.11).

- **TDDB:** The influence of electric fields over the gate oxide film originates a conductive path in the dielectric, shorting the anode and cathode (see Figure 3.12).

- **SM:** Materials differ in their thermal expansion rate; this difference, under conditions of mechanical stress, leads to the migration of metal atoms from the interconnect. The resistance rise associated with the void formation may cause electrical failures.

- **TC** Each time a device undergoes a normal power-up and power-down cycle, a permanent damage is accumulated that will eventually lead to a circuit failure.

Figure 3.13 shows the interface of the *Reliability Model*, connected to the thermal and power models. As external input, it only receives the system temperatures. The dashed arrow on the right represents a feedback loop for the reliability; this is to reflect the fact that, at any point of the emulation, the reliability of the system depends on the past history (i.e., the aging effects are accumulative). Initially, we load the nominal value of the MTTF (expressed as the 100 % of the MTTF provided by the manufacturer) and, in each iteration of the model, we substract the contribution (a percentage) from the EM, TDDB, SM and TC effects. These calculations are made for each thermal cell. Eventually, the cell with the worst MTTF will determine the reliability of the whole system.

The input parameters of the *Reliability Model* can also be classified according to the moment when they are required:

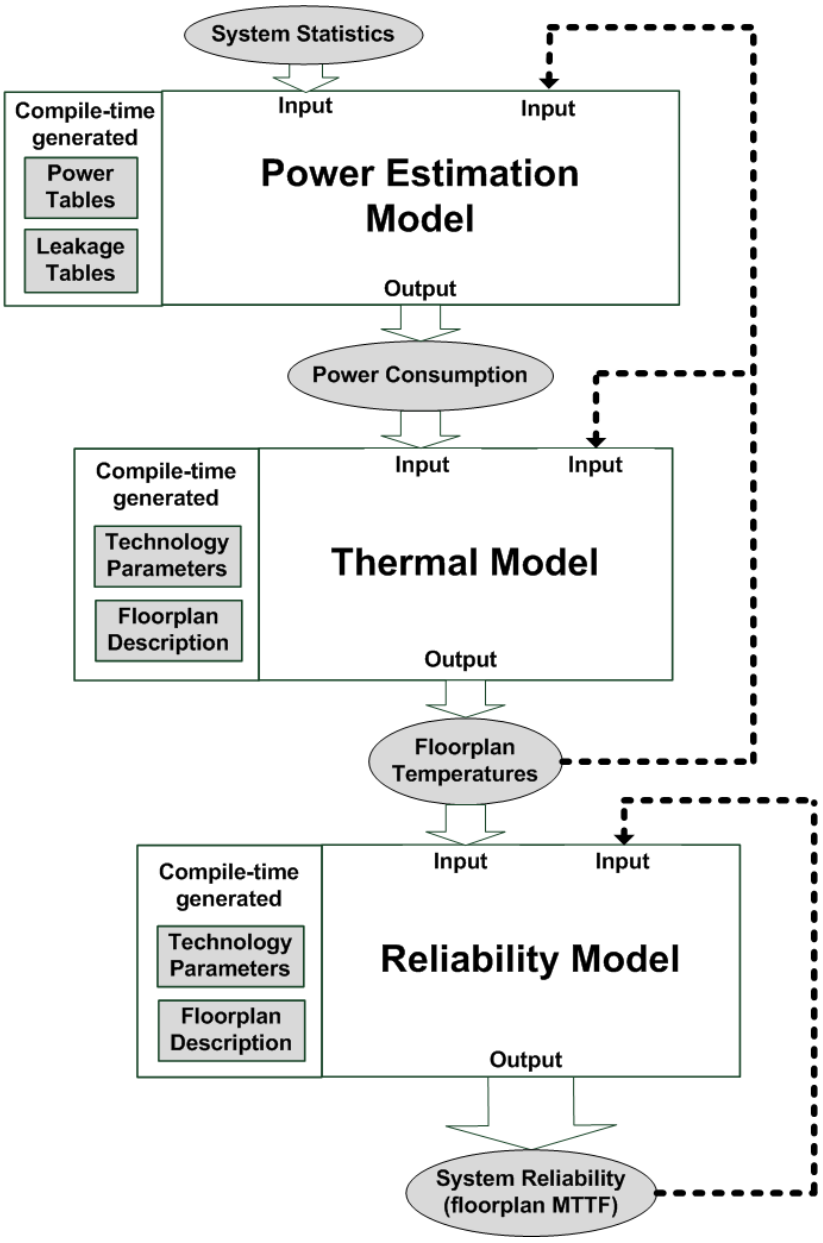


Figure 3.13: Interface of the *Reliability Model*.

1. At compile time: The floorplan description, indicating the components that are present in the system, and the technology parameters used for the implementation.

2. At run-time: The temperatures of the system elements.

Reliability wearout of CMOS chips occurs after years of utilization. If a chip is to fail in 20 years, for instance, ideally, we would need to simulate, at least, 20 years of the chip behaviour. While this holds true when we want to calculate strict MTTF figures, we can simplify the calculations if we only need an estimation of the worst case, which is tipically the case for CMOS chip manufacturers, where a study of the expected lifetime of a chip under the worst operational conditions is the simplest (and cheapest) option for them. In this situation, we do not need to run the reliability simulation for 20 years; instead, we run it for the time required to profile the operation of the chip, observe the trend of the MTTF degradation, and extend it along the years by applying simple mathematical extrapolation.

3.4.1. The implementation of the reliability model

Regarding the implementation of the *Reliability Model*, it follows the same structure as the thermal library: the reliability is updated in small increments (*Emulation Steps*).

Algorithm 9 shows the new function calls (in bold) added to the original *Thermal Model* in order to calculate also reliability numbers. There are two modifications with respect to Algorithm 7:

First, after the call to *loadInitialTemp()*, that loads the initial temperature (cell.temp) into the model, we must add a call to *loadInitialReliability()* that, in a similar way, loads the initial reliability numbers from a file. These values are stored in the fields *cell.reliability[MTTF]*, *cell.reliability[EM]*, *cell.reliability[TDDB]*, *cell.reliability[SM]*, and *cell.reliability[TC]*.

The second modification includes modifying the main loop: Upon finalization of an emulation slot, the statistics, power and temperatures of the system are updated. Once the new temperatures are available (i.e., just after the call to *updateTemperatures()*), we place a call to *updateReliability()* in order to update the reliability values.

The new values depend on the past history (the former values of the reliability), the current temperature of the circuit, and a set of (technological) constants fixed at design time. The detailed formulas can be found in [CSM+06; SABR05; Sem00].

Algorithm 9: Thermal model with reliability

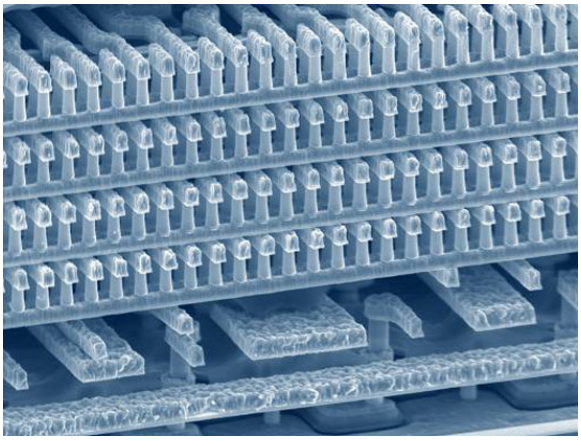
```
Load Floorplan
calculateCellsResistances()
calculateCellsCapacitance()
loadInitialTemp()
// Populates the fields (MTTF, EM, TDDB, SM, and TC) of
// the record cell.reliability.
loadInitialReliability()
time ← 0
emulationfinished ← false
Initialize Emulation
while NOT emulationfinished do
  runEmulationStep()
  retrieveStatistics(systemstatistics, emulationfinished)
  updatePower() {starts the power model.}
  updateTemperatures() {starts the thermal model.}
  updateReliability() {starts the reliability model.}
end while
```

3.5. 3D thermal modeling

Figure 3.14 shows a chip designed using the 3D stacking technology. A key component of 3D technology are the through-silicon vias (TSVs) [Mot09]. They are vertical electrical connections (vias) passing completely through a silicon wafer or die. Their function is to enable communication between two dies as well as with the global package. TSVs are a high performance technique to create 3D packages and 3D integrated circuits, compared to former alternatives such as package-on-package [DYIM07], because the density of the vias is substantially higher.

This solution increases the integration capabilities and frequency of forthcoming MPSoCs [DWM+05; HVE+07] but, on the other hand, it substantially increases power density due to the placement of computational units on top of each other; therefore, temperature-induced problems exacerbate in 3D systems, offering a huge space for design improvements: By carefully choosing their locations on the floorplan, for example, the TSVs can be used to control the SoC temperature. Another method used in state-of-the-art solutions, to tackle the heat-removal challenges of 3D architectures, is to employ microchannels, carrying liquid coolants (water has the ability to capture heat about 4,000 times more efficiently than air), to remove the heat generated [BMR+08].

With the purpose to study this kind of systems and the multiple possibilities they offer for optimizations, I have integrated into the EP a model to



Courtesy: Matrix Semiconductor, Inc.

Figure 3.14: The Matrix's 3D memory chip, an example of the 3D stacking technology.

characterize the thermal behaviour of 3D MPSoCs manufactured with stacking technology. It takes into account the effect of the TSVs, and contains a model for active liquid cooling microchannels. In the following sections, I describe the implementation details, that correspond to these two steps: First, defining a thermal resistor-capacitor (RC) network of the 3D chip stack and, second, adding models for the interlayer material (which includes the liquid flow and TSVs distribution).

The validation of the 3D *Thermal Model* was done experimentally, manufacturing a 3D chip with the multilayer structure of Figure 3.15, containing aluminium heaters and temperature sensors. The heaters allow us to warm-up specific parts of the chip and, reading the sensors, we study how the heat propagates through its structure. The details of the validation process are out of the scope of this thesis, but can be consulted in [RLSC10].

3.5.1. RC network for 2D/3D stacks

As we can observe in Figure 3.15, a 3D chip is made of several silicon layers (tiers) stacked together, and interleaved with inter-tier material, that contains the TSVs and the microchannels [RLSC10]. Based on my *Thermal Model* for 2D chips (Section 3.3), that uses an equivalent electrical circuit (RC grid) to model the heat flow, I have extended it to include 3D modeling capabilities, by adding new elements to model the inter-tier material.

Similar to the work done for the 2D case, the chip structure is divided into small cubical thermal cells. Figure 3.16 shows an example of 3D layout divided in cells; it represents two tiers of silicon plus the inter-tier material. As explained in Section 3.3.1, the cell resolution of the *Thermal Model* can be

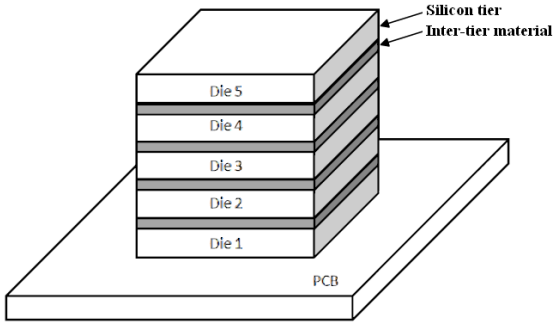


Figure 3.15: Structure of a 3D stacked chip.

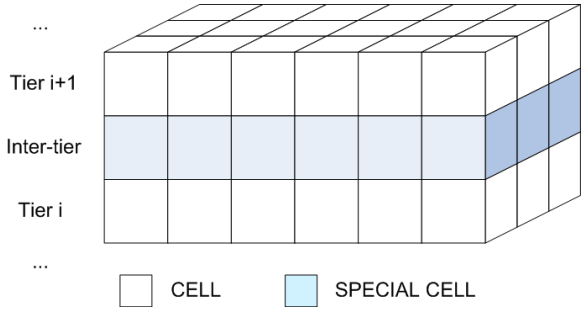


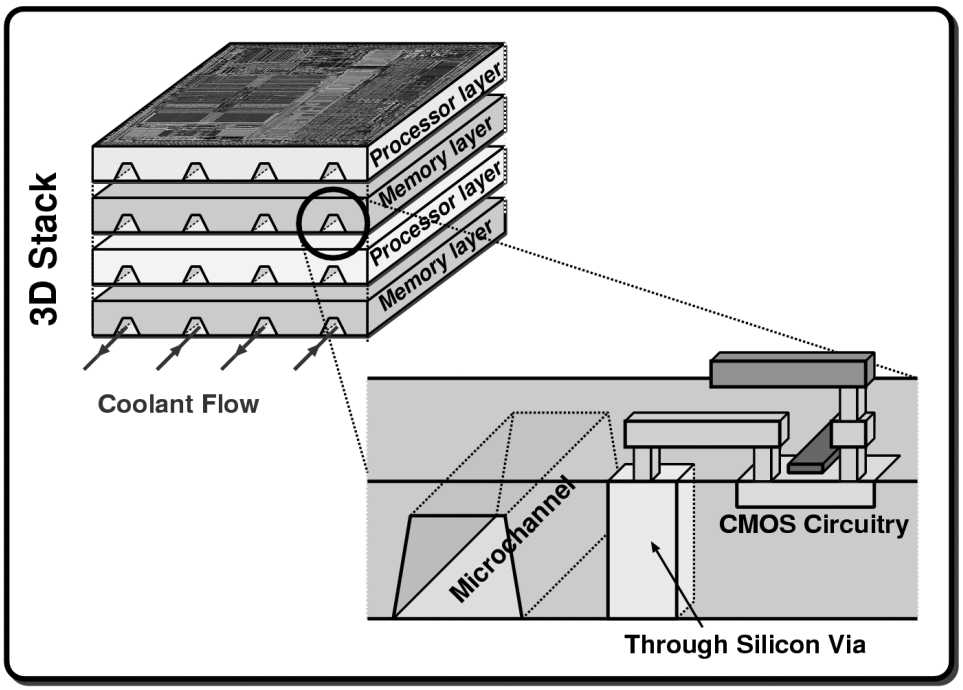
Figure 3.16: Horizontal slice of a 3D chip divided in thermal cells, showing two tiers of silicon plus the inter-tier material.

freely adjusted but, for simplicity, in this case I have considered all the cells as having the same size, including a new type of cells, marked as *SPECIAL CELLS* in the figure, used to model the interface material.

Each cell is then modeled as a node containing six resistances that represent the conduction of heat in all the six directions (top, bottom, north, south, east and west), and a capacitance that represents the heat storage inside the cell, exactly like in the 2D case (Figure 3.8). However, due to the special characteristics of the inter-tier material, see Section 3.5.2 for details, the resistivity value of some of these *SPECIAL CELLS* can vary at run-time.

Finally, current sources are connected to the *active cells* (Figure 3.9), in the regions representing the sources of heat, where the functional units are present. The entire circuit is grounded to the ambient temperature at the top and the side boundaries of the 3D stack through resistances, which represent the thermal resistance from the chip to the air ambient.

At this point, the circuit is completely specified as an RC network, similar to the ones used for single-tier chips (see Figure 3.10). Although we have included new types of cells, internally, they all contain resistances, capacitances and, in some cases, current sources. Therefore, the equation that



Courtesy: IBM Zürich - CMOSAIC - Nano-Tera consortium.

Figure 3.17: Detail of the microchannels and TSVs in the 3D stacked chip.

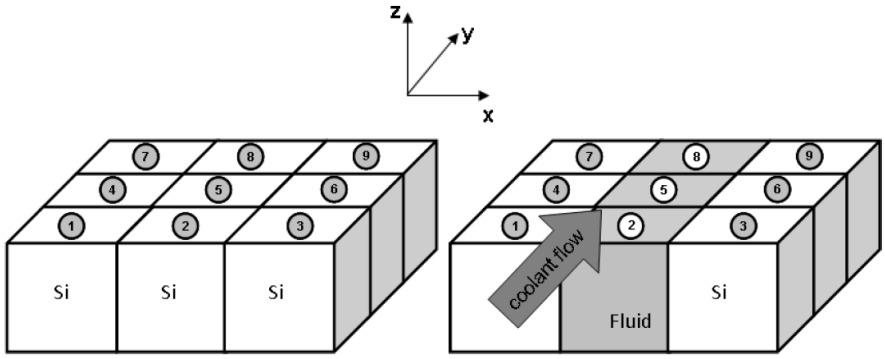
describes the circuit is, again, Equation 3.5, solved applying the methodology explained in Section 3.3.1.

3.5.2. Modeling the interface material and the TSVs

Figure 3.17 shows the 3D stacked chip internals. In this figure, we appreciate different tiers containing the processing cores and memories, interleaved with interlayer material, where the microchannels and TSVs are.

In order to model the heterogeneous characteristics of this interlayer material, I introduce two major differences to other works: (1) as opposed to having a uniform thermal resistivity value for the layer, my infrastructure enables having various resistivity values for each grid, (2) the resistivity value of the cell can vary at run-time.

As depicted in Figure 3.18, the interlayer material is divided into a grid, where each grid cell, except for the cells of the microchannels, has a fixed thermal resistance value depending on the characteristics of the interface material and TSVs. For my considered TSV density (less than 1 % of total chip area, as proposed in [SAAC11]), I assume a homogeneous via distribution on the die, and calculate the combined resistivity of the interface material based on the TSV density (details in Section 3.5.2.1). On the other hand, the



(a) without microchannel (b) with microchannel.

Figure 3.18: Discretization of one layer of interface material into thermal cells.

thermal resistivity of the microchannel cells is computed based on the liquid flow rate through the cell, and the characteristics of the liquid at run-time (details in Section 3.5.2.2).

3.5.2.1. TSVs thermal interference

The TSVs are a key component of the 3D stacking technology that can not be neglected in the thermal studies. They are vertical metallic vias that communicate adjacent layers; thus, affecting the heat propagation [GS05]. Next, It follows a brief study of the impact of the TSVs on the chip temperature, to determine which modeling granularity is required to accurately model the effects of TSVs on the thermal behaviour of 3D MPSoCs.

Figure 3.19 shows the joint resistivity, of the interface material plus the TSVs, as a function of the density of vias (d_{TSV}; the ratio of the total area overhead introduced by the TSVs to the total layer area). It can be observed that, even when the TSV density reaches 1-2 %, the effect on the resistivity is limited to a variation of less than 0.4 mK/W, which represents only a few degrees, and justifies using a homogeneous TSV density in the model.

Therefore, through the rest of this thesis, I can safely assume that the effect of the TSV insertion to the heat capacity of the interface material is negligible, since I keep the area overhead of TSVs below 1 %, a very small percentage of the interface material area. In cases with high thermal interference of the TSVs, however, this effect can be used as an advantage to control on-chip temperatures, through thermal via planning [CZ05].

In my model, I assign a TSV density to each unit (floorplan component) based on its functionality and system design choices (a crossbar structure requires a high TSV density, while a processing core does not require any modeling of TSV interference). In the experiments, each via has a diameter

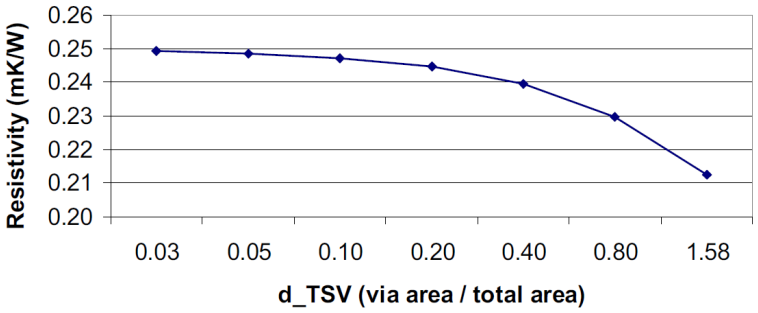


Figure 3.19: Relationship between the TSV density and the resistivity of the interface material.

of 10μm, and the spacing required around the TSVs is assumed as 10μm, according to the current TSV technology [ZGS+08; CAA+09]. I used a joint interlayer resistivity value of $0{,}23mK/W$, assuming an abundant number of vias (total number of vias is 1024) while keeping the area overhead below 1 %. Note that, while the exact location of TSVs might demonstrate a further reduction in temperature in comparison to the homogeneous TSV distribution model, this assumption places over 8 TSVs per mm^2. Assuming a relatively high TSV density in the model reduces the temperature difference in comparison to modeling the exact location of TSVs.

3.5.2.2. Active cooling modeling

A 3D stacked architecture with liquid cooling requires advanced thermal packaging structures. A basic schema is depicted in Figure 3.20. Such a chip uses nano-surfaces (microchannels) that pipe coolants, including water and environmentally-friendly refrigerants, within a few millimeters of the chip to absorb the heat, like a sponge, and draw it away. Once the liquid leaves the circuit in the form of steam, a condenser returns it to a liquid state, where it is then pumped back to the circuit, completing the cycle.

In such a 3D system, the local junction temperature in the microchannels can be accurately computed with conjugate heat and mass transfer modeling. The complexity of the resulting model (for the fluid, only) is in the range of billion nodes to be simulated and, thus, unsuitable for my real-time EP. Instead of using this expensive, in terms of computation, method, I worked with some partners from the Embedded Systems Laboratory (ESL) and the Laboratory of Integrated Systems (LSI) at EPFL, Switzerland, to develop an alternative model based on resistive networks; it runs at a fraction of the computation requirements, while keeping the loss in accuracy negligible.

Inside the 3D grid structure described in Section 3.5.1, I model active cooling properties (i.e., liquid cooling) using a special type of thermal cells

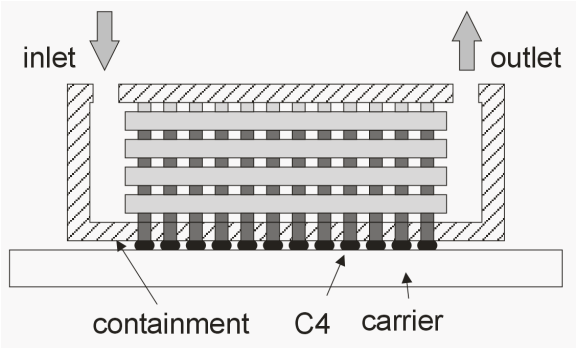


Figure 3.20: Schema of a 3D chip with liquid cooling.

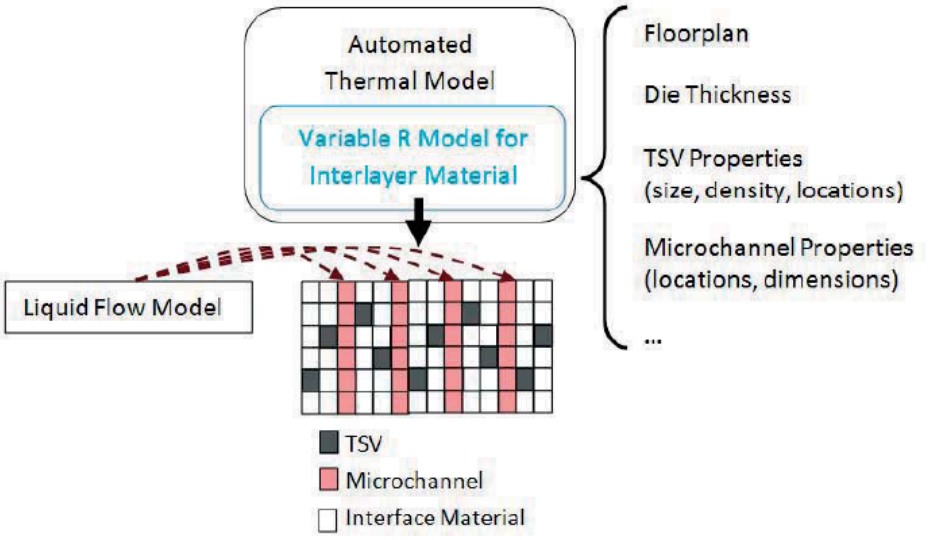


Figure 3.21: Grid structure of an inter-tier layer, showing the layout with the microchannels and TSVs.

Algorithm 10: Thermal Model With Liquid Cooling

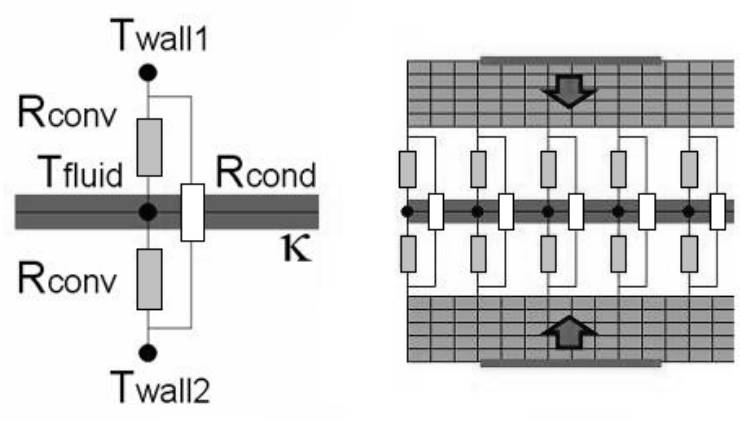
```
Load Floorplan
calculateCellsResistances()
calculateCellsCapacitance()
loadInitialTemp()
time ← 0
emulationfinished ← false
Initialize Emulation
while NOT emulationfinished do
  runEmulationStep()
  retrieveStatistics(systemstatistics, emulationfinished)
  updateParametersOfMicrochannelCells()
  updatePower() {starts the power model.}
  updateTemperatures() {starts the Thermal Model.}
end while
```

with different cooling thermal conductance and resistance properties than silicon and metal layers. These new cells, that model the microchannels, are a special type of *passive cells* (see the *Thermal Model*, in Section 3.3.1), that have a variable thermal resistance and capacity, whose value directly depends on the velocity of the refrigerating fluid being injected through the channel. Figure 3.21 shows the grid structure of an inter-tier layer, where we can appreciate the three types of thermal cells: microchannels, interface material, and TSVs.

When running the thermal model, the actual values of the microchannel cells (resistances and capacitance) must be updated before estimating the system temperatures for each emulation step. Algorithm 10 shows the original thermal model modified to include these calculations (call to *updateParametersOfMicrochannelCells()*).

Therefore, in order to fully specify a cell that models a microchannel, I only need to calculate the equivalent thermal resistances and capacitance. This process requires characterizing the chip stack using a porosity model, i.e. the cavities are seen as 2D-porous media, to study the fluid-solid thermal field-coupling (the heat transfer from the silicon to the liquid).

Figure 3.22a, depicts a single heat transfer unit cell of the resistor network representing the thermal field-coupling of the 2D-porous media (T_{fluid}) with the adjacent 3D-solid walls (T_{wall}). The convective thermal resistance; i.e., the solid-fluid heat transfer, is represented with two grey resistors, labeled as R_{conv}, that connect the walls with the liquid. The conductive thermal resistance; i.e., the solid-solid heat transfer, corresponds to the white resistor, labeled as R_{cond}, that connects both walls. κ represents the cavity permeability. The whole channel is modeled by replicating this discrete element; see



(a) Heat transfer unit cell (b) Model of the whole channel

Figure 3.22: Microchannel modeling.

Figure 3.22b: As fluid advances through the channel, it removes the excess of heat from the adjacent walls. In this new context, the parameters to calculate depend on the permeability of the channel, that depends on the laminar flux being injected through it. The details of the process that obtains, from this system, the equivalent (six) resistances and capacitance of the cells, required by my *Thermal Model* (see Section 3.3.1), are published in [PN06].

3.6. Conclusions

This chapter has been dedicated to describe the SW part of the EP, a set of estimation libraries, written in C++, that run on a desktop computer. They receive the statistics from the FPGA as input and, depending on the model, as output, they generate an estimation of the power consumption, the working temperature, or the system reliability.

The components (libraries) have been described in an incremental way, starting with the power modeling, followed by the thermal library for 2D MPSoCs, and the reliability library. In the last section, I have introduced a generalization to support state-of-the-art 3D thermal modeling, that includes a model for active cooling solutions.

Figure 3.23 describes the interfaces of the three libraries. Although they are depicted chained together, they can also be configured to work individually, receiving the input data from pre-recorded traces, or plugged into external (third-party) simulators. In this figure, the static inputs (i.e., the data that are known at compile time, before starting the emulation), like the floorplan description and the technology parameters, are on the left side of the boxes representing the models; from the upper part, they receive

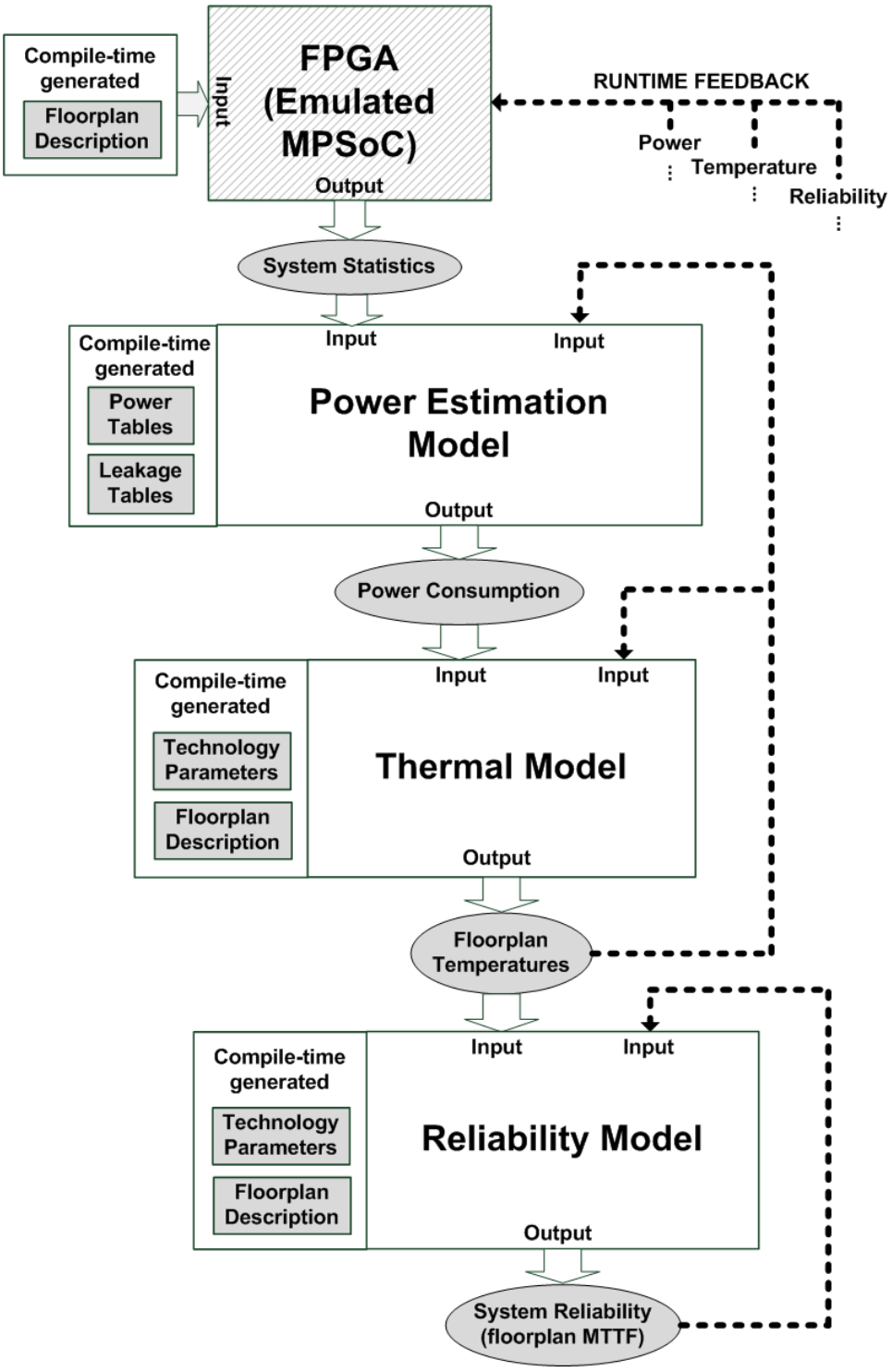


Figure 3.23: Interfaces of the *SW Libraries for Estimation*.

the run-time parameters, like the *System Statistics* or the temperatures, for example. The box representing the FPGA also follows the same format; e.g., at compile time, it needs to know the chip layout, contained in the floorplan description, in order to format the statistics packets accordingly.

This capability to estimate MPSoCs power, temperature and reliability, completes my EP, converting it into a powerful tool for system designers. In the next chapter, I show how to combine the HW and SW parts explained in the previous and current chapters, respectively, and describe the whole emulation flow.

Chapter 4

The Emulation Flow

This chapter describes the EP considered as a whole. On one side, there is the HW running on the FPGA, explained in Chapter 2. On the other, the SW models that run on the host PC, explained in Chapter 3. Both parts work together inside the EP, constituting one integrated framework for MPSoC development.

I describe the platform integration (how to instantiate all the components, configure the system, and perform an emulation), detailing the emulation flow that allows designers to speed-up the design cycle of MPSoCs, the design considerations that arise when putting the different parts together, and the HW and SW elements necessary to setup an EP.

4.1. The HW/SW MPSoC emulation flow

The key advantage of this framework for a realistic exploration of MPSoC designs is that it uses FPGA emulation to model the HW components of the system at megahertz speeds and extract detailed system statistics while, in parallel, these statistics are fed into a SW model that runs in a computer, and calculates the power, temperature, reliability... profile of all MPSoC architectural blocks. Everything is integrated into one overall flow; the one depicted in Figure 4.1. At design time, we first need to configure the FPGA and the host computer. These steps have been labeled, respectively, as Phase 1 and Phase 2 in the figure. Next, at Phase 3, the system initiates the emulation. It follows a detailed description, including numbers that reference the steps in the Figure 4.1.

1. First of all, in Phase 1, **the HW and SW components of the *Emulated System* are defined** (note that this is the SW that will run on the cores inside the FPGA, not the estimation libraries).

   Regarding HW, the user specifies in this phase one concrete architecture (number and type of cores, bus topologies, etc.) (1), configures

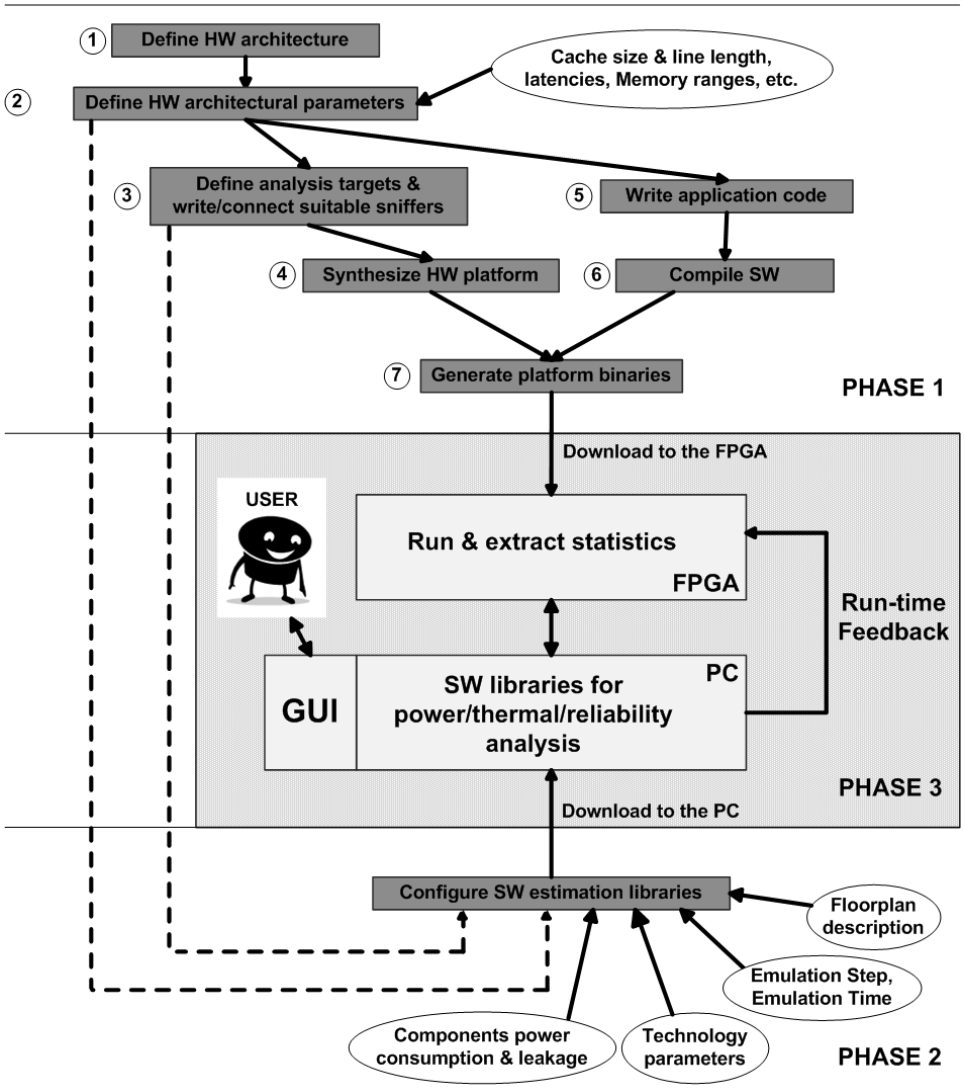


Figure 4.1: The HW/SW MPSoC emulation flow of the Emulation Platform.

the parameters, such as the memory sizes, replacement policies, latencies, etc. (2), and defines the elements that will be monitored (3). *HW Sniffers* are included in the system to extract statistics from each of the three main architectural components that constitute the final MPSoC: processing cores, memory subsystem, and interconnection. This is done by instantiating, in a plug-and-play fashion (cf. Chapter 2), the predefined HDL modules available in the repository for each of the previous three components and the respective sniffers. Next, the HW is synthesized (4).

Related to the SW part, in this phase, the application/s to be executed in the emulated MPSoC (5) is compiled. Using a cross compiler, the binary code is generated for the target processors (6).

At this point, both the HW and the SW are ready, and we can generate the platform binaries (7).

2. In the next phase, Phase 2, **the *SW Libraries for Estimation* are configured**: As the minimum information, we need to indicate the components present in the system, and their types, so that the received statistics can be interpreted correctly (cf. Section 2.2.3.1). If additional analysis, like temperature or reliability are performed, then, we also need to input the characteristics of the thermal cells, the aging parameters, etc. (cf. Chapter 3).

   For thermal studies, for example, the floorplan/s to be evaluated according to the previous (Phase 1) HW definition is defined. Note that, for one architecture, we may have different floorplan solutions. The floorplan description comprises the dimensions, location, and power consumption for each HW component in the emulated MPSoC.

   During this configuration phase, by varying the cell size and number of cells, for example, we can trade-off simulation speed of the SW libraries with its accuracy. The coarser the cells become, the less cells we need to simulate, but the less accurate the temperature estimates become. Figure 3.3, shows a floorplan thermal map generated with the thermal library, where we can appreciate the practical significance of *the cell-resolution of the model*: The temperature evolution in the system is calculated *per cell*. In the figure, observe the division of the floorplan into a set of cells, and that the temperature is constant within a cell.

   The size of the cells minimally affects the time taken by the initialization of the *Thermal Model* (calculation of resistances, etc.). It is the number of cells what determines the time taken by each iteration of the thermal calculations. This dependence is linear.

   Finally, the configurable granularity of the statistics updates and communication between the FPGA and the SW libraries is specified at

this moment (*Emulation Step*). As it can be appreciated in the figure, the maximum duration of the emulation (*Emulation Time*) is also input as a parameter but, normally, the process finishes before, when the *Emulated System* notifies that the SW application under execution completed.

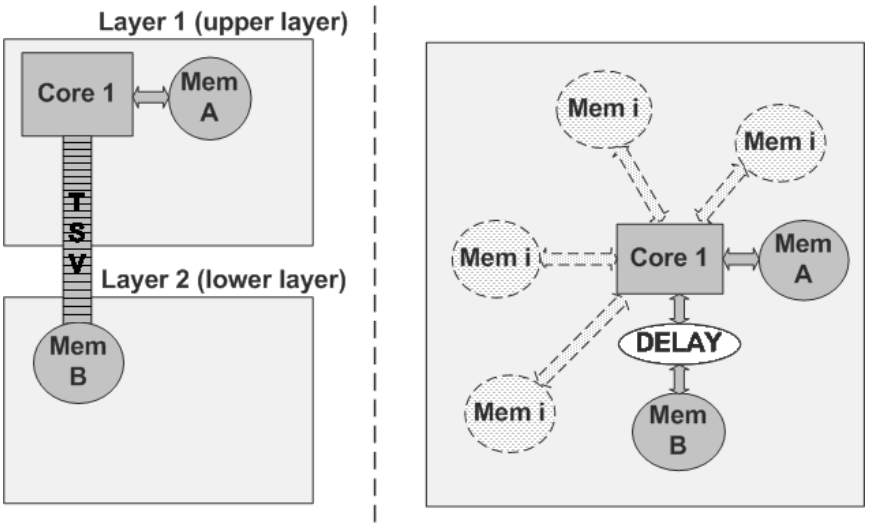
3. At this point, the EP is fully specified. **The last step, Phase 3, is the emulation itself**. It requires connecting the FPGA to the PC: The HW of the EP (*Emulated System* + *Statistics Extraction Subsystem*) is downloaded to the FPGA; next, a graphical interface (GUI, in the figure) is launched in the host computer that provides visual feedback during the process of the emulation, and allows the user to issue a *start* command. After this point, the framework runs autonomously.

   While the *Emulated System* is running, the statistics for each cell defined in the layout are concurrently extracted, and sent to the SW libraries running onto the host computer. They will generate output values (power, temperature, reliability) that may be just logged down, or sent back to the FPGA (see the 'Run-time Feedback" arrow in Figure 4.1). In this later case, the *Emulated System* can use this information to modify its own behaviour in real-time. As an example, since the thermal simulator calculates in real-time the new temperatures, we can feed the updated values back into the FPGA, and store them in registers that emulate the presence of thermal sensors in the target MPSoC in certain positions of the floorplan. If these registers are mapped in the memory hierarchy of the *Emulated System*, so that they are accessible from the running multi-processor OS, providing real-time temperature information, we make up a closed-loop thermal monitoring system.

4.1.1. Emulation of a 3D chip with an FPGA

To understand how we model a 3D architecture using a 2D FPGA, take a look at the 3D system depicted in Figure 4.2a. It consists of two layers: In the upper layer there is a core that can access two local memories: A (in the same layer), and B (located in the lower layer). An access to memory A will take less time and will consume less power than accessing B.

When emulating this system in an FPGA, we have to map everything in a 2D layout (Figure 4.2b). If we abstract the floorplanning information, what is different in the behaviour of systems a and b is the latency. Assume, for example, that accessing memory A takes 1 cycle, whereas accessing memory B takes 6 cycles. We instantiate in the FPGA a processor connected to two memories symmetrically and, then, we simply add a new element that simulates this extra latency (*DELAY* oval in the figure). The behaviour of the system, then, will be the same as the one in the 3D case.



(a) Modeled two-layered floorplan.

(b) Mapping alternatives of the components on the FPGA.

Figure 4.2: Emulation of a 3D chip with an FPGA.

From the point of view of the *Thermal Model*, the data interface remains the same: we only receive *activity numbers* associated to the different elements of the floorplan (number of accesses to memory X, number of transactions in bus Y...), so there is no difference. Nevertheless, when calculating temperatures, the *Thermal Model* knows that the bus of memory A is different from the bus of memory B: different materials, capacitances, etc... It should be noted that, inside the FPGA, it is completely irrelevant where we place memory B, as far as the behaviour is the same (number of cycles per access, type of the bus...); the actual floorplan of the final chip is in the *Thermal Model*, that runs on the PC. Any of the positions suggested in Figure 4.2b as *Mem i* would be valid: no matter at what side of the processor we place the memory in the FPGA, since it will be modeled as being underneath it, in a different layer, as it appears in Figure 4.2a.

4.1.2. Emulating virtual frequencies

The EP makes possible to emulate HW configurations that run at a different speed than the allowed clocked speed of the available HW components. In fact, it is similar to the mechanism used in SW simulations, but at a higher frequency. For instance, it is possible to explore the effects in thermal modeling of a final system clocked at 500 MHz, even if the present cores of the FPGA can only work at 100 MHz. To this end, instead of using a 10 ms statistics sampling frequency with a clock running at 500 MHz, we must use

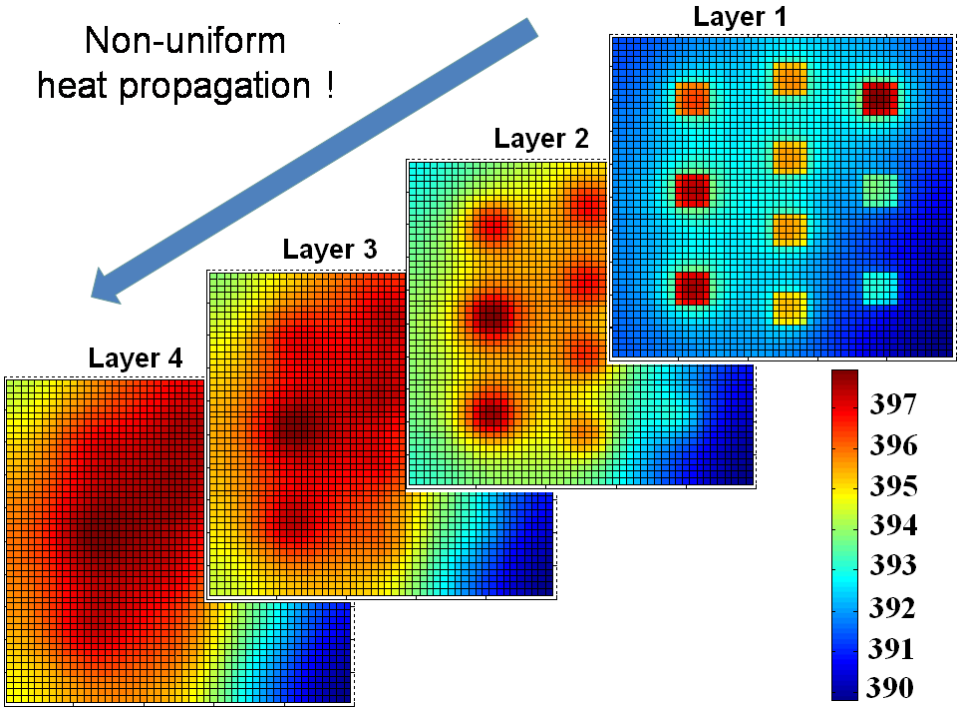


Figure 4.3: Instantaneous thermal map generated with the Emulation Platform for a four-layered 3D MPSoC.

a *Virtual Clock* of 100 MHz (maximum clock allowed in the FPGA emulation after synthesis), and collect the statistics every 50 ms. The switching activity in each MPSoC component monitored at this interval is equivalent to the target system for 10 ms. Therefore, the HW inside the FPGA samples every 50 ms of real execution, but it is analyzed by the SW estimation library (running in the PC) as representing 10 ms of the target MPSoC emulated execution. The major requirement, in this case, is the definition of the sampling/emulating frequency and the target MPSoC frequency to configure the SW estimation model accordingly.

4.1.3. Benefits of one unified flow

The proposed emulation framework integrates in one single tool the benefits of HW emulation and fast SW simulators to estimate power, temperatures, and reliability of 2D/3D MPSoCs. Overall, it is a powerful tool that allows system designers to easily characterize the system under development, speeding-up the development cycle. Figure 4.3 represents an example of such characterization.

It shows a detailed transient thermal map of a 4-tier chip containing 10 cores per layer. Each of them with different activity profiles. The system is

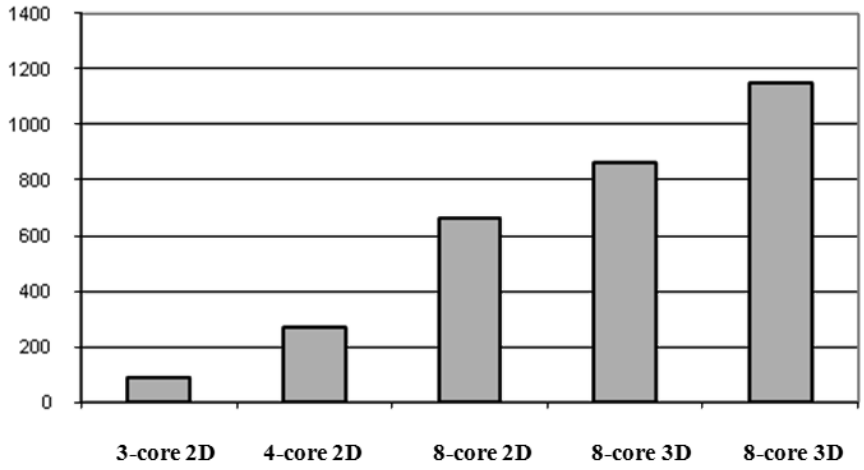


Figure 4.4: Speed-ups of the proposed HW/SW thermal emulation framework for transient thermal analysis with respect to state-of-the-art 2D/3D thermal simulators.

modeled dividing each layer of the floorplan in a regular grid of 50x50 thermal cells. The graphical representation of the system temperatures propitiates to easily appreciate the non-uniform propagation of the heat inside the stack.

The EP can obtain a transient thermal map of the *Emulated System*, like the one in Figure 4.3, for any particular moment of the emulation. With this information, the system designer can issue a command to adapt the system, e.g.: reducing the working frequency of a particular core, and see its effects inmediatly. Using independent flows, like obtaining the application execution traces and, afterwards, feeding them into a thermal simulator does not provide realistic results. The integrated flow of the EP allows system designers to test the real applications on the final HW before going to silicon.

Regarding performance, the next chapter details several experiments conducted in order to compare the EP with a SW simulator. For a quick estimation, I have synthesized those results in Figure 4.4. The numbers show significant speed-ups with respect to state-of-the-art temperature estimation frameworks [MPB+08] [ADVP+06] [CAA+09]. In particular, these results outline that the proposed modeling approach for MPSoC HW/SW thermal emulation scales significantly better than state-of-the-art SW simulators for transient thermal analysis. In fact, the results of the exploration of 2D thermal behaviour on a commercial 8-core MPSoC [KAO05] have shown that the proposed thermal emulation can achieve speed-ups of more than to 800x with respect to state-of-the-art SW-based thermal simulators [BBB+05].

Moreover, the thermal exploration of 3D MPSoCs with active cooling (liquid) modeling shows even larger speed-ups (more than 1000x) due to power extraction and thermal synchronization overhead in thermal simulators

[SSS+04; PMPB06; CAA+09].

4.2. Requirements: FPGAs, PCs, and tools

In this section, I describe the necessary elements (FPGA, PC and tools) required by the EP. Everything has been intentionally designed in a very generic way, to avoid dependences on a specific manufacturer, board, PC, or tool.

For the sake of standarization, both the *Emulation Engine* and the *Emulated System* are specified in standard and parameterizable VHDL, because all the existing FPGAs support this hardware description language. However, they can be specified in any other language: from Verilog or SystemC, to high level synthesis languages. The decision is left at the designer's choice. He can even use a mixture of different languages, as long as it can all be translated into a final netlist and mapped into the target FPGA. The only additional requirements are the availability of a communications port onboard, to interact with the SW libraries running on the host computer; a compiler for the included cores; and a method to upload both the FPGA synthesis of the framework and the compiled code of the application under study.

In this research, I have been working with Xilinx FPGAs. This manufacturer provides all the basic tools (to synthesize the VHDL, compile the SW for the embedded cores, and download both binaries to the target board) in its *Embedded Development Kit (EDK)* framework for FPGAs. Xilinx's EDK tool is an integrated environment, intended for the creation of mixed HW/SW systems. It includes an HDL code editor and synthesis engine, called Integrated System Environment (ISE). Any developed module with this tool can be added to a repository, and instantiated in EDK by dragging-and-dropping it with the mouse. Included, as well, are GNU C (`GCC`) and C++ (`G++`) compilers/linkers for the PowerPC and Microblaze cores available in the repository. Also, EDK enables loading different binaries on each processor of the system. Thus, if the application to be run is already written in any of these languages, no effort is required for the designer.

Regarding area requirements, the size of the FPGA depends on the dimensions of the *Emulated System*. It may vary from tiny FPGAs, when only a module or core is being characterized/optimized/debugged, to the biggest FPGAs available on the market. However, for typical MPSoCs, an off-the-shelf mid-range FPGA suffices. My main development platform, for example, was a Xilinx Virtex 2 Pro vp30 board (or V2VP30) with 3M gates, which costs $2000 approximately in the market, and that includes two embedded PowerPCs, various types of memories (SRAM, DDR, flash...) and an Ethernet port. Table 4.1 shows some of the target boards used during this research, including the capacity of the FPGAs (in Slices) and the internal RAM me-

Table 4.1: FPGA boards used during this thesis.

Board	FPGA	Slices	Block RAM
XUP Virtex II Pro Devel. System	XC2VP30	30,816	2,448 KB
XUP Virtex 5 Devel. System	XC5VLX110T	17,280	5,328 KB
AVNET Virtex-II Pro Devel. Kit	XC2VP30	30,816	2,448 KB
ML505 Eval. Platform	XC5VLX50T	7,200	2,160 KB
Spartan-3 Starter Kit	XC3S200	4,320	216 KB
Platform Baseboard for the ARM11 MPCore	XC4VLX40	18,432	1,728 KB

Table 4.2: Contents of one slice in different FPGA families.

Family	FPGA	Contents of one slice
Spartan-3, Virtex-2 Pro	XC3S200, XC2VP30	One 4-input Look-Up Table (LUT), and one D flip-flop.
Virtex-4	XC4VLX40	Two 4-input LUTs, and two flip-flops.
Virtex-5	XC5VLX50T, XC5VLX110T	Four LUTs that can be configured as 6-input LUTs with 1-bit output or 5-input LUTs with 2-bit output, and four flip-flops.

mory available. Since the size of a slice depends on the FPGA family, I have included, in Table 4.2, the contents of the different families of Slices. The column "Block RAM" in Table 4.1, indicates the amount of RAM embedded inside the FPGA chip (the on-chip RAM), that receives this denomination in the particular case of Xilinx FPGAs.

The complete development flow can be observed in Figure 4.5. Initially, the HW and the SW are developed independently. When they are both mature, boths flows are merged to generate the system bitstream. Observe, in the figure, the rhombuses. They represent the processes of simulating, debugging and verifying the design. At any of these points, if a design error is detected, the system designer must roll back to a previous development stage in order to solve the problem.

In addition to the aforementioned SW, required to synthesize the HW platform, the user needs to compile the SW libraries that run on the host PC and interact with the FPGA to estimate power, temperature and reliability. As stated in Chapter 2, they have been written in the C++ language. Thus, any standard C++ compiler (like G++) can be employed to generate the executables. In my particular case, I have used the Visual Studio Suite [Mic], from Microsoft, to write, compile, and debug this code. In one single IDE it integrates the editor, compiler and debugger.

The communication FPGA-PC is resolved in the C++ source code with the help of a custom API that I have developed to make the code more portable and versatile. Table 4.3 summarizes the API interface. In my current

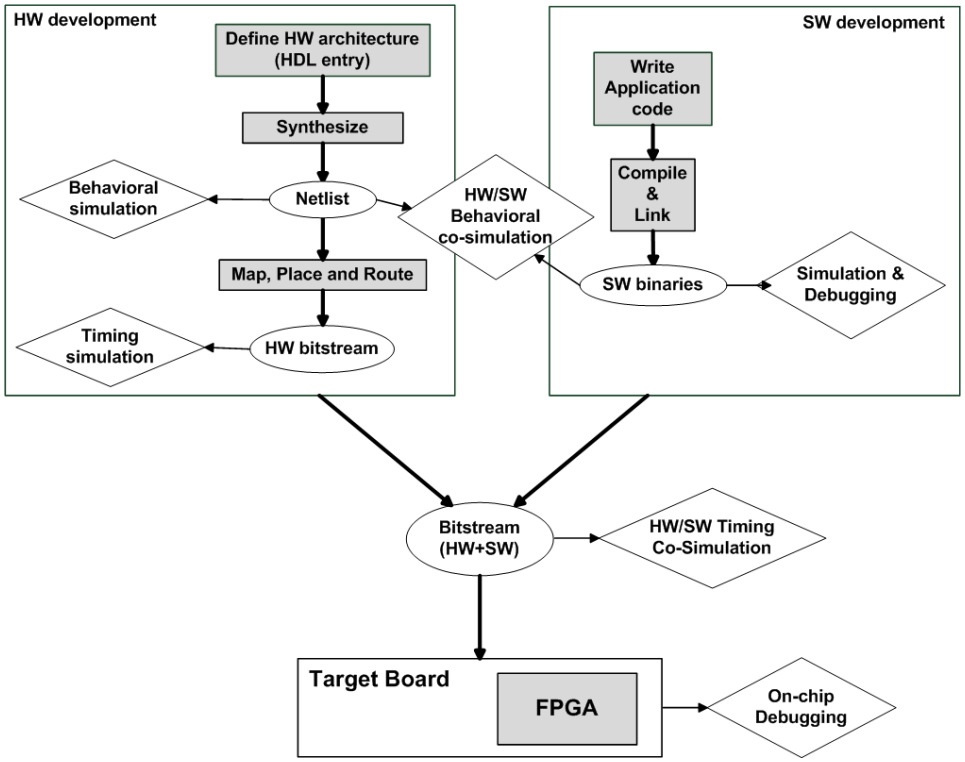


Figure 4.5: FPGA design flow.

Table 4.3: Functions of the communications library.

initializeConnection	Sets the values that configure the communications channel, and performs the necessary initialization.
receiveData	Receives the statistics of one emulation slot from the FPGA.
sendData	Sends the data calculated by the SW models in one emulation step (temperatures, reliability...) from the PC to the FPGA.
dumpDataToFile	Stores the information generated (by the FPGA or by the SW models) in one emulation step, into a file.

implementation, I used an Ethernet connection; thus, the communications library, internally, makes use of functions like "send/receive Ethernet packet" to implement the interface functions "send/receive data". These low-level functions to handle the Ethernet packets come from the *libpcap* library, a portable (multi-platform) C/C++ library for network traffic capture that is available as an open source project, in [bibe].

I conclude this section with a brief reference to the characteristics of the computer used as the host PC: Although I can not indicate the exact minimum system requirements, during the development process of the platform, I have used off-the-shelf desktop computers, starting from a Pentium 4 with 256 MB of RAM, and it was enough to run the platform at full speed (with the FPGA at 100MHz). In fact, as I explain in Chapter 6, the only observed stalls were due to the bandwidth limitations of the communications port.

4.3. Synthesis results

For completeness, I present, in this section, some practical use-cases of the platform, including a summary of the synthesis reports, showing the amount of resources occupied.

The FPGA fabric is made of Flip flops (FFs), LookUp tables (LUTs), and some memory elements, that are typically grouped into Slices. The resources utilization of the FPGA is given as the percentage of the total number of Flip-Flops and LUTs used (Slice Logic Utilization). However, the mapper packs the individual LUTs and FFs into Slices, and often they are only partially used. For this reason, I include another number in the synthesis reports: the Slice Logic Distribution, that indicates the percentage of the board Slices being used (either totally or partially). The details of the FPGA boards can be consulted in Tables 4.1 and 4.2. In addition to these numbers, the reports also include the percentage of internal RAM used (BlockRAM).

Emulation Engine:

The *Emulation Engine* is common to all the modeled MPSOC cases, and it is composed of the following elements:

- A microcontroller: either a Microblaze (cases 1 and 2) or a PowerPC (case 3). Both are simple 32-bit RISC processors. The numbers indicated correspond to the case of the Microblaze. Using the PowerPC reduces the logic utilization by 1 %, since the processor is already implemented in the silicon, requiring only a few slices to interface the peripherals.
- 128KB local memory with local memory bus and memory controller.
- Peripherals bus with timer and interrupt controller.
- Clock and Reset generator.
- Debug Module: enables the HW on-chip debugging of the platform via the JTAG connector.
- Ethernet controller.
- CompactFlash controller (to load configurations).

The synthesis results for the described system:

Board : Xilinx Virtex 5 University Program Board
Target Device : xc5vlx110t
FPGA Family : Virtex 5

Design Summary :
Slice Logic Utilization: Flip-flops 10 % and LUTs 11 %
Slice Logic Distribution: 27 %
Total BlockRAM Memory used: 39 %

Emulated Systems:

The first two examples give us a hint of how the framework scales. Case 1 shows a simple *Emulated System* containing 1 32-bit RISC processor, while case 2 is the generalization to 5 processors. The *Emulation Engine* takes 10 % of Slices, and 39 % of BRAM, while each Emulated Subsystem (made of one emulated processor with its corresponding peripherals) takes, approximately, the 6 % of the FPGA (and 5 % of the BRAM).

CASE 1: One simple 32-bit RISC processor subsystem, containing:

- Two local memories, of 8KB each, connected to independent local memory buses: one for instructions, and one for data.
- Peripherals bus with timer, interrupt controller and UART.
- Main memory controller (with 512 MB of DDR RAM).

Sniffers: In the core and the memory controller.

The synthesis results for this emulated system:

Board : Xilinx Virtex 5 University Program Board
Target Device : xc5vlx110t
FPGA Family : Virtex 5

Design Summary :
Slice Logic Utilization: Flip-flops 16 % and LUTs 18 %
Slice Logic Distribution: 37 %
Total BlockRAM Memory used: 44 %

CASE 2: Five 32-bit RISC processor subsystems.

Each of them connected to the components described in CASE 1, and with access to a shared bus, containing the following elements:

- Shared UART.
- Main memory controller (with 512 MB of DDR RAM).
- Inter-processor synchronization modules.

Sniffers: In each core and memory controller, as well as in the shared modules.

The synthesis results for this emulated system:

Board : Xilinx Virtex 5 University Program Board
Target Device : xc5vlx110t
FPGA Family : Virtex 5

Design Summary :
Slice Logic Utilization: Flip-flops 41 % and LUTs 47 %

Slice Logic Distribution: 80 %
Total BlockRAM Memory used: 64 %

Case 3 presents a different scenario. It contains the Leon 3 core, a complex processor from Gaisler research with SPARC V8 architecture [Gaib], used for microarchitectural study.

CASE 3: One Leon 3 processor (Sparc architecture), configured with:

- SPARC V8 instruction set with V8e extensions and 7-stage pipeline
- Hardware multiply, divide and MAC units
- Separate instruction and data cache (Harvard architecture)
- Caches: 2 ways, 32 KB. LRU replacement
- Local instruction and data scratch pad RAM, 32 KB
- SPARC Reference MMU (SRMMU) with TLB
- AMBA-2.0 AHB (Advanced High-performance Bus) bus interface. AHB peripherals: timer, interrupt controller and UART.
- Advanced on-chip debug support with instruction and data trace buffer

Sniffers: Connected to the register file of the Leon.

The synthesis results for this emulated system:

Board : Xilinx Virtex-II Pro XUP Evaluation Platform Rev C
Target Device : xc2vp30
FPGA Family : Virtex-II Pro

Design Summary :
Slice Logic Utilization: Flip-flops 15 % and LUTs 40 %
Slice Logic Distribution: 51 %

In this case, the *Emulated System* (the Leon) occupies 36.4 % of the board Slices (4,911 out of 13,696), while 2,000 Slices are dedicated to the *Emulation Engine*, that represents 14.6 % of the FPGA occupation. The area occupancy of the *Emulation Engine* differs from that shown for cases 1 and 2. The reason is that we use different FPGAs: A Virtex II Pro in case 3, and a Virtex 5 for cases 1 and 2, so a word of caution when comparing the number of Slices with different FPGA models.

The results shown in this section prove that within a standard FPGA we can emulate complex systems made of several microprocessors (Case 2 shows an MPSoC with 5 cores) or include really complex cores (like the Leon3 of Case 3) and, still, there is plenty of free space to include more elements.

Regarding the time required to set up an emulation, it highly depends on the skills of the designer but, as a reference, for a person who is familiar with the development tools, for a complex MPSoC with 8 processors and 20 additional HW modules (all of them already verified), the set-up phase requires 10 to 12 hours overall, including the complete synthesis phase. Moreover, modifications in the current configurations of the cores take less than 1 hour to be re-synthesized, while the compilation of additional SW only takes minutes.

4.4. Conclusions

While the previous chapters were dedicated to describe in detail the different parts of the EP, this one, however, presents the FPGA-based emulation framework as a whole, as the integrated tool developed to aid state-of-the-art chip designers.

First, I have detailed the complete emulation flow, explaining, step by step, the procedure to setup an emulation. For completeness, I have included in this section information about how to emulate a 3D chip with an FPGA, and how to manage virtual frequencies. I have also explored the benefits of using a unified design flow for the MPSoC design cycle. Next, I have enumerated all the necessary elements required to build an EP: both on the HW side (FPGAs and PCs) and on the SW side (tools).

Finally, I have concluded the chapter giving some examples of synthesized platforms, so that system designers can get a rough idea of the area requirements.

Chapter 5

Experiments

This chapter presents three case studies aimed at showing the practical use of the EP to evaluate the impact of design decitions (ranging from the floorplan layout to the compiler selection) into the performance, temperature or reliability of the target MPSoC. The results are compared against other exploration frameworks, for a reference.

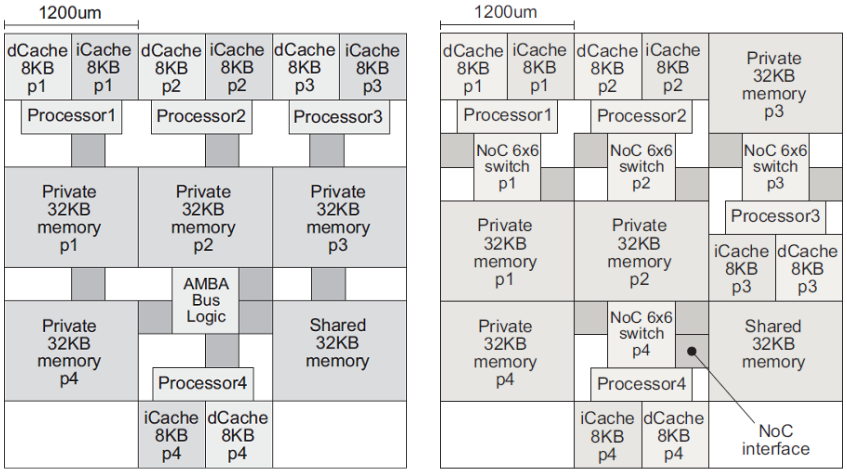
In the first part, I use the EP to evaluate the impact that different HW design alternatives have into the thermal profile of the final chip. Having this information at design time, allows the designer to choose the right floorplan, the best package, or decide if it is worth implementing Dynamic Frequency Scaling (DFS) support.

In the second set of experiments, I introduce a reliability enhancement policy aimed at extending the lifespan of a processor by reducing the stress induced in its register file. The policy is validated empirically using the EP onto a Leon3 Sparc V8 processor core, showing its benefits at the microarchitectural level.

The last experiment shows the application of the EP to the elaboration of system-level thermal management policies, implemented at the operating system (OS) level: When several microprocessors come into play, multiprocessor operating systems (MPOSes) require a middleware able to offer advanced mechanisms, such as task migration and task scheduling, to effectively regulate the temperature.

5.1. Thermal characteristics exploration

In the following sections, I apply the presented framework to different stages of the design cycle of a complex MPSoC case study based on ARM7 cores [Hol]. First, in Section 5.1.1, I describe the experimental setup. Then, I assess the performance and flexibility of the proposed emulation framework in comparison with the MPARM framework [BBB+05], by running several examples of multimedia benchmarks (Section 5.1.2). Next, I perform a detai-



(a) AMBA bus (b) NoC

Figure 5.1: Two interconnect solutions for the baseline architecture of the case study.

led thermal analysis of the system, using the EP as a tool to test a run-time DFS mechanism (Section 5.1.3), evaluate different thermal-aware floorplan solutions (Section 5.1.4), and compare various packaging techniques (Section 5.1.5).

5.1.1. Experimental setup

In the first place, I describe the basic experimental setup: I detail the base architecture (HW and SW) of the MPSoC structure chosen as case study, the configuration of the *Thermal Model*, and some details of the MPARM framework, that will be used as a reference for comparisons.

5.1.1.1. Emulated Hardware

From the HW viewpoint, I have defined a system that can be generalized to n RISC-32 processing cores. Each core is attached to two local, 8KB, direct-mapped, instruction and data caches, using a write-through replacement policy and to a 32KB cacheable private memory. A 32KB shared memory is included in the system.

The memories and processors are connected using either an AMBA bus, or a simple NoC created using XPipes [JMBDM08]. Figure 5.1 depicts the two alternative floorplans resulting with n=4: Figure 5.1a contains the bus-based solution, while Figure 5.1b uses the NoC interconnect instead, with 4 6x6 switches and 9 Network Interface (NI) modules.

Both floorplans have been designed in 0.13 μm technology. The 4 ARM7

can be clocked at, up to, 500 MHz, and the interconnect is clocked at the same frequency than the cores. Each of the components present in the picture contains one associated sniffer that monitors the activity of that particular module.

I evaluated various configurations of interconnections and processors (1 to 8). As an example, the MPSoC design with bus interconnect and 4 processors (the one in Figure 5.1a), contains 30 HW MPSoC components in total (and the 30 *HW Sniffers* associated), consumes 66 % of the V2VP30 FPGA and runs at 100 MHz. Next, I have explored the use of NoCs [JMBDM08] instead of buses. The tested NoC (see Figure 5.1b) has 4 32-bit switches with 6 inputs/outputs and 3-package buffers. This NoC-based MPSoC required 80 % of the FPGA.

5.1.1.2. Emulated Software

As SW applications, first, I created a kernel, called MATRIX, that performs independent matrix multiplications at each processor private memory and combines the results in shared memory at the end. Second, I implemented a dithering filter, named as DITHERING, that uses the Floyd algorithm [FS85] over two 128x128 grey images, divided in 4 segments and stored in shared memories; this application is highly parallel and imposes almost the same workload in each processor. Finally, I defined the MATRIX-TM benchmark, that keeps the workload of the processors close to 100 % all the time, pushing the MPSoC to its processing power limits to observe effects in temperature. This benchmark implements a pipeline of 100K matrix multiplications kernels based on the MATRIX benchmark: each processor executes a matrix multiplication between an input matrix and a private operand matrix, then feeds its output to the logically following processor. The platform receives a continuous flow of input matrices and produces a continuous flow of output matrices. Every core follows a fixed execution pattern: (i) copy of an input matrix from the shared memory to its private memory; (ii) multiplication of the new matrix by a matrix already stored in the private memory; (iii) copy of the resulting matrix back to the shared memory. During the whole execution, interrupt and semaphore slaves are queried to keep synchronization, creating an important amount of traffic to the memories.

5.1.1.3. Thermal Model Setup

The considered floorplans, shown in Figure 5.1, have been divided into 128 thermal cells. The cell sizes used are $150um * 150um$. I consider that the power is uniformly burnt in this region, which represents 1/8th of the size of an ARM7 processor in 0.13 μm. For technologies with a worse thermal conductance, such as fully depleted silicon-on-insulator [biba], it is possible to use smaller thermal cells (down to the level of standard cells).

Table 5.1: Thermal properties used in the experimental setup.

silicon thermal conductivity	$150 \cdot \left(\frac{300}{T}\right)^{4/3} W/mK$
silicon specific heat	$1.628e-12 J/um^3K$
silicon thickness	$350um$
copper thermal conductivity	$400W/mK$
copper specific heat	$3.55e-12 J/um^3K$
copper thickness	$1,000um$
package-to-air conductivity (low-cost)	$40K/W$

Table 5.1 enumerates the thermal properties used during the experiments. Regarding package-to-air resistance, I consider the case of very low-cost packaging, where a good average value is 40W/K [BEAE01], because of the uncertainty of final MPSoC working conditions. However, since this value is higher than the actual figures published by some package vendors, in Section 5.1.5 I also study the effect of different packaging solutions for MPSoCs; in the case of embedded systems, the amount of heat that can be removed by natural convection strongly depends on the environment and the placement of the chip on the PCB.

5.1.1.4. MPARM

Throughout this chapter, the MPARM framework [BBB+05] is used as the reference MPSOC SW simulator. Initially developed at the Department of Electronics, Computer sciences and Systems (DEIS) of the University of Bologna, MPARM is a complete environment for MPSoC architectural design and exploration. Its structure can be observed in Figure 5.2. It integrates in one platform the simulation of both the HW and the SW components. Internally, it is an event-based simulator, written in SystemC.

The main features of MPARM are:

- Supports full modeling of HW and SW architectures for heterogeneous platforms, including a wide range of CPUs, memories, and communication architectures (Buses, NoCs...).
- Several OSes have been ported, offering inter-processor communication libraries.
- It can be connected to third-party models; e.g: cycle-accurate power models and thermal libraries, to provide temperature estimations.
- Higly integrated with third-party tools, that can be incorporated to the design flow. E.g.: XpipesCompiler [JMBDM08] (NoC design) and Sunfloor [SMBDM09] (floorplanning tool).

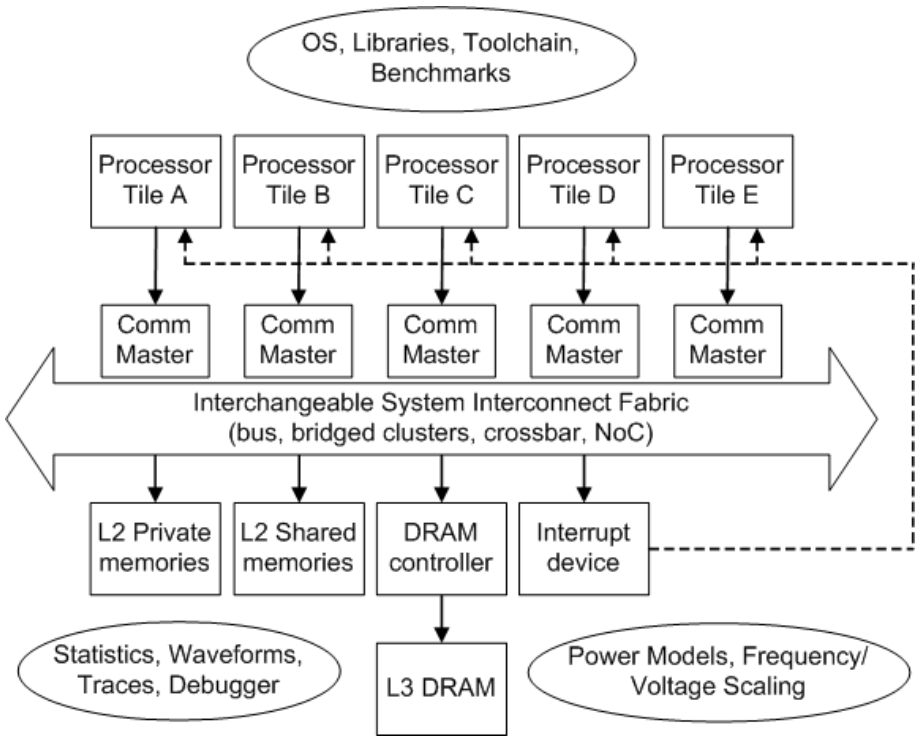


Figure 5.2: The MPARM SystemC virtual platform.

In all the experiments, MPARM is executed on a Pentium 4 at 3.0GHz with 1GB SDRAM and running GNU/Linux 2.6.

5.1.2. Cycle-accurate simulation vs HW/SW emulation

After completing the implementation of the bare MPSoC emulation framework for system architecture exploration, I performed the first set of experiments, aimed at testing the functionality of the integrated framework and to assess the performance of the tool in comparison to cycle-accurate simulators.

5.1.2.1. MPSoC architecture exploration

In the experiment, I have compared the time taken by the EP and the MPARM to complete the execution of the selected SW applications on the different HW architectures (see Table 5.2). As SW kernels, I used the Matrix application, and the dithering filter (Dithering), both explained in Section 5.1.1.2, particularized for the actual number of cores in the system.

The obtained timing results are depicted in Table 5.2. These results show that the HW/SW emulation framework scales better than SW simulation. In fact, the exploration of MPSoC solutions with 8 cores for the Matrix driver

Table 5.2: Timing comparisons between my MPSoC emulation framework and MPARM.

Benchmark	MPARM	HW Emulator	Speed-Up
Matrix (1 core)	106 sec	1.2 sec	88×
Matrix (4 cores)	5 min 23 sec	1.2 sec	269×
Matrix (8 cores)	13 min 17 sec	1.2 sec	664×
Dithering (4 cores-bus)	2 min 35 sec	0.18 sec	861×
Dithering (4 cores-NoC)	3 min 15 sec	0.17 sec	1,147×
Matrix-TM (4 cores-NoC)	2 days	5'02 sec	1,612×

took 1.2 seconds per run in the EP, but more than 13 minutes in MPARM (at 125 KHz), resulting in a speed-up of 664×. Moreover, the exploration of NoCs with complex SW drivers (DITHERING) shows larger speed-ups (1,147×) due to signal management overhead in cycle-accurate simulators. As a result, the HW/SW emulation framework achieved an overall speed-up of more than three orders of magnitude (1,147×), illustrating its clear benefits for the exploration of the design space of complex MPSoC architectures compared to cycle-accurate simulators.

5.1.2.2. Thermal modeling

Using the experimental setup described in the previous section for the MPSoC with four cores and a NoC (Figure 5.1b), I have verified the capabilities of real-time interaction between the HW FPGA-based emulation and the SW thermal library, and compared them to pure cycle-accurate SW simulation.

In order to model the system temperature, I have divided the considered floorplan in 128 thermal cells, and used the thermal properties from Section 5.1.1.3). As SW application, I use the *Matrix-TM* benchmark. The obtained timing results (last row of Table 5.2) show that the HW/SW emulation framework takes 5 minutes approximately for the whole execution of the benchmark, including thermal monitoring, versus 2 days in MPARM for just 0.18 sec of real execution (left corner on Figure 5.3); Thus, my framework achieves a speed-up of 1,612×, more than three orders of magnitude compared to SW-based thermal simulation, making feasible to study in a reasonable time long thermal effects.

5.1.3. Testing dynamic thermal strategies

In order to observe thermal effects on the MPSoC, I have performed a long emulation in the EP framework, running at 500 MHz, with real-life embedded applications. I ran the MATRIX-TM workload for 100K iterations. The results, shown in Figure 5.3, indicate the need to perform long

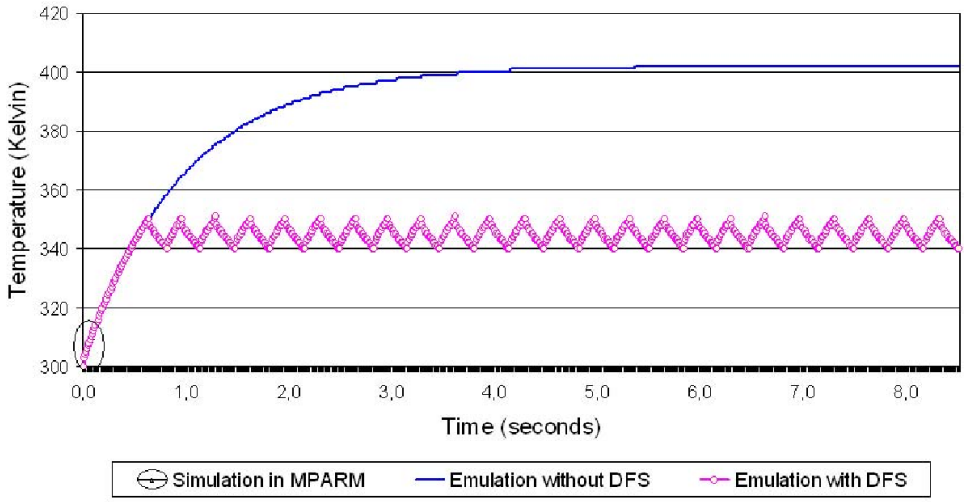
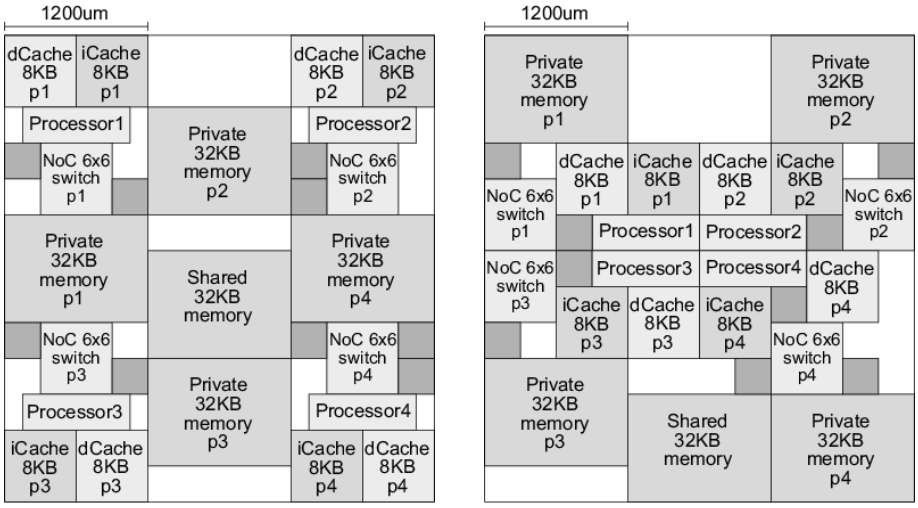


Figure 5.3: System temperature evolution with and without DFS.

emulations to estimate thermal effects (note in Figure 5.3 that the previous simulation in MPARM only represents a very limited part of the overall MPSoC thermal behaviour). Due to the high rise in temperature observed in the MPSoC design, I used the HW/SW emulation framework to explore the possible benefits of DTM techniques. To this end, I implemented a simple threshold monitoring policy using the available HW temperature sensors in my framework.

I modified the VPCM module, implemented with several Digital Clock Managers (DCMs) available in the FPGA fabric, and able to generate multiple clock frequencies (see Section 2.2.1). Inside it, I included a simple pure-HW controller with the ability to dynamically change the output frequencies of the VPCM based on the information it receives. The policy consists on a simple dual-state machine that monitors at run-time if the temperature of each MPSoC component increases/decreases above/below two certain thresholds previously defined (350 or 340 Kelvin in this example) and, then, selects the system frequency (500 or 100 MHz) accordingly. Whenever any of the monitored modules (the thermal controller reads the current temperature from the temperature sensors) exceeds the 350 Kelvin, the frequency of the system is set to 100 MHz; once all the modules return to a safe temperature (below 340 Kelvin), the frequency is restored to 500 MHz.

The results obtained employing the VPCM module with DFS are included in Figure 5.3 (trace *Emulation with DFS*), and indicate that this simple thermal management policy could be highly beneficial in MPSoC designs using low-cost packaging solutions (i.e., with values of package-to-air resistance of more than 40K/W). Furthermore, these results outline the potential benefits of this HW/SW emulation tool to explore the design space of



(a) scattered in the corners (b) clustered together in the center

Figure 5.4: Alternative MPSoC floorplans with the cores in different positions.

complex thermal management policies in MPSoCs, compared to SW cycle-accurate simulators that suffer from important speed limits.

5.1.4. Exploring different floorplan solutions

After deciding which MPSoC components to use, and how to interconnect them, there are still many design decisions to take that affect the system performance. One of them is the placement of the different elements; a technique called *thermal-aware floorplanning* [HVE+07], for example, aims at reducing the system hotspots by placing strategically the system components. In this section, I use the EP to evaluate three different floorplans for the initial case study, with four processing cores and NoC-based interconnect working at 500 MHz.

The original floorplan is depicted in Figure 5.1b. The first alternative floorplan scatters the processing cores in the corners of the chip (Figure 5.4a) while, in the second one, all the cores are clustered together in the center of the chip (Figure 5.4b). I assumed the use of a low-cost packaging solution in all the cases (see parameter *package-to-air conductance* in Table 5.1).

Regarding the configuration of the *Thermal Model*, I used the same thermal cells (dimensions and size), but changed their location on the floorplan.

The results are shown in Figure 5.5. In this case, we can observe that the best floorplan to minimize temperature (15 % less heating speed on average than the initial floorplan of Figure 5.1b) was achieved with the placement

technique that tries to assign the processing cores to the corners of the layout (labelled as *Scattered* in Figure 5.5). Hence, this solution is the best out of the three placement options because it delays the most the need to apply the available DFS mechanism, although its interconnects experience more heating effects due to the longer and more conflicting connection paths between components, that originate more NoC congestion effects. Then, the solution that tries to place all the processing cores in the center of the chip (labelled as *Clustered* in Figure 5.5) shows the worst thermal behaviour, but just slightly worst in temperature (5 % on average) than the original manual placement of cores used for this MPSoC design, while the delays in the interconnections between cores are minimal for the former due to their closest locations in the floorplan (see Figure 5.4b).

The main conclusion from this study is that a more aggresive temperature-aware placement must be applied (e.g., placement of cores scattered in the corners of the chip) to justify the placement of cores apart, as tried in the original manual design, to compensate for the heating effects due to longer interconnects. Otherwise, the possible penalty for long interconnects may not be justified in the end since a uniform distribution of power sources does not need to lead to a uniform temperature in the die. Moreover, these results clearly outline the importance for designers of tools to explore the concrete thermal behaviour of each design, and to select the most appropriate placement at an early stage of the integration flow, in order to facilitate a better diffusion of heat and minimize the risk of hotspots.

5.1.5. Exploring different packaging technologies

The EP can also be used to test the thermal behaviour of different packaging solutions for a given MPSoC, so that designers can quickly get the different thermal profiles, and decide which solution to adopt. Using the same MPSoC with four RISC-32 processing cores working at 500 MHz and NoC interconnect (see floorplan at Figure 5.1b), and the same setup as in the previous sections (sniffers, thermal cells, etc...), I simulated and compared the thermal behaviour of three packaging technologies; the low-cost value of 45K/W, higher than the initial value considered (Table 5.1), and two additional smalller values, namely, 12K/W in the case of standard packaging [ARM04b] and 5K/W in the case of high-cost and high-performance embedded processors [AMD04] (see Table 5.3).

The results of this experiment are synthesized in Figure 5.6, that shows the thermal behaviour of the MPSoC along time:

In the case of the standard packaging solution, the MPSoC design required more time to heat up and it reached a maximum value of 360 Kelvin when the DFS mechanism was not applied, which is lower than the case of low-cost packaging (45K/W) that reached a temperature of more than 500 Kelvin. However, when the presented threshold-based DTM strategy of Sec-

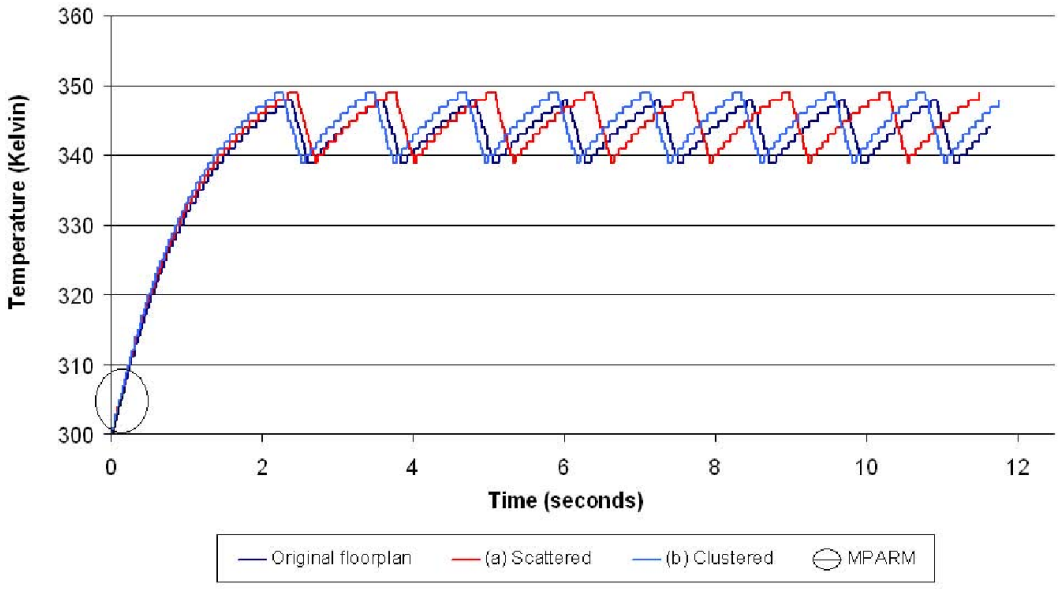


Figure 5.5: Average temperature evolution with different floorplans for Matrix-TM at 500 MHz with DFS on.

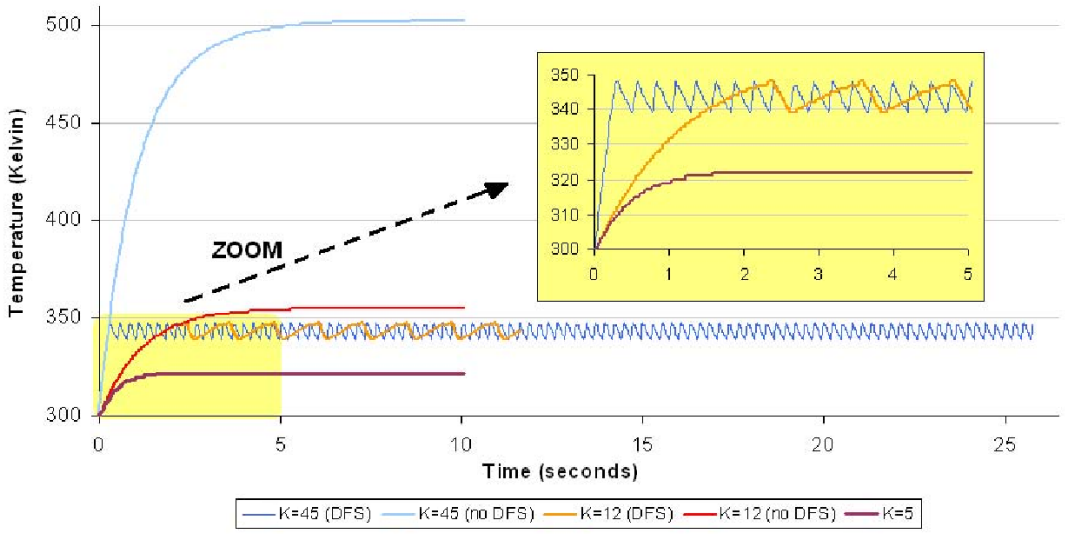


Figure 5.6: Thermal behaviour using low-cost, standard and high-cost packaging solutions.

Table 5.3: Three packaging alternatives for embedded MPSoCs.

Package solution	Conductivity (package-to-air)
Low-cost	$45K/W$
Standard	$12K/W$
High performance	$5K/W$

tion 5.1.3, fixed at 350 and 340 Kelvin, was applied, the thermal behaviour of the standard packaging system was similar to the low-cost solution (only its starting point was slightly shifted to the right due to the less steep temperature rise curve). Therefore, in this case, with this threshold value, no significant improvements were obtained with the standard package, and the low-cost solution would be preferably selected for this design using DTM. However, in the case of the high-cost packaging solution (for 5K/W), the system showed a completely different temperature behaviour, where the chip never went beyond 325 Kelvin; Therefore, this packaging solution creates a much lower thermal stress in the overall MPSoC implementation, and it does not require the application of DFS because the design never reaches a temperature above the 350-Kelvin threshold. As a result, this solution could significantly increase the expected mean-time-to-failure of the components and be interesting in highly reliable versions of the chip. Nevertheless, note that this type of package has the important drawback of the high cost for the manufacturer of the final embedded system, typically 5 to 12× more than standard package solutions and more than 20× the low-cost package solution [IBM06]; Thus, it can seriously increase the price of the final product and developers would like to avoid it, if possible.

The final conclusion is that this type of experiments and the presented framework can be a very powerful tool for designers to decide which type of packaging technique would be enough for a specific set of constraints in forthcoming generations of MPSoC designs.

5.2. Reliability exploration framework

In this section, I introduce a reliability enhancement policy aimed at extending the lifespan of a processor by reducing the stress induced in the register file. It is SW-based, and only implies modifying the compiler; thus, it can be applied to a broad range of processors. In this particular case, it has been implemented on the IEEE-1754 Leon3 Sparc v8 processor core [Gaib], and validated using the EP. To this end, I have added my HW/SW thermal-reliability infrastructure around a Leon3 system.

In the first place, I describe the Leon3 and its microarchitecture, putting special emphasis in the register file. Next, I describe the setup of the EP

to perform reliability analysis of the Leon3 system. The following step is to obtain and study an initial reliability profile. Finally, a new register allocation policy is proposed, implemented, and tested in the Leon3, through emulation, to prove that it efectively reduces the MTTF of the register file.

5.2.1. The Leon3 processor

The Leon3 processing core is a 32-bit CPU based on the Sparc-V8 RISC architecture and instruction set, conceived as a fully customizable microprocessor, and designed primarily for embedded systems applications. A synthesizable version of the Leon3 has been developed by Aeroflex Gaisler Research [Gaib]; it includes the core, the peripherals, and the toolchain to generate, download and debug both the SW and the HW. The complete source code is publicly available under the GNU GPL license (directly from Gaisler's website [Gaia]), allowing free and unlimited use for research and education. This fact converts the Leon into a perfect candidate for microarquitectural studies.

The version of the Leon3 available from Gaisler contains multiple features common to those found commercially. Moreover, it is highly configurable, and particularly suitable for SoC designs. The main features include separate instruction and data caches, a HW multiplier and divider, a memory management unit (MMU), separate (or combined) instruction and data translation lookaside buffers (TLBs), and the system has the potential to be extended to a multicore configuration (Figure 5.7 shows an example architecture featuring four Leon3 cores). Each Leon3 core supports a large range of customizations (e.g., size or replacement policy of the register file, caches, and TLBs), which allows the designer to specify the concrete system architecture to test.

5.2.1.1. The register file

The Sparc microarchitecture uses a special type of register file, based on register windows, that facilitates the sharing of data between procedure calls. This mechanism makes 32 general purpose integer registers visible to the program at any given time but, internally, it keeps several sets of registers for the different parts of the program, reducing the need to load/save them from/to memory. Of these, 8 registers are global registers, and 24 registers belong to the current register window. The structure of the register windows is specified by the Sparc v8 standards [Inc] and contains 8 *local* registers, 8 *in* registers, and 8 *out* registers. A Sparc implementation can have from 2 to 32 windows; thus, the number of registers varies from 40 to 520.

To provide communication between the register windows, the *in* and *out* registers are shared between the previous and next register windows, respectively, with the local registers being exclusive to the currently selected

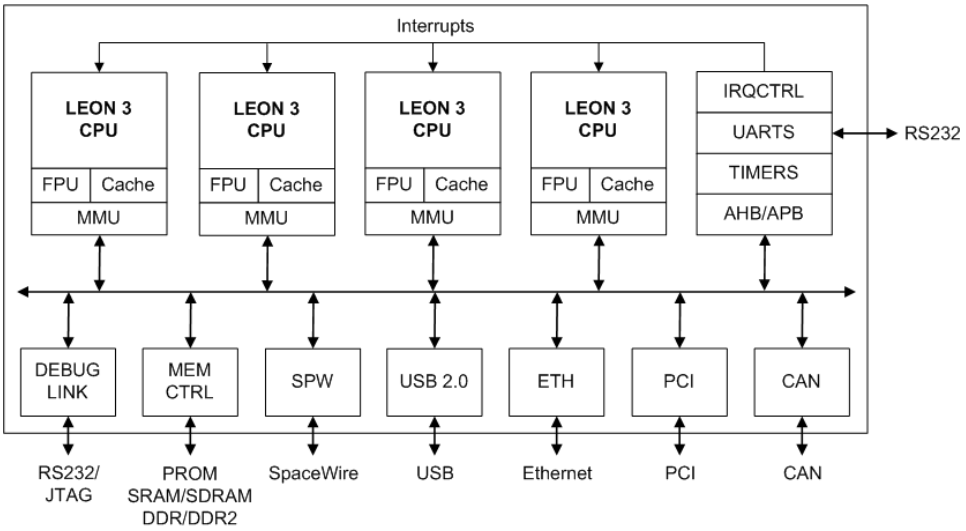


Figure 5.7: Multicore Leon3 architecture.

register window. Figure 5.8 represents graphically this overlapping of the register windows: Every time a new procedure is called, the register window is shifted upwards; once the procedure finishes, it comes back to the previous state (i.e., it is shifted downwards).

5.2.2. The Leon3 emulation platform

Figure 5.9 shows the block diagram of the reliability emulation framework created to study the Leon3 register file.

The emulated architecture (left side of Figure 5.9) contains one Leon3 core with a 3-port register file of 256 registers (with 8 register windows), has a SDRAM memory controller, 16Kb 4-way set associative instruction and data caches, and separate instruction and data TLB's, each containing 32 entries. The replacement policy is set to LRU. Furthermore, the Leon3 system includes 64KB of on-chip ROM and RAM (not shown), 512MB DDR Memory, AMBA buses, timers, and interrupt controllers. Finally, the communication interface to load applications is provided through a serial UART (RS232) port.

Physically, the specific layout of the register file considered in this case study is depicted in Figure 5.10: It contains 256 registers arranged into 32 rows and 8 columns; and each register features two read ports and one write port, with each port having separate address and data buses.

The *Statistics Extraction Subsystem* from Section 2.2.2 has been instantiated and particularized with the necessary components to control and monitor the emulated Leon3 system. Its main component is the *HW Sniffer* used to snoop signals within the Leon. In this case, I have included sepa-

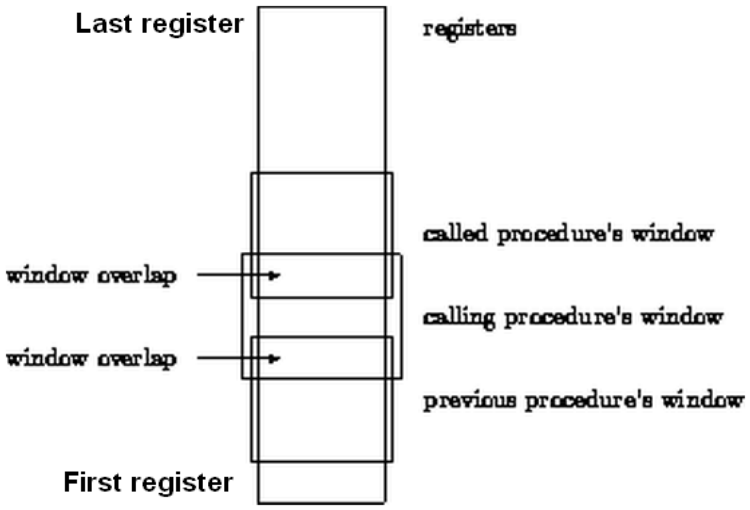


Figure 5.8: Leon3 register windows.

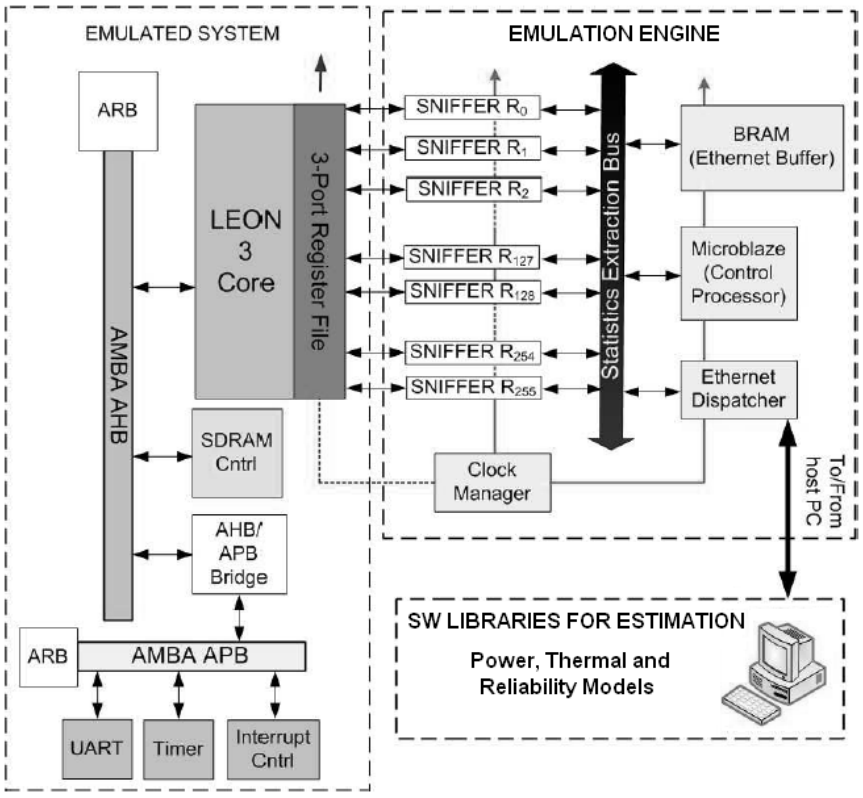


Figure 5.9: Overview of the reliability emulation framework used to monitor the Leon3 register file.

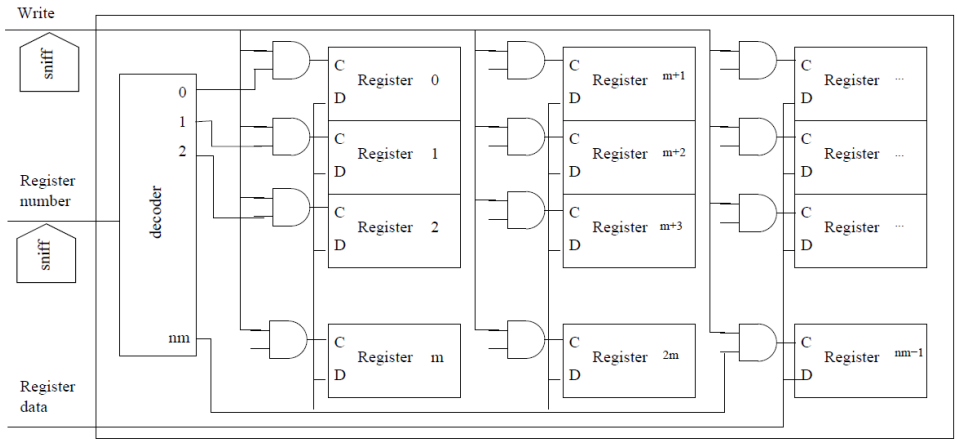


Figure 5.10: Layout considered for the Leon3 register file (256 registers, arranged in 32 rows and 8 columns).

rate monitors (sniffers) for each register of the register file, as shown in the top-right side of Figure 5.9.

5.2.3. Case study

The register file reliability emulation platform described in the previous sections has been used to perform a complete reliability analysis of the register file of the Leon3 core. In this analysis, I have explored the effects of the application domain, as well as the code transformations regulated by the compiler. Then, as an example of the potential benefits of reliability-aware design for nanoscale MPSoCs, using the outcome from this analysis, I have redefined the register assignment policy in the compiler to enhance the MTTF of the register file.

Regarding the setup of the *SW Libraries for Estimation*, the register file is modeled as implemented with the 90 nm process technology, with 256 cells, one per register (thus, arranged in the same 32 $\times$ 8 grid layout). The dimensions of each cell (register) are 300μm $\times$ 300 μm, and the thermal characteristics of the materials are those depicted in Table 3.4. In order to analyze the worst case scenario, the RF is sorrounded by cells with a constant temperature close to the hotspot (considered to be at 328 Kelvin); this is 318 Kelvin. Outside these cells it is the ambient environment.

With respect to the SW running on the Leon3 processor, a set of embedded applications from MiBench [GRE+01] and CommBench [WF00] suites has been selected to analyze the effects that the application domain has on the reliability. Among these applications, data-processing (*FFT*, *reed*), mathematical and graph theory (*basicmath*, *dijkstra*) and ordering/searching (*bitcount*, *qsort*, *stringsearch*, etc.) algorithms can be found. These applica-

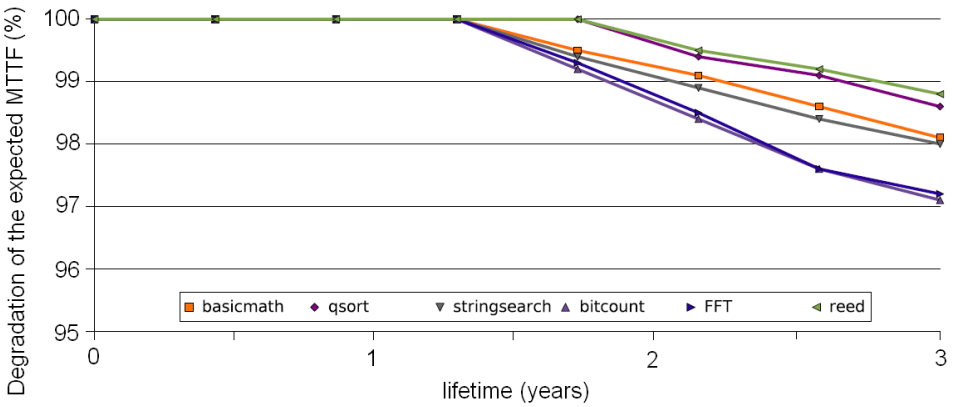


Figure 5.11: Evolution of the MTTF degradation along 3 years for various benchmarks.

tions have been compiled with a cross-generated version of GCC 3.2.3 for the Sparc architecture. Also, four versions of each benchmark have been generated using the four optimization levels of GCC (-O0 to -O3).

5.2.3.1. Reliability emulation

The first set of experiments studies the effect of the target application on the MTTF of the register file. The results are synthesized, in Figure 5.11, as the evolution of the degradation of the expected MTTF (see Section 3.4) along 3 years of operation. The main conclusion is that, independently from the application domain, the key differentiator used to identify the worst benchmarks, from the reliability viewpoint, is the analysis of which ones make intensive use of a reduced number of registers, namely *FFT* and *bitcount*; those are the benchmarks that experience the most severe MTTF reduction (up to 2.9 % in 3 years, following the normalized pattern of Figure 5.11), due to the hotspots that appear in the highly-accessed registers. On the other hand, those data-processing benchmarks with an extended number of assigned registers (i.e., *qsort* and *reed*) experience a lower impact on the MTTF prediction.

The second set of experiments evaluates the effect of the different compiler optimizations, from -O0 to -O3:

As Figure 5.12 shows, the less optimized policy (-O0 option) is the one that provides a lower impact on the MTTF reduction (1.5 %), while the register reuse conducted by the most extensive compiler optimization options impacts the MTTF negatively (2.5 % and 3 % for the -O2 and -O3 options, respectively, in the sampled interval). The last trace of the figure, *MODIFIED*, is explained in the next section.

Another graph, Figure 5.13, gives us an insight into the four main relia-

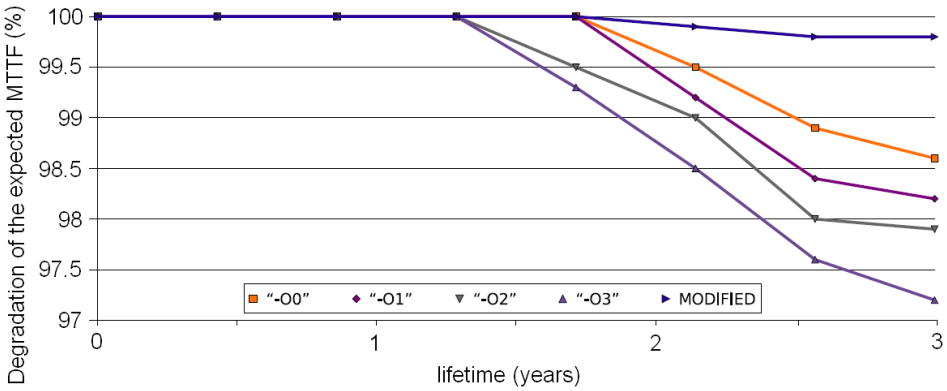


Figure 5.12: Evolution of the MTTF degradation for the *FFT* benchmark under different compiler optimizations.

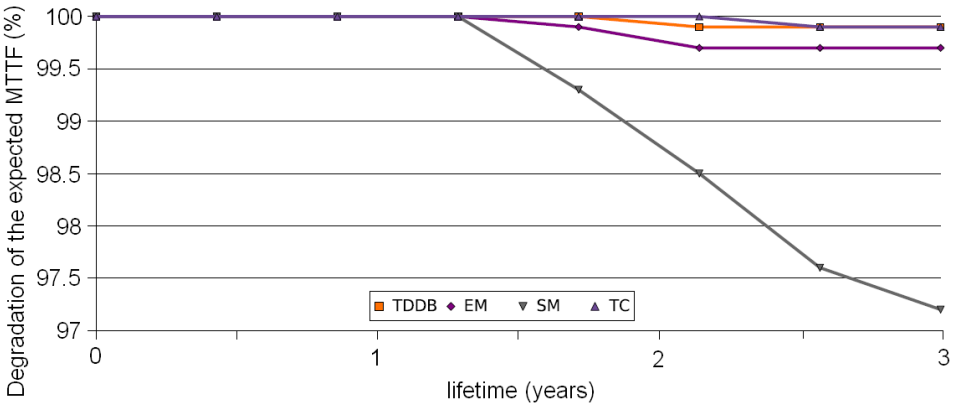
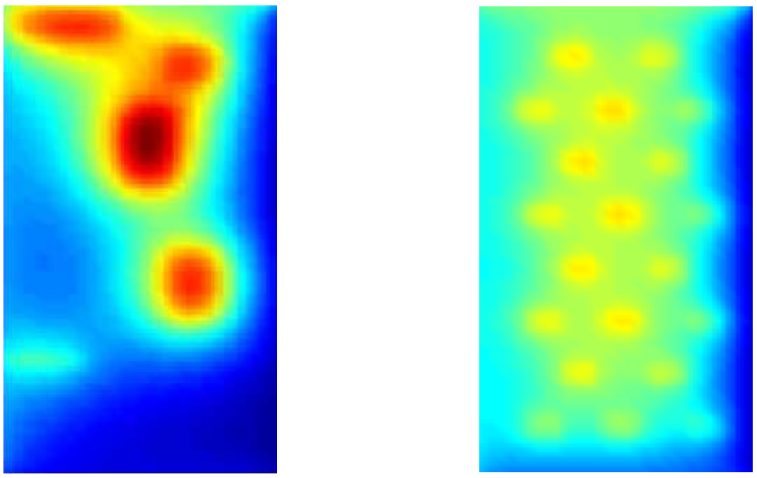


Figure 5.13: Contribution of the four main reliability factors to the degradation of the expected MTTF for the *FFT* benchmark compiled with -O3.

bility factors that contribute to the degradation of the MTTF of the *FFT* benchmark under the -O3 optimization; as predicted by the different thermal models for sub-micron technologies [ADVP+08; CSM+06; SSS+04], SM is the dominant factor in the reduction of the MTTF due to the fast thermal dynamism of the system in different execution phases (i.e., 12°C difference can occur in few seconds).

Finally, I have estimated the number of damaged registers as a way to quantify the degree of device failure: A register is considered to be damaged if its MTTF is below 2 % of the nominal value. This information is very useful, for the microarchitecture designer, to understand the consequences of the optimization policies applied by the compiler in the register file lifetime. The number of damaged registers, at the end of a sample interval of 2 years, for the *bitcount* benchmark, one case study with high pressure in the register



(a) Traditional (b) Modified

Figure 5.14: Thermal distribution of the register file of the Leon3 core using different register allocation policies.

file, is depicted in Figure 5.15. On average, it varies between 1 and 4 for the studied interval, depending on the optimization level used by the compiler. In the worst case, code compiled with the (-O3) option, the probability of having at least 4 registers damaged in the first 2 years reaches 99.5 %, making critical the development of reliability-aware register assignment policies. The last trace of the figure, *MODIFIED*, is explained in the next section.

5.2.3.2. Reliability enhancement policy

Using the register file information obtained from the reliability emulation framework in the previous section, I have modified the register assignment policy made by the GCC compiler with the goal to reduce the hotspots:

The algorithms included in the current versions of GCC [bib03] assign registers from a pool of free registers. My proposed register allocation technique, called *MODIFIED*, selects the target register after checking that the neighbours have not been previously assigned, if possible. In order to implement it, I modified the graph coloring algorithm found in [JYC00]. This pattern of assigning registers results in a thermal map that resembles a chess board, as we can observe in Figure 5.14b. Compared to the original register allocation policy (Figure 5.14a), *MODIFIED* facilitates a better diffusion of heat within the different register windows and a broader selection of registers that, eventually, reduce the hotspots and improve the reliability of the register file.

As depicted in Figure 5.15, my new register assignment policy (*MODIFIED*) reduces the number of damaged registers. In fact, the spreading of the

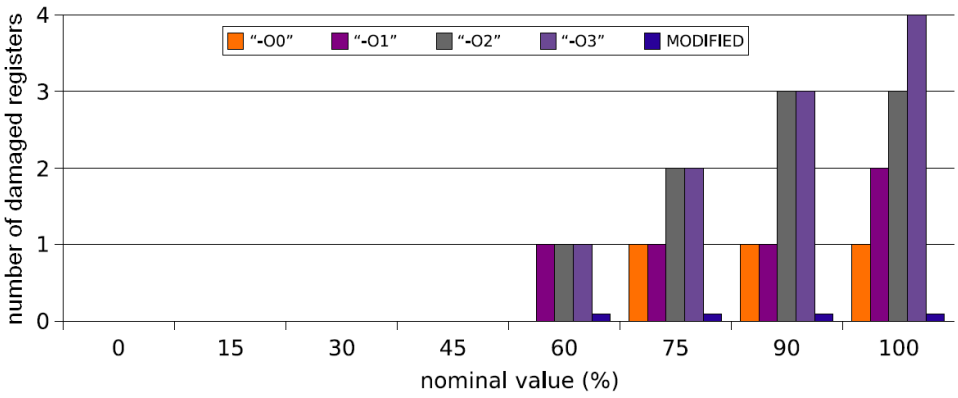


Figure 5.15: Number of damaged registers, after 2 years, for the *bitcount* benchmark, under different compiler optimizations, and using my reliability-aware algorithm (*MODIFIED*).

register assignment per window performed by this policy eliminates any damaged register in the sampled interval (2 years) for the *bitcount* benchmark. Moreover, Figure 5.12 indicates that this policy is very effective to minimize the MTTF degradation: it is only reduced by 0.2 % in the sampled interval, much smaller figures than any other policy. In fact, in comparison with -O3 (Figure 5.13), these results indicate that my policy reduces significantly (by 20 % on average) the impact of all factors related to MTTF degradation.

5.3. System-level HW/SW thermal management policies

Multi-Processor Systems-On-Chip (MPSoCs) are a design solution that succesfully provides the performance levels required by high-end embedded applications, while respecting the demanding design constraints (power consumption, reliability, etc.) of the embedded HW. The conception of a new MPSoC involves not only the design of a HW architecture, but also the development of the SW architecture that exploits it. Section 5.2 already introduced the importance of the SW in the thermal behaviour of a mono-processor SoC, experimenting with compiler modifications on a Leon3-based system running C applications. When several microprocessors come into play, multi-processor operating systems (MPOSes) and middleware are required to efficiently exploit the interaction of the various components of the underlying HW, while ensuring flexibility and providing a standard HW-abstraction layer for heterogenous application development.

While this layered approach eases the programmer's job, SW and HW designers have the responsibility of efficiently managing non-functional system

constraints, such as power and temperature. The high HW and SW complexity provides high degree of freedom at the price of increased design effort at SW (OS) and middleware level. Hence, mechanisms to efficiently evaluate the effectiveness of advanced thermal-aware OS strategies (e.g. task migration, task scheduling policies) onto the available MPSoC HW are needed.

In this context, I have enhanced the flexible HW/SW FPGA-based emulation infrastructure presented in Chapter 2, with the necessary HW and SW extensions (only the *Emulated System* is affected) to support MPOSes and middleware emulation, and enable the exploration of OS-level thermal management policies.

The following sections are organized as follows: In Section 5.3.1, I present the architectural extensions to MPSoC designs to provide an efficient implementation of MPOSes: First, describing the changes at the HW level and, then, introducing the foundations of the ported MPOS to enable a complete framework to explore thermal-aware OS-level strategies. Next, in Section 5.3.2, I detail the complete MPOS MPSoC emulation flow, that has incorporated minor changes. Finally, in Section 5.3.3, I present a real-life example, aimed at developing a system thermal balancing policy. The results prove the benefits of advanced temperature management using task migration.

5.3.1. The multi-processor operating system MPSoC architecture

Figure 5.16 depicts the HW architecture of the multi-processor operating system emulation framework with thermal feedback. The *Emulated System* is composed of a variable number of soft-cores (MB0..MB3, in the figure). Each core runs its own instance of the uClinux OS [url06] on a private memory, physically mapped into the available off-chip DDR memory on the board, for space reasons (the included on-chip BRAM memories of the FPGA are too small for containing the OS image). A shared memory, also mapped into the external DDR memory, is used by a middleware layer running on top of each OS to add communication and sychronization capabilities (such as process synchronization, resource management, and tasks scheduling) among the OSes.

The *Emulation Engine* presents no modifications with respect to the standard one, presented in Chapter 2. However, it is worth mentioning that the thermal sensors, described in Section 2.1.2.1, are now mapped into the memory range of the processors.

The fact that we are now emulating a MPOS is completely transparent for the *SW Libraries for Estimation*, that interact only with the HW: they receive the system statistics from the activity of the HW cores, and write the output temperatures in the HW sensors. At the SW level, the middleware

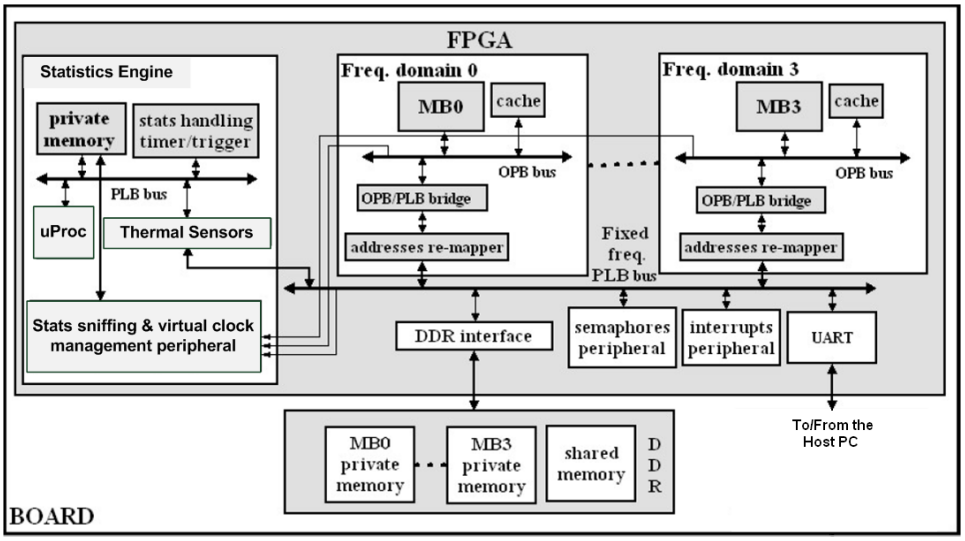


Figure 5.16: Overview of the HW architecture of the multi-processor operating system emulation framework with thermal feedback.

has access to these sensors, so that it can use the computed temperature values to implement a thermal-aware task migration strategy. An example policy that exploits temperature feedback is described in the experimental results section (Section 5.3.3).

5.3.1.1. MPOS HW: Architectural extensions

Any MPOS requires some basic HW support (not included in the basic blocks of the baseline architecture described in Chapter 2) to enable the inter-processor communications and, at the same time, garantee the exclusive access to the shared elements. I solved these two issues implementing a special inter-processor interrupt controller and a semaphore memory. In addition to these two modules, I created a HW address translator to facilitate the porting of the OS, and a UART multiplexer to simplify the control of the processors. Finally, I added a frequency scaler that can be SW-controlled. I next describe each of these elements, implemented in VHDL; three of them were designed from scratch while, the other two, the interrupt controller and the address translator, were implemented modifying the source code of the *opb interrupt controller*, and *opb v20 bus*, respectively, two modules included in the EDK pcores library (from Xilinx):

1. **Inter-processor interrupt controller:** This component is needed to enable interrupt-based wake up of tasks sleeping while waiting for a shared resource to be freed. Without interrupt support, a task is forced to perform busy waiting on shared variables for accessing shared

data, such as messages from tasks in other processors. With the inter-processor interrupt controller, any processor can generate an interrupt in the selected target processor by writing a word in a memory-mapped control register.

2. **Semaphore memory (MUTEX):** Mutual exclusive access to the common resources (e.g., shared memory) is provided through a HW module, the mutex, that implements the *test-and-set-lock* (TSL) primitive [HP07], an atomic operation used, for instance, in the construction of semaphores.

   The mutex is mapped on a shared memory area, and contains a variable number (configurable from 1 to 1,024) of special memory positions, known as "locks", or semaphores. A lock can be acquired by any of the processors included in the emulated MPSoC. When the lock is free and a processor reads it, a "zero" value is read and, atomically, that memory position becomes a "one". The behaviour of the rest of read and write operations is as in a normal memory afterwards.

   A user-defined number of semaphores can be defined as variables into the shared memory region. Every processor should then periodically check its shared area for new incoming messages, which would result in extra bus traffic. However, in order to avoid this polling overhead, the mutex is able to monitor all accesses to the shared memory, and fire an interrupt for the corresponding processor only when new data are available.

   Semaphores are an effective mechanism to avoid the simultaneous use of a common resource, such as a global variable or a shared peripheral, where all the processors can deliver their messages.

3. **Address translator:** All the private memories of the processors are mapped into the same SDRAM. They lie in non-overlapping address ranges. Due to the absence of a Memory Management Unit, to avoid static linking of OS and program code at different locations, it is needed to provide to each core the same view of the private memory. This is obtained by translating the addresses generated by the cores to the appropriate memory range, so that all the processes can execute independently from the processor where they run. This operation is transparently performed by this HW module.

4. **Multiplexed UART:** A Terminal is a simple way for users to establish a bidirectional communication with Linux-based systems. It is a text window where, basically, the system prints messages with information, and the user inputs the commands. In embedded processors, without human IO devices, this interchange of information is normally performed through the serial port [Sta11]. Following this idea, the EP uses a

serial connection to communicate the host PC with the FPGA. In the PC, we receive the information coming from the emulated processors in independent instances of the Minicom [bibc] application that, at the same time, allow the user to input interactive commands. Figure 5.17 depicts this EP-user interface. There is one Terminal per processor in the *Emulated System*, plus two additional ones for debugging. For the sake of simplification, I implemented a module that multiplexes all the Terminal communications into one single serial connection that can, then, be mapped into one serial port. Thus, removing the need to add one extra port (and cable) per processor present in the *Emulated System*.

5. **Frequency scaling:** This module allows the SW to individually set the frequency of the different processors of the *Emulated System*. Programmable dividers have been combined with the platform clock generators to obtain SW-configurable frequency scaling support. Each core can set its own frequency, as well as the frequency of other cores, by accessing the memory locations where the described dividers are mapped.

5.3.1.2. MPOS SW: Inter-processor communication libraries

Each processor in the *Emulated System* runs its own instance of the uClinux OS. UClinux is a collection of Linux 2.x kernel releases intended for single microcontrollers without Memory Management Units (MMUs), as well as a collection of user applications and libraries. In this work, the standard uClinux distribution has been extended with a SW abstraction layer aimed to support inter-processor task migration. This layer also includes the SW drivers to access the HW modules described in the previous section: the interrupt controller, the semaphores, the multiplexed UART, and the frequency scaler. The address translator is a transparent element for the SW.

In the programming model I adopted, each task is represented as a process. This means (as opposed to multi-threaded programming) that each task has its own private address space, and that task communication has to be explicit, because shared variables between tasks are not allowed.

The SW abstraction layer is depicted in Fig. 5.18 as OS/middleware. It is based on three main components: (i) standalone OS (uClinux) for each processor, running in private memory; (ii) lightweight middleware layer providing synchronization and communication services; (iii) task migration support and dynamic resource management layer. Together, the base OS image plus the libraries and the basic filesystem take 1.44 MB.

Each task runs on a single OS at a time, and can transparently migrate from one OS to another. Data can be shared between tasks using explicit services given by the underlying middleware/OS, using one or both of the

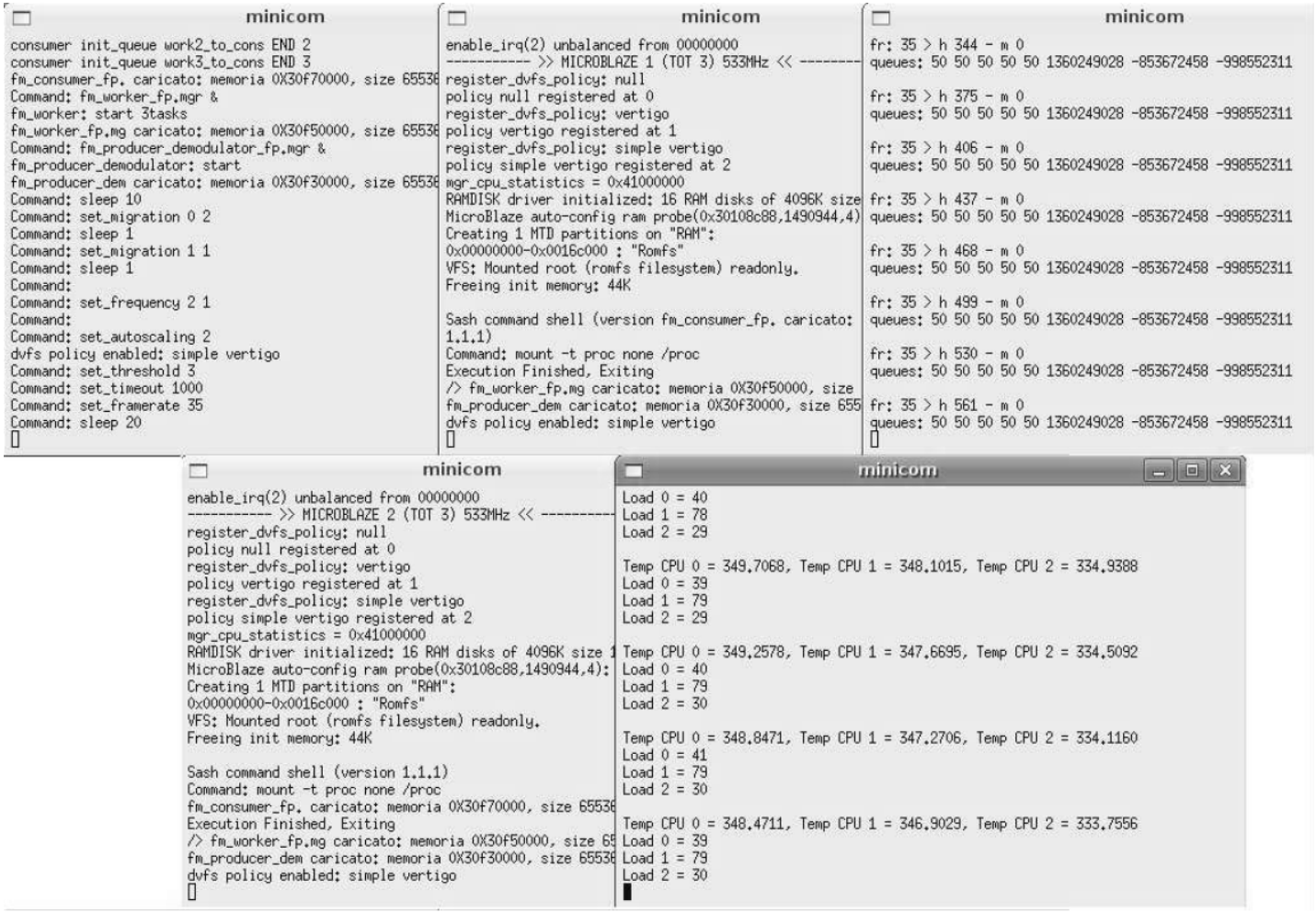


Figure 5.17: Multiplexed UART connections. From top-left to bottom-right: Minicom Core 1, Minicom Core 2, Minicom tasks queues, Minicom Core 3, Minicom miscellaneous information (temperatures, frequencies, and loads).

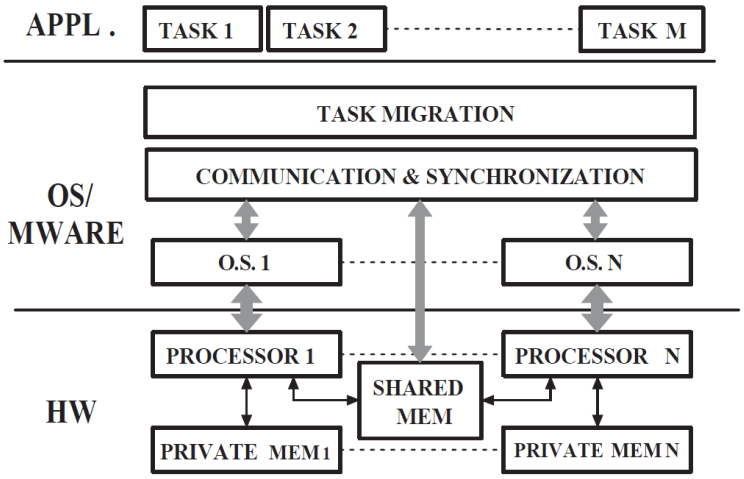


Figure 5.18: The software abstraction layers.

available communication models: *message passing* and *shared memory*. In addition to all this infrastructure, dedicated services run in the background to enable tasks synchronization: the *Communication and Synchronization Support*, the *Task migration Support* and the *Decision Engine*.

Communication and synchronization support

Using *message passing* paradigm [Sta11], when a process requests a service from another process (which is in a different address space), it creates a message describing its requirements, and sends it to the target address space. A process in the target address space receives the message, processes it, and services the request. I implemented a lightweight message passing scheme able to exploit both scratch-pad memories and shared memory to implement independent mailboxes for each processor core. It consists of a library of mixed user-level functions and system calls that each process can use to perform blocking write/read of messages in the data buffers. I defined a mailbox for each core and not for each task to avoid allocation/deallocation of mailboxes depending on process lifetime.

The second inter-processor communication method is the *shared memory* paradigm [Sta11], where two or more tasks are enabled to access the same memory segment. The call to "malloc" is replaced by a call to "shared malloc", that returns pointers to the same actual memory. When one task changes a shared memory location, all the other tasks see the modification. Allocation in shared memory is implemented using a parallel version of the Kingsley allocator [Sta11], commonly used in linux kernels.

Task and OS synchronization is supported providing basic primitives like binary or general semaphores [HP07]. Both spinlock and blocking versions of semaphores are provided. The spinlock semaphores are based on the HW *test-and-set-lock* memory-mapped peripherals, while the non-blocking semaphores also exploit HW inter-processor interrupts.

Task migration support

I define task migration as the ability of the MPOS to suspend the execution of a task running in one processor, and resume its execution in a different processor, preserving the state. Inside the MPOS, I consider two types of tasks: those that can and can not be migrated.

In order to enable task migration among processors, the data structure used by the OS to manage an application that can be migrated is replicated in each private OS. When an application is launched, a *Fork* System Call [Sta11] is performed for each task of the application on the local OS. However, only one processor at a time can run an instance of the task; in this processor, the task is executed normally while, in the other processors, the replicas are in a suspended tasks queue. In this way, tasks which can be migrated and task which are not enabled for migration can coexist transparently for the private OS. Not all the data structures of a task are replicated, just the Process Control Block (PCB) [Sta11], which is an array of pointers to the resources of the task and the local resources.

To simplify the process of migrating a task, I introduced an additional SW layer (the *Task Migration* layer, in Figure 5.18), that handles the data replication and keeps everything synchronized. It uses kernel daemons that run on the background, transparently to the user. With these helper daemons, a task migration can be triggered with the high-level command "migrate task T to processor P", that can be issued directly by the user (using a *Terminal*), from a script, or from an application.

Two kinds of kernel daemons, master and slave, exist. There is only one instance of the master daemon, that runs in the processor where the user launches or terminates the tasks. For simplification, there is only one master processor. Thus, when we launch a task in a slave processor, internally, it is created in the master processor and, then, migrated to the slave processor. It is an implementation decision transparent to the user, who only issues a "Create task T in processor P" command. On the other hand, there is one slave daemon running in each processor of the system (including the processor where the master daemon runs). The master daemon is directly interfaced to the *Decision Engine*, a mechanism (an autonomous application, or the user himself) that determines when and where the tasks are to be migrated. All the communications between master and slave daemons are implemented

using dedicated, interrupt-triggered mailboxes in shared memory.

The master daemon performs four operations:

1. The master periodically reads a data structure in shared memory, where each slave daemon writes the statistics related to its local processor, and provides it to the *Decision Engine* that, at run-time, processes these data and decides the task allocation, eventually issuing task migrations; i.e., implements the dynamic task allocation policy.

2. When a new task or an application (i.e., a set of tasks), is launched by the user, the master daemon communicates this information to the *Decision Engine* and sends a message to each slave communicating that the application should be initialized.

3. When the *Decision Engine* decides a task migration, it triggers the master daemon, which signals to the slave daemon of the processor source of the migration that the task X has to be migrated to the processor Y.

4. When the master receives the notification that an application finished, it forwards this information to the slave daemons, that deallocate the task; and to the *Decision Engine*, that updates its data structures.

The slave daemon performs four operations:

1. When a new migratable application is launched, each slave daemon forks an instance task for each task of the application. Each task is stopped at its initial checkpoint and it is put in the suspended tasks kernel queue. The memory for the process is not allocated yet.

2. It writes periodically in the dedicated data structure (in shared memory), the statistics related to its processor. They are the base for the actions of the *Decision Engine*.

3. When the master signals that a task has to be migrated from a source processor to a destination processor, it performs the following actions: i) it waits until the task to be migrated reaches a checkpoint, and puts it in the queue of the suspended tasks; ii) it copies the block of data of the task to the scratch-pad memory of the destination process (if it is available and if there is enough space) or to the shared memory; iii) it communicates to the slave daemon of the processor where the task must be moved that the data of the task are available in the scratch-pad or in the shared (a dedicated interrupt-based mailbox is used); iv) it deallocates the memory dedicated to the block of the migrated task, making it available for new tasks or for the kernel; v) it puts the migrated task PCB in the suspended tasks queue.

4. When the slave daemon of the processor source of the migration communicates an incoming task, the receiver (i.e., the slave daemon of the processor destination of the migration) allocates the memory for the data of the incoming task, and copies the data from the scratch-pad or from the shared memory to its private memory. Finally, it puts the PCB of the incoming task in the ready queue.

Decision Engine

The middleware provides real-time thermal information to the running uClinux. At any moment, an application can read the current temperatures of any of the system modules by simply calling the function *getTemperature(IdOfTheModule)*. Internally, this function accesses the memory locations where the sensors are mapped, and returns the value that was previously introduced by the *Thermal Model*. This function is used by the *Decision Engine* that continuously monitors the die temperature to dynamically adjust system operation. Therefore, the *Decision Engine* can be defined as a dynamic workload allocator that decides when a task must be migrated, and to which processor. It is a task implemented in the kernel of the compiled uClinux image, that runs on the processor where the master daemon runs.

Modifying the *Decision Engine*, the user can program his own migration policies, algorithms that will depend on the actual temperatures of the system, the workload of the processors, the past history, or even random heuristics.

5.3.2. MPOS MPSoC thermal emulation flow

Figure 5.19 represents the flow to emulate a custom MPOS MPSoC design. The only difference with respect to the baseline EP flow, described before in Chapter 4, is that the SW side (stripped parts in the figure) has been extended to include the MPOS support. The SW binaries are now generated using the uClinux toolchain [url06] that enables to include OS support in the same image that contains the application binaries to be executed; In fact, the binary file generated contains the OS kernel plus the filesystem with the application.

When the designer describes a MPSoC architecture, all the information related to the HW resources present in the system (included processing cores, additional I/O blocks, memory addresses, custom parameters, interrupt numbers...) is embedded into a configuration file (in my case, automatically generated by EDK) that, once fed into the uClinux toolchain, it allows to build a custom uClinux OS image, tailored to the current particular HW.

The OS setup is an interactive process where the user can customize the

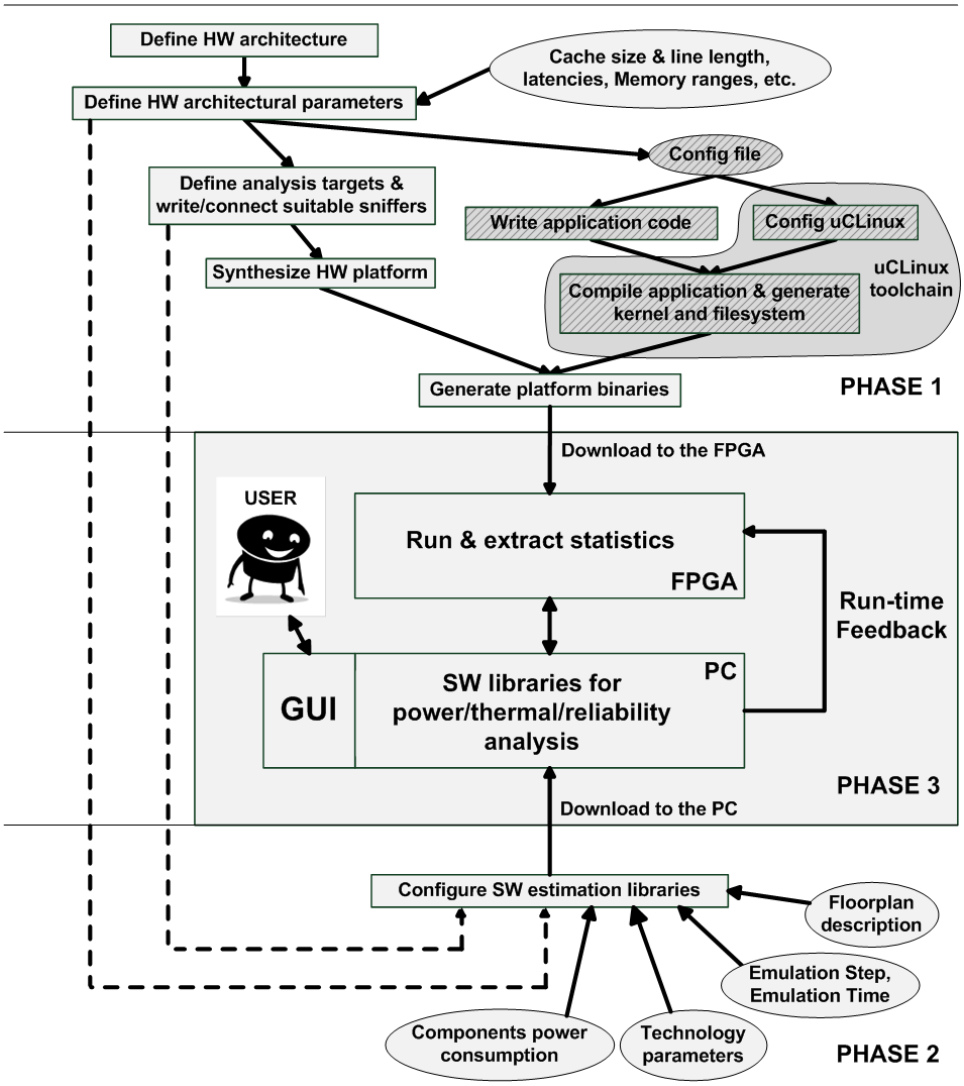


Figure 5.19: Complete HW/SW flow for the MPOS-enabled Emulation Platform.

kernel (e.g., choose the number of semaphores to use, enable/disable debugging support and thermal monitoring services, etc.) based on the available HW services (indicated in the configuration file). Provided this information, together with the available drivers for the included HW resources and the applications to run in the final MPSoC, a self-contained binary file is generated. It is not merged with the HW binaries; due to its size, the HW is first downloaded to the FPGA and, then, the SW is directly copied to the memories of the processors through the JTAG connection. After the uClinux images have been downloaded, the emulation starts.

During the emulation, the *Thermal Model* uses the statistics, collected by the sniffers, to compute the temperature of various chip components. It is, then, fed back into the thermal sensors inside the *Emulated System*, where it can be read by the OS and used to elaborate thermal management policies.

Overall, designers can use this framework to assess the impact of task migration and scheduling on system temperature, as well as to design thermal-aware policies at the OS level. The next section presents a practical example.

5.3.3. Case study

In order to assess the effectiveness of the enhanced MPOS MPSoC emulation framework, in this section I include a set of experiments to study the evolution of the temperature of an MPSoC architecture including 4 cores, when frequency scaling and task migration are available at the OS level to perform thermal management of the final chip.

5.3.3.1. Experimental setup

The considered floorplan is shown in Figure 5.20. It includes 4 ARM7 cores. Each one has a 64KB cacheable private memory, and there is a shared memory of 32KB. There are two independent caches (instruction and data) per processor, of 8KB each. The memories and processors are connected using an AMBA bus interconnect. The dimensions of the AMBA circuits were obtained by synthesizing and building a layout. The dimensions of the memories and processors are based on numbers provided by an industrial partner.

From the emulation point of view, the floorplan is divided into 128 regular thermal cells, and there is one sniffer per element present in Figure 5.20. The running frequencies and the workload of the processor are the activity monitored from the cores.

In the *Emulated System*, the clocks of the cores are generated by the frequency scaling module (see Section 5.3.1.1), that generates 10 frequencies equally distributed in the range of 10-51.2MHz. Analogously, in the final chip, the core frequencies will range between 100 and 512MHz. Since the emulation is ten times slower and, in the *Thermal Model*, I define the emulation slot as

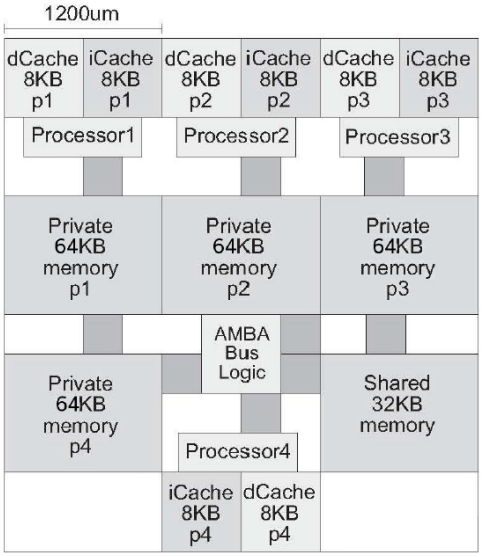


Figure 5.20: MPSoC floorplan with uneven distribution of cores on the die and shared bus interconnect.

10ns of emulated time, it means that we will effectively gather statistics every 100ns of real-life execution. The MPOS can dynamically set the frequency of the cores, at run-time, to effectively reduce power consumption as the workload of the MPSoC changes over time.

Regarding the SW side, I have defined a benchmark that stresses the processing power of the MPSoC design to observe effects in temperature. This benchmark implements a synthetic task that imposes a load near 100 %, and can be migrated from one core to another.

In the current example, we can include up to four cores, due to the size of the underlying Virtex-II Pro v2vp30 FPGA. However, the system can be scaled to any number of cores by using available larger FPGAs. Four processors are mapped into the system: MB0, MB1, MB2 and MB3, but the experiments are run only using the first three processors, for it results in more clear images.

In the first image (see Figure 5.21, where MB0 is the processor 1 of the floorplan, MB1 is the processor 2, and so on), it can be observed the thermal behaviour of the processors when a task is being executed only in one of them. The other two are idle. The OS in each processor automatically adjusts the frequency of the core using a policy based on the processor load observed over time intervals [FM02]. As expected, the frequencies of the idle processors are lowered to the minimum (100 MHz) and, after a brief delay, their temperature stops increasing (it even drops a bit) and remains stable, while temperature in MB0 keeps on going up until the limit imposed by the physical properties of the chip (around 360 Kelvin). In the figure, we

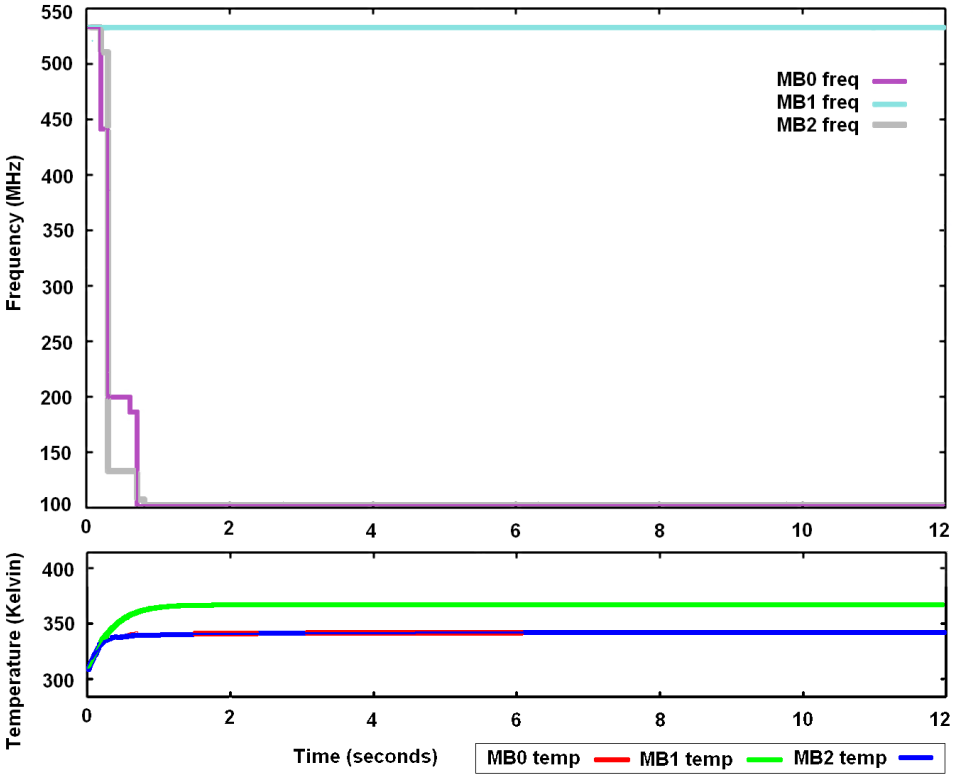


Figure 5.21: Temperature-frequency waveform with one task running on MB0.

appreciate how the temperatures of the idle processors are affected by MB0; however, being all the processors unloaded, they stay below 340K.

The second depicted image (see Figure 5.22), is more interesting from a practical point of view. It shows a more reasonable approach for a real situation where noone wants only one processor running at the highest frequency all the time; instead, the synthetic task running on the MB1 can now be migrated among the available cores. A simple rotational policy is applied: the owner of the task is periodically shifted, from MB1 to MB0, to MB2, and again to MB1, whenever the temperature surpasses a given threshold.

The middleware system is periodically monitoring the processor temperatures and comparing them with the predefined threshold, that I set to be 365 Kelvin in this experiment.

The curves in Figure 5.22 show temperature and frequency waveforms of each core over time: When the temperature of MB1 reaches the threshold, the middleware system triggers the task migration to the colder processor MB2; as a consequence, the temperature of MB1 starts decreasing and, in parallel, the temperature of MB2 starts increasing; when the later one reaches the

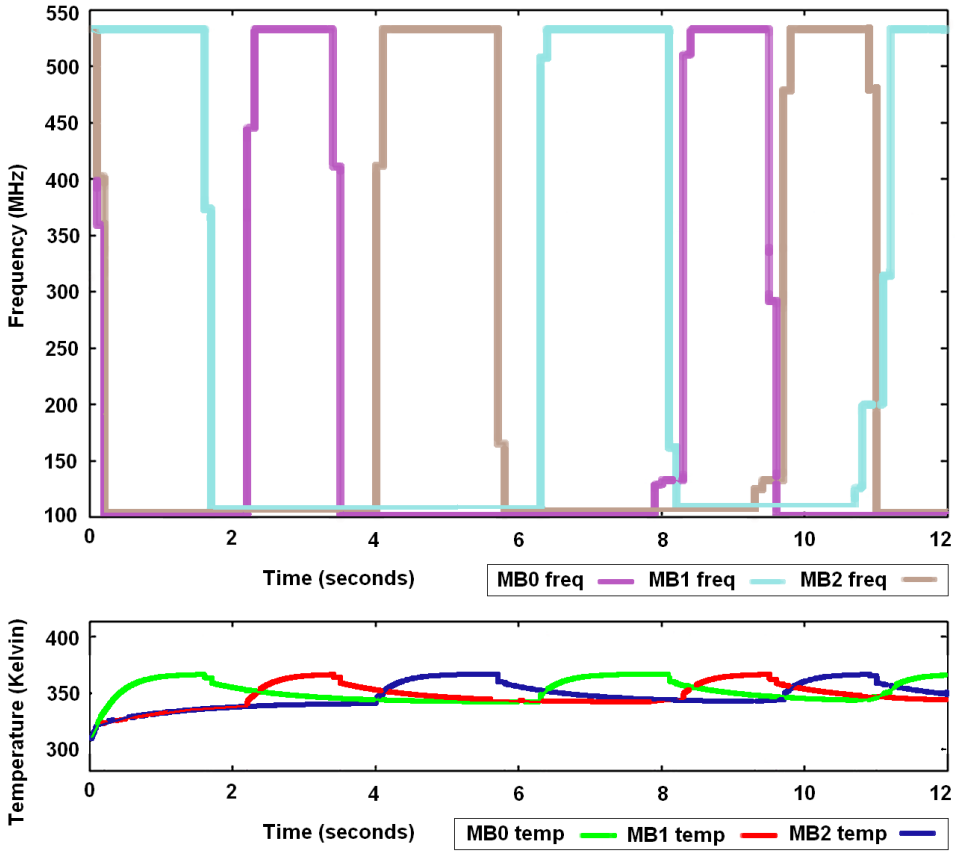


Figure 5.22: Temperature effect of a simple temperature-aware task migration policy.

threshold, it triggers another task migration to MB3.

From this simple experiment, we can draw out several interesting consequences of MPSoC temperature management:

- The temperature of each core is affected by the others but strongly depends on the load, which can be efficiently monitored by the OS since this layer has full knowledge of the task being executed and, even more importantly, which are the following tasks that need to be executed. Hence, the OS can define a proper task migration policy according to possible prior (design) knowledge of the location of the cores in the floorplan and the thermal conductivity between their cells.

- Thermal time constants are larger with respect to task migration delays; this is the necessary condition for task migration to be effective in controlling the temperature of the cores. However, task migration imposes an overhead due to data exchange between processors and to task shut-off and resume delays (a technique to reduce this overhead could be, for instance, to limite the number of migrations per time unit). As the results indicate, since temperature variations are slow with respect to the implemented migration overhead, moving tasks between processors is a viable technique to keep the temperature of this chip controlled.

- Regarding exploration efficiency, the duration of both experiments was 90 seconds for 6 seconds of real-time, which indicates more than 1000× speed-up with respect to cycle-accurate MPSoC simulators including OS [PMPB06]. Emulation time depends on two contributions: i) the *Emulated System* is ten times slower than the final system; ii) there is an additional time overhead to synchronize the FPGA and the PC. Overall, the performance of the emulation is efficient enough for very fast system prototyping and MPOS thermal policies validation.

5.4. Conclusions

In the first set of experiments, I have shown the benefits of the EP to perform detailed exploration of the thermal characteristics of a chip under design:

I have demonstrated that the proposed HW/SW framework obtains detailed cycle-accurate reports of the thermal features of final MPSoC floorplans, with speed-ups of three orders of magnitude compared to cycle-accurate MPSoC simulators. Also, the addition of more processing cores and more complex memory architectures in the emulation framework suitably scales; Thus, almost no loss in emulation speed occurs (conversely to cycle-accurate simulators), which enables long simulations of complex MPSoCs as thermal

modeling requires. Next, I have introduced a simple DFS mechanism in order to illustrate the flexibility of the proposed HW/SW FPGA-based framework to explore, in real-time, temperature-management policies.

In the next experiment, I have used the EP to evaluate a thermal-aware placement technique that tries to compensate the heating effects on MPSoCs by changing the location of the hot cores [HVE+07]. This study indicates that, in addition to the *a priori* benefits of separating the hot cores, significant overheads of power dissipated in long interconnects can clearly affect the overall thermal behaviour of the final MPSoC, and that a uniform distribution of power sources in the die does not need to produce a uniform temperature in the final chip. Hence, MPSoCs designed in latest technology nodes require the use of tools to study their suitable placement at an early stage of system integration, according to the applications that will be executed in the final system.

Finally, I have illustrated the effectiveness of the EP to rapidly study the effects of different packaging options for concrete MPSoC solutions. The results indicate that the selection of the final packaging solution clearly depends on the thermal management techniques included in the target MPSoCs, and that more costly packagings may show from the same heating effects as low-cost ones; Thus, the need of expensive packaging solutions cannot be justified without prior extensive thermal exploration.

In the second set of experiments (Section 3.4), I have illustrated the feasibility and benefits of reliability-aware design by performing a complete reliability analysis of the register file architecture of a Leon3 processor. Since this type of analysis is very time-consuming for pure-SW simulators, I have applied my HW/SW emulation framework, which enables an exhaustive exploration of the various reliability factors for a complete range of different benchmarks.

The obtained results outline that, on the one hand, the target application domain can have a very negative impact on the reliability of the register file, as well as the use of aggressive compiler optimizations. However, on the other hand, effective reliability-aware register assignment algorithms can significantly enhance the MTTF of the register file (up to 20 %, on average) for different kinds of applications.

Additionally, the complexity of this system serves as an example of the scalability of the EP. If we compare this experiment with the one in Section 5.1, the target processor is a Leon3 instead of a simple 32-bit RISC core, and the thermal analysis is performed at a greater level of granularity (microarchitectural).

Finally, in the third set of experiments (Section 5.3.1), I have presented an extension to the original emulation framework: Inside the *Emulated System*, I have included the necessary architectural support, at the HW level, to implement a MPOS based on the uCLinux distribution; on top of which, I have

added inter-processor communication and task migration capabilities. The resulting framework enables long thermal emulations of MPSoC architectures running a MPOS with task migration support. This enhanced version of the EP is used to explore the benefits of thermal-aware management at the OS level in MPSoC designs, that allows from simple control of the rise of temperature in the die, to the definition of advanced thermal-aware MPOS strategies.

Overall, in this chapter I have presented several case studies that demonstrate the flexibility and usefulness of the EP at different stages of the MPSoC development cycle. Designers can use it to evaluate both the HW and SW modifications: from the impact of changing the register file layout at the microarchitectural level, to the importance of the tasks scheduling policies implemented in the MPOS kernel.

Before going to fabrication, with the EP, we get realistic statistics of the final chip running the real (i.e.: final) SW applications, as well as an early estimation of the power, temperature and reliability values, that help designers to choose the right packaging solution, floorplan layout, thermal management techniques, etc. that will be implemented on the final system in order to meet the design constrains. Once the chip is manufactured, the EP is still a valuable instrospection technique to refine the SW of the system that, as I have demonstrated through the examples, seriously affects not only the performance, but also the power consumption, the temperatures, and the reliability of the system.

Chapter 6

Conclusions

Traditional SoCs are not able to meet the tight design constrains (e.g. size, cost, energy consumption) of high performance embedded systems; their increasing complexity, coupled with a reduced time-to-market window, has revolutionized the design process. Nowadays, designing a state-of-the-art dedicated system starting from scratch and trying to optimize globally all the necessary modules is an extremely complex task; thus, the only valid alternative, at least at short- and mid-term, is the application of the new paradigm of MPSoCs, that consists on designing a system by using composition and reuse of existing components designed independently. Nevertheless, the high density of logic inside these MPSoCs brings new problems, like extreme on-chip temperatures and reliability issues, to the system designers.

One of the main design challenges, for example, is the fact that the SW must be capable of efficiently using the optimizations that the HW offers to enhance the performance of the embedded applications and reduce the power consumption, that has become a critical issue with the last technologies. When they are developed independently, there is little opportunity to optimize the HW-SW interaction; they must be evaluated together, task that results in a huge design space.

In this situation, intensive testing of the system at early stages of the design process is mandatory in order to correctly tune the final architecture and efficiently reach the specified funtionality satisfying the given set of constraints (e.g., development time, cost, power consumption, performance, technology, etc.).

In general, the exploration techniques must be able to investigate a large part of the design and manufacturing spectrum of MPSoC implementations (e.g., various floorplan layouts or packaging technologies, multiple frequencies and supply voltages, etc). Hence, I believe that a promising solution to effectively provide performance, power, temperature and reliability studies are the hybrid HW/SW exploration frameworks [ADVP+08]. These frameworks can merge cycle-accurate HW emulation (to obtain the switching activity of

internal components at fast speed, with respect to pure MPSoC architectural simulators [BBB+05]), with flexible SW estimation models.

In this context, the goal of this thesis has been to introduce a new HW/SW emulation framework, the EP, that allows designers to speed up the design cycle of MPSoCs. The HW part of the EP (Chapter 2) is based on an FPGA, that hosts the emulation and extracts the run-time information from the *Emulated System*, while a desktop computer receives these data and uses them as the input to SW models (Chapter 3) that predict the power consumption, the temperature, and the reliability of the final system. Both parts are integrated in one single flow (Chapter 4), that simplifies the task of the system designer.

The experimental results, in Chapter 5, show that the proposed framework obtains detailed reports of the power, thermal and reliability features of the final MPSoCs, with speed-ups of three orders of magnitude compared to cycle-accurate MPSoC simulators. Also, the addition of more processing cores and more complex memory architectures to the emulation framework, suitably scales, enabling long simulations of complex systems (as required by thermal and reliability modeling, for example).

First, the framework has been used to study the thermal profile of different packaging solutions and floorplan alternatives (where I proposed an, *a priori*, intelligent placement of the on-chip components).

Second, since the real-time interaction between HW emulation and SW thermal modeling enables the application of Dynamic Thermal Management (DTM) policies to the emulated MPSoC at run-time, the EP has been used to validate several of these techniques, from pure-HW solutions to elaborated Operating-System-level policies, suitable for a wide range of MPSoCs, depending on the needs of each design.

Regarding reliability, a deep study at the microarchitectural level has been performed, with the help of the EP, in order to extend the lifespan of a Leon3 core by modifying the compiler.

Finally, the versatility of the EP has been extended by adding a Multi-Processor Operating System (MPOS) with task migration support to the *Emulated System*. It is a simple but complete MPOS that opens the door to the experimentation with advanced thermal-aware MPOS strategies. The initial results show the usefulness of this framework to explore the benefits of thermal-aware management at the OS level in MPSoC designs.

In the next section, I synthesize the main contributions of this thesis.

6.1. Main Contributions

As the main contribution of this thesis, I have developed a HW/SW FPGA-based emulation framework, the EP, that allows designers to explore a wide range of design alternatives of complete MPSoC systems, characterizing them (in terms of behaviour, performance, power, temperature and reliability) at a very fast speed with respect to MPSoC architectural simulators, while retaining cycle-accuracy.

The EP offers one integrated design flow that reduces the complexity of the MPSoC development cycle. Through examples and experiments I have shown how this HW/SW framework allows designers to test run-time thermal management strategies with real-life inputs, observe their long-term effects on chip reliability, and analyze different MPSoC design alternatives, for example. More exactly, the EP has been effectively used to:

- Reduce the hotspots of a system by using thermal-aware placement techniques, that assign a suitable placement to the diferent MPSoC components at an early stage of system integration.

- Study the effects of using DTM techniques or different packaging alternatives for specific MPSoC solutions; some of the improvements will come for free, and some others at very little cost (economical, performance). The results indicate that the selection of final packaging solutions clearly depends on the thermal management techniques included in the target MPSoCs, and that significant overheads of power dissipated in long interconnects can clearly affect the overall thermal behaviour of the final MPSoC. On the other hand, other non-evident conclusions are also found out with this framework, like the fact that costly packagings may show from the same heating effects as low-cost ones, or that a uniform distribution of power sources in the die does not need to produce a uniform temperature in the final chip. Overall, this kind of design decisions are not trivial, and require extensive thermal exploration to justify, for instance, the need of expensive packaging solutions.

- Modify the register assignment policy of the compiler to reduce the hotspots and improve reliability at the microarchitectural level. This experiment showed the importance of studying the HW interaction while running the final SW application, instead of using synthetic benchmarks.

- Create a thermal-aware OS, by modifying the task scheduling policy (at the kernel level) of a uCLinux distribution, to balance the temperature in a multi-processor environment by migrating tasks at run-time. As

part of the results, it has been quantified the penalty (in time) of the migrations with respect to the temperature evolution.

6.2. Legacy

The Emulation Platform is an ambitious project whose seed was planted, back in 2005, at the University Complutense of Madrid. More exactly, inside my Computer Systems Engineering Master's Project. Nevertheless, the project fully flourished thanks to the collaboration with other research groups from around the Globe:

- The group of Architecture and Technology of Computing Systems (ArTeCS) of the Complutense University of Madrid, Spain.
- The Embedded Systems Laboratory (ESL), and the Integrated Systems Laboratory (LSI), at the Institute of Electrical Engineering within the School of Engineering (STI) of EPFL, Switzerland.
- The Department of Mathematics and Computer Science of the University of Cagliari, Italy.
- Department of Electronic Engineering and Information Science (DEIS), University of Bologna, Italy.
- The Department of Computer Science and Engineering at the Pennsylvania State University, EEUU.

Next, I present the list of publications, related to the Emulation Platform, that I have produced during my PhD:

1. "A Fast HW/SW FPGA-Based Thermal Emulation Framework for Multi-Processor System-on-Chip", David Atienza, Pablo G. Del Valle, Giacomo Paci, Francesco Poletti, Luca Benini, Giovanni De Micheli, Jose M. Mendias, 43rd Design Automation Conference (DAC), ACM Press, San Francisco, California, USA, ISSN:0738-100X, ISBN: 1-59593-381-6, pp. 618-623, July 24-28, 2006.

2. "A Complete Multi-Processor System-on-Chip FPGA-Based Emulation Framework", Pablo G. Del Valle, David Atienza, Ivan Magan, Javier G. Flores, Esther A. Perez, Jose M. Mendias, Luca Benini, Giovanni De Micheli, Proc. of 14th Annual IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC), Nice, France, ISBN: 3-901882-19-7 2006 IFIP, IEEE Catalog: 06EX1450, pp. 140 -145, October 2006.

3. "Architectural Exploration of MPSoC Designs Based on an FPGA Emulation Framework", Pablo G. del Valle, David Atienza, Ivan Magan, Javier G. Flores, Esther A. Perez, Jose M. Mendias, Luca Benini, Giovanni De Micheli, XXI Conference on Design of Circuits and Integrated Systems (DCIS), Barcelona, Spain. Publisher Departament dÉlectrónica-Universitat de Barcelona, pp. 1-6, November 2006.

4. "HW-SW Emulation Framework for Temperature-Aware Design in MPSoCs", David Atienza, Pablo G. Del Valle, Giacomo Paci, Francesco Poletti, Luca Benini, Giovanni De Micheli, Jose M. Mendias, Roman Hermida, ACM Transactions on Design Automation for Embedded Systems (TODAES), ISSN: 1084-4309, Association for Computing Machinery, Vol. 12, Nr. 3, pp. 1 - 26, August 2007.

5. "Application of FPGA Emulation to SoC Floorplan and Packaging Exploration", Pablo G. Del Valle, David Atienza, Giacomo Paci, Francesco Poletti, Luca Benini, Giovanni De Micheli, Jose M. Mendias, Roman Hermida. Proc. of XXII Conference on Design of Circuits and Integrated Systems (DCIS), Sevilla, Spain. Publisher Departament dÉlectrónica-Universitat de Barcelona, November 2007.

6. "Reliability-Aware Design for Nanometer-Scale Devices", David Atienza, Giovanni De Micheli, Luca Benini, José L. Ayala, Pablo G. Del Valle, Michael DeBole, Vijay Narayanan. Proceedings of the 13th Asia South Pacific Design Automation Conference, ASP-DAC 2008, Seoul, Korea, January 21-24, 2008. IEEE 2008.

7. "Emulation-Based Transient Thermal Modeling of 2D/3D Systems-on-Chip with Active Cooling", Pablo G. Del Valle, David Atienza. Microelectronics Journal, Elsevier Science Publishers B. V., Vol. 42, Nr. 4, pp. 564 - 571, April 2011.

8. "Performance and Energy Trade-offs Analysis of L2 on-Chip Cache Architectures for Embedded MPSoCs", Aly, Mohamed M. Sabry, Ruggiero Martino, García del Valle, Pablo. Proceedings of the 20th symposium on Great lakes symposium on VLSI, 2010, p. 305-310. ISBN: 978-1-4503-0012-4.

In addition to the aforementioned publications, this framework has been used by third parties to validate their research ideas. Amongst the most relevant publications derived from this work, where I did not participate directly, we can find:

- "Adaptive task migration policies for thermal control in MPSoCs", D. Cuesta, J.L. Ayala, J.I. Hidalgo, D. Atienza, A. Acquaviva, E. Macii. ISVLSI, IEEE Computer Society Annual Symposium on VLSI, 2010.

- "Thermal-aware floorplanning exploration for 3D multi-core architectures", D. Cuesta, J.L. Ayala, J.I. Hidalgo, M. Poncino, A. Acquaviva, E. Macii. Proceedings of the 20th symposium on Great lakes symposium on VLSI, GLSVLSI 2010.
- "Thermal balancing policy for multiprocessor stream computing platforms", F. Mulas, D. Atienza, A. Acquaviva, S. Carta, L. Benini, and G. De Micheli. Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2009.
- "Thermal-aware compilation for register window-based embedded processors", Mohamed M. Sabry, J.L. Ayala, and D. Atienza. Embedded Systems Letters, 2010.
- "Thermal-aware compilation for system-on-chip processing architectures.", Mohamed M. Sabry, J.L. Ayala, and D. Atienza. Proceedings of the 20th symposium on Great lakes symposium on VLSI, GLSVLSI 2010).
- "Impact of task migration on streaming multimedia for embedded multiprocessors: A quantitative evaluation.", M. Pittau, A. Alimonda, S. Carta and A. Acquaviva. Proceedings of the 2007 5th Workshop on Embedded Systems for Real-Time Multimedia, ESTImedia 2007.
- "Assessing task migration impact on embedded soft real-time streaming multimedia applications.", A. Acquaviva, A. Alimonda, S. Carta and M. Pittau. EURASIP Journal on Embedded Systems, 2008.
- "Energy and reliability challenges in next generation devices: Integrated software solutions", Fabrizio Mulas. PhD. Thesis at the Mathematics and Computer Science Department of the University of Cagliari, 2010.

6.3. EP enhancements

In this section, I propose several improvements that could be introduced in the EP. They possibilitate new uses of the platform, or facilitate the existing ones. However, they all require a strong implementation effort. Some of them are interesting extensions, while others will be a necessity as the system grows:

Multi-FPGA environment

As observed in the experiments, with five simple RISC processing cores mapped in the platform, we are close to the limits of a Virtex II Pro VP30

FPGA, in terms of resource usage. In order to model bigger environments, two alternatives are valid: (i) migrate to bigger FPGAs, or (ii) expand the framework by adding support for multiple FPGAs. The first option is a straight forward solution that does not require any modifications. However, the second option is more interesting from the economical point of view, since it keeps the price of the platform low: despite the advancements in FPGA technology, the biggest models are orders of magnitude more expensive. Therefore, the importance of investigating multi-FPGA extensions.

Regarding the implementation, the main challenge is the synchronization of the different *emulation islands*, that should run together at megahertz speeds. In addition to the data interchange that takes place inside the *Emulated System*, the *Emulation Engine* and the *SW Libraries for Estimation* must also be synchronized, in order to pause, resume, etc... the emulation at the same moment, and process simultaneously the collected data corresponding to the last *Emulation Step*.

FPGA-PC communication

As the volume of exchanged information (FPGA-PC and PC-FPGA) grows, mainly due to the increasing size of FPGAs capacity, or a possible extension of the EP to a multi-FPGA environment, the Ethernet connection will become insuficient. More efficient methods should be explored in order to avoid this bottleneck. Upgrading the Ethernet connection to a Gigabit link would be the next reasonable step. However, more advanced solutions should also be explored, like using the PCI bus, or the high-speed Serial IO implemented in Xilinx FPGAs.

Porting new processors

It is always interesting to port new cores to the emulation framework. The source code (Verilog) of the OpenRisc1200 Processor [bibd], for example, is available on the internet, and free of charge, which converts it in a good candidate for architecture exploration.

Third party tools

Incorporating third party tools into the flow of the EP would simplify the task of system designers and, at the same time, would extend the versatility of the framework. For those tools that are already compatible with the input, output, or intermediate file formats used in the EP (see Section 4.2), it would

be very nice to include scripts that allow the complete automation of the design flow.

Sunfloor [SMBDM09], for instance, is a tool that guides designers in the task of creating a floorplan, generating automatically system layouts given a set of constrains. Integrating it into the design flow would possibilitate that several design alternatives could be automatically explored without requiring user interaction.

6.4. Open research lines

In this section, I present some possible application fields that I have identified as the most promising ways to use my framework.

Complex dynamic thermal management policies

With the increasing power density of current MPSoC designs, thermal control has become a priority in the design cycle. In the experiments chapter, I introduced some simple thermal management policies (HW and SW based) in order to demonstrate the usefulness of the EP to conduct this type of experiments.

Therefore, we can use the EP to test advanced DTM techniques: Similarly to DFS, we can implement cache throttling, fetch-toggling, speculation control... the only requirement, in addition to implementing the HW support for the selected techniques, is the modification of the *Decision Engine* (see Section 5.3.1.2); i.e.: the algorithm that autonomously triggers the thermal countermeasures. It can be modified to take decitions based on classic control theory, for instance, or employ complex neural networks that self-learn from the past thermal history of the system. The perfect policy always depends on the particular case, and the goals of the optimization: maximize the performance of the system, reduce the power consumption, extend the lifespan of the chip, ensure a minimum QoS...

In the future, I would like to study more in detail the relationship between complex OS-based thermal management techniques and the reliability of the MPSoCs.

Fault injection

New proposals and studies are being developed around the idea that computation can or cannot be correct at a certain moment. Using fault injection techniques [HTI97], designers can analyze the behaviour of a system

under unexpected circumstances. The architecture can then be modified, in order to improve the error handling, and reduce the vulnerabilities of the system. One way to enhance robustness, for instance, is to introduce redundancy in the operations (HW or SW). In any case, the emulation framework offers the possibility to test fault injection theory. Since the *Emulation Engine* has total control, at any moment, of the *Emulated System*, we would only need to add a mechanism to inject faults at the specified points. The mechanism is similar to introducing the temperature of the system from the thermal library output, back into the *Emulated System* sensors (see Section 2.1.2.1).

This field is specially interesting for the aerospatial market, for instance, where a high reliable version of a chip is normally preferred over a similar one offering higher performance, even orders of magnitude, that cannot provide a certain level of determinism.

Side channel attacks

Side channel attacks [BE] are attacks to electronic cyptosystems that are based on the "side" information that can be retrieved during the operation of the encryption device (such as timing information, power consumption, electromagnetic leaks or even sound), which can be exploited to break the system.

The idea is to use the EP to rapidly evaluate the robustness of different implementation alternatives against side channel attacks.

Currently, in the EP, we already have models to estimate the power and the temperature of the final MPSoC. However we would need to increase the precission of the calculations, and calculate the variations in power consumption or temperature every cycle, as side channel attacks require, instead of after the last *Emulation Step*. We could add, as well, models for the noise or electromagnetism generated, for example.

High-level synthesis

High-level synthesis [ANRD04], is an automated design process that interprets an algorithmic description of a desired behaviour and creates HW that implements that behaviour.

Solutions given by high-level synthesis algorithms must be examined carefully [dBG11]; one algorithm aiming at the minimization of cycle time can increase the overall area at an unafordable price or, the opposite effect, performance can be degraded while trying to minimize power consumption, as this is quite influenced by the frequency, i.e. the cycle time inverse. All these

aspects can be rapidly evaluated using the EP. In fact, we could create an automated flow to evaluate the multiple implementation choices generated by a parametrizable high-level synthesis engine without user interaction at all.

Appendix A

Resumen en Español

En cumplimiento del Artículo 4 de la normativa de la Universidad Complutense de Madrid que regula los estudios universitarios oficiales de postgrado, se presenta a continuación un resumen en español de la presente tesis que incluye la introducción, objetivos, principales aportaciones y conclusiones del trabajo realizado.

A.1. Introducción

Cuando mencionamos la palabra procesador, generalmente y de manera intuitiva, pensamos en los procesadores de propósito general (GPP's); aquellos que funcionan como servidores, estaciones de trabajo, u ordenadores personales, fabricados por marcas de renombre, como Intel, que están ampliamente extendidos por el mundo, y que sirven para solucionar un amplio abanico de problemas. Sin embargo, existen otros tipos de procesadores mucho más presentes en nuestra vida diaria: los procesadores empotrados y los microcontroladores. Son procesadores sencillos que se encuentran en sistemas dedicados como, por ejemplo, dentro del microondas, de la lavadora, del secador, de los reproductores de DVD, o en el automóvil.

En los últimos años, los avances en tecnología han propiciado una significativa evolución de los sistemas empotrados. Muchos de ellos han pasado de ser simples sistemas de control designados específicamente para realizar una tarea o un conjunto reducido de tareas, a convertirse en sistemas más complejos, que ejecutan aplicaciones similares a las que encontramos en ordenadores de sobremesa, pero con fuertes requisitos que satisfacer. Este nuevo tipo de sistemas se denominan *sistemas empotrados de altas prestaciones.*

El mercado de la electrónica de consumo, por ejemplo, está dominado por dispositivos como tablets, teléfonos inteligentes, cámaras digitales, o sistemas de navegación GPS. Estos sistemas son complejos de diseñar, puesto que deben ejecutar múltiples aplicaciones a la vez que respetar restricciones adicionales de diseño, como un consumo reducido de energía, o un tamaño

pequeño. Por si fuera poco, la rápida evolución de la tecnología está reduciendo cada vez más el tiempo de salida al mercado y el precio de estos sistemas [JTW05], lo que no permite el rediseño de un chip para cada producto. En este escenario, los Sistemas-en-Chip (SoCs) son una solución efectiva de diseño, puesto que integran en un sólo chip diferentes IP cores que ya han sido verificados en diseños anteriores. Cuando tenemos varios procesadores dentro de un mismo chip, pasan a denominarse Sistemas-en-Chip Multi-Procesador, o MPSoCs.

Diseñar un MPSoC es una tarea muy compleja; incluso si fijamos los IP cores a utilizar, el espacio de exploración aún es gigantesco. Los diseñadores deben decidir múltiples detalles HW, desde los aspectos de alto nivel (e.g., la frecuencia del sistema, la ubicación de los cores, o la interconexión), a los de más bajo nivel, como el rutado de la red de distribución del reloj, la tecnología a emplear, etc. Por si fuera poco, encima de todo esto viene el SW: si un procesador ejecuta aplicaciones en C, o tiene un Sistema Operativo completo, son decisiones que se han de tener en cuenta en tiempo de diseño.

Un cambio en cualquiera en los parámetros HW o SW de un MPSoC no sólo afectará al rendimiento del sistema final, sino que también puede repercutir en el tamaño físico del chip, el consumo de potencia, o la temperatura y fiabilidad de los componentes (e.g., dando lugar a la aparición de puntos calientes que comprometan la fiabilidad del chip [SSS+04]). Así pues, uno de los principales retos en el diseño de MPSoCs es conseguir herramientas que permitan explorar, en tiempo de diseño, las múltiples opciones HW y SW de implementación con estimaciones fieles del rendimiento del sistema final (en cuanto a energía, potencia, temperatura, etc...).

A.1.1. Trabajo relacionado

En lo que respecta al modelado térmico de MPSoCs, varios trabajos estudian la aparición de puntos calientes en los sistemas empotrados de alto rendimiento: [SSS+04] presenta un modelo de potencia y térmico para arquitecturas superescalares que predice las variaciones de temperatura de los diferentes componentes de un procesador. En [SLD+03] se ha investigado el impacto de las variaciones de temperatura y voltaje de un core empotrado; sus resultados muestran variaciones de hasta 13.6 grados a lo largo del chip. En [LBGB00] se mide la temperatura de trabajo de FPGAs, usadas como procesadores reconfigurables, usando osciladores de anillo que pueden ser dinámicamente insertados, reubicados, o eliminados. A pesar de que este método es interesante, sólo es aplicable a diseños donde las FPGAs son el dispositivo final.

En conjunto, estos trabajos resaltan la importancia y necesidad de estudiar el comportamiento (en cuanto a rendimiento, potencia, temperatura, y fiabilidad) de los MPSoCs en las etapas tempranas del ciclo de diseño. Para ello, los diseñadores se valen de una serie de herramientas, que pode-

mos clasificar, principalmente, en simuladores SW y emuladores HW (existe también el prototipado HW, pero no será incluido en mi estudio debido a que se aplica en las etapas finales de diseño).

En cuanto a los simuladores SW, se han propuesto soluciones a diferentes niveles de abstracción, con el objeto de ofrecer un compromiso entre fidelidad de las estimaciones y tiempo de simulación. Por ejemplo, modelos analíticos con lenguajes de alto nivel (C/C++) [BWS+03], o simuladores como Symics [MCE+02], que son muy rápidos y útiles para la depuración del SW, pero que no capturan con exactitud las medidas de potencia y rendimiento del HW. A más alto nivel, describiendo el sistema a nivel transaccional en SystemC [PPB02] y [BBB+05] en el ámbito académico, así como [CoW04] y [ARM02] en el industrial, ofrecen más detalle, pero a costa de perder velocidad (100-200 KHz). Por último, simuladores como [Gra03] y [Syn03] usan bibliotecas post-síntesis, y ofrecen gran nivel de detalle; sin embargo, la velocidad de simulación se ve reducida a 10-50 KHz.

La mayor desventaja de usar simuladores SW a nivel de RTL para estudiar los MPSoCs es la gran pérdida de rendimiento asociada al aumento del número de elementos en el sistema a simular (que trae consigo un mayor número de señales que hay que modelar y mantener sincronizadas).

La emulación HW solventa este problema pero, como contrapartida, ofrece una menor flexibilidad. Así, en la industria, tenemos Palladium II [Cad05], que opera en torno a los 1.6 MHz, y cuesta alrededor de 1 millón de dólares. ASIC Integrator [ARM04a] es mucho más rápido, pero está limitado a 5 cores ARM, e interconexiones AMBA. Heron SoC emulation [Eng04] tiene similares limitaciones. System Explore [Apt03] y Zebu-XL [EE05] usan FPGAs para emular a velocidades del orden de los MHz, pero no son lo suficientemente flexibles a la hora de extraer las estadísticas. En el mundo académico, tenemos TC4SOC [NBT+05], que permite estudiar cores VLIW y Redes-en-Chip. Sin embargo, tampoco permite extraer estadísticas detalladas. Una solución interesante se describe en [NHK+04], donde utilizan un entorno mixto FPGA-PC para la emulación, realizando una sincronización ciclo-a-ciclo del SW que corre en el PC con un array de registros compartidos mapeados en la FPGA, y llegando al Megahercio de velocidad. Recientemente, ha aparecido el proyecto RAMP (Research Accelerator for Multi-Processors) [AAC+05], que también explota una infraestructura mixta HW/SW.

Utilizando estas herramientas, tanto simuladores como emuladores, para estudiar el comportamiento de los MPSoCs, se han empezado a proponer soluciones para los problemas de consumo, temperatura y fiabilidad dentro del chip. De hecho, técnicas para reducir el consumo máximo de potencia, la temperatura media, o mantenerlas por debajo de un límite, por ejemplo, están siendo implementadas en los chips actuales.

Estudios recientes [CW97; CS03; GS05] han demostrado que un emplazamiento inteligente de los cores puede reducir el gradiente térmico dentro

del chip. Esto lleva a nuevas líneas de investigación para futuros MPSoCs, como pueden ser la *síntesis consciente de la potencia*, y el *emplazamiento consciente de la temperatura*.

En [SSS+04] se usa teoría formal de control como método para implementar técnicas adaptativas. [SA03] propone un algoritmo de control de temperatura predictivo para aplicaciones multimedia. También [BM01] ha realizado extensos estudios sobre técnicas a aplicar (DVS, DFS, fetch-toggling, throttling, control de especulación), cuando el consumo de un procesador cruza un determinado límite. A más alto nivel, en [RS99], el procesador deja de planificar tareas calientes cuando la temperatura supera un cierto valor, de manera que la CPU pasa más tiempo en estados de bajo consumo, lo que permite reducir la temperatura local o globalmente.

Añadiendo mecanismos SW o HW al sistema fabricado para limitar dinámicamente la máxima potencia o temperatura permitida en tiempo de ejecución, podemos reducir el coste del empaquetado y extender la vida útil del chip, por ejemplo. La desventaja fundamental de los métodos dinámicos es el impacto en el rendimiento, asociado al hecho de detener o ralentizar el procesador [SSS+04]. Es en esta línea donde los MPSoCs abren nuevas posibilidades, como la asignación de trabajos, o migración de tareas en función de las temperaturas [CRW07], [DM06]. Sin embargo, en estos casos tambien necesitamos de detallados estudios y potentes herramientas para determinar el mejor método a implementar, que dependerá de las restricciones de cada diseño particular (rendimiento, temperatura máxima, coste, ...).

A.1.2. Objetivos de esta tesis

Como ya he explicado durante la introducción, uno de los principales retos a los que se enfrentan los diseñadores de MPSoCs es a poder explorar rápidamente múltiples alternativas de implementación (HW y SW), con estimaciones certeras de rendimiento, energía, potencia, temperatura y fiabilidad, para poder ajustar la arquitectura del sistema en etapas tempranas del proceso de diseño.

En este trabajo de investigación, presento un nuevo entorno de emulación HW/SW, basado en FPGA, que permite a los diseñadores de MPSoCs explorar una amplia variedad de alternativas de diseño, analizando su comportamiento a nivel de ciclo de reloj más rápidamente que con simuladores SW. Mediante ejemplos y experimentos, demuestro que este entorno permite no sólo evaluar el sistema, sino probar estrategias de control (de potencia, temperatura y fiabilidad) en tiempo real, y observar sus efectos a largo plazo en el chip, que variarán dependiendo de las distintas alternativas de diseño seleccionadas.

Como veremos, una característica primordial del entorno es que ha sido concebido desde el principio para ser versátil y flexible de manera que, en el

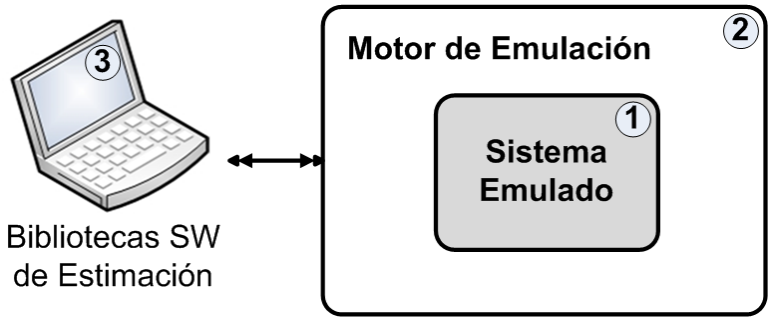


Figura A.1: Esquema de alto nivel de la Plataforma de Emulación.

futuro, pueda ser fácil incorporar nuevas características a la plataforma.

A.2. La plataforma HW de emulación

La Plataforma de Emulación (PE) está compuesta por tres partes, tal y como aparecen en la Figura A.1:

1. **El Sistema Emulado:** Es el MPSoC que está siendo optimizado, el sistema en observación, que será refinado hasta que cumpla con las restricciones de diseño.

2. **El Motor de Emulación:** Es toda la arquitectura HW que hay alrededor del Sistema Emulado, y que se encarga de controlarlo, monitorizarlo, y extraer estadísticas en tiempo de ejecución para enviarlas a un PC. El Motor de Emulación funciona de manera similar a un simulador arquitectónico SW, al que le tenemos que introducir la arquitectura MPSoC a simular.

3. **Las Bibliotecas SW de Estimación:** Se ejecutan en un PC, y calculan la potencia consumida, la temperatura, la fiabilidad, etc. del Sistema Emulado, en base a los datos recibidos en tiempo de ejecución desde el Motor de Emulación.

En el flujo de trabajo con la PE, el usuario descarga, desde el PC, el entorno HW completo (tanto el Sistema Emulado como el Motor de Emulación) a la FPGA. A continuación, se lanza una interfaz gráfica que permitirá al usuario monitorizar el proceso de emulación, e interactuar con el sistema introduciendo comandos de control. La emulación comienza tras ejecutar un comando de "start", y se desarrolla de forma autónoma: las estadísticas generadas son periódicamente enviadas a través de un puerto de comunicaciones al PC, que las registra, y las usa como entrada a las Bibliotecas SW de Estimación que calculan potencia, temperatura, fiabilidad, etc... del MPSoC

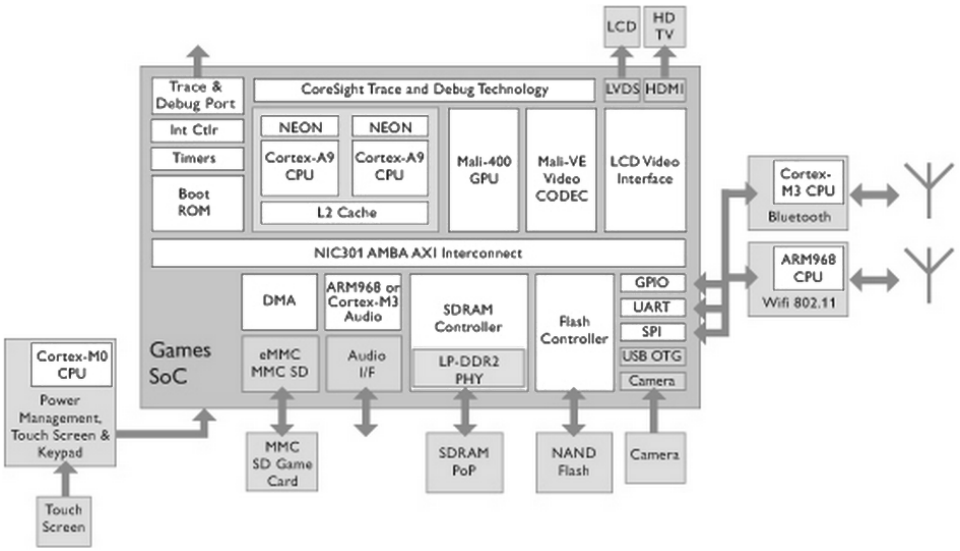


Figura A.2: Plataforma de videojuegos ARM: un ejemplo de arquitectura MPSoC heterogénea.

final; así pues, el proceso de emulación está dividido en *pasos de emulación*: corre durante un número prefijado de ciclos, se detiene, intercambia información (hacia y desde el PC), y continúa con el siguiente paso.

A continuación, se describe la parte HW de la PE; i.e., la que reside en la FPGA, formada por el Sistema Emulado y el Motor de Emulación.

A.2.1. El Sistema Emulado

A grandes rasgos, un MPSoC consta de cores de procesamiento (ARM, VLIW, etc...), una arquitectura de memoria, y un sistema de interconexión.

La Figura A.2 muestra un ejemplo de arquitectura MPSoC. Se trata de una plataforma de videojuegos diseñada por ARM. En el diagrama de bloques podemos observar una pareja de Cortex-A9, que son los procesadores principales. Ambos contienen el coprocesador NEON, diseñado para acelerar las operaciones de procesado de señal. A través de un bus AMBA AXI, también tienen acceso a dos aceleradores multimedia Mali, varias memorias en chip (flash, ROM), y a interfaces de entrada/salida (USB, tarjetas de memoria, etc.). Hay procesadores ARM adicionales para gestionar operaciones especiales, como la entrada de pantalla táctil, el audio de alta definición, y las comunicaciones WIFI y Bluetooth.

La PE permite instanciar sistemas heterogéneos, como el de este ejemplo. Tras el proceso de síntesis, todo elemento del Sistema Emulado es convertido a una *netlist*, y mapeado en la FPGA correspondiente. Por tanto, como formatos de entrada, se pueden utilizar desde *netlists* directamente, hasta

otros lenguajes HDL que ofrezcan niveles más altos de abstracción, como Verilog, VHDL, o SystemC sintetizable.

A.2.1.1. Emulación y elementos modelados

En el diseño de circuitos integrados, emulación HW es el proceso de imitar el comportamiento de uno o más elementos de HW, con otro elemento HW; tipicamente, un sistema de emulación de propósito especial. Por otro lado, el prototipado HW es el proceso de obtener un circuito con un diseño muy cercano al final. Mientras que la emulación HW puede incluir elementos modelados, el prototipado HW, sin embargo, requiere que los componentes finales estén disponibles, y se aplica típicamente en las etapas finales del ciclo de diseño.

A la hora de diseñar Sistemas Emulados, la PE permite utilizar tanto elementos completamente especificados, como elementos modelados. Estos últimos, también llamados componentes virtuales, sólo existen dentro de la emulación. Se usan cuando el componente real no está aún implementado, o en situaciones donde no puede ser incluido en la plataforma o, sencillamente, no interesa trabajar con él (e.g., porque ocupa muchos recursos). En la implementación final, serán reemplazados por un componente final, o incluso por otro chip conteniendo la funcionalidad que fue previamente modelada en la emulación.

Los sensores de temperatura introducidos en el Sistema Emulado constituyen un ejemplo de componentes modelados: dado que no tiene sentido poner un sensor real en la FPGA (recordemos que no es el *target device*), usamos sensores falsos, que devuelven temperaturas previamente introducidas por el Motor de Emulación.

A.2.2. El Motor de Emulación

El Motor de Emulación consta de los siguientes elementos (ver Figura A.3):

1. **El Gestor del Reloj Virtual de la Plataforma (VPCM):** Se encarga de sincronizar los diferentes dominios de reloj del Sistema Emulado.

2. **El Subsistema de Extracción de Estadísticas:** Extrae, de forma transparente, la información del Sistema Emulado.

3. **El Gestor de las Comunicaciones:** Se encarga de controlar la comunicación bidireccional entre la FPGA y el PC.

4. **El Director del Motor de Emulación:** controla y sincroniza el sistema entero, dirigiendo la extracción de estadísticas, y la sincronización FPGA-PC.

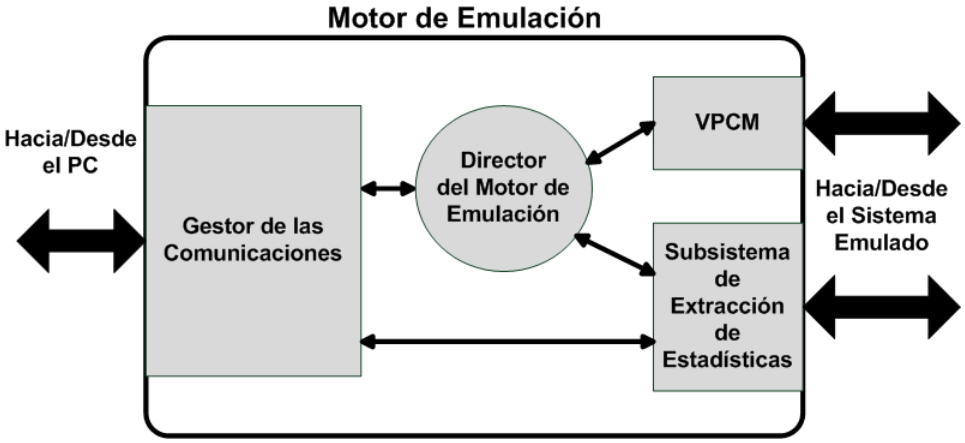


Figura A.3: Partes del Motor de Emulación.

A.2.2.1. El Gestor del Reloj Virtual de la Plataforma (VPCM)

El funcionamiento de la PE es análogo a los simuladores SW disparados por eventos: cada vez que sucede un evento de reloj, se desencadenan una serie de actualizaciones de señales, hasta que todas las señales quedan estables. El emulador espera entonces preparado para emular el siguiente ciclo de reloj.

Internamente, la PE utiliza múltiples dominios de reloj, denominados "relojes virtuales". El VPCM es el módulo que se encarga de generarlos y gestionarlos, permitiendo inhibirlos temporalmente, con objeto de sincronizar el sistema, u ocultar latencias de módulos modelados. Cada paso de emulación consta de un número prefijado de ciclos de reloj virtual.

A.2.2.2. El Subsistema de Extracción de Estadísticas

El Subsistema de Extracción de Estadísticas tiene como objetivo extraer la información del Sistema Emulado de manera transparente. A tal efecto, se han diseñado e implementado los *sniffers* HW; unos módulos que monitorizan las señales internas de los cores y el pinout externo de los elementos incluídos en el MPSoC emulado. La Figura A.4 muestra varios de estos dispositivos (nombrados como *Sniffer 1...4*) conectados a los cores monitorizados correspondientes (con patrón de rayas en el dibujo).

En la Figura A.5 se muestra el esquema completo del Subsistema de Extracción de Estadísticas; en ella se aprecian sus tres componentes: los *sniffers*, el Bus de Estadísticas, y el Extractor de Estadísticas.

El Bus de Estadísticas ha sido diseñado para permitir una eficiente recolección de las mismas (almacenadas en los *sniffers*), y permite, además, acceder a los *sniffers* para tareas de control (activar/desactivar la recolección, etc...).

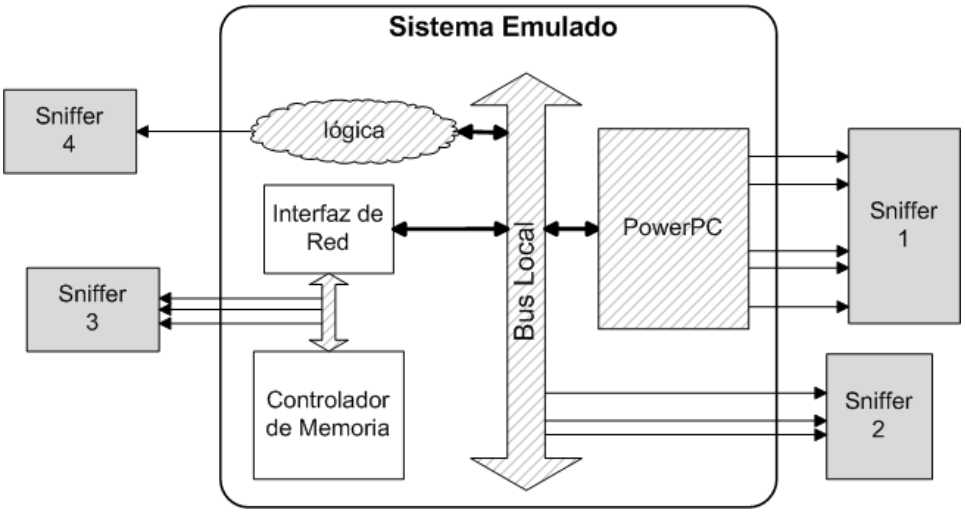


Figura A.4: Sistema Emulado con varios *sniffers*.

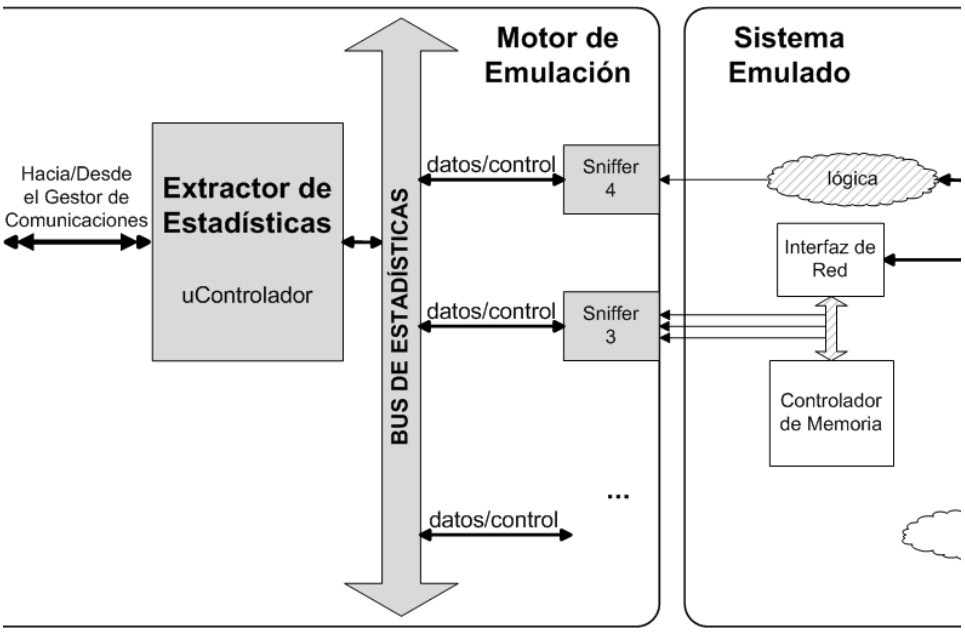


Figura A.5: Esquema del Subsistema de Extracción de Estadísticas.

El tercer elemento, que completa el Subsistema de Extracción de Estadísticas, es el Extractor de Estadísticas; un microcontrolador encargado de acceder a los *sniffers* (a través del Bus de Estadísticas), e intercambiar información con el PC, a través del Gestor de Comunicaciones.

La siguiente sección está dedicada a describir en datalle el funcionamiento de los *sniffers*, que constituyen el elemento fundamental de la PE.

Los sniffers HW

Los *sniffers* HW son elementos que, de forma transparente, extraen las estadísticas de cada componente del Sistema Emulado (i.e., no interfieren ni modifican el comportamiento normal de los cores estudiados). Todos los *sniffers* tienen una interfaz dedicada para capturar las señales internas del módulo que están monitorizando, lógica que convierte esta actividad de señales en estadísticas, una pequeña memoria para almacenarlas, y una conexión al Bus de Estadísticas, que permite la extracción de las mismas.

Dependiendo de cómo esté especificado un módulo, el diseñador podrá acceder a más o menos información del mismo. En algunas ocasiones se dispone de la totalidad del código fuente mientras que, en otras, sólamente tenemos acceso parcial (a través de puertos de depuración, de análisis, o de sincronización) para conocer el estado en que se encuentra el core. A veces, incluso, el componente es una caja negra (encriptada, o *hard-coded* en silicio), cuyo comportamiento hemos de averigüar estudiando las señales de entrada/salida del mismo.

Para crear un nuevo sniffer, en primer lugar, el diseñador ha de definir qué quiere monitorizar del componente en cuestión. El procedimiento general consiste en observar una serie de señales y procesarlas para obtener información útil. Para facilitar esta tarea, he diseñado plantillas de los tipos más comunes de *sniffers*. En ellas, la conexión al Bus de Estadísticas está ya implementada, de manera que sólo se necesita implementar la interfaz con el módulo monitorizado. Enumero a continuación las cinco plantillas, junto con una breve descripción de cada una, que nos ayudará a entender el funcionamiento de los *sniffers*:

1. **Sniffer guarda eventos:** Guarda detalladamente todos los eventos que suceden, al estilo: "en el ciclo 24 hubo un acceso de lectura de 1 byte a la dirección xFFAA del banco de memoria 2".

2. **Sniffer de conteo de eventos:** Cuenta los eventos de un cierto tipo que sucedieron; e.g., "el controlador de memoria realizó 320 lecturas y 470 escrituras". Es lo que típicamente demandan los diseñadores de los simuladores SW a nivel de ciclo.

3. **Sniffer de chequeo de protocolo:** Chequea si todas las transacciones ocurridas siguen la especificación. Útil para tareas de depuración.

4. **Sniffer de utilización de recursos:** Provee información acerca del grado de saturación de un elemento, como puede ser la utilización de un bus; e.g.: "trabaja al 80 %".

5. **Sniffer de postprocesado:** Procesa la información de un sniffer de conteo de eventos para convertirla en otro tipo de datos (e.g.: consumo), extraer patrones, etc...

Los *sniffers* más relevantes en la PE son los guarda eventos y los de conteo de eventos, pues guardan la información necesaria para realizar análisis de potencia, temperatura y fiabilidad.

A.2.2.3. El Gestor de Comunicaciones

En el lado izquierdo de la Figura A.3 se aprecia la interfaz (flecha "Hacia/Desde el PC") que permite la conexión entre la FPGA y el PC. Se trata de un link bidireccional, puesto que, además de permitir hacer llegar las estadísticas desde la FPGA al PC, hace posible controlar la emulación desde éste último. Dicho mecanismo de control nos permite descargar un nuevo Sistema Emulado a la placa, controlar la evolución de la emulación (parar, continuar, resetear), gestionar el sistema de extracción de estadísticas (activar, desactivar, resetear) e, incluso, realizar tareas de depuración.

El único requerimiento para poder implementar el Gestor de Comunicaciones es la existencia de un medio que, físicamente, comunique la FPGA con el PC. Puede ser un puerto serie, un JTAG, un slot PCI, una conexión Ethernet, o una combinación de conexiones.

En el caso particular de mi implementación, para el sistema de comunicación FPGA-PC, he utilizado una conexión de Ethernet estándar. El Gestor de Comunicaciones, por tanto, contiene un módulo encargado de gestionar los paquetes de red, al que he denominado Gestor de Red, y que explico a continuación.

El Gestor de Red

El Gestor de Red es el elemento que maneja los detalles de bajo nivel de la comunicación FPGA-PC. A más alto nivel, se trabaja directamente con un buffer en donde se colocan los datos, y se da una señal de envío. Análogamente, cuando se reciben datos, estos son procesados automáticamente y colocados en un buffer. A continuación, se genera una interrupción para notificar al módulo superior.

En mi implementación (ver Figura A.6), he usado el módulo Ethernetlite de Xilinx, junto con un Microblaze para realizar el control, una memoria BRAM para los buffers, y un bus PLB para interconectar todo.

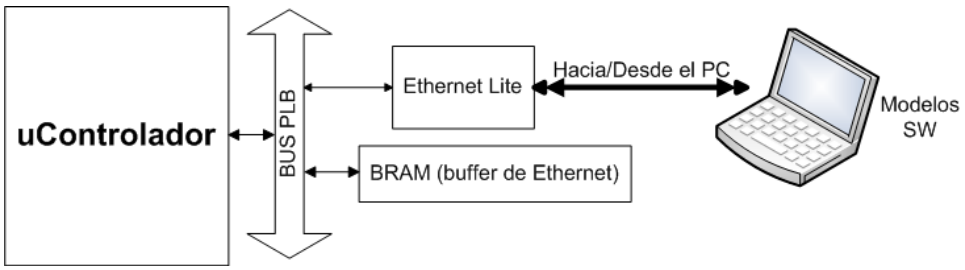


Figura A.6: Detalle de implementación del Gestor de Red.

El Gestor de Red se encarga, automáticamente, de encapsular la información intercambiada en paquetes Ethernet o MAC, dividiéndola, en caso necesario, en múltiples paquetes. Internamente, los datos van en un formato propio (ver Sección 2.2.3.1). Las estadísticas viajan en un sentido, mientras que en el otro van las temperaturas, la fiabilidad, etc... Los comandos de control viajan en ambos sentidos.

A.2.2.4. El Director del Motor de Emulación

En la Figura A.3 se aprecia (en el centro) el Director del Motor de Emulación, conectado al Subsistema de Extracción de Estadísticas, al Gestor de Comunicaciones, y al VPCM. Todos estos módulos intercambian información, entre ellos, y con el PC. En este escenario, con múltiples módulos intercambiando datos en tiempo real, es necesario un mecanismo de sincronización; esta es la tarea del Director del Motor de Emulación. Durante la emulación, continuamente recibe eventos, y genera respuestas, que requieren coordinar uno o varios componentes del Motor de Emulación.

Los diferentes eventos que suceden en la plataforma se pueden clasificar atendiendo a la fuente que los originó. Así pues, distingo entre eventos externos, comandos introducidos por el usuario, o eventos internos (como la saturación de la conexión FPGA-PC, o la expiración de un paso de emulación), originados en los propios elementos del Motor de Emulación.

La Tabla 2.1 ofrece la lista detallada de todos los comandos de control que acepta la plataforma. Entre ellos están las órdenes para iniciar, pausar, parar, resetear la emulación, o aquellas que gestionan el sistema de estadísticas (activar, desactivar, resetear...).

A.3. Los modelos SW de estimación

Tal y como se indicó al principio del capítulo, la PE tiene dos componentes: la plataforma HW de emulación (el HW que se instancia en la FPGA), descrita en la anterior sección, y las Bibliotecas SW de Estimación (el SW

que corre en un PC), que se tratan a continuación.

Los modelos SW de estimación son unas bibliotecas, implementadas en C++, que se ejecutan en un PC y reciben, en tiempo real, las estadísticas provenientes del Sistema Emulado. Como salida, calculan la potencia, temperatura, y fiabilidad del sistema final. Así pues, el funcionamiento de la PE consiste en emular durante un número predefinido de ciclos (paso de emulación), y detener el sistema para recoger las estadísticas de los buffers de la FPGA y enviarlas al PC; tras esto, la emulación continúa en un nuevo paso de emulación. En ocasiones, cuando se emplea lazo de realimentación, los números calculados por las bibliotecas SW son introducidos de nuevo en la plataforma antes de continuar con el siguiente paso de emulación (e.g., la temperatura calculada se introduce en los sensores de temperatura del Sistema Emulado).

La Figura A.7 muestra los interfaces de los distintos modelos de estimación (de potencia, temperatura y fiabilidad) y su conexión con la FPGA.

A.3.1. Estadísticas del Sistema

En la PE, el término Estadísticas del Sistema hace referencia a toda la información recolectada del Sistema Emulado en tiempo de ejecución. Esto comprende las frecuencias y voltajes del sistema, así como las Estadísticas de Actividad; un log exhaustivo de todos los eventos de interés que ocurren en la plataforma, recogidos en tiempo real por los *sniffers* que monitorizan las señales de los cores cada ciclo (la siguiente sección muestra ejemplos de Estadísticas de Actividad).

La información extraída de la FPGA es enviada al PC, donde puede ser simplemente almacenada para posteriores análisis, o procesada mediante scripts para obtener la información deseada, un resumen de la misma, etc...

A.3.2. Estimación de potencia

El consumo de potencia de los diferentes elementos que forman un MPSoC es a menudo caracterizado por los propios fabricantes de chips. Dependiendo de las características del IP core en particular, se nos facilitará la media de consumo, valores mínimos y máximos (dependientes de la actividad del core), o estados de consumo (e.g., durmiendo/activo). Estos valores dependen de la tecnología de fabricación, de la frecuencia, el voltaje, y la temperatura actual del circuito, de manera que vienen indicados en tablas que podemos acceder con los parámetros actuales. Por otro lado, tenemos que en la PE, gracias a los sniffers, podemos hacer un registro exhaustivo de todos los eventos que ocurren, desde la actividad de las señales, hasta los eventos de alto nivel (e.g. fallos de caché). Por tanto, generar el consumo estimado de un componente a partir de estos datos es bastante directo. Con tal objetivo, he desarrollado una biblioteca en C++ que estima la po-

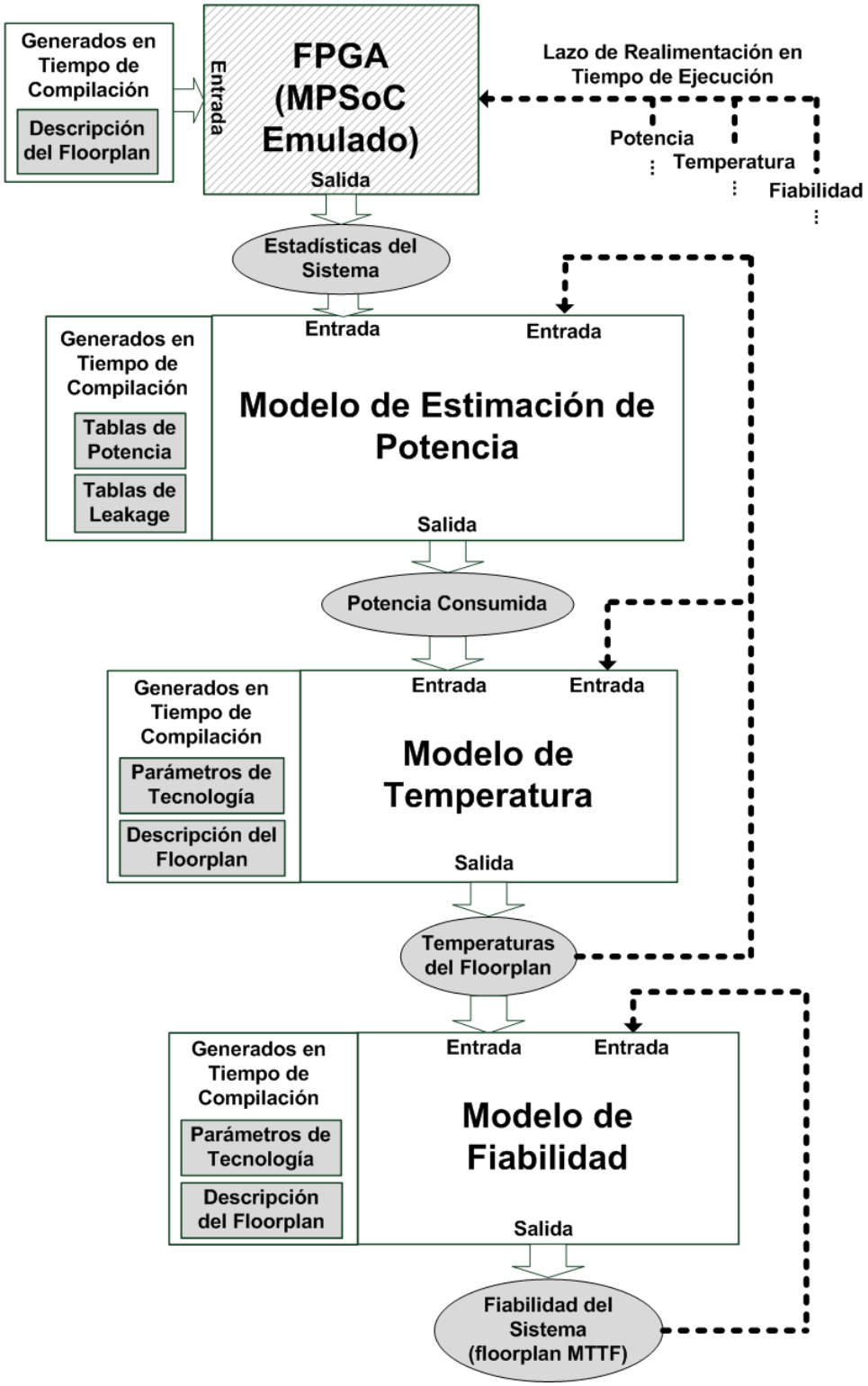


Figura A.7: Interfaces de las Bibliotecas SW de Estimación.

tencia consumida en el Sistema Emulado realizando los cálculos indicados anteriormente. La he denominado *Modelo de Estimación de Potencia*.

Tal y como se indica en [SSS+04], la contribución al consumo debida a la corriente de leakage es de vital relevancia en las nuevas generaciones de chips. Por este motivo, en mi *Modelo de Estimación de Potencia*, los cálculos de consumo lo tienen en cuenta; en concreto, la corriente de leakage se ha modelado como un incremento de un tanto por ciento del total de la potencia consumida. Dicho porcentaje también viene dado en tablas, que dependen de los mismos parámetros que las tablas de potencia.

La interfaz del modelo se puede observar en la Figura A.7. Previamente a la ejecución, en tiempo de compilación (etapa de configuración), el usuario ha de introducir información acerca del Sistema Emulado (definición de todos los componentes del sistema, con sus tablas de consumo y de leakage, que dependerán de la tecnología usada). En tiempo de ejecución, como entrada recibe las Estadísticas del Sistema (ya sea desde una traza predefinida, o desde la FPGA), junto con la temperatura de cada elemento que está siendo observado (que puede venir de una traza predefinida, o de la salida del modelo térmico, ver Sección A.3.3); como salida, el modelo calcula el consumo de potencia de cada elemento del sistema.

A.3.3. Modelado térmico en 2D

El Modelo de Temperatura es otra biblioteca SW, que se encarga de estimar temperaturas a partir de los números de potencia. Este procedimiento es un poco más complejo que el cálculo de potencia; por ello, necesitamos algo más de información que en el caso anterior: En tiempo de compilación, configuramos el modelo con el tamaño y la ubicación de todos los componentes del sistema (el layout), la tecnología, y el empaquetado. En tiempo de ejecución, es necesario el consumo de potencia de los elementos del sistema, que depende de la frecuencia, el voltaje, la temperatura, y la actividad.

Como podemos ver en la Figura A.7, la temperatura depende del consumo de potencia y, la potencia, a su vez, de la temperatura del sistema. Por esta razón, los modelos de potencia y temperatura han de trabajar conjuntamente, para mantener la exactitud en los cálculos. Tanto el cálculo de la potencia, como el de la temperatura, se realizan en pequeños pasos de emulación; esto es, el tiempo de emulación se discretiza, de manera que una llamada al modelo térmico devuelve la temperatura en el momento i. Dado que la temperatura en el momento i+1 depende de la temperatura en el momento i, la temperatura calculada se introduce de nuevo como entrada al modelo para la próxima iteración.

A continuación, paso a detallar el modelo matemático que, internamente, utiliza la biblioteca térmica para calcular cómo se va propagando el calor desde las capas inferiores del chip hasta que se elimina por conveccion en el

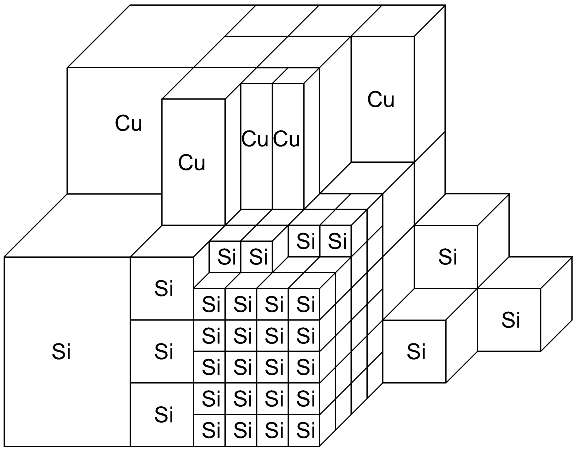


Figura A.8: Esquema de un chip dividido en celdas regulares de diferentes tamaños.

aire.

En primer lugar, el chip (considerado como un bloque de silicio envuelto en su empaquetado, colocado sobre un PCB, y con un disipador en su parte superior) se discretiza, dividiéndolo en pequeñas celdas (cubos). La división en celdas nada tiene que ver con la anterior división en componentes que hicimos con el floorplan. Una celda puede equivaler a un componente (e.g.: core, una subunidad funcional, etc...), o un componente puede estar formado por muchas celdas, de la misma manera que una celda puede comprender muchos componentes. Como veremos más adelante, el tamaño de las celdas dependerá de la exactitud que queramos tener en el modelado.

La Figura A.8 muestra el esquema de un chip dividido en celdas. Aprovechando que la forma en que se propaga el calor en un medio físico se puede equiparar a cómo se propaga la corriente en un circuito eléctrico de tipo RC, he elaborado un modelo equivalente que es mucho más eficiente en términos de tiempo de cómputo.

De esta manera, he modelado cada una de las celdas en que he dividido el sistema mediante seis resistencias y un condensador (ver Figura A.9). El condensador representa el calor (corriente) almacenado en esa celda, mientras que las resistencias representan la facilidad (o resistencia) de esa celda a perder calor (corriente) por cada una de sus seis caras.

La generación de calor se debe a la actividad de las celdas; esto es, de las unidades funcionales que ocupan el lugar de las celdas. Por tanto, las celdas activas (en oposición a las que son sólo pasivas) contienen también una fuente de corriente para "inyectar calor" (representada entre las resistencias *top* y *west* de la Figura A.9). A partir del valor de dicha fuente, y de los valores del condensador y de las resistencias asociadas, se determina la propagación

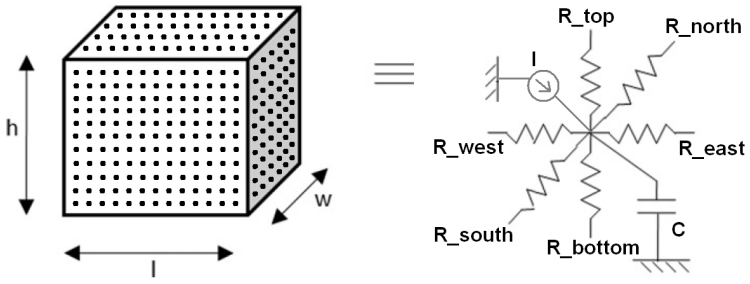


Figura A.9: Circuito RC equivalente para una celda activa.

de, y hacia, los vecinos.

Durante la etapa de configuración del modelo, el diseñador especifica el tamaño de las diferentes celdas que componen el sistema (la resolución espacial), y la tecnología con la que se fabricará el chip; i.e., la capacitancia térmica de los materiales que lo componen, incluyendo los parámetros de empaquetado (e.g.: calidad del disipador). Estos datos se traducirán en unos valores de R y C para las diferentes celdas. El valor de las fuentes de corriente de las celdas activas, en cambio, varía en tiempo de ejecución, y dependerá del consumo de potencia de la unidad funcional modelada por cada celda.

La disipación con el aire ambiental se modela mediante una resistencia conectada en serie con las que ocupan la capa superior del chip. De manera similar, la difusión que ocurre desde el IC hasta el empaquetado (tanto lateralmente, como hacia abajo), se modela incrementando el valor de las resistencias limítrofes.

El comportamiento del circuito RC resultante se puede expresar con el siguiente sistema de ecuaciones:

$$G \cdot X(t) + C \cdot \dot{X}(t) = B \cdot U(t), \tag{A.1}$$

Donde X(t) es el vector de temperaturas de las celdas del circuito en el instante t, G y C son las matrices de conductancia y capacitancia del circuito, U(t) es el vector de corriente (calor) entrante al circuito, y B es una matriz de selección.

El sistema de ecuaciones A.1 se puede simplificar en nuestro caso particular pues, en el modelo térmico, las temperaturas se actualizan en pequeños pasos de emulación, dentro de los cuales consideramos que las propiedades del circuito no varían. La nueva ecuación resultante, que describe la respuesta del circuito en estado estable, es la Ecuación A.2

$$G \cdot X = B \cdot U \tag{A.2}$$

que, al no ser lineal, resuelvo aplicando el método de Euler. El proce-

dimiento consiste, básicamente, en realizar una estimación del valor inicial de la matriz X, resolver las ecuaciones para el actual paso de emulación, y calcular el error cometido. Si es menor que un límite preestablecido, significa que las temperaturas convergen. En otro caso, debo iterar el proceso, corrigiendo el valor estimado. En la mayoría de los casos estudiados, 5 o 6 iteraciones fueron suficientes para alcanzar la convergencia con un error de 10^{-6}.

De la descripción del modelo se desprende que, variando el tamaño y el número de celdas, podemos ajustar la exactitud de los cálculos. Cuanto más pequeñas sean las celdas, más exactos serán los cálculos, a costa de invertir más tiempo en los mismos.

A.3.4. Modelado de fiabilidad

Se trata de una biblioteca SW que analiza la influencia de la temperatura en la fiabilidad del sistema mediante el uso de varios modelos matemáticos que permiten estimar el tiempo medio de fallo de cada uno de los componentes. Los efectos incluídos son la electromigración, la ruptura del dieléctrico, la migración por estrés, y los ciclos térmicos. Algunos de ellos son reversibles, mientras que otros son de carácter permanente.

Desde el punto de vista de la implementación, el modelo de fiabilidad sigue la misma estructura que la biblioteca térmica: la fiabilidad se actualiza en pequeños incrementos (pasos de emulación). De esta manera (ver interfaz del modelo en la Figura A.7), las temperaturas calculadas por el modelo térmico se pasan, como entrada, al modelo de fiabilidad, que predice el tiempo medio de fallo de los componentes del sistema en función de la historia del chip (fiabilidad "acumulada"), de las temperaturas actuales, y de un conjunto de constantes (tecnológicas) fijadas en tiempo de diseño. Las fórmulas detalladas se pueden consultar en [CSM+06; SABR05; Sem00].

A la hora de estimar la fiabilidad de un sistema, debemos de tener en cuenta que los fallos en el funcionamiento de un chip aparecen al cabo de los años. Si procediéramos estrictamente, deberíamos emular ese tiempo para poder dar números exactos de fiabilidad; sin embargo, normalmente los fabricantes necesitan una estimación del tiempo esperado de vida (funcionamiento correcto) del chip en el escenario peor. En este caso, lo que hacemos es simular durante un tiempo mucho más reducido, y extrapolar la tendencia observada al número deseado de años vista.

A.3.5. Modelado térmico en 3D

La tecnología de apilado 3D es una innovadora técnica de fabricación que permite diseñar un chip en tres dimensiones mediante el apilamiento de varias obleas de silicio, una encima de otra, intercomunicadas mediante una serie de vías-a-través-del-silicio (TSVs, por sus siglas en inglés).

Por un lado, esta solución incrementa las posibilidades de integración en-chip pero, por otro, también aumenta sustancialmente la densidad de potencia y, con ello, los problemas derivados de la aparición de puntos calientes. Sin embargo, la existencia de esta tercera dimensión nos ofrece un espacio de exploración más grande, que propicia la aparición de nuevas metodologías para solventar los problemas de temperatura, como el emplazado inteligente de los componentes en el mapa 3D, o el uso de refrigeración líquida (por microcanales) entre las capas del chip.

Con el objetivo de estudiar este tipo de sistemas, así como las múltiples posibilidades que ofrecen para optimizaciones, he integrado en la PE un modelo para caracterizar el comportamiento térmico de MPSoCs 3D fabricados con la tecnología de apilamiento. Internamente, se trata de extender el modelo RC desarrollado para el caso 2D, explicado en la Sección A.3.3, para tener en cuenta el efecto de las TSVs y de los microcanales de la refrigeración líquida activa. Esto se ha conseguido mediante dos modificaciones: (i) se ha añadido un nuevo material, el material entre-capas, cuyas características térmicas (resistencias y condensadores equivalentes) se calculan teniendo en cuenta no sólo la tecnología usada, sino también la densidad de TSVs y los microcanales presentes en el material entre-capas; y (ii) el modelo térmico ha sido modificado para que los valores de las resistencias puedan variar en tiempo de ejecución, y reflejar así la acción del líquido refrigerante, cuyo flujo puede ser regulado bajo demanda.

A.4. El flujo de emulación

Las ventajas fundamentales del entorno de emulación presentado, frente a otros que también permiten realizar exploraciones de diseños MPSoC, son dos: (i) se trata de un entorno combinado, que usa una FPGA para modelar los componentes a velocidades de megahercios y extraer detalladas estadísticas en tiempo real, mientras que, en paralelo, estas estadísticas son introducidas en un modelo SW, que se ejecuta en un PC, y calcula la potencia, temperatura y fiabilidad del sistema emulado; y (ii) todo está integrado en un único flujo de trabajo, lo que simplifica en gran medida la tarea del diseñador.

La Figura A.10 muestra el flujo de trabajo con la PE. En primer lugar, configuramos la FPGA y el PC (fases 1 y 2 de la figura). A continuación, en la fase 3, comienza la emulación. Detallo a continuación los pasos:

1. Se definen los elementos que serán alojados en la FPGA. Esto incluye los componentes HW (arquitectura) y SW (aplicaciones a ejecutar) del Sistema Emulado; así como la infraestructura del Sistema de Emulación (indicando los elementos a monitorizar, número y tipo de *sniffers*, etc.). Tras los procesos de síntesis (HW) y compilación (SW), obtenemos los

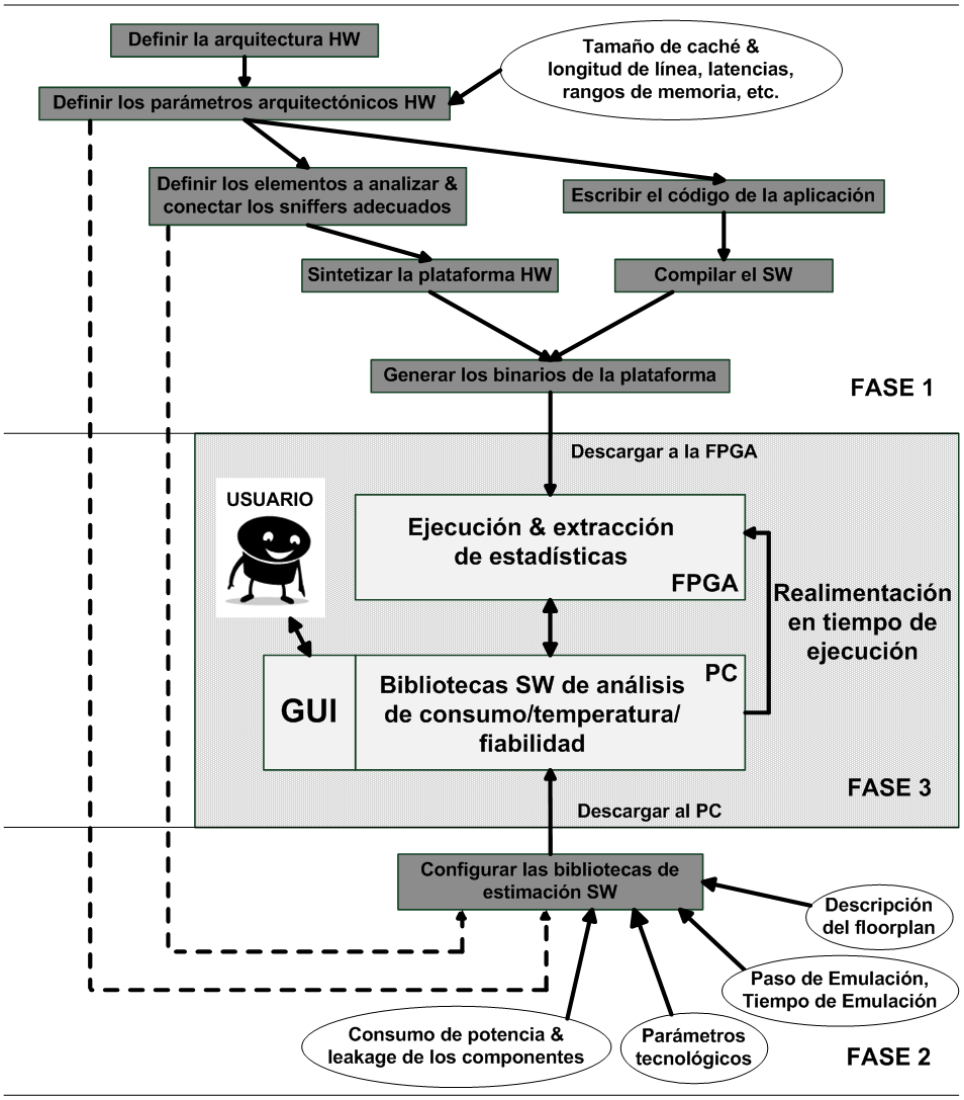


Figura A.10: Flujo de diseño HW/SW con la Plataforma de Emulación.

binarios que contienen la plataforma.

2. Se configuran las Bibliotecas SW de Estimación que correrán en el PC. Para ello necesitamos introducir los datos necesarios (ver Sección A.3), como la tecnología usada, el floorplan del sistema, las tablas de consumo de potencia, de leakage, etc... En esta fase se definen también la resolución a la que trabajará el modelo térmico (tamaño de las celdas), y la duración del paso de emulación.

3. El sistema está completamente especificado. Descargamos los binarios generados (en la fase 1) a la FPGA y, en el PC, damos la orden de comienzo (el PC ofrece una interfaz gráfica que permite controlar, en todo momento, el proceso de emulación). La emulación comenzará a funcionar de manera sincronizada y autónoma: Las estadísticas del MPSoC emulado llegan al PC, en donde se usan como entrada a los modelos de estimación, que calculan la potencia, temperatura y/o fiabilidad del sistema final. En caso de que así se desee, estos valores se pueden devolver a la FPGA, de manera que sean accesibles desde el propio Sistema Emulado (ya sea desde el HW, o desde el Sistema Operativo), que podrá utilizarlos para elaborar políticas de gestión de recursos.

A.4.1. Requisitos: FPGAs, PCs, y herramientas

La PE ha sido diseñada del modo más genérico posible, evitando depender de una herramienta, placa, o PC de un determinado fabricante. Tanto el Motor de Emulación como el Sistema Emulado están especificados en VHDL estándar y parametrizable, de manera que pueden ser utilizados en cualquier FPGA. Normalmente, el fabricante de la misma provee una herramienta para generar los binarios a partir del VHDL (y de los archivos fuente del SW que se ejecutará en los cores). El único requerimiento, por tanto, es que la placa tenga conectividad para poder comunicarla con el PC (e.g.: a través de un puerto Ethernet, PCI, etc...).

A lo largo de este trabajo de investigación, he utilizado varias FPGAs de Xilinx. En la Sección 4.3, he incluido varios ejemplos de uso de la PE, que dan una idea aproximada del tamaño de los MPSoCs que se pueden instanciar dentro de distintos modelos de FPGA. Como plataforma principal, he elegido la Virtex 2 Pro vp30 board, con 3M de puertas, dos PowerPC empotrados, memorias SRAM y DDR, y puerto de Ethernet, que tiene un coste de alrededor de 2,000 dólares en el mercado, y puede acomodar un core complejo, como el Leon3, junto con el sistema de emulación, en el 50 % de sus recursos.

El fabricante, Xilinx, provee las herramientas Embedded Development Kit e Integrated Design Environment que, junto con la herramienta de simulación Modelsim (de Mentor Graphics), han sido las utilizadas para el desarrollo.

En cuanto a los modelos SW, las bibliotecas que se ejecutan en el PC han sido escritas en C++. Por tanto, podemos emplear cualquier compilador estándar, como G++, para generar los ejecutables. En mi caso, he usado el entorno Visual Studio Suite, de Microsoft, para escribir, compilar, y depurar el código.

Por último, tampoco hay requerimientos específicos en cuanto al PC sobre el que corren los modelos de estimación. En mis experimentos, he utilizado siempre un PC estándar (desde un Pentium 4 con 256 MB de RAM), y fue suficiente para hacer funcionar la plataforma a plena velocidad (con la FPGA trabajando a 100MHz). De hecho, tal y como explico en la Sección A.6, las únicas pausas observadas se debieron a las limitaciones del ancho de banda del puerto de comunicaciones.

A.5. Experimentos

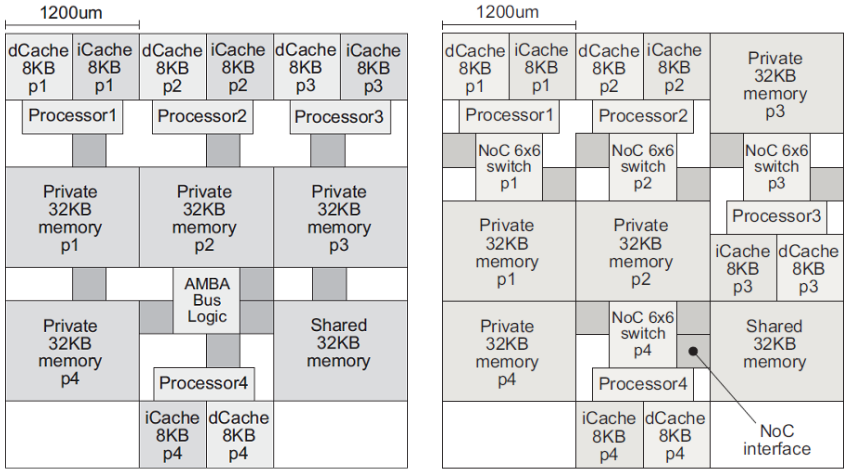
En esta sección, presento tres casos de estudio dirigidos a ilustrar el uso práctico de la PE para evaluar el impacto que las decisiones de diseño (desde el layout del floorplan, a la selección del compilador) tienen sobre el rendimiento, la temperatura, o la fiabilidad del MPSoC final.

A.5.1. Exploración de las características térmicas

En este primer experimento, aplico el entorno de emulación a la fase de diseño de un MPSoC que contiene 4 cores RISC, con el objeto de estudiar su comportamiento térmico bajo diferentes configuraciones.

El Sistema Emulado contiene 4 procesadores ARM7, cada uno conectado a dos cachés locales con mapeo directo y escritura directa, de 8KB cada una, y a una memoria privada de 32KB. Por último, existe otra memoria compartida entre todos, también de 32KB. Tal y como se muestra en la Figura A.11, las memorias y los procesadores están conectados, bien mediante un bus AMBA, Figura A.11a, o mediante una simple NoC (creada usando XPipes [JMBDM08], con cuatro switches de 6x6, e interfaces de red (módulos NI)), Figura A.11b, lo que da lugar a dos floorplans diferentes, ambos diseñados con tecnología de 0.13 μm. Los ARM7 pueden funcionar hasta 500MHz, y las interconexiones funcionan siempre a la misma frecuencia que los cores. Cada componente presente en la Figura A.11 contiene un sniffer asociado que monitoriza la actividad de ese módulo en particular.

En cuanto a las aplicaciones SW, he diseñado un programa, MATRIX, que realiza multiplicaciones de matrices de manera colaborativa entre cores; un filtro de dithering, DITHERING, que aplica el algoritmo de Floyd [FS85] sobre dos imágenes de 128x128 y, finalmente, la aplicación MATRIX-TM, que impone en todos los procesadores una carga cercana al 100 % para permitir observar fácilmente los efectos en la temperatura.



a) Bus AMBA (b) NoC

Figura A.11: Dos soluciones de interconexión diferentes para la arquitectura básica del caso de estudio.

Ambos floorplans considerados en la Figura A.11 han sido divididos en 128 celdas térmicas, de tamaño $150um * 150um$ cada una. En la Tabla A.1 enumero las propiedades térmicas de los materiales usados en los experimentos. Como valor por defecto para la resistencia empaquetado-ambiente, tomo el valor de 40KW/K, que se corresponde con un empaquetado económico típico para sistemas empotrados [BEAE01].

Por último, describo brevemente el entorno del MPARM, que ha sido el simulador usado en varios resultados como punto de referencia contra el que comparar la PE. El MPARM [BBB+05] es un simulador SW, escrito en SystemC, que permite modelar MPSoCs con una resolución a nivel de ciclo; no sólo el HW, sino también el SW: desde simples aplicaciones, hasta Sistemas Operativos multiprocesador. Soporta multitud de componentes

Tabla A.1: Propiedades térmicas de los materiales utilizados en los experimentos.

conductividad térmica del Silicio	$150 \cdot \left(\frac{300}{T}\right)^{4/3} W/mK$
calor específico del Silicio	$1.628e-12 J/um^3K$
grosor del Silicio	$350um$
conductividad térmica del Cobre	$400W/mK$
calor específico del Cobre	$3.55e-12 J/um^3K$
grosor del Cobre	$1,000um$
conductividad empaquetado-ambiente	$40K/W$ (bajo coste)
resistividad eléctrica del Aluminio	2.82×10^{-8} (1+0.0039$\triangle$T)Ωm, $\triangle$T = T-293.15K

Tabla A.2: Comparaciones de tiempo entre la Plataforma de Emulación y el simulador MPARM.

Benchmark	MPARM	PE	Speed-Up
Matrix (1 core)	106 seg	1.2 seg	88×
Matrix (4 cores)	5 min 23 seg	1.2 seg	269×
Matrix (8 cores)	13 min 17 seg	1.2 seg	664×
Dithering (4 cores-bus)	2 min 35 seg	0.18 seg	861×
Dithering (4 cores-NoC)	3 min 15 seg	0.17 seg	1,147×
Matrix-TM (4 cores-NoC)	2 días	5'02 seg	1,612×

HW y sistemas heterogéneos, y puede conectarse a bibliotecas térmicas, y demás herramientas de otros fabricantes, para ampliar sus posibilidades. Por ejemplo, *XpipesCompiler* [JMBDM08] y *Sunfloor* [SMBDM09], para el diseño de NoCs y de floorplans, respectivamente. En todos los experimentos, el MPARM se ha ejecutado en un Pentium 4, a 3.0GHz, con 1GB de SDRAM, y ejecutando GNU/Linux 2.6.

A.5.1.1. Arquitecturas MPSoC: Simulación contra emulación

Con objeto de estudiar el rendimiento de la PE, he evaluado varias configuraciones del MPSoC emulado: con interconexión basada en bus y basada en NoC, variando el número de procesadores (de 1 a 8), y con distintas aplicaciones SW (Matrix, Dithering y Matrix-TM). Como ejemplo, el MPSoC con bus y 4 procesadores (i.e., aquel de la Figura A.11a), consume el 66 % de la FPGA V2VP30, y se ejecuta a 100MHz.

Los resultados se muestran en la Tabla A.2. Los tiempos obtenidos indican cómo el entorno HW/SW de emulación escala mucho mejor que el simulador SW. De hecho, la exploración del MPSoC con 8 cores llevó 1.2 segundos en la PE, pero más de 13 minutos en el MPARM (a 125KHz), lo que significa una mejora de 664×. Además, la exploración de NoCs muestra aún mayores mejoras (1,147×), debido a la sobrecarga que tiene el simulador SW para gestionar las diferentes señales en paralelo.

A.5.1.2. Modelado térmico a nivel de ciclo de MPSoCs

Con el objetivo de estudiar la evolución de la temperatura en el MPSoC, ejecuté 100K iteraciones del programa Matrix-TM, con el sistema corriendo a 500MHz. Los resultados, Figura A.12, demuestran la necesidad de realizar largas emulaciones para poder apreciar los efectos térmicos dentro del chip: la PE tardó 5 minutos en emular la aplicación corriendo sobre el MPSoC, incluidos los cálculos de temperatura, mientras que el MPARM tardó dos días para simular los 0.18 segundos de ejecución (representados en el óvalo de la esquina inferior-izquierda de la Figura A.12). La simulación en MPARM,

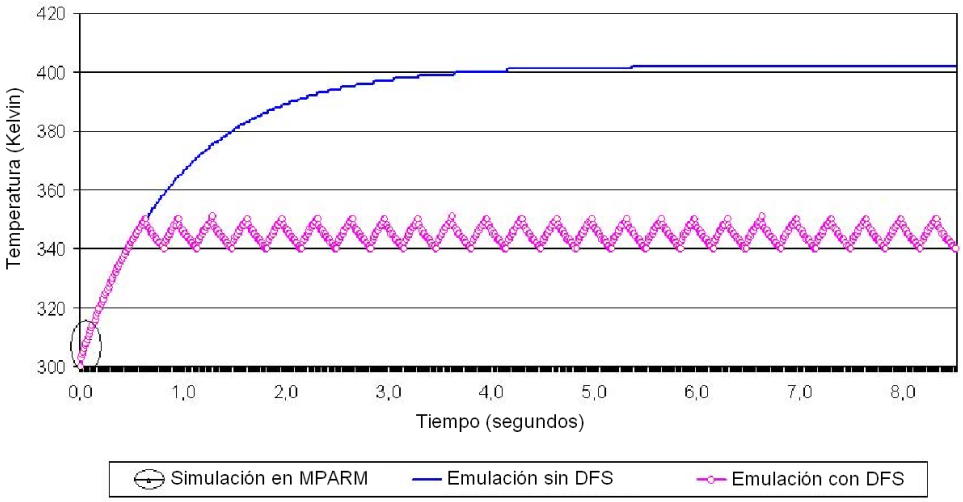


Figura A.12: Evolución de la temperatura con y sin DFS.

por tanto, representa sólo una pequeña parte del comportamiento térmico del MPSoC.

Debido a las altas temperaturas observadas en este diseño, realicé unas pequeñas modificaciones en el sistema de cara a poder explorar los posibles beneficios de aplicar técnicas sencillas de control de temperatura. En particular, modifiqué el módulo VPCM para permitir cambiar la frecuencia del sistema en tiempo de ejecución (i.e., realizar escalado dinámico de la frecuencia, o DFS). El control es puramente HW: si el VPCM, a través de los sensores de temperatura presentes en el sistema, detecta que se ha superado un cierto límite, reduce la frecuencia de 500 a 100 MHz. En cuanto se vuelve a una temperatura segura, se elimina esta limitación. En este ejemplo he usado los límites de 350 y 340 Kelvin, respectivamente.

Los resultados obtenidos empleando DFS se muestran en la Figura A.12 (traza *Emulación con DFS*), e indican que esta simple política de gestión de temperatura podría ser altamente beneficiosa para diseños de MPSoCs que usan empaquetado de bajo coste. Además, demuestran la conveniencia de usar herramientas como la PE, en lugar de simuladores SW, para realizar exploraciones rápidas y detalladas del comportamiento térmico de los MPSoCs.

A.5.1.3. Exploración para la selección de floorplan

Tal y como expliqué en la introducción, una vez seleccionados los componentes que formarán un determinado MPSoC, quedan aún muchas decisiones por tomar como, por ejemplo, el lugar que ocupará cada uno de ellos en el floorplan. El siguiente experimento va destinado a estudiar el comportamiento térmico de distintos floorplans alternativos para una misma arquitectura

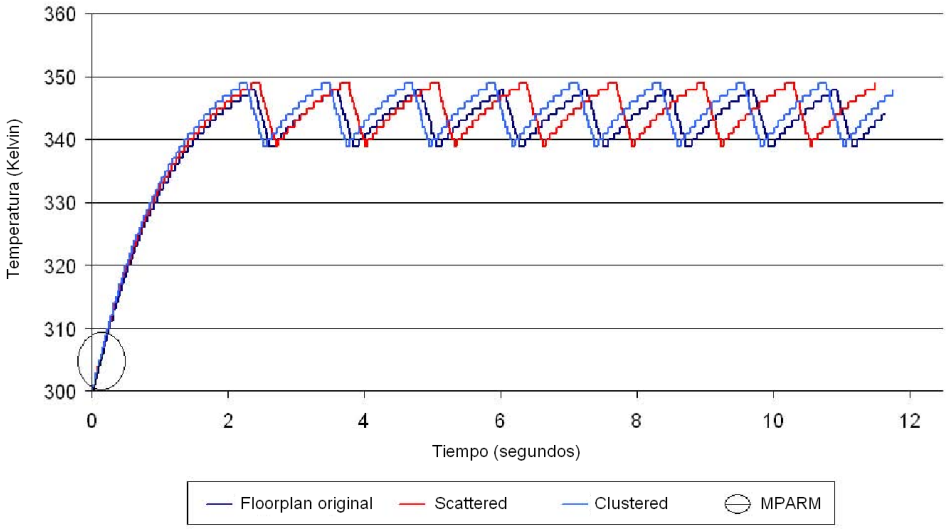


Figura A.13: Evolución de la temperatura para diferentes floorplans, con el sistema ejecutando Matrix-TM a 500 MHz, con DFS.

base; en concreto, aquella de la Figura A.11b, que contiene 4 cores, interconexión NoC, y que corre a 500MHz. Junto al floorplan original, he considerado otras dos alternativas: con los cores concentrados en el centro del chip, y con los cores dispersos en los extremos.

Los resultados se muestran en la Figura A.13. El mejor floorplan para minimizar las temperaturas es el que tiene los cores dispersos (Scattered) (un 15 % menos de calentamiento, de media) que, además, retrasa la necesidad de aplicar DFS. Como contrapartida, se observó que sus interconexiones se calientan más debido a su mayor longitud, que puede dar lugar a congestiones en la NoC. La peor distribución, la que concentra los cores en el centro (Clustered), se calienta tan sólo un 5 % más que la solución inicial, diseñada a mano, y que presenta las interconexiones más cortas. Por tanto, este estudio demuestra la necesidad de explorar diferentes soluciones arquitectónicas antes de poder decidirnos por una que, a priori, pudiera parecer más ventajosa.

A.5.1.4. Exploración para la selección de empaquetado

La Figura A.14 muestra los diferentes perfiles térmicos que presenta el MPSoC base, con el floorplan de la Figura A.11b, para distintas soluciones de empaquetado: el de bajo coste (45KW/W), el estándar (12 KW/W), y el de alto coste (5KW/W).

En el caso del empaquetado estándar, el MPSoC alcanzó una temperatura máxima de 360 Kelvin, cuando no se aplicó DFS; mientras que, con el empaquetado barato, subió hasta los 500 Kelvin. Sin embargo, ambas so-

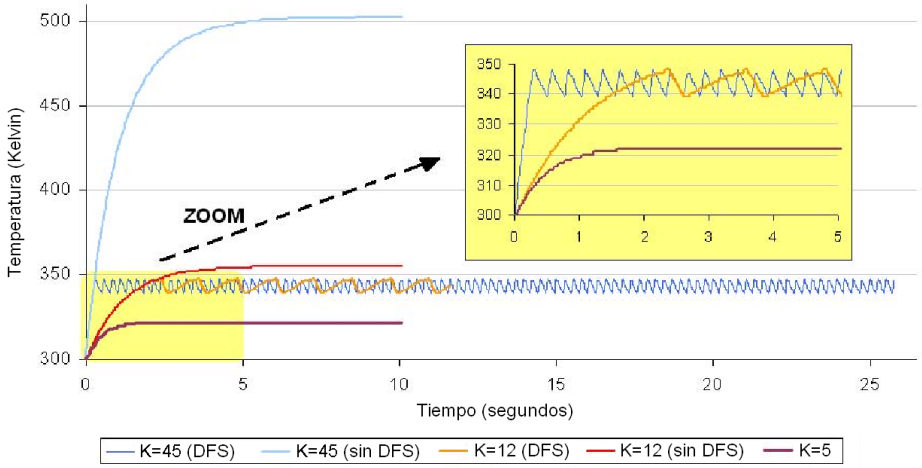


Figura A.14: Evolución de la temperatura para tres soluciones de empaquetado diferentes: de bajo coste, estándar, y de alto coste.

luciones presentan un comportamiento parecido al activar la estrategia de DFS (para valores del umbral de 350 y 340 Kelvin). Por tanto, en este caso no hay mejoras significativas, y sería preferible la solución barata. El comportamiento térmico con el empaquetado de alto coste, por el contrario, es totalmente diferente: el chip nunca supera los 325 Kelvin. Luego, ésta solución, además de no requerir DFS, alargaría el tiempo de vida del sistema, y sería muy interesante para aplicar en versiones del chip de alta fiabilidad. Como contrapartida, tendríamos el sobreprecio del sistema final, que será entre 5 y 12 veces más que aquel que usa empaquetado estándar, y hasta 20 veces más que la solución con empaquetado de bajo coste.

Los resultados de este experimento indican los beneficios de realizar un detallado análisis térmico, considerando distintas tecnologías de empaquetado, durante la fase de diseño de un MPSoC. Dicho estudio nos puede permitir ahorrarnos un empaquetado más caro que apenas presenta ventajas o, por el contrario, el implementar el mecanismo de DFS cuando no resulta necesario. La decisión adecuada, en cualquier caso, dependerá de las restricciones que tengamos en nuestro diseño particular.

A.5.2. Entorno de exploración de fiabilidad

En este segundo conjunto de experimentos, he aplicado la PE a un core complejo, un procesador Leon3, con el objetivo de estudiar cómo las modificaciones en el compilador pueden afectar a la temperatura observada a nivel de microarquitectura; en concreto, al banco de registros.

El Leon3 [Gaib] es una CPU de 32 bits con arquitectura Sparc-V8 que se utiliza para aplicaciones empotradas; tiene una arquitectura similar a los

cores comerciales, con cachés separadas de instrucciones y datos, unidad de gestión de memoria, buffer de traducción anticipada... y puede ser extendido a una configuración multiprocesador. El banco de registros presenta la típica estructura basada en ventanas de los Sparc [Inc], y puede ser configurado con un número variable de registros, que va desde los 40 a los 520. En la página web del fabricante [Gaia], Gaisler Research Inc., está disponible una versión sintetizable del core, que incluye todo el código fuente para poder realizar modificaciones, así como las herramientas necesarias para completar el desarrollo HW y SW.

El Sistema Emulado utilizado en este experimento consta de un Leon3 con 256 registros, de tres puertos cada uno (dos de lectura y uno de escritura), y organizados físicamente en una estructura regular de 32 filas y 8 columnas; tiene una memoria SDRAM, cachés de instrucciones y datos asociativas por conjuntos (de 16KB y con cuatro vías), y TLBs independientes de 32 entradas cada uno. La política de reemplazo es LRU. Además, el sistema incluye 64KB de ROM y RAM, 512 MB de memoria DDR, buses AMBA, un temporizador, y el controlador de interrupciones. Por último, una interfaz serie permite comunicarse y depurar el procesador.

Cada registro tiene un sniffer asociado que monitoriza su funcionamiento. El Motor de Emulación extrae los datos y los envía al PC cada 10ms, para calcular la potencia consumida y estimar la temperatura y el Tiempo Medio de Fallo (MTTF, por sus siglas en inglés) de cada registro, haciendo uso de los modelos SW de estimación de potencia, temperatura y fiabilidad, respectivamente.

Las Bibliotecas SW de Estimación han sido configuradas para modelar un banco de registros implementado con una tecnología de 90nm, dividido en 256 celdas térmicas, una por registro (organizadas, por tanto, en una rejilla regular de 32 filas y 8 columnas). Cada celda mide 300μm $\times$ 300 μm, y las características térmicas de los materiales considerados son las representadas en la Tabla A.1. De cara a analizar el caso peor, el banco de registros se modela rodeado de celdas con una temperatura cercana al punto caliente (establecido en 328 Kelvin): 318 Kelvin. El exterior de estas celdas es el aire ambiente.

En cuanto al SW que corre en el Leon3, se ejecutan un subconjunto de aplicaciones tomadas de los benchmarks MiBench [GRE+01] y CommBench [WF00], compiladas con el GCC 3.2.3 para arquitectura Sparc usando distintos niveles de optimización (ver figuras). Los resultados muestran la fiabilidad a tres años vista.

A.5.2.1. Elaboración de la política de mejora de la fiabilidad

De cara a determinar los factores que afectan a la fiabilidad del banco de registros, se ha llevado a cabo un estudio inicial, cuyos resultados sintetizo a continuación (las gráficas muestran el porcentaje de degradación del MTTF

inicialmente previsto por el fabricante):

En primer lugar, la Figura A.15 nos permite ver que los benchmarks que peores valores de fiabilidad arrojan son aquellos que hacen uso intensivo de un número reducido de registros, como *FFT* y *bitcount*, que analizaremos más detalladamente.

Así pues, por un lado, la Figura A.16 muestra los resultados obtenidos tras recompilar la aplicación *FFT* usando distintas opciones de compilación (desde -O0 hasta -O3): un mayor grado de optimización favorece el reuso de registros y, por tanto, la aparición de puntos calientes, haciendo disminuir la fiabilidad (hasta un 3 %, en el caso de -O3). Por otro lado, una tercera gráfica, Figura A.17, muestra en detalle los factores que contribuyen al valor final del MTTF, cuando *FFT* es compilado con -O3; siendo SM el factor dominante.

Finalmente, la Figura A.18 muestra el número de registros dañados (consideramos como tales aquellos con un MTTF por debajo del 2 % del valor nominal), al cabo de 2 años, para el benchmark *bitcount*: varía entre 1 y 4, dependiendo del nivel de optimización usado por el compilador.

A la vista de estos resultados, he redefinido la política de asignación de registros que hace el compilador GCC, que asigna los registros de una lista de registros libres [bib03]. En lugar de ello, mi política selecciona un registro tras comprobar previamente que sus vecinos no han sido asignados, siempre y cuando sea posible. De esta manera, se busca crear un patrón similar a un tablero de ajedrez, que facilite la difusión del calor. La Figura A.19 muestra el mapa térmico instantáneo del banco de registros, donde se observa cómo la nueva política contribuye a un mejor balance térmico, reduciendo eficazmente el número de puntos calientes.

Reanalizo, a continuación, dos de las gráficas anteriores en las que se muestran los beneficios de esta nueva política (con el nombre de *MODIFIED*): En la Figura A.18, se aprecia la reducción en el número de registros dañados; de hecho, con este benchmark (*bitcount*), se consigue que no haya ningún registro dañado al cabo de dos años. En la Figura A.16 se ve, más en detalle, que esta política es muy eficaz para minimizar la degradación del MTTF; se reduce tan sólo un 0.2 % en el intervalo representado.

A.5.3. Políticas de gestión térmica a nivel de sistema

En este experimento muestro un ejemplo de cómo se puede gestionar la temperatura en un entorno multiprocesador a nivel de Sistema Operativo (OS); elaboro, implemento, y aplico una política de gestión térmica de MPSoCs basada en la migración de tareas en tiempo real.

La arquitectura HW del *entorno con Sistema Operativo Multi-Procesador (MPOS) para emulación con retroalimentación térmica* consta de los siguientes componentes:

- El Sistema Emulado contiene un número configurable de procesadores *soft-cores*, que ejecutan uClinux, y se comunican y sincronizan a través de una memoria compartida.
- El Motor de Emulación es análogo al que se presentó inicialmente, con la salvedad de que los sensores de temperatura se han mapeado en el rango de memoria de todos los procesadores.
- Las Bibliotecas SW de Estimación no se ven modificadas, puesto que operan con estadísticas provenientes de los *sniffers*, y generan temperaturas para los sensores térmicos. No influye si es un sistema multiprocesador, o tiene SO, dado que los *sniffers* analizan los componentes HW del sistema.

A.5.3.1. Extensiones HW y SW del MPOS

La parte más importante de este experimento ha sido el dotar a la PE con el soporte necesario para poder realizar migraciones de tareas entre procesadores. Para ello, he realizado modificaciones tanto en el HW como en el SW del Sistema Emulado.

Como base para el SO, he utilizado uCLinux; una distribución, de tipo Linux, enfocada a sistemas muy sencillos, uniprocesador, y sin unidad de gestión de memoria.

A nivel HW, ha sido necesario añadir los siguientes elementos: un controlador de interrupciones inter-procesador, para poder señalizar eventos sin necesidad de realizar espera activa; un módulo de exclusión mutua, necesario para implementar este mecanismo en el SW; un traductor de direcciones, para suplir la falta de MMU; un multiplexador de conexiones serie, para comunicarnos de manera sencilla con todos los procesadores; y un módulo de escalado de frecuencia, para ajustar independientemente la de cada core.

El objetivo del mencionado HW, es dar soporte al SW que va por encima, para poder ejecutar un MPOS con migración de tareas.

La arquitectura SW completa se muestra en la Figura A.20; tal y como se aprecia, se fundamenta en tres componentes: (i) un SO (uCLinux) para cada procesador, que se ejecuta en memoria privada, (ii) una capa intermedia que ofrece servicios de sincronización y comunicación, y (iii) el soporte de migración de tareas y de gestión dinámica de recursos. Cada tarea se ejecuta en un sólo SO, y puede ser migrada de uno a otro. Los datos se comparten entre tareas mediante el uso de servicios explícitos. Para que todo esto funcione, existen una serie de servicios que se ejecutan en segundo plano:

- El Soporte de Comunicaciones y Sincronización: Ofrece mecanismos de paso de mensajes y de memoria compartida.
- El Soporte de Migración de Tareas: Permite suspender la ejecución de

una tarea en un procesador y continuarla en otro diferente, manteniendo el estado. Para ello, se replica parte de la estructura de datos que gestiona las tareas en el núcleo del SO. Se cuenta, además, con unos demonios (un maestro y varios esclavos, uno por procesador) que trabajan a nivel de núcleo, y que colaboran conjuntamente para realizar las migraciones que se ordenan desde el SW a nivel de usuario.

- El Motor de Decisión: Es una aplicación que monitoriza la temperatura del sistema (facilitada por el SW intermedio, en tiempo real, al MPOS), y se encarga de distribuir dinámicamente el trabajo entre los distintos procesadores, activando para ello migraciones de tareas. El usuario deberá modificarla para programar sus propias políticas de gestión de temperatura.

A.5.3.2. Caso de estudio

En esta sección, incluyo un conjunto de experimentos enfocados a comprobar la eficacia de la PE para estudiar técnicas de gestión de temperatura a nivel de SO en sistemas multiprocesador (MPSoCs).

En primer lugar, he de comentar el flujo de diseño de la "PE mejorada con MPOS", que contiene una pequeña modificación con respecto al presentado en la Sección A.4. A la hora de generar los binarios para la FPGA, se debe indicar al SO la configuración HW subyacente. En la práctica, esto requiere, simplemente, facilitar un fichero de configuración, generado automáticamente por el EDK, al conjunto de herramientas que genera el SW; de esta manera, el SO incluye en su kérnel los drivers de los módulos instanciados, y da soporte al usuario para accederlos. El conjunto de herramientas que acompañan a uCLinux se encarga, automáticamente, de generar el kérnel del SO, así como de compilar la aplicación de usuario, incluirla en el sistema de archivos, y generar la imagen SW final (kérnel + sistema de archivos).

La arquitectura del Sistema Emulado se puede observar en el floorplan de la Figura A.21; Consta de 4 cores ARM7, cada uno con una memoria privada, cacheable, de 64KB, y con acceso a una memoria compartida de 32KB. Hay dos cachés independientes (instrucciones y datos) por procesador, de 8KB cada una. Las memorias y los procesadores están conectados mediante un bus AMBA. Como SW, ejecutan tareas sintéticas que imponen una carga cercana al 100 %.

En cuanto a la infraestructura de emulación, el floorplan se ha dividido en 128 celdas térmicas regulares, y hay un sniffer por elemento presente en la Figura A.21. Las frecuencias y las cargas de trabajo de los cores constituyen la información monitorizada.

La Figura A.22 muestra el sistema ejecutando una tarea SW que se va migrando, de forma rotacional, entre los distintos cores. Obsérvese que, de los cuatro procesadores que se han mapeado en el sistema, tan sólo se usan

los tres primeros; esto nos permite simplificar las figuras, pues el cuarto procesador no aporta información relevante en este caso particular. Así pues, el propietario de la tarea cambia periódicamente, de MB1 a MB0, a MB2, y de vuelta a MB1. En el floorplan, Procesador1 se refiere a MB0, Procesador2 a MB1, etc...

Las curvas de la Figura A.22 muestran las temperaturas y las frecuencias de cada core a lo largo del tiempo. Dentro del Sistema Emulado, el SW intermedio está continuamente monitorizando las temperaturas de los procesadores, y comparándolas con un límite, fijado en 365 Kelvin. Como se puede observar, en cuanto la temperatura de MB1 supera el límite preestablecido, el Motor de Decisión inicia la migración de la tarea hacia el procesador MB2, que está frío. Tras esto, la frecuencia de MB1 puede decrementarse, lo que da lugar a un descenso de la temperatura del mismo. Al cabo de un tiempo, la misma situación se repetirá: primero, entre MB2 y MB3 y, posteriormente, entre MB3 y MB1.

La realización de este experimento me ha permitido, no sólo probar la eficacia de la PE para realizar este tipo de estudios sino, también, extraer las siguientes conclusiones:

- La temperatura de cada core depende de los elementos adyacentes, y está fuertemente afectada por la carga de trabajo. Este hecho puede ser aprovechado por el SO, puesto que tiene conocimiento pleno de las tareas que se están ejecutando y, lo que es más importante, de las que están planificadas para ejecutarse en el futuro, lo que le permitirá tomar decisiones "más térmicamente inteligentes" para el sistema.

- El tiempo necesario para observar cambios en la temperatura es mucho mayor que el requerido para migrar una tarea. Por esta razón, la migración de tareas, a pesar de la sobrecarga que presenta debida a la replicación de datos, es un mecanismo válido para realizar gestión térmica.

- En cuanto al rendimiento de la plataforma, la duración del experimento fueron 90 segundos para emular 6 segundos de tiempo real, lo que significa una mejora de más de 1000× con respecto a simuladores SW que modelan el SO [16].

A.6. Conclusiones y trabajo futuro

A día de hoy, las arquitecturas MPSoC son la mejor alternativa a los sistemas tradicionales, que ya no son capaces de cumplir con las estrictas restricciones de diseño actuales (rendimiento, tamaño, consumo, ...). Sin embargo, su alta complejidad trae nuevos retos para el diseñador que, en tiempo

de diseño, ha de tener en cuenta futuros problemas de temperatura, fiabilidad, etc... Por tanto, se necesitan nuevas herramientas que permitan acelerar este nuevo flujo de diseño.

En esta tesis, he introducido un nuevo entorno de emulación HW/SW, que permite a los diseñadores de MPSoCs estudiar el comportamiento de este tipo de sistemas en cuanto a rendimiento, consumo de potencia, temperatura, y fiabilidad, con mayor rapidez (hasta tres órdenes de magnitud) que utilizando simuladores SW. Además, a diferencia de estos, no presenta los problemas de escalabilidad asociados al crecimiento del número de señales a gestionar dentro del Sistema Emulado.

El trabajo incluye una sección experimental con ejemplos en los cuales utilizo la PE para explorar el espacio de diseño de varias arquitecturas MPSoC de cara a aplicar modificaciones, tanto HW como SW, para mejorar sus propiedades. Los resultados obtenidos son representativos, y no hacen sino abrir la puerta a futura experimentación con el entorno.

Así como en la Sección A.5.3, se describieron una serie de modificaciones para dotar a la PE de un MPOS con migración de tareas, propongo a continuación una lista de posibles mejoras arquitectónicas para la plataforma. Debe tenerse en cuenta que todas ellas requieren un gran esfuerzo de implementación:

- Expandir a un entorno multi-FPGA: Para modelar MPSoCs más grandes, podemos usar FPGAs con más capacidad o migrar a una plataforma multi-FPGA. La segunda opción es más atractiva desde el punto de vista económico, puesto que los modelos grandes de FPGAs son órdenes de magnitud más caros.

- Mejorar la comunicación FPGA-PC: A medida que el Sistema Emulado crece en tamaño, también crece la cantidad de información a intercambiar. Cuando la conexión Ethernet sature, se podría pasar a una conexión Gigabit-Ethernet y, de ahí, evolucionar a una conexión PCI, o a usar el Serial IO de Xilinx, por ejemplo.

- Portar nuevos cores y procesadores: Como, por ejemplo, el procesador OpenRisc1200, cuyo código fuente está disponible, libre de cargos, en internet.

- Integrar herramientas de terceros: Desarrollar scripts que automaticen la ejecución de otras herramientas (e.g.: Sunfloor), sin salir del flujo de la PE.

Para concluir, presento varios campos de trabajo que se pueden beneficiar en gran medida de la plataforma:

- Estudio de técnicas complejas de gestión de temperatura: En los experimentos, introduje varias políticas sencillas de control de temperatura

para demostrar la utilidad de la PE. La plataforma ofrece un entorno perfecto para desarrollar técnicas más avanzadas, como las basadas en teoría de control o en redes neuronales (que aprenden de la historia pasada), por citar dos ejemplos. De cara a la implementación, en principio, tan sólo se necesita modificar el Motor de Decisión, el algoritmo que autónomamente activa las contramedidas térmicas, aunque, adicionalmente, sería conveniente añadir un mayor soporte HW (e.g.: cachés reconfigurables, adaptación de la anchura de estructuras, control de especulación, etc...), para ampliar el espacio para la optimización.

- Inyección de fallos: Usando técnicas de inyección de fallos, los diseñadores estudian el comportamiento de los sistemas ante circunstancias inesperadas. La arquitectura puede ser entonces modificada para mejorar el manejo de errores y reducir las vulnerabilidades del sistema. En la PE, el Motor de Emulación tiene control total de todo lo que sucede; bastaría añadir un mecanismo para inyectar fallos bajo demanda. Desde el punto de vista de la implementación, es análogo a introducir la temperatura, procedente de la salida de la biblioteca térmica, de vuelta en los sensores térmicos.

- Ataques de canal auxiliar: Se trata de un tipo de ataques a sistemas de encriptación electrónicos que explotan la información "auxiliar" emitida durante la operación del dispositivo (i.e.: consumo de potencia, emisión electromagnética, sonido, detalles de temporización, etc.) para romper el sistema. Utilizando la PE podemos evaluar rápidamente, por ejemplo, lo robustas que son las distintas opciones de implementación de un determinado sistema frente a un ataque de canal auxiliar. Para llevarlo a cabo, debemos aumentar la precisión de los modelos de estimación, que deben ofrecer estimaciones a nivel de ciclo, en lugar de a nivel de paso de emulación. También se podrían añadir nuevos modelos, para el espectro electromagnético emitido, el ruido generado, etc.

- Síntesis de alto nivel: Se denomina así a un proceso de diseño automático que consiste en interpretar un algoritmo y crear el HW que implementa dicho comportamiento. Dependiendo de los parámetros a optimizar (e.g., área, consumo, rendimiento), la herramienta de síntesis de alto nivel generará distintas soluciones. Mediante la PE, podemos evaluarlas y caracterizarlas de manera automática.

A.6.1. Legado

La Plataforma de Emulación es un ambicioso proyecto cuya semilla fue plantada, allá por 2005, en la Universidad Complutense de Madrid. En concreto, nació dentro de mi Proyecto de Fin de Carrera de Ingeniería en In-

formática titulado: "Desarrollo de una plataforma de emulación de sistemas empotrados multiprocesador". Con el paso de los años, el proyecto floreció de manera espectacular gracias a la colaboración con otros grupos de otras partes del Globo:

- El grupo de Arquitectura y Tecnología de Computadores (ArTeCS) de la Universidad Complutense de Madrid, España.
- El Laboratorio de Sistemas Empotrados (ESL), y el Laboratorio de Sistemas Integrados (LSI), del Instituto de Ingeniería Electrónica de la Escuela de Ingeniería (STI) de la EPFL, Suiza.
- El Departamento de Matemáticas e Informática de la Universidad de Cagliari, Italia.
- El Departamento de Ingeniería Electrónica e Informática (DEIS), de la Universidad de Bolonia, Italia.
- El Departamento de Informática e Ingeniería de la Universidad del Estado de Pensilvania, Estados Unidos.

A continuación, presento la lista de publicaciones, relacionadas con la Plataforma de Emulación, que he producido durante mi doctorado:

1. "A Fast HW/SW FPGA-Based Thermal Emulation Framework for Multi-Processor System-on-Chip", David Atienza, Pablo G. Del Valle, Giacomo Paci, Francesco Poletti, Luca Benini, Giovanni De Micheli, Jose M. Mendias, 43rd Design Automation Conference (DAC), ACM Press, San Francisco, California, USA, ISSN:0738-100X, ISBN: 1-59593-381-6, pp. 618-623, July 24-28, 2006.

2. "A Complete Multi-Processor System-on-Chip FPGA-Based Emulation Framework", Pablo G. Del Valle, David Atienza, Ivan Magan, Javier G. Flores, Esther A. Perez, Jose M. Mendias, Luca Benini, Giovanni De Micheli, Proc. of 14th Annual IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC), Nice, France, ISBN: 3-901882-19-7 2006 IFIP, IEEE Catalog: 06EX1450, pp. 140 -145, October 2006.

3. "Architectural Exploration of MPSoC Designs Based on an FPGA Emulation Framework", Pablo G. del Valle, David Atienza, Ivan Magan, Javier G. Flores, Esther A. Perez, Jose M. Mendias, Luca Benini, Giovanni De Micheli, XXI Conference on Design of Circuits and Integrated Systems (DCIS), Barcelona, Spain. Publisher Departament dÉlectrónica-Universitat de Barcelona, pp. 1-6, November 2006.

4. "HW-SW Emulation Framework for Temperature-Aware Design in MPSoCs", David Atienza, Pablo G. Del Valle, Giacomo Paci, Francesco Poletti, Luca Benini, Giovanni De Micheli, Jose M. Mendias, Roman Hermida, ACM Transactions on Design Automation for Embedded Systems (TODAES), ISSN: 1084-4309, Association for Computing Machinery, Vol. 12, Nr. 3, pp. 1 - 26, August 2007.

5. "Application of FPGA Emulation to SoC Floorplan and Packaging Exploration", Pablo G. Del Valle, David Atienza, Giacomo Paci, Francesco Poletti, Luca Benini, Giovanni De Micheli, Jose M. Mendias, Roman Hermida. Proc. of XXII Conference on Design of Circuits and Integrated Systems (DCIS), Sevilla, Spain. Publisher Departament dÉlectrónica-Universitat de Barcelona, November 2007.

6. "Reliability-Aware Design for Nanometer-Scale Devices", David Atienza, Giovanni De Micheli, Luca Benini, José L. Ayala, Pablo G. Del Valle, Michael DeBole, Vijay Narayanan. Proceedings of the 13th Asia South Pacific Design Automation Conference, ASP-DAC 2008, Seoul, Korea, January 21-24, 2008. IEEE 2008.

7. "Emulation-based transient thermal modeling of 2D/3D systems-on-chip with active cooling", Pablo G. Del Valle, David Atienza. Microelectronics Journal, Elsevier Science Publishers B. V., Vol. 42, Nr. 4, pp. 564 - 571, April 2011.

8. "Performance and Energy Trade-offs Analysis of L2 on-Chip Cache Architectures for Embedded MPSoCs", Aly, Mohamed M. Sabry, Ruggiero Martino, García del Valle, Pablo. Proceedings of the 20th symposium on Great lakes symposium on VLSI, 2010, p. 305-310. ISBN: 978-1-4503-0012-4.

Por último, incluyo también una lista con publicaciones relevantes, en las cuales no he intervenido directamente, que derivan de la investigación de terceras personas que han utilizado el entorno de emulación para validar sus ideas:

- "Adaptive task migration policies for thermal control in MPSoCs", D. Cuesta, J.L. Ayala, J.I. Hidalgo, D. Atienza, A. Acquaviva, E. Macii. ISVLSI, IEEE Computer Society Annual Symposium on VLSI, 2010.

- "Thermal-aware floorplanning exploration for 3D multi-core architectures", D. Cuesta, J.L. Ayala, J.I. Hidalgo, M. Poncino, A. Acquaviva, E. Macii. Proceedings of the 20th symposium on Great lakes symposium on VLSI, GLSVLSI 2010.

- "Thermal balancing policy for multiprocessor stream computing platforms", F. Mulas, D. Atienza, A. Acquaviva, S. Carta, L. Benini, and

G. De Micheli. Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2009.

- "Thermal-aware compilation for register window-based embedded processors", Mohamed M. Sabry, J.L. Ayala, and D. Atienza. Embedded Systems Letters, 2010.
- "Thermal-aware compilation for system-on-chip processing architectures.", Mohamed M. Sabry, J.L. Ayala, and D. Atienza. Proceedings of the 20th symposium on Great lakes symposium on VLSI, GLSVLSI 2010).
- "Impact of task migration on streaming multimedia for embedded multiprocessors: A quantitative evaluation.", M. Pittau, A. Alimonda, S. Carta and A. Acquaviva. Proceedings of the 2007 5th Workshop on Embedded Systems for Real-Time Multimedia, ESTImedia 2007.
- "Assessing task migration impact on embedded soft real-time streaming multimedia applications.", A. Acquaviva, A. Alimonda, S. Carta and M. Pittau. EURASIP Journal on Embedded Systems, 2008.
- "Energy and reliability challenges in next generation devices: Integrated software solutions", Fabrizio Mulas. PhD. Thesis at the Mathematics and Computer Science Department of the University of Cagliari, 2010.

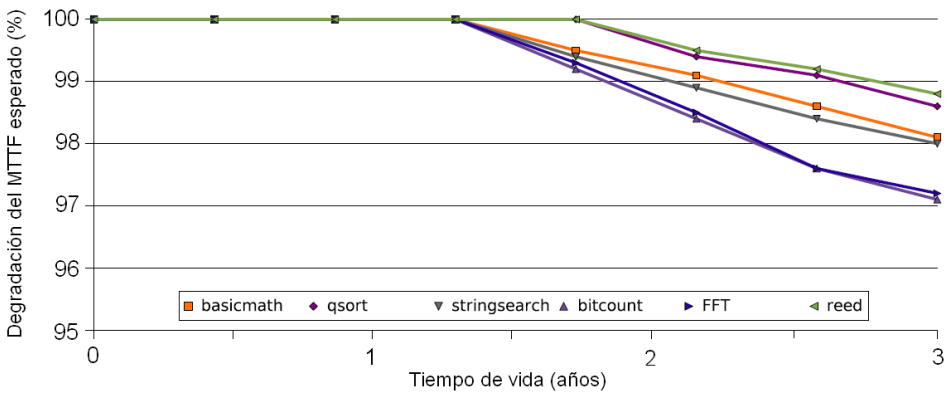


Figura A.15: Evolución de la degradación del MTTF, a lo largo de 3 años, para varios benchmarks.

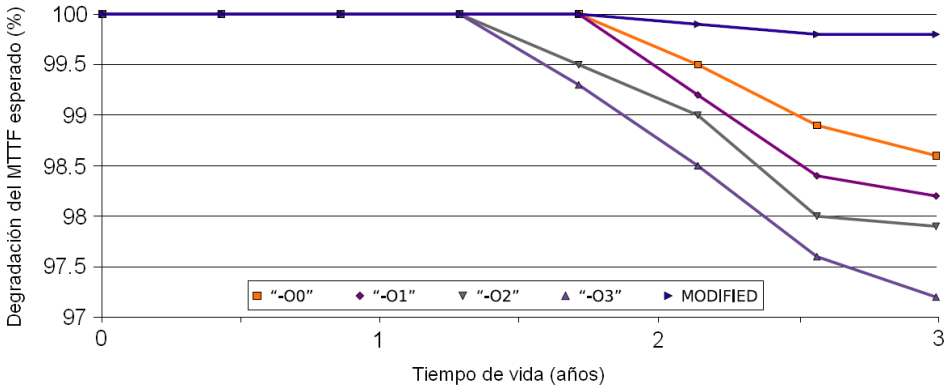


Figura A.16: Evolución de la degradación del MTTF para el benchmark *FFT*, bajo diferentes niveles de optimización del compilador.

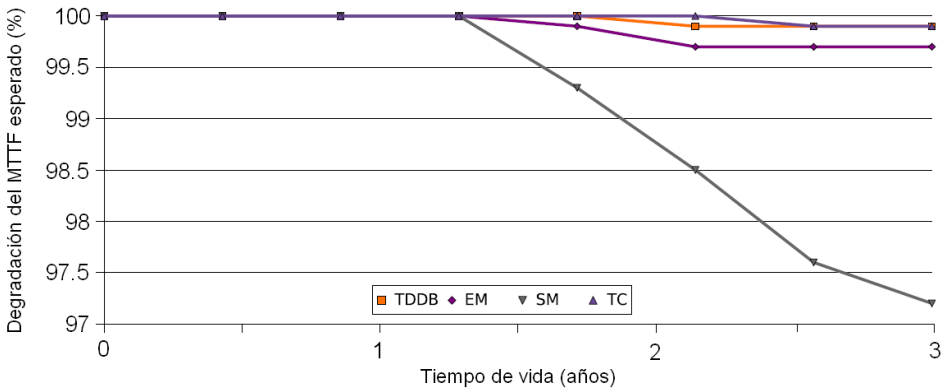


Figura A.17: Contribución de los cuatro factores principales a la degradación del MTTF esperado para el benchmark *FFT* compilado con -O3.

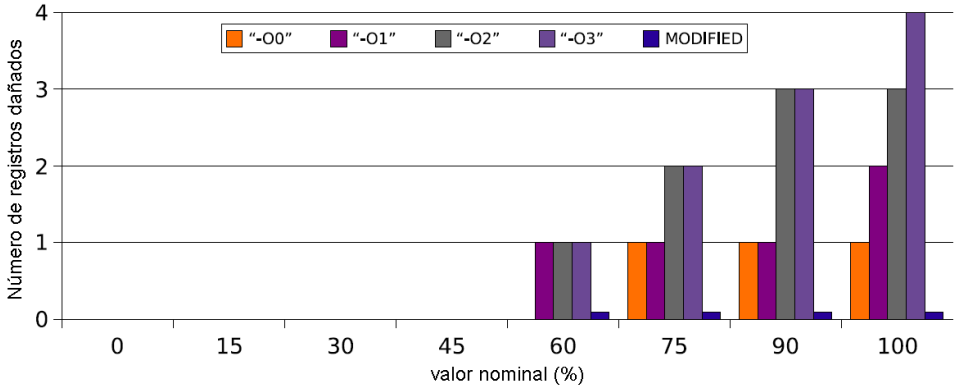
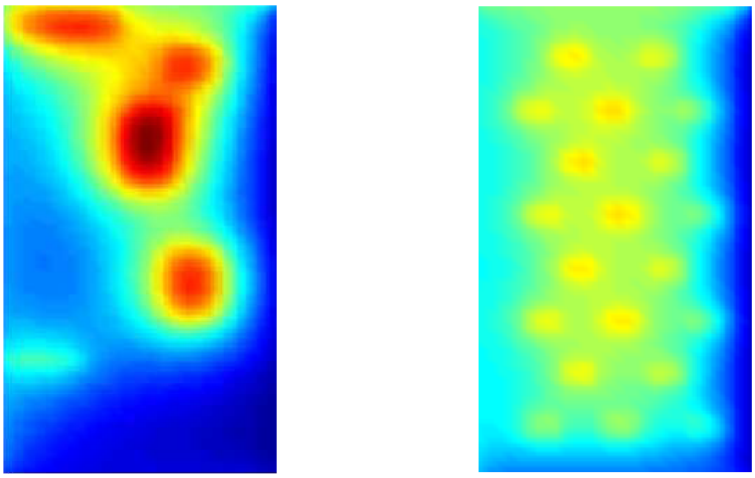


Figura A.18: Comparación del número de registros dañados, al cabo de 2 años, usando diferentes optimizaciones del compilador, y mi algoritmo de mejora de fiabilidad (MODIFIED).



(a) Tradicional (b) Modificada (*MODIFIED*)

Figura A.19: Distribución de temperaturas en el banco de registros del Leon3, utilizando diferentes políticas de asignación de registros.

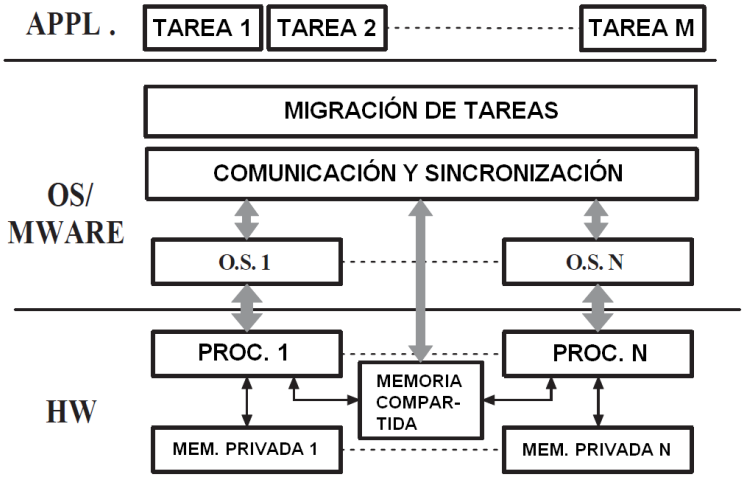


Figura A.20: Arquitectura de las capas de abstracción de SW del MPOS con migración de tareas.

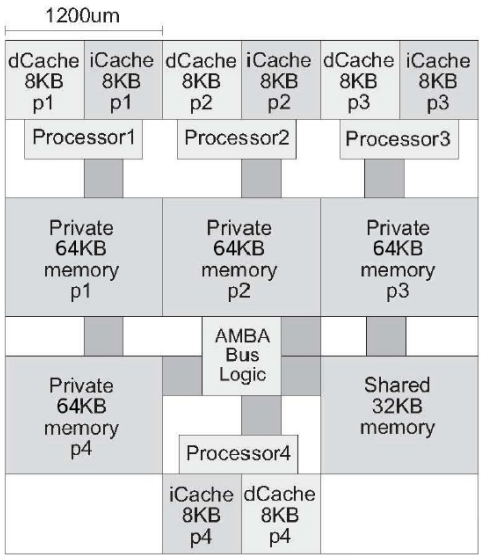


Figura A.21: MPSoC con distribución no uniforme de cores en el floorplan, y con bus compartido.

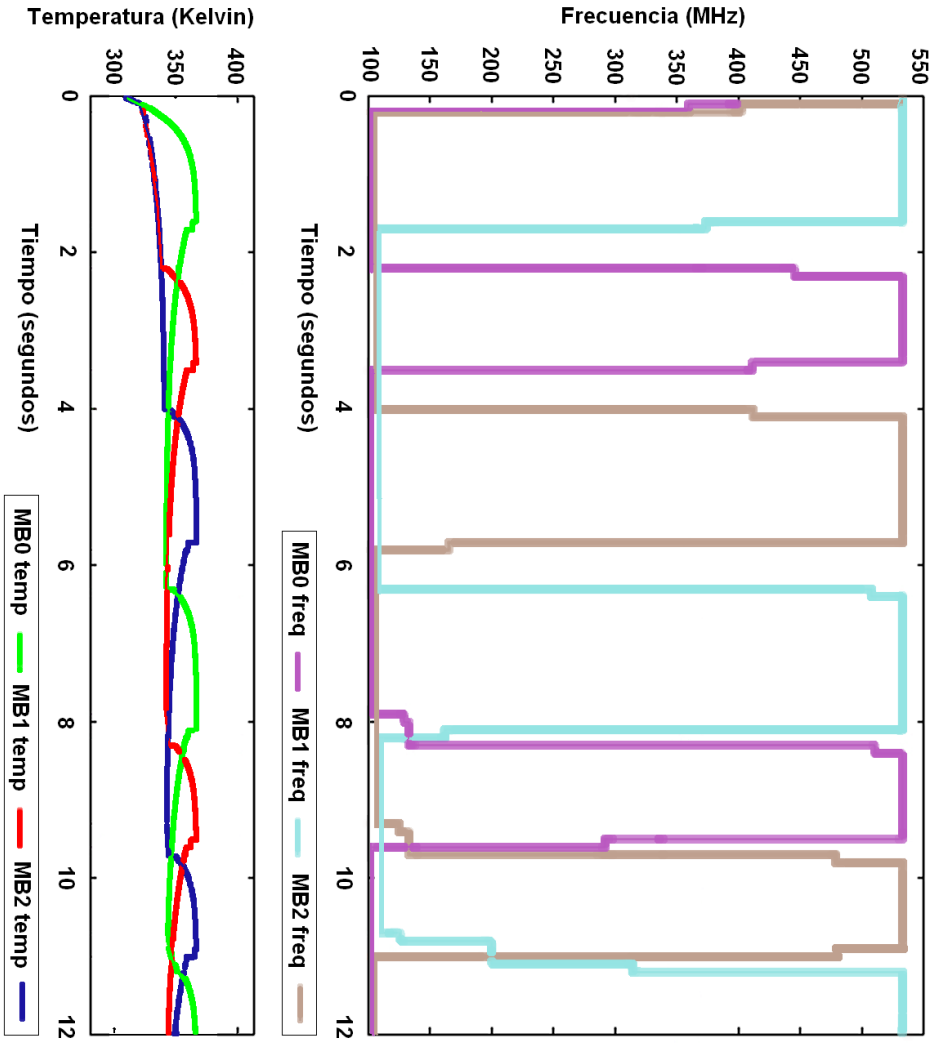


Figura A.22: Evolución de las temperaturas y frecuencias de los elementos de un MPSoC que implementa una política sencilla de migración de tareas en función de la temperatura.

Bibliography

[AAC+05] Arvind, Krste Asanovic, Derek Chiou, James C. Hoe, Christoforos Kozyrakis, Shih-Lien Lu, Mark Oskin, David Patterson, Jan Rabaey, and John Wawrzynek. Ramp: Research accelerator for multiple processors - a community vision for a shared experimental parallel hw/sw platform. Technical Report UCB/CSD-05-1412, EECS Department, University of California, Berkeley, Sep 2005.

[ADVP+06] David Atienza, Pablo G. Del Valle, Giacomo Paci, Francesco Poletti, Luca Benini, Giovanni De Micheli, and Jose M. Mendias. A fast hw/sw fpga-based thermal emulation framework for multi-processor system-on-chip. In *Proceedings of the 43rd annual Design Automation Conference*, DAC '06, pages 618–623, New York, NY, USA, 2006. ACM.

[ADVP+08] David Atienza, Pablo G. Del Valle, Giacomo Paci, Francesco Poletti, Luca Benini, Giovanni De Micheli, Jose M. Mendias, and Roman Hermida. Hw-sw emulation framework for temperature-aware design in mpsocs. *ACM Trans. Des. Autom. Electron. Syst.*, 12:26:1–26:26, May 2008.

[ALE02] Todd Austin, Eric Larson, and Dan Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35:59–67, February 2002.

[AMD04] AMD. Thermal performance comparison for am486dx2 and dx4 in pdh-208 vs pde-208 package. `http://www.amd.com`, 2004.

[ANRD04] Arvind, Rishiyur Nikhil, Daniel Rosenband, and Nirav Dave. High-level synthesis: An essential ingredient for designing complex asics. In *Proceedings of the International Conference on Computer Aided Design*, ICCAD '04, November 2004.

[Apt03] Aptix. System explore. `http://www.aptix.com`, 2003.

[ARM02] ARM. Primexsys platform architecture and methodologies, white paper. http://www.arm.com/pdfs/ARM11%20Core%20&%20Platform%20Whitepaper.pdf, 2002.

[ARM04a] ARM. Arm integrator application. http://www.arm.com, 2004.

[ARM04b] ARM. Arm7tdmi-str71xf tqfp144 and tqfp64 10x10 packages - product datasheets. http://www.arm.com/products/CPUs/ARM7TDMI.html, 2004.

[ASC10] José L. Ayala, Arvind Sridhar, and David Cuesta. Invited paper: Thermal modeling and analysis of 3d multi-processor chips. *Integr. VLSI J.*, 43:327–341, September 2010.

[BBB+05] Luca Benini, Davide Bertozzi, Alessandro Bogliolo, Francesco Menichelli, and Mauro Olivieri. Mparm: Exploring the multi-processor soc design space with systemc. *Journal of VLSI signal processing systems for signal image and video technology*, 41:169 – 182, 2005. [ARTICOLO].

[BDT03] S.S. Bhattacharyya, E.F. Deprettere, and J. Teich. *Domain-specific processors: systems, architectures, modeling, and simulation.* Signal processing and communications. Marcel Dekker, 2003.

[BE] Hagai Bar-El. Introduction to side channel attacks, white paper.

[BEAE01] Vandevelde B., Driessens E., Chandrasekhar A., and Beyne E. Characterisation of the polymer stud grid array, a lead-free csp for high performance and high reliable packaging. *Proceedings of the SMTA*, 2001.

[biba] The international technology roadmap for semiconductors.

[bibb] Matrix semiconductor, inc.

[bibc] Minicom. http://alioth.debian.org/projects/minicom/.

[bibd] Openrisc 1200 risc/dsp core specification, ip core overview. http://openrisc.net/or1200-spec.html.

[bibe] Tcpdump and libpcap libraries. http://www.tcpdump.org/.

[bib03] Proceedings of the gcc developers summit, May 2003.

[BM01] David Brooks and Margaret Martonosi. Dynamic thermal management for high-performance microprocessors. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, HPCA '01, pages 171–, Washington, DC, USA, 2001. IEEE Computer Society.

[BMR+08] T. Brunschwiler, B. Michel, H. Rothuizen, U. Kloter, B. Wunderle, H. Oppermann, and H. Reichl. Interlayer cooling potential in vertically integrated packages. *Microsyst. Technol.*, 15:57–74, October 2008.

[BWS+03] Gunnar Braun, Andreas Wieferink, Oliver Schliebusch, Rainer Leupers, Heinrich Meyr, and Achim Nohl. Processor/memory co-exploration on multiple abstraction levels. In *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*, DATE '03, pages 10966–, Washington, DC, USA, 2003. IEEE Computer Society.

[CAA+09] Ayse K. Coskun, Jose L. Ayala, David Atienza, Tajana Simunic Rosing, and Yusuf Leblebici. Dynamic thermal management in 3d multicore architectures. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 1410–1415, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[Cad05] Cadence. Cadence palladium ii. `http://www.cadence.com`, 2005.

[Cla06] T.A.C.M. Claasen. An industry perspective on current and future state of the art in system-on-chip (soc) technology. *Proceedings of the IEEE*, 94(6):1121 –1137, june 2006.

[CoW04] CoWare. Convergence and lisatek product lines, 2004.

[CRW07] Ayse Kivilcim Coskun, Tajana Simunic Rosing, and Keith Whisnant. Temperature aware task scheduling in mpsocs. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '07, pages 1659–1664, San Jose, CA, USA, 2007. EDA Consortium.

[CS03] Guoqiang Chen and Sachin Sapatnekar. Partition-driven standard cell thermal placement. In *Proceedings of the 2003 international symposium on Physical design*, ISPD '03, pages 75–80, New York, NY, USA, 2003. ACM.

[CSM+06] Ayse Kivilcim Coskun, Tajana Simunic, Kresimir Mihic, Giovanni De Micheli, and Yusuf Leblebici. Analysis and op-

timization of mpsoc reliability. *J. Low Power Electronics*, 2(1):56–69, 2006.

[CW97] Chris C. N. Chu and D. F. Wong. A matrix synthesis approach to thermal placement. In *Proceedings of the 1997 international symposium on Physical design*, ISPD '97, pages 163–168, New York, NY, USA, 1997. ACM.

[CZ05] J. Cong and Yan Zhang. Thermal via planning for 3-d ics. In *Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design*, ICCAD '05, pages 745–752, Washington, DC, USA, 2005. IEEE Computer Society.

[Daw10] Paul Dawkins. *Differential Equations*. 2010.

[dBG11] Alberto Antonio del Barrio García. Using Speculative Functional Units in High-Level Synthesis. Master's thesis, Computer Architecture and Automation Department of the University Complutense of Madrid., 2011.

[DD97] Timothy A. Davis and Iain S. Duff. An unsymmetric-pattern multifrontal method for sparse lu factorization. *SIAM J. Matrix Anal. Appl.*, 18:140–158, January 1997.

[DM06] James Donald and Margaret Martonosi. Techniques for multicore thermal management: Classification and new exploration. In *Proceedings of the 33rd annual international symposium on Computer Architecture*, ISCA '06, pages 78–88, Washington, DC, USA, 2006. IEEE Computer Society.

[dMG97] Giovanni de Micheli and Rajesh K. Gupta. Hardware/software co-design. In *Proceedings of the IEEE, Vol. 85, No. 3, 349*, March 1997.

[dOFdLM+03] Julio A. de Oliveira Filho, Manoel Eusébio de Lima, Paulo Romero Maciel, Juliana Moura, and Bruno Celso. A fast ip-core integration methodology for soc design. In *Proceedings of the 16th symposium on Integrated circuits and systems design*, SBCCI '03, pages 131–, Washington, DC, USA, 2003. IEEE Computer Society.

[DWM+05] W. Rhett Davis, John Wilson, Stephen Mick, Jian Xu, Hao Hua, Christopher Mineo, Ambarish M. Sule, Michael Steer, and Paul D. Franzon. Demystifying 3d ics: The pros and cons of going vertical. *IEEE Des. Test*, 22:498–510, November 2005.

[DYIM07] Moody Dreiza, Akito Yoshida, Kazuo Ishibashi, and Tadashi Maeda. High density pop (package-on-package) and package stacking development. In *Electronic Components and Technology Conference*, 2007.

[EE05] Emulation and Verification Engineering. Zebu xl and zv models. `http://www.eve-team.com`, 2005.

[EH92] R. Ernst and J. Henkel. Hardware-software codesign of embedded controllers based on hardware extraction. In *Proceedings International Workshop on Hardware/Software Co-Design*, Estes Park, Colorado, USA, September 1992.

[Eng04] Heron Engineering. Heron mpsoc emulation. `http://www.hunteng.co.uk`, 2004.

[FM02] Krisztián Flautner and Trevor Mudge. Vertigo: automatic performance-setting for linux. In *Proceedings of the 5th symposium on Operating systems design and implementation*, OSDI '02, pages 105–116, New York, NY, USA, 2002. ACM.

[FS85] R.W. Floyd and L. Steinberg. Adaptive algorithm for spatial gray scale, 1985.

[Gaia] Aeroflex Gaisler. Aeroflex gaisler research. `http://www.gaisler.com`.

[Gaib] Aeroflex Gaisler. Leon3 sparc v8 processor ip core. `http://www.gaisler.com/cms/index.php?option=com_content&task=view&id=13&Itemid=53`.

[GD92] Rajesh K. Gupta and Giovanni DeMicheli. System synthesis via hardware-software co-design. Technical report, Stanford, CA, USA, 1992.

[GK89] J. P. Gray and T. A. Kean. Configurable hardware: a new paradigm for computation. In *Proceedings of the decennial Caltech conference on VLSI on Advanced research in VLSI*, pages 279–295, Cambridge, MA, USA, 1989. MIT Press.

[Gra03] Mentor Graphics. Platform express and primecell. `http://www.mentor.com/products/embedded_software/platform_baseddesign/`, 2003.

[GRE+01] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE*

International Workshop, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[GS05] Brent Goplen and Sachin Sapatnekar. Thermal via placement in 3d ics. In *Proceedings of the 2005 international symposium on Physical design*, ISPD '05, pages 167–174, New York, NY, USA, 2005. ACM.

[HBA03] Seongmoo Heo, Kenneth Barr, and Krste Asanović. Reducing power density through activity migration. In *Proceedings of the 2003 international symposium on Low power electronics and design*, ISLPED '03, pages 217–222, New York, NY, USA, 2003. ACM.

[HLZW05] R. Hon, S.W.R. Lee, S.X. Zhang, and C.K. Wong. Multistack flip chip 3d packaging with copper plated through-silicon vertical interconnection. In *Electronic Packaging Technology Conference, 2005. EPTC 2005. Proceedings of 7th*, volume 2, page 6 pp., dec. 2005.

[Hol] ARM Holdings. Arm7 processor family. `http://www.arm.com/products/processors/classic/arm7/index.php`.

[HP07] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach (4. ed.)*. Morgan Kaufmann, 2007.

[HTI97] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. Fault injection techniques and tools. *Computer*, 30:75–82, April 1997.

[HVE+07] Michael Healy, Mario Vittes, Mongkol Ekpanyapong, Chinnakrishnan Ballapuram, Sung Kyu Lim, Hsien-Hsin S. Lee, and Gabriel H. Loh. Multiobjective microarchitectural floorplanning for 2d and 3d ics. In *In IEEE Transactions on Computer Aided Design (TCAD), vol. 26(1), pp. 38-52*, January 2007.

[IBM06] IBM. Ibm packaging solutions. `http://www-03.ibm.com/chips/asics/products/packaging.html`, 2006.

[Inc] SPARC International Inc. The sparc architecture manual version 8.

[JMBDM08] Antoine Jalabert, S Murali, Luca Benini, and G De Micheli. xpipescompiler: A tool for instantiating application-specific networks on chip. *Design Automation and Test in Europe.*

The Most Influential Papers of 10 Years DATE, page 157, 2008.

[JTW05] Ahmed Jerraya, Hannu Tenhunen, and Wayne Wolf. Guest editors introduction: Multiprocessor systems-on-chips. *Computer*, 38:36–40, July 2005.

[JYC00] et al. J-Y. Choi. Low power register allocation algorithm using graph coloring, Sept 2000.

[KAO05] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25:21–29, March 2005.

[KPPR00] F. Koushanfar, V. Prabhu, M. Potkonjak, and J.M. Rabaey. Processors for mobile applications. In *International Conference on Computer Design, pp. , Austin, TX*, pages 603–608, September 2000.

[LBGB00] Sergio Lopez-Buedo, Javier Garrido, and Eduardo Boemo. Thermal testing on reconfigurable computers. *IEEE Des. Test*, 17:84–91, January 2000.

[MCE+02] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35:50–58, February 2002.

[Mic] Microsoft. Visual studio. `http://www.microsoft.com/visualstudio/en-us`.

[Mot09] Makoto Motoyoshi. Through-silicon via (tsv). In *Proceedings of the IEEE Vol. 97, No. 1*, January 2009.

[MPB+08] Fabrizio Mulas, Michele Pittau, Marco Buttu, Salvatore Carta, Andrea Acquaviva, Luca Benini, and David Atienza. Thermal balancing policy for streaming computing on multiprocessor architectures. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '08, pages 734–739, New York, NY, USA, 2008. ACM.

[NBT+05] Mario Diaz Nava, Patrick Blouet, Philippe Teninge, Marcello Coppola, Tarek Ben-Ismail, Samuel Picchiottino, and Robin Wilson. An open platform for developing multiprocessor socs. *Computer*, 38:60–67, July 2005.

[NHK+04] Yuichi Nakamura, Kouhei Hosokawa, Ichiro Kuroda, Ko Yoshikawa, and Takeshi Yoshimura. A fast hardware/software co-verification method for system-on-a-chip by using a c/c++ simulator and fpga emulator with shared register communication. In *Proceedings of the 41st annual Design Automation Conference*, DAC '04, pages 299–304, New York, NY, USA, 2004. ACM.

[PG03] Magarshack Philippe and Paulin Pierre G. System-on-chip beyond the nanometer wall. In *Proceedings of the 40th annual Design Automation Conference*, DAC '03, pages 419–424, New York, NY, USA, 2003. ACM.

[PMPB06] G. Paci, P. Marchal, F. Poletti, and L. Benini. Exploring temperature-aware design in low-power mpsocs. In *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, DATE '06, pages 838–843, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[PN06] Massoud Pedram and Shahin Nazarian. Thermal modeling, analysis, and management in vlsi circuits: principles and methods. In *Proceedings of the IEEE*, 2006.

[PPB02] Pierre Paulin, Chuck Pilkington, and Essaid Bensoudane. Stepnp: A system-level exploration platform for network processors. *IEEE Des. Test*, 19:17–26, November 2002.

[Rab00] J. Rabaey. Low-power silicon architectures for wireless communications. In *Proc. Asian and South Pacific Design and Automation Conference (ASP-DAC)*, pages 377–380, June 2000.

[RLSC10] Ayala Rodrigo, José Luis, Arvind Sridhar, and David Cuesta. Thermal modeling and analysis of 3D multi-processor chips. *Integration -Amsterdam-*, 43(7):1–15, 2010.

[RS99] Erven Rohou and Michael D. Smith. Dynamically managing processor temperature and power. In *In 2nd Workshop on Feedback-Directed Optimization*, 1999.

[RSV97] J.A. Rowson and A. Sangiovanni-Vincentelli. Interface-based design. In *Design Automation Conference, 1997. Proceedings of the 34th*, pages 178 –183, jun 1997.

[SA03] Jayanth Srinivasan and Sarita V. Adve. Predictive dynamic thermal management for multimedia applications. In *Pro-*

ceedings of the 17th annual international conference on Supercomputing, ICS '03, pages 109–120, New York, NY, USA, 2003. ACM.

[SAAC11] Mohamed Sabry, David Atienza Alonso, and Ayse Kivilcim Coskun. Thermal Analysis and Active Cooling Management for 3D MPSoCs. In *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS'11)*, volume 1, pages 2237–2240, New York, 2011. IEEE Press.

[SABR05] Jayanth Srinivasan, Sarita V. Adve, Pradip Bose, and Jude A. Rivers. Lifetime reliability: Toward an architectural solution. *IEEE Micro*, 25:70–80, May 2005.

[Sem00] International Sematech. Semiconductor device reliability failure models, 2000.

[SLD+03] Haihua Su, Frank Liu, Anirudh Devgan, Emrah Acar, and Sani Nassif. Full chip leakage estimation considering power supply and temperature variations. In *Proceedings of the 2003 international symposium on Low power electronics and design*, ISLPED '03, pages 78–83, New York, NY, USA, 2003. ACM.

[SMBDM09] Ciprian Seiculescu, Srinivasan Murali, Luca Benini, and Giovanni De Micheli. Sunfloor 3d: a tool for networks on chip topology synthesis for 3d systems on chips. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 9–14, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[SSS+04] Kevin Skadron, Mircea R. Stan, Karthik Sankaranarayanan, Wei Huang, Sivakumar Velusamy, and David Tarjan. Temperature-aware microarchitecture: Modeling and implementation. *ACM Trans. Archit. Code Optim.*, 1:94–125, March 2004.

[Sta11] William Stallings. *Operating Systems - Internals and Design Principles (7th ed.)*. Pitman, 2011.

[Syn03] Synopsys. Realview maxsim esl environment. `http://www.synopsys.com`, 2003.

[url06] Embedded linux/microcontroller project. `http://www.uclinux.org/`, 2006.

[VS83] Jiri Vlach and Kishore Singhal. *Computer Methods for Circuit Analysis and Design.* John Wiley & Sons, Inc., New York, NY, USA, 1983.

[WF00] T. Wolf and M. Franklin. Commbench-a telecommunications benchmark for network processors. In *Proceedings of the 2000 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 154–162, Washington, DC, USA, 2000. IEEE Computer Society.

[WVC02] Stephan Wong, Stamatis Vassiliadis, and Sorin Cotofana. Future directions of (programmable and reconfigurable) embedded processors. In *In Embedded Processor Design Challenges, Workshop on Systems, Architectures, Modeling, and Simulation - SAMOS*. Marcel Dekker, Inc, 2002.

[Xil05] Xilinx. Opb block ram (bram) interface controller, December 2005.

[Xil10a] Xilinx. Logicore ip on-chip peripheral bus v2.0 with opb arbiter (v1.00d), April 2010.

[Xil10b] Xilinx. Logicore ip processor local bus (plb) v4.6 (v1.05a), September 2010.

[Xil10c] Xilinx. Powerpc 405 processor block reference guide ug018 (v2.4), January 2010.

[ZADM09] F. Zanini, D. Atienza, and G. De Micheli. A control theory approach for thermal balancing of mpsoc. In *Design Automation Conference, 2009. ASP-DAC 2009. Asia and South Pacific*, pages 37 –42, jan. 2009.

[ZADMB10] Francesco Zanini, David Atienza, Giovanni De Micheli, and Stephen P. Boyd. Online convex optimization-based algorithm for thermal management of mpsocs. In *Proceedings of the 20th symposium on Great lakes symposium on VLSI*, GLSVLSI '10, pages 203–208, New York, NY, USA, 2010. ACM.

[ZGS+08] C. Zhu, Z. Gu, L. Shang, R. P. Dick, and R. Joseph. Three-dimensional chip-multiprocessor run-time thermal management. In *IEEE Transactions on Computer-Aided Design, vol. 27(8):1479-1492, no.3*, August 2008.

–¿Qué te parece desto, Sancho? – Dijo Don Quijote –
Bien podrán los encantadores quitarme la ventura,
pero el esfuerzo y el ánimo, será imposible.

Segunda parte del Ingenioso Caballero
Don Quijote de La Mancha
Miguel de Cervantes

–What dost thou think of this, Sancho? – Said Don Quixote –
The enchanters may be able to rob me of good fortune,
but of fortitude and courage they cannot.

Don Quixote, Part II
Miguel de Cervantes

Ahora, si vuesas mercedes me disculparan,
una cita me aguarda; se trata de
labrar ciertos puntitos...

Pablo