Fast Simulation Strategies and Adaptive DVFS Algorithm for Low Power MPSoCs

M. Gligor

UNIVERSITE DE GRENOBLE
INSTITUT POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque
978-2-84813-157-3

T H È S E

pour obtenir le grade de

DOCTEUR DE L'Université de Grenoble
délivré par L'Institut polytechnique de Grenoble

Spécialité : Micro et Nano Électronique

préparée au laboratoire **TIMA**

dans le cadre de **l'École Doctorale d'Électronique, Électrotechnique, Automatique et Traitement du Signal**

présentée et soutenue publiquement

par

Marius GLIGOR

Le 9 Septembre 2010

Titre

Fast Simulation Strategies and Adaptive DVFS Algorithm for Low Power MPSoCs

Directeur de thèse : Frédéric PÉTROT

JURY

M.	Olivier GRUBER	Président
M.	Alain GREINER	Rapporteur
M.	Wolfgang MUELLER	Rapporteur
M.	Kees GOOSSENS	Examinateur
M.	Laurent MAILLET-CONTOZ	Examinateur
M.	Vania JOLOBOFF	Examinateur
M.	Frédéric PÉTROT	Directeur de Thèse

Acknowledgments

First, I would like to thank my PhD advisor, Frédéric Pétrot, head of the SLS department in TIMA Laboratory, for the time dedicated to my work supervision and for the excellent technical and non technical advices.

I would also like to thank my thesis jury members: the president, Prof. Olivier Gruber, the reviewers, Prof. Alain Greiner and Dr. Wolfgang Mueller, the examiners, Prof. Kees Goossens, Dr. Laurent Maillet-Contoz and Prof. Vania Joloboff.

I want to thank all my colleagues who supported and helped me during these years: Patrice Gerin, Xavier Guérin, Pierre Guironnet-de-Massas, Aimen Bouchhima, Hao Shen and Mian Muhammad Hamayun. I thank Lobna Kriaa and Dr. Ahmed Amine Jerraya who, at the very beginning, believed in my project and allowed me to follow this road. Special thanks to Nicolas Fournel, the colleague with whom I worked for the last two years and who helped me to correct and to improve both English and French versions of this thesis.

And finally, my wife, Claudia, who encouraged and supported me and without whom nothing would have been possible.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The miniaturization of the transistors on silicon plays the most important role in the microelectronics industry, enabling the integrated systems to become more and more complex. Complete systems embedding processors, memories and other peripherals on a single chip, called *System on Chip* (SoC), exists in our days thanks to the miniaturization.

An increasing number of SoC devices (laptops, cellular phones, PDAs, aeronautic and space devices) are used both in every day life and in critical missions. The new applications as high resolution and high compression video/audio decoder/encoder and 3D video games running on these devices require more and more computational power (Figure 1.1). For covering the gap between the processing requirement and the available processing performance, in addition to improving the device performances, the number of processing engines (PE) should be increased [ITR07a].

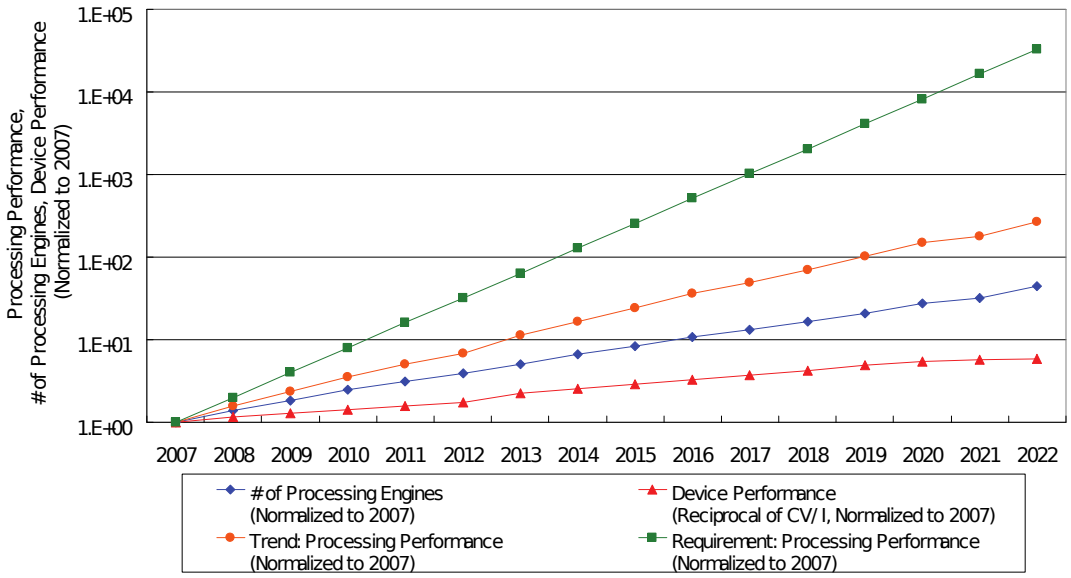


Figure 1.1: SOC Consumer Portable Processing Performance Trends (ITRS 2007)

The ITRS (International Technology Roadmap for Semiconductors) expects that the average circuit complexity of the PEs and of the peripherals to maintain constant complexity, while the number of PEs is expected to grow rapidly in subsequent years. The amount of main memory is assumed to increase proportionally with the number of PEs [ITR07a].

The number of processing engines in SoCs expected by ITRS for next years is presented in Figure 1.2.

The processors of a *Multiprocessors System On Chip* (MPSoC) can be organized in a symmetric or an asymmetric architecture. All processors of a *symmetric multiprocessing* (SMP) architecture are identical and are connected to a logically unique shared memory. There is a single *operating system* (OS) executed by all processors, which schedules any ready task on any free processor. SMP architectures provide an interesting power/performance/flexibility trade-off, allowing to obtain the performances of processor working at a very high frequency using several processors working at lower frequencies, thus consuming less energy provided that the application can be parallelized as a set of coarse grain tasks. The *asymmetric multiprocessing* (ASMP) architectures usually contain some general purpose processors (GPP) and much more digital signal processors (DSP). The GPP allocates and schedules tasks to DSPs, which are specialized processors with a high processing power. Thanks to the specialized processors, the ASMP architectures have higher performance and lower energy consumption compared to the SMP architectures.

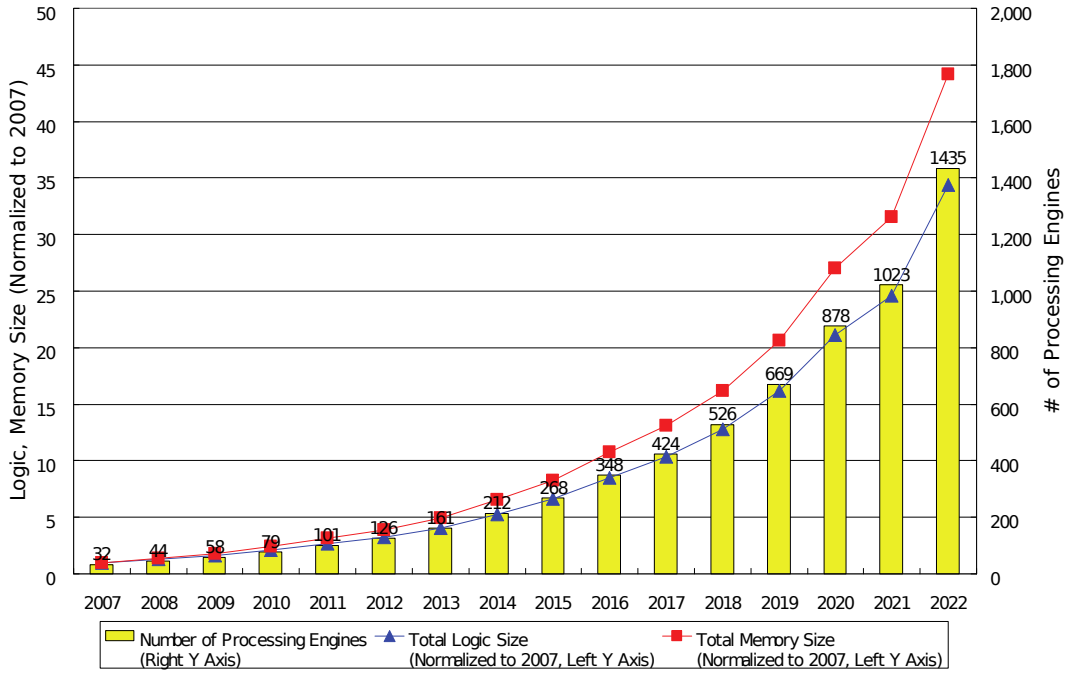


Figure 1.2: SoC Consumer Portable Design Complexity Trends (ITRS 2007)

In order to be profitable, a product must be competitive and its production cost must be low. ITRS predicts that the mask cost will increase in the next ten years 23 times [ITR07b]. For reducing the total cost of the future MPSoCs, the specialized processors (DSP) will be replaced by general purpose ones (GPPs) and their functionalities will be taken by software. Replacing specialized processors by software has an impact on the system performances, but it offers a great flexibility and allows to cope with non functioning processing elements. The future MPSoC architectures containing such a huge number of general purpose processors are expected to be more and more homogeneous and parallel and to use shared memory, as the architecture presented in Figure 1.3.

The homogeneity and the flexibility of the architecture assure short design and verification times and allows the reuse of the same platform not only for sequel versions of the

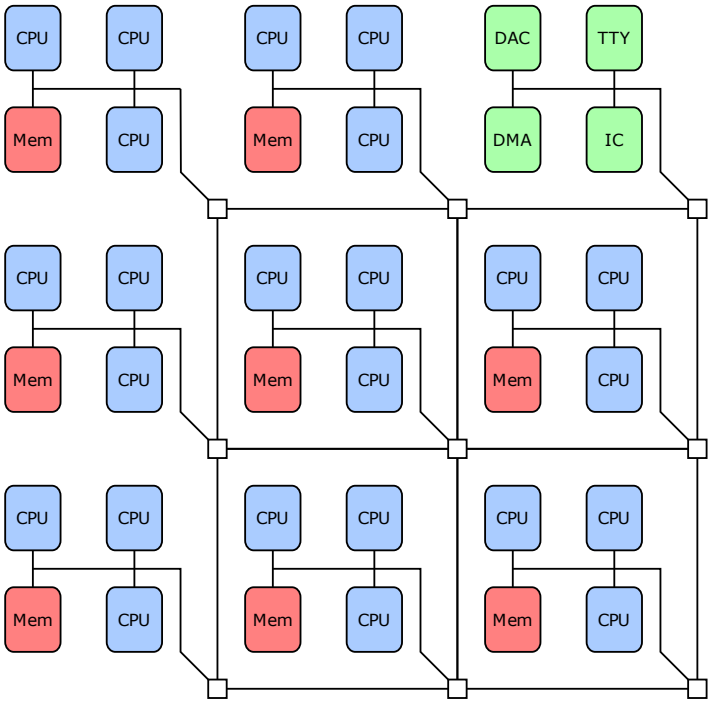


Figure 1.3: Future SoC Architecture

same application, but also for other applications. These features ameliorate the development time leading to a shorter time-to-market, that permits the product to have success. Figure 1.4 presents the gap between the technology capabilities which doubles every 36 months (Moore's law) and the hardware/software design productivity. Beside the reusable design platforms and flexible hardware architectures, a system level design is required to fill the presented gap.

The increasing complexity of the MPSoCs also determines the growth of their energy consumption. The battery technology is also developing, but its progress is slower than the technology integration growth. Almost all portable devices (mobile phones, video cameras, etc.) rely on batteries for the power supply. The battery life directly influences their size, weight and functioning time. Even for pluggable devices, lowering the power consumption also reduces the dissipated heat, what decreases the packaging and cooling solution costs for the integrated circuits. Mastering the increasing energy necessities of these devices represents one of the biggest challenges in MPSoC domain.

Energy savings can be achieved at both software and hardware level. From the hardware point of view, the hardware components can be energy optimized (reducing chip area, keeping high activities wires local and short, etc. [Hav00]). Additional energy can be saved by using components supporting several running modes. Processors supporting frequency and voltage variations, displays with many luminosity levels and other hardware components with power saving capabilities are now available for MPSoCs.

From the software point of view several approaches have been addressed. One of them is based on energy efficient code compilation (reordering instruction, operands swapping, etc. [LTMF95]). Another approach consists in taking advantages, at software level, of the hardware components that supports several running modes.

The first contribution of this thesis consists of a software algorithm that tries to save energy by modifying the processors frequencies and voltage when the system utilization

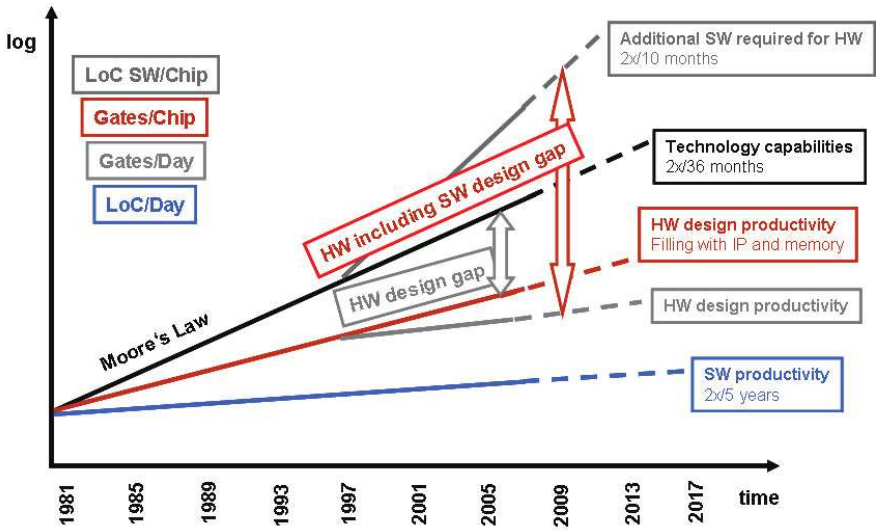


Figure 1.4: Hardware and Software Design Gaps Versus Time (ITRS 2007)

permits. The algorithm relies on information that concern the running time of the tasks and does not need any input from applications.

ITRS proposed solutions for reducing the power consumption of the portable devices include architecture optimization at high-level design stages.

For validating the architecture and for being able to begin the software coding as soon as possible, and thus reducing the entire system design time, there is a common agreement on the necessity of the system simulation. Because the MPSoC architectures are too complex to be described analytically, the simulation represents the only solution for their validation. The simulation also represents the best solution for optimizing the MPSoCs performances and consumption through the architecture exploration, due to the fact that it is very expansive, time consuming and practically almost impossible to physically build and test all candidate architectures.

Architecture can be simulated at different levels of abstraction. The *transistor level*, *gate level* and *RTL* (register transfer level) simulations are used exclusively for hardware. These simulation levels help in designing the circuit as close to perfect as possible before the integrated circuit is first built, what is essential due to the high costs of photo-lithographic masks and other manufacturing prerequisites.

VHDL and Verilog are two programming languages that allow the description of systems such as FPGA (Field-Programmable Gate Array) and integrated circuits using *a hardware description language* (HDL). Although it is possible to execute software using processors and other hardware components modeled with one of these languages, due to the very low simulation speed, the executed software usually only validates the hardware behavior.

Using software programming languages such as C/C++ for the hardware description allows getting rid of all unimportant hardware details from timing and behavior points of view, which increases the simulation speed. This enables the execution of the entire software stack using the hardware model in an acceptable time. As example, SystemC is a C++ library that offers support for modeling and simulation of electronic systems at different levels of abstraction.

In order to test and to see the effectiveness of the proposed energy saving algorithm and also to choose the most appropriate architecture for a specific application, we need fast

and accurate simulation strategies that support individually frequency change for each processor.

The other contributions of this thesis are related to MPSoC simulations targeting power estimation at different level of abstraction. The first simulator combines the accuracy of the hardware focused simulators with the speed of the behavior focused simulators. Other two simulators consist in adaptations of a static scheduling simulator for being able to simulate architectures having multiple frequency.

This thesis has two main research axes. The first axis is related to saving energy at software level. The second axis is related to simulation strategies allowing the execution and validation of the energy saving algorithms. As the energy saving algorithms depend on the simulation platform for their validation, we will always present during this thesis the second axis before the first one.

The rest of this thesis is organized as follows:

Chapter 2 presents the context of this thesis and defines the problems.

Chapter 3 presents the state of the art of the existing non-RTOS (Real-Time Operating System) energy saving algorithms and the existing simulation strategies related to the ones proposed here.

Chapter 4 presents the proposed simulation strategy that combines the accuracy of the hardware focused simulators with the speed of the behavior focused simulators.

Chapter 5 presents the two proposed simulation strategies for the static scheduling of the architectures containing components that work at different frequencies.

Chapter 6 presents the proposed energy saving algorithm.

Chapter 7 concludes this thesis and gives some perspectives.

Chapter 2

Problem definition

Contents

This chapter presents the different problems to which this thesis provides a solution. The chapter begins by presenting a few aspects of the MPSoCs, which represents the general context of this work.

We present then the main features of the C++ based language most commonly used for modeling and simulating the MPSoC architectures that are relevant to our problem. The need of using simulation models at different abstraction levels has been highlighted by many researchers. The presentation continues with an overview of the major abstraction levels. We define then the problems of the abstraction levels we target to improve.

After presenting the basis of the power and energy consumption, we define the problems of the saving energy at software level.

2.1 Context

MPSoCs are heterogeneous systems containing a set of processing nodes interacting through a global communication network (Figure 2.1). The required functionalities are assigned statically or dynamically to the processing nodes, depending on the desired performances in terms of computation, energy *etc*. The processing nodes may be hardware or software.

2.1.1 Hardware nodes

The hardware nodes execute performance critical parts of the application. They provide a fix set of functionalities and can not be programmed to perform others. They are usually hardware accelerators. As example, a deblocking filter, which improves the visual quality of the decoded video frames by smoothing the sharp edges between blocks, takes more than half of the decoding time if it is implemented as a software task running on the software node. By implementing this task as a hardware node that allows to take benefit from the mid grain data parallelism, the decoding speed increases while the consumed energy is reduced.

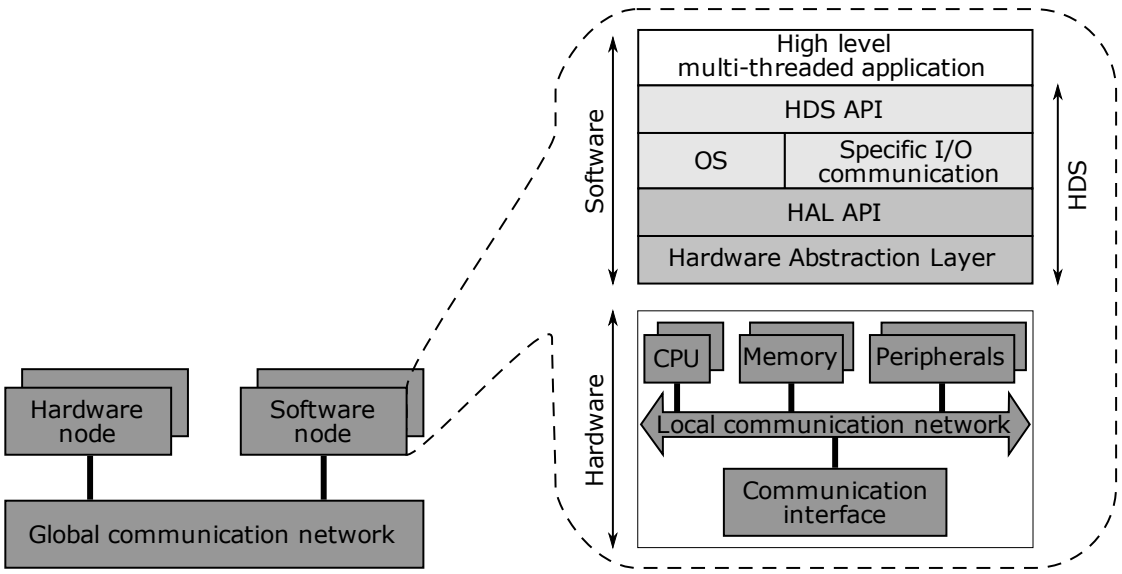


Figure 2.1: MPSoC architecture

2.1.2 Software nodes

A software node includes a software part and a hardware part (Figure 2.1).

The hardware part, called the processor subsystem, consists of processors with their caches, different types of memories, DMA (Direct Memory Access), interrupt controllers and other peripherals communicating together using a local communication network. The communication interfaces component of a processor subsystem makes possible the communication between the components in that software node and the rest of the hardware and software nodes.

The processors of a software node are usually identical (or, at least, support the same instruction set) and they execute the same software stack. So, the software nodes are

organized as SMP architectures.

For facilitating the development of the software running on the more and more complex embedded systems, the software is organized in layers. The software layers executed by the processors of a software node form the software stack of that software node.

The software stack is formed of the multi-threaded application(s), the operating system, the communication libraries and the HAL (Hardware Abstraction Layer).

The application layer contains the tasks of application(s). These tasks implements the functionalities assigned to the software processing node. They are executed concurrently on the processors of the software node.

The rest of the layers are specific to a processor subsystem. They form the Hardware Dependent Software (HDS).

The layer composed of the operating system and the I/O communication libraries represent an interface between the applications and hardware. This layer is responsible for the management and coordination of activities and the sharing of the processor subsystem resources.

The hardware abstraction layer hides the differences in hardware from the operating system, so that the kernel code does not need to be changed to run on systems with different hardware [YBB+03].

Between the application layer and the operating system layer sometimes there is another layer called middleware. It consists of a set of services that allows the interaction between the processes of a distributed application running on different software nodes with different operating systems.

The presented structure of the software stack of a software node is the same as the one used in the classical computers [Tan84].

Due to their programmability, the software nodes offer a great flexibility.

2.2 SystemC

The software programming languages do not offer by default the semantics required for modeling the hardware systems. The time behavior of the hardware components, the interface between them and their parallel execution in time are not easy to model in software programming languages such as C/C++. Many researchers have worked on this topic and proposed C/C++ based languages providing the semantics required for the hardware modeling. These languages allow the validation [GL97, KD90] and sometimes even the synthesis [DM99] of the modeled systems. Due to their provided abstraction, the unimportant hardware details can be eliminated, enabling thus the simulation and the architecture exploration of the MPSoC architectures. This is almost impossible for the architectures modeled at lower levels of abstraction (*e.g.* RTL, gate level).

All simulation platforms presented in this thesis are modeled with SystemC or with a static version of it suited to a given modeling type, called SystemCass. We also use SystemC as reference for demonstrating the correct behavior of our proposed static scheduling simulators described in section 5.

SystemC [SYS] is a library of C++ classes and macros providing support for the modeling of electronic systems. Besides these classes providing the structural elements required for the system modeling, SystemC also provides an event-driven simulation kernel that allows the simulation of the modeled system. The system model is compiled with the SystemC library. The execution of the resulted executable represents the simulation of the modeled system.

SystemC can be used for specification at different levels of abstraction, ranging from high-level functional modeling to detailed clock cycle accurate RTL modeling.

The OSCI (Open SystemC Initiative) organization is responsible to disseminate and prepare specifications for SystemC.

2.2.1 Main elements

The SystemC classes provide the structural and functional elements required for the architecture modeling. Some of these classes can be directly instantiated, others represent base classes and have to be derived in order to add the required functionalities.

The main structural elements of SystemC are the modules, ports, interfaces and channels.

Modules (hardware components, alias *entity* in VHDL) represent the basic building elements of a SystemC design hierarchy. The modules communicate between them using channels, connected to the ports of the modules. A module may contain other modules.

Channels create connections between module ports allowing modules to communicate. Each channel class is derived from one or more interfaces and implements their virtual methods.

Interfaces are C++ abstract classes derived from the SystemC class *sc_interface*. All member functions of an interface class are pure virtual methods. These methods define the communication semantic between the ports and the channels connected to them. The interfaces assure the independence between the implementation and the definition of the communication mechanism.

Ports allow the modules to communicate with their outside world. A port forwards the interface method calls to the channel to which the port is bound.

The main functional elements of SystemC are the processes and the events.

Processes represent the main functional elements. SystemC provides a macro to specify that a method of a module is a process. The processes are executed concurrently by the simulation kernel. The method of a process can call other methods from its module or from the module ports. The called methods will be executed in the context of the calling process. A possible call of the time synchronization function (*wait*) during the execution of the called methods would suspend the calling process. While the process is suspended, other processes can execute and even be suspended in the same method.

Each process has a sensitivity list. There are two types of SystemC processes. The processes from the first type, called method processes, are executed from beginning to end when an event from their sensitivity list is notified. They can not call the *wait* function.

The processes from the second type, called thread processes, are called only once, at the beginning of the simulation, by the simulation kernel. During their execution, they use the *wait* function to wait for an event in their sensitivity list or for another event or future moment. Their execution is not preempted between two *wait* function calls. Each thread process requires its own execution stack and imposes a context switching each time it is suspended or resumed.

As its name says, the entire simulation is driven by **events**. They allow synchronization between processes. When a process notifies an event, the method processes having that event in the sensitivity list are executed and the thread processes waiting for that event are resumed. Timed events can be created by calling the *wait* or *notify* functions with a time value. After all unblocked processes are executed, the simulation kernel advances the simulation time to the time of the next timed event.

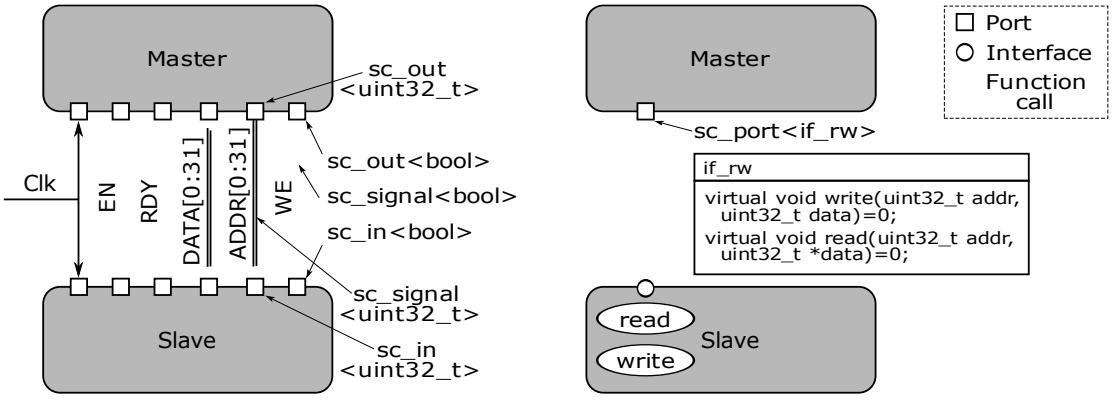
2.2.2 Modeling abstractions

The structural and functional elements provided by SystemC enable the modeling of the architectures at different levels of abstraction.

In the Figure 2.2 example, the communication between two hardware components is modeled at two levels of abstraction.

Figure 2.2(a) models in details this communication. All control, data and address signals of the real hardware components are modeled. SystemC provides a channel class, called *sc_signal*, which can model signals and buses of different sizes. All ports of the components in this figure are connected using only channels of this type. The two interfaces implemented by this channel are the same for all data types. One interface allows writing data to the channel, the other one reading the data. So, the modules communicate between them only through the value of the signals. For executing correctly the transfers, the two modules have to implement precisely the communication protocol. For this, they usually implement a finite state machine (FSM).

Figure 2.2(a) offers a higher level modeling of the communication between the two components. In this case, the hardware signals are not modeled. The modules communicate by direct function calls, not by signals. Between the two modules there is usually no channel. The port of the master module is directly connected to the slave module, which implements the methods of the communication interface. In SystemC terms, we can say that the slave module is a channel that is connected only to the master module. This way of modeling is called Transaction Level Modeling (TLM).



(a) Detailed modeling

(b) High level functional modeling

Figure 2.2: SystemC modeling abstractions

2.3 Abstraction levels

Hardware/software co-simulation has emerged in the early '90s and is now recognized as a key and mature CAD (computer-aided design) technology for the design of systems on a chip for more than a decade. In the quest for simulating systems that on the one hand include more and more IPs and on the other hand embed more and more software, it has been necessary to trade-off accuracy for speed. Many researchers have advocated the use of simulation approaches that make use of different abstraction levels [JBP06]. This technology is now in use everyday in the actual SoC design teams [Ghe06], and some clear proof

of this continuing interest of the semi-conductor industry is the recent combined effort towards standardization of the languages supporting it, *i.e.* [SYS] and [Ber06]. The literature on the inter-related subjects of hardware/software codesign and multiple abstraction levels modeling is very broad, and several textbooks have gathered contributions in the field, *i.e.* [DMEW02, SLM06].

The aim of using these different levels is to allow gradual refinement of the model of the designed system from the most abstract level to the most concrete one. Figure 2.3 gives an overview of the major abstraction levels proposed in the literature for a system consisting of two software tasks communicating with a hardware one. The hybrid object, made of software and hardware, represents the interface between the software and the simulation model for each abstraction level.

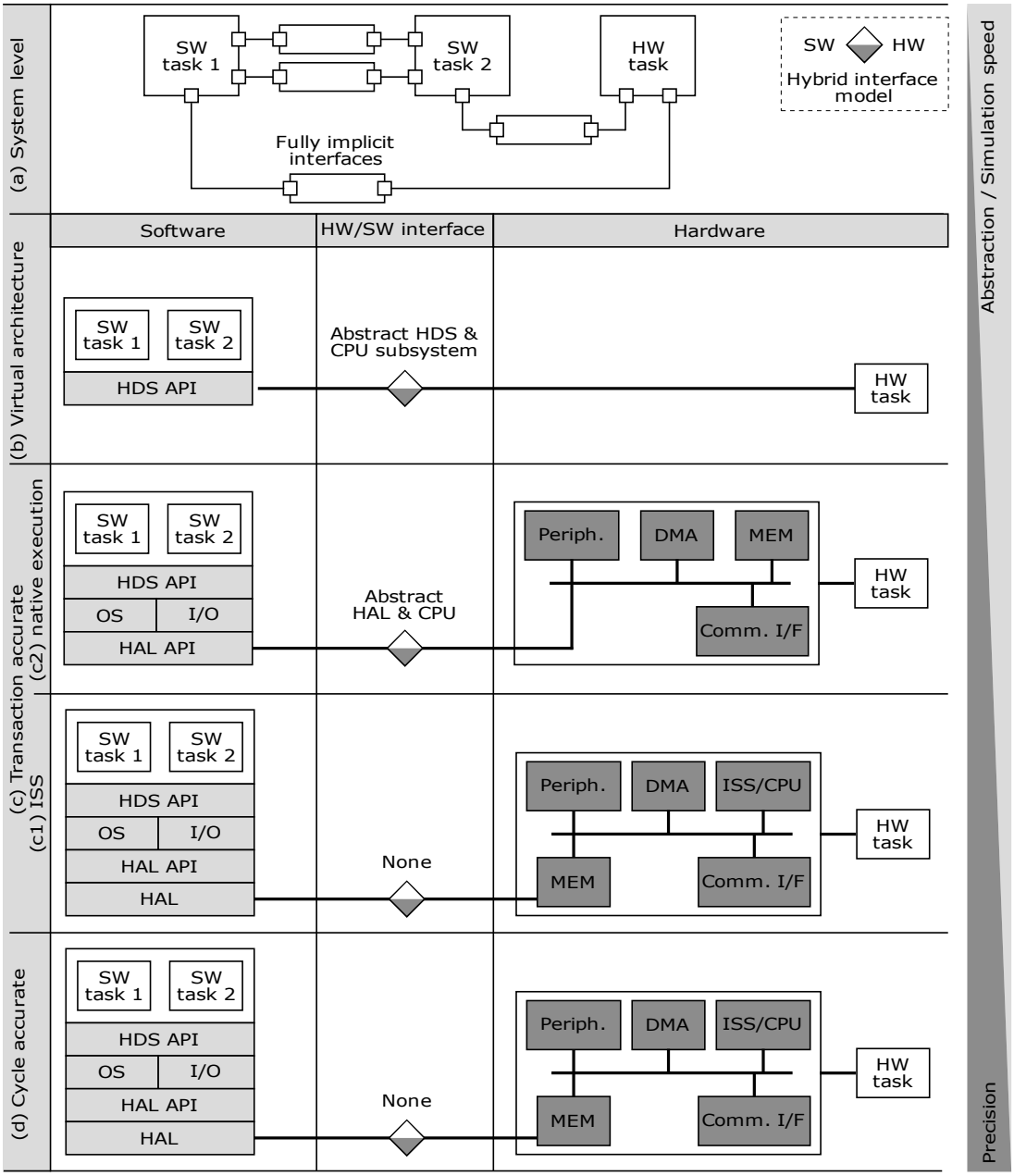


Figure 2.3: Abstraction levels

2.3.1 System level

At the highest level, Figure 2.3(a), system level (SL), all tasks are communicating through abstract channels. The simulation model can be implemented directly in C++ [VKS00] or in specific languages like SystemC and Simulink [PJ07, HHP+07].

At this abstraction level, all hardware characteristics of the architecture on which the software will be executed at the end are ignored. Except the software tasks themselves, there is no relation with the software stack of the final system. Even for the tasks, it is not yet decided which of them will be implemented as hardware and which as software. Due to these reasons, the simulation at the system level abstraction level is very fast, but offers no time information.

2.3.2 Cycle accurate

At the opposite, the lowest level of abstraction is cycle approximate / accurate level (CA) (Figure 2.3(d)). This is the most precise abstraction level considered in this thesis.

At this level, every functional part of the system is represented as a hardware component. The software stack is complete and cross-compiled for the target processor. The resulted target binaries are loaded in the simulated memory when the architecture model is initialized. During the simulation, this binary code is interpreted by the ISSes (instruction set simulator) of the processors. There is no hybrid interface between the software and the simulated hardware.

An ISS models the behavior of a processor. As the real processor, the ISS loads from the simulated instruction cache the instruction code pointed by the simulated program counter register. The loaded instruction is decoded and its corresponding actions are then performed (*e.g.* operation with the simulated registers, access to the main memory or to other devices through the data cache). When an instruction or data cache miss occurs, the cache line is searched over the cycle accurate hardware model. To achieve a high accuracy, an ISS has to take into account the internal architecture of the processor, *i.e.* pipeline, dependence between instructions *etc*.

The hardware components of this abstraction level are modeled using FSMes and communicate between them using signals as in Figure 2.2(a). Clock component(s) are used for generating the cycles. As the hardware components are modeled at the cycle accurate level, their time advances as the cycles pass. This is the only abstraction level which does not require the time annotation of the hardware components.

At this level, the simulation gives precise timing evaluations but implies long simulation times. Thanks to its precision, SoC designers use this abstraction level on some designs to gain confidence on their behavior. The cycle accurate abstraction level represents the last step in the design flow. It is followed by the RTL modeling used for the final circuit synthesis.

Between these two extremes, two intermediate levels of abstraction are commonly proposed [CG03, vdWdKH+04, PGR+08, Don04] to gradually close the gap between fully implicit and fully explicit hardware/software interfaces.

2.3.3 Virtual architecture

The first intermediate abstraction level is the virtual architecture (VA) (Figure 2.3(b)), where the software tasks are grouped into indivisible software subsystems. At this level, the software tasks form a unique software stack, which also contains the hardware dependent

software (HDS) API (Application Programming Interface). This API defines the abstract communication channels and the operating system primitives. The rest of the system is abstracted through the channels. At this level, the hybrid object is represented by the abstraction of the HDS implementation and of the processor subsystem.

2.3.4 Transaction accurate

The second intermediate level, the transaction accurate level (TA), is presented in Figure 2.3(c). The communication between the hardware parts of the system uses transactions.

Two main approaches have been proposed for this abstraction level. In the first approach, the target binary code is interpreted by an ISS as in the cycle accurate abstraction level. In the second one, called native execution, the software, instead of being cross-compiled and interpreted by an ISS, is connected to the hardware architecture model, compiled and executed together.

2.3.4.1 ISS based approach

As in the CA case, for this approach (Figure 2.3(c1)), the simulated software stack is complete, cross-compiled for the target processor, loaded in the memory module(s) and interpreted by the processors ISSes. The differences from the CA abstraction level consist of the way the hardware components are modeled and the communication between them. While the CA hardware components communicate through signals, the hardware components modeled for this approach communicate using an interface having two methods, one for sending and the other for receiving packets. For consuming the time required for transferring the packets and for performing different computations, the components have to be annotated.

The simulation speed in this approach is enhanced compared to the CA abstraction level. However, it may be insufficient for validating heavy applications (*e.g.* H.264 decoder running on top of Linux OS). The main limiting factors are the interpretation of each simulated instruction and the large number of transactions issued over the interconnect.

2.3.4.2 Native execution based approach

In the native execution based approach (Figure 2.3(c2)), the software stack integrates an operating system and some I/O specific code. The software tasks communicate with the hardware tasks using the transaction APIs provided by hardware abstraction layer (HAL). The HAL implementation and the processor are abstracted by the hybrid object. The software stack is compiled and executed together with hardware architecture model.

Similar to the ISS based approach, the hardware components have to be annotated for modeling the time spent by different operations. As there is not component model for the processors, the annotations corresponding to the time required for the target instructions execution have to be inserted directly into the target code. [Ger09] presents a possibility of annotating the simulated software stack with target processor time information, when it is compiled for the host processor.

This approach ameliorates significantly the simulation performances, but it faces some problems inherent to the native execution (*e.g.* conflict between the address space of the simulated application and the address space of hardware model, simulating software stacks which use the MMU (Memory Management Unit)).

2.4 Virtualization

For increasing the simulation speed of the transaction accurate simulation models based on ISSes, it would be desirable to avoid the high price of interpreting each simulated instruction. In order to replace the slow interpretive ISS from Figure 2.3(c2) with a faster ISS, we take a look at what other domains can provide.

More or less during the same period with the hardware/software co-simulation, a totally unrelated effort took place in computer science to make possible the execution of several OSes in isolation concurrently on the same processor [SCK+93], based on the so called *virtual machine* technology developed in the early 60' [Cre81]. This can be done by different now well understood techniques that can target *full virtualization*, requiring a very accurate view of the hardware internals, as it can run an entire OS including drivers without modifications. Other approaches, called *para-virtualization* are heading to more lightweight solutions, and require tiny modification of the OSes to prevent the use of some specific instructions or hardware related accesses.

Many commercial [DBR98] and freely available solutions [BDF+03] have shown the maturity of the approach, and the ability to run several OSes on the same platform in isolation is now common practice in the industry. The approach is now viable even on handheld devices such as phones [Dev05].

The full virtualization allows the execution of an operating system, called guest, on top of another operating system, called host. For not changing the guest operating system, it must not be conscious of host operating system. One of the main issues of the virtualization is the execution of privileged instructions of the guest operating system in the unprivileged mode of the simulator application running on the host operating system. Another issue is the interception of I/O operations of guest operating system.

The classical solution for the full virtualization, called trap-and-emulate, is presented in Figure 2.4. It presumes the direct execution of the simulated machine binary code on the host processor. For doing this, the host and the guest processors must be identical. This solution is based on a *Virtual Machine Monitor* (VMM) which takes the control whenever a trap caused by a privileged operation executed in the unprivileged context of the virtual machine occurs.

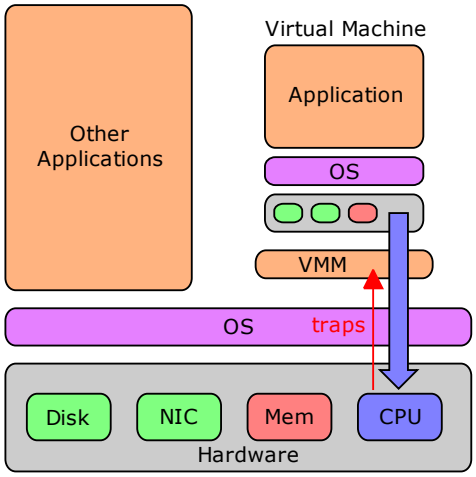


Figure 2.4: Trap and emulate based virtualization

These techniques have been increasingly worked on, and are now able to execute very

efficiently close to any software code compiled and linked on a given computer with a given processor on an other computer with an other processor with very few, if any, modifications. This efficiency relies on dynamic optimization [BDB00] and dynamic recompilation [Bel05].

The goals of the hardware/software co-simulation and virtualization efforts are indeed very different, and each community seems to have ignored the advances of the other.

Due to the constraints of consumer system integration (form factor, packaging, power, heat *etc.*), each application or class of application still calls for a specific circuit in order to fit into the performance/power budget. In that context, being able to rapidly explore the design space taking into account the specificities of the application is a necessity. Therefore, the MPSoC design approaches aim firstly at clearly separating computation from communication, using interfaces that are standardized, allowing to quickly exchange one IP by another and to add or retrieve a processor. Secondly, these approaches aim at producing figures of merits, such as code sequence run times and interrupt latencies, used bandwidth on an interconnect, even energy or power information, depending of the architectural choices. So it is necessary to have structural view of the platform for easy modification, extension and analysis.

If modularity is of primary importance in MPSoC design, it is not the case for virtualization that targets a single one shot hardware platform, usually uniprocessor, with an as high as possible execution speed. In order to achieve the appropriate performance level, the virtualization platforms that run cross-compiled software rely on dynamic translation of instruction or dynamic recompilation [HCKT99]. The idea is to dynamically, *i.e.* during the actual code execution, translate the original (cross-compiled) binary code into native (host) binary code on a basic block per basic block basis, and to preserve the translated basic block in a large buffer, which acts as a cache when full, to avoid paying the translation overhead again.

The binary translation approach is depicted in Figure 2.5. The translation from the guest to host binary code is performed even if the guest and the host processors are identical. This way the privileged instructions of the guest binary code are replaced by unprivileged instructions and the expensive trap mechanism is avoided. The QEMU framework [Bel05] is a widely used example of this kind of emulator.

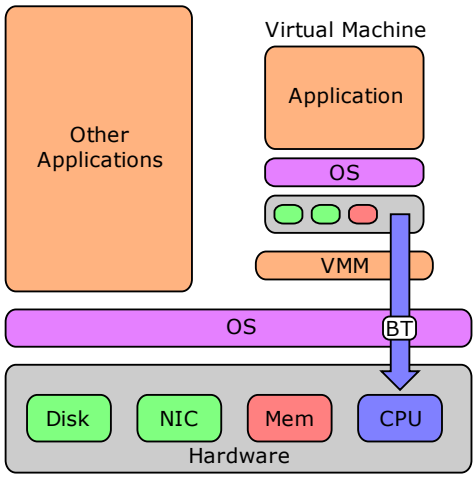


Figure 2.5: Binary translation based virtualization

We believe that, due to the increase in the number of programmable cores and software

in the near to come SoCs, the availability of virtual platforms providing a structural view of the system and fast application and OS code execution with a reliable accuracy is an important issue.

2.4.1 Questions

- How to combine the accuracy of the transaction accurate simulation models with the speed of the untimed virtualization approaches?

- Is it possible to reduce the number of transactions over the interconnect for trading-off accuracy for speed?

2.5 Static scheduling

The simulations at the cycle accurate level are usually performed using an event driven simulator. The dynamic scheduling simulators offer a great flexibility in hardware modeling, but they are slow. For speeding up the simulation, static scheduling versions of the simulators have been constructed. These simulators impose several constrains in order to assure that a static scheduling can be build. As an example of constraint, all processes of all hardware components must be sensitive to the edges of a unique system clock. The simulation speedup is due to the fact that the static scheduling simulators do not have to recompute at each moment the list of processes to be called. The static scheduling simulators are around five times faster than their dynamic versions, depending on the ratio between the number of generated events and the number of instruction executed for these events.

A large number of hardware/software co-design designers use simulators based on static scheduling for taking advantage of their speed. However, many designs require some hardware components whose modeling is impossible under the constraints imposed by the static scheduling simulators. A simple yet useful example is the components which use dynamic voltage and frequency scaling (DVFS) techniques for saving energy. These techniques require the change of the processors frequencies under the software control, and thus a static schedule can not be used.

Figure 2.6 presents one of the difficulties that a static scheduling simulator would face if it simulates an architecture containing a clock signal source whose frequency can change at runtime. In the figure, several processes of the architecture hardware components depend on each edge of this clock. P_i represents the set of processes depending on the edge i of the clock, where i is p for the positive edge and n for the negative one.

At moment t_1, the clock signal source reduces the frequency by a factor of three. Being sensitive to the events generated by the clock edges, in a dynamic scheduling simulator the processes will be call correctly all the time, even after a frequency change. As the scheduling of a static scheduling simulator is computed before the start of the simulation, the processes will be called using the same pattern even after the frequency changes. As there is only one clock in the system and all processes are dependent on its edges, while the calling order of the processes remains correct, the calling moment in the simulated time will be wrong. In the figure, processes that should be executed at t_4 are executed at t_2, those at t_7 are executed at t_3 *etc*. In fact, the frequency changes are intrinsically ignored by the static scheduler.

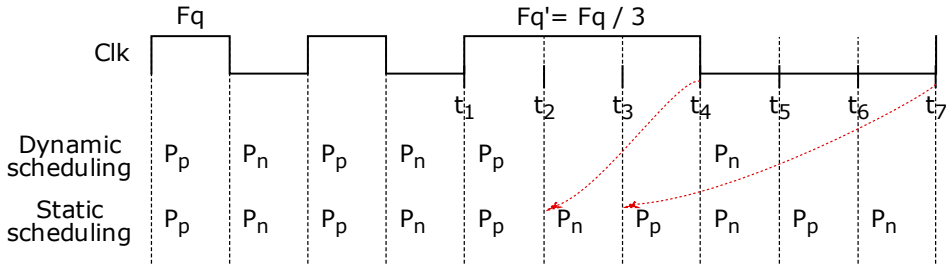


Figure 2.6: Static scheduling accuracy problem

2.5.1 Questions

- How to statically schedule the processes of the architectures having components working at different frequencies?
- How to support the runtime change of these frequencies without reducing the simulation accuracy?

2.6 Energy bases

In order to reduce the energy consumed by the electronic systems, we have to know the sources of the power dissipation. Most electronic systems nowday are implemented using the CMOS (complementary metal–oxide–semiconductor) technology. This section presents the power dissipation sources in the CMOS technology and the relation between power and energy.

For understating the causes of the energy consumption, we have to descend to the transistor level. Transistor is the basic element with which the logic gates are constructed. The inverter is the simplest CMOS logic gate. Its internal structure containing two transistors is depicted in Figure 2.7. Its behavior is very simple. If the input signal has the voltage corresponding to logic "1", the P-MOS transistor is closed and the N-MOS is opened, thus the output will be logic "0". Whether the input signal is logic "0", the state of the transistors is inverted and the output gives logic "1". We use this gate for presenting the sources of the energy consumption in the CMOS technology.

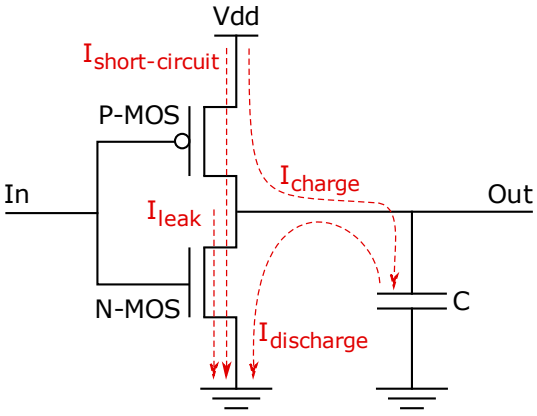


Figure 2.7: CMOS power consumption

The power dissipated on a CMOS circuit is composed of two basic components: static and dynamic (Equation 2.1). While the dynamic part of dissipated power depends on the activity generated at the gate input(s), the static part is independent of this activity.

$$P_{CMOS} = P_{static} + P_{dynamic} \tag{2.1}$$

When the input signals of a gate are stable, the outputs of the gates are also stable. In this case, one of the N-MOS and P-MOS transistors is closed while the other is opened. As there is no open path from source to ground, the CMOS circuits should not dissipate power while there is no change on the input signals. However, a leakage current through the closed transistors always exists. This current depends on the threshold voltage, the technological process and the temperature of the die at a given moment. This current determines the static component of the dissipated power. The value of this power can be computed using the Equation 2.2, where V_{dd} is the supply voltage and I_{leak} is the leakage current. In Figure 2.7, the N-MOS transistor is transversed by a leakage current when the input signal of the inverter gate is "0". The static power is today about two orders of magnitude smaller than the total power. For the modern designs using transistors with reduced threshold voltage, the static power is more significant. A good overview of leakage and reduction methods is given in [NC05].

$$P_{static} = V_{dd} \times I_{leak} \tag{2.2}$$

Switching between the logic levels determines the dynamic component of the CMOS dissipated power. This dynamic component has two main sources: the short circuit current and the charging of the output capacitors.

When an output signal of a logic gate changes its logic level, during an interval of time both N-MOS and P-MOS transistors are opened. A short circuit between the source and the ground is created while the voltage of the input signal is about the value of the threshold voltage. The short circuit current is noted with $I_{short-circuit}$ in Figure 2.7. The dissipated power caused by a short circuit is given by Equation 2.3, where τ represents the transition time and K a technological constant. This power represents 10 – 15 % of the total power consumption.

$$P_{short-circuit} = K \times (V_{dd} - 2 \times V_{th})^3 \times \tau \tag{2.3}$$

An output of a logic gate can be viewed as a capacitor that has to be charged for changing the logic value of that output from "0" to "1". A transition from logic "0" to logic "1" of the output signal charges the output capacitor with the energy $1/2 \times C \times V_{dd}^2$ drained from the source through the current I_{charge} created through the opened P-MOS transistor. During a transition from logic "1" to logic "0", the output capacitor is discharged to the ground through the current $I_{discharge}$ created through the opened N-MOS transistor. The total dissipated power due to the charging of the capacitors is depicted in Equation 2.4, where C represents the capacitance of the output, α represents the switching activity of the signals, f represents the frequency of the clock at which the synchronous circuit is working. $\alpha \times f$ gives the average number of "0" to "1" transitions in a second. This component of the power dissipation represents around 85 – 90 % of total power consumption in a CMOS circuit.

$$P_{capacitor-charge} = \frac{1}{2} \times \alpha \times f \times C \times V_{dd}^2 \tag{2.4}$$

The presented types of dissipated power are valid for any logic gate and even for the

entire CMOS circuits, not only for the inverter gate.

The power dissipated on a CMOS circuit is often approximated to the power consumed for charging of the output capacitors (Equation 2.5).

$$P_{CMOS} \approx P_{dynamic} \approx P_{capacitor-charge} \sim \alpha \times f \times C \times V_{dd}^2 \tag{2.5}$$

Between these factors of the Equation 2.5 there are some dependencies. For instance, for a given supply voltage there is a maximum clock frequency that can be used. This dependence is depicted in Equation 2.6, where V_{bs} represents the substrate voltage, $\alpha_1 > 0$, $\alpha_2 > 0$ and $\beta \simeq 1$ for the current technology [MFMB02]. So the operating frequency is linearly dependent on the supply voltage.

$$f_{max} \approx (V_{dd} - \alpha_1 - \alpha_2 \times V_{bs})^{\beta} \simeq V_{dd} - \alpha_1 - \alpha_2 \times V_{bs} \tag{2.6}$$

Low power techniques try to reduce one or more of the factors of the Equation 2.5: switching activity, capacitance of the capacitor, clock frequency and supply voltage.

The energy consumed by an electronic system during a period of time is given by Formula 2.7.

$$E = \int_0^t P(t)dt \tag{2.7}$$

Both factors, the dissipated power and the time required for the execution of a given task, have to be taken in consideration by a technique that tries to reduce the energy consumption. Lower power does not necessarily imply lower energy consumption. For example, by halving the working frequency, the power is also halved (Equation 2.5), while the time required for the execution of the given task doubles. So, the simple reduction of the frequency does not save energy for the execution of the given task, but only slows down the execution. According to Equation 2.6, lowering the frequency enables the decreasing of the supply voltage. By using together the voltage and the frequency scaling, the energy consumption may be reduced at the price of a slower execution.

2.7 Energy saving algorithms

The energy can be saved at both software and hardware level. Although at the both levels the energy is saved by reducing the factors implied in Equation 2.7, the methods used at the two levels are different. For instance, the switching activity (α) can be reduced at the hardware level by lowering the number of clock nodes, by reducing the number of false transitions before the correct value of the signals is stabilized *etc*. For processors, the same switching activity factor can be reduced at the software level by an energy efficient code compilation that includes reordering instructions, operand swapping, optimized compilation *etc*.

Sometimes the hardware and the software work together for achieving the energy saving. It is the case of the hardware components supporting several running modes. The running mode of these devices is changed by the software executed on the processors whenever this software considers that the change is applicable. For instance, the software can reduce gradually the luminosity of the system display supporting this feature when there is no activity in the system. Also, when the system utilization permits, the software can reduce the voltage and frequency of the processors in a DVFS scheme.

The hardware and the software levels face their own difficulties when trying to reduce the energy consumption.

The hardware problems are mainly related to the interdependences between the factors of Equation 2.5. For instance, the device capacitance and leakage current can be reduced by decreasing the device size, but this leads in the end to reducing the operating speed of the device.

The software difficulties concerning the energy saving through a DVFS scheme are linked to the decisions about *when* and *how much* to scale the frequency and the voltage of the processors. These decisions are taken usually at the operating system level. The running applications may or may not help the energy saving algorithm in taking these decisions by providing information about their computation demands.

The application execution delays caused by the frequency reduction must be acceptable for the running application(s). A pessimistic answer to these questions reduces the saved energy. Depending on the application and operating system type, the delays caused by a too optimistic answer could lead even to a total failure of the system. There are two main types of operating systems: real time (RTOS) and non real time.

2.7.1 RTOS

The RTOSes are intended for real-time applications. The main characteristic of the real-time systems is that they have to respond to certain stimuli within a finite and specified delay [BW01]. For these systems, the correct behavior depends not only on the computation results, but also on the time at which these results are obtained [SRS98]. For meeting the required time constrains, the RTOSes allow the controlling of the tasks priorities. The RTOSes can be hard or soft. While a hard RTOS must always meet the tasks deadlines, a soft RTOS can miss occasionally a task deadline.

As the operating system knows the deadlines of the tasks, we can say that the applications always offer the required information for the voltage and frequency scaling. An energy saving algorithm running on a RTOS has all information required for avoiding missing tasks deadline and detecting the overrun of a deadline.

2.7.2 non-RTOS

The application tasks in a non-RTOS do not have a deadline. A little longer execution of the applications caused by the frequency reduction does not represent a problem. However, for most applications, a sensitive deadline can be defined. These deadlines are not defined at task level, but at the entire application level. These deadlines represent the limit from which the applications delay becomes visible to the exterior. For instance, if an operation with a visible result takes one minute instead of one second, we can say that the sensitive deadline of that operation was missed. Unfortunately, the operating system is not conscious of these deadlines and it can not detect whether they have been missed.

By default, a non-RTOS knows nothing about the computation requirements of the application. An API can be defined for allowing the applications to provide their information. In this case, the DVFS decisions can be taken as in the RTOS case.

If the applications do not provide any information, the energy saving algorithm has to manage all by itself. For this, the energy saving algorithms usually computes the utilization (workload) of the system and changes the processor frequency accordingly. While the processor works at a reduced frequency, the algorithm can not determine how the execution would develop at the maxim frequency. Thus, the execution could be stretched without the algorithm realizing it. So, the main difficulty in saving energy in a non-RTOS, even for the mono-processor systems, is to detect whether the execution of the tasks was too stretched and the system performances diminished too much.

For the SMP architectures the number of problems increases compared to the mono-processor systems. The new problems come from the tasks migration between processors and from the difficulty of differentiating between the idle time caused by the lack of activity and the idle time caused by the synchronization between tasks.

Although the non-RTOSes are the most used operating systems, there are very few proposed energy saving algorithms for them.

2.7.3 Questions

- How to build an energy saving algorithm running on a non-RTOS and dealing with the particularities of the SMP architectures?

2.8 Conclusion

We summarize here the questions at which this thesis tries to answer.

For improving the simulation speed at the TA abstraction level by replacing the slow interpretative ISSes, the following questions can be posed:

- How to combine the accuracy of the transaction accurate simulation models with the speed of the untimed virtualization approaches?
- Is it possible to reduce the number of transactions over the interconnect for trading-off accuracy for speed?

For taking advantage at the CA abstraction level of the speed of the static scheduling based simulators for all SoC architectures, the following questions arise:

- How to statically schedule the processes of the architectures having components working at different frequencies?
- How to support the runtime change of these frequencies without reducing the simulation accuracy?

Regarding the saving energy at the software level, the question is:

- How to build an energy saving algorithm running on a non-RTOS and dealing with the particularities of the SMP architectures?

Chapter 3

State of the art

Contents

This chapter presents the existing solutions that partially respond to the questions that we previously asked ourselves.

We will first depict the works related to the simulation questions. The chapter begins by presenting the approaches that use more or less binary translation techniques for software execution coupled with event driven simulation, and continues by introducing a few simulators based on a static scheduling approach.

We will present then the state of the art of the energy saving algorithms designed for the non-RTOSes.

3.1 System simulation making use of binary translation based ISSes

Some solutions taking advantage of the multi abstraction level simulation have been already proposed [HCKT99]. But in almost all the works investigating the SoC simulation, the simulation of processors is accomplished by the mean of an Instruction Set Simulation (ISS) based on the interpretive technology. This kind of ISS has the great advantage of being flexible enough to allow an easy trade-off between accuracy and simulation time. This trade-off can then lead to cycle accurate simulators simulating in details the behavior of the processor pipeline [ACG+07, PPB02], but at the high price of a very slow simulation.

Only few works propose SoC simulation based on other ISS technologies than the interpretive one. Compiled simulation offers a low flexibility, thus it presents little interest. The binary translation technology is instead of a great promise.

As it can be realized from [CM96], binary translation can hardly be classified in the previously presented abstraction levels without modifications. A set of modifications are

required for getting an ISS based on this technology closer to these abstraction levels, more precisely closer to the transaction accurate and cycle accurate levels.

Schnerr *et al.* investigates in [SBR05, SHR03] the use of binary translation for cycle accurate SoC prototyping. The main goal of this work is to perform hardware assisted prototyping. This means that they use a specific processor, different from the target one but dedicated, for emulating the target processor and the real hardware for the rest of the hardware components. The emulation system is presented in Figure 3.1. In this way, they use static binary translation to produce target code running on the dedicated VLIW (very long instruction word) processor. The FPGAs contain a synchronization device and the bus interface that adapts the bus of the VLIW processor to the SoC bus of the emulated processor core. The synchronization device generates the clock cycles for the attached hardware.

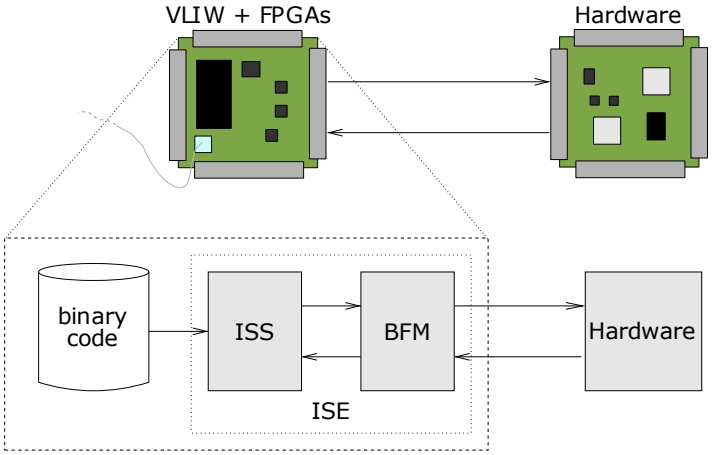


Figure 3.1: Architecture of the emulation system presented in [SHR03]

For interacting with the rest of the hardware, [SHR03] annotates the translated code. At the beginning of each basic block it is inserted a write instruction that informs the synchronization device about the number of cycles the target processor would require for the execution of that basic block. The synchronization device and the VLIW processor execute in parallel and synchronize together at the end of the basic block. The I/O operations are replaced in the translated code by instructions accessing the hardware of the bus model. As a local memory is used for the simulated main memory, the estimated number of cycles required by the instruction cache miss is also issued to the synchronization device. The data cache is not modeled.

This approach is closer to a normal hardware execution than to a simulation. The only hardware component that is simulated is the processor, the rest of them are the real ones. The parallel execution is not managed by a simulation program, but by real hardware. This offers a great speed, but prevents architecture exploration. By using a local memory and static binary translation, the accuracy of the simulated processor is not the maximum.

Our goal is to simulate the entire architecture, the target processors and the other hardware components, on a general processor.

[MPM+07] proposes a simulation environment that gives the designers the possibility of developing and testing the hardware modules and their associated software parts (drivers, applications) in early development stage. The Figure 3.2 introduces the architecture of the QEMU/SystemC plugin. QEMU is an emulator that can emulate many architectures. The characteristics of this emulator will be presented in section 4.2.

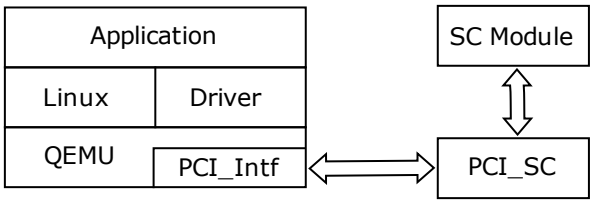


Figure 3.2: Architecture of the plug-in QEMU/SystemC presented in [MPM+07]

This proposal models only poorly the system (close to a transaction based model). The plugin does not modify the way the QEMU platform is simulated. The entire QEMU platform is simulated in the context of a single SystemC process. In the case of a multi processor platform, the processors are not simulated concurrently. Two simulated processors cannot communicate with two SystemC hardware devices at the same simulation time because the processors are simulated in the context of the same SystemC process and a SystemC synchronization issued by one processor blocks the entire QEMU platform. Furthermore the emulation by QEMU is purely functional. The time notion of the simulated platform remains unmodeled. The SystemC simulation time does not have any correlation with the code executed by each processor and the time required for the communication with other devices. A similar approach is presented in [FDW].

[JH08] proposes a full system simulation architecture, called *SimSoC*. SimSoC integrates ISSes as SystemC modules with TLM interfaces to the other platform components (Figure 3.3). The ISSes use dynamic translation for running the target binary code. The target binary code is translated by the dynamic translator into an intermediate representation. The intermediate representation is partly dependent on the target architecture and totally independent on the host (both machine architecture and operating system). This representation maps each target binary instruction to a specific function. For speeding up the simulation, SimSoC specializes the most frequently used target instructions. For example, for a target instruction using a register and an immediate value stored in the instruction, a specialized function can be generated for each pair of values of the register and the immediate value supported by that instruction. The generated specialized C++ functions corresponding to the unfolded instructions of a target processor and the generic C++ functions corresponding to the rest of the target instructions are compiled and included in a library.

During the simulation, SimSoC calls for each target instruction the corresponding function in the library. The simulator regains the control after each target instruction simulated. The SimSoC ISS calls the SystemC *wait* function and verifies the interrupts after simulating a predefined number of target instructions. The goal of the SimSoC ISS is to simulate the behavior of the target processor with instruction accuracy.

For each instruction simulated, SimSoC verifies whether the target instruction at the address given by the program counter of the ISS has already been decoded. This translation and management at the instruction level correlated with the fact that a function call is issued for each simulated target instruction reduces a lot the simulation speed. The accuracy level is also not very high. SimSoC obtains only the number of instruction simulated without providing the time required for executing these instructions. The instruction and data caches, write buffer *etc.* are not modeled and the processor subsystem is untimed.

[NBS+02] proposes an ISS simulation technique, called just-in-time cache compiled simulation (JIT-CCS). This technique is integrated in the retargetable LISA processor design platform [HSN+01]. A generator back-end for the LISA processor compiler has been de-

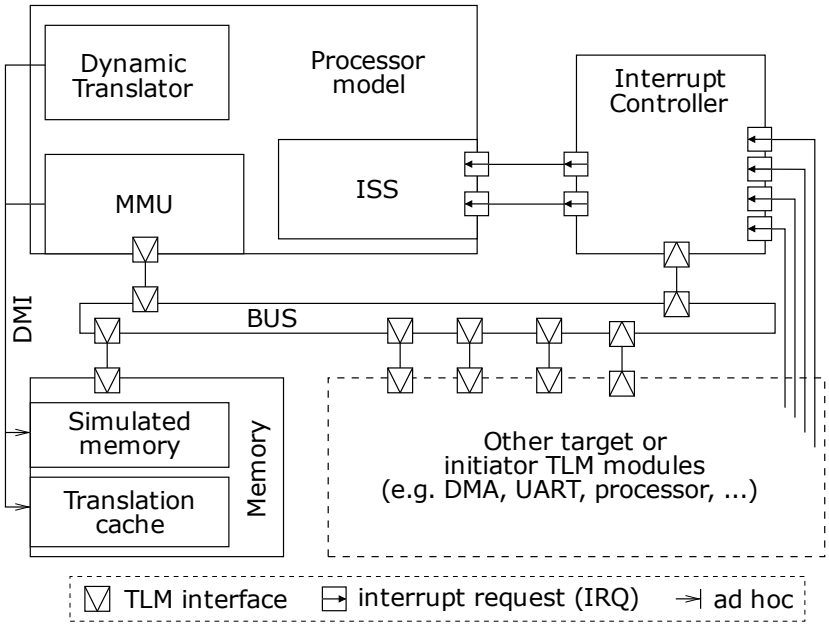


Figure 3.3: SimSoC architecture presented in [JH08]

veloped, which automatically constructs a JIT-CCS simulator from a LISA machine description. The processor instruction set including instruction coding, assembly syntax, functional behavior, and timing is contained in so-called LISA operations. A single target processor instruction can be composed by multiple LISA operations. The behavioral C code of all LISA operations is precompiled into C functions which are part of the simulator. The JIT simulation compiler selects the appropriate operations, which are required to simulate a target instruction. References to the selected C functions are subsequently stored in the simulation cache. These references are used by the simulator to execute the instructions behavior.

As [NBS+02] performs a dynamic translation of the target binary code at the instruction level. The pipelined execution of the processor is modeled at a cycle accurate level. The highly detailed modeling, the instruction level management of the simulation and calling a function for each operation of each simulated instruction reduce the simulation performance. The modeled ISS is simulated alone, not concurrently with other hardware component models of a full MPSoC architecture. No event driven simulator is used for the simulation.

A similar approach for the ISS modeling is presented by CoWare in [WKL+04]. [WKL+04] proposes a methodology to jointly design and optimize the processor architecture together with the on-chip communication based on the LISA Processor Design Platform in combination with SystemC Transaction Level Models. This methodology advocates a successive refinement flow of the models of both the processor cores and the communication architecture. LISA provides capabilities for automatic generation of code generation tools (C compiler, assembler, linker *etc.*) as well as the ISS and the profiling environment. Using LISA, the processor architecture can be modeled at the instruction or cycle accurate level. The instruction accurate model is not cycle accurate since pipeline effects are not considered at all. In contrast, cycle accurate processor models fully simulate the processor pipeline.

This approach offers a trade-off between accuracy and simulation performances. The

simulation management is required before each simulated instruction and a function is called for each operation of the simulated instructions. It would be interesting to have an accuracy level between the two proposed, where the processor model would take into account the pipeline effects without fully modeling the pipeline execution.

3.2 Static scheduling

Thanks to its flexibility that allows the easily handling of the both synchronous and asynchronous types of design, the event driven simulation technique is used in the implementation of many simulators. A dynamic scheduling determines at each simulation moment the active components that have to be evaluated. This continuous scheduling leads to a slow simulation. Although much effort has been made, the simulation speed remains a major problem of this technique.

Since the early '70s, for increasing the simulation performances, many researchers have tried to replace the event driven simulation technique with a static scheduling based simulation technique whenever it was possible.

[WHPZ87], [WM90] and [Han88] have proposed in '80s the levelization technique which enabled the use of static scheduling to simulate the synchronous circuits composed of a set of logic gates. This technique groups the logic gates of the design in one-level deep, two-level deep *etc*, starting from the primary inputs and going to the outputs of the circuit. The logic gates whose inputs are all primary inputs constitute the first level. Figure 3.4 presents the levelization of a XOR gate. The description of each gate is translated into a C code. These code sequences are ordered according to the depth level of the gates. This ensures that whenever a logic gate is evaluated, the correct values of its inputs are available. Every gate is evaluated once during each clock cycle, independently of the input pattern dynamics. So, the simulation performances of this approach do not depend on the gate activity in the design. Since this approach eliminates the need for event management, it is extremely efficient.

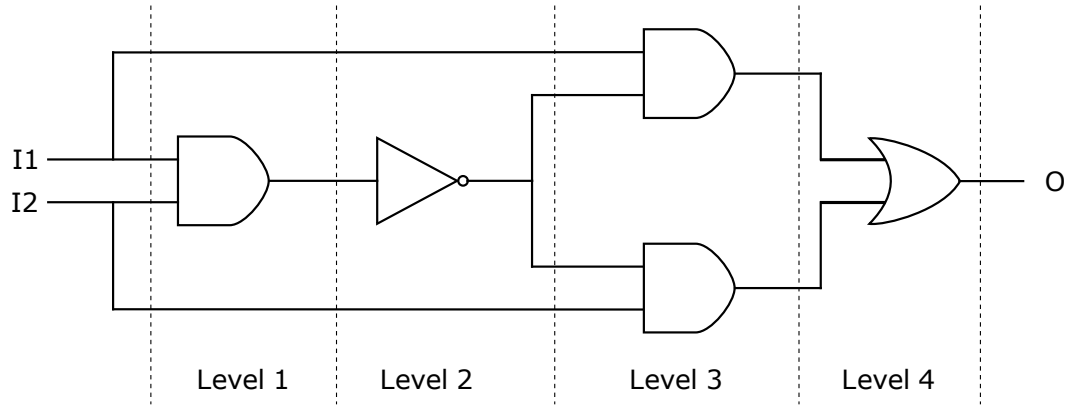


Figure 3.4: 2-input XOR gate levelization

In [WHPZ87], the synchronous design is first compiled by a netlist compiler into a netlist. A code generator, called *SIMGEN*, generates the levelized C code and then compiles it. For every clock cycle, the entire program is executed once. For constructing the levelisation, [WHPZ87] uses the following algorithm. The nets connected to the primary inputs and to the latches outputs are marked as available and the fan-out gates of these

nets are inserted into a queue. The gates having all inputs available are removed one after another from the queue. When a gate is removed, its output nets are marked as available, the C code corresponding to that gate is generated and the gates connected to that gate output nets are inserted in the queue. If not all gates could be extracted from the queue, an error is reported as the circuit contains feedback loops.

The compiled levelized simulation can not handle asynchronous circuits. It can not be applied to the circuits containing feedback paths such as a RS latch. This limits its application to combinational and preidentified synchronous circuits.

[WM90] addresses these problems by proposing a simulator, called *LECSIM*, which combines the levelized technique with the event driven simulation using a compiled implementation. LECSIM uses a standard depth-first search algorithm for identifying the strongly connected components. A segment of a circuit is strongly connected if it is connected and the output of every gate in the segment depends on the output of every other gate in the segment. Within a strongly connected component, each fan-out branch of each net is identified as either a forward path or a feedback path. LECSIM levelizes each strongly connected component, ignoring the feedback paths.

For the simulation, a list of blocks (gate or strongly connected component) that have to be evaluated is maintained for each level of the levelized architecture. The blocks are evaluated according to their level. After evaluation, they are removed from the lists. When an evaluated block generates an event, its fan-outs blocks are inserted in the corresponding lists. If the level of an inserted block is lower than the level of the block that caused the insertion, the circuit is marked as unstable. After finishing the current lists evaluation, a new evaluation begins if the circuit is not yet stable.

[Jen91] readopts the completely static scheduling, where the components models are scheduled before simulation begins. It assumes a single system clock and that the components are D-flip-flops or combinational (unclocked). A directed cyclic bipartite graph is created for the circuit. An example of such graph is depicted in Figure 3.5, where D_i name the flip-flops, C_i are combinational and Net_i are the nets. This graph contains two types of vertices: components, drawn as rectangles, and nets, drawn as ellipses. The vertices are joined by arrows, called edges. Nets with no input edges are circuit inputs and nets with no output edges can only be circuit outputs. For the simulation, combinational components will become subroutines and nets will become static variables. The D flip-flop model consists of three subroutines and two variables for each flip-flop instance. These variables store the current and the future value of the flip-flop. The D flip-flop subroutines offer the possibility of reading the current value, setting the future value and copying the future value into the old value. A component is scheduled by generating a call of its subroutine with the relevant nets passed as parameters. The static scheduling is formed of an ordered list of subroutine calls. The static scheduling is obtained using three recursive functions that implement a depth-first search algorithm. [Jen91] does not accept asynchronous, purely combinational, cyclic paths in the circuit.

[BPG04] and [HP98] propose a cycle accurate simulator, called *SystemCass*, which uses static scheduling. SystemCass is usually used for simulating architectures at system level, not at gate level as the previous simulators. The hardware architectures are modeled using a set of communicating finite state machines. The hardware components composing the architectures are modeled with one or more of these FSMs. Each FSM is composed of a set of functions, which model the behavior of the component at different moments of the clock cycle. Some of these functions are sensitive only to the system clock edges, while others are sensitive to the negative clock edge and also to the input ports of the component. Similar to the D flip-flops modeling in [Jen91], the state variables in SystemCass have a current

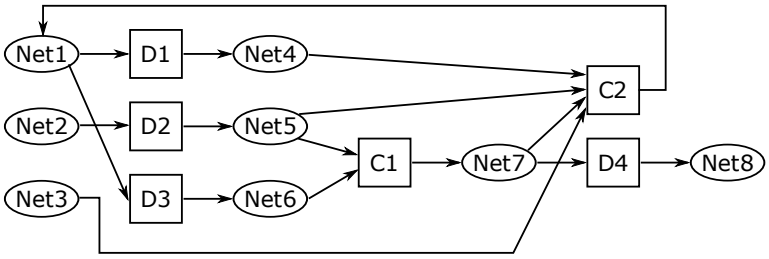


Figure 3.5: Bipartite graph example, presented in [Jen91]

value and a future value. The signals that connect the components are also modeled as in [Jen91], using a common variable for the current and the future value.

As the functions sensitive only to the clock edges can not create cyclic dependencies, they can be easily statically scheduled by calling successively all functions of all FSMs depending on the positive clock edge and then those depending on the negative edge. SystemCass builds a dependency graph containing the functions sensitive to the inputs. The dependency graph is used for constructing the static scheduling of these functions. Cyclic dependencies may exist between these functions (*e.g.* function f_1 controls a signal on which function f_2 depends and function f_2 controls a signal on which function f_1 depends). The functions not taking part of a dependency cycle are executed only once, after all their dependencies have been executed. The functions forming a dependency cycle are executed in a loop until they stabilize. This way, SystemCass is able to efficiently handle combinational loops existing between the FSMs.

All presented simulators can statically schedule only the systems using a single clock signal. As far as we know, there is no simulator able to statically schedule hardware architectures containing components whose working frequencies can change at runtime.

3.3 Energy saving algorithms

Various solutions to the question of how to best take advantage of the different low power features of the hardware components have been proposed. These solutions try to minimize the energy/performance (joules/instruction) ratio by reducing the energy consumption and keeping the system performance as high as possible.

DVFS is the main mechanism used for reducing the consumed energy under the software control. This mechanism is widely exploited in RTOSes but less studied in non real-time systems. This is unfortunate, because many applications that have time constraints cannot be efficiently modeled with the RTOS template.

For RTOS, many works propose frequency adaptation algorithms using DVFS capabilities in off-line static scheduling whether in mono-processor systems [SR03, YCHK07] or in MPSoC systems [YWM+01]. These works achieve the frequency assignment by using detailed information on the task graph running on the systems, such as tasks arrival dates, worst case execution times (WCET) and periodicity. On top of that, they often rely on strict assumptions concerning task independence and communication predictability.

For instance, the *proportional* algorithm [XMM05] adjusts the speed for each task individually and allocates them a time proportional with their WCET. The *Greedy* algorithm [XMM05] gives to the next task its WCET plus all idle time remained in the period. [LS04] demonstrates that it is not optimal to keep the speed constant during the execution of a

task and proposes the *PACE* algorithm, which gradually increases the processor speed for the current task in the hope that it will not require its WCET.

All the information used by these RTOSes frequency assignment algorithms is not available when using a non-RTOS.

3.3.1 Monoprocessor

The non real-time OS has to make decisions dynamically about how to save energy without reducing too much the system performances.

Some approaches [HK03] add information about the characteristics of the tasks, basic blocks, memory boundedness *etc.* when the programs are compiled. These characteristics are then used at runtime by the operating system for setting the frequency. Isci *et al.* [ICM06] uses memory boundedness without requiring pre-runtime computations. It identifies the phases with distinct memory characteristics using hardware event counters. The future workload is predicted, and then the frequency is set accordingly, based on the patterns of these characteristics.

Varma *et al.* [VGS+03] uses PID (proportional-integral-derivative controller) to predict the future workload. A drawback of using a PID controller is given by the need of a correct parameterization. The extensive parameterization of the algorithm determines the algorithm to become tailored to a certain system.

In their *PAST* algorithm, Weiser *et al.* [WWDS94] divides the execution time in intervals of the same length. For each interval, the number of non-idle cycles executed (workload) in that interval is computed. Considering that the next interval will have a close number of non-idle cycles, the algorithm tries to eliminate the pure idle cycles by modifying the processor frequency. The processor frequency is set at the beginning of each interval.

Listing 3.1 presents how the PAST algorithm computes the system utilization. *idle_cycles* is the processor idle cycles, split between hard and soft idle time. The hard idle time is composed of waitings for hardware responses (*e.g.* disk). The soft idle time is produced by waitings for keyboard events or network packets. The number of non-idle processor cycles in the last interval (*run_cycles*) comes from two sources, the runtime in the trace data for the interval, and the cycles left over from the previous interval because the processor speed was too low to accommodate the entire load that was supplied during that interval (*excess_cycles*). Only soft idle is available for elimination.

Listing 3.1 PAST run percent computation [WWDS94]

```
1 idle_cycles = hard_idle + soft_idle;
2 run_cycles += excess_cycles;
3 run_percent = run_cycles / (idle_cycles + run_cycles);
4 next_excess = run_cycles - speed * (run_cycles + soft_idle)
```

The energy saved is minimal because the algorithm keeps oscillating between two frequencies instead of using the optimal frequency between the two.

Govil *et al.* [GCW95] refined the PAST algorithm by splitting it in two parts: prediction and speed-setting. The first part predicts at the beginning of each interval the processor workload for that interval. The second part sets the processor frequency according to this value. In the same paper, Govil *et al.* tried to enhance the prediction part: *Flat* algorithm presumes that processor utilization has always a constant value; *Long short* and *Aged averages* algorithms compute the workload as a weight sum of terms calculated in different

ways; *Cycle*, *Pattern* and *Peak* algorithms try to exploit the fact that the processor utilization is periodical.

There are different algorithms for the speed-setting part. In [WWDS94], the processor frequency is computed with different formulas, depending on the interval the processor predicted utilization value is situated in. In all algorithms presented in [GCW95], the frequency is computed with a linear formula (predicted utilization multiplied by maximum frequency). Grunwald *et al.* defined in [GIL+00] different methods: *Peg* (sets the clock at maximum/minimum frequency), *One* (increments/ decrements the clock value by one step), *Double* (doubles/halves the clock step). The length of the interval can have different values.

3.3.2 Multiprocessors

Juang *et al.* [JWP+05] proposes a distributed DVFS algorithm that tries to identify the threads that lie on the critical path. These threads should be run at maximum speed to preserve the performance, while others slowed to maximize the saved energy without impacting the performance. The frequency can be set individually for each processor.

[MB08] applies DVFS to the processors running memory-bound threads. The approach is based on the fact that, in this situation, the processors are mostly stalled due to the congestion on the common memory bus and thus reducing the frequency will have little impact on their performance.

[DPR08] uses the deep sleep states of the processor instead of using DVFS. Their experiments show that by executing the useful work always at the maximum frequency and then switching the processors to deep sleep during the idle periods, more energy is saved than by executing the useful work at a lower frequency and voltage.

3.3.3 Monoprocessor and multiprocessors

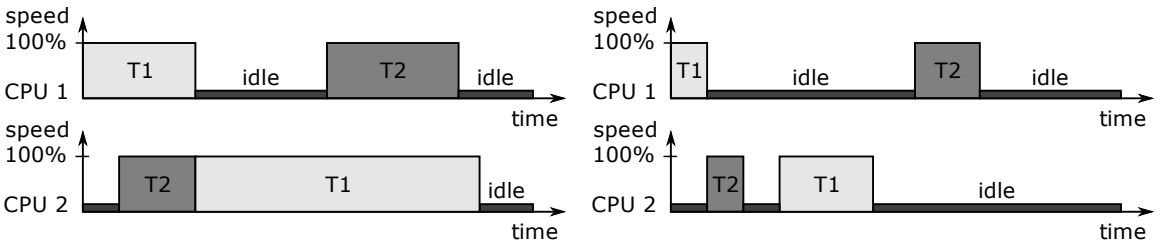
Some simple techniques are applicable to both monoprocessor and SMP architectures.

[BR03] proposes a simple technique that saves processors energy by reducing to minimum the processors frequency during the idle time. The processors are stalled if they have a *sleep* state.

Figure 3.6 presents the behavior of the technique. In Figure 3.6(a), the tasks execution is the longest possible, *i.e* WCET. The tasks execution time decreases in Figure 3.6(b). The tasks are always executed at the maximum frequency and the idle task always at the minimum. The difference between the actual execution time and the WCET is replaced by idle time. The code overhead can be approximated to zero. No future or past knowledge about applications demands is required. Applying this simple technique, no sensitive deadline will be missed as the tasks are executed at the maximum speed.

[GML+00], [BR03] and [She04] propose energy saving techniques that use information provided by the applications for setting the processors frequencies. The applications inform the power control element situated at the OS level about their computation requirements using an ad-hoc API.

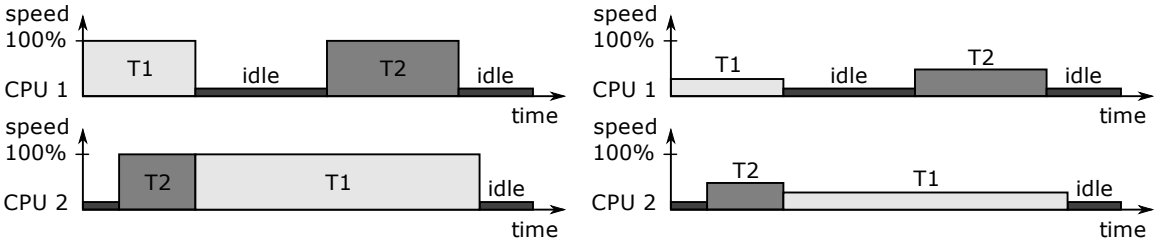
An example of how this technique works is depicted in Figure 3.7. In Figure 3.7(a), both tasks inform the power control element that they need full speed. Consequently, the power control element keeps the frequency of both processors at the maximum value. In Figure 3.7(b), task 1 and task 2 requires one third and task 2 only half of their maximum workload. When task 1 is scheduled on either of processors, the frequency of that processor is set at one third of the maximum frequency. Similarly, for task 2, the frequency is divided



(a) $T1$ and $T2$ at WCET (b) $T1$ and $T2$ not at WCET

Figure 3.6: Saving energy by reducing the speed on idle

by 2. The idle periods are executed at the minimum frequency, as in the reduce speed on idle technique.



(a) $T1$ and $T2$ at WCET (b) $T1$ and $T2$ not at WCET

Figure 3.7: Saving energy using the application information

The energy is saved due to the execution of the tasks at a lower frequency and voltage, not because of stretching the tasks execution over the idle time. The saved energy is superior to that saved using the reduce speed on idle technique as, for a given workload that has to be done in an interval of time, it is better to execute all the interval at a medium frequency than to execute a part of it at the maximum frequency and the rest at the minimum one. The sensitive deadlines are still met if they were met for the case when all tasks need their WCET. The disadvantage of this technique is that the applications must be aware of their computing necessities.

3.4 Conclusion

We have presented in this chapter the main solutions proposed for the two axis of this thesis: energy saving algorithms and the simulation strategies allowing the validation and the simulation of these algorithms.

The presented translation based ISSes are not very fast because they do not translate the binary code of the target processor into the binary code of the host processor, but into an intermediate representation. For each simulated instruction, one or more functions are called. Most of the presented ISSes are not used in a complete multiprocessor architecture simulation.

There is no proposal for static scheduling of the architectures containing multiple and runtime changing frequencies.

There are only few proposed algorithms that try to save energy in non real-time OSes running on SMP architectures.

The following two chapters will present our solutions for the simulation strategies and the DVFS algorithm for the SMP architectures.

Chapter 4

Using binary translation based ISS in event driven simulation

Contents

The first proposed simulation strategy tries to combine the speed of the binary translation based ISSes with the accuracy of the event driven simulators. To have an accurate timing behavior for an instruction set simulator based on binary translation, we had to first solve timing issues in processor modeling, second define fast and precise cache models, and third solve the synchronization issues due to the different models of computation used in the ISSes and in the rest of the system. We present an integration solution that covers these issues and detail its implementation.

The simulator is presented from the multiprocessor, time, frequency and energy modeling points of view. Even though our approach is applicable to any binary translation based ISS, we use the ISSes from QEMU and SystemC TLM as simulation environment for the whole platform.

Before presenting our simulation strategy, an overview of the binary translation is first given.

4.1 Binary translation bases

The binary translation represents the emulation of the instruction set of a processor by the instruction set of another processor using code translation. Sequences of simulated (also called target) processor instructions are translated into executing (also called host) processor instructions.

The translation can be performed statically or dynamically. In the static version, an entire executable file compiled for the target processor is converted through translation into an executable for the host processor. Due to the fact that the file is not executed as it is translated, it is possible to miss portions of code that are accessible only through indirect branches whose jump address can be obtained only at execution time. Self modifying code and runtime loaded programs can not be statically translated, and as such, these limitations make the approach unusable in our context. We, thus, focus on the dynamic approach.

The simulators based on dynamic binary translation perform the translation of a code sequence during the simulation when that code sequence has to be executed. The difficulties of the static translation do not pose problems to the dynamic one because at the execution time all addresses and registers contents are known. Also, an already translated sequence of instructions, but which has been meanwhile modified, may be translated again next time it has to be executed.

The dynamic binary translation can be performed in different ways. Figure 4.1 presents the general simulation model of the binary translation based simulators. The simulator verifies if the sequence of instructions starting at the address given by the program counter of the simulated processor has already been translated. In the case it was not translated before, the *binary translation* stage begins.

The instruction corresponding to the program counter of the simulated processor is *fetched* from the target binary code. The fetched instruction is then *decoded* into several host instructions. If the current instruction is not a branch instruction, the next target instruction is fetched and decoded. The binary translation stage ends after a branch instruction is decoded. The sequence of instructions together treated by the binary translation stage forms a translation block. The host instructions generated for a translation block are grouped together and stored into a *translation cache entry* of the translation cache. Once the generated code is *executed*, the simulator verifies the existence of the translation corresponding to the new program counter. If the required translation already exists in the translation cache, it is directly executed.

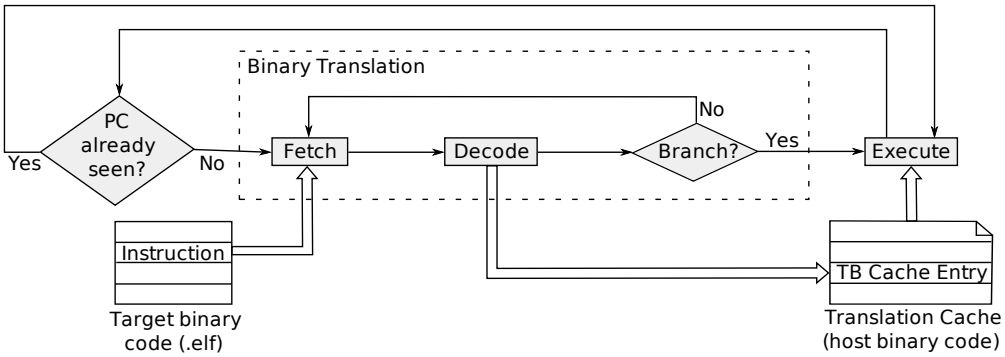


Figure 4.1: Binary translation simulation model

The binary translation based ISSes differ from the interpretative ISSes by the fact that the latter does not generate the host code and the fetching and decoding is performed each time an instruction has to be simulated. The idea behind dynamic binary translation is that the price paid for the host code generation will be amortized as the translated code sequences are executed several times.

The notion of translation block is similar in spirit to the *basic block* used by compilers, but they are not identical. Both translation blocks and basic blocks end after a branch instruction. Beside the branch instructions, there are other conditions that end only the translation blocks or only the basic blocks. As an example, a translation block also ends at the MMU page boundary of the target processor, because the page mapping can change during the simulation. Other conditions for ending a translation block include instructions generating exceptions (*e.g.* undefined instruction), change of the execution mode of the target processor etc. By definition, the basic blocks also end before the instructions which are the target of jump or branch instructions. In this case, a binary translation simulator would generate a new translation block starting at the jumping address. So, an instruction can be part of several translation blocks, but only of a single basic block.

Binary translator that makes the conversion from the target code directly to the host code is strongly dependent on both target and host processors. The change in one of the target or host processors requires a new translator. Considering that a simulator wants to be able to simulate N targets on M hosts, $N \times M$ translators are necessary.

4.1.1 Binary translation simulators using an intermediate representation

For avoiding the large number of translators that have to be implemented for simulating several targets on several hosts, some simulators use an intermediate representation (IR) for translation. Instead of being translated directly to host code, the target code is first translated to an intermediate representation common to all targets. The target code represented in the intermediate representation is then translated to the host code.

This way the target and host translations are independent. For adding a new target processor, it is necessary to create only a translator of the instruction set of that processor to the intermediate representation, no matter of the number of hosts on which the new target processor can be simulated. By adding a translator from the IR to a new host processor, all target processors could be simulated on the new host processor. So, for simulating N targets on M hosts, $N + M$ translators are required.

The execution of these simulators (Figure 4.2) is close to that presented in Figure 4.1. The target instruction *decoder* generates the IR corresponding to the translation block under

consideration. After the decoding of the target translation block ends, the *host code generator* creates the host code corresponding to this translation block for the executing host processor.

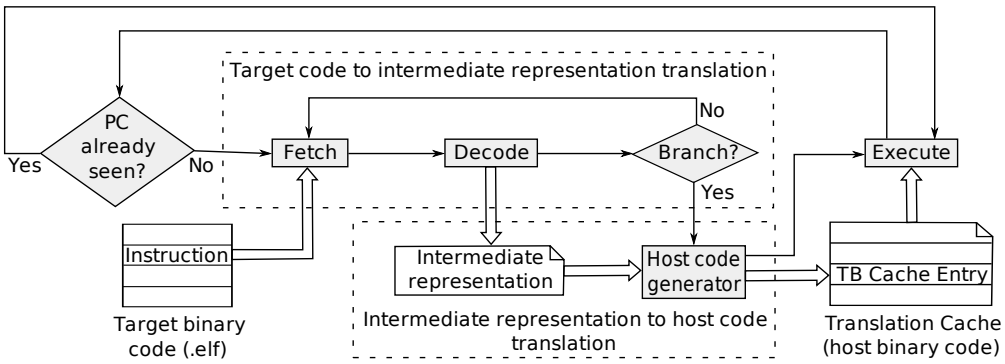


Figure 4.2: Binary translation using intermediate representation simulation model

The intermediate representation may consist of the language instructions of an abstract machine. In this case, each target instruction of a translation block is translated into several instructions of the intermediate language. The instruction set of the intermediate language is much reduced compared to the instruction set of the target processors. These instructions are called *micro-operations*. The abstract machine also has a set of "registers" used as temporary variables by the micro-operations.

For each micro-operation, a host code sequence is generated. However, there are simulators ([Law]) that interpret the micro-operations instead of generating the corresponding host code. Even though the interpretation can be very simple, these simulators are typically slower than the ones generating host code.

4.2 QEMU

In the following, and for concreteness reasons, we use the ISS from the QEMU framework to illustrate a system simulator that makes use of binary translation.

QEMU is a fast and portable emulator which emulates many architectures (X86, ARM, SPARC etc.) and runs on several architectures. It models the MMU of the simulated processors, but it does not model the processors caches. QEMU provides fast simulation performances: systems emulated by QEMU are about 5 to 20 times slower than native code execution. To simulate guest systems, QEMU uses the dynamic binary translation presented in the previous section and a dynamic code generation strategy, that we will detail below.

QEMU uses micro-operations as intermediate representation in the target to host code translation process. These micro-operations represent the instruction set of a simple 2-address abstract machine. This machine has only three general purpose registers (called T0, T1 and T2). The operations are the usual logic and arithmetic ones, plus load and store actions for the different modes (*e.g.* user, kernel) supported by the target processor. The set of micro-operations is different for each target processor. Table 4.1 gives an overview of the QEMU micro-operations which are relevant for an ARM processor. The *movl_T0_r0* micro-operation copies the register *r0* of the target ARM processor into register *T0* of the abstract machine. *movl_T0_r0* loads the long value from the virtual address given be the register *T1* of the abstract machine and stores it the register *T0* of the abstract machine.

addl_T1_im adds to the value of the register *T0* an immediate value. *bx* sets the program counter of the target processor to the address given by the register *T0*. *exit_tb* contains an assembly host instruction for returning from the generated function (*ret* for a X86 host).

Table 4.1: QEMU ARM micro-operations

micro-operation type	Examples
Data handling	movl_T0_r0, movl_r4_T1, movl_T1_T2, movl_T2_im
Load/Store	ldl_kernel, stb_user, stlex_user
Arithmetical/Logical	addl_T1_im, subl_T1_T2, andl_T0_T1
Control flow	bx
QEMU specific	exit_tb, end, undef_insn, exception_exit

The micro-operations are hand coded C functions and a unique integer identifier is given to each micro-operation. An example of QEMU micro-operation is given in Listing 4.1, where *env* represents the simulated processor state, a structure which contains the general purpose registers, flags, co-processor state, the address translation information and QEMU specific data (*e.g.* current translation block simulated). *regs[1]* in the listing refers to the general purpose register *r1* of target ARM processor.

Listing 4.1 QEMU micro-operation example

```
1 void op_movl_T0_r1 (void)
2 {
3     T0 = env->regs[1];
4 }
```

These micro-operations are compiled by GCC with specific options to an object file. These GCC options assure that the host binary code does not contain the function prologue and epilogue. The simulator needs during simulation to be able to extract from this object file the host code corresponding to each micro-operation. Technically, a tool called *Dyngen* generates from this object file the *Tiny Code Generator* (TCG) source code [Bel05] (Figure 4.3). Listing 4.2 depicts the code sequence from TCG that performs the extraction of the host code corresponding to the micro-operation presented in Listing 4.1. The *gen_code_ptr* variable in the listing represents the memory address where the micro-operation host code is copied to.

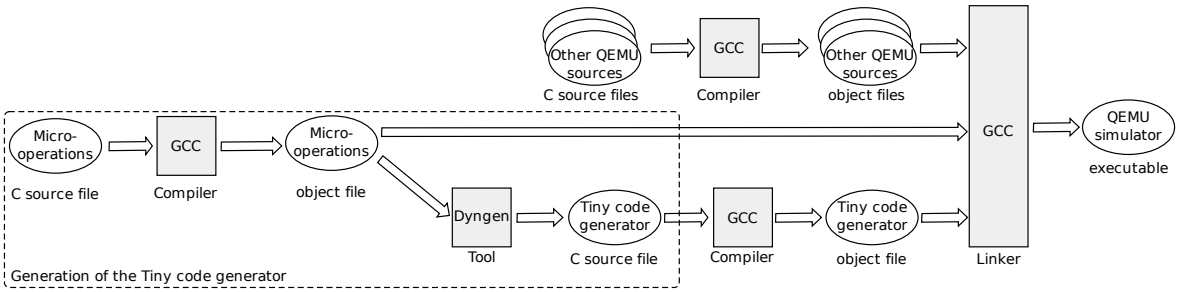


Figure 4.3: The compilation of the QEMU simulator

As the translation of the micro-operations to host processor code is done automatically by the TCG, this translation does not require any special effort.

Listing 4.2 Code sequence from generated TCG

```
1 extern void op_movl_T0_r1();
2 memcpy(gen_code_ptr, (void *)((char *)&op_movl_T0_r1+0), 3);
3 gen_code_ptr += 3;
```

The QEMU execution (Figure 4.4) is similar to the general execution scheme of the simulators using an intermediate representation. Each target instruction is translated as a sequence of micro-operations during the decode phase. The micro-operations identifiers of all instructions belonging to the currently treated translation block are concatenated into a micro-operations identifier buffer. Once the block translation is finished, the TCG generates a host function which is made of the concatenation of the micro-operations compiled code corresponding to the identifiers stored in the buffer. The resulted compiled code is stored as an entry into the translation cache for subsequent use. Figure 4.5 presents the micro-operations and the host code generated for an ARM instruction.

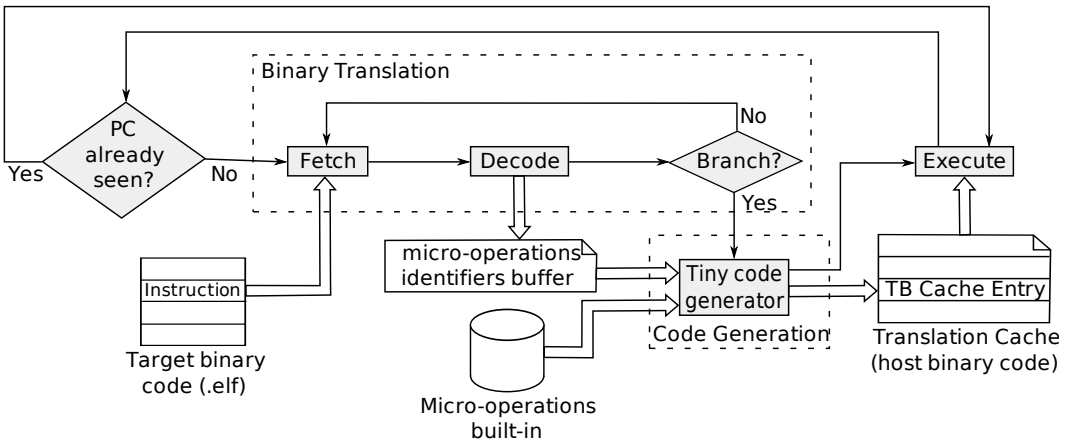


Figure 4.4: QEMU simulation model

Target code (ARM)	Description	Micro-operations generated	Description	Generated host code (x86)	
ands r2, r1, #5	r2 = r1 & 5	movl_T1_im 5	T1 = 5	mov	$0x05,%esi
		movl_T0_r1	T0 = r1	mov	0x04(%ebp),%ebx
		andl_T0_T1	T0 = T0 & T1	and	%esi,%ebx
		movl_r2_T0	r2 = T0	mov	%ebx,0x08(%ebp)
	flags = r2	logic_T0_cc	flags = T0	mov	%ebx,0xc0(%ebp)

Figure 4.5: Micro-operations and host code generated for a target instruction by QEMU

The dynamic code generation technology used by QEMU is able to offer portability across hosts at a low cost. Usually, to port an ISS from one host to another, the whole code generator has to be rewritten for the new host. However, with QEMU's tiny code generator, portability is assured by simply compiling the file containing the machine-specific pieces of code, which are the micro-operations, at the cost of some code size overhead and performance degradation. To increase the probability that the instructions are fetched and

decoded only once, QEMU uses a 16 MB translation cache.

QEMU supports self-modifying code, hardware interrupts, exceptions *etc*. To support self-modifying code, when a target processor performs a write access, the micro-operation corresponding to that write verifies if the target address is part of a MMU page for which translation blocks have been translated. If it is the case, QEMU invalidates all translation cache entries corresponding to these translation blocks. QEMU also introduces some optimizations (*e.g.* direct translation block chaining). The registers of the target processor are data members of a structure. The starting address of this structure and the three registers of the abstract machine are mapped to host processor registers. In Figure 4.5 we can see that the registers *T0* and *T1* of the abstract machine were mapped to registers *ebx* and *esi* of the x86 host processor. The starting address of the structure containing the target processor registers is mapped to *ebp* x86 register. A target register is accessed using its offset in this structure (*e.g. ebp + 4* for *r1*).

4.3 Multiprocessor modeling

We will now present our simulator approach based on the binary translation technology.

The goal of the binary translation simulators is to be very fast and functionally correct, but not to provide information about hardware execution time. Therefore, the implementation of multi-processor platforms simply call the processors interpretation function one after another in a predefined order. The return from a processor interpretation function, that we call ISS function return, happens when an interrupt or an exception occurs but only at the border of a translation block and in any case does not accumulate time. The simulation order of the processors is always the same. As our goal is to estimate timings and act on power using DVFS, we need to have a mean to control (and measure) the scheduling of the processor execution. To do so, the default behavior of the initial simulators must be modified, at the price of reducing the simulation speed.

4.3.1 ISS wrapping and connection

For each processor in the platform, we instantiate a SystemC module wrapper (*iss_wrapper*) as depicted in Figure 4.6. The execution of each processor is performed in the context of the SystemC process (*SC_THREAD*) of its wrapper. This way, the processors are simulated concurrently. It is necessary that each processor is simulated in the context of its own SystemC process, so that they are simulated independently under the control of SystemC.

In addition, the wrapper of each processor implements a DVFS (dynamic voltage and frequency scaling) module allowing each simulated processor to work at different frequencies.

In order to avoid the binary translation of the same target code for each simulated processor, the processors that may share the same translation cache are encapsulated into a SystemC module, called *iss_group*. To ensure that the host code generated for a translation block is correct for all processors in the group, the processors in a *iss_group* must be identical.

The processor wrappers are connected to an interconnect, through which they can communicate with other hardware components (traffic generator, timers, interrupt controller, memory, spinlocks *etc.*) also connected to it. All hardware components are implemented as SystemC modules. The interrupt lines of the different components (*e.g.* timers) are connected to the processor wrappers, which implement the interface between SystemC interrupt wires and the ISS.

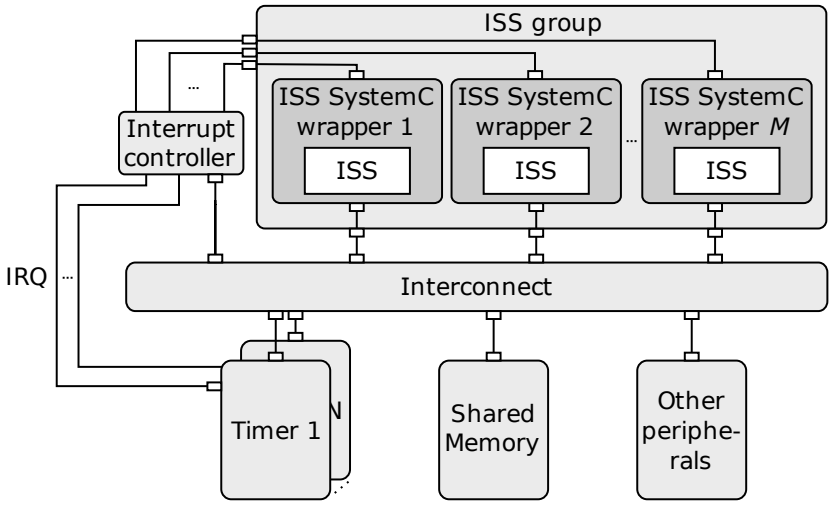


Figure 4.6: Binary translation - SystemC simulation platform

From the initial simulator, our approach uses only the processor models with, if required, their MMUs (Memory Management Unit). All other devices are externalized and implemented as SystemC modules for accuracy reasons. The main memory is also implemented as one or more SystemC modules. For accessing SystemC modules, other than the main memory, a few ranges of the physical addresses are considered as I/O addresses by the ISS. The I/O requests from the simulated processors are transformed by the processor wrappers into SystemC requests, using the protocol understood by the interconnect. This is detailed in the following subsection.

4.3.2 QEMU / SystemC implementation details for a TLM STNoC interconnect

Figure 4.7 presents our internal TLM modeling of the architectures using binary translation based ISSes. The figure is focused more on the SystemC threads and their actions.

The processor wrapper contains four threads. The processor thread is basically an infinite loop (Listing 4.3) in which the initial simulator function that simulates a target processor (line 2) is called. This function returns under several conditions. The action undertaken depends on the return reason. In the case of an interrupt or a reset, the initial simulator function is just called again for executing the interrupt or the boot routines. If the reason of return is the shutdown, the simulation stops. When a processor enters the idle state waiting for an interrupt to wake up, the wrapper of that processor waits for the reception of an interrupt (*m_ev_wakeup* at line 6).

The read and write accesses of the simulated processor occur while the processor thread executes the *cpu_exec* function. In the case of a cache miss or an uncached access, a request packet is built.

For a non-blocking write access, the processor thread places the request in the FIFO of the write buffer (❶). The write buffer is implemented by the processor wrapper. If this FIFO is full, the processor thread waits until a slot is available. The processor wrapper does not wait for the response packet. For a blocking access (all read accesses and uncached & unbuffered write accesses), the request packet is sent directly to the interconnect (❷), but only once the write buffer FIFO is empty. After the request is sent, the processor thread waits for the response packet.

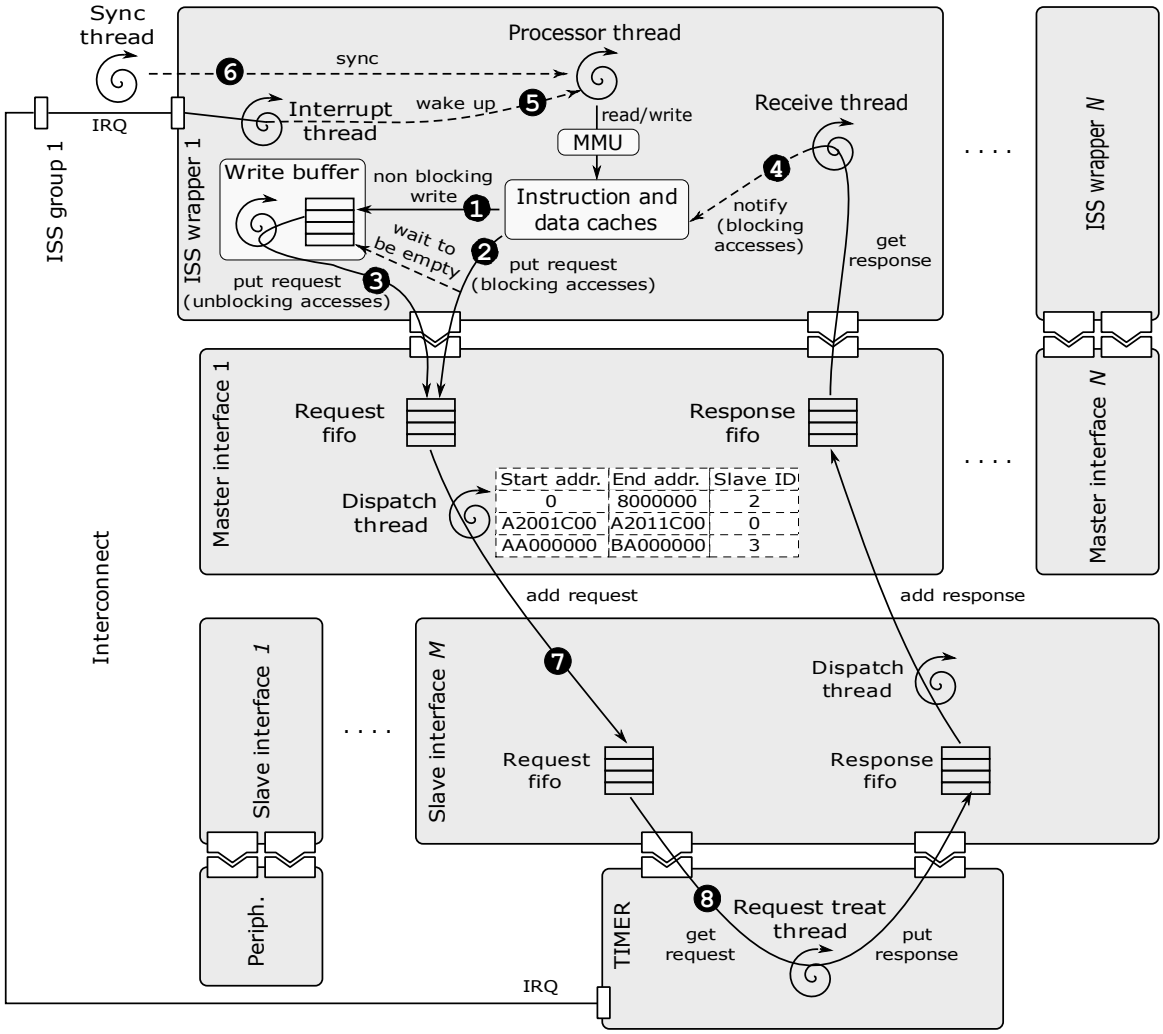


Figure 4.7: Internal structure of our TLM architecture

The second thread of the processor wrapper takes the requests from the write buffer FIFO and sends them to the interconnect (❸).

The receive thread gets the responses arriving for the processor. The response packets of the non-blocking writes are simply dismissed. For the responses corresponding to blocking accesses, the processor thread is unblocked (❹).

The fourth thread of the processor wrapper waits for the SystemC value change events of the interrupt signals. When this happens, it informs the initial simulator about the change (❺) and, if the processor is in the idle state, it wakes up the processor thread.

The ISS group contains a thread used for logs. It assures that the information of the processors (the number of cycles simulated, the time spent at each frequency *etc.*) are updated at logging time. At equal intervals of time it notifies the synchronization event of all processor wrappers in the platform and then, performs the log (❻).

Our model of interconnect contains an interface for each master and one for each slave. The master interfaces implement two SystemC interfaces: *if_put_request* and *if_get_response*. The slave interfaces implements the *if_put_response* and *if_get_request* SystemC interfaces. The master and the slave components are connected to the corresponding interconnect

Listing 4.3 CPU thread loop

```
1  while (1) {
2      hr = m_initial_simulator->cpu_exec (m_cpu_env);
3      ...
4      switch (hr) {
5          case IDLE:
6              wait (m_ev_wakeup);
7          break;
8          case SHUTDOWN:
9              sc_stop ();
10         break;
11         ...
12     }
13 }
```

interfaces using SystemC ports.

The request packet structure contains a control and a data part. The control part is composed of the operation code (some bits represent the operation type and some bits represent the number of bytes implied in the operation), a bitmap field specifying which bytes in the data part contain useful information, the destination address, the transaction identifier, the source master identifier and a field that specifies whether the packet is the last one in the transaction. All master and slave interfaces of the interconnect use the same data width. This width determines the number of bytes in the data part of the request structure. If the number of bytes required by the operation is greater than the number of bytes of the data part, the operation is split by the master components into several requests. The transaction identifier is set by the master to the desired value.

The response packet structure is similar. A few bits of the operation code are used for the error code.

The processor thread and the write buffer thread of the processor wrapper put the requests in the request FIFO of the corresponding master interface. If the FIFO is full, these threads are blocked until a slot becomes available. In the STNoC model, each master interface has a map whose entries contain the start address, the end address and the slave identifier. The dispatch SystemC thread of the master interface takes the requests from the request FIFO, finds in the map the identifier of the destination slave and subtracts the slave start address from the address field for obtaining the accessed offset in the slave address space. The requests are then added to the request FIFO of the destination slave interface (❼).

The dispatch thread of the slave interface transfers the response packets to the source master interfaces.

The requests and responses are exchanged between the master and slave interfaces directly, without passing through intermediate nodes. So, the possible congestion on these nodes is not modeled. The FIFOs of the master and slave interfaces model the access congestion. The time required for the packets transfer between the master and slave interfaces is modeled by calling the SystemC wait function inside the dispatch functions.

The timer slave component presented in the figure contains a single SystemC thread that gets the requests, treats them, constructs the response packet(s) and sends them back (❽). The time required for all these operations is modeled with a SystemC wait.

4.3.3 SystemC synchronization points

In our model, a processor is simulated while it does not communicate with the world behind its caches and the initial simulator does not stop it. When an instruction/data cache miss or an I/O occurs, the processor simulation stops and the processor wrapper synchronizes itself with the rest of the SystemC platform by consuming the time (SystemC *wait* function) required by the real processor to execute the instructions simulated since the last synchronization.

So, unlike the cycle accurate simulators where the cycles of the ISSes are simulated concurrently, the proposed binary translation based ISS simulate a group of cycles before synchronizing with SystemC and allowing other ISSes to execute. The number of cycles in these groups may vary from one to many cycles, depending on the simulated target code. The simulation of these cycles takes zero SystemC time.

The synchronization also takes place after a normal ISS function return and after a predefined period of time without synchronization. For the target processor instructions designed for the synchronization of the software running on a SMP architecture, a SystemC synchronization should also be generated (*e.g.* exclusive load and store).

The processors simulation order depends on the time consumed by the processors at synchronizations. A synchronization condition may occur at any time during the simulation of a translation block (*e.g.* cache miss), thus the unscheduling does not respect anymore the border of the translation blocks. As the initial ISS function returns only at the translation block border, we save the simulation "context" before synchronization and restore it afterwards, to reduce the impact as much as possible on the existing code.

4.3.3.1 SystemC synchronization after long intervals lacking in synchronization

Due to the fact that simulated processors do not synchronize at each memory access, if two or more simulated processors read/write from/to the same memory address, the instructions executed by these processors may differ from those executed on a cycle accurate platform. A write to an address should invalidate its corresponding cache line in all caches of the rest of processors and these processors should see the new value at their next read from that address. In our simulation strategy, if the writing processor is simulated before the reading one, the write is visible by the rest of processors before it happens in the simulated timeline. If the order of processors simulation is inverted, the reading processor does not see the effect of writing until it synchronizes and the writing processor executes the writing code.

Unscheduling the processor after a predefined time without synchronization is needed for cases when a processor waits in a loop for a simple variable to be changed by another processor or even by an interrupt handler on the same processor. This kind of loops would prevent the interrupts to occur for that processor because the interrupt pending flag is set during the SystemC synchronization. For example, for computing the processor speed, Linux waits in a loop for the *jiffies* variable to be incremented by the timer interrupt handler (Listing 4.4). The condition of unscheduling due to lack of synchronization is verified at the beginning of each translation block. The time period for this unscheduling condition determines the maximum lag of the interrupts. We arbitrarily set this parameter 250 times lower than the timer period.

Using this SystemC synchronization, blocking the simulator in an infinite loop is avoided. However, extra target instructions are simulated until this synchronization is triggered, determining a time inaccuracy of the simulator.

Listing 4.4 Example of waiting for a variable to be changed by an interrupt

```
1 void __cpuinit calibrate_delay (void) {
2     ...
3     /* wait for "start of" clock tick */
4     ticks = jiffies;
5     while (ticks == jiffies)
6         /* nothing */;
7     ...
8 }
```

4.3.3.2 SystemC synchronizations caused by target synchronization instructions

Multithread software application require the threads to synchronize. Spinlock is an example of software synchronization mechanism. The lock and unlock functions of the spinlocks are usually implemented using exclusive load and store instructions.

Figure 4.8 presents an example of software running on two processors and using a spinlock for the software synchronization. This figure shows the behavior of our simulator and what would happen if the simulator would not generate a SystemC synchronization for this type of target instructions or if the spinlock functions would not be implemented using exclusive access instructions.

Figure 4.8.a presents the execution on a real hardware. The first processor ($P1$, represented by continuous line) locks a spinlock at $t1$, at $t2$ a cache miss occurs, at $t4$ $P1$ releases the spinlock and at $t6$ it executes an I/O operation. The second processor ($P2$, represented by continuous dashed line) tries to lock the spinlock ($t3$) just before $t4$, it actually obtains the lock at $t4'$ and releases the spinlock at $t5$.

The execution on our platform in the case when the simulator would not generate SystemC synchronization for the exclusive accesses is depicted in Figure 4.8(b). Considering that $P1$ is first scheduled for simulation, it locks the spinlock at $t1$ without being unscheduled (the spinlock is placed in the main memory), but at $t2$ it is unscheduled for synchronization before loading the cache line. $P2$ is now scheduled and at $t3$ it begins to try taking the lock. $P2$ reads in an infinite loop the spinlock locked value from the main memory.

After the predefined time without synchronization, $P2$ would be however unscheduled at time $t4 + N$. Then, $P1$ is scheduled, it unlocks the spinlock and then it is unscheduled for synchronization before the I/O operation. $P1$ will be able now to take the spinlock, but it has over simulated the time N.

The simulation behavior when spinlocks functions use exclusive access functions and the SystemC synchronizations that are generated for them are presented in Figure 4.8(c). In this case, the processors synchronize before each lock and unlock. $P1$ synchronize at $t1$ and $P2$ is scheduled. $P2$ is unscheduled before its first lock attempt. $P1$ synchronizes at $t2$, but it is rescheduled because $P1$ is more advanced in simulation time ($t3 > t2$). At $t4$, before releasing the lock, $P1$ synchronizes and it is unscheduled ($t4 > t3$). Between $t3$ and $t4$ (the simulation time of $P1$) $P2$ synchronizes and it is rescheduled at each attempt to lock the spinlock. After $t4$, $P1$ is scheduled and it releases the lock and it is simulated until $t6$. $P2$ gets the lock at $t4'$ (immediately after $P1$ has released it) and release it at $t5$. This corresponds to the expected behavior.

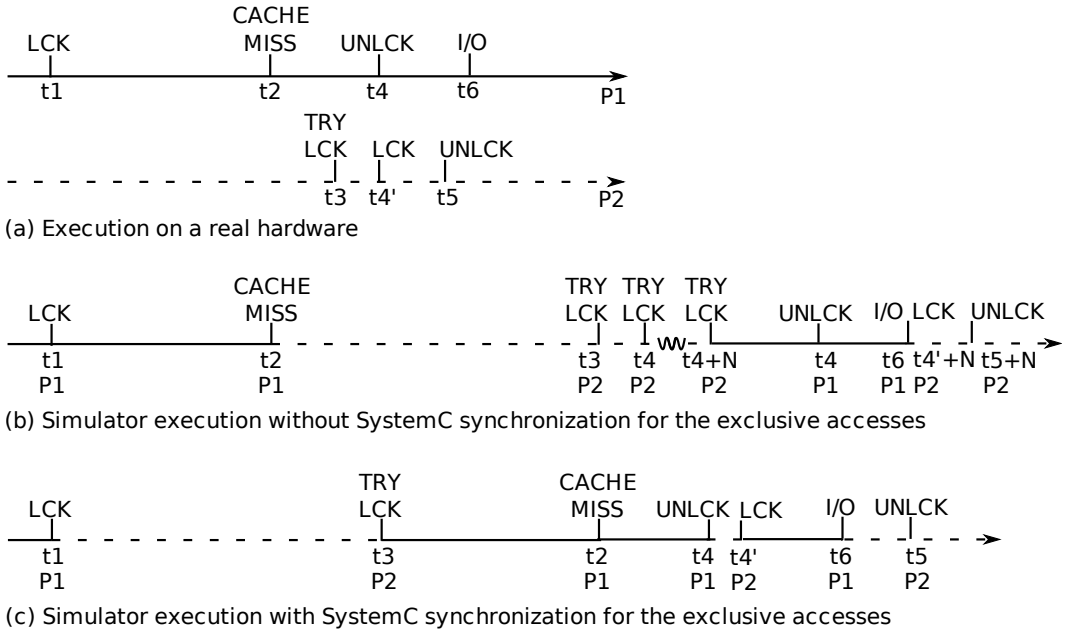


Figure 4.8: Simulation behaviors based on the spinlock implementation

4.4 Time modeling

Precise energy consumption estimations may be obtained only if the timing is accurate. The binary translation based simulators do not have a time notion for the simulated platform. For the timeout events required by the timer devices, they usually use different external mechanisms and sometimes asynchronous to the simulated platform. For instance, QEMU uses one of the host timers (hpet, rtc etc.), that generates an alarm signal on the host machine. The timeout value provided to the host timer has no correlation with the simulated time. The instructions duration and the time required by the processors to communicate with the memory and with other devices are not taken into account by the binary translation based simulators. These aspects have no importance for the goal of these simulators, which is to be as fast as possible and functionally correct.

We try to obtain an acceptable accuracy of the simulated platform and, at the same time, to keep the simulation speed at an acceptable level. Instead of using the host time, our platform uses the SystemC simulation time. The host timer is no longer used. The timer interrupts are generated by the SystemC timer modules.

For time accuracy of the processor model, a few changes have been made to the initial simulator in order to model the time required to execute instructions. These changes are performed by adding information when the target processor binary code is translated into the host processor binary code.

4.4.1 Time accuracy annotations

The first annotation tries to make simulated processors accurate from the point of view of **the time internally required for instructions execution**. These annotations are required as there is no possibility to calculate the number of target cycles simulated by a sequence of host code executed.

When the target binary code is translated, we add a micro-operation before the micro-

operations of each translated target instruction. The added micro-operation increments the number of cycles of the instructions that have been simulated since the last synchronization. The increment uses the number of cycles of the instructions given by the datasheet of the simulated processor.

Sometimes the number of cycles required by an instruction depends on information available only when the instruction is executed. As example, the number of cycles required by a multiplication instruction may depend on the values of the operands. The operands are read from memory and may differ from one execution to another. For an accurate cycle counting, the initial translation of such instruction is modified by adding a part that increases the number of simulated cycles based on the runtime information.

Another modification applied to the initial simulator for improving the time accuracy consists of presence verification and loading of the required cache line into the **instruction cache**. The verification and the loading are done in a function called by a micro-operation inserted at the beginning of each translated translation block and, inside a translation block, before the first instruction of each instruction cache line. In case of an instruction cache miss, the currently simulated processor is synchronized with the rest of the SystemC platform by consuming the time required to execute the number of cycles obtained by the first annotation. The transformation from cycles to time is performed using the current frequency of the simulated processor. After the synchronization, the processor wrapper sends a request over the interconnect for a burst transfer from memory of the implied cache line. The SystemC time required to accomplish the transfer is independent of the simulated processor frequency, but depends on the latency of the interconnect, the latency of the addressed memory module and the concurrency with other transfers.

A similar modification refers to the main memory read/write data accesses. Each time a main memory location is read, its presence in the **data cache** is verified. The mechanism described for the instruction cache miss is also applied in the case of a data cache miss. A write operation to the main memory for a write-through policy generates an update of the value in the data cache (if the written address is present in the data cache), a synchronization of the currently simulated processor and a write request over the interconnect to the addressed memory module.

The I/O operations with SystemC modules also generate a SystemC synchronization followed by a request in the SystemC subsystem.

Figure 4.9 presents a simplified example of how the list of micro-operations generated by the initial binary translation simulator is annotated from the point of view of the simulated time. The first and second columns contain the addresses and the instructions at that addresses of the target processor code. The third column illustrates the fact that each target instruction is translated by the initial simulator into a number of micro-operations. The last columns shows the annotations added to the list of micro-operations generated by the initial simulator. Considering that this target code represents the beginning of a translation block and that the length of an instruction cache line is 32 bytes, the instruction cache is verified before the first and the third instructions. The number of cycles simulated since the previous synchronization is updated for each instruction. The annotations added for a load/store instruction are visible for the second, respectively by the third, instruction.

4.4.2 Precision levels

The simulation platform can be configured to be more or less accurate. All configurations include the MMU provided by the initial simulator.

The first configuration of the simulation platform, called **"cache full" configuration**, the most accurate one, fully implements the caches and uses SystemC module(s) for the

Instr address	Target code	List of micro-operations generated by QEMU	Annotated list of micro-operations generated by QEMU
18	instr1_reg_operation	micro-op1_for_instr1 ..... micro-opN1_for_instr1	instr_cache_verify (18); nb_cycles += cpu_datasheet [instr1]; micro-op1_for_instr1 ..... micro-opN1_for_instr1
1C	instr2_load_from_1000	micro-op1_for_instr2 ..... micro-opN2_for_instr2	nb_cycles += cpu_datasheet [instr2]; data_cache_verify (1000); micro-op1_for_instr2 ..... micro-opN2_for_instr2
20	instr3_store_5_to_2000	micro-op1_for_instr3 ..... micro-opN3_for_instr3	instr_cache_verify (20); nb_cycles += cpu_datasheet [instr3]; write_access (2000, 5); micro-op1_for_instr3 ..... micro-opN3_for_instr3

Figure 4.9: Annotation example of the list of micro-operations generated by the initial binary translation simulator

main memory (Figure 4.10). The explanations of the entire chapter are focused on this configuration. The time required for executing the number of cycles corresponding to the instructions simulated is consumed using the SystemC *wait* function.

The data and the instruction caches are implemented by the wrapper of each processor. In the current implementation, the caches are placed after the MMU, so they are physically addressed. They are direct mapped and use the write-through policy. However, the principles presented here can be used with other cache strategies.

As opposed to native simulation or compiled simulation [GGP08, vM96], binary translation uses the exact address for instructions fetch and data accesses. However, the target instructions are fetched from main memory only once, for translation, before their first execution. The instructions needed for translation are searched directly in the memory module, without issuing a request over the interconnect (zero SystemC time). The loading price will be paid by the instruction cache (in function *instr_cache_verify* from Figure 4.9) when the generated code is executed from the translation cache.

An instruction or data cache miss issues a request over the interconnect for loading the cache line. As the simulator always executes the generated binary host code stored in the Translation Cache, the data part of the instruction cache is ignored. Coherence messages can be sent to the other caches.

The communication with other peripherals is performed by sending requests over the interconnect. The time consumed for these accesses is composed of the time consumed by each SystemC components implied in the transmission and the reception of the request packets.

This is the most accurate configuration of our simulation platform, its accuracy depends only on the modeling accuracy of the SystemC platform components. Also, the simulation for this configuration is the slowest, as the requests and responses pass through all SystemC components that are required for the transfer.

The second configuration of the simulation platform, called **"no cache" configuration**, does not implement caches and it uses the main memory internally allocated by the initial simulator (Figure 4.11). In this configuration it is considered that the memory is always available for all processors, without any cycle cost for accessing it. As for the previous

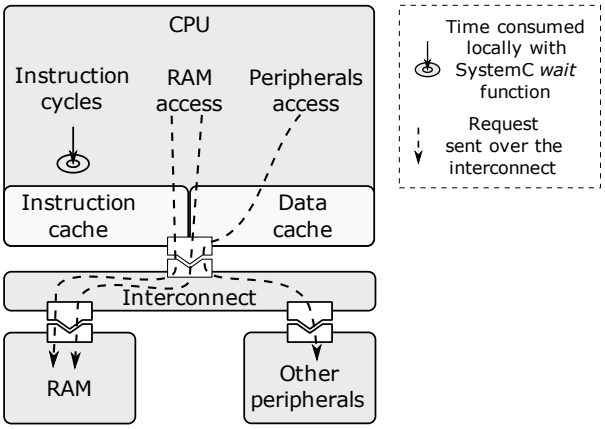


Figure 4.10: "Cache full" precision level

configuration, the I/O operations involve the interconnect and other SystemC hardware components.

In this configuration, a simulated processor synchronizes with the rest of the SystemC platform only when an I/O operation is executed and when the ISS function returns. Due to the reduced number of SystemC synchronizations, large pieces of translated code are executed without having to resort to SystemC. As a result, the simulation will be very fast (close to the initial simulator). The accuracy in this case will be low as the cache effects and the time required to communicate with the main memory over the interconnect are not counted. Since all memory accesses are done without going through the interconnect, there is no need for an explicit support for cache coherence mechanisms.

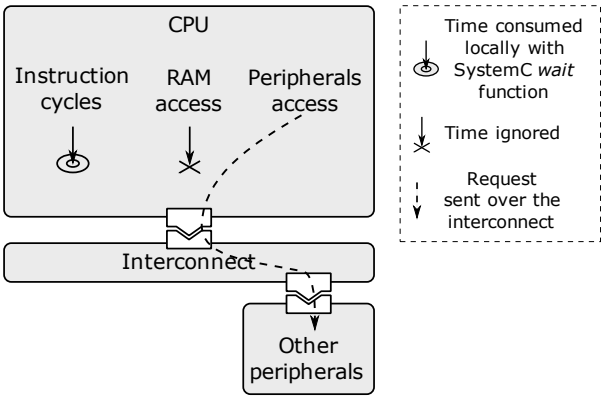


Figure 4.11: "No cache" precision level

In the third configuration of the simulation platform, called **"cache wait" configuration**, the caches are implemented only from the hit/miss point of view. This configuration uses the main memory of the initial simulation platform (Figure 4.12).

In this configuration, we model the instruction and data cache as a pure directory, such that a simple array access indicates if the instruction would in reality be in the cache. As for the first configuration, in the current implementation, the caches are placed after the MMU and are direct mapped.

A cache miss issues a SystemC wait for a time precomputed to be required to load a

cache line, without actually sending the request over interconnect. The time corresponding to the simulated cycles is consumed at the beginning of the next synchronization with SystemC. For this configuration, the processors are synchronized with SystemC when a cache miss occurs, an I/O is executed or their ISS function return.

The simulation speed for this configuration is reduced a lot compared to the previous configuration because of the large number of SystemC synchronizations produced by the cache misses. As the precomputed time is consumed directly in the cache model, a single SystemC timed event is generated for each cache miss. This is not much, considering that the transfer of a single byte over the interconnect requires more than 10 timed and untimed SystemC events. The accuracy increases by using a precomputed mean value for the time required for a memory transfer over the interconnect. However, the interconnect load, hardly guessable as it highly non linear when congested, is not taken into account.

The **"cache late" configuration** is similar to the "cache wait" configuration. Instead of consuming the precomputed time when the cache misses occur, the "cache late" configuration postpones this time consumption until the first synchronization produced by an I/O operation or by a normal return from the ISS function. At the synchronization moment, the sum of the precomputed times required by all write accesses and cache misses that have occurred since the previous synchronization is consumed. This way, the number of SystemC synchronizations is reduced, increasing thus the simulation speed.

The chances of preventing other processors from modifying a variable waited on by the current simulated processor (explained before in Figure 4.8.b) are higher in this configuration compared to the previous one because of the reduced number of synchronizations. This has an impact on the simulation accuracy.

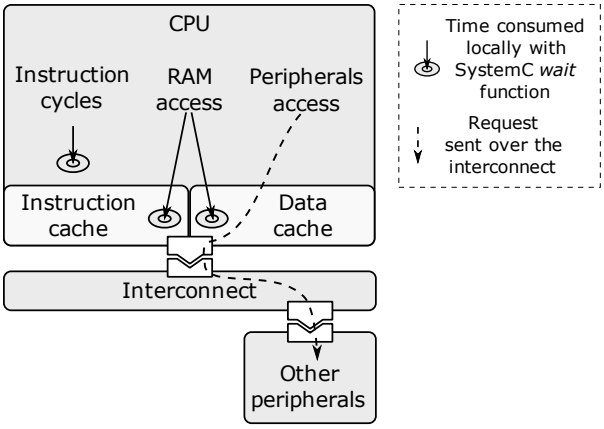


Figure 4.12: "Cache wait" and "cache late" precision levels

4.4.3 Interrupts treatment

The interrupt requests are taken into account a little bit differently in our simulation strategy as compared to the initial simulator. In the Figure 4.13 example, an interrupt request (IRQ) is produced by a device at $t2$, while the processor simulates the translation block TBx.

In the initial simulator, the IRQ represents a host operating system alarm that asynchronously interrupts the simulation process. The alarm handler sets the interrupt pending flag and, if applicable, destroys the direct block chaining to the next translation block for

assuring that the simulator will get the control at the end of the current translation block execution. After the alarm handler is finished, the simulation resumes. At the end of the translation block, the simulator will begin the treatment of the interrupt.

In our simulator case, being simulated concurrently with the processors, the hardware components generate the interrupt requests during the SystemC activities of the processors (synchronization or communication with other components). In the example, the interrupt is generated by the device while the wrapper of the processor consumes the cycles simulated by the processor in the interval between $t1$ and the first synchronization moment ($t3$ if a cache miss occurs at $t3$, $t5$ otherwise). During this SystemC waiting, at moment $t2$, the device will set the interrupt pending flag of the processor. As for the initial simulator, the direct block chaining from the current simulation block to the next one is destroyed. At the interrupt pending flag setting moment, the current simulation block is TBx (respectively, TBy). When the processor resumes from the SystemC activity, it continues the simulation. At the end of the current translation block, the processor will see the new value of the interrupt pending flag and will begin treatment of the interrupt. In the example, depending on whether a cache miss occurred or not at $t3$, this moment is $t4$ or $t6$.

So, the interrupts are treated at the end of the translation block in which the first SystemC synchronization after the interrupt generation occurs. This is not necessarily the translation block corresponding to the interrupt generation moment. The initial simulator treats the interrupt at the end of the translation block interrupted by the host alarm, but the host alarm triggering has no correlation with the simulated time.

A little delay in the interrupt treatment does not influence the execution from the functional point of view. However, the simulated time may be influenced.

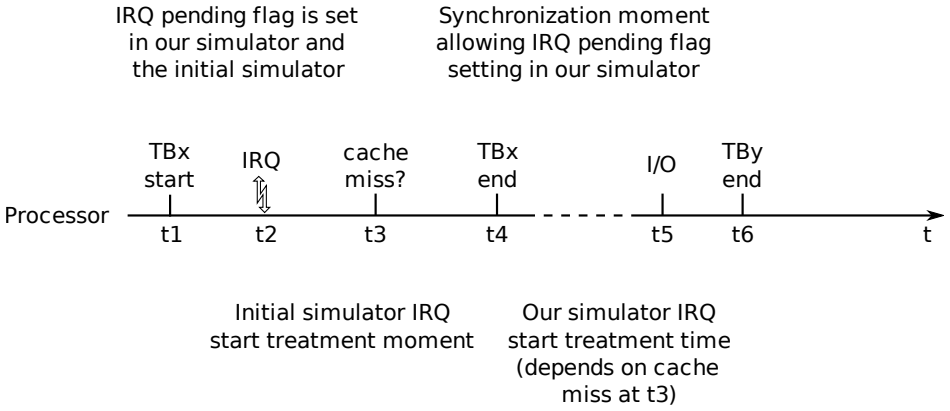


Figure 4.13: Interrupt treatment moment

4.4.4 Backdoor access to the hardware components

During the simulation, sometimes the required values should be obtained from SystemC hardware components without advancing the simulated time. For instance, although the translation from the target binary code to host binary code requires many target instructions from the SystemC main memory components, for a correct time modeling, the translation should take zero SystemC time. The commands issued by the user during the debugging of the simulated software should also not modify the SystemC time.

For supporting zero time accesses to the SystemC hardware components, our simulator implements a backdoor mechanism. This mechanism does not use the SystemC ports and

communication channels.

Listing 4.5 presents the API function provided by the backdoor mechanism to the binary translation based ISS for zero SystemC time accesses. This function transforms a target physical address into a host address. The returned host address points to the required offset inside the storage variable of the SystemC component referred by the target address. The simulated processors can directly read and write values of the desired size to the host address returned by this API.

Regarding implementation, the master interconnect interface to the calling ISS returns the identifier of the slave it sees at the specified address (line 8). It also provides the referred offset inside the slave address space. This offset is added to the address of the slave storage for obtaining the required host address (line 12 and 16). The functions used by this API do not call any SystemC function.

Listing 4.5 Backdoor for accessing the storage of the SystemC devices

```
1  long systemc_get_mem_addr (iss_wrapper *iw,
2      unsigned long target_addr, unsigned long *host_addr)
3  {
4      unsigned long   ofs_in_target, addr;
5      int             slave_id;
6
7      slave_id = noc->get_master (iw->node_id)->
8          get_slave_id_for_mem_addr(target_addr, &ofs_in_target);
9      if (slave_id == -1)
10         return ERR_MAP_ADDR_TO_SLAVE;
11
12     addr = slaves[slave_id]->get_storage ();
13     if (addr == 0)
14         return ERR_SLAVE_NO_STORAGE;
15
16     *host_addr = addr + ofs_in_target;
17     return 0;
18 }
```

We introduced in the simulator a flag variable that specifies whether the backdoor should be used when a value is required from a hardware device. For the translation of the target binary code to host binary code, this flag is set at the beginning of the translation function and reset at the end.

4.4.5 Pipeline related inaccuracy

In our platform, the instruction and data cache loading never overlaps like they do on a real platform. This is because we do not implement the pipeline of the simulated processors. However, on most of the real platforms, the instruction and data caches accesses to the interconnect are serialized because there is only one interface between the cache and the interconnect. This fact generates a behavior equivalent to that of our platform.

Although the processor pipeline is not implemented, the number of cycles required by a processor for instruction execution in simple single issue in-order RISC processors is accurate since instructions are annotated taking into account the inter-instruction effects induced by the real pipeline.

4.5 Frequency modeling

The processor frequency modeling is based on the time that would be internally spent by the real target processors for executing the instruction cycles. For synchronizing with SystemC, the number of simulated cycles is transformed into SystemC wait time using the current frequency of the simulated processor ($t = nb\_cycles/f$), *e.g.* a given number of cycles is executed by a simulated processor in a double time for a halved frequency. The current frequency of each simulated processor can be changed using its DVFS. A processor can change the frequency of another processor by programming its DVFS. If the frequency change occurs between two processor synchronization points, the number of cycles executed with the first frequency and the number of cycles executed with the second frequency should be taken into account. Figure 4.14 illustrates this situation. The first processor ($P1$) synchronizes at $t1$ and $t3$. The second processor ($P2$) halves the frequency of the first processor at $t2$. A simplified version of the synchronization function is given in Algorithm 4.1. The processor waits for the time required for the execution of *nb_cycles* cycles at the current frequency. The wait is interrupted earlier if the frequency changes during the waiting, by sending an event *ev_fq_change* event to the SystemC scheduler. If the frequency has changed during the waiting, remaining therefore unsynchronized cycles, the SystemC wait function will be called again to consume the remaining cycles. The new frequency will be used for computing the new waiting time.

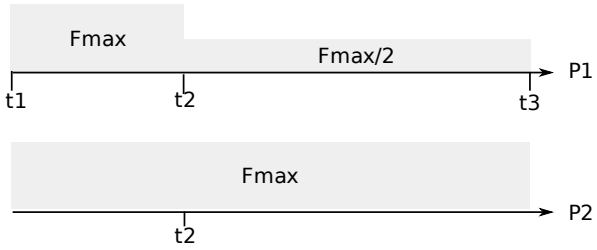


Figure 4.14: Frequency change example

Algorithm 4.1 Synchronization function

```
while (nb_cycles ≥ 1) do
  start_time ⇐ sc_time_stamp()
  wait(nb_cycles/frequency, ev_fq_change)
  time_passed ⇐ sc_time_stamp() − start_time
  nb_cycles_done ⇐ time_passed × frequency
  nb_cycles ⇐ nb_cycles − nb_cycles_done
end while
```

4.6 Energy modeling

4.6.1 Modeling

For the modeling of the consumed energy we used the approach presented in [Fou07].

The power consumed by a processor at a given moment depends on the current activity (*e.g.* instruction execution, IDLE state) and the current frequency and voltage of the processor.

The hardware measurements made in [Fou07] on a real processor show that there are many instructions which act identically from the point of view of the energy consumed per time unit. Thus, the instructions are split in several classes. The instructions of a class consume almost the same amount of energy during the same period of time, for all running modes supported by the processor. A running mode represents a voltage-frequency pair at which the processor can work. A table that specifies the power consumed by each instruction class for each running mode was filled by hardware measurements. An example of such table is presented in Table 4.2. The instruction classes are *ALU* (contains *NOP* instruction, transfers and operations between registers *etc.*), *LD/ST* (contains load and store instructions) and *IDLE*. The latter class does not contain any instruction. It models the power consumed by the processor when it is in the IDLE state (wait for interrupt). If a cache miss occurs, while the required data is searched over the interconnect, the processor consumes the power of the NOP instruction corresponding to the current running mode. The energy consumed for an instruction is $P[class][mode] \times t_{Fq} + P[ALU][mode] \times t_{cache\_miss}$, where P is the power cost presented in the table; t_{Fq} is the required time to execute the instruction at the frequency Fq; t_{cache_miss} is the sum of the time spent in the eventual cache misses. The energy consumed during an IDLE period t is $P[IDLE][mode] \times t$.

Table 4.2: Power table example (unfilled)

	0.8 V / 7 MHz	...	1.1 V / 117 MHz
ALU			
LD/ST			
COPROC			
IDLE			

Figure 4.15 presents the energy consumed by an instruction when it is executed at running mode of the maximum frequency and, respectively, at the running mode corresponding to the half frequency. In the latter case, the instruction is executed in double time. The total energy consumed in this case is smaller than the energy consumed at the full frequency.

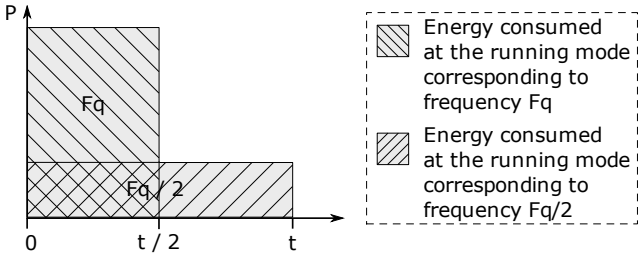


Figure 4.15: Energy consumed by an instruction at two running modes

Besides the processor, the energy consumed by the rest of the hardware components can be obtained using a similar approach. Their consumed energy is modeled as follows. Each hardware component has one or more running mode during which the consumed device power is considered to be constant. As example, the running modes of a RAMDAC (Random Access Memory Digital-to-Analog Converter) are IDLE and DISPLAY. While a hardware component works in a running mode, several events can arrive. These events increase for short periods the consumed power. The read and write requests are two

event examples for the RAMDAC device. The energy required for an event treatment is considered to be consumed all together, at the event occurrence moment. The energy consumed by a hardware component in a running mode in a period of time t, if no event occurs, is $P_{mode} \times t$. Each event adds E_{event} to the energy consumed by that component.

4.6.2 Implementation

For applying the presented methodology, an energy tracing monitor is added in our simulator. This monitor keeps track of the energy consumed by each processor independently. When the frequency of a processor changes, the monitor is informed about the new running mode. For informing the energy monitor about the time spent in an instruction class, two functions were defined. The first one works with cycles, the other one uses directly the simulated time.

A call to the cycle based function is added before the generated code of each target instruction. The parameters of this function are the instruction class of the corresponding instruction and the number of cycles simulated since the last synchronization. The energy trace monitor keeps for each call of this function a record containing these parameters. This record will be used when the consumed energy is computed. Before the synchronization of the simulated cycles, this function is also called for creating the record required for computing the energy consumed by the last simulated instruction. During the SystemC synchronization, after each piece of time is synchronized, a function of the energy trace monitor is called for computing the energy consumed by the instructions in this period. The parameters of this function are the frequency, the running mode and length of this period. For computing the energy, this function uses the records corresponding to the simulated instructions. The energy consumed by an instruction i is $E_i = P[class_i][mode] \times (cycles_{i+1} - cycles_i)/Fq$. The records corresponding to the synchronized instructions are deleted.

A function of the energy trace monitor is called at the beginning and at the end of the waiting periods for the interconnect response. The consumed energy is computed at the end of these periods. The energy consumed between two moments (A and B) of a waiting period is $E_{t_A \to t_B} = P[ALU][mode] \times (t_B - t_A)$. The same methodology is applied for the IDLE periods, for which the formula becomes $E_{t_A \to t_B} = P[IDLE][mode] \times (t_B - t_A)$. If the processor frequency is changed by another processor during these periods, the energy consumed until the change moment is computed. The new running mode will be used for the next portion of the waiting period.

Being completely modeled in SystemC, the energy consumption of the rest of the hardware components can be obtained more easily than that of the binary translation based ISS which synchronizes with SystemC only from time to time. When the running mode changes for a device, a function of the energy trace monitor is called with the occurrence moment and the new running mode parameters. The function updates the consumed energy with the energy consumed at the previous running mode ($E_{total} += P_{mode_{previous}} \times t_{mode_{previous}}$). When an event arrives, the energy required by the event treatment is added to the total energy ($E_{total} += E_{event_x}$).

4.7 Implementation details

This section presents some implementation details of the experimenting simulator.

4.7.1 QEMU isolation and encapsulation

From the initial QEMU simulation platform, we removed all peripherals, functions and variables except the functions and variables related to the processor model. We added then the modifications corresponding to the multiprocessor and time modeling. The resulted QEMU instance can not be simulated anymore by itself. We compile these QEMU instances as external dynamic libraries. At runtime, the required libraries are linked to the SystemC platform executable into a single host operating system process. The communication between the modified QEMU and the simulation platform is implemented as function calls. In this way the communication is fast and we do not modify too much the way QEMU is compiled. The initial implementation of [MPM+07] uses for the communication with the entire QEMU platform the inter processes communication (IPC) through Unix sockets, which is very time consuming.

We want to support several processors in a QEMU instance and several QEMU instances of the same/different processor type. An example of such architecture is presented in Figure 4.16. This architecture contains a QEMU instance with two ARM processors, another QEMU instance with one ARM processor, a QEMU instance with one SPARC processor and SystemC modules for interconnect, memory *etc*. The ARM processors (all three) execute a software stack cross-compiled for them and the SPARC processor executes another software stack. The two software stacks are placed in the same SystemC memory module.

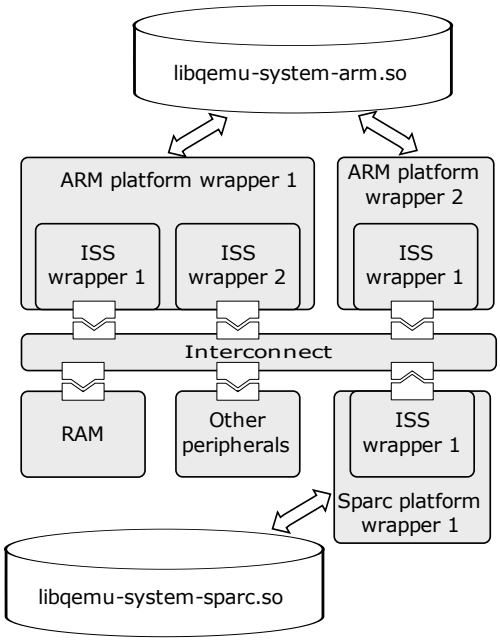


Figure 4.16: QEMU architecture containing multiple processor types

QEMU is designed to simulate platforms containing only one type of processor. For each processor type, QEMU builds an executable that is able to simulate real platforms based on that processor type. QEMU uses more than 150 global variables during simulation. It changes the simulating processors just at the translation block border, only if an exception or another special condition occurs. During the simulation of a translation block, several global variables are used. Due to these initial design properties of QEMU, we had to encapsulate a QEMU processor in an isolated way to ensure reentrancy of the code in order to correctly simulate the architecture in Figure 4.16.

The first isolation makes possible the concurrent simulation of several processor types. Being compiled as dynamic libraries, two QEMU instances of different processor types do not share the functions and global variables. The name conflict between the same function of two libraries must be avoided if that function is called from the SystemC simulation. For this, we ensure that the dynamic library of each QEMU processor type exports only an initialization function (*qemu_init*). The required QEMU dynamic libraries are loaded by the ISS group using the *dlopen* POSIX.1 function. The addresses of *qemu_init* functions are obtained with *dlsym*. The address of the interface functions between QEMU and the ISS group are exchanged using a structure passed as argument to *qemu_init*. Compiling as dynamic libraries instead of static libraries (*.a*) and the runtime loading of these libraries realize the complete isolation between the processor types.

Two QEMU instances of the same processor type use the same dynamic library, so they share the functions and global variables. The QEMU instances of the same processor type have to be isolated from each other. Also, as the processors are unscheduled during a translation block, the processors of a QEMU instance have to be isolated from each other as well. For this purpose, we made a list (using the *nm* tool) of all global variables of QEMU and divide them in four categories:

A. Global variables that are common for all instances (*e.g.* *gen_shift_T2_0*, *qemu_host_page_size*)

B. Temporary global variables specific to an instance that does not have to be encapsulated if the QEMU instances are not executed concurrently on the host (*e.g.* *code_gen_buffer*)

C. Global variables specific to an instance (*e.g.* translation cache, *io_mem_read*)

D. Global variables specific to an instance that must remain global for speed reasons (*e.g.* the abstract machine registers mapped as host registers, *env*) or other reasons.

We leave the variables from group (A) as they are. Because SystemC uses a single host thread, the variables from group (B) remain global too.

We encapsulated the variables from group (C) (around 15 variables) in a structure. An instance of this structure is created for each QEMU instance by the *qemu_init* function. This represents the second isolation.

The variables from group (D) (around 5 variables), plus a new global variable which points to the structure of the current instance, are saved and restored at each SystemC synchronization. This is the third and the last isolation that has to be done for proper reentrancy of the processors models.

4.7.2 Hardware-software co-debugging

For debugging the simulated software, we implemented a GDB server. The GDB server execution is integrated in the SystemC simulation and uses the same host thread.

When the simulator is run, a command line argument can tell it to activate the debugging capability. When this happens, the simulator initializes the architecture and, before the beginning the simulation, waits for a connection on a TCP port also specified on the command line. Using the *target remote* command we can connect from the GDB tool to our simulator. The code executed by all simulated processors of the same type is debugged using a single GDB. For the architectures containing multiple processor types, a GDB connection per processor type is permitted.

A new annotation is added to the translated code when the debug capability is activated. Before the generated code for each target instruction, a call to a function of the GDB server is inserted.

During the execution of the simulated code, this function verifies if at least a condition for stopping the SystemC simulation and giving the control to the remote GDB is met. The conditions for entering the debug mode from this function are the step by step execution and the hit of a breakpoint.

The read and write functions of the data cache model, after sending the request over the interconnect, verify if the accessed address is in the range of a GDB watch. If so, they enter the debug mode.

The debug mode is executed in the context of the simulated processor that activated it. The SystemC simulation is suspended while the simulator is in this mode. All accesses to the hardware components are done using the backdoor mechanism. This has to be done for several reasons: 1) The simulation accuracy is the first one. The user commands (*e.g.* print the value of a variable) entered in the GDB console should not advance the simulated time as they have no correlation with the actual simulation. 2) Another reason, and the most important one, is that a SystemC activity could determine the SystemC to unschedule the current simulated processor and schedule another one. The new scheduled processor could also enter the debug mode. In this case, the communication of the newly scheduled processor with GDB would interfere with the unterminated communication of the unscheduled one, what would generate an error on the GDB side. Preventing the newly scheduled processor from entering the debug mode would also prevent debugging of this processor code.

In the debug mode, the requests of the remote GDB are treated in a loop. We implemented a subset of the possible requests (continue, step, read/write register/memory, add/remove breakpoint/watch, detach *etc.*). The user commands entered in the GDB console are transformed into one or more requests for the GDB server. The debug mode is left when a continue/step/detach request is received.

This GDB server has some particularities. One of them concerns the write watchpoints. For non-blocking writes, the written value has not yet arrived in the main memory when the processor thread considers that it sent the request. It enters the debug mode and informs GDB that the watched address was modified. GDB reads the value from the main memory (through a request), considers that our server lied and issues a continue request. For this reason a command line option allows the deactivation of the write buffer, thus removing the non-blocking accesses.

The GDB server enables the debugging of the entire software stack. As it supports virtual addresses, the Linux can also be debugged from the first instruction of the boot sequence to the moment when the control is passed to the initial ramdisk.

Using a host debugger for the simulator, the hardware model and the software running on it can be simulated at the same time. Also, when the debugging capability is enabled, an unexpected value observed by the hardware model, that would normally cause the end of the simulation, can start the software debug mode. Once entered in the debug mode, the user can inspect the simulated processor registers, stack backtrace, main memory values *etc.* The time required for the bug detection is thus reduced.

The limitations of this approach are presented in Limitation 4.1.

Limitation 4.1 Binary translation based ISS strategy limitations

- The processors pipeline is not modeled
- Target code that waits in a loop for a variable to be change from outside the loop produces time inaccuracies

4.8 Experimental results

This section presents the results obtained by our TLM simulation strategy using binary translation based ISSes. The first case study presents accuracy of the different configurations of our simulation platform relatively to a given cycle accurate platform containing a classical ISS. The second case study shows the speed of the binary translation based ISS. The third case study uses our platform for the simulation of a complex application, in order to show that it scales well. In the last case study, we profit from the platform simulation speed for performing design space exploration. We have investigated different techniques for timing and power performance optimization: parallelism of the application, hardware accelerator and DVFS policy.

In all these experiments, the ISSes are based on the binary translation provided by the QEMU simulator and SystemC TLM is used as simulation environment for the whole platform.

This section begins by presenting the Motion-JPEG decoding application, which is used in the first case study of this simulation strategy and also in the experiments of the next chapters.

4.8.1 Motion-JPEG decoding application

This application is executed on a SMP SoC simulation platform. This section briefly presents the application software stack and the hardware platform on which it is executed.

4.8.1.1 Software stack

The software stack is composed of the multi-threaded Motion-JPEG decoding application running on top of a POSIX compliant lightweight multiprocessor operating system, named Mutek [PG03].

The Motion JPEG decoding application is a frame-based periodical application. Each video frame of a Motion JPEG video is separately compressed as a JPEG image. The threads of the Motion JPEG decoding application and the communication between them are presented in Figure 4.17. The traffic generator (TG) reads data from the video file, DEMUX extracts the picture information, tables and compressed data, VLD unpacks the 8x8 DCT (Discrete Cosine Transform) blocks, IQ unquantifies the 8x8 DCT blocks, ZZ reorders the 8x8 DCT blocks, IDCT inverses the 8x8 DCT blocks, LIBU reconstructs the image lines and RAMDAC shows the images on the screen.

4.8.1.2 Hardware platform

The hardware platform is modeled at the cycle accurate abstraction level using hardware components provided by the SoCLib [soc] library. This simulation platform (Figure 4.18) contains one or more processors subsystems (SS CPU), caches, DVFSes, timers, frequency

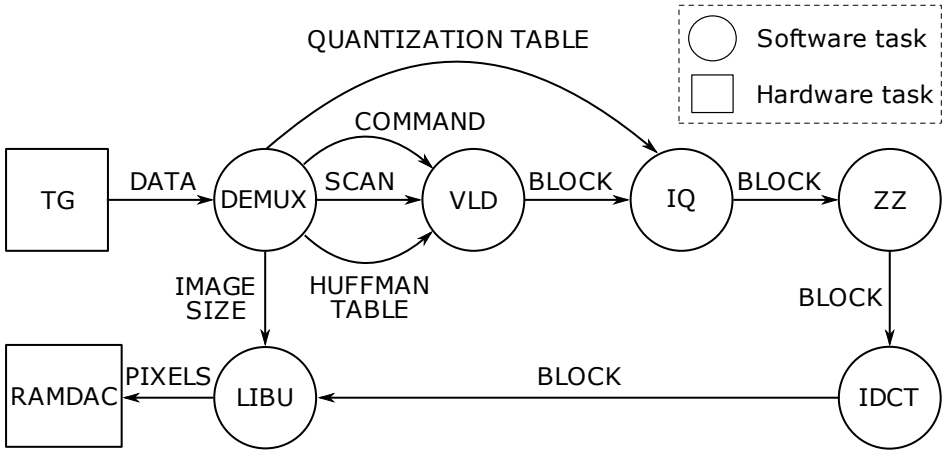


Figure 4.17: Motion JPEG task graph

adapters, different types of memory *etc*. The components communicate using a VCI (Virtual Component Interface) compliant abstract network on-chip, named GMN (Generic Micro Network). Each DVFS can be controlled individually. The frequency adapters make possible the communication between the GMN, working at a fixed frequency, and caches of the processors, working at different frequencies.

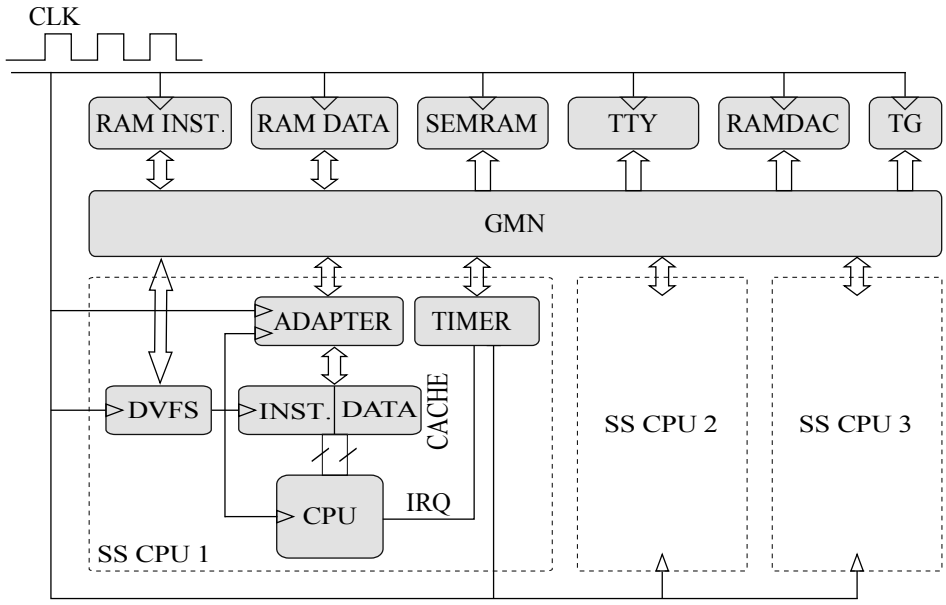


Figure 4.18: Hardware platform

4.8.2 Accuracy

This case study shows the accuracy of the proposed simulation platform using as reference the cycle accurate simulation platform presented in section 4.8.1.2. The SoCLib platform is simulated using the SystemCASS simulator. Both simulation platforms use an ARM7 processor model.

For the software part of this case study, the Motion-JPEG decoding application described in section 4.8.1.1 is used. The entire software stack is cross-compiled for the target processor. The resulting target executable is simulated by both simulation platforms, ours and the SoCLib one. In order to be able to run the same cross-compiled code on both simulation platforms, each device used by the software must be mapped in memory at the same base address for both platforms and provide the equivalent services through registers mapped at the same offset from their base address. The devices must also have the same communication protocol.

For a meaningful comparison of the two platforms, we tried to model our platform like the SoCLib platform. At the processor level, we used, for annotating each target instruction with the number of cycles required for its execution, the number of cycles internally required by the SoCLib model of the processor to execute that instruction, even if it is not exactly the number of cycles specified in the datasheet of the target processor (typically timings that depend on the dynamic value of registers are ignored in the SoCLib model). We modeled a write-through cache implementing a direct mapping scheme and having 256 lines and 8 words (32 bytes) per cache block, like the ones used by the SoCLib platform. For the rest of the hardware devices, except for the DVFSes and the frequency adapters implemented by the platform wrapper, we have created a simple TLM model of the respective SoCLib devices.

We tested the behavior of the different configurations of our simulation platform in two cases: monoprocessor and multiprocessor.

4.8.2.1 Monoprocessor experiments

In a monoprocessor system, the correctness of the processor modeling can be observed by studying the simulation results. If there is no other master in the platform (*e.g.* DMA), the interconnect is used only by the processor. As there is no congestion on the interconnect, the time required for a transfer over the interconnect is predictable.

The cross-compiled software, the Motion-JPEG decoding application together with the Mutek operating system, is executed on the two platforms from boot instant until the first frame is decoded.

The Table 4.3 depicts the results obtained by SoCLib, in the first column, and the different configurations of our simulation platform, in the next groups of columns. The next column after each presented configuration gives the error in percent of the configuration compared to the SoCLib platform. The error percent (*err*) is computed using Equation 4.1, where N_{BT} and N_{SoCLib} are the numbers obtained by our simulator, respectively, by the SoCLib simulator for the aspect for which the error percent is computed. The lines present the number of instructions executed, the number of cycles of the instruction executed, the number of instructions and data cache misses issued by the simulated code, the number uncached accesses, the number of cycles of the entire simulation, the simulation time on the host machine that ran the tests and the simulation speedup obtained by each configuration compared to the "cache full" configuration.

$$err = \frac{(N_{BT} - N_{SoCLib})}{N_{SoCLib}} \tag{4.1}$$

The first presented configuration (column *No cache* in the table) does not implement data and instruction caches. In this configuration it is considered that the memory is always available for all processors, without any cycle cost for accessing it. The communication with the hardware components other than memory is normally done, using the

interconnect. The number of cycles required for the instruction execution is modeled. Due to the unmodeled caches, an error of 37 % is obtained for the number of simulated cycles.

The second configuration (column *Cache late* in the table) models the same cache configuration than the SoCLib ones, but only from the hit/miss point of view. When a synchronization with SystemC takes place, the SystemC wait function is called for consuming the sum of the precomputed times required by all write accesses and cache misses that have occurred since the previous synchronization. The results obtained by this configuration prove that the processor is internally well modeled.

The third configuration (column *Cache wait*) is similar to the second one with the only difference that the precomputed time required for write accesses and cache misses is consumed when they occur. The simulation speed decreases because of the large number of SystemC synchronizations.

For the last presented configuration (column *Cache full*), a cache miss determines a request over the interconnect to access the data in the memory component. This way the interconnect congestion is taken into account. The simulation speed decreases even more.

Table 4.3: Monoprocessor results

	SoCLib	No cache	% err	Cache late	% err	Cache wait	% err	Cache full	% err
Instructions	24114066	24114012	-0.00	24114066	0.00	24114066	0.00	24114066	0.00
Cycles instr.	31303545	31303503	-0.00	31303569	0.00	31303569	0.00	31303569	0.00
Instr. cache misses	290428	0	-100.00	290428	0.00	290428	0.00	290428	0.00
Data cache misses	240517	0	-100.00	236717	-1.58	236717	-1.58	236717	-1.58
Uncached accesses	89820	89820	0.00	89820	0.00	89820	0.00	89820	0.00
Cycles sim. ($*10^3$)	50635	32053	-36.70	50617	-0.04	50617	-0.04	50617	-0.04
Sim. time on host (ms)	224005	405	N.A.	629	N.A.	4044	N.A.	7847	N.A.
Sim. speedup (vs. Cache full)	N.A.	19.38	N.A.	12.48	N.A.	1.94	N.A.	1	N.A.

As we can see in the table, all configurations that implement caches act identically in the monoprocessor case. This is because the time required for loading a cache line is always equal to the precomputed time used in the "Cache late" and "Cache wait" configurations, as there is no congestion on the interconnect. Thus, the "Cache late" configuration, which is about 20 times faster than the "Cache full" configuration, can be use in a monoprocessor system where the cache miss penalty can be precomputed.

4.8.2.2 Multiprocessor experiments

Since the Motion JPEG decoding application gives the best results with three processors, both simulation platforms use this number of processors.

As in the monoprocessor case, the two platforms execute the cross-compiled software from boot moment until the first frame is decoded.

In a multiprocessor system, the behavior in time is not as obvious as in the monoprocessor case. A little delay that appears in the execution of the processor of a monoprocessor system does not influence too much the execution. The same instructions are executed by the processor and the same cache misses occur. In the multiprocessor system, a little delay in the execution of a processor may generate different tasks scheduling, different cache misses and different traffic over the interconnect.

Table 4.4 depicts the results obtained by SoCLib and by our simulation platform configurations. The results in this case are not as linear as those obtained on the monoprocessor architecture. Table 4.4 has the same format as Table 4.3.

As we can see in the two tables (monoprocessor and multiprocessor), the simulation speed of the SoCLib cycle accurate simulation platform is higher for the multiprocessor

case. This is because the image is decoded in fewer cycles. The speed of our simulation platform reduces with the number of processors, since a greater number of SystemC synchronizations are generated, even if the simulated time decreases.

For the configuration without caches, although there are more instructions executed than on SoCLib, the number of simulated cycles is smaller because the cache load price is not paid. The error for the number of the instructions executed decreases when the number of SystemC synchronizations increases.

The "cache late" configuration obtains a smaller error rate than the previous configuration, but it still significant due to rare synchronizations with SystemC. However, the errors compensate each other for this case study.

The configuration that actually makes the transfers between memory and processors executes 3 % more instructions than the SoCLib platform. The use of mean access time for all types of access leads to an acceptable 4 % overall error.

Table 4.4: Multiprocessor results

	SoCLib	No cache	% err	Cache late	% err	Cache wait	% err	Cache full	% err
Instructions	25331336	34231326	35.13	30982673	22.31	26657541	5.24	25979818	2.56
Cycles instr.	32931244	44303562	34.53	40179879	22.01	34721888	5.44	33869784	2.85
Instr. cache misses	238916	0	-100.00	235518	-1.42	244415	2.30	239513	0.25
Data cache misses	261273	0	-100.00	246550	-5.64	255281	-2.29	255289	-2.29
Uncached accesses	169614	88963	-44.54	152989	-9.80	166210	-2.01	183522	8.19
Cycles sim. ($*10^3$)	19020	15013	-21.07	19275	1.34	17415	-8.44	18369	-3.42
Sim. time on host (ms)	176408	463	N.A.	716	N.A.	4904	N.A.	9932	N.A.
Sim. speedup (vs. Cache full)	N.A.	21.45	N.A.	13.87	N.A.	2.03	N.A.	1	N.A.

4.8.3 Simulation speed

In order to determine the simulation speed of the proposed binary based ISS, we constructed a new simulation platform. This new simulation platform is based on the interpretative SoCLib ARM ISS. The rest of the hardware components of this new platform are taken from our TLM platform (caches, interconnect, memory *etc.*). A specific wrapper of the SoCLib ISS was implemented.

The software stack consists of the Motion-JPEG decoding application running on top of the Mutek operating system.

The simulation results show that the "Cache full" configuration of our simulation platform is about 2 times faster than the platform using the interpretative ISS for an identical simulated time.

4.8.4 Running a complex software

In this case study, we try to benefit from the simulation speed of the proposed simulation platform by running on it a complete legacy operating system and a complex application.

Regarding the software stack (Figure 4.19), we use a *Linux* operating system (version 2.6.26). We have created a new Linux configuration for our simulation architecture and implemented the device drivers for the SystemC models of the peripherals (*e.g.* RAMDAC, TTY *etc*). The Linux OS is cross-compiled for the target architecture. On top of Linux, we execute a multithreaded H264 decoding application using the POSIX Thread API. This application (provided by Thales in the context of the LoMoSA Medea+ project) has four threads for decoding the images, each thread decoding its own slice of the image.

The configuration "no caches" of the simulation platform was used in this case study.

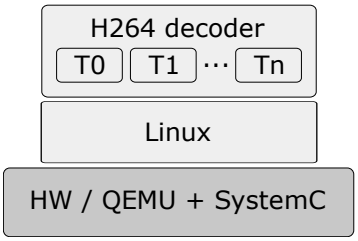


Figure 4.19: Software architecture

Booting Linux takes around 31 seconds on the host machine on which the architecture model was simulated. The simulation of an image decoding takes 2 seconds on the host machine. On the cycle accurate platform, Linux booting would take around 3.3 hours and the decoding of a frame 12.7 minutes.

4.8.5 Design space exploration

The application used in this case study is the one used in the previous case study. As different characteristics of the H264 decoding application will be varied for the design space exploration, we begin by a brief presentation of this application.

4.8.5.1 H264 decoding application

The decoding algorithm is divided into the following steps. First, the decoder reads the input stream and extracts packets. In this basic implementation, a packet corresponds to a slice. Once the packets have been extracted, each of them is sent to a task that handles part of the decoding process.

The block called entropic decoder first decodes the slice header, which contains decoding parameters. The process then iterates on all the macroblocks of the slice. For each macroblock, the process first decodes prediction information and residuals. Prediction parameters are then sent to the intra and inter prediction blocks that reconstruct the predicted macroblock. The residuals are sent to the inverse transform block that reconstructs original residuals. Residuals and the predicted macroblock are then added in order to reconstruct the original macroblock: this process is called rendering. The reconstructed macroblock is then sent to the deblocking filter process. The software architecture of the application is depicted in Figure 4.20, where each *Ti* is a task.

4.8.5.2 Influence of parallelism on H264 performances

The objective of this experimentation is to study the influence of the level of parallelism of the H264 decoder software.

We investigate in the same experiment set the influence of image size and frame rate on the video quality in order to define the different degraded modes of the system. The studied configuration are CIF (Common Intermediate Format) and QCIF (Quarter CIF) image sizes (*i.e.* 352x288 pixels for CIF and 176x144 pixels for QCIF) and 10 and 25 frame per second as frame rates.

Configurations were studied under four different levels of parallelism (from 1 to 4) adapted to the hardware platform architecture. It was not possible to experiment QCIF parallelism 2 and 4 because we did not have an encoding software producing it.

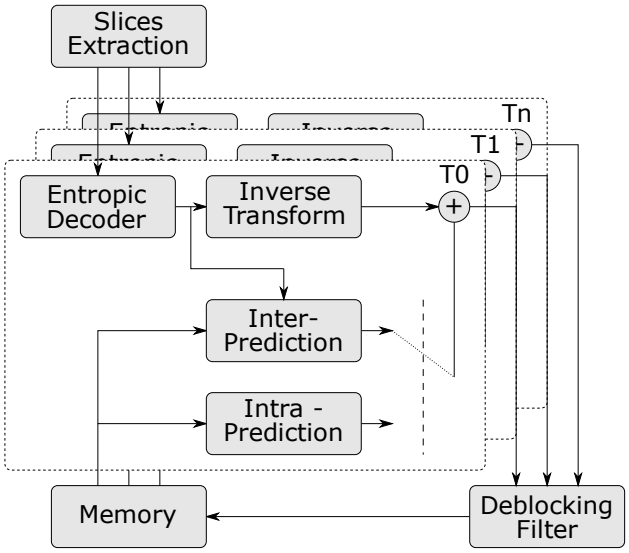


Figure 4.20: Software architecture of the H264 decoder

All experiments results are reported in Table 4.5, which shows average processors usage and average energy consumption for the different video quality configurations against the different parallelism levels.

Table 4.5: Performances of the H264 decoder against number of threads.

	1 thread		2 threads		3 threads		4 threads	
	Util.	Power	Util.	Power	Util.	Power	Util.	Power
cif@25	441.46	697	229.03	910	168.88	1060	121.81	1261
cif@10	176.59	697	91.56	841	64.53	832	63.90	821
qcif@25	108.43	686	x	x	69.48	658	x	x
qcif@10	43.36	555	x	x	28.12	549	x	x

These results are mean utilization rate and mean power consumption (in mW) for each configuration. The mean utilization rate is obtained by averaging over 50 frames the ratio between the time needed to decode the frame and its deadline. For example, for a 10 frame per second video, a 50% utilization means that the average time to decode an image is 50 ms. The mean power represents the average power consumed by the entire platform.

As it can be seen from the table, the platform is not able to keep the pace of a 25 CIF frames per second video, which is demonstrated by a utilization rate over 100%.

These results however validate the fact that parallelizing this decoder is a valuable operation, since when the number of threads grows, the mean utilization and mean power decrease. These decreases are due to the fact that frames are decoded faster, so processors go to idle for a longer time.

4.8.5.3 DBF function

[HJKH03] indicates that the DBF function is one of the most computation hungry of the H264 decoder. Due to the fact that it is hardly efficiently parallelized, we will take a closer look to the behavior of this function.

Although it is very valuable from the image quality point of view, the DBF consumes a great part of the decoding process. For example, the mono-threaded version requires 160 ms for decoding a CIF frame if the DBF algorithm is used, whereas only 70 ms are required if it is not used. This means that the DBF algorithm represents about 55% of the results presented in 4.8.5.2.

In order to limit the processor workload of the H264 decoding software, we investigate the solution of using the deblocking filter as a hardware accelerator accessible through the interconnect. We used the deblocking filter SystemC model provided by Thales Group [tha]. This hardware component has both initiator and target interfaces. Both interfaces are connected to the interconnect, as presented on Figure 4.21. The filter can be configured to inform the software when the process is finished by sending interrupts.

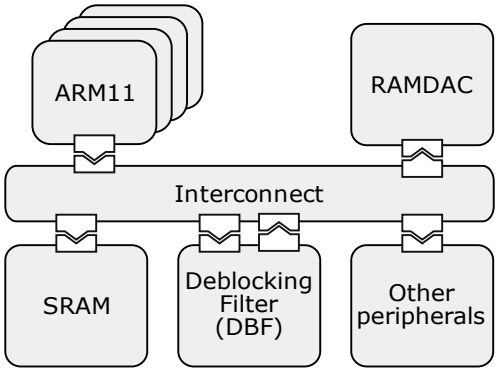


Figure 4.21: Hardware architecture with hardware deblocking filter

As a consequence, the hardware deblocking filter reduces decoding time and reduces the platform power consumption. It is well known that dedicated hardware is much more power efficient than software. The hardware implementation can process up to four slices in parallel in CIF and QCIF images.

We reproduced the experiments of section 4.8.5.2 but with the version of the decoder virtual platform which now integrates the hardware deblocking filter. On the software side, a device driver for the filter was added in the Linux kernel. As far as timings and energy consumption are concerned, calibrations were made thanks to fine grained approximation based on the internal processing of the component and the gate level synthesis results.

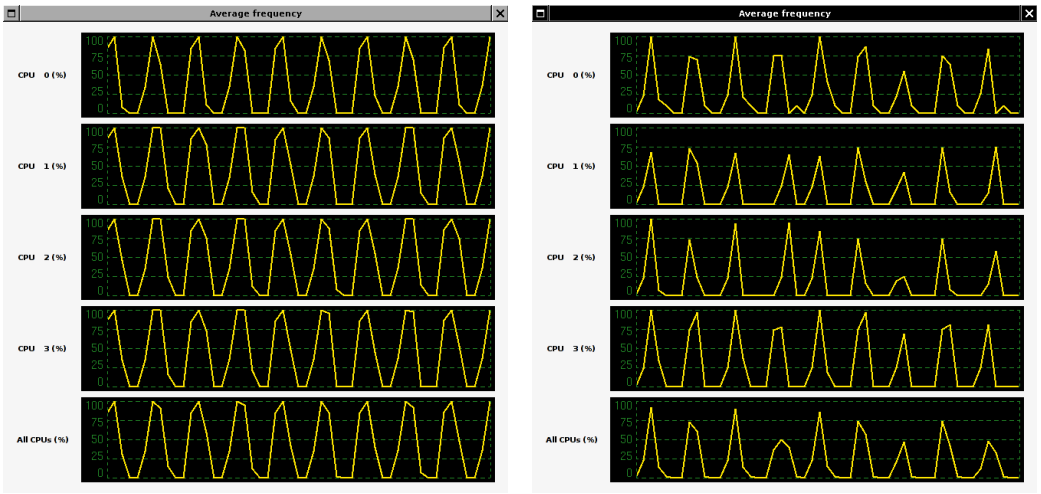
The results of this new run are given in Table 4.6. It clearly shows that the use of this hardware accelerator allows the optimization of the system usage and of the energy consumption since the processors are more often in idle mode. For example, for a 4 threads version of the decoder with a 10 CIF frames per second video, the utilization rate drops from 64% to 48% (a 25% reduction) and mean power decreases from 821 mW to 740 mW (a 10% reduction).

Table 4.6: Performances of the H264 decoder with hardware DBF.

	1 thread		2 threads		3 threads		4 threads	
	Util.	Power	Util.	Power	Util.	Power	Util.	Power
cif@25	285.18	693	182.09	844	141.32	980	163.07	927
cif@10	114.08	680	72.77	716	57.68	727	48.62	740
qcif@25	98.46	637	x	x	98.56	702	x	x
qcif@10	39.39	529	x	x	39.42	555	x	x

4.8.5.4 DVFS capabilities

As we can see in Figure 4.22, implementing the deblocking filter in hardware frees some processor time. The 4 threads version depicted in this figure is run in a software DBF flavor (4.22(a)) and in a hardware DBF flavor (4.22(b)). The workload difference can be mainly observed on the fifth curve representing the overall processors usage.



(a) Average CPU frequency with software deblocking filter

(b) Average CPU frequency with hardware deblocking filter

Figure 4.22: Benefit of hardware deblocking filter on a 4 threaded version

The goal is now to take advantage of the DVFS capabilities of the simulation platform by using an energy saving algorithm. The DVFS algorithm aims at reducing processors lazy times by spreading the application tasks workload over the complete period (given by the frame rate). This allows to lower the voltage since maximum frequency is lower, allowing to gain energy. In this work, the frequency is chosen by the application according to the previous deadlines (frames) and the previous decoding times. Technically, Linux returns the execution times to the application and sets its frequency.

We applied the same experiment setup to the DVFS aware version of the system. The results are depicted in Table 4.7. As we can see, reducing frequency on lazy times increases the average processors usage, but effectively reduces the energy consumption. The 10 CIF frames per second video running on the 4 threads version decoder now generates a 73% utilization rate with a 509 mW mean power. Compared to the 64% (respectively 49%) utilization rate and 821 mW (respectively 740 mW) mean power consumption of the original software version (respectively the hardware DBF version), this DVFS policy achieves a power saving of 40% (respectively 31%).

Table 4.7: Performances of the H264 decoder with DVFS policy

	1 thread		2 threads		3 threads		4 threads	
	Util.	Power	Util.	Power	Util.	Power	Util.	Power
cif@25	285.19	693	185.30	838	145.93	966	127.72	1035
cif@10	114.08	681	80.22	628	73.17	507	72.62	509
qcif@25	98.46	637	x	x	98.54	703	x	x
qcif@10	64.39	316	x	x	59.17	334	x	x

4.9 Conclusion

We presented in this chapter a simulation strategy implemented at the TLM abstraction level. This simulation strategy provides the possibility of runtime changing of the simulated processor frequency. It is based on existing untimed binary translation ISSes. We presented the modifications required to be applied to such ISS for transforming it into a time accurate hardware component that can be used in an event driven simulator.

We have shown that the proposed approach allows to trade simulation speed for accuracy, and that even with the more detailed model, the simulation speed is still good.

Chapter 5

Static scheduling simulator accepting multiple and dynamically changing frequencies

Contents

This chapter presents our proposed simulation strategies that adapt a static scheduling event driven simulator for being able to simulate platforms that contain components working at different frequencies.

The dynamic scheduling simulators, usually used for the CA simulations, offer a great flexibility in hardware modeling. Due to the reduced simulation speed of these simulators, static scheduling versions of the simulators have been constructed. For assuring the static schedulability, they have to impose several constrains. Thanks to these constrains, the list of processes to be called do not have to be recomputed at each simulation moment. The static scheduling is created before the start of the simulation and does not change during the simulation.

For a static scheduling, all processes of the hardware components usually have to be sensitive to the edges of a unique system clock. Due to this constrain, the modeling of the hardware components requiring runtime change of their working frequencies is impossible.

This section presents the modifications required to be brought to a static scheduling simulator for being able to simulate platforms with hardware components whose frequencies can change at runtime.

For understanding these modifications, we will first present some details of the dynamic and static scheduling simulators. During this chapter, SystemC is chosen as example of dynamic scheduling event driven simulator and SystemCass as example of static simulator.

5.1 How a dynamic scheduling event driven simulator works?

We will present two aspects of the event driven simulators, the dynamic and the structural ones. The dynamic aspect refers to the manner in which simulation of the architecture is conducted. The structural aspect consists in how hardware components models are instantiated and connected.

5.1.1 Simulation

Although an event driven simulator is generally executed in a single thread of the host operating system, it must manage to simulate the concurrency of the processes.

For this, the dynamic scheduling event driven simulators keep a list of the future timed events of the simulated platform. The events list is ordered by the occurrence required time of the events. The simulation goes as follows. The simulator takes the first timed events with the same occurrence time from the list and advances the simulation time to this value. The processes containing one of these events in their sensitivity lists and the processes blocked for one of these events are added to the runnable processes list.

Listing 5.1 presents the actions performed for the current simulation time. The processes from the runnable processes list are triggered in an indeterministic order (line 9). During their execution, the processes may issue a timed waiting, adding thus a new timed event in the ordered list. If a process writes a value to a signal or to a register, the new value will be visible only after all unblocked processes are executed (current *delta cycle*, lines 4-17). In fact, when a different value from the current one is written to a signal or to a register, that signal/register is added to a list of signals/registers to be updated.

After all unblocked processes have been executed, the signals and the registers from this list are updated with the new values (line 12). The change of a signal value usually generates one or more events (*e.g.* a value changed event, a positive/negative edge event in the case of two states signals). These events are added to a delta events list.

The processes sensitive or waiting on the events in the delta events list are then added to the runnable processes list (line 16). As a result, a process unblocked by an untimed event is executed in the next delta cycle to the process which determined that event. If this list is not empty, a new delta cycle begins.

The simulator advances to the next timed event when there are no more unblocked processes for a new delta cycle of the current simulation time.

An event already present in the timed events list can be rescheduled for occurrence by the architecture model at another simulated time. In this case, the event will be triggered by the simulator only once, at the smallest simulated time of the two.

5.1.2 Architecture construction

During the construction of the simulation architecture, the modules and the communication channels of the architecture are instantiated. Also, the ports of the modules are connected

Listing 5.1 The simulation of a time instance by a dynamic scheduling event driven simulator

```
1  void simulate_a_time_instance ()
2  {
3      while (runnable_processes->is_empty () == false)
4      {
5          delta_cycles++;
6
7          // evaluate phase
8          while ((process = runnable_processes->pop ()) != NULL)
9              process->execute ();
10
11         // update phase
12         update_signals ();
13
14         // process delta notifications
15         while ((event = delta_events->pop ()) != NULL)
16             event->trigger ();
17     }
18 }
```

to the corresponding channels.

In SystemC, the architecture construction (called *elaboration*) and the architecture simulation represent two well delimited phases. When the simulation starts, all components of the architecture must be instantiated and all their port must be connected. No instantiation and no connection are possible after the simulation has begun.

As its name says, an event driven simulator is based on events. When an event is triggered, the processes from its list are executed. This section presents how these lists are constructed in SystemC.

The SystemC example presented in Figure 5.1 is focused on the connection of two modules through a channel. The two modules may have other processes and other ports connected through other channels to other modules for forming the final architecture. The process of the first module writes to the output port. The process of the second module is sensitive to an event of the input port.

In SystemC, each port contains a pointer to the interface it requires. When a port is connected to a channel, its pointer to the interface is initialized with the pointer to the interface implementation in the channel. When a process writes or read to/from a port, the corresponding implemented interface method of the channel is called. The channel contains the current and the next values of the signal and also the events on which the processes of the modules connected to the channel may be sensitive. The *update* method of the channel called in the update phase of the simulation changes the next value and notifies the events whose notification condition is met.

The Listing 5.2 presents the C++ code of the example in Figure 5.1. The second module is instantiated at line 46 and its process is declared as SystemC method at line 32. This process requires to be executed when the custom event of the input port is triggered (line 33). As the input port is not yet connected to a channel, its pointer to the interface is *NULL* and its sensitivity event can not be obtained. As result, at the module instantiation time, the processes can not be added to the list of processes of the corresponding events. For this reason, each port maintains a binding list. An item of this list contains the handle of the process and the pointer to the interface function whose implementation should return the

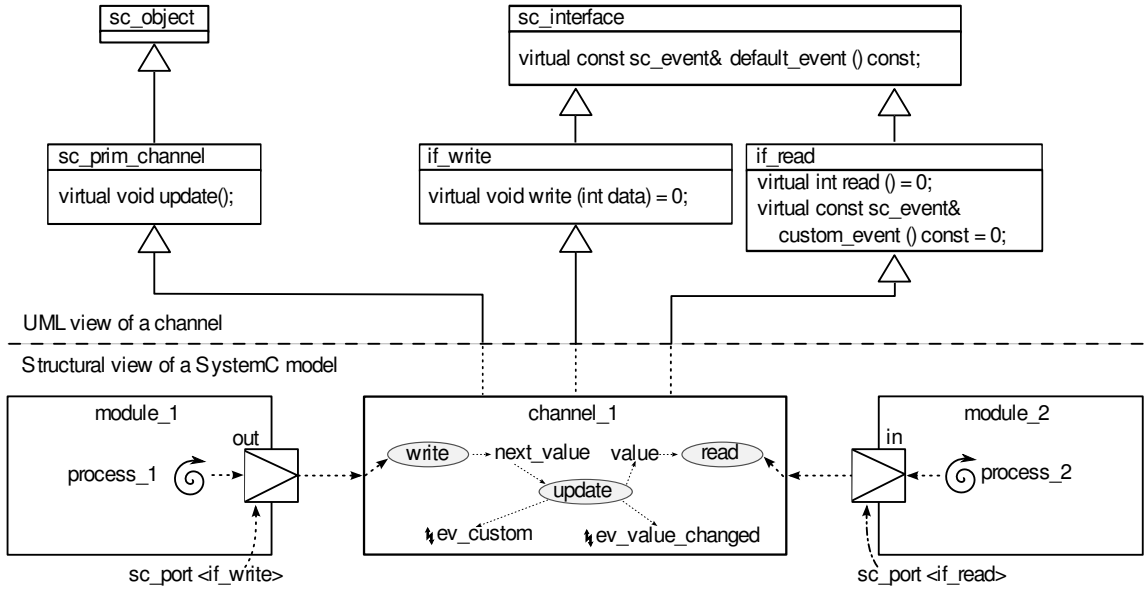


Figure 5.1: SystemC example with a custom channel

event on which process is sensitive.

Later, when the channel is binded to the port of the module (line 47), the pointer to the interface implementation of the channel is copied to the pointer to the interface of the port.

Before the start of the simulation (line 49), the pointer to the interface of each port and the pointer to the function of each item in the corresponding binding list are used for obtaining the required event. The process can be now added to the list of processes of the event.

Beside the static sensitivities of the processes on the events, some processes (called *threads*) may issue during their execution a wait for an event to be triggered. The events keep a separate list of processes for the processes dynamically blocked on them. When an event is triggered, the processes from this list are unblocked and they are removed from it.

The elaboration phase presented here is simplified as compared to real one, but it is sufficient for understanding the rest.

A dynamic scheduling event driven simulator can simulate a large number of hardware architectures. The architecture may have multiple clocks and the processes can be statically sensitive or they can dynamically wait for events provided by the interfaces of the ports, local events, timed events, etc. In fact, the clock is implemented as a simple channel that provides three events: one for the positive edge of the clock, another for the negative edge and the last for the both edges. The clock signal is not connected to any output port. The value of the clock signal is changed by internal processes of the clock. These processes triggered by timed events.

5.2 How a static scheduling simulator works?

A static scheduling simulator constructs several lists of processes before the beginning of the simulation. The simulation consists in the execution of these lists as a pattern which repeats in time, each list being attached to a time offset in this pattern. The execution

Listing 5.2 SystemC code example with a custom channel

```
1  ...
2  class channel_1 : public sc_prim_channel,
3      public if_read, public if_write
4  {
5  public:
6      ...
7      virtual const sc_event& custom_event() const {
8          return ev_custom;
9      }
10 private:
11     sc_event            ev_value_changed, ev_custom;
12     int                 value, next_value;
13 };
14 class module_1 : public sc_module
15 {
16 public:
17     module_1 (sc_module_name name) : sc_module (name) {
18         SC_METHOD (process_1);
19         ...
20     }
21     void process_1 () {
22         out->write (...);
23     }
24 public:
25     sc_port <if_write>  out;
26     ...
27 };
28 class module_2 : public sc_module
29 {
30 public:
31     module_2 (sc_module_name name) : sc_module (name) {
32         SC_METHOD (process_2);
33         sensitive << *new sc_event_finder_t <if_read>
34             (in, &if_read::custom_event);
35         sensitive << ...;
36         ...
37     }
38 public:
39     sc_port <if_read>   in;
40     ...
41 };
42 int main(int, char**) {
43     channel_1           sgn;
44     module_1            om1 ("om1");
45     om1.out (sgn);
46     module_2            om2 ("om2");
47     om2.in (sgn);
48     ...
49     sc_start (...);
50     return 0;
51 }
```

order of the processes in each list is always the same. Figure 5.2 presents an example of static scheduling with two simulation points in the pattern, their lists containing two (respectively, three) processes.

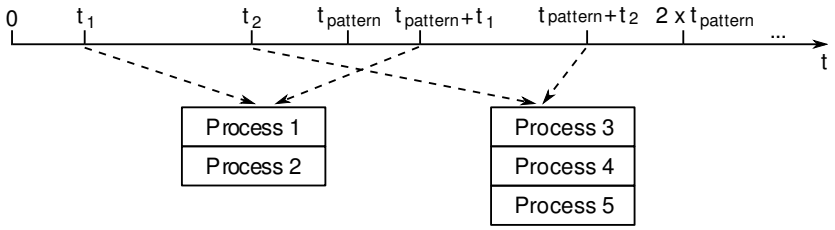


Figure 5.2: Static scheduling example

A static scheduling simulator must assure that the simulation of the architectures which can be modeled with its hardware description language can be statically scheduled and that the lists of processes can be easily built.

5.2.1 Finite state machine (FSM) based simulators

Hardware architecture can be modeled using a set of synchronous finite state machines. The state machines communicate between them through signals. Each hardware component is modeled with one or more FSMs connected to the input and the output ports of the component. This solution has been proposed in [Buc07] and refined in SystemCass [Buc06].

There are two types of FSMs: Moore and Mealy. The outputs of a Moore finite state machine depend only on the current state. The outputs of a Mealy finite state machine depend on the current state and the inputs (Figure 5.3(a) and 5.3(b)). It is possible to combine the two types of FSMs like in Figure 5.3(c). In this case, some outputs are set by the Moore output logic (called *Moore function*) and the rest by the Mealy output logic (called *Mealy function*).

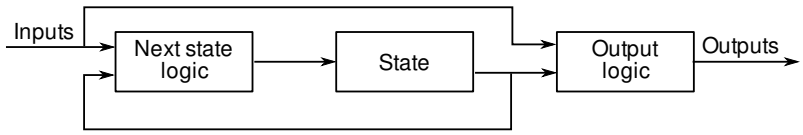
The FSMs of the architecture model may all be synchronous on a unique clock. In this case, the logics which compute the next states (called *Transition function*) may be evaluated only on an edge of the clock (*e.g.* the positive clock edge event), when the inputs are considered to be stabilized. The Moore functions will be evaluated only on the other edge of the clock (*e.g.* the negative clock edge event). The Mealy functions will be evaluated on the same clock edge as Moore functions and also each time one of its inputs changes.

As the Transition functions of the FSMs do not modify the outputs, only the Moore and Mealy functions can modify the output. Their writing to the outputs will generate a new delta cycle if there is at least one Mealy function dependent on an input connected to an output which has been modified. All these delta cycles take place at the simulation time of the negative clock edge.

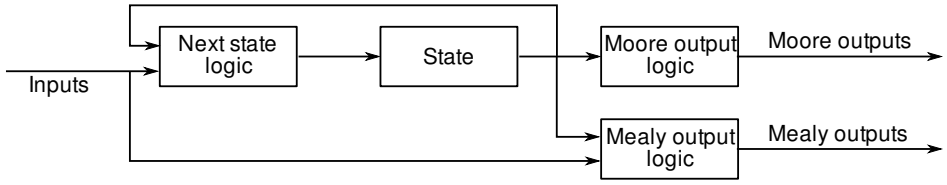
The $t_{pattern}$ in the Figure 5.2 is the period of the clock. The simulation points of the pattern are represented by the positive and negative edges of the clock. The list of processes of the positive edge is composed of the Transition functions.

For the second simulation point of the pattern, as the Moore functions are executed only once in a cycle, it is preferable to construct a separate list of processes with these functions. As the Mealy functions are also sensitive to inputs, a dynamic scheduling would be normally required for computing at each delta cycle which processes should be executed. The dynamic scheduling is complex and time consuming, if we consider that the combinational

(a) Moore



(b) Mealy



(c) Moore-Mealy

Figure 5.3: FSMs logic

functions are usually very small. In order to avoid the costs of a dynamic scheduling, all Mealy functions can be executed at each delta cycle.

The function from Listing 5.3 is an example of clock cycle simulation for a static scheduling simulator. The Transition functions are executed by the function called at line 3. The state registers modified by the Transition functions are updated with their new value at line 4. After the simulated time advances to the negative edge of the clock (line 6), the Moore functions are executed.

The Mealy functions are executed (lines 10 - 14) while a new delta cycle is required. When the loop stops, the Mealy functions are stabilized.

The signals in the dynamic scheduling contain several events. When the value of a signal changes, the events which meet the changing value condition are notified. The processes depending on these notified events are scheduled for execution.

In static scheduling, the entire scheduling is constructed before the beginning of the simulation, so the events do not require to be notified during the simulation. Due to this fact, no event is notified when the value of a signal changes. This way, the time required by dynamic scheduling is saved.

Since no event must be notified when the value of a signal changes, since Mealy functions stabilize correctly even if they miss intermediate combinations of input values during their stabilization [HP98] and since the outputs seen by the Transition functions at the beginning of the next cycle are the stabilized outputs, the implementation of the signal may contain only the current value, not the new one too. When a value is written to an output port, the written value becomes directly the current value of the signal connected to that port. The signal is not added to a list of signals to be updated like in the dynamic scheduling case. Instead, a flag that informs that a new delta cycle is required is set. This increases more the simulation speed.

The approach of maintaining only the current value can not be applied to the state registers. The Transition function which modifies the register or another Transition function of same module would read the modified value of the register, what could determine transitions to wrong states.

The Moore functions must be executed before the stabilizing loop for the Mealy functions. Otherwise, the writings of the Moore functions would destabilize again the Mealy functions.

Listing 5.3 The simulation of a clock cycle by a static scheduling simulator

```
1  void simulate_a_cycle (void)
2  {
3      transition_processes ();
4      update_registers ();
5
6      simulated_time += clock_pos_time;
7
8      moore_processes ();
9
10     do
11     {
12         new_delta_cycle = false;
13         mealy_processes ();
14     } while (new_delta_cycle);
15
16     simulated_time += clock_neg_time;
17 }
```

Table 5.1 summarizes when and where the read and write actions on signals must occur in the FSM model to ensure correct simulation with a static scheduler.

If a Transition function writes to an output port (❶), as the value of the attached signal is directly modified, the next executed Transition functions completely miss the initial value of the signal and see directly the new value. This could determine transactions to wrong states.

A Moore function should set its output based only on the current state. Using the values of the input ports (❷) for setting the outputs is not legal because those values may be modified by other Moore functions previously executed. This could lead to wrong outputs. So, an input value required for setting the outputs should be buffered in the Transition function and this buffered value should be used in the Moore function. We can say that the buffered inputs are part of the state.

A Moore function should also not modify the states (❸). The Transition functions of the next cycle would not see the new states, because the states are updated only after these Transition functions are executed. This could cause lost cycles or even wrong transitions.

Modifying the states in the Mealy functions (❹) may also determine lost cycles and wrong transitions for the same reasons as in the Moore functions case. For setting the outputs, beside the buffered inputs, a Mealy function can also read the current value of the input ports (❺).

5.2.2 Architecture construction

Like for the dynamic scheduling, the architecture construction consists of modules and communication channels instantiation and connection of the ports to the corresponding channels. No instantiation and no connection are possible after the simulation has begun.

SystemCass is a static scheduling version of the SystemC for models that respect Table 5.1 constraints. The interface classes and their methods provided by SystemCass for

Functions	State		Port		Clock edge	Number of calls per cycle
	Read	Write	Read	Write		
Transition	✓	✓	✓	✗❶	↗	1
Moore	✓	✗❷	✗❸	✓	↘	1
Mealy	✓	✗❹	✓❺	✓	↘	1 + nb_deltas

Table 5.1: Constraints and characteristics of the FSM functions

architecture modeling are a subset of those provided by SystemC. SystemCass removed all interface elements that would prevent the static scheduling.

The SystemCass processes can have only static dependencies. A process can not stop its execution in waiting for any kind of events.

SystemCass counts only the number of cycles, *i.e.* it does not implement lines 6 and 16 from Listing 5.3. From this point of view, we can say that SystemCass uses only one simulation point during a cycle. All Transition, Moore and Mealy functions are seen as executing at the same simulated time.

Several clocks may be instantiated, but all of them will be considered as the same clock having the period of the last clock instantiated. The number of simulated cycles and the global period of the clock are used for computing the simulated time, when it is requested. Hardware architecture constructed with the SystemC modeling language and which respects the FSM constraints can be compiled and simulated with both SystemC and SystemCass. The simulation results will be identical, but the simulation using the SystemCass version will be faster.

However, what is happening behind the scene of the elaboration phase for a static scheduling simulator and for a dynamic scheduling simulator is completely different.

Each process keeps a list of its sensitivities. An element of this list contains the input port and the type of the event on which the process is sensitive. There are only three types of events: value changed, negative edge and positive edge. These lists are filled during the instantiation of the modules, when the processes declare their sensitivities. Also, for each signal it is maintained a list of ports binded to that signal.

In SystemCass, the current value is not maintained by the signals to which the ports are connected, but by the output ports themselves. The current value is directly modified when a process writes to the output port. Each input port contains a pointer to the current value of the output port that controls the signal to which it is binded. This way, the signals that connect the ports are not used during the simulation.

Before the beginning of the simulation, the list of ports maintained for each signal is used for setting the pointer to the current value of the input ports.

The type of each sensitivity item from the sensitivity list of a process is determined based on the type of the signal to which the input port of that item is binded and based on rest of the ports from the list maintained for that signal. The dependency type includes dependency on a clock, on a state register and on the value of an output port.

The processes are then sorted according to the type of their sensitivities. The processes that are dependent only on the positive edge of clocks are added to the Transition function list. A process is included in the Moore function list if each of its dependencies is either on the negative edge of a clock, either on a state register. The Mealy function list contains the processes that have at least one dependency on the value of an output port.

Notice that there can be processes that are not included in any of the three lists. An example is a process that is sensitive to the positive or negative edge of an output port

value. These processes will be completely ignored in the simulation. There are cases when a process is included in one list, but it should also be included in other list(s). It is the case of a process dependent on the positive edge of a clock and on the value of an output port. These processes will not be called each time they require. All these processes are unsatisfied due to the fact they do not comply to the FSM rules.

The processes from a list can be executed using a loop. For avoiding the cost of the list iteration, a function that unrolls the calls of the processes can be generated. The Listing 5.4 presents an example of the generated function for a Transition function list. The listing presents the calling of the transition process of an ARM processor, a cache and a memory module. These generated functions are compiled as a dynamic library.

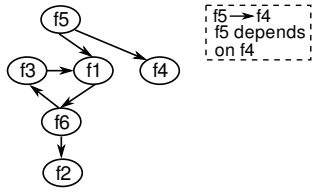
Listing 5.4 Example of generated function for a Transition function list

```
1  typedef void (*ENTRY_FUNC)(void *);
2  typedef union {
3      unsigned long long int integer;
4      ENTRY_FUNC pf;
5  } fct;
6
7  inline void transition_processes (void) {
8      register fct p;
9
10     p.integer = 0x808db60ULL;
11     p.pf ((void *) 0xa506618);  // ARM1->transition ()
12     p.integer = 0x80c3b60ULL;
13     p.pf ((void *) 0xa5099a8);  // CACHE1->transition ()
14     p.integer = 0x80c7f00ULL;
15     p.pf ((void *) 0xbf821818); // RAM0->transition ()
16     ...
17 }
```

For speeding up the simulation, SystemCass avoids the repeated execution of all Mealy functions until system is stabilized. For this, a graph of dependencies between the Mealy functions is constructed. A Mealy function is considered to be dependent on all Mealy functions in the modules that contain at least one output on which that Mealy function is dependent. The resulted graph is topologically sorted and the strong components are extracted. A Mealy function that does not take part of a dependency cycle will be executed only once, after all its dependencies have been executed. The Mealy functions that form a dependency cycle will be executed in a loop until they stabilize between them. As example, the Mealy functions with the dependency graph presented in Figure 5.4(a) can be stabilized with the code depicted in Figure 5.4(b). There can be several equivalent optimized solutions for a dependency graph. A function using pointers instead of the objects and function names (as the Transition function in Listing 5.4) is generated for the found optimized solution. The call to the generated function will replace the stabilizing code from Listing 5.3 (lines 10 to 14).

5.3 Proposed static scheduling simulators accepting multiple and dynamically changing frequencies

Our goal is to be able to model and simulate architectures that contain components working at different frequencies from the rest of the system. An example of such architecture is



(a) Dependency graph

```
f2 ();
do {
   stable = true;
   f3 ();
   f6 ();
   f1 ();
} while (!stable);
f4 ();
f5 ();
```

(b) Optimized stabilization solution

Figure 5.4: Example of optimized stabilization of Mealy functions

given in Figure 5.5. The TTY, the memory, the interconnect and the other peripherals in the figure are working at the same frequency, the one given by the main clock. The frequency of each processor and its cache is controlled by a DVFS component. The DVFS components work at the frequency of the main clock and are connected to the interconnect. The software running on the processors can change anytime the frequency of any processor by issuing a command through the interconnect to the corresponding DVFS component. The frequency adapters make possible the communication between caches, working at the frequency provided by the DVFS components, and interconnect, working at frequency of the main clock.

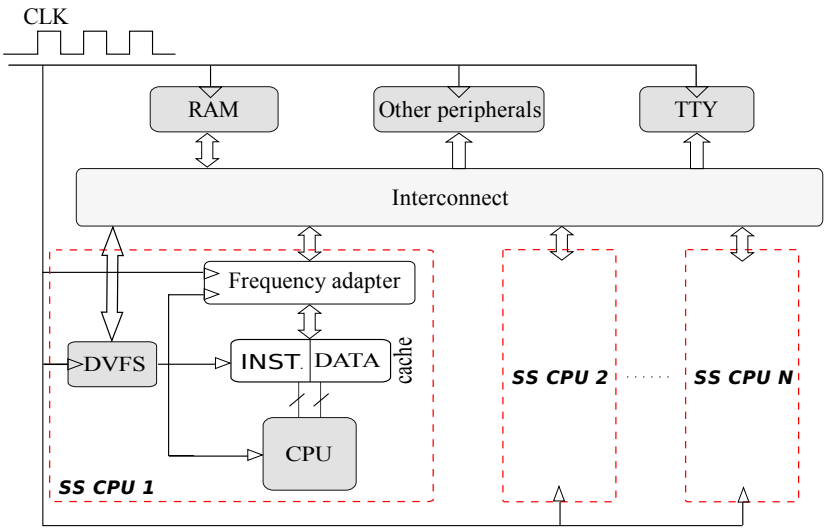


Figure 5.5: Example of architecture with several frequencies

Such architectures can easily be modeled with the HDL of a dynamic scheduling event driven simulator. The output of a DVFS represents the clock of its corresponding processor and caches. For generating the clock edges on this output, the DVFS has several possibilities. A possibility is to use timed events in similar way to the SystemC clock module. Another possibility is to divide the frequency of another clock, by waiting for a number of edge changes of this new clock before changing value of the output. The new clock may be internal or external to the DVFS.

On the other hand, such architectures can not be modeled and simulated by a static scheduling simulator. The processor and their caches can not have as clock other signals than the main clock. Their FSM functions could be sensitive to the DVFS components

outputs, but only on the change value event. In this case, the FSM functions would be considered as Mealy functions and they would always be executed on the negative edge of the main clock. Due to the restriction ❹ from Table 5.1 that prohibits the status changes from the Mealy functions, the processor and cache FSMs could not be implemented. Regarding the generation of the clock edges to the DVFS output, the techniques used in the dynamic scheduling are not available because the timed events do not exist and all clocks of the architecture are considered to be the same clock.

The static scheduling simulation strategy must be adapted for dealing with architectures using several frequencies. We propose two approaches which are able to statically schedule the processes of such architectures.

To prove that the proposed simulators have a correct behavior, we will demonstrate that their behavior is identical to an event driven simulator for any simulated architecture. Two simulators are considered to behave identically if they produce the same output value, at the same simulated time, for all components of the simulated architecture.

5.3.1 Static scheduling simulator based on multiple clocks

In our first solution for static scheduling of the architectures having components that work at different frequencies, each signal that is used by some components as clock signal is generated by an actual clock component. The architectures modeled for this approach use multiple clock components and do not contain components whose output port represents the clock signal for other components as in Figure 5.5.

We developed two versions of this simulator. The static version can only simulate architectures containing components whose working frequencies may be different but do not change during the simulation. The frequencies used are the ones corresponding to the periods specified when the clock components are instantiated.

The dynamic version of the simulator accept the modification of the working frequency during the simulation. Architectures as in Figure 5.5 can be simulated using the dynamic version of the simulator if their model is changed accordingly.

5.3.1.1 Static version of the simulator

The main idea of this approach consists in choosing the period of the scheduling pattern equal to the least common multiple (LCM) of the periods of the clocks.

The example given in Figure 5.6 includes three clocks with the cycle period of 4 ns, 6 ns, respective 2 ns (frequencies 250 MHz, 166 MHz and 500 MHz). The period of the scheduling pattern will be in this case 12 ns.

Each clock has a positive and a negative edge. The edges of a clock succeed normally during the scheduling pattern period. As the period of the scheduling pattern is multiple of the cycle period of any clock in the architecture, at the end of a pattern, each clock will be at the same offset in its period as it was at the beginning of that pattern. In the example, all clocks begin the pattern with their positive edge.

Along the pattern period, the edges of several clocks may overlap. The moments that contain at least one clock edge represent the simulation points of the scheduling pattern. The number of simulation points in a pattern depends on the number of clocks, the positive and negative edge time of the clocks and the offset in their period at the start of the pattern. The pattern of the presented example has twelve simulation points.

The proposed simulator uses two level of scheduling. The first scheduling level consists in setting the next simulation point, managing the simulated time and calling the second scheduling level.

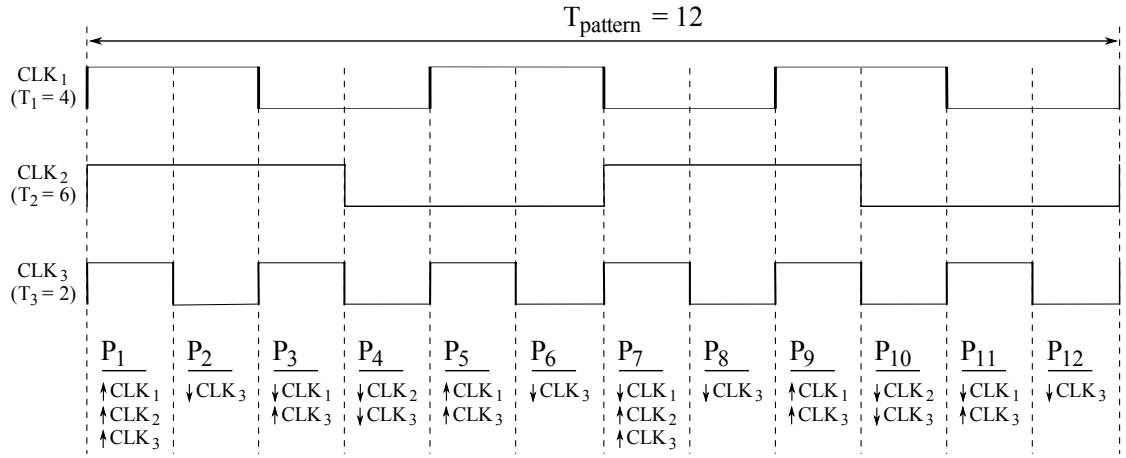


Figure 5.6: Execution pattern for multiple clocks architecture

The processes depending on the clock edges of the current simulation point are called in the second scheduling level. A simulation point may contain both positive and negative edge types. In the example from Figure 5.6, for the third simulation point (P_3) the negative edge of the first clock overlaps over the positive edge of the third clock.

In a dynamic scheduling event driven simulator, the timed events corresponding to the clock edges of a simulation point would have the same time stamp and the processes from the static list of these events would be all executed in the same delta cycle. The written values to the output ports and to the status registers would be visible only after the end of the delta cycle. In the possible case when delta cycles are generated, only Mealy functions would be executed because the Transition and Moore functions do not depend on the outputs.

The scheduling scheme for a simulation point is depicted in Listing 5.5. This scheduling scheme assures an identical behavior to an event driven simulator. The scheduling scheme is based on the fact that the simulator applies the SystemCass optimizations regarding the writing to the output ports and the Mealy function scheduling. Due to the writing to the output ports optimization, all Transition functions depending on positive clock edges in the simulation point must be executed before the Moore functions implied by the simulation point. Otherwise, a Transition function executed after a Moore function would see directly the value just written by that Moore function, instead the right value from the beginning of the delta cycle. This could lead to wrong state transitions and finally a wrong behavior.

Listing 5.5 Simulation point execution for a multiple clock scheduler

```
1 void simulation_point (void) {
2     transition_processes ();
3     moore_processes ();
4     update_registers ();
5
6     mealy_processes ();
7 }
```

The update of the status registers (line 4) is done after executing all Moore functions depending on negative clock edges in the simulation point. This is because the Transition

and Moore functions of the simulation point should be executed in the same delta cycle. A Moore function depending on a clock should see status value from the beginning of the delta cycle, not the one modified by a Transition function from the same module, but depending on another clock. The status registers update marks the end of the first delta cycle of the simulation point.

The Mealy functions are then stabilized using the SystemCass optimization solution. Unlike the Transition and Moore functions, which are called only in the simulation points where they are implied, all Mealy functions are called at each simulation point.

Regarding the simulator implementation, unlike SystemCass, each clock is considered separately with its own period and ratio between the positive and negative edge duration.

The component and signal instantiation and the binding of the component port to the signal are conducted normally, as in SystemCass.

For each clock instantiated in the system it is maintained a list of processes sensitive to the positive edge of the clock and a list of processes sensitive to the negative one. When the processes are sorted, each Transition and Moore function is added to the corresponding list of the clock to which it is sensitive.

The processes lists of a clock do not change during the simulation, not even if the frequency of the clock changes, as we will describe later. For avoiding the iteration cost of these lists, a function is generated for each list. A generated function example for the transition list of a clock is presented in Listing 5.6.

Listing 5.6 Generated function example for a clock

```
1  void transition_clock_0xbfb10238 (void) {
2      //clock clk_400MHz
3      //timer0.transition
4      ((ENTRY_FUNC) 0x80950a0ULL) ((void *) 0x9a46600);
5      //gmn.transition
6      ((ENTRY_FUNC) 0x807ece0ULL) ((void *) 0x9cc2990);
7      //ram0_text.transition
8      ((ENTRY_FUNC) 0x809f600ULL) ((void *) 0x9e4fad0);
9      //ram1_data.transition
10     ((ENTRY_FUNC) 0x809f600ULL) ((void *) 0x9edb478);
11     ...
12 }
```

The list of simulation points of the initial scheduling pattern is built before the beginning of the simulation. Each simulation point contains a list of positive and a list of negative clock edges that have to be treated in that simulation point. In fact, the list of clock edges stores the pointers to the generated functions for those clock edges. As example, one list for the simulation point P_7 in Figure 5.6 contains the pointers to the generated functions for the positive edge of the second and third clocks and the other list contains pointer to the generated function for the negative edge of the first clock. A simulation point also contains a time offset that has to be added for passing to the next simulation point.

During the simulation, the list the pointers to the generating functions are used for calling all Transition and Moore functions of the simulation point.

For speeding up the simulation, a function that unrolls both scheduling levels can be generated. This function would unroll all simulation points of the scheduling pattern and would update the simulated time after each simulation point. For each simulation point, the lists used for calling the Transition and Moore functions would be unrolled. The simulator can in this case model architectures with multiple clocks and simulate them at

the speed of SystemCass.

The HDL of this static version of the proposed static scheduling simulator does not change over the HDL of the simulator from which it is derived. The person who models the architecture does not have to be aware that another simulator is used.

5.3.1.2 Dynamic version of the simulator

For simulating architectures using multiple frequencies that can change during the simulation, several adjustments have to be done both to the simulator and to its HDL. We target architectures whose input signals used as clock signals by the hardware components of that architecture respect the characteristics of a clock signal: periodicity and a rate between the positive and negative edges. The frequency of these input signals can change during the simulation.

An example of the targeted architecture was already presented in Figure 5.5. For modeling this kind of architecture, the output ports that control the clock signal of other components are replaced by actual clocks. In SystemCass HDL terms, these *sc_out*s of the components that are at the border of two clock domains are replaced by *sc_clock*s.

In Figure 5.7, the output port of the DVFS components is replaced by a clock component. The processor, the caches and the frequency adapter will be directly connected to this clock component. Doing so, the architecture can be simulated only with the frequencies initially specified for the DVFS clocks.

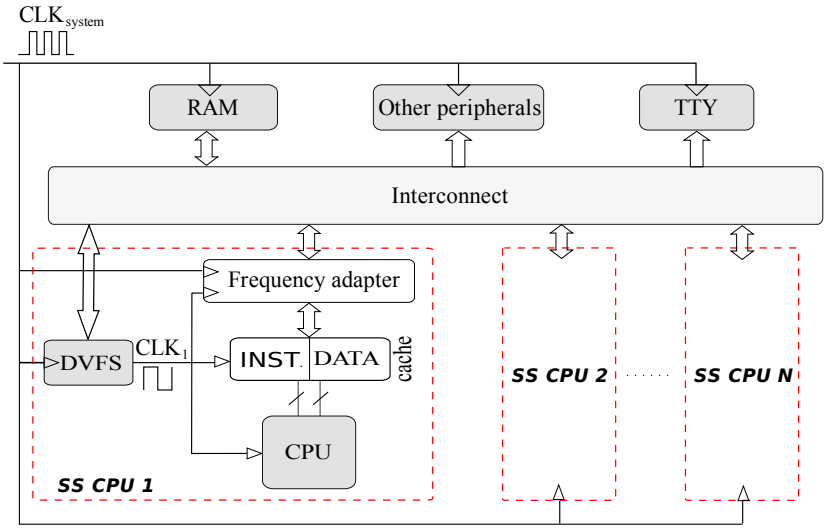


Figure 5.7: Modeling multiple frequencies architectures by using multiple clocks

For having a mean of modifying the frequencies of the clocks, we introduce the following API function to the clock component:

void change_period (double new_period, double new_duty_cycle = 0.5);

In the example, instead of continuously generating the output signal viewed as clock by other components, the DVFS component has to call now this new API function when it wants to change the output frequency.

This function will be called on an edge of one clock and will modify the frequency of another clock. Figure 5.8 presents an example of frequency change for the clocks in Figure 5.6. A Moore process executed on the negative edge of the third clock (simulation point P_8) wants to change the frequency of the second clock.

The frequency change for a clock is not done necessarily at simulated time of the request. If the clock whose frequency has to be changed does not change its value in the simulation point of the request, the change is postponed until the first edge event of that clock. We have done this choice because it is probable that a hardware device would do so to avoid glitches. In the example, the frequency change of the second clock is postponed until the negative edge arrives at simulation point P_{10}.

A list of future frequency changes is kept. Each item of this list contains a pointer to the clock whose frequency has to be changed, the programmed time for change and the future properties of that clock (period, duty cycle and which edge will be the first). This list is verified at the end of each simulation point.

If at current simulated time at least a clock frequency has to be changed, a new scheduling pattern is dynamically built. In the example, considering that the new period for the second clock is 2 ns, the new period will be 4 ns.

The edge events of the clocks whose frequency does not change should occur in the new scheduling pattern at the same simulated time as they would occur in the old scheduling pattern. In other words, while the frequency of a clock does not change, the waveform of that clock remains constant. Usually there are clocks that do not begin the pattern with an edge. In the example, the first clock will have the first edge event at an offset of 1 ns from the beginning of the pattern. Also, the first clock does not end the pattern with an event. As the pattern period is multiple of any clock period, the time left after the last event of a clock to the end of the pattern, with the time from the beginning of pattern to the first event of that clock, completes exactly the next edge time of that clock.

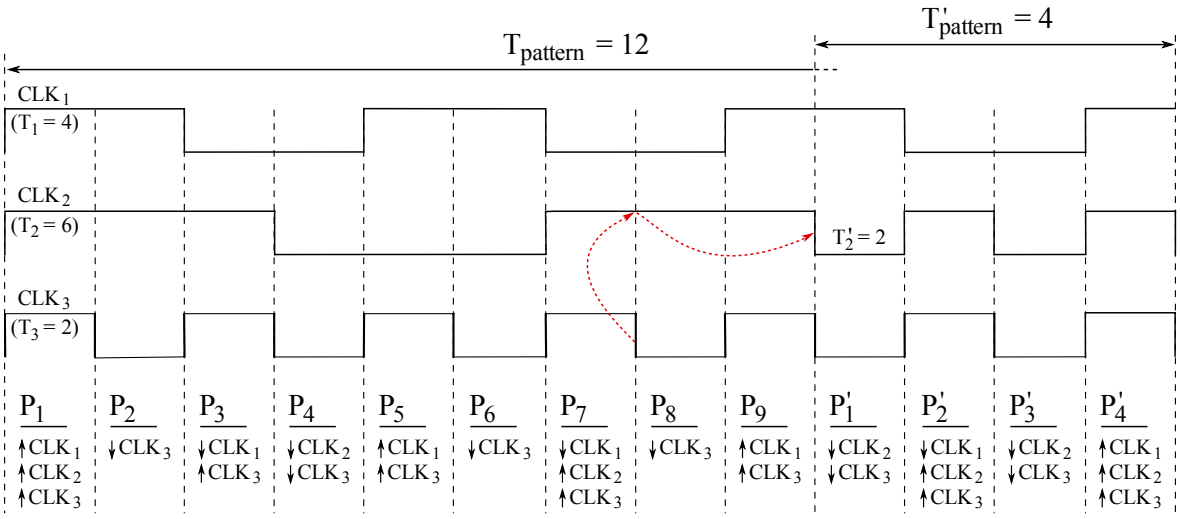


Figure 5.8: Changing frequency in a multiple clocks architecture

The proposed static scheduling simulator can now simulate architectures with multiple and dynamically changing frequencies.

The HDL is a little bit modified and the architecture model is not the same as for a dynamic scheduler. Components like DVFS have to be written specially for this simulator.

The simulation speed is lower than that of the static version of simulator. In fact, the simulation speed is strongly influenced by the number of frequency changes. The scheduling pattern must be rebuilt for each frequency change. It could be possible to generate at the beginning of the simulation or with an external tool a function for each possible combination of frequencies for the clocks. As example, for 4 clocks each having 3

frequency levels, 81 functions should be generated. When a frequency would change, the right generated function would be chosen.

The only problem is that the frequencies do not necessarily change at the limit of the scheduling pattern. Because of this, not all clocks have the positive edge at the beginning of the new pattern. As the pattern of all generated functions begins with the positive edge of all clocks, the two scheduling patterns do not fit. The number of all possible combinations of frequencies and offsets explodes and it is not feasible in practice to generate the corresponding functions. A possibility would be to postpone the frequency change until the end of the scheduling pattern. This would reduce the accuracy of the simulation.

For that LCM of the clock periods can be computed, there must exist a time unit allowing these periods to be expressed as integer values.

The least common multiple of the clock periods is sometimes big and so is the number of simulation points. Generating scheduling patterns with a large number of simulation points also reduces the simulation speed. Sometimes the transformations from frequency to period represent the cause for the large number of simulation points. Let us consider as example three frequencies: 1000 MHz, 750 MHz and 500 MHz. The corresponding periods for these frequencies are not in a harmonic ratio. Their value can be approximated to 1000 ps, 1333 ps and 2000 ps. The LCM for the periods is 2666000, which is 1333 times bigger than the longest period. Multiplying these periods by 3, the LCM of the new periods (3 ns, 4 ns and 6 ns) would be 12, which is 2 times bigger than the longest period. This operation would reduce drastically the number of simulation points. For allowing such operations, we introduced an API through which the multiplication factor can be specified. When it is requested, the real simulated time is computed by dividing the simulated time to the multiplication factor. By default, the multiplication factor is equal to 1.

The limitations of this approach are presented in Limitation 5.1.

Limitation 5.1 Multiple clocks based simulator approach limitations

- Can not be used for architectures containing input signal used as clock signal that does not has the waveform of a clock signal
- The clock periods must be in a harmonic ratio to allow their LCM computation

5.3.2 Static scheduling simulator based on frequency division

The second solution for the static scheduling of the architectures containing components working at different frequencies consists of choosing the frequency of the single clock in the architecture equal to the least common multiple of all possible frequencies of all components in the architecture.

The frequencies required by hardware components are obtained by dividing the LCM frequency. The period of the scheduling pattern will the period corresponding to this new frequency. The scheduling pattern contains two simulation points, corresponding to the positive and negative clock edges.

Figure 5.9 shows how the architecture presented in Figure 5.5 is modeled using the frequency division technique.

The interconnect, the memories, a part of the frequency adapters and all other peripherals excepting the caches and processors are working at the same frequency. This frequency does not change during the simulation. A new introduced component (called *Frequency*

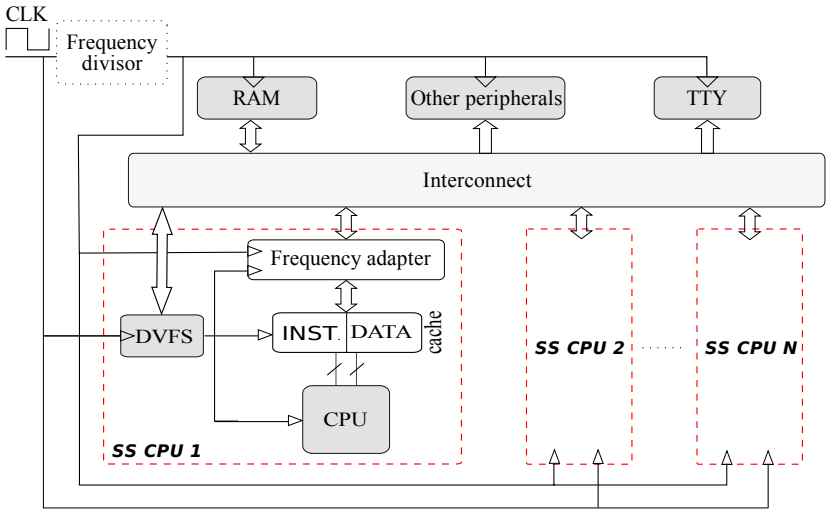


Figure 5.9: Modeling multiple frequencies architectures by dividing a single frequency

divisor in the figure) obtains this frequency dividing the LCM frequency by an integer constant. If the constant divisor is 1, the frequency adapter is not necessary. The frequency of each processor, its caches and the other part of its frequency adapter is controlled by a DVFS component. So, all components excepting the frequency divisors and the DVFSes use as clock the output of another component.

For accepting dependency on an edge of an output signal, we modified the way a cycle is simulated. Listing 5.7 presents the simulation of a cycle. It assumes that the simulator applies the SystemCass optimizations regarding the writing to the output ports and the Mealy function scheduling. Comparing to the standard cycle simulation, a call of the processes depending on an output signal was added (lines 7 to 9).

Listing 5.7 Cycle simulation in the frequency division based simulator

```
1  void simulate_a_cycle (void) {
2      transition_processes ();
3      update_registers ();
4
5      moore_processes ();
6
7      out_signal_clock_transition_processes ();
8      out_signal_clock_moore_processes ();
9      update_registers ();
10
11     mealy_processes ();
12 }
```

The Transition processes having as clock an output signal are grouped together in a list. The Moore functions are grouped in another list. The processes from these lists are called only when their required edge condition is met. As usually, the Transition functions should be executed before the Moore functions, otherwise wrong state transitions could occur. In a dynamic scheduling event driven simulator, these Transition and Moore processes would be all executed in the same delta cycle. For simulating the same behavior, the registers are updated after the execution of the Moore processes.

According to the FSM rules, only the Moore and Mealy processes can write to the output ports. The Mealy processes should not participate at the clock signal generation because their output variation during the stabilization phase would create fake cycles for the components for which that output represents the clock signal. These are the reasons why the processes depending on an output signal are not called after the execution of the Transition and Mealy processes.

The restriction of writing from the Transition processes makes impossible the implementation of a divisor capable of making a division by 1. A solution for solving this problem is to double the frequency of the system clock. The division by 1 becomes a division by 2, which can be now implemented. At least half of the simulated cycles will not be used.

Another solution would be to disobey the FSM rule and to allow the Transition process of the frequency divisors to generate the clock signal. This is not a problem if the generated signal is used only as clock signal by other components. In this case, the processes depending on an output signal must also be called before the Moore processes (line 5 in the listing).

All Mealy processes in the architecture are stabilized together, no matter of the type of their clock signal.

Regarding the simulator implementation, the two lists of processes depending on output signals are constructed when the processes are sorted.

Before the beginning of the simulation, a function is generated for each of these two lists. Figure 5.8 is an example of such generated function. For each output signal on which at least one process depends, a static variable is created. These variables maintain the old value of the output signals, as only the current value is maintained by default by the output ports. The type of the variable depends on the data size of the signal. Using the two values we can determine if the required event occurred. When an event occurs, all processes from the list depending on that event are executed.

Listing 5.8 Generated function for processes sensitive to the positive edge of output signals

```
1  inline void out_signal_clock_transition_processes (void) {
2      register fct p;
3
4      static unsigned char val1 = 0;
5      if (*(unsigned char *) 0x934cb84ULL != val1) {
6          val1 = *(unsigned char *) 0x934cb84ULL;
7          if (val1) {
8              // ARM1->transition ()
9              p.integer = 0x80a7160ULL;
10             p.pf ((void *) 0x86d1fa8);
11             // CACHE1->transition ()
12             p.integer = 0x80ae630ULL;
13             p.pf ((void *) 0x86d4128);
14             // FQADAPTER1->transition_initiator ()
15             p.integer = 0x8089cb0ULL;
16             p.pf ((void *) 0x9355cb0);
17             ...
18         }
19         ...
20     }
21 }
```

The presented static scheduling simulator can simulate architectures with multiple and dynamically changing frequencies. It can not simulate architectures with multiple clocks. Only one clock is used and the rest of frequencies are obtained by dividing the frequency of this clock.

A single level of signal dependency is allowed. It is not allowed that a component, whose clock signal is given by the output of another component, to generate the clock signal for other components.

The HDL is not modified for this simulator. However, the architecture model designer must be aware of the simulator characteristics. The multiple clock simulator presented in 5.3.1 computes itself the LCM of the clock periods when it is required. For this simulator, the architecture designer must compute the LCM of all possible frequencies when he constructs the architecture. He must also add the frequency divisor and set its division constant.

This simulator can not be used if there are frequencies in system unknown when the architecture is constructed. An unknown future frequency makes impossible the LCM computation.

The ratios between the system clock frequency and the frequencies of the components influence the simulation speed. Big values for these ratios determine a large number of unused simulated cycles. Only the Transition and Moore functions of the frequency divisors and the Mealy functions of all components are executed in these cycles, slowing thus the simulation.

The number of frequency changes does not influence the simulation speed. As the frequency of the system clock is multiple of all frequencies in the system, no particular computation takes place when a frequency is changed. The price of this multiplicity is paid all the time during simulation through the unused cycles.

The limitations of this approach are presented in Limitation 5.2.

Limitation 5.2 Frequency division based simulator approach limitations

- Can not be used if there is at least one frequency unknown when the architecture is modeled

5.3.3 Approaches comparison

Both approaches can simulate the architectures using multiple frequencies that can be changed anytime by the software running on the processors. While the simulation speed of the multiple clocks based simulator is diminished by the frequency changes, the unused cycles reduces simulation speed of the second simulator. The multiple clocks based simulator is expected to be faster than the frequency division based simulator if the frequencies do not change often. Their effectiveness will be presented in the next section.

5.4 Experiments

We have tested the proposed cycle accurate static scheduling simulators through two case studies. The first case study uses the Motion JPEG/SoCLib application/hardware architecture presented in section 4.8.1. For a higher flexibility in simulation, the second case study uses a configurable synthetic hardware architecture.

5.4.1 Motion JPEG/SoCLib

In this case study, the software stack consists of the Motion JPEG decoding application running on top of Mutek operating system. The software stack is cross-compiled for the target processor and simulated on the SoCLib components based hardware architecture model. The hardware architecture model is compiled and linked with each simulator library that participates in the comparison. We obtain this way several executable corresponding to the SoCLib based architecture simulated with the following simulators: SystemC, SystemCass, frequency division based static scheduler simulator and multiple clock based static scheduler simulator.

Table 5.2 depicts the results obtained by the simulators for a single frequency, respectively, for multiple frequencies in the architecture. For the configuration with a single frequency, both hardware architecture and software were changed. On the hardware side, the frequency adapters and the DVFS components were removed. The data and the instruction caches are connected directly to the interconnect. The processors and the caches use the system clock signal. On the software side, all frequency changes have been removed. For the configuration with several frequency, the hardware architecture is the one presented in section 4.8.1. The working frequency of the processors and their caches is controlled from the simulated software, by the energy saving algorithm presented in chapter 6, which has been applied to the operating system. The DVFSes can generate 4 frequencies, which are 1, 0.75, 0.5, respectively, 0.25 of the frequency at which the rest of system works. The energy saving algorithm generates an average of about 1.5 frequency changes per simulated millisecond.

The results obtained by SystemC are considered as reference for the rest of the simulators.

As we can see in the table, for this case study, SystemCass is 3.5 times faster than SystemC, when the architecture contains only one frequency. This speedup factor depends on the average number of hardware architecture model instructions executed for an event generated in SystemC. SystemCass can not simulate architectures containing multiple frequencies.

Simulator	1 frequency		Multiple frequencies	
	Sim. time (s)	Speedup	Sim. time (s)	Speedup
SystemC (reference)	542	1	1136	1
SystemCass	155	3.50	✗	✗
Frequency division	155	3.50	343	3.31
Multiple clocks	154	3.53	298	3.81

Table 5.2: Simulators comparison using the Motion JPEG/SoCLib architecture

The implementation of the proposed static scheduling simulators is derived from SystemCass.

The simulator based on frequency division obtains the same results as SystemCass for the architecture with a single frequency. As no component uses the output of another component as clock signal, the generated C functions, which should call the Transition and Moore processes of this kind of components, are empty. So, for the frequency division based simulator, the code executed during the simulation reduces to the initial SystemCass code. For the architecture containing multiple clock components, the simulation speedup is bit smaller because of the unused cycles from the main frequency.

The simulator based on multiple clocks obtains for the single frequency architecture almost the same simulation speedup as SystemCASS. For multiple frequencies, the speedup is about 3.81. These values are obtained for both versions of this simulator: the one that computes the new scheduling each time a frequency changes and the one that precomputes all possible schedules before the start of the simulation.

5.4.2 Synthetic architecture

In the previous case study, some characteristics of the hardware and software architectures that may influence the simulation speed of the simulators can not be easily varied. The frequency change rate is given by the energy saving algorithm in function of the system utilization. The hardware characteristics (number of components, number of ports per component *etc.*) are fixed by the SoCLib components required by the application.

For being able to vary all these parameters, we have constructed a configurable synthetic architecture. The architecture (Figure 5.10) contains N modules working on the system frequency (*CLK0*). A part of these modules (*NC*) generate the clock signal for other modules. Each generated clock signal is used by M modules. The frequencies set of these clock signals ($CLK1$ to $CLKNC$) is configurable. So, the architecture contains $N + NC \times M$ modules. Each module of the architecture contains a configurable number of input ports, output ports and registers. The output ports and the input ports of the modules working at the same frequency are interconnected in a circular way. Each module has a Moore and a Transition process, which use the ports and the registers. These processes also execute a loop having a configurable number of iterations for allowing the variation of the number of instruction executed for a simulation event. The number of frequency changes in a millisecond can also be set. The frequencies of clock signals are modified in round-robin scheme. The new frequencies are also pre-specified.

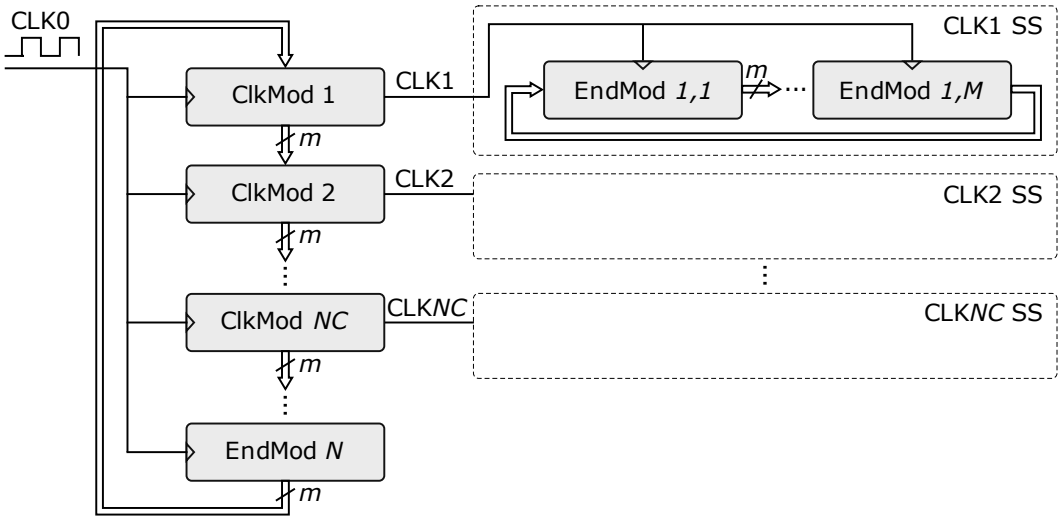


Figure 5.10: Test platform for the static scheduling simulators

This simulation architecture is a purely hardware model. It does not execute any cross-compiled software. The frequencies are changed by the *NC* modules at equal intervals of time, so that the required frequency change rate is respected. As in the first case study, the hardware architecture model is compiled and linked with each simulator library.

Figure 5.11 presents the proposed simulators speedup dependency on the number of modules in system. The simulations in this figure were performed using a single frequency and 10 loop iterations. The simulation speedup decreases as the number of modules increases. This is because the cost of the SystemC dynamic scheduling decreases with the number of modules, as the same scheduling is used by more modules. For 2 modules the speedup is about 13, while for 300 modules it is about 3.6.

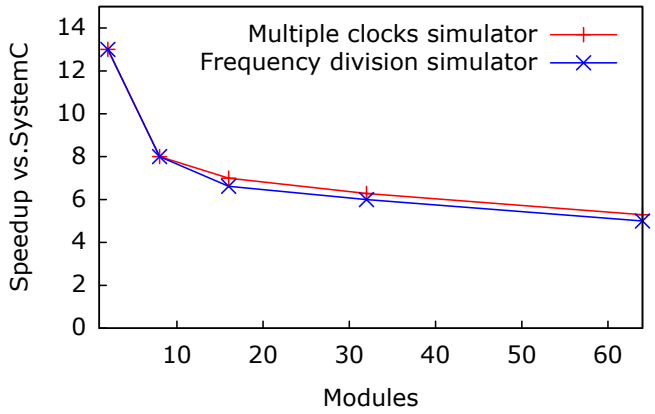


Figure 5.11: Speedup dependency on the number of modules

Figure 5.12 shows the influence of the loop iteration number on the studied simulators speedup. The simulations in the figure use a single frequency and 30 modules. As we can see, the simulation speedup decreases with the number of iterations, as the time required by SystemC for dynamic scheduling becomes less significant.

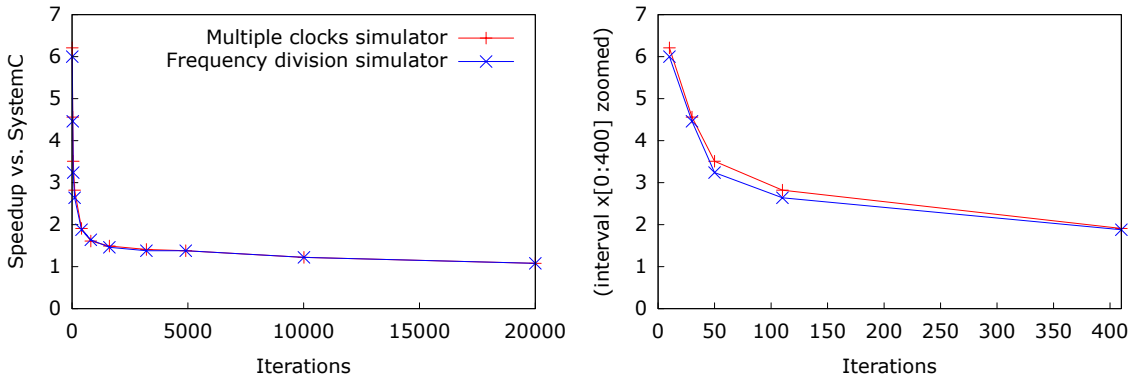


Figure 5.12: Speedup dependency on the loop iterations number

Figure 5.13 presents the speed dependency of the multiple clocks simulator on the number of frequency changes and the number of simulation points. The figure presents 5 curves corresponding to 5 different sets of frequencies. For each curve is given in parenthesis the virtual number of simulation points for its frequency set. This number is computed using formula: $2 \times \sum_{i=1}^{m} \frac{LCM_{T_1-T_m}}{T_i}$, where m is the number of frequencies in the set and T_i is the period of the frequency i. The real number of simulation points is lower than this number, as several clock edge events may share the same simulation point. The tests in this figure

use 42 modules and 40 loop iterations. The simulation speed decreases with the number of simulation points as building a new scheduling when the frequency of a clock signal changes takes longer. For large number of simulation points and frequency changes, the simulation speed falls below that of SystemC. The Motion JPEG decoding application with the low energy algorithm presented in the first case study corresponds to curve *G*1.

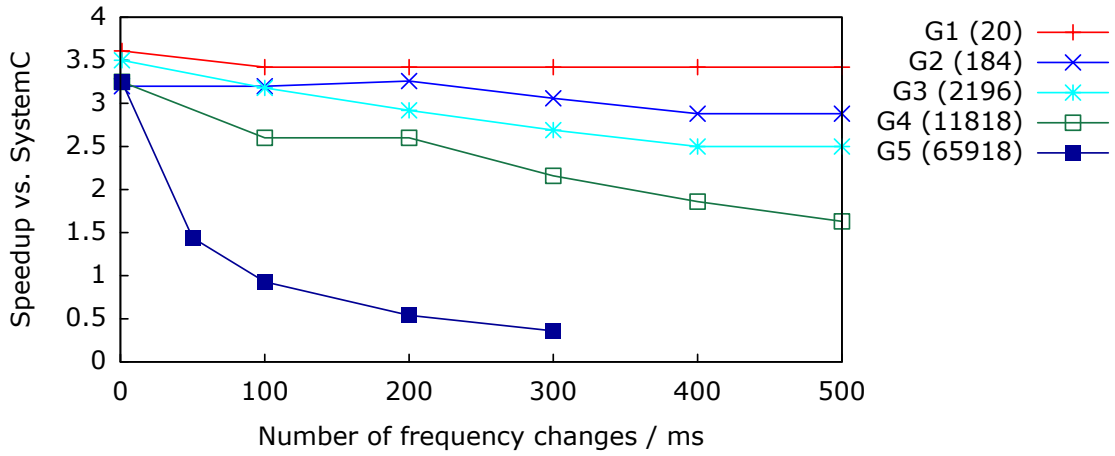


Figure 5.13: Multiple clocks simulator speed

Figure 5.14 presents the results obtained by the frequency division based simulator for the same sets of frequencies. The hardware architecture model contains for this simulator an extra frequency division module, which obtains the system frequency from the unique frequency. As we can see, the simulation speed does not depend on the number of frequency changes, but on the ratio between the LCM of the frequencies set and the system frequency. A high ratio determines many unused cycles. This ratio is specified in parenthesis for each curve.

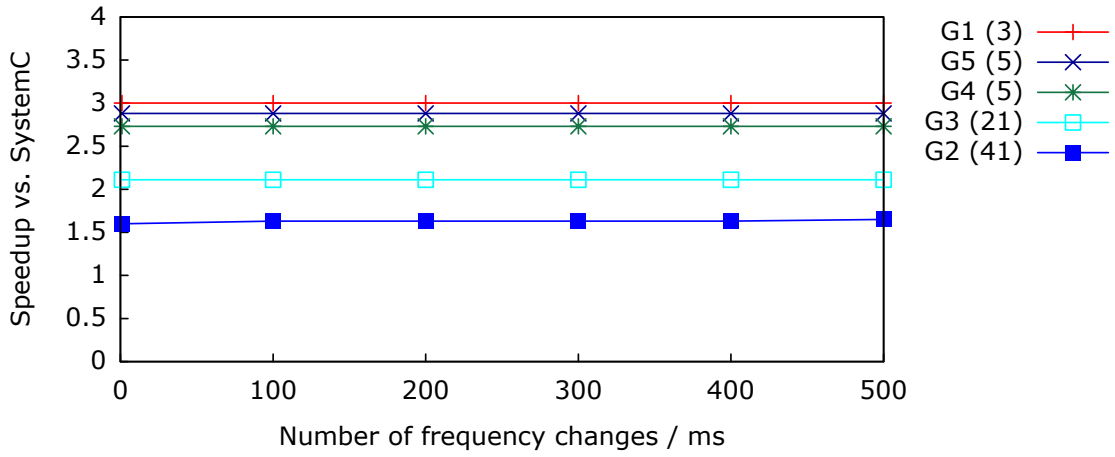


Figure 5.14: Frequency division simulator speed

5.5 Conclusion

The simulation strategies presented in this chapter are modeled at CA abstraction level and use static scheduling for simulation. These simulators use different approaches for supporting runtime changing multiple frequencies. The first one uses multiple clock components permitting runtime change of their period. The second one is based on the division of a single frequency.

The proposed simulators offer an unnegligible simulation speedup comparing to a simulator using dynamic scheduling. For some frequency sets, the multiple clocks simulator is faster, while for others, the frequency division simulator obtains better results.

Chapter 6

Adaptive DVFS algorithm for SMP architecture

Contents

In this chapter, we describe our adaptive energy saving algorithm for SMP architectures running a preemptive non-RTOS. The algorithm is implemented at the operating system level and it assumes that the applications running on the operating system do not provide any information about their processing requirements. Unlike the RTOS, where the deadline and the WCET (worst-case execution time) of each task in the system are known, none of this information is known on a non-RTOS.

Like any other DVFS algorithm, the proposed algorithm saves energy by reducing the processors frequencies and voltage when the system utilization permits. A part of the idle time that would occur due to the lack of computation is in this way replaced by the computation stretched by the frequency reduction.

Before presenting the algorithm, we will first present the difficulties faced by the energy saving algorithms designed for non-RTOSes. Some problems apply to both monoprocessor and SMP architectures, others only to the SMP architectures.

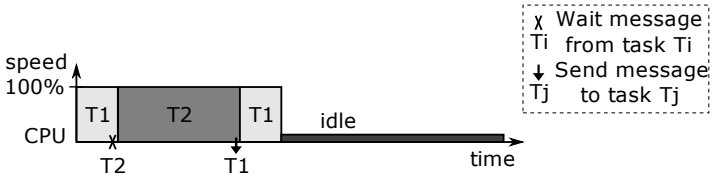
6.1 Motivational example

PAST, proposed by Weiser *et al.* [WWDS94] and enhanced by several other researchers, is one of the most efficient energy saving algorithms for mono-processors running a non

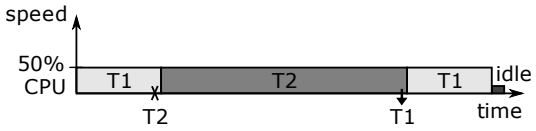
RTOS. Its main idea is to divide the execution time in equal time intervals and, for each interval, to adapt the processor frequency so that the same amount of useful work as in the previous interval can be executed. The required frequency is computed and set only at the beginning of each interval. This subsection explains why this kind of algorithm can not be directly applied to SMP architecture.

6.1.1 Mono-processor case

Figure 6.1 presents scheduling in time of two tasks, T_1 and T_2, of an application on a mono-processor system. T_1 requires some data from T_2. Figure 6.1(a) illustrates an interval with no energy saving algorithm applied, while the interval in Figure 6.1(b) applies the PAST algorithm exploiting information from its previous time interval (the one in Figure 6.1(a)).



(a) Interval I_k at maximum frequency



(b) Interval I_{k+1} at an optimized frequency

Figure 6.1: Time scheduling on mono-processor

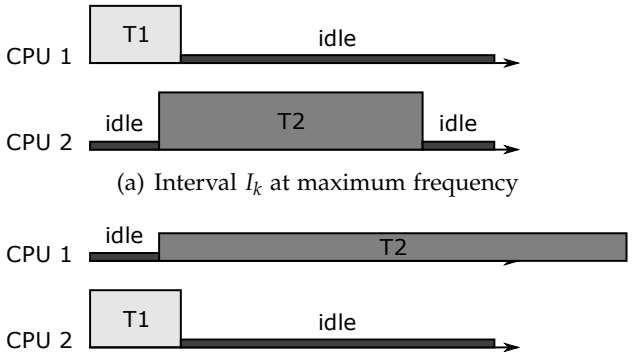
As it can be observed, the algorithm manages to execute the whole workload of the second considered interval at 50% of the processor maximum frequency. It can be equally noticed that the task dependencies do not incur idle periods. Generally, in the mono-processor case, the processor idle periods result from either complete lack of activity or from input/output operations.

6.1.2 SMP Case

The OS of a SMP system can schedule any task at any time during its execution on any available processor. This fact assures good performances and a balanced work distribution on processors. From the adaptive energy saving algorithm point of view, the SMP architecture poses specific problems. These problems come from migration of the tasks without any restriction between processors, parallel tasks execution, synchronization among tasks etc. In the following, we will present a few of these problems.

Figure 6.2 presents the case of two tasks (T_1 and T_2) running on two processors (P_1 and P_2). T_1 has a short execution time and T_2 a long one. Consider the case when a energy saving scheduling algorithm analyses independently the processors utilization.

For processor P_1, as its utilization is low, a energy saving algorithm will try to save energy by considerably reducing the processor frequency. The frequency of P_2 will be reduced much less, because of its higher utilization. However, the scheduler can exchange the tasks on the processors for the next interval. In this case, the execution of T_2 will be



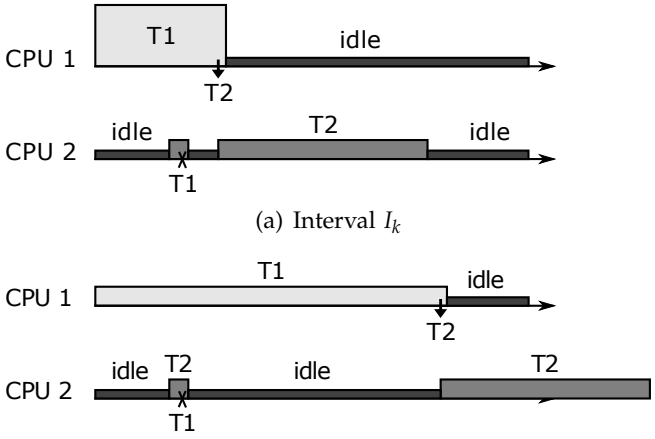
(a) Interval I_k at maximum frequency

(b) Interval I_{k+1} at "optimized" frequency

Figure 6.2: Possible task migration effect of a energy saving algorithm in a SMP architecture

much stretched, while the T_1 will be executed at a greater frequency than the optimum one.

Another problem is presented in Figure 6.3. T_2, after a short time from the beginning of its execution, waits for some data provided by T_1 near the end of its execution. Seeing that the system utilization is low, a energy saving algorithm can take the decision to reduce the processor frequency for the next interval. Doing so, the T_1 execution will be stretched, so T_1 will provide the data required by T_2 close to the end of the interval. Thus, the execution of T_2 will be much delayed, reducing the system performances. Worse than that, the algorithm will consider that the processor P_2 workload has decreased and that the idle time caused by the waiting time after T_1, is a lack of activity time. This kind of problems is due to the misinterpretation of the idle time.



(a) Interval I_k

(b) Interval I_{k+1} at "optimized" frequency

Figure 6.3: Possible synchronization effect of a energy saving algorithm in a SMP architecture

To conclude, SMP architectures differs from mono-processors systems in two major points. First the load balancing of SMP OS scheduling may lead in huge performances decrease due to task migration. Second SMP systems introduce a new source of idle time due to tasks communication and synchronization.

6.2 Exceeding work detection

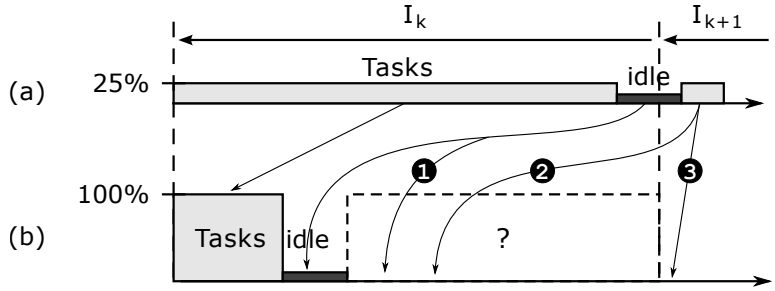
The detection of the exceeding work of an interval represents the biggest problem that affects both types of architectures.

The execution time in Figure 6.4 is divided in equal time intervals, like in the PAST algorithm. Figure 6.4(a) presents the execution of an interval at the frequency computed by a energy saving algorithm (25% in the figure).

Figure 6.4(b) presents the same interval whether it would be executed at full speed. In this case, the useful work would be executed much faster. All work of the interval, useful and idle, would be executed in less time. As the algorithm has no input from the applications, it can not know what kind of work would fill the period since the end of the work from the speed optimized interval to the end of the interval.

This type of energy saving algorithms considers that the workload does not vary much from an interval to another. It reduces the frequency when there is idle time that can be eliminated. Thus, the period with the unknown type of cycles would normally contain idle cycles (❶). These idle cycles could be produced by waiting for a user input, waiting for a period of time to pass *etc*. The useful cycles executed in the next interval in the optimized frequency case would also be executed in the next interval in the full speed case (❸).

If the workload of the executed interval has increased, the period with unknown type of cycles would contain, in the full speed case, a part of work executed in the next interval, in the optimized frequency case (❷). These cycles that should be executed in one interval but are executed later form the exceeding work.



(a) Interval I_k executed at an optimized speed
(b) The same interval virtually executed at the full speed

Figure 6.4: Exceeding work detection problem

The exceeding work should be detected for increasing the frequency in order to recuperate the execution delay or, at least, to not continue increasing the delay.

The area of the possible exceeding work (dashed line in the figure) depends on the optimized frequency of the processors. For the maximum frequency, this area reduces to zero, *i.e.* it is impossible to do more work. The maximum value of the possible exceeding work is obtained when the processors work at the smallest frequency.

6.3 Conceptual Algorithm Description

The proposed energy saving algorithm uses equal intervals of time and expands the workload of each interval over the entire next interval, like PAST and other algorithms for mono-processor systems. The algorithm addresses the SMP specific features including

task migration by a global management of the system workload and thus working frequency. Moreover, for effectiveness reasons, it considers a supplementary order interval subdivision, as detailed further.

The system is modeled as a set of N tasks $T_j \in \{T_1, \ldots, T_N\}$ running on M processors $P_i \in \{P_1, \ldots, P_M\}$ capable of running a frequency $f_i \in \{f_1, \ldots, f_m | f_1 < \ldots < f_m\}$. Time intervals named I_k with non-idle cycles $c_{j,k}$ for task T_j in the kth time interval. Thus c_k, the number of non-idle cycles over the interval k is given by $c_k = \sum_{j=1}^{N} c_{j,k}$. The time intervals I_k have a constant period t_I, which represents C_I full speed (at frequency f_m) cycles. Subintervals will be named S_l. They have a constant period t_S, which represents C_S full speed (at frequency f_m) cycles.

To illustrate the idea, Figure 6.5 shows how the algorithm works using an example application with three tasks (T_1, T_2 and T_3). In this example, T_2 needs some data provided by T_1 at a moment of its execution and T_3 is started by T_2. In interval I_k (Figure 6.5(a)), all three processors are presented working at the maximum frequency. As tasks can migrate from one processor to another each time they are scheduled, T_1 is first executed on P_3, then on P_1 (Figure 6.5(a)).

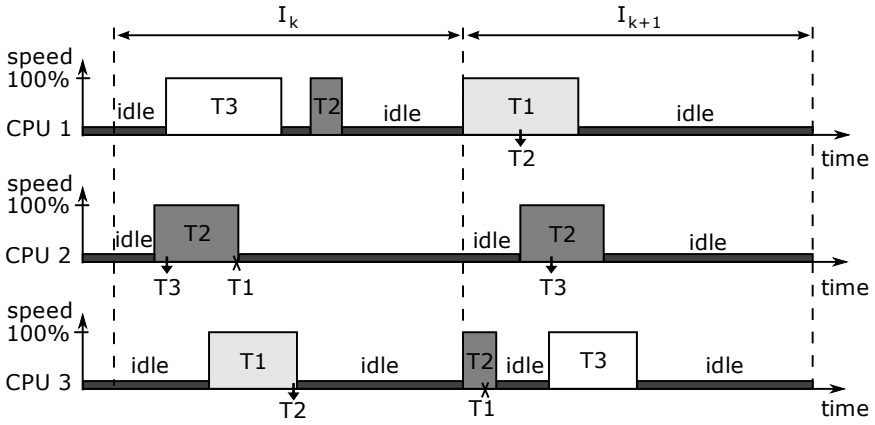
The algorithm obtains the workload of the interval by counting all non-idle cycles executed in the interval by all processors. The utilization of the SMP system U_k for an interval k is obtained using Equation (6.1), where the numerator is the number of non-idle cycles of all processors and the denominator is the total possible number of cycles of all processors (M is the number of processors, c_k is number of non-idle cycles executed by the processors during the kth interval, C_I is the number of maximum frequency cycles of the processors in one interval).

$$U_k = \frac{c_k}{M \times C_I} \tag{6.1}$$

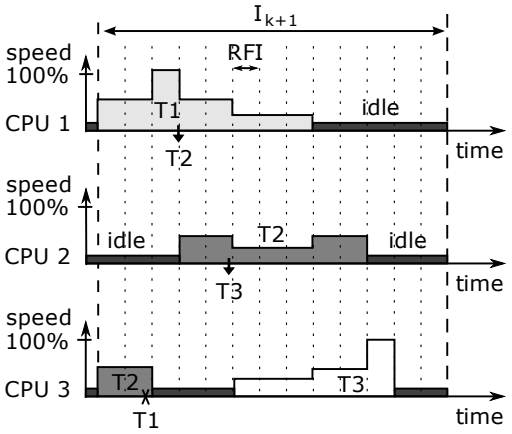
For the situation in Figure 6.5, the calculated system utilization in the interval I_k using Equation 6.1 is around 33%. Setting the frequency at 33% for the entire next interval (I_{k+1}) is not a good solution because not all processors have useful work to execute during the entire next interval. For example, idle periods are produced during the blockage of a task waiting for synchronization with another task, if there is no other task(s) ready to be scheduled during the blockage period.

Another reason is that the tasks do not have the same length and one task can not be executed in parallel by more processors. In order to overcome this problem, our algorithm divides each interval (I_k) in subintervals, called Recompute Frequency Interval (RFI), delimited by dotted lines in Figure 6.5(b). At the beginning of each RFI, the running frequency of the processors is recomputed to permit finishing useful work remaining in the interval, using only the current number of non-idle processors. This intuitively determines reduced probability to decrease application performances, by increasing frequency on the critical path. Figure 6.5(b) shows how frequency of processors increases and decreases during the interval.

To summarize, at the end of each interval, the workload of the next interval is estimated. Then, at the beginning of each subinterval, the running frequency of the processors is recomputed so that the remaining workload in the current interval can be completed using the current number of non-idle processors.



(a) Interval I_k and I_{k+1} at maximum frequency



(b) Interval I_{k+1} at a frequency optimized by our algorithm

Figure 6.5: Proposed adaptive DVFS algorithm for SMP

6.4 Implementation

This section describes the algorithm implementation. We considered that the processors can work at only a few predefined frequency values. Depending on the architecture, the idle task may or may not consume energy. Our implementation slightly differs from the conceptual algorithm described before due to practical hardware constraints. It has three parts, classified by the moment of execution and functionality criteria.

6.4.1 Interval workload estimation

The first part contains the code executed when an interval ends. It is summarized by Algorithm 6.1. The timer interrupt of one processor is programmed to occur when each interval is finished. The interrupt handler saves the current interval counters into "last interval" counters in order to be available during the next interval.

This first part also estimates the number of non-idle cycles for the next interval. If in the current interval there were fewer non-idle cycles than in the previous one, the estimated number of non-idle cycles for the next interval is expected to be the average of the two previous values. Otherwise, if the number of non-idle cycles increased, the algorithm expects it to increase during the next interval with the same amount.

Algorithm 6.1 I_{k+1} Workload estimation

```
if c_k < c_{k-1} then {current workload smaller than the previous one}
    c_{k+1} ⇐ c_{k-1} − (c_{k-1} − c_k)/2      {next workload expected between the two}
else {otherwise}
    c_{k+1} ⇐ c_k + (c_k − c_{k-1})            {next workload expected to increase more}
end if
c^{est}_{k+1} ⇐ c_{k+1} + Margin(t_I)          {keep some room for workload increasing detection}
```

Additionally, in any case, the algorithm leaves a margin which is a percentage of the interval length (*e.g.* 5 to 10 %). This margin has two main goals: first it allows the algorithm to keep up with increasing workload by detecting exceeding work and on the other hand it allows to compensate the latency of the DVFS mechanism. This second part can not be neglected with current technologies, but smarter mechanisms were proposed recently like VDD Hopping [MVR07] where frequency stabilization effects are limited. The length of this margin depends on the utilization of the system. When the system utilization is high, the length of the margin is small as the processors will work at high frequencies and the exceeding workload will be easy to detect. The length of the margin increases as the utilization decreases, because the exceeding workload becomes harder to detect.

6.4.2 RFI computation

The second part of the algorithm recomputes the frequency of the processors at the beginning of each RFI. The code of this part is described by Algorithm 6.2 and is located in the timer interrupt handler of one processor. The frequency (f_l) for the subinterval l of the current interval (I_k) is computed with Equation 6.2. It is based on the number of estimated and executed non-idle cycles of the current interval (c_k^{est} and c_k^{done}), the total number of cycles at the maximum frequency until the end of the interval given by $C_I - (l-1) \cdot C_S$ where C_I and C_S are respectively the number of such cycles in the complete interval and RFI, the number of processors (M) and the number of non-blocked tasks (N^{ready}).

$$r_l = \frac{c_k^{est} - c_k^{done}}{C_I - (l-1) \cdot C_S} \times \frac{1}{min(M, N^{ready})} \tag{6.2}$$

The selection of the frequency f_l for sub-interval S_l is made such that $f_l / f_m \geq r_l$. If the number of cycles estimated for this interval has already been executed, the frequency is set to maximum until the end of the interval because we can not know how much work is required. If the number of tasks available for execution is less than the number of processors, the number of cycles at the maximum frequency that remains in the interval is calculated using the number of available tasks instead of using the number of processors. The final frequency value is the lowest frequency accepted by the processor that is greater or equal to the frequency required for finishing the work until the end of the interval.

The length of the RFI is important. A value too big (at the limit, equal to the length of the interval) would make impossible the completion of the estimated workload, while a value too small would introduce unnecessary overhead.

The frequency is not changed at each RFI since RFI handling is made by one unique processor which makes frequency change on other processor very expensive. Frequency change is then handled by the next part of the implementation.

Algorithm 6.2 RFI Frequency computation

if $\left(c_k^{est} \leq c_k^{done}\right)$ **then**
  $f_l \Leftarrow f_m$
else
  $r_l \Leftarrow \frac{c_k^{est} - c_k^{done}}{C_I - (l-1).C_S} \times \frac{1}{min(M, N^{ready})}$
  $f_l \Leftarrow \text{select\_fq}(r_l)$
end if

6.4.3 Scheduler interaction

The third and last part actualizes the workload of the current interval and sets the processors frequency. Each time a task is unscheduled, the number of cycles of its execution is used for updating the counters of the current interval. The number of tasks available for execution is also maintained. When a task is scheduled, if the task is the idle task, the frequency of the processor it will be executed on is set to minimum, otherwise it is set to the frequency computed using the strategy presented in the previous sections.

6.4.4 Algorithm behavior and adjustments

This section analyzes the behavior of the algorithm when encountering the situations presented in subsection 6.1.2, its reaction to workload variations and its limitations. The task migration problem (Figure 6.2) will not affect this algorithm, as the workload is computed globally and the algorithm takes into account the number of available tasks.

The synchronization problem (Figure 6.3) is also bypassed, as the algorithm adapts the frequency every RFI. Specifically, for Figure 6.3, during almost all the time of the interval, there is only one task available. Thus, the algorithm will set the frequency so that both tasks can be completed using only one processor.

When the workload of an interval decreases, the idle time increases because the algorithm expects to have at least the workload of the previous interval and sets the frequency in consequence. As the end of the interval approaches, the algorithm will consider that there is still much work to do and it will increase the frequency. The energy saved is not the maximum possible for that interval, but the work is finished in time.

When the workload increases, the detected exceeding work will be executed at maximum frequency.

Unfortunately, it is possible not to detect the exceeding work. The undetected exceeding work normally reduces the idle time of the next interval, becoming, thus, detectable. Considering that the software is running on a non-RTOS, the exceeding work of one or several intervals should not be a problem. However, a large number of intervals with undetected exceeding work would make the reduction of the system performances visible to the user (*e.g.* user input taken too slow, program execution sensitively longer).

In some special cases, the exceeding work may not be detected for many intervals. To bypass this problem, we propose to have regularly intervals executed at the maximum frequency (*e.g.* one forced full frequency interval at 5 to 20 normal intervals).

Figure 6.6 presents a situation when the algorithm fails to execute the estimated workload even if the workload remains constant.

In the figure, task T_1 needs to be executed at maximum frequency during the whole interval. The other tasks executed on other processors start at the same time as the T_1 and require close values of work. Let us consider that the tasks are executed at maximum frequency during the interval I_k. In interval I_{k+1} the algorithm tries to save some energy.

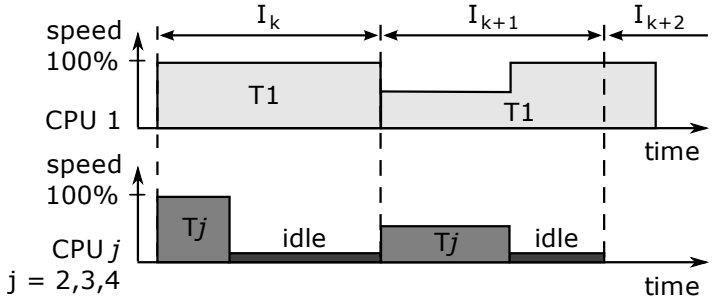


Figure 6.6: Estimated work completion problem

In this case, the required work of T_1 will not be finished until the end of the interval. This is because the algorithm extrapolates the fact that, at the beginning of the interval, all processors have a task to execute and decreases the frequency.

This problem would not occur if the tasks T_j would be situated at the end of the interval since the frequency indication computed at each RFI edge takes into account the time until the end of the interval. The frequency of T_1 at the beginning of the interval will then be set to the maximum and at the arrival of other tasks the frequency would stay at the maximum because of them.

Figure 6.6 represents the worst case of the problem. We will compute the maximum workload that can remain uncompleted. The estimated workload for I_{k+1} at the end of the interval I_k is given by Equation 6.3. The notations are those already presented.

At the beginning of the interval I_{k+1} all processors have a task to execute. The running frequency of the processors for the period while all processors execute a task is constant. Its value, f_{T_j}, is given by Equation 6.4. If the selected frequency for this period is not the maximum frequency, a part of the required workload cannot be completed.

$$c_{k+1}^{est} = (M-1) \times c_{j,k} + C_I + \text{Margin}(t_I) \times C_I \times M \tag{6.3}$$

$$f_{T_j} = \text{select\_fq}\left(\frac{c_{k+1}^{est}}{M \times C_I}\right) = \text{select\_fq}\left(\frac{(M-1) \times c_{j,k}/C_I + 1 + \text{Margin}(t_I) \times M}{M}\right) \tag{6.4}$$

The uncompleted workload for the interval I_{k+1} is presented in Equation 6.5. As we can see, the uncompleted workload depends only on the ratio between $c_{j,k}$ and C_I, not on their individual values. The workload of the T_j tasks ($c_{j,k}$), for which is obtained the maximum uncompleted workload can be obtained by deriving the c_{k+1}^{undone} (Equations 6.6 and 6.7, where $c_{j,k}^{Max}$ is the maximum uncompleted workload). For a system containing 4 processors accepting any frequency between the maximum and minimum values, the maximum uncompleted workload represents 27 % of the task T_1 workload. For 2 processors, this percentage is 13 % and for 20 processors, it is 49 %. This percentage has a maximum of 60 % obtained for more than 1500 processors.

$$c_{k+1}^{undone} = c_{j,k} \times \left(\frac{f_m}{f_{T_j}} - 1\right) \approx c_{j,k} \times \left(\frac{M \times C_I}{c_{k+1}^{est}} - 1\right) \tag{6.5}$$

$$\frac{d}{d\ c_{j,k}} c_{k+1}^{undone}(c_{j,k}) = 0 \tag{6.6}$$

$$c_{j,k}^{Max} = C_I \times \left(\frac{\sqrt{M + M^2 \times \text{Margin}(t_I)} - 1 - \text{Margin}(t_I) \times M}{M - 1} \right) \tag{6.7}$$

Now that the maximum uncompleted workload have been computed, we show how this situation can be detected and treated. For the detection part, it is not enough to detect if a task was executed during the entire previous interval, as the repartition of the workload on the processors does not have to be exactly the one in the figure. A group of tasks that synchronize between them can produce an almost identical workload as a task from the figure. Figure 6.7 presents an example that reproduces the presented problem. The task T_1 is not executed on the same processor all the time. At t_1 it is preempted and it is scheduled to another processor. At t_2, it unblocks the task T_6 and waits for a response. The task T_6 is executed on another processor. The tasks T_1 and T_6 from Figure 6.7 produce together almost the same workload as task T_1 from Figure 6.6. Thus, computing the execution time of each task is not a solution. A synchronization graph of the tasks could be kept. The management of this graph and taking frequency decisions based on it would introduce a significant overhead. However, it is not easy to use the synchronization graph for setting the frequencies as a task may synchronize with another task in several points of its execution. Also, a synchronization may produce different tasks scheduling order, depending on the availability of the processors.

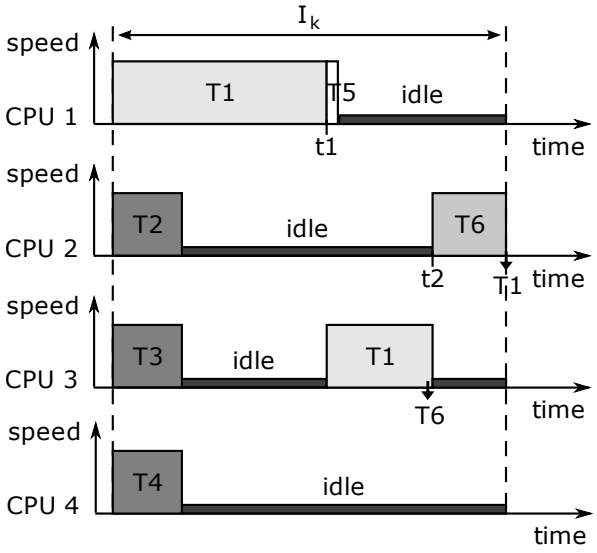


Figure 6.7: Workload similar to Figure 6.6

For detecting this problem, we keep track of the time in which all processors are idle in the scheduler interaction part of the algorithm. We expect this idle time to be a percentage (*e.g.* 5 %) from the entire interval length. If it is not, the number of cycles estimated for the next interval is increased with the number of cycles corresponding to the missing idle time multiplied by a factor that depends on the number of processors in the system. The added cycles should determine the frequency to increase in a few cycles to the maximum value.

6.5 Experimental results

The energy saving algorithm was tested through the Motion JPEG decoding application, which allows us to check that algorithm does not alter performances and thus frame timelines. The entire software stack, composed of the Motion JPEG decoding application and the Mutek operating system, is cross-compiled for the target processor and simulated using the SoCLib components based cycle accurate simulation platform. The simulation platform contains 3 Sparc V8 processors.

Figure 6.8 shows the system frequency evolution over 7 frames with the adaptive DVFS algorithm during the application execution. During the first frame simulation, the frequency is set to maximum for useful cycles and to minimum for the idle ones. The average frequency oscillates around the maximum value (117 MHz) for the decoding period (zone (a) in Figure 6.8). The idle time during the frame decoding generates an oscillating average, not a constant value. Between the end of decoding of one frame and the beginning of decoding of the next frame (zone (b)), all processors are idle, so the average frequency is constant (minimum, 7 MHz).

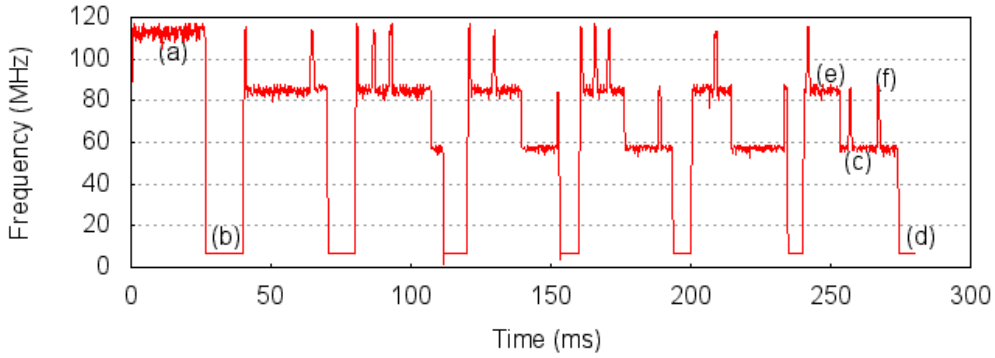


Figure 6.8: System frequency evolution during Motion-JPEG execution

Then the adaptive DVFS algorithm for SMP tries to stretch progressively the useful work over the entire frame period (Figure 6.8). As the system utilization is about 51%, the frequency during the frame decoding is most of the time halved (zone (c) in Figure 6.8). The time reserved for exceeding work detection determines at the beginning of presented frame (zone (e)) a greater system utilization. Until the system utilization falls under 50%, the frequency is set to the next upper step, $3/4 * f_m$. A pin zone (like (f)) may appear if the number of non-idle processors increases just after the frequency recomputation part of the algorithm. The oscillating average during the frame decoding is caused by the uneliminated idle time. The idle period between the frames (zone (d)) is much shorter compared to first frame (without adaptive algorithm) case. It represents the time reserved for exceeding work detection.

In the following, we describe the methodology used for estimating the energy saved by the algorithm and apply it to the case study. Table 6.1 shows the frequency and the voltage operation limits of the chosen processor.

Considering that the frequency is linearly dependent on the voltage [MFMB02], the voltage (V_i) required by a custom frequency (f_i) is given by Equation 6.8. The dynamic power of a processor is given by Equation 6.9, where α is the transition activity, C is the switch capacitance, f is the frequency and V is the voltage. Using Equation 6.8 and 6.9, one can calculate the power at a custom frequency relative to the full power (Equation 6.10).

Table 6.1: Processor characteristics

	f_1(min)	f_2 (1/4)	f_3 (1/2)	f_4 (3/4)	f_5 (max)
Frequency (MHz)	7	29.25	58.5	87.75	117
Voltage (V)	0.8	0.86	0.94	1.02	1.1
$\frac{P_{f_i}}{P_{f_m}}$	0.03	0.15	0.36	0.64	1

The operating frequencies of the processors are listed in Table 6.1, scaling from 7 to 117 MHz. The minimum frequency (7 MHz) is only used when a processor has no task to execute (idle). By using Equation 6.10 and first two lines of Table 6.1, we can obtain power dissipation ratio against the maximum speed dissipation. The power dissipation ratio is listed in the last line of Table 6.1.

$$V_i = V_1 + \frac{f_i - f_1}{f_m - f_1} \times (V_m - V_1) \tag{6.8}$$

$$P \approx \alpha \times C \times V^2 \times f \tag{6.9}$$

$$\frac{P_{f_i}}{P_{f_m}} = \frac{\alpha \times C \times V_i^2 \times f_i}{\alpha \times C \times V_m^2 \times F_m} \tag{6.10}$$

$$= \frac{f_i}{f_m} \times \left(\frac{V_i}{V_m}\right)^2 \tag{6.11}$$

All log information is traced at hardware level by our SoCLib components. This assures a very high accuracy and does not introduce any overhead into the software. The simulated video has 25 frames/second (a frame at each 40 ms). The average time spent by each processor at each frequency during one frame of the simulated video is depicted in Figure 6.9. Because the system utilization is about 51%, the most used frequency is $f_m/2$. Anyway, because of synchronizations, there are moments when not all processors have a task to execute. For those periods, upper frequencies (f_m and $f_m * 3/4$) are used. The frequency $f_m/4$ is never used for this video because the estimated system utilization never drops under 25%.

The following paragraph explains how we compute the percentage of saved energy. We considered that the processors consume the same amount of energy in each cycle, independently of the instructions. For a given frequency and for its required voltage, the power in Equation 6.9 is constant. The energy consumed by a processor running at a constant frequency (f_i), with the required voltage (V_i) and for a period of time (t) is shown in Equation 6.12. Ignoring the changing frequency delays, a processor supporting m frequencies and running for a period of time T_i at frequency f_i ($i \in [1, m]$) consumes the energy depicted by Equation 6.13. The energy saved (E_s) by using these m frequencies relative to the energy that would be consumed if the entire period would be executed at the maximum frequency is shown in Equation 6.14.

$$E_{f_i} = \int_0^t P_{f_i} dt = t \times P_{f_i} \tag{6.12}$$

$$E = \sum_{i=1}^{m} t_i \times P_{f_i} \tag{6.13}$$

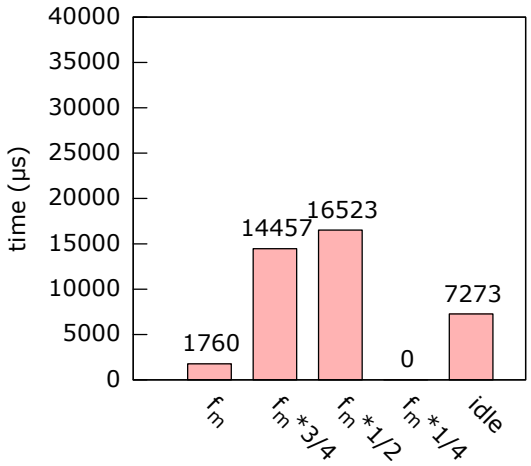


Figure 6.9: Average time spent at each supported frequency

$$E_s = \left(1 - \frac{\sum_{i=1}^{m} t_i \times P_{f_i}}{P_{f_m} \times \sum_{i=1}^{m} T_i}\right) \times 100 \tag{6.14}$$

Using Equation 6.14, the values for the time spent at each frequency presented in Figure 6.9 and the P_{f_i} previously computed with Equation 6.10, the energy saved by the algorithm is computed for our application. For the average time spent at each frequency presented in Figure 6.9, the saved energy is computed in Equation 6.15.

$$E_s = 1 - \frac{P_{f_m} \times (1 \times 1760 + 0.64 \times 14457 + 0.36 \times 16523 + 0.15 \times 0 + 0.03 \times 7273)}{P_{f_m} \times (1760 + 14457 + 16523 + 0 + 7273)} = 57.07\% \tag{6.15}$$

Figure 6.10 depicts the influence of the workload sampling interval on the performance of the algorithm, namely the energy saving achieved. The three curves represents three different RFI length: 1 ms, 5 ms and 10 ms. The processor average energy saved using the adaptive DVFS algorithm is about 45% while the overhead in term of computation power is limited, only 3% in this example. As can be realized from the figure, the energy savings of the algorithm can grow up to 55%, at specific values where the intervals I_k are synchronous with the application periodicity, namely $t_I = 40\ ms$.

Figure 6.11 presents the average deadline miss time per frame for different lengths of the workload sampling interval and the RFI. The value of the average deadline miss time is computed using formula: $(t_{exec} - T_{frm} * N_{frm})/N_{frm}$, where t_{exec} is the execution time of the N_{frm} frames having the period T_{frm}. The deadline misses are caused by the overflow of the useful work produced by our energy saving algorithm and by the imprecision of the operating system *nsleep* (nano sleep) function. This function is used for waiting for the end of the current frame period after the decoding of the next frame is finished. For almost all tests presented in the figure, the deadline miss time is about 16 μs, representing 0.03 % from the frame period, and is mainly produced by the nsleep function. For the test in which the workload sampling interval is 10 ms and RFI is 5 ms, the average deadline miss time is 1118 μs (2.79 % from the frame period) and is mainly caused by the energy saving algorithm.

Even if the workload of the Motion JPEG decoding application does not vary very much

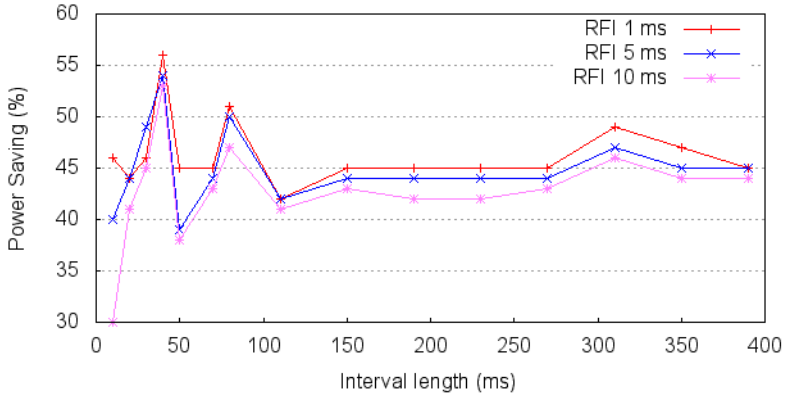


Figure 6.10: Energy savings versus Interval and RFI lengths

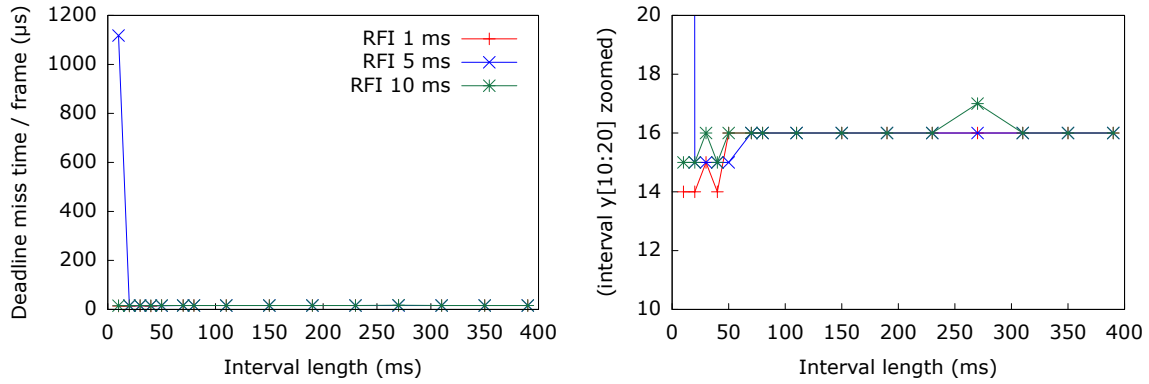


Figure 6.11: Time deadline miss

from a frame to another, the workload sampling intervals, whose lengths are not multiple of the frame period, have sensibly different workloads. In all tested cases, the proposed algorithm does not reduce significantly the system performances.

For this case study, the technique that executes the useful work at the maximum speed and the idle time at the minimum one saves about 35 % of energy. Comparing to this simple technique, our algorithm saves with 3 % to 21 % more energy.

6.6 Conclusion

This chapter presented a energy saving algorithm for SMP architectures running a preemptive non-RTOS. It addresses the SMP specific features by a global management of the system workload and a frequency setting adapted to the runtime system computation capabilities.

The complete lack of information from the applications makes very difficult the detection of the application computation requirements. Unlike the energy saving algorithms running on RTOS, which have all information required for avoiding the deadline missing of the tasks, the energy saving algorithms running on non-RTOS have no clue whether the execution of the tasks was too stretched and the system performances diminished too much, becoming unusable and annoying to the user.

The proposed algorithm tries to save energy without major degradations of the system performances. Because of this, the saved energy is often not the maximum possible. Even so, sometimes the algorithm does not detect the exceeding work. However, given the low amount of information it has, the algorithm behaves fairly well in practice on use cases, saving an important percent of energy without reducing too much the system performances.

Chapter 7

Conclusion and future works

The overall objective of this thesis was to propose a DVFS based energy saving algorithm for SMP architectures and some fast and accurate MPSoC simulation platforms required for its validation and simulation.

We investigated the possibility of modeling such simulation strategies at different levels of abstraction.

For improving the simulation speed at the transaction accurate abstraction level, we replaced the interpretative ISSes with binary translation based ISSes. In the following, we present our answers to the related questions posed at the end of the problem definition chapter.

- **Question:** How to combine the accuracy of the transaction accurate simulation models with the speed of the untimed virtualization approaches?

  Answer: For an accurate timing behavior, we modeled the processor timing behavior, created fast and precise cache models and solved the synchronization issues due to the different models of computation used in the binary translation based ISS and in the rest of the system. For modeling the time internally required by the the simulated processors for the instructions execution, we annotated the translated code using the number of cycles corresponding to the simulated instructions. A processor is simulated while it does not communicate with the world behind its caches. When an instruction/data cache miss or an I/O occurs, the processor synchronizes itself with the rest of the hardware components simulated concurrently by the event driven simulator.

- **Question:** Is it possible to reduce the number of transactions over the interconnect for trading-off accuracy for speed?

  Answer: For a speed versus accuracy trade-off, we proposed several configurations for the simulation strategy. At an extremity, the most accurate but also the slowest configuration fully implements the caches. A cache miss in this configuration determines a request over the interconnect to access the data in the memory component. In the opposite, the fastest but also the less accurate configuration does not implement data and instruction caches. The processor synchronizes with the rest of the system by consuming the time corresponding to the simulated instructions when an I/O operation is simulated or after a predefined number of cycles simulated. Between these two extremes, there are other two configurations that implements the caches only from the hit/miss point of view. A predefined time is consumed when a cache miss

occurs. The experiments show that even for the most precise configuration, the simulation speedup is still significant (about 18). For the configuration without caches, the speedup is about 380.

Our answers to the questions related to the use of static scheduling for the cycle accurate abstraction level simulations are:

- **Question:** How to statically schedule the processes of the architectures having components working at different frequencies?

  Answer: We proposed two static scheduling strategies that can simulate such architectures. For the first one, each frequency signal is generated by an actual clock component. A scheduling pattern having the period equal to the least common multiple of the clocks periods is created. The edges of clocks succeed normally during the scheduling pattern period. The simulation consists of executing the processes sensitive to the first event in the scheduling pattern, then those sensitive to the second event *etc*. The second approach uses a single clock, whose frequency is equal to the LCM of all possible frequencies of all components in the architecture. The frequencies required by hardware components are obtained by dividing this frequency. The simulation consist of executing the processes sensitive to the global clock. When one of these processes modifies an output signal which represents the clock signal for other components, the processes sensitive to that clock signal are executed. No event is generated when the value of a signal is changed. The conditions of running the processes sensitive to generated clock signals are verified by the simulator after each cycle of the global clock. These strategies give a speedup of 3.5 compared to the usual event driven simulators.

- **Question:** How to support the runtime change of these frequencies without reducing the simulation accuracy?

  Answer: For supporting runtime change of the frequencies in the first simulator, an API has been added to the clock component. This API function offers a mean of modifying the frequencies of the clocks. Whenever the frequency of a clock component changes, the proposed simulator computes a new scheduling pattern. The second simulator does not have to do anything special for supporting runtime changes of the frequencies. The components that generate a clock signal compute the new divisor for obtaining the required frequency from the global frequency. To the best of our knowledge, no such strategies have been devised before.

Regarding the energy saving at the software level:

- **Question:** How to build an energy saving algorithm running on a non-RTOS and dealing with the particularities of the SMP architectures?

  Answer: We proposed an energy saving algorithm that divides the execution time in equal time intervals and expands the workload of each interval over the entire next interval by reducing the processors frequency. The algorithm addresses the SMP specific features (tasks migration between processors, tasks communication and synchronization) by a global management of the system workload and a supplementary order interval subdivision. At the end of each interval, the workload of the next interval is estimated. At the beginning of each subinterval, the running frequency of the processors is recomputed so that the remaining workload in the current interval can be completed. Thanks to the fast changing frequency and AC/DC converter, this

strategy is proved efficient and, although leading sometimes to response time longer than expected, leads to 45% energy savings in average.

Short term future works. For the binary translation based ISSes, the short term future work consists of integrating these ISS models with cycle-accurate hardware models of the SoCLib library. We also plan in the near future to create a binary translation based ISS model for VLIW processors.

The short term future work for the cycle accurate static scheduling simulator based on multiple clocks consists of improving the simulation speed when the frequencies change often by memorizing the schedules already computed and creating an equivalence scheme between the schedules.

The short term future work for the energy saving algorithm will be focused on applying this algorithm to Linux OS.

Long term future works. The long term future work for the binary translation based ISSes consists of parallelizing the simulation in order to accelerate it, thus taking advantage of the multi-core architectures. The challenge is to minimize the synchronization between the host threads executing different parts of the simulation and to balance the simulation code executed by these threads.

Improving the exceeding work detection represents the long term future work for the adaptive DVFS algorithm.

Chapter 8

Résumé en français (French summary)

Contents

8.1 Introduction

La miniaturisation des transistors sur le silicium joue un rôle très important dans l'industrie de la microélectronique, permettant aux systèmes intégrés de devenir de plus en plus complexe. Des systèmes complets intégrant des processeurs, mémoires et autres périphériques sur une seule puce, appelés systèmes sur puce (SoC), existent de nos jours grâce à la miniaturisation.

Un nombre croissant de systèmes sur puce (ordinateurs portables, téléphones cellulaires, PDA, appareils de l'aéronautique et l'espace) est utilisé à la fois dans la vie quotidienne et dans les missions critiques. Les nouvelles applications de décodage/encodage à haute résolution et de compression haute vidéo/audio et les jeux vidéo 3D exécutées sur ces appareils nécessitent une puissance de calcul de plus en plus élevés (Figure 8.1). Pour couvrir l'écart entre les performances de calcul nécessaires et disponibles, en plus d'améliorer les performances des dispositifs, le nombre d'unités de calcul (PE) devrait augmenter [ITR07a].

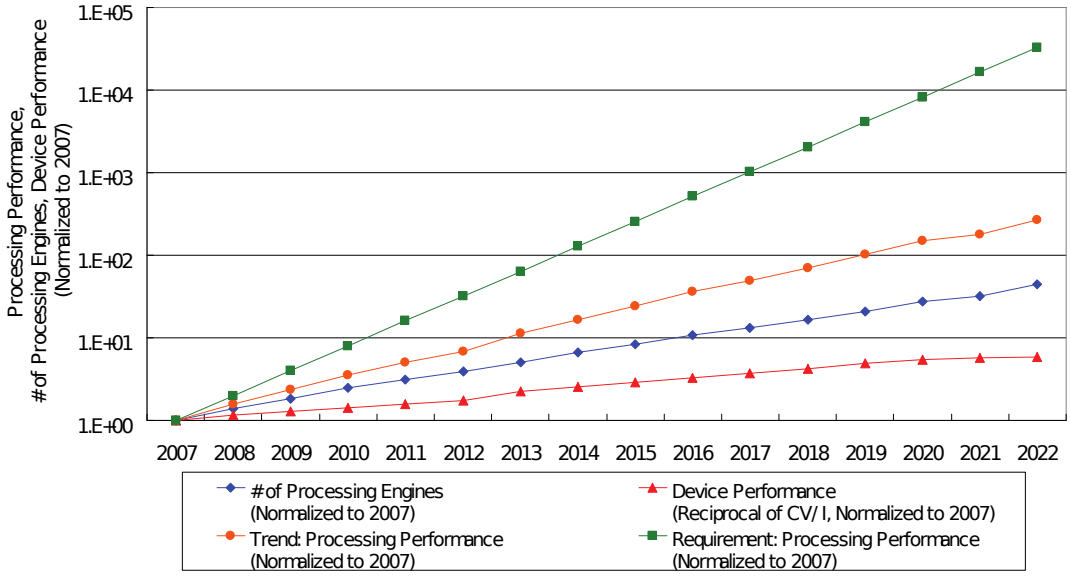


Figure 8.1: L'évolution des performances des SoC dédiés aux applications grand public portable (ITRS 2007)

L'ITRS (International Technology Roadmap for Semiconductors) s'attend à ce que la complexité moyenne des circuits, tels que les PE et les périphériques, reste constante, tandis que le nombre de PE devrait augmenter rapidement dans les années à venir. La quantité de mémoire principale est supposée augmenter proportionnellement avec le nombre de PEs [ITR07a]. Le nombre d'unités de calcule dans les SoCs attendu par l'ITRS pour les prochaines années est présentée dans la Figure 8.2.

Les processeurs du système multiprocesseurs sur puce (MPSoC) peuvent être organisés de manière symétrique ou asymétrique. Tous les processeurs d'une architecture symétrique (SMP) sont identiques et sont reliées à une mémoire partagée logiquement unique. Il y a un seul système d'exploitation (OS) exécuté par tous les processeurs, qui exécute les tâches prêtes sur n'importe quel processeur libre. Les architectures SMP fournissent une compromis intéressant entre puissance, performance et flexibilité, ce qui permet d'obtenir les performances d'un processeur fonctionnant à une fréquence très élevée en utilisant plusieurs processeurs travaillant à des fréquences plus basses, consommant ainsi moins d'énergie à condition que la demande puisse être parallélisée comme un ensemble de tâches. Les architectures asymétriques (ASMP) contiennent généralement des processeurs généralistes (GPP) et beaucoup plus de processeurs spécialisés (DSP). Le GPP attribue et planifie des tâches au DSP, qui sont des processeurs spécialisés avec une puissance de traitement élevée. Grâce aux processeurs spécialisés, les architectures ASMP ont de meilleures performances et une faible consommation d'énergie par rapport aux architectures SMP.

Pour être rentable, un produit doit être compétitif et son coût de production doit être faible. L'ITRS prévoit que le coût des masques va augmenter d'un facteur 23 dans les dix prochaines années [ITR07b]. Pour réduire le coût total des futures MPSoCs, les processeurs spécialisés (DSP) seront remplacés par ceux généralistes (GPP) et leurs fonctionnalités seront prises en charge par le logiciel. Le remplacement de processeurs spécialisés par le logiciel a un impact sur les performances du système, mais il offre une grande flexibilité et permet de faire face à des éléments de traitement non fonctionnel. Les futures architectures MPSoC contenant un si grand nombre de processeurs à usage général

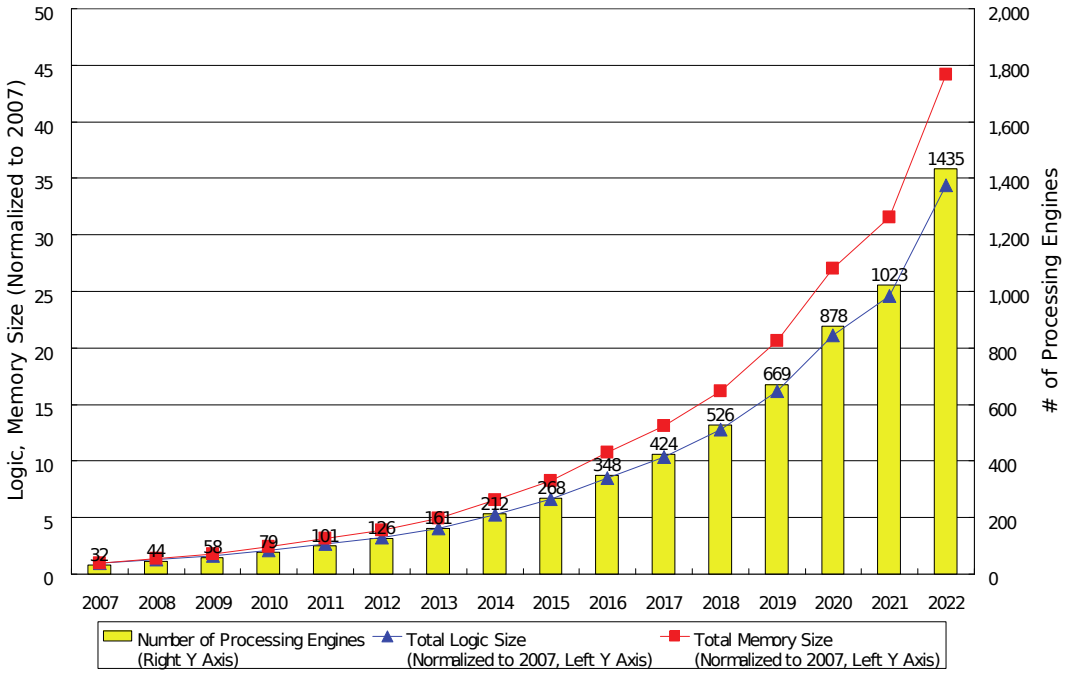


Figure 8.2: SoC dédiés aux applications grand public portables (ITRS 2007)

devraient être de plus en plus homogène et parallèle et utiliser une mémoire partagée, comme l'architecture présentée dans la Figure 8.3.

L'homogénéité et la souplesse de l'architecture assure des temps de conception et de vérification courts et permet la réutilisation de la même plate-forme non seulement pour les versions suivantes de la même application, mais aussi pour d'autres applications. Ces fonctionnalités améliorent le temps de développement conduisant à un temps de mise sur le marché plus courts, qui permet au produit d'avoir du succès. La Figure 8.4 présente l'écart entre les capacités de la technologie qui double tous les 36 mois (la loi de Moore) et la productivité de conception du matériel/logiciel. Outre les plates-formes de conception réutilisables et les architectures matérielles flexibles, une conception au niveau système est nécessaire pour combler le déficit présenté.

La complexité croissante des MPSoCs détermine également la croissance de leur consommation d'énergie. La technologie des batteries est également en développement, mais sa progression est plus lente que la croissance d'intégration de la technologie. Presque tous les appareils portables (téléphones portables, caméras vidéo, etc.) s'appuient sur des batteries pour l'alimentation. La vie de la batterie influe directement sur leur taille, poids et temps de fonctionnement. Même pour les dispositifs alimentés par secteur, abaisser la consommation d'énergie permet également de réduire la chaleur dissipée, ce qui diminue les coûts d'emballage et de solution de refroidissement pour les circuits intégrés. Maîtriser les besoins énergétiques croissants de ces dispositifs constitue l'un des plus grands défis dans le domaine des MPSoC.

Les économies d'énergie peuvent être réalisées à la fois au niveau du logiciel et du matériel. Du point de vue matériel, les composants matériels peuvent être optimisés en énergie (réduction de la surface de la puce, en gardant les fils à hautes activités locales et courts, *etc.* [Hav00]). De l'énergie peut également être économisée en utilisant des composants qui acceptent plusieurs modes de fonctionnement. Des processeurs qui acceptent

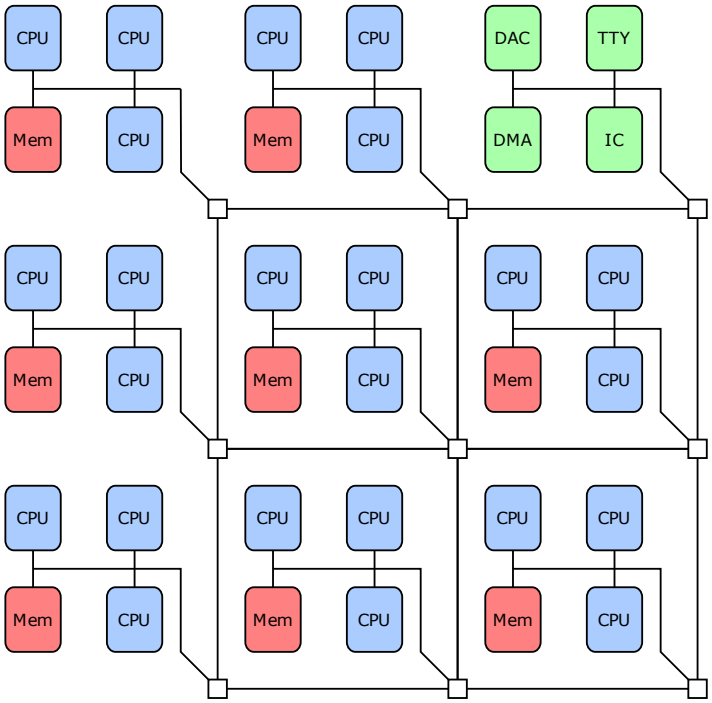


Figure 8.3: Architecture des SoC future

les variations de la fréquence et de la tension, des écrans avec de nombreux niveaux de luminosité et d'autres composants matériels avec des capacités d'économie d'énergie sont maintenant disponibles pour les MPSoCs.

La première contribution de cette thèse consiste en un algorithme logiciel qui cherche à économiser de l'énergie en modifiant la fréquence et la tension des processeurs lorsque l'utilisation du système le permet. L'algorithme s'appuie sur les informations qui concernent le temps d'exécution des tâches et ne nécessite pas d'information sur les applications.

Pour valider l'architecture et pour pouvoir commencer le développement logiciel le plus tôt possible, et donc réduire le temps complet de conception du système, la simulation du système est largement reconnue comme nécessaire. Puisque les architectures MPSoC sont trop complexes pour être décrites de façon analytique, la simulation représente la seule solution pour leur validation. La simulation représente également la meilleure solution pour optimiser les performances et la consommation MPSoCs par l'exploration d'architecture. En raison du fait qu'elle est très coûteuse et prend beaucoup de temps, il est en pratique impossible de physiquement construire et de tester toutes les architectures candidates.

L'architecture peut être simulée à différents niveaux d'abstraction. Le niveau transistor, le niveau porte et le niveau RTL (niveau de transfert registre) de simulations sont utilisés exclusivement pour la conception matérielle. Ces niveaux de simulation aident à la conception du circuit de plus proche de la perfection que possible avant que le circuit intégré soit initialement fondu, ce qui est essentiel, en raison des coûts élevés de masques photo-lithographique et de conditions des fabrication.

Le VHDL et le Verilog sont deux langages de programmation qui permettent la description des systèmes tels que les FPGA (Field-Programmable Gate Array) et les circuits intégrés en utilisant un langage de description matériel (HDL). Bien qu'il soit possible d'exécuter des logiciels utilisant des processeurs et d'autres composants matériels modélisés avec un de ces langage, en raison d'une vitesse de simulation très faible, le logiciel

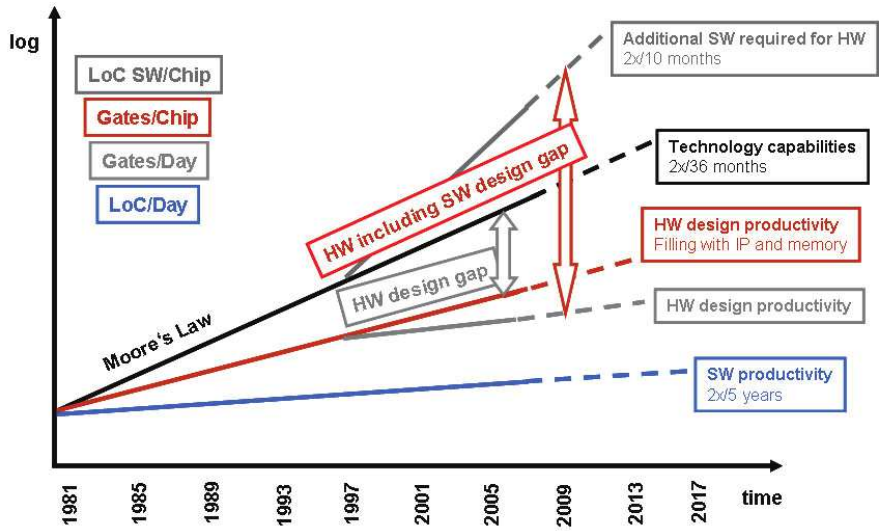


Figure 8.4: Les lacunes de conception du matériel/logiciel en fonction du temps (ITRS 2007)

exécuté ne valide habituellement que le comportement du matériel.

L'utilisation de langages de programmation logicielle tels que C/C++ pour la description du matériel permet de se débarrasser de tous les détails du matériel d'importance mineure d'un point de vue temporelle et comportementale, ce qui augmente la vitesse de simulation. Cela permet l'exécution de la totalité de la pile logicielle utilisant le modèle de matériel dans un délai acceptable. A titre d'exemple, SystemC est une bibliothèque C++ qui offre le support pour la modélisation et la simulation de systèmes électroniques à différents niveaux d'abstraction.

Afin de tester et de voir l'efficacité de l'algorithme d'économie d'énergie proposé et aussi de choisir l'architecture la plus appropriée pour une application spécifique, nous avons besoin de stratégies de simulation rapide et précise qui support le changement de fréquence individuellement pour chaque processeur.

Les autres contributions de cette thèse sont liées à des simulations MPSoC visant l'estimation de la puissance à différents niveaux d'abstraction. Le premier simulateur combine la précision de simulateurs axés sur le matériel avec la vitesse de simulateurs axés sur le comportement. Deux autres simulateurs consistent en des adaptations d'un simulateur utilisant un ordonnancement statique pour être capable de simuler des architectures avec des fréquences multiples.

Le reste de ce chapitre présente ces trois contributions. Le chapitre se termine par les conclusions et les travaux futurs.

8.2 Utilisation de l'ISS basé sur la traduction binaire dans la simulation dirigée par événements

La première stratégie de simulation proposée tente de combiner la rapidité des ISSes basés sur la traduction binaire avec la précision des simulateurs dirigés par les événements. Pour avoir un comportement temporel précis pour les ISSes basés sur la traduction binaire, il fallait tout d'abord résoudre les problèmes de synchronisation dans la modélisation du

processeur, d'autre part définir des modèles de cache rapide et précis, et enfin de résoudre les problèmes de synchronisation dûs aux différents modèles de calcul utilisés dans les ISSes et dans le reste du système. Nous présentons une solution d'intégration qui couvre ces questions et décrivons sa mise en œuvre.

Le simulateur est présenté des points de vue multiprocesseurs et temporelle. Même si notre approche est applicable à tous les ISSes basés sur la traduction binaire, nous utilisons l'ISS de QEMU et SystemC TLM en tant qu'environnement de simulation pour la plate-forme entière.

8.2.1 Modélisation multiprocesseur

Nous allons maintenant présenter notre approche de simulation basée sur la technologie de traduction binaire.

L'objectif des simulateurs basés sur la traduction binaire est d'être très rapides et fonctionnellement corrects, mais pas de fournir des informations sur le temps d'exécution matériel. Par conséquent, la mise en œuvre de plates-formes multiprocesseurs appelle simplement la fonction d'interprétation de chaque processeur l'une après l'autre dans un ordre prédéfini. Le retour d'une fonction d'interprétation de processeur, que nous appelons le retour de fonction de l'ISS, arrive quand une interruption ou une exception se produit, mais seulement à la frontière d'un bloc de traduction et dans tout les cas n'accumule pas du temps. L'ordre de simulation des processeurs est toujours le même. Comme notre objectif est d'estimer le temps et agir sur alimentation à l'aide du DVFS, nous avons besoin d'un moyen pour contrôler (et mesurer) l'ordre de l'exécution des processeurs. Pour ce faire, le comportement par défaut des simulateurs initiaux doit être modifié, au prix de la réduction de la vitesse de simulation.

8.2.1.1 L'adaptation et la connexion de l'ISS

Pour chaque processeur dans la plate-forme, un module d'adaptation SystemC est instancié (*iss_wrapper*), comme la Figure 8.5 le montre. L'exécution de chaque processeur est effectuée dans le contexte du processus SystemC (*SC_THREAD*) de son adaptateur. De cette façon, les processeurs sont simulés concurremment. Il est nécessaire que chaque processeur soit simulé dans le contexte de son propre processus SystemC, de sorte qu'ils soient simulés de manière indépendante sous le contrôle de SystemC.

En outre, l'adaptateur de chaque processeur met en oeuvre un module DVFS (adaptation dynamique de la tension et de la fréquence) permettant à chaque processeur simulé de travailler à des fréquences différentes.

Afin d'éviter la traduction binaire du même code cible pour chaque processeur simulé, les processeurs pouvant partager le même cache de traduction sont encapsulés dans un module SystemC appelé *iss_group*. Pour assurer que le code hôte généré pour un bloc de traduction est correct pour tous les processeurs du groupe, les processeurs de l'*iss_group* doivent être identiques.

Les adaptateurs de processeurs sont reliés à une interconnexion par laquelle ils peuvent communiquer avec les autres composants matériels (générateur de trafic, timers, contrôleur d'interruption, mémoire, spinlocks *etc.*) également connectés à l'interconnexion. Tous les composants matériels sont implémentés comme modules SystemC. Les lignes d'interruption des différents composants (par exemple, des timers) sont connectées aux adaptateurs de processeur, qui mettent en oeuvre l'interface entre les fils d'interruptions SystemC et ceux de l'ISS.

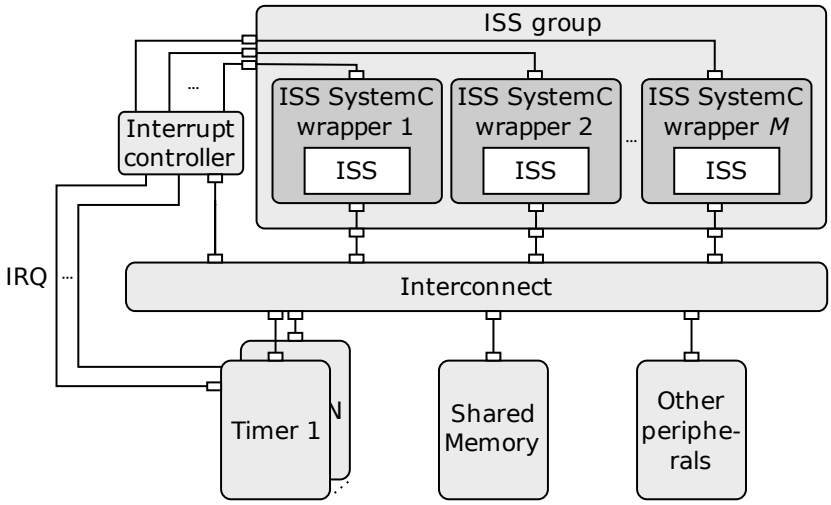


Figure 8.5: Plateforme de simulation utilisant traduction binaire et SystemC

Du simulateur initial, notre approche utilise uniquement les modèles de processeurs avec, si nécessaire, leur MMUs (unité de gestion mémoire). Tous les autres composants sont extériorisés et implémentés sous forme de modules SystemC pour des raisons de précision. La mémoire principale est également implémentée sous forme d'un ou plusieurs modules SystemC. Pour accéder aux modules SystemC, autres que la mémoire principale, certaines portions des adresses physiques sont considérées comme des adresses I/O par l'ISS. Les demandes d'I/O des processeurs simulées sont transformées par les adaptateurs de processeur dans les demandes SystemC, utilisant le protocole compris par l'interconnexion.

8.2.1.2 Points de synchronisation avec SystemC

Dans notre modèle, un processeur est simulé alors qu'il ne communique pas avec le monde derrière ses caches et le simulateur initial ne l'arrête pas. Quand un défaut de cache d'instruction/données ou un opération d'I/O se produit, la simulation du processeur s'arrête et l'adaptateur du processeur se synchronise avec le reste de la plate-forme SystemC en consommant le temps (fonction wait de SystemC) requis par le processeur réel pour exécuter les instructions simulées depuis la dernière synchronisation.

Donc, contrairement aux simulateurs précis au cycle où les cycles des ISSes sont simulés concurremment, l'ISS basé sur la traduction binaire proposé simule un groupe de cycles avant de se synchroniser avec SystemC et de permettre l'exécution d'autres ISSes. Le nombre de cycles dans ces groupes peut varier de un à plusieurs cycles, selon le code cible simulé. La simulation de ces cycles prennent un temps SystemC égal à zéro.

La synchronisation a lieu également après un retour normal de la fonction de l'ISS et après une période de temps prédéfinie sans synchronisation. Pour les instructions du processeur cible conçu pour la synchronisation des logiciels tournant sur une architecture SMP, une synchronisation SystemC devrait également être générée (par exemple les instructions *load* et *store* exclusive).

L'ordre de simulation des processeurs dépend du temps consommé par les processeurs aux points de synchronisation. Une condition de synchronisation peut se produire à tout moment pendant la simulation d'un bloc de traduction (par exemple défaut de cache), donc le désordonnancement ne respecte plus la frontière des blocs de traduction. Comme la fonction de l'ISS initial revient seulement à la frontière des blocs de traduction, nous

sauvons le "contexte" de la simulation avant la synchronisation et le restaurons par la suite, pour réduire l'impact autant que possible sur le code existant.

8.2.1.2.1 Synchronisation avec SystemC après de longs intervalles sans synchronisation

En raison du fait que les processeurs simulés ne se synchronisent pas à chaque accès à la mémoire, si deux ou plusieurs processeurs simulés lisent/écrivent de/à la même adresse mémoire, les instructions exécutées par ces processeurs peuvent différer de celles exécutées sur une plate-forme de cycle précis. Une écriture à une adresse devrait invalider sa ligne de cache correspondant dans tous les caches du reste des processeurs et ces processeurs devraient voir la nouvelle valeur à la lecture suivant de cette adresse. Dans notre stratégie de simulation, si le processeur écrivain est simulé avant le processeur lecteur, l'écriture est visible par le reste des processeurs avant qu'elle se passe dans la ligne du temps simulé. Si l'ordre de simulation des processeurs est inversé, le processeur lecteur ne voit pas l'effet de l'écriture jusqu'à ce qu'il se synchronise et le processeur écrivain exécute le code d'écriture.

Désordonnancer le processeur après un temps prédéfini sans synchronisation est nécessaire pour les cas où un processeur attend en boucle le changement par un autre processeur ou même par un gestionnaire d'interruption sur le même processeur d'une variable simple. Ce genre de boucles empêcherait les interruptions de se produire pour ce processeur parce que le fanion d'interruption est activé pendant la synchronisation SystemC. Par exemple, pour le calcul de la vitesse du processeur, Linux attend en boucle l'incrémentation de la variable *jiffies* par le gestionnaire d'interruption du timer (Listing 8.1). La condition de désordonnancement en raison du manque de synchronisation est vérifiée au début de chaque bloc de traduction. Le délai pour cette condition de désordonnancement détermine le délai maximum des interruptions. Nous avons arbitrairement fixé ce paramètre 250 fois inférieure à la période de timer.

Listing 8.1 Exemple d'attente pour une variable à être changé par une interruption

```
1 void __cpuinit calibrate_delay (void) {
2     ...
3     /* wait for "start of" clock tick */
4     ticks = jiffies;
5     while (ticks == jiffies)
6         /* nothing */;
7     ...
8 }
```

En utilisant cette synchronisation SystemC, le verrouillage du simulateur dans une boucle infinie est évité. Toutefois, des instructions supplémentaires cible sont simulées jusqu'à ce que cette synchronisation soit déclenchée, provoquant des inexactitudes de temps du simulateur.

8.2.1.2.2 Synchronisations SystemC causés par les instructions cible de synchronisation

Les logiciels multithread exigent que les threads se synchronisent. Le spinlock est un exemple de mécanisme de synchronisation du logiciel. Les fonctions de verrouillage et

déverrouillage des spinlock sont généralement mises en œuvre en utilisant les instructions load et store exclusive.

Figure 8.6 présente un exemple de logiciel en cours d'exécution sur deux processeurs utilisant un spinlock pour la synchronisation. Cette figure montre le comportement de notre simulateur et ce qui se passerait si le simulateur ne générerait pas une synchronisation SystemC pour ce type d'instructions cible ou si les fonctions spinlock n'étaient pas mis en œuvre en utimisant les instructions d'accès exclusif.

Figure 8.6.a présente l'exécution sur un matériel réel. Le premier processeur ($P1$, représenté par la ligne continue) verrouille un spinlock à $t1$, en $t2$ un défaut de cache se produit, en $t4$ $P1$ libère le spinlock et à $t6$ il exécute une opération I/O. Le deuxième processeur ($P2$, représentée par la ligne pointillée) essaie de verrouiller le spinlock ($t3$) juste avant $t4$, il obtient effectivement le verrou en $t4'$ et libère le spinlock à $t5$.

L'exécution de notre plate-forme dans le cas où le simulateur ne générerait pas de synchronisation SystemC pour les accès exclusif est représentée dans la Figure 8.6(b). Considérant que $P1$ est le premier ordonnancé pour la simulation, il verrouille le spinlock en $t1$ sans être désordonnancé (le spinlock est placé dans la mémoire principale), mais en $t2$ il est désordonnancé pour la synchronisation avant de charger la ligne de cache. $P2$ est maintenant ordonnancé et en $t3$ il commence à essayer de prendre le spinlock. $P2$ lit dans une boucle infinie la valeur verrouillée du spinlock depuis la mémoire principale.

Après le temps prédéfini sans synchronisation, $P2$ serait toutefois désordonnancé au temps $t4 + N$. Puis, $P1$ est ordonnancé, il déverrouille le spinlock et il est désordonnancé pour la synchronisation avant l'opération d'I/O. $P1$ sera maintenant en mesure de prendre le spinlock, mais il a simulé en plus le temps N.

Le comportement de simulation lorsque les fonctions des spinlocks utilisent les instructions d'accès exclusif et les synchronisations SystemC qui sont générées pour eux est présenté dans la Figure 8.6(c). Dans ce cas, les processeurs se synchronisent avant chaque verrouillage et déverrouillage. $P1$ se synchronise en $t1$ et $P2$ est ordonnancé. $P2$ est désordonnancé avant sa première tentative de verrouillage. $P1$ se synchronise à $t2$, mais elle est réordonnancé parce que $P1$ est plus avancé dans le temps de simulation ($t3 > t2$). Au $t4$, avant de relâcher le spinlock, $P1$ se synchronise et il est désordonnancé ($t4 > t3$). Entre $t3$ et $t4$ (le temps de simulation de $P1$), $P2$ se synchronise et il est réordonnancé à chaque tentative de verrouillage du spinlock. Après $t4$, $P1$ est ordonnancé, il libère le spinlock et il est simulé jusqu'à $t6$. $P2$ obtient le spinlock au $t4'$ (immédiatement après que $P1$ l'ait libéré) et le déverrouille à $t5$. Cela correspond au comportement attendu.

8.2.2 La modélisation du temps

Des estimations précises sur la consommation d'énergie ne peuvent être obtenues que si l'estimation temporelle est précise. Les simulateurs basés sur la traduction binaire n'ont pas de notion de temps simulé. Les événements d'échéance requis par les composants timers, utilisent habituellement différents mécanismes externes et parfois asynchrones à la plate-forme de simulation. Par exemple, QEMU utilise l'un des timers de l'hôte (HPET, rtc *etc.*), qui génère un signal d'alarme sur la machine hôte. La valeur d'expiration prévue pour le timer de l'hôte n'a pas de corrélation avec le temps simulé. La durée des instructions et le temps requis par les processeurs pour communiquer avec la mémoire et avec d'autres composants ne sont pas prises en compte par les simulateurs basés sur la traduction binaire. Ces aspects n'ont aucune importance pour le but de ces simulateurs, qui est d'être aussi rapide que possible et être fonctionnellement corrects.

Nous essayons d'obtenir une précision acceptable de la plate-forme de simulation et, en même temps, de maintenir la vitesse de simulation à un niveau acceptable. Au lieu

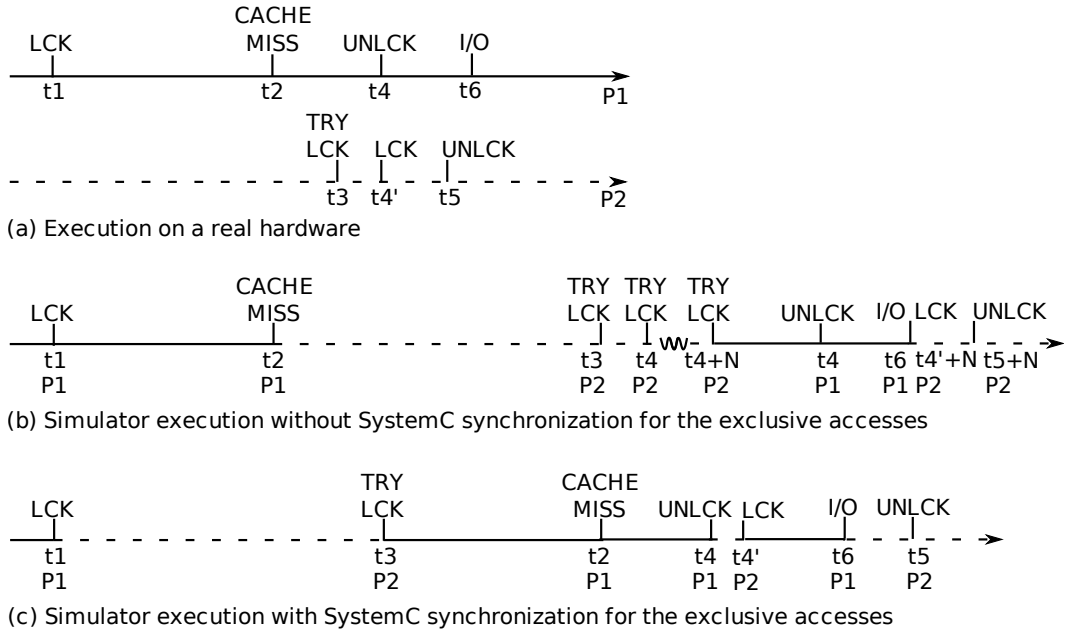


Figure 8.6: Comportements de simulation basée sur la mise en œuvre de spinlock

d'utiliser le temps hôte, notre plate-forme utilise le temps de simulation SystemC. Le timer de l'hôte n'est plus utilisé. Les interruptions d'horloge sont générées par les modules timers de SystemC.

Pour une modélisation précise du processeur, quelques modifications ont été apportées au simulateur initial afin de modéliser le temps requis pour exécuter les instructions. Ces modifications sont effectuées en ajoutant des informations temporelles lorsque le code binaire du processeur cible est traduit dans le code binaire du processeur hôte.

8.2.2.1 Annotations pour précision temporelle

La première annotation tente de rendre les processeurs simulés exacts du point de vue du **temps requis pour l'exécution interne des instructions**. Ces annotations sont nécessaires car il n'est pas possible de calculer le nombre de cycles cible simulés par la séquence de code hôte exécutée.

Lorsque le code binaire cible est traduit, nous ajoutons une micro-opération avant les micro-opérations de chaque instruction cible traduit. La micro-opération ajoutée incrémente le nombre de cycles des instructions qui ont été simulées depuis la dernière synchronisation. L'augmentation utilise le nombre de cycles des instructions données par la fiche technique du processeur simulé.

Parfois, le nombre de cycles nécessaires par une instruction dépend d'informations disponibles que lorsque l'instruction est exécutée. A titre d'exemple, le nombre de cycles nécessaires pour une instruction de multiplication peut dépendre des valeurs des opérandes. Les opérandes sont lus dans la mémoire et peuvent différer d'une exécution à l'autre. Pour un comptage précis des cycles, la traduction initiale de cette instruction est modifiée par l'ajout d'une partie qui augmente le nombre de cycles de simulation sur la base des informations d'exécution.

Une autre modification appliquée au simulateur initial pour améliorer la précision du temps consiste en la vérification de présence et le chargement de la ligne de cache requises

dans le **cache d'instruction**. La vérification et le chargement se fait dans une fonction appelée par une micro-opération insérée au début de chaque bloc de traduction, dans un bloc de traduction, avant la première instruction de chaque ligne de cache d'instruction. Dans le cas d'un défaut du cache d'instruction, le processeur actuellement simulé est synchronisé avec le reste de la plate-forme SystemC en consommant le temps requis pour exécuter le nombre de cycles obtenus par la première annotation. La transformation des cycles en temps est effectuée en utilisant la fréquence actuelle du processeur simulé. Après la synchronisation, l'adaptateur du processeur envoie une requête sur l'interconnexion pour un transfert en rafale depuis la mémoire de la ligne de cache nécessaire. Le temps SystemC requis pour accomplir le transfert est indépendant de la fréquence du processeur simulé, mais dépend de la latence de l'interconnexion, la latence du module de mémoire adressée et la concurrence avec les autres transferts.

Une modification similaire se réfère à l'accès de lecture/écriture aux données de la mémoire principale. Chaque fois qu'un emplacement en mémoire principale est lu, sa présence dans le **cache de données** est vérifiée. Le mécanisme décrit pour le défaut du cache d'instructions est également appliqué dans le cas d'un défaut du cache de données. Une opération d'écriture sur la mémoire principale pour une politique write-through génère une mise à jour de la valeur dans le cache de données (si l'adresse écrite est présente dans le cache de données), une synchronisation du processeur actuellement simulé et une demande d'écriture sur l'interconnexion vers le module de mémoire adressée.

Les opérations d'I/O avec modules SystemC génèrent également une synchronisation SystemC suivie d'une demande dans le sous-système de SystemC.

La Figure 8.7 présente un exemple simplifié de la façon dont la liste des micro-opérations générées par le simulateur basé sur la traduction binaire initial est annoté du point de vue du temps simulé. Les première et deuxième colonnes contiennent les adresses et les instructions du code du processeur cible. La troisième colonne illustre le fait que chaque instruction cible est traduite par le simulateur initial en un certain nombre de micro-opérations. Les dernières colonnes montre les annotations ajoutées à la liste des micro-opérations générées par le simulateur initial. Considérant que ce code cible représente le début d'un bloc de traduction et que la longueur d'une ligne de cache d'instruction est de 32 octets, le cache d'instructions est vérifié avant la première et la troisième instruction. Le nombre de cycles de simulation depuis la synchronisation précédente est mis à jour pour chaque instruction. Les annotations ajoutées pour une instruction de lecture/écriture sont visibles pour la deuxième, respectivement pour le troisième, instruction.

8.2.2.2 Niveaux de précision

La plate-forme de simulation peut être configurée pour être plus ou moins précise. Toutes les configurations comprennent la MMU fournie par le simulateur initial.

La première configuration de la plate-forme de simulation, appelée configuration **"cache full"**, la plus précise, implémente totalement les caches et utilise de(s) module(s) SystemC pour la mémoire principale (Figure 8.8). Les explications de toute cette section se concentrent sur cette configuration. Le temps requis pour exécuter le nombre de cycles correspondant aux instructions simulées est consommé en utilisant la fonction wait SystemC.

Les caches d'instructions et de données sont implémentés par chaque adaptateur du processeur. Dans l'implantation actuelle, les caches sont placés après la MMU, ils sont donc adressés physiquement. Ils sont "direct mapped" et utilisent la politique write-through. Toutefois, les principes présentés ici peuvent être utilisés avec d'autres stratégies de cache.

Instr address	Target code	List of micro-operations generated by QEMU	Annotated list of micro-operations generated by QEMU
18	instr1_reg_operation	micro-op1_for_instr1 ..... micro-opN1_for_instr1	instr_cache_verify (18); nb_cycles += cpu_datasheet [instr1]; micro-op1_for_instr1 ..... micro-opN1_for_instr1
1C	instr2_load_from_1000	micro-op1_for_instr2 ..... micro-opN2_for_instr2	nb_cycles += cpu_datasheet [instr2]; data_cache_verify (1000); micro-op1_for_instr2 ..... micro-opN2_for_instr2
20	instr3_store_5_to_2000	micro-op1_for_instr3 ..... micro-opN3_for_instr3	instr_cache_verify (20); nb_cycles += cpu_datasheet [instr3]; write_access (2000, 5); micro-op1_for_instr3 ..... micro-opN3_for_instr3

Figure 8.7: Exemple d'annotation de la liste des micro-opérations générées par le simulateur basé sur la traduction binaire initial

Par opposition à la simulation native ou simulation compilée [GGP08, vM96], la traduction binaire utilise l'adresse exacte pour obtenir des instructions cherchées et pour accéder aux données. Toutefois, les instructions cible sont récupérées de la mémoire principale une seule fois, pour la traduction, avant leur première exécution. Les instructions nécessaires pour la traduction sont recherchées directement dans le module mémoire, sans émettre une requête sur l'interconnexion (temps zéro pour SystemC). Le prix de chargement sera payé par le cache d'instructions (en fonction *instr_cache_verify* de la Figure 8.7) lorsque le code généré sera exécuté à partir du cache de translation.

Un défaut de cache d'instructions ou de données émet une requête sur l'interconnexion pour charger la ligne de cache. Comme le simulateur exécute toujours le code binaire hôte généré et stocké dans le cache de traduction, la partie donnée du cache d'instruction est ignorée. Les messages de cohérence peuvent être envoyés aux autres caches.

La communication avec d'autres périphériques s'effectue en envoyant des requêtes sur l'interconnexion. Le temps consacré à ces accès se compose du temps consommé par chaque composant SystemC impliqué dans la transmission et la réception des paquets de requête.

C'est la configuration la plus précise de notre plate-forme de simulation, sa précision ne dépend que de la précision de la modélisation des composants de la plate-forme SystemC. En outre, la simulation pour cette configuration est la plus lente, comme les demandes et les réponses passent toutes par les composants SystemC qui sont nécessaires pour les transférer.

La deuxième configuration de la plate-forme de simulation, appelée **configuration "no-cache"**, n'implémente pas les caches et utilise la mémoire principale interne allouée par le simulateur initial (Figure 8.9). Dans cette configuration, on considère que la mémoire est toujours disponible pour tous les processeurs, sans aucun coût en cycle pour y accéder. Comme dans la configuration précédente, les opérations d'I/O impliquent l'interconnexion et d'autres composants matériels SystemC.

Dans cette configuration, un processeur simulé se synchronise avec le reste de la plate-forme SystemC uniquement lorsqu'une opération d'I/O est exécutée et lorsque la fonction d'ISS retourne. En raison du nombre restreint de synchronisation SystemC, de gros

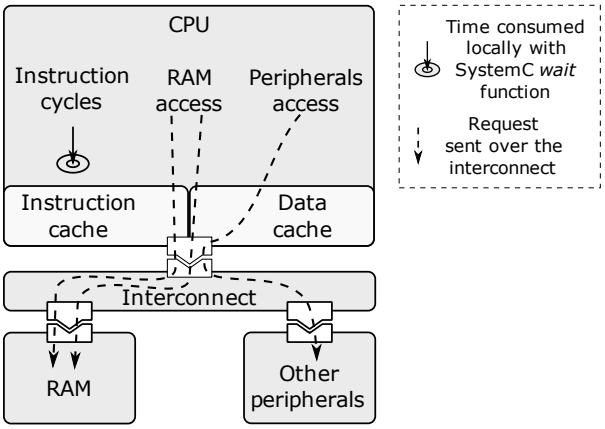


Figure 8.8: Le niveau de précision "cache full"

morceaux de code traduits sont exécutés sans avoir à recourir à SystemC. En conséquence, la simulation sera très rapide (proche du simulateur initial). La précision dans ce cas sera faible car les effets de cache et le temps nécessaire pour communiquer avec la mémoire principale sur l'interconnexion ne sont pas comptés. Comme tous les accès mémoire se font sans passer par l'interconnexion, il n'y a pas besoin d'un support explicite à des mécanismes de cohérence de cache.

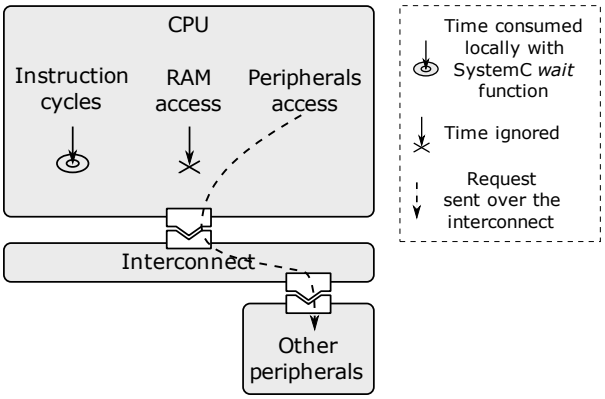


Figure 8.9: Le niveau de précision "no cache"

Dans la troisième configuration de la plate-forme de simulation, appelée configuration **"cache wait"**, les caches sont implémentés seulement du point de vue présence/défaut. Cette configuration utilise la mémoire centrale de la plate-forme de simulation initiale (Figure 8.10).

Dans cette configuration, nous modélisons le cache d'instructions et de données comme des répertoires purs, tels qu'un accès simple au tableau indique si l'instruction est en réalité dans le cache. Comme pour la première configuration, dans la mise en œuvre actuelle, les caches sont placés après la MMU et sont direct mapped.

Un défaut de cache émet un SystemC wait d'un temps précalculé nécessaire au chargement d'une ligne de cache, sans transmettre la demande à l'interconnexion. Le temps correspondant aux cycles de simulation est consommé au début de la prochaine synchronisation avec SystemC. Pour cette configuration, les processeurs sont synchronisés avec

SystemC quand un défaut de cache se produit, un I/O est exécuté ou la fonction de l'ISS retourne.

La vitesse de simulation pour cette configuration est réduite beaucoup par rapport à la configuration précédente en raison du grand nombre de synchronisations avec SystemC produite par le cache. Comme le temps précalculé est consommé directement dans le modèle de cache, un seul événement temporel SystemC est généré pour chaque défaut de cache. Ce n'est pas beaucoup, étant donné que le transfert d'un seul octet sur l'interconnexion nécessite plus de 10 événements SystemC temporel et non temporel. La précision augmente en utilisant une valeur moyenne précalculée pour le temps requis pour un transfert de mémoire sur l'interconnexion. Cependant, le contention de l'interconnexion, difficile à prévoir car très non linéaire lorsqu'elle est chargée, n'est pas prise en compte.

La **configuration "cache late"** est similaire à la configuration "cache wait". Au lieu de consommer le temps précalculé lorsque un défaut du cache se produit, la configuration "cache late" reporte cette consommation de temps jusqu'à la première synchronisation produite par une opération d'I/O ou par un retour normal de la fonction de l'ISS. Au moment de synchronisation, la somme des temps précalculés requis par tous les accès d'écriture et des défauts de cache qui ont eu lieu depuis la synchronisation précédente est consommée. De cette façon, le nombre de synchronisations SystemC est réduit, augmentant ainsi la vitesse de simulation.

Les chances d'éviter la modification par d'autres processeurs d'une variable attendue par le processeur courant simulé (expliqué avant dans la Figure 8.6.b) sont plus élevées dans cette configuration par rapport à la précédente en raison de la réduction du nombre de synchronisations. Cela a un impact sur la précision de la simulation.

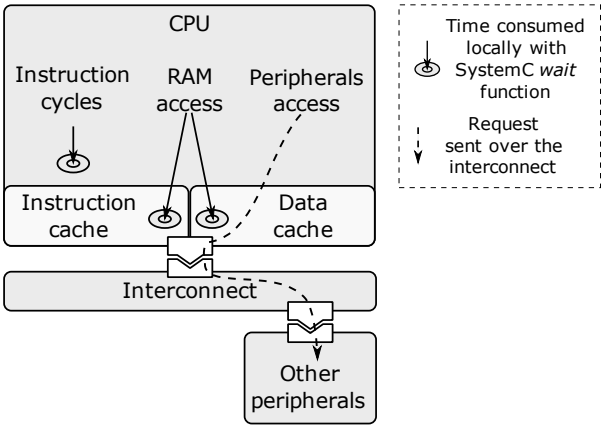


Figure 8.10: Les niveaux de précision "cache wait" et "cache late"

8.2.2.3 Traitement des interruptions

Les demandes d'interruption sont prises en compte un peu différemment dans notre stratégie de simulation comparé au simulateur initial. Dans l'exemple de la Figure 8.11, une demande d'interruption (IRQ) est produite par un composant matériel à $t2$, tandis que le processeur simule le bloc de traduction TBX.

Dans le simulateur initial, l'IRQ représente une alarme du système d'exploitation hôte, qui interrompt le processus de simulation de manière asynchrone. Le gestionnaire d'alarme active le fanion d'interruption en attendant et, le cas échéant, détruit le chaînage direct du

bloc de traduction courant au bloc de traduction suivant pour assurer que le simulateur obtient le contrôle à la fin de l'exécution du bloc de traduction en cours. Une fois le gestionnaire d'alarme terminé, la simulation reprend. À la fin du bloc de traduction, le simulateur va commencer le traitement de l'interruption.

Pour notre simulateur, étant simulés concurremment avec les processeurs, les composants matériels génèrent les demandes d'interruption pendant les activités SystemC des processeurs (synchronisation ou la communication avec d'autres composants). Dans l'exemple, l'interruption est générée par le dispositif tandis que l'adaptateur du processeur consomme des cycles simulés par le processeur dans l'intervalle entre $t1$ et le moment de la première synchronisation ($t3$ si un défaut de cache se produit au $t3$, $t5$ autrement). Pendant cette attente SystemC, au moment $t2$, le dispositif active le fanion d'interruption en attente du processeur. Comme pour le simulateur initial, le bloc de chaînage direct à partir du bloc de simulation courant au suivant est détruit. Au moment de l'activation du fanion d'interruption en attente, le bloc de simulation courant est TBX (respectivement TBY). Lorsque le processeur retourne de l'activité SystemC, il continue la simulation. À la fin du bloc de traduction en cours, le processeur va voir la nouvelle valeur du fanion d'interruption en attente et va commencer le traitement de l'interruption. Dans l'exemple, selon que le défaut du cache ait lieu ou non à $t3$, ce moment est $t4$ ou $t6$.

Ainsi, les interruptions sont traitées à la fin du bloc de traduction dans lequel la première synchronisation SystemC après la génération d'interruption se produit. Ce n'est pas nécessairement le bloc de traduction correspondant au moment de la génération de l'interruption.

Un peu de retard dans le traitement d'interruption n'a pas d'influence sur l'exécution du point de vue fonctionnel. Cependant, le temps de simulation peut être influencé.

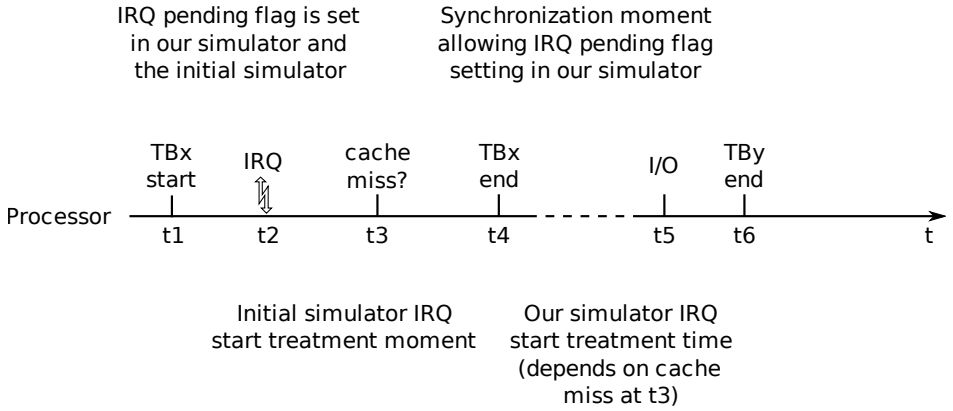


Figure 8.11: Le moment du traitement d'interruption

8.2.2.4 L'accès implicite (backdoor) aux composants matériels

Pendant la simulation, les valeurs nécessaires doivent parfois être obtenues à partir de composants matériels SystemC sans faire avancer le temps simulé. Par exemple, bien que la traduction du code binaire cible en code binaire de l'hôte nécessite plusieurs instructions cibles du composant SystemC de mémoire principale, pour une modélisation correcte du temps la traduction devrait prendre un temps SystemC égal à zéro. Les commandes émises par l'utilisateur pendant le débogage du logiciel simulé ne devrait pas non plus modifier le temps SystemC.

Pour permettre les accès qui nécessitent un temps zéro aux composants matériels SystemC, notre simulateur met en œuvre un mécanisme de backdoor. Ce mécanisme n'utilise ni les ports ni les canaux de communication SystemC.

8.3 Simulateur d'ordonnancement statique acceptant des fréquences multiples et changeant dynamiquement

Notre objectif est d'être capable de modéliser et de simuler des architectures qui contiennent des composants qui fonctionnent à des fréquences différentes du reste du système. Un exemple de cette architecture est donné dans la Figure 8.12. Le TTY, la mémoire, l'interconnexion et les autres périphériques dans la figure travaillent à la même fréquence, celle donnée par l'horloge principale. La fréquence de chaque processeur et de sa mémoire cache est contrôlée par un composant DVFS. Les composants DVFS fonctionnent à la fréquence de l'horloge principale et sont connectés à l'interconnexion. Le logiciel qui s'exécute sur les processeurs peut changer à tout moment la fréquence d'un processeur par l'émission d'une commande via l'interconnexion au composant DVFS correspondant. Les adaptateurs de fréquence rendent possible la communication entre les caches, travaillant à la fréquence prévue par les composants DVFS, et d'interconnexion, en travaillant à la fréquence de l'horloge principale.

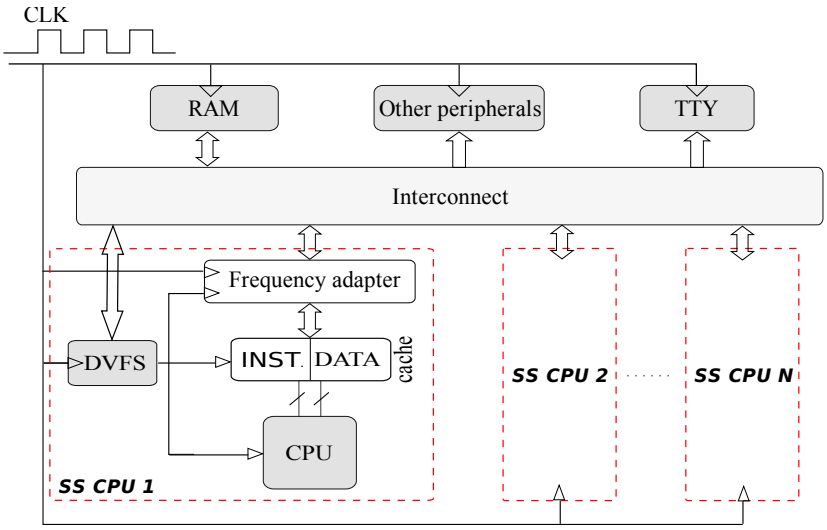


Figure 8.12: Exemple d'architecture avec plusieurs fréquences

Ces architectures peuvent être facilement modélisés avec le HDL d'un simulateur piloté sur événements à ordonnancement dynamique. La sortie d'un DVFS représente l'horloge de son processeur et de ses caches. Pour générer les fronts d'horloge sur cette sortie, le DVFS a plusieurs possibilités. Une possibilité est d'utiliser les événements temporels de façon similaire au module d'horloge SystemC. Une autre possibilité consiste à diviser la fréquence d'une autre horloge, par l'attente d'un certain nombre de changements de fronts d'horloge de cette nouvelle horloge avant de changer la valeur de sortie. La nouvelle horloge peut être interne ou externe au DVFS.

D'autre part, de telles architectures ne peuvent pas être modélisées et simulées par un simulateur utilisant un ordonnancement statique. Le processeur et ses caches ne peuvent avoir comme horloge d'autres signaux que l'horloge principale. Leurs fonctions FSM

(machine à états finis) pourraient être sensibles aux sorties des composants DVFS, mais seulement sur l'événement de changement de la valeur. Dans ce cas, les fonctions FSM seraient considérées comme des fonctions de Mealy et ils seront toujours exécutés sur le front négatif de l'horloge principale. En raison de la restriction qui interdit le changement d'état des fonctions de Mealy, la FSM du processeur et de la mémoire cache ne peut pas être mise en œuvre. En ce qui concerne la génération des fronts d'horloge à la sortie du DVFS, les techniques utilisées dans l'ordonnancement dynamique ne sont pas disponibles parce que les événements temporels n'existent pas et toutes les horloges de l'architecture sont considérées comme la même horloge.

La stratégie d'ordonnancement de simulation statique doit être adaptée pour traiter les architectures utilisant plusieurs fréquences. Nous proposons deux approches qui sont en mesure d'ordonnancer statiquement les processus de ces architectures.

Pour prouver que les simulateurs proposés ont un comportement correct, nous allons démontrer que leur comportement est identique à un simulateur événementiel pour n'importe quelle architecture simulée. Deux simulateurs sont considérés comme ayant un comportement identique s'ils produisent les mêmes valeurs de sortie, en même temps simulé, pour tous les composants de l'architecture simulée.

8.3.1 Le simulateur utilisant l'ordonnancement statique basé sur des horloges multiples

Dans notre première solution pour l'ordonnancement statique des architectures ayant des composants qui fonctionnent à fréquences différentes, chaque signal qui est utilisé par certains composants comme un signal d'horloge est générée par un composant d'horloge réel. Les architectures modélisées pour cette méthode utilisent des composants d'horloge multiples et ne contiennent pas de composants dont le port de sortie représente le signal d'horloge pour d'autres composants, comme dans la Figure 8.12.

Nous avons développé deux versions de ce simulateur. La version statique ne peut simuler que les architectures contenant des composants dont les fréquences de travail peuvent être différentes, mais ne changent pas pendant la simulation. Les fréquences utilisées sont celles correspondant aux périodes spécifiées lorsque les composants d'horloge sont instanciés.

La version dynamique du simulateur accepte la modification de la fréquence de travail au cours de la simulation. Des architectures comme dans la Figure 8.12 peuvent être simulées en utilisant la version dynamique du simulateur si leur modélisation est modifiée en conséquence.

8.3.1.1 La version statique du simulateur

L'idée principale de cette approche consiste à choisir la période du motif d'ordonnancement égale au plus petit commun multiple (PPCM) des périodes de l'horloge.

L'exemple donné dans la Figure 8.13 inclut trois horloges avec des périodes de cycle de 4 ns, 6 ns, respectivement 2 ns (fréquences 250 MHz, 166 MHz et 500 MHz). La période du motif d'ordonnancement sera dans ce cas 12 ns.

Chaque horloge a un front montant et un front descendant. Les fronts d'une horloge se succèdent normalement pendant la période du motif d'ordonnancement. Comme la période du motif d'ordonnancement est un multiple de toutes les périodes du cycle d'horloge dans l'architecture, à la fin d'un motif d'ordonnancement, chaque horloge sera au même décalage dans sa période comme qu'au début de ce motif. Dans l'exemple, toutes les horloges commencent le motif avec leur front montant.

Pendant la période du motif d'ordonnancement, les fronts de plusieurs horloges peuvent se chevaucher. Les moments qui contiennent au moins un front d'horloge représentent les points de simulation du motif d'ordonnancement. Le nombre de points de simulation dans un motif dépend du nombre d'horloges, de la durée de niveau positif et négatif de l'horloge et du décalage dans la période au début du motif. Le motif de l'exemple présenté contient douze points de simulation.

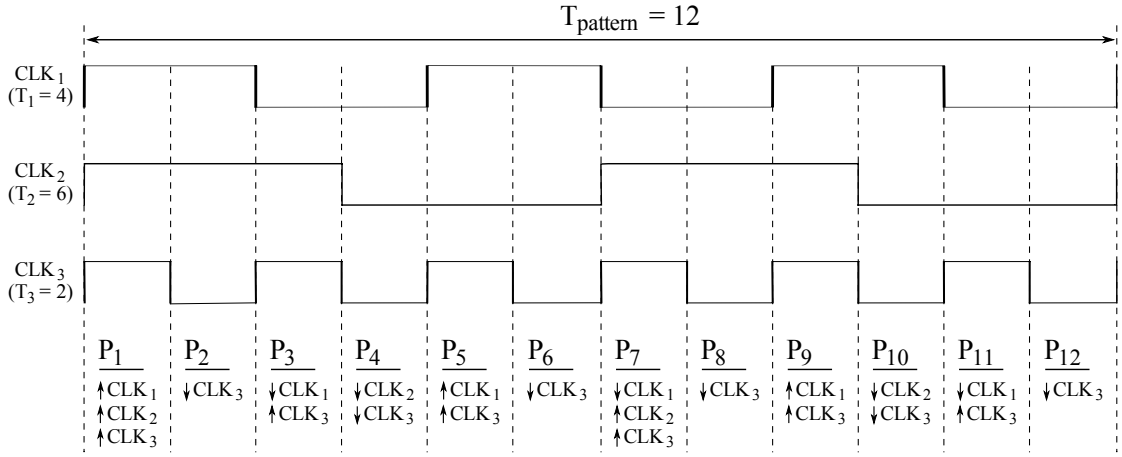


Figure 8.13: Motif d'exécution pour une architecture contenant horloges multiples

Le simulateur proposé utilise deux niveaux d'ordonnancement. Le premier niveau d'ordonnancement établit le point suivant de la simulation, gère le temps de simulation et appelle le deuxième niveau d'ordonnancement.

Les processus dépendant des fronts d'horloge du point de simulation actuel sont appelés dans le deuxième niveau d'ordonnancement. Un point de simulation peut contenir les deux types des fronts, montant et descendant. Dans l'exemple de la Figure 8.13, pour le troisième point de simulation (P_3) le front descendant de la première horloge chevauche sur le front montant de la troisième horloge.

Dans un simulateur basé sur événements à ordonnancement dynamique, les événements temporels correspondant aux fronts d'horloge d'un point de simulation auraient la même étiquette de temps et les processus de la liste statique de ces événements seraient tous exécutés dans le même cycle delta. Les valeurs écrites aux ports de sortie et les registres d'état ne seraient visibles qu'après la fin du cycle delta. Dans le cas possible lorsque les cycles delta sont générés, seules les fonctions de Mealy seraient exécutées puisque les fonctions de Transition et de Moore ne dépendent pas des sorties.

Le schéma d'ordonnancement pour un point de simulation est illustré dans le Listing 8.2. Ce schéma d'ordonnancement assure un comportement identique à un simulateur basé sur événements. Le schéma d'ordonnancement est basé sur le fait que le simulateur applique les optimisations SystemCASS concernant l'écriture des ports de sortie et l'ordonnancement des fonctions de Mealy. En raison de l'optimisation de l'écriture des ports de sortie, toutes les fonctions de Transition dépendant des fronts montants d'horloge dans le point de simulation doivent être exécutées avant les fonctions de Moore requises par le point de simulation. Sinon, une fonction de Transition exécutée après une fonction de Moore verrait directement la valeur que vient d'écrire cette fonction de Moore, au lieu de la valeur du début du cycle delta. Cela pourrait conduire à une transition dans un mauvais état et finalement à un mauvais comportement.

Listing 8.2 L'exécution d'un point de simulation pour l'ordonnanceur statique basé sur des horloges multiples

```
1 void simulation_point (void) {
2     transition_processes ();
3     moore_processes ();
4     update_registers ();
5
6     mealy_processes ();
7 }
```

La mise à jour des registres d'état (ligne 4) est effectuée après l'exécution de toutes les fonctions de Moore dépendant des fronts d'horloge descendant du point de simulation. Cela est dû a fait que les fonctions de Transition et de Moore du point de simulation doivent être exécutées dans le même cycle delta. Une fonction de Moore dépendant d'une horloge devrait voir la valeur d'état du début du cycle delta, pas celle modifiée par une fonction de Transition du même module, mais dépendant d'une autre horloge. Les registres d'état mis à jour marquent la fin du premier cycle delta du point de simulation.

Les fonctions de Mealy sont ensuite stabilisées en utilisant la solution d'optimisation SystemCASS. Contrairement aux fonctions de Transition et de Moore, qui sont appelées dans les points de simulation où elles sont impliquées, toutes les fonctions de Mealy sont appelées à chaque point de simulation.

En ce qui concerne la mise en œuvre du simulateur, contrairement à SystemCASS, chaque horloge est considérée séparément avec ses propres période et rapport cyclique.

L'instanciation des composants et des signaux et la liaison des ports de composants aux signaux sont effectués, comme dans SystemCASS.

Pour chaque horloge instanciée dans le système, une liste de processus dépendant du front montant de l'horloge et une liste de processus dépendant du front descendant de l'horloge sont mises à jour. Lorsque les processus sont triés, chaque fonction de Transition et de Moore sont ajoutées à la liste de l'horloge dont elles dépendent.

La liste des processus d'une horloge ne change pas pendant la simulation, même si la fréquence d'horloge change, comme nous le décrirons plus tard. Pour éviter le coût d'itération de ces listes, une fonction est générée pour chaque liste. Un exemple de fonction générée pour la liste de fonctions de Transition d'une horloge est présenté dans le Listing 8.3.

Listing 8.3 Exemple de fonction généré pour une horloge

```
1  void transition_clock_0xbfb10238 (void) {
2      //clock clk_400MHz
3      //timer0.transition
4      ((ENTRY_FUNC) 0x80950a0ULL) ((void *) 0x9a46600);
5      //gmn.transition
6      ((ENTRY_FUNC) 0x807ece0ULL) ((void *) 0x9cc2990);
7      //ram0_text.transition
8      ((ENTRY_FUNC) 0x809f600ULL) ((void *) 0x9e4fad0);
9      //ram1_data.transition
10     ((ENTRY_FUNC) 0x809f600ULL) ((void *) 0x9edb478);
11     ...
12 }
```

La liste des points de simulation du motif d'ordonnancement initial est construite avant le début de la simulation. Chaque point de simulation contient une liste des fronts montants et une liste des fronts descendants d'horloge qui doivent être traités dans ce point de simulation. En fait, la liste des fronts d'horloge conserve les pointeurs vers les fonctions générées pour les fronts d'horloge. A titre d'exemple, une liste pour le point de simulation P_7 dans la Figure 8.13 contient les pointeurs vers les fonctions générées pour le front montant de la deuxième et troisième horloge. L'autre liste contient un pointeur vers la fonction générée pour le front descendant de la première horloge. Un point de simulation contient également un décalage temporel qui doit être ajouté pour le passage au point de simulation suivant.

Pendant la simulation, la liste des pointeurs vers les fonctions générées est utilisée pour appeler toutes les fonctions de Transition et de Moore du point de simulation.

Pour accélérer la simulation, une fonction qui déroule à la fois les deux niveaux d'ordonnancement peut être générée. Cette fonction déroulerait tous les points de simulation du motif d'ordonnancement et mettrait à jour le temps simulé après chaque point de simulation. Pour chaque point de simulation, les listes utilisées pour appeler les fonctions de Transition et de Moore seraient déroulées. Le simulateur peut dans ce cas modéliser des architectures avec des horloges multiples et les simuler à la vitesse de SystemCASS.

Le HDL de cette version statique du simulateur ne change pas par rapport au HDL du simulateur duquel il dérive. La personne qui modélise l'architecture ne doit pas être au courant qu'un autre simulateur est utilisé.

8.3.1.2 La version dynamique du simulateur

Pour simuler les architectures utilisant des fréquences multiples qui peuvent changer au cours de la simulation, plusieurs ajustements sont à faire, tant au simulateur que à son HDL. Nous visons les architectures dont les signaux d'entrée utilisés comme signaux d'horloge par les composants matériels de cette architecture ont les caractéristiques d'un signal d'horloge: périodicité et un rapport de cycle. Les fréquences de ces signaux d'entrée peuvent changer au cours de la simulation.

Un exemple d'architecture visée a été déjà présenté dans la Figure 8.12. Pour modéliser ce type d'architecture, les ports de sortie qui contrôlent le signal d'horloge d'autres composants sont remplacés par des composants d'horloges réelles. En ce qui concerne le HDL de SystemCASS, les *sc_out*s des composants qui sont à la frontière de deux domaines d'horloge sont remplacées par *sc_clock*s.

Dans la figure, le port de sortie des composants DVFS est remplacé par un composant d'horloge. Le processeur, la mémoire cache et l'adaptateur de fréquence sera directement relié à ce composant d'horloge. Ce faisant, l'architecture peut être simulé uniquement avec les fréquences initialement prévues pour les horloges des DVFS.

Pour avoir un moyen de modifier les fréquences des horloges, nous introduisons l'API suivante au composant d'horloge:

void change_period (double new_period, double new_duty_cycle = 0.5);

Dans l'exemple, au lieu de générer en permanence le signal de sortie considéré comme signal d'horloge par d'autres composants, le composant DVFS doit appeler cette nouvelle fonction quand il veut changer la fréquence de sortie.

Cette fonction sera appelée sur le front d'une horloge et elle va modifier la fréquence d'une autre horloge. La Figure 8.15 présente un exemple de changement de fréquence pour les horloges dans la Figure 8.13. Un processus de Moore exécuté sur le front descendant de la troisième horloge (point de simulation P_8) veut changer la fréquence de la deuxième horloge.

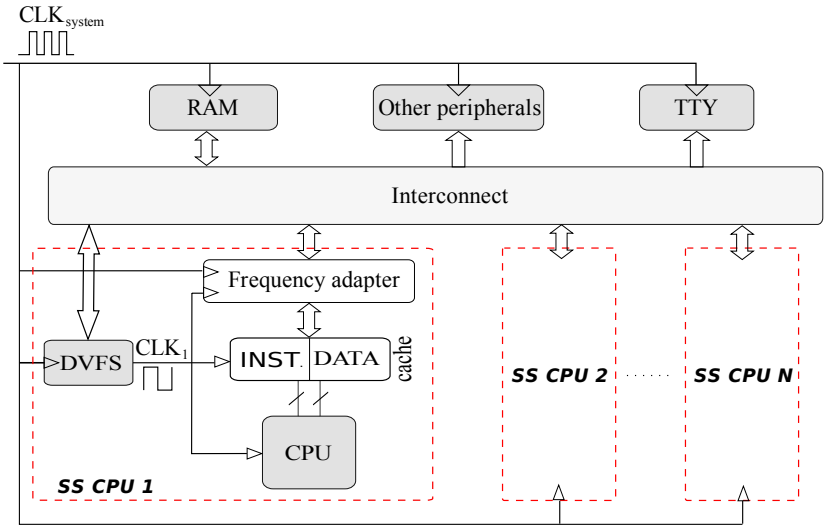


Figure 8.14: La modélisation des architectures avec fréquences multiples en utilisant plusieurs horloges

Le changement de fréquence d'une horloge ne se fait pas nécessairement au temps simulé de la demande. Si l'horloge dont la fréquence doit être changée ne change pas son niveau logique dans le point de la simulation de la demande, le changement est reportée jusqu'au premier front de l'horloge. Nous avons fait ce choix car il est probable qu'un périphérique matériel le ferait ansi pour éviter les "glitches". Dans l'exemple, le changement de fréquence de la deuxième horloge est reporté jusqu'à ce que le front descendant arrive au point de simulation P_{10}.

Une liste des changements de fréquence à venir est maintenue. Chaque élément de cette liste contient un pointeur vers l'horloge dont la fréquence doit être changée, le temps programmé pour le changement et les propriétés futures de l'horloge (période, rapport cyclique (duty cycle) et quel front sera le premier). Cette liste est vérifiée à la fin de chaque point de simulation.

Si au moment simulé courant au moins une fréquence d'horloge doit être changée, un nouveau motif d'ordonnancement est construit dynamiquement. Dans l'exemple, étant donné que la nouvelle période de la seconde horloge est de 2 ns, la nouvelle période sera de 4 ns.

Les événements des fronts d'horloge dont la fréquence ne change pas devraient se produire dans le nouveau motif d'ordonnancement au même temps simulé qu'ils apparaîtraient dans l'ancien modèle d'ordonnancement. En d'autres termes, tandis que la fréquence d'une horloge ne change pas, la forme d'onde de l'horloge reste constante. Habituellement, il y a des horloges qui ne commencent pas le motif avec un front. Dans l'exemple, la première horloge aura le premier front à un décalage de 1 ns à partir du début du motif. Aussi, la première horloge ne finit pas le motif avec un événement. Comme la période du motif est un multiple de toutes les périodes d'horloge, le temps restant après le dernier front d'une horloge jusqu'à la fin du motif, avec le temps depuis le début du motif jusqu'au premier front d'horloge, complète exactement la durée du prochain front de l'horloge.

Le simulateur d'ordonnancement statique proposé peut désormais simuler les architectures avec des fréquences multiples et changeant dynamiquement.

Le HDL est un peu modifié et le modèle d'architecture n'est pas la même que pour un

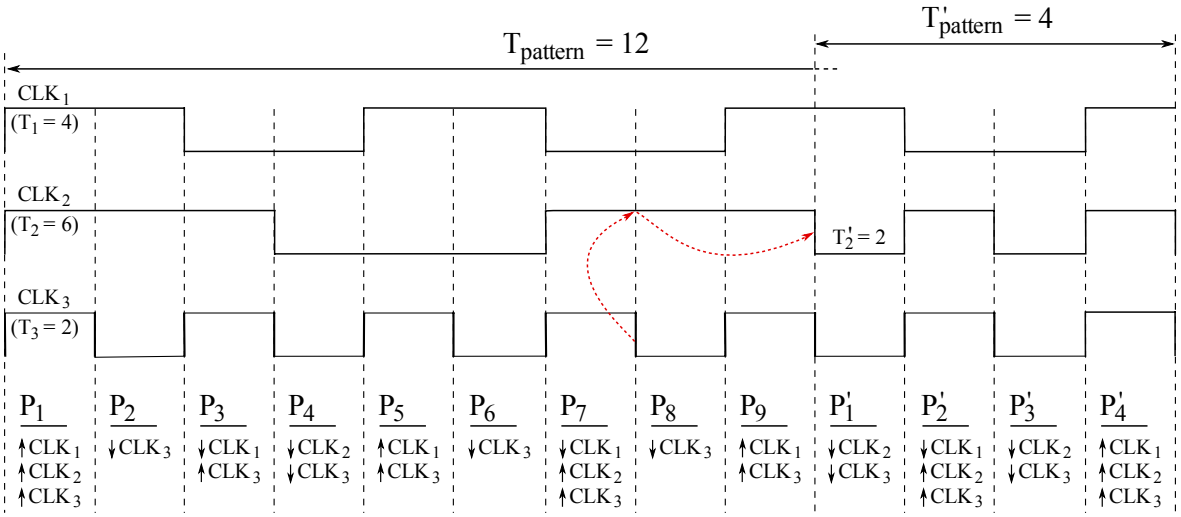


Figure 8.15: Changement de fréquence dans une architecture avec multiples horloges

ordonnanceur dynamique. Les composants comme le DVFS doivent être écrits spécialement pour ce simulateur.

La vitesse de simulation est inférieure à celle de la version statique du simulateur. En fait, la vitesse de simulation est fortement influencée par le nombre de changements de fréquence. Le motif d'ordonnancement doit être reconstruit pour chaque changement de fréquence.

8.3.2 Simulateur utilisant l'ordonnancement statique basé sur division de fréquence

La deuxième solution pour l'ordonnancement statique des architectures contenant des composants utilisant des fréquences différentes consiste à choisir la fréquence de l'horloge unique dans l'architecture égale au plus petit commun multiple de toutes les fréquences possibles de tous les composants dans l'architecture.

Les fréquences requises par les composants matériels sont obtenues en divisant la fréquence PPCM. La période du motif d'ordonnancement sera la période correspondant à cette nouvelle fréquence. Le motif d'ordonnancement contient deux points de simulation, correspondant aux fronts positif et négatif d'horloge.

La Figure 8.16 montre comment l'architecture présentée dans la Figure 8.12 est modélisée en utilisant la technique de division de la fréquence.

L'interconnexion, la mémoire, une partie de l'adaptateur de fréquence et tous les autres périphériques à l'exception des caches et des processeurs travaillent à la même fréquence. Cette fréquence ne change pas au cours de la simulation. Un nouvel élément introduit (appelée *Frequency divisor* dans la figure) obtient cette fréquence divisant la fréquence PPCM par une constante entière. Si le diviseur de constante est 1, l'adaptateur de fréquence n'est pas nécessaire. La fréquence de chaque processeur, de sa mémoire cache et d'une autre partie de son adaptateur de fréquence est contrôlée par un composant DVFS. Ainsi, tous les composants à l'exception des diviseurs de fréquence et des DVFSes utilisent comme horloge la sortie d'un autre composant.

Pour accepter la dépendance sur un front d'un signal de sortie, nous avons modifié la façon dont un cycle est simulé. Le Listing 8.4 présente la simulation d'un cycle. Il

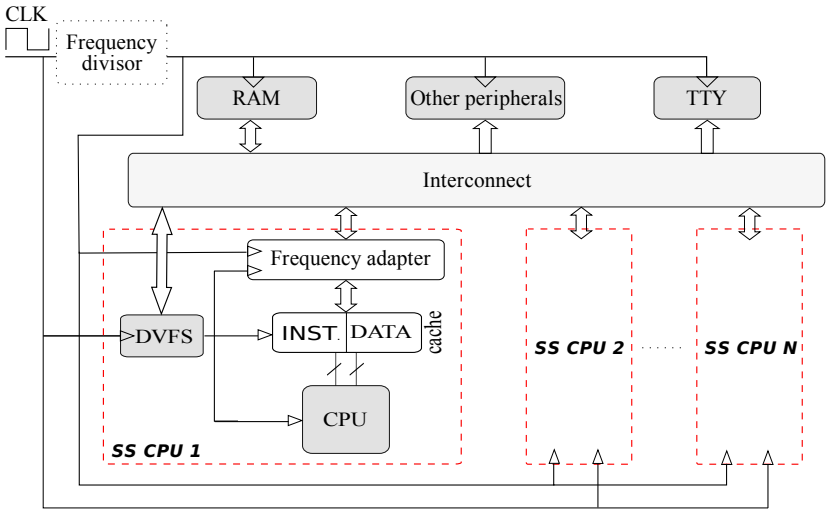


Figure 8.16: La modélisation des architectures avec fréquences multiples en divisant une seule fréquence

suppose que le simulateur applique les optimisations SystemCASS concernant l'écriture des ports de sortie et l'ordonnancement des fonctions Mealy. Par rapport au cycle normal de simulation, un appel aux processus dépendant sur un signal de sortie a été ajouté (lignes 7 à 9).

Listing 8.4 La simulation d'un cycle dans le simulateur basé sur la division de fréquence

```
1  void simulate_a_cycle (void) {
2      transition_processes ();
3      update_registers ();
4
5      moore_processes ();
6
7      out_signal_clock_transition_processes ();
8      out_signal_clock_moore_processes ();
9      update_registers ();
10
11     mealy_processes ();
12 }
```

Les processus de Transition utilisant comme horloge un signal de sortie sont regroupés dans une liste. Les fonctions de Moore sont regroupées dans une autre liste. Les processus de ces listes sont appelés lorsque leur front requis arrive. Comme d'habitude, les fonctions de Transition doivent être exécutées avant les fonctions de Moore, autrement une transition dans un mauvais état peut se produire. Dans un simulateur basé sur événements pour l'ordonnancement dynamique, ces processus de Transition et de Moore seraient tous exécutés dans le même delta cycle. Pour simuler le même comportement, les registres sont mis à jour après l'exécution du processus de Moore.

Au regard des règles de FSM selon lesquelles seulement les processus de Moore et de Mealy peuvent écrire sur les ports de sortie, un processus de Mealy ne devrait pas participer à la génération du signal d'horloge parce que la variation de sa sortie pendant la phase de stabilisation créerait de faux cycles pour les composants pour lesquels cette sortie

représente le signal d'horloge. Ce sont les raisons pour lesquelles les processus dépendant d'un signal de sortie ne sont pas appelés après l'exécution des processus de Transition et de Mealy.

La restriction d'écriture des processus de Transition rend impossible la mise en œuvre d'un diviseur capable de faire une division par 1. Une solution pour résoudre ce problème est de doubler la fréquence de l'horloge système. La division par 1 devient une division par 2, ce qui peut être désormais mis en œuvre. Au moins la moitié des cycles simulés ne seront pas utilisés.

Tous processus de Mealy dans l'architecture sont stabilisés ensemble, peu importe le type de leur signal d'horloge.

En ce qui concerne la mise en œuvre du simulateur, les deux listes des processus dépendant des signaux de sortie sont construites lorsque les processus sont triés.

Avant le début de la simulation, une fonction est générée pour chacune de ces deux listes. La Figure 8.5 présente un exemple de fonction générée. Pour chaque signal de sortie dont au moins un processus dépend, une variable statique est créé. Ces variables gardent l'ancienne valeur des signaux de sortie, puisque seule la valeur actuelle est maintenue par défaut par les ports de sortie. Le type de la variable dépend de la taille de donnée du signal. En utilisant les deux valeurs, nous pouvons déterminer si l'événement nécessaire s'est produit. Lorsqu'un événement se produit, tous les processus de la liste dépendant de cet événement sont exécutés.

Listing 8.5 Fonction générée pour les processus dépendant sur les fronts montants des signaux de sortie

```
1  inline void out_signal_clock_transition_processes (void) {
2      register fct p;
3
4      static unsigned char val1 = 0;
5      if (*(unsigned char *) 0x934cb84ULL != val1) {
6          val1 = *(unsigned char *) 0x934cb84ULL;
7          if (val1) {
8              // ARM1->transition ()
9              p.integer = 0x80a7160ULL;
10             p.pf ((void *) 0x86d1fa8);
11             // CACHE1->transition ()
12             p.integer = 0x80ae630ULL;
13             p.pf ((void *) 0x86d4128);
14             // FQADAPTER1->transition_initiator ()
15             p.integer = 0x8089cb0ULL;
16             p.pf ((void *) 0x9355cb0);
17             ...
18         }
19         ...
20     }
21 }
```

Le simulateur d'ordonnancement statique présenté permet de simuler les architectures avec des fréquences multiples et changeant dynamiquement. Il ne peut pas simuler les architectures avec des horloges multiples. Une seule horloge est utilisée et les autres fréquences sont obtenues en divisant la fréquence de cette horloge.

Un seul niveau de dépendance du signal est autorisé. Il n'est pas permis à un composant, dont le signal d'horloge est fourni par la sortie d'un autre composant, de générer

le signal d'horloge d'autres composants.

Le HDL n'est pas modifié pour ce simulateur. Le concepteur d'architecture doit calculer le PPCM de toutes les fréquences possibles quand il construit l'architecture. Il faut également ajouter le diviseur de fréquence et définir sa constante de division.

Ce simulateur ne peut pas être utilisé s'il y a des fréquences du système inconnues lorsque l'architecture est construite. Une fréquence future inconnue rend impossible le calcul du PPCM.

Les rapports entre la fréquence d'horloge système et les fréquences des composants influencent la vitesse de simulation. De grandes valeurs de ce ratio engendrent un grand nombre de cycles de simulation non utilisés. Seules les fonctions de Transition et de Moore des diviseurs de fréquence et les fonctions de Mealy de toutes les composantes sont exécutées dans ces cycles, ce qui ralenti ainsi la simulation.

Le nombre de changements de fréquence n'a pas d'influence sur la vitesse de simulation. Comme la fréquence de l'horloge système est un multiple de toutes les fréquences dans le système, aucun calcul particulier n'a lieu lorsque une fréquence est modifiée.

8.4 Algorithme adaptatif DVFS pour les architectures SMP

Dans cette section, nous décrivons notre algorithme adaptatif d'économie d'énergie pour les architectures SMP exécutant un non-RTOS (système d'exploitation non temps réel) préemptif. L'algorithme est mis en œuvre au niveau du système d'exploitation et il suppose que les applications exécutées sur le système d'exploitation ne fournissent aucune information sur leurs besoins de traitement. Contrairement aux RTOS, où le délai et le WCET (pire temps d'exécution) de chaque tâche dans le système sont connues, aucune de ces informations ne sont connues sur un non-RTOS.

Comme tous les autres algorithmes de DVFS, l'algorithme proposé permet d'économiser de l'énergie en réduisant la fréquence et la tension des processeurs lorsque l'utilisation du système le permet. Une partie du temps d'inactivité qui se produit en raison de l'absence de calcul est remplacée de cette façon par le calcul étendue par réduction de la fréquence.

8.4.1 Description conceptuelle de l'algorithme

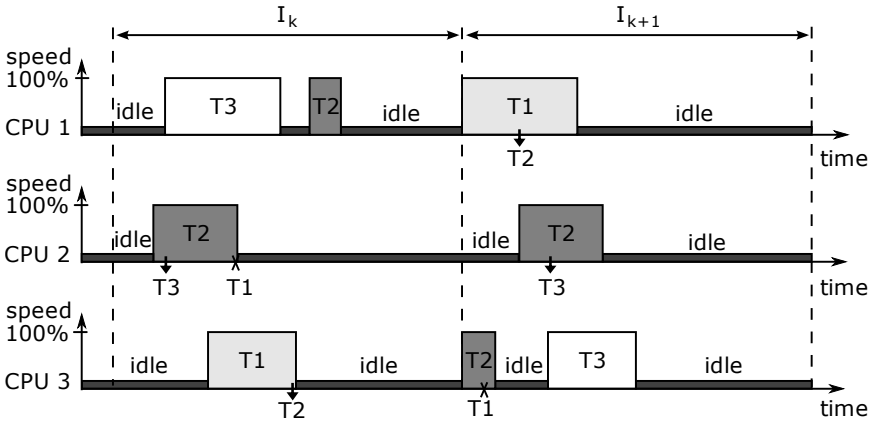
L'algorithme d'économie d'énergie proposé utilise des intervalles de temps égaux et étend la charge de travail de chaque intervalle sur tout l'intervalle suivant, comme PAST et les autres algorithmes pour systèmes monoprocesseur. L'algorithme gère les caractéristiques spécifiques des SMP, y compris la migration des tâches, par une gestion globale de la charge du travail du système et donc la fréquence de travail. En outre, pour des raisons d'efficacité, il considère une subdivision supplémentaire d'intervalle, comme détaillé plus loin.

Le système est modélisé comme un ensemble de N tâches $T_j \in \{T_1, \ldots, T_N\}$ exécutées sur M processeurs $P_i \in \{P_1, \ldots, P_M\}$ capables de fonctionner à une fréquence $f_i \in \{f_1, \ldots, f_m | f_1 < \ldots < f_m\}$. Les intervalles de temps nommés I_k contiennent $c_{j,k}$ cycles non-idle pour la tâche T_j dans le k-ème intervalle de temps. Ainsi, c_k, le nombre de cycles non-idle sur l'intervalle k, est donnée par $c_k = \sum_{j=1}^{N} c_{j,k}$. Les intervalles de temps I_k ont une durée constante t_I, qui représente C_I cycles à la fréquence maximale (f_m). Les sous-intervalles sont nommés S_l. Ils ont une période constant t_S, qui représente C_S cycles à la fréquence maximale.

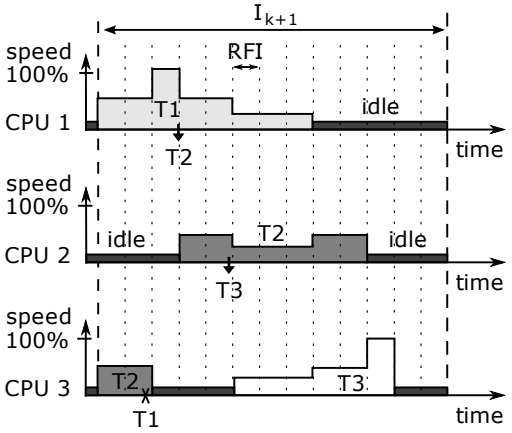
Pour illustrer l'idée, la Figure 8.17 montre en utilisant une exemple d'application avec trois tâches (T_1, T_2 and T_3) comment fonctionne l'algorithme. Dans cet exemple, T_2 a besoin de certaines données fournies par T_1 à un moment de son exécution et T_3 est lancé par T_2. Dans l'intervalle I_k (Figure 8.17(a)), les trois processeurs sont présentés travaillant à la fréquence maximale. Comme les tâches peuvent migrer d'un processeur à l'autre chaque fois qu'ils sont ordonnancés, T_1 est d'abord exécuté sur P_3, puis sur P_1 (Figure 8.17(a)).

L'algorithme obtient la charge de travail de l'intervalle en comptant tous les cycles non-idle exécutés dans l'intervalle par tous les processeurs. L'utilisation du système SMP (U_k) pour un intervalle k est obtenue en utilisant l'Equation (8.1), où le numérateur est le nombre de cycles non-idle de tous les processeurs et le dénominateur est le nombre total de cycles possibles de tous les processeurs (M est le nombre de processeurs, c_k est le nombre de cycles non-idle exécutés par les processeurs pendant l'intervalle k, C_I est le nombre de cycles à la fréquence maximale des processeurs dans un intervalle).

$$U_k = \frac{c_k}{M \times C_I} \tag{8.1}$$



(a) Intervalle I_k et I_{k+1} à la fréquence maximale



(b) Intervalle I_{k+1} à une fréquence optimisée par notre algorithme

Figure 8.17: Algorithme DVFS adaptatif proposé pour les SMP

Pour la situation de la Figure 8.17, l'utilisation du système dans l'intervalle I_k en utilisant l'Equation 8.1 est d'environ 33%. Régler la fréquence à 33% pour tout l'intervalle suivant n'est pas une bonne solution parce que tous les processeurs n'ont pas un travail

utile à exécuter pendant tout l'intervalle suivant. Par exemple, des périodes d'inactivité sont produites pendant le blocage d'une tâche en attente pour la synchronisation avec une autre tâche, s'il n'y a pas d'autre tâche prête à être ordonnancée pendant la période de blocage.

Une autre raison est que les tâches n'ont pas la même durée et une tâche ne peut pas être exécutée en parallèle par plusieurs processeurs. Afin de surmonter ce problème, notre algorithme divise chaque intervalle (I_k) en sous-intervalles, appelés intervalle de recalcul de fréquence (*RFI*), délimités par des pointillés dans la Figure 8.17(b). Au début de chaque RFI, la fréquence de fonctionnement des processeurs est recalculée pour permettre l'achèvement du travail util restant dans l'intervalle, en utilisant uniquement le nombre des processeurs non-idle courant. De manière intuitive, ceci réduit la probabilité de perte de performance des applications, en augmentant la fréquence sur les chemins critiques. La Figure 8.17(b) montre comment la fréquence des processeurs augmente et diminue au cours d'un intervalle.

Pour résumer, à la fin de chaque intervalle, la charge de travail de l'intervalle suivant est estimée. Puis, au début de chaque sous-intervalle, la fréquence de fonctionnement des processeurs est recalculée de sorte que la charge de travail restante dans la période actuelle peut être achevée en utilisant le nombre actuel de processeurs non-idle.

8.4.2 Implémentation

Cette section décrit l'implémentation de l'algorithme. Nous avons considéré que les processeurs peuvent travailler à seulement quelques valeurs de fréquence prédéfinies. Selon l'architecture, le temps d'inactivité (la tâche idle) consomme ou ne consomme pas d'énergie. Notre mise en œuvre diffère légèrement de l'algorithme conceptuel décrit ci-dessus en raison de contraintes matérielles concrètes. Il comporte trois parties, classées selon leur moment d'exécution et leur fonctionnalité.

8.4.2.1 Estimation de la charge de travail de l'intervalle

La première partie contient le code exécuté lorsque l'intervalle se termine. Elle est résumée par l'Algorithme 8.1. L'interruption de timer d'un processeur est programmée pour se produire lorsque chaque intervalle est terminé. Le gestionnaire d'interruption enregistre les compteurs de l'intervalle courant dans les compteurs du "dernier intervalle" afin qu'ils soient disponible pendant l'intervalle suivant.

Algorithm 8.1 I_{k+1} L'estimation de charge de travail

if $c_k < c_{k-1}$ **then** {current workload smaller than the previous one}
  $c_{k+1} \Leftarrow c_{k-1} - \frac{(c_{k-1}-c_k)}{2}$ {next workload expected between the two}
else {otherwise}
  $c_{k+1} \Leftarrow c_k + (c_k - c_{k-1})$ {next workload expected to increase more}
end if
$c^{est}_{k+1} \Leftarrow c_{k+1} + \text{Margin}(t_I)$ {keep some room for workload increasing detection}

Cette première partie estime également le nombre de cycles non-idle pour l'intervalle suivant. Si dans l'intervalle courant, il y avait moins de cycles non-idle que dans le précédent, le nombre de cycles non-idle estimé pour le prochain intervalle devrait être la moyenne des deux valeurs précédentes. Sinon, si le nombre de cycles non-idle a augmenté, l'algorithme prévoit qu'il va augmenter aussi au cours de l'intervalle suivant avec la même quantité.

En outre, dans tous les cas, l'algorithme laisse une marge qui est un pourcentage de la longueur de l'intervalle (par exemple 5 à 10 %). Cette marge a deux objectifs principaux: premièrement, elle permet à l'algorithme de maintenir un bon fonctionnement avec l'augmentation de la charge de travail en détectant les dépassements et, d'autre part, il permet de compenser la latence du mécanisme DVFS. Cette seconde partie ne peut pas être négligée avec les technologies actuelles, mais des mécanismes plus intelligents ont été proposés récemment, comme la VDD Hopping [MVR07], où les effets de stabilisation de fréquence sont limités. La longueur de cette marge dépend de l'utilisation du système. Lorsque l'utilisation du système est élevée, la longueur de la marge est faible car les processeurs fonctionneront à des fréquences élevées et la charge dépassant de travail sera facile à détecter. La longueur de marge augmente que l'utilisation diminue, parce quand la charge dépassant de travail devient plus difficile à détecter.

8.4.2.2 Calcul de RFI

La deuxième partie de l'algorithme recalcule la fréquence des processeurs au début de chaque RFI. Le code de cette partie est décrit par l'Algorithm 8.2 et il se trouve dans le gestionnaire d'interruption de timer d'un processeur. La fréquence (f_l) pour le sous-intervalle l de l'intervalle courant (I_k) est calculée avec l'Equation 8.2. Elle est basée sur le nombre de cycles non-idle estimés et exécutés dans l'intervalle courant (c_k^{est} et c_k^{done}), le nombre total de cycles à la fréquence maximale jusqu'à la fin de l'intervalle (donné par $C_I - (l-1) \cdot C_S$), le nombre de processeurs (M) et le nombre de tâches non-bloqués (N^{ready}). C_I et C_S sont le nombre de cycles de ce type dans l'intervalle complet et, respectivement, dans le RFI.

$$r_l = \frac{c_k^{est} - c_k^{done}}{C_I - (l-1) \cdot C_S} \times \frac{1}{min(M, N^{ready})} \tag{8.2}$$

La sélection de la fréquence f_l pour le sous-intervalle S_l est faite de telle sorte que $f_l / f_m \geq r_l$. Si le nombre de cycles estimés pour cet intervalle a déjà été exécuté, la fréquence est réglée au maximum jusqu'à la fin de l'intervalle parce que nous ne pouvons pas savoir combien de travail est nécessaire. Si le nombre de tâches disponibles pour l'exécution est inférieur au nombre de processeurs, le nombre de cycles à la fréquence maximale qui reste dans l'intervalle est calculé selon le nombre de tâches disponibles au lieu d'utiliser le nombre de processeurs. La valeur de la fréquence finale est la fréquence la plus basse acceptée par le processeur qui est supérieure ou égale à la fréquence requise pour l'achèvement les travaux jusqu'à la fin de l'intervalle.

La longueur du RFI est importante. Une valeur trop grande (à la limite, égale à la longueur de l'intervalle) rendrait impossible l'achèvement de la charge de travail estimée, tandis qu'une valeur trop petite introduirait une surcharge inutile.

Algorithm 8.2 Calcul de fréquence RFI

if $\left(c_k^{est} \leq c_k^{done}\right)$ **then**
  $f_l \Leftarrow f_m$
else
  $r_l \Leftarrow \frac{c_k^{est} - c_k^{done}}{C_I - (l-1).C_S} \times \frac{1}{min(M, N^{ready})}$
  $f_l \Leftarrow \text{select\_fq}(r_l)$
end if

8.4.2.3 Interaction avec l'ordonnanceur

La troisième et dernière partie actualise la charge de travail de l'intervalle courant et règle la fréquence des processeurs. Chaque fois qu'une tâche est ordonnancée, le nombre de cycles de son exécution est utilisé pour mettre à jour les compteurs de l'intervalle courant. Le nombre de tâches disponibles pour l'exécution est également maintenu. Quand une tâche est ordonnancée, si la tâche est la tâche idle, la fréquence du processeur sur lequel la tâche sera exécutée est réglée au minimum, sinon elle est réglée à la fréquence calculée dans la section précédente.

8.5 Conclusion et perspectives

L'objectif général de cette thèse était de proposer un algorithme DVFS d'économie d'énergie pour les architectures SMP et des plateformes MPSoC de simulation rapides et précises nécessaires pour la validation et la simulation de cet algorithme.

Nous avons étudié la possibilité de la modélisation de telles stratégies de simulation à différents niveaux d'abstraction.

Pour améliorer la vitesse de simulation au niveau d'abstraction transactionnel, dans la première contribution, nous avons remplacé les ISSes basés sur l'interprétation par des ISSes basés sur la traduction binaire. Pour un comportement temporel précis, nous avons modélisé le comportement temporel du processeur, créé des modèles de cache rapides et précis et résolu les problèmes de synchronisation dûs aux de modèles de calcul différents utilisés dans la traduction binaire de l'ISS et dans le reste du système. Pour la modélisation du temps nécessaire en interne pour l'exécution des instructions par les processeurs simulés, nous avons annoté le code traduit en utilisant le nombre de cycles correspondant aux instructions simulées. Un processeur est simulé alors qu'il ne communique pas avec le monde extérieur au delà de ses caches. Quand un défaut de cache de données ou d'instruction se produit ou une instruction I/O est exécutée, le processeur se synchronise avec le reste des composants matériels simulés concurremment par le simulateur basé sur événements. Pour la configuration la plus rapide du simulateur, l'accélération de la simulation est d'environ 380.

Dans la deuxième contribution, nous avons proposé deux stratégies d'ordonnancement statique qui peuvent simuler des architectures contenant des composants matériels travaillant à des fréquences différentes. Pour la première stratégie, chaque signal de fréquence est généré par un composant d'horloge réelle. Un motif d'ordonnancement ayant la période égale au plus petit commun multiple des périodes d'horloges est créé. Les fronts d'horloges se succèdent normalement pendant la période du motif d'ordonnancement. La simulation consiste à exécuter les processus dépendant sur les fronts du premier point de simulation dans le motif d'ordonnancement, puis ceux qui dépendent sur les fronts du deuxième point de simulation *etc.*. La deuxième approche utilise une horloge unique, dont la fréquence est égale au PPCM de toutes les fréquences possibles de toutes les composantes de l'architecture. Les fréquences requises par les composants matériels sont obtenues en divisant cette fréquence unique. La simulation consiste à exécuter les processus dépendant des fronts de l'horloge unique. Lorsque l'un de ces processus modifie un signal de sortie qui représente un signal d'horloge pour les autres composants, les processus dépendant de ce signal d'horloge sont exécutées. Aucun événement n'est généré lorsque la valeur d'un signal change. La condition d'exécution des processus dépendant des signaux d'horloge générés est vérifiée par le simulateur après chaque cycle de l'horloge. Ces stratégies donnent une accélération de 3,5 par rapport à des simulateurs basés sur événements.

Pour supporter le changement dynamique des fréquences dans le premier simulateur, une API a été ajoutée au composant d'horloge. Cette fonction API offre un moyen de modifier les fréquences des horloges. Lorsque la fréquence d'un composant d'horloge change, le simulateur proposé calcule un nouveau motif d'ordonnancement. Le second simulateur n'a rien à faire pour supporter les changements dynamiques de fréquences. Les composants qui génèrent un signal d'horloge calculent le nouveau diviseur pour obtenir la fréquence requise à partir de la fréquence globale.

Dans la dernière contribution, nous avons proposé un algorithme d'économie d'énergie qui divise le temps d'exécution en intervalles de temps égaux et étale la charge de travail de chaque intervalle sur tout l'intervalle prochain en réduisant la fréquence des processeurs. L'algorithme traite les caractéristiques spécifiques des architectures SMP (la migration des tâches entre les processeurs, la communication et la synchronisation des tâches) par une gestion globale de la charge de travail du système et une subdivision supplémentaire d'intervalle. À la fin de chaque intervalle, la charge de travail de l'intervalle suivant est estimée. Au début de chaque sous-intervalle, la fréquence de fonctionnement des processeurs est recalculée de sorte que la charge de travail restant dans l'intervalle actuel puisse être achevée. Grâce au changement rapide de fréquence et au convertisseur AC/DC, cette stratégie s'est révélée efficace et, bien que conduisant parfois à un temps de réponse plus long que prévu, conduit à des économies d'énergie de l'ordre de 45% en moyenne.

Travaux futurs à court terme. Pour les ISSes basés sur la traduction binaire, les travaux futurs à court terme consistent à intégrer ces modèles d'ISS avec les modèles de matériel cycle-précis de la bibliothèque SoCLib.

Les travaux futurs à court terme pour le simulateur utilisant l'ordonnancement statique basé sur des horloges multiples consistent à améliorer la vitesse de simulation lorsque les fréquences changent souvent en mémorisant les motifs d'ordonnancement déjà calculés et à créer un modèle d'équivalence entre les motifs d'ordonnancement.

Pour l'algorithme d'économie d'énergie, les travaux futurs à court terme seront axés sur l'intégration de cet algorithme dans l'OS Linux.

Travaux futurs à long terme. Pour les ISSes basés sur la traduction binaire, les travaux futurs à long terme se consistent en la parallélisation de la simulation afin de l'accélérer, tirant ainsi parti des architectures multi-core.

L'amélioration de la détection du travail dépassant représente le travail à long terme pour l'algorithme adaptatif.

Glossary

ALU	Arithmetic Logic Unit
API	Application Programming Interface
ASMP	Asymmetric MultiProcessing
CA	Cycle Accurate
CAD	Computer-Aided Design
CMOS	Complementary Metal–Oxide–Semiconductor
DMA	Direct Memory Access
DSP	Digital Signal Processor
DVFS	Dynamic Voltage and Frequency Scaling
FIFO	First In, First Out
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
GCC	GNU Compiler Collection
GDB	GNU Debugger
GNU	GNU's not Unix!
GPP	General Purpose Processor
HAL	Hardware Abstraction Layer
HDL	Hardware Description Language
HDS	Hardware Dependent Software
IP	Intellectual Property
IR	Intermediate Representation
IRQ	Interrupt Request
ISS	Instruction Set Simulator
ITRS	International Technology Roadmap for Semiconductors
LCM	Least Common Multiple
MMU	Memory Management Unit
MPSoC	MultiProcessors System on Chip
NOP	No Operation Performed
OS	Operating System
PE	Processing Engine
RAMDAC	Random Access Memory Digital-to-Analog Converter
RTL	Register Transfer Level
RTOS	Real-Time Operating System
SL	System Level
SMP	Symmetric MultiProcessing
SoC	System on Chip
TA	Transaction Accurate
TCG	Tiny Code Generator
TLM	Transaction Level Modeling
VA	Virtual Architecture
VHDL	VHSIC Hardware Description Language
VHSIC	Very-High-Speed Integrated Circuit
VLIW	Very Long Instruction Word
VMM	Virtual Machine Monitor
WCET	Worst-Case Execution Time

Bibliography

[ACG+07] David August, Jonathan Chang, Sylvain Girbal, Daniel Gracia-Perez, Gilles Mouchard, David A. Penry, Olivier Temam, and Neil Vachharajani. UNISIM: An Open Simulation Environment and Library for Complex Architecture Design and Collaborative Development. *IEEE Comput. Archit. Lett.*, 6(2):45–48, 2007. 3.1

[BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 1–12, Vancouver, British Columbia, Canada, 2000. 2.4

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, Bolton Landing, NY, USA, 2003. 2.4

[Bel05] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference*, pages 41–46, Anaheim, CA, 2005. USENIX Association. 2.4, 4.2

[Ber06] V. Berman. Standards: The P1685 IP-XACT IP Metadata Standard. *Design & Test of Computers, IEEE*, 23(4):316–317, April 2006. 2.3

[BPG04] Richard Buchmann, Frédéric Pétrot, and Alain Greiner. Fast cycle accurate simulator to simulate event-driven behavior. In *ICEEC '04*, pages 35–38, 2004. 3.2

[BR03] Bishop Brock and Karthrick Rajamani. Dynamic power management for embedded systems. In *Proceedings of the IEEE International SOC Conference*. IBM Research, September 2003. 3.3.3, 3.3.3

[Buc06] Richard Buchmann. Modélisation et Simulation Rapide au Niveau Cycle Pour l'Exploration Architecturale de Systemes Intégrés sur Puce. In *Thesis*, 2006. 5.2.1

[Buc07] Richard Buchmann. A fully static scheduling approach for fast cycle accurate systemC simulation of MPSoCs. In *Proceedings of the Internatonal Conference on Microelectronics (ICM'2007)*, pages 101–104, Cairo, Egypt, 2007. 5.2.1

[BW01] Alan Burns and Andrew J. Wellings. *Real-Time Systems and Programming Languages: ADA 95, Real-Time Java, and Real-Time POSIX*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. 2.7.1

[CG03] Lukai Cai and Daniel Gajski. Transaction level modeling: an overview. In *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 19–24, New York, NY, USA, 2003. ACM. 2.3.2

[CM96] Cristina Cifuentes and Vishv M. Malhotra. Binary Translation: Static, Dynamic, Retargetable? In *ICSM '96: Proceedings of the 1996 International Conference on Software Maintenance*, pages 340–349, Washington, DC, USA, 1996. IEEE Computer Society. 3.1

[Cre81] R. J. Creasy. The origin of the VM/370 time-sharing system. *IBM Journal of Research & Development*, 25(5):483–490, 1981. 2.4

[DBR98] S. Devine, E. Buignon, and M. Rosenblum. Virtualization System including a virtual machine monitor for computer with segmented architecture, October 1998. US patent number 6397242. 2.4

[Dev05] Fabrice Devaux. MECHANISMS FOR CPU VIRTUALIZATION, 2005. International patent number WO/2006/027488. 2.4

[DM99] Giovanni De Micheli. Hardware synthesis from C/C++ models. page 80, 1999. 2.2

[DMEW02] Giovanni De Micheli, Rolf Ernst, and Wayne Wolf, editors. *Readings in hardware/software co-design*. Kluwer Academic Publishers, 2002. 2.3

[Don04] Adam Donlin. Transaction level modeling: flows and use models. In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 75–80, New York, NY, USA, 2004. ACM. 2.3.2

[DPR08] Gaurav Dhiman, Kishore Kumar Pusukuri, and Tajana Rosing. Analysis of dynamic voltage scaling for system level energy management. In *HotPower '08*, 2008. 3.3.2

[FDW] Mischkalla Fabian, He Da, and Mueller Wolfgang. Closing the gap between UML-based modeling, simulation and synthesis of combined HW/SW systems. 3.1

[Fou07] Nicolas Fournel. Estimation et optimisation des performances temporelles et énergétiques pour la conception de logiciels embarqués. In *Thesis*, 2007. 4.6.1

[GCW95] Kinshuk Govil, Edwin Chan, and Hal Wasserman. Comparing algorithm for dynamic speed-setting of a low-power cpu. In *MobiCom '95: Proceedings of the 1st annual international conference on Mobile computing and networking*, pages 13–25, New York, NY, USA, 1995. ACM. 3.3.1

[Ger09] Patrice Gerin. Modéles de simulation pour la validation logicielle et l'exploration d'architectures des systémes multiprocesseurs sur puce. In *Thesis*, 2009. 2.3.4.2

[GGP08] Patrice Gerin, Xavier Guérin, and Frédéric Pétrot. Efficient implementation of native software simulation for MPSoC. In *DATE '08: Proceedings of the*

conference on Design, automation and test in Europe, pages 676–681, Munich, Germany, 2008. 4.4.2, 8.2.2.2

[Ghe06] Frank Ghenassia, editor. *Transaction Level Modeling With SystemC: TLM Concepts And Applications for Embedded Systems*. Springer Verlag, 2006. 2.3

[GIL+00] D. Grunwald, C. B. Morrey III, P. Levis, M. Neufeld, and K. I. Farkas. Policies for Dynamic Clock Scheduling. In *Proceedings Fourth Symposium OS Design and Implementation*, Oct. 2000. 3.3.1

[GL97] Rajesh K. Gupta and Stan Y. Liao. Using a Programming Language for Digital System Design. *IEEE Des. Test*, 14(2):72–80, 1997. 2.2

[GML+00] Dirk Grunwald, Charles B. Morrey, III, Philip Levis, Michael Neufeld, and Keith I. Farkas. Policies for dynamic clock scheduling. In *OSDI'00: Proceedings of the 4th conference on Symposium on Operating System Design & Implementation*, pages 6–6, Berkeley, CA, USA, 2000. USENIX Association. 3.3.3

[Han88] Craig Hansen. Hardware logic simulation by compilation. In *DAC '88: Proceedings of the 25th ACM/IEEE Design Automation Conference*, pages 712–716, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press. 3.2

[Hav00] Paul Johannes Mattheus Havinga. Design techniques for energy efficient and low-power systems. In *Mobile multimedia systems*, pages 2.1–2.52. University of Twente, 2000. 1, 8.1

[HCKT99] Graham R. Hellestrand, Ricky L. K. Chan, Ming Chi Kam, and James R. Torossian. Hardware and software co-simulation including executing an analyzed user program, October 1999. patent nb : 6230114. 2.4, 3.1

[HHP+07] Kai Huang, Sang-il Han, Katalin Popovici, Lisane Brisolara, Xavier Guerin, Lei Li, Xiaolang Yan, Soo-lk Chae, Luigi Carro, and Ahmed Amine Jerraya. Simulink-based MPSoC design flow: case study of Motion-JPEG and H.264. In *DAC '07: Proceedings of the 44th annual Design Automation Conference*, pages 39–42, New York, NY, USA, 2007. ACM. 2.3.1

[HJKH03] M. Horowitz, A. Joch, F. Kossentini, and A. Hallapuro. H.264/avc baseline profile decoder complexity analysis. *Circuits and Systems for Video Technology, IEEE Transactions on*, 13(7), 2003. 4.8.5.3

[HK03] Chung-Hsing Hsu and Ulrich Kremer. The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 38–48, New York, NY, USA, 2003. ACM. 3.3.1

[HP98] Denis Hommais and Frédéric Pétrot. Efficient Combinational Loops Handling for Cycle Precise Simulation of System on a Chip. In *EUROMICRO '98: Proceedings of the 24th Conference on EUROMICRO*, page 10051, Washington, DC, USA, 1998. IEEE Computer Society. 3.2, 5.2.1

[HSN+01] Andreas Hoffmann, Oliver Schliebusch, Achim Nohl, Gunnar Braun, Oliver Wahlen, and Heinrich Meyr. A methodology for the design of application specific instruction set processors (ASIP) using the machine description language LISA. In *ICCAD '01: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 625–630, Piscataway, NJ, USA, 2001. IEEE Press. 3.1

[ICM06] Canturk Isci, Gilberto Contreras, and Margaret Martonosi. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 359–370, Washington, DC, USA, 2006. IEEE Computer Society. 3.3.1

[ITR07a] ITRS. International Technology Roadmap for Semiconductors. In *System Drivers*, 2007. 1, 1, 8.1, 8.1

[ITR07b] ITRS. International Technology Roadmap for Semiconductors. In *Design*, 2007. 1, 8.1

[JBP06] Ahmed A. Jerraya, Aimen Bouchhima, and Frédéric Pétrot. Programming models and HW-SW interfaces abstraction for multi-processor SoC. In *DAC '06: Proceedings of the 43rd annual Design Automation Conference*, pages 280–285, New York, NY, USA, 2006. ACM. 2.3

[Jen91] Glenn Jennings. A case against event-driven simulation for digital system design. In *ANSS '91: Proceedings of the 24th annual symposium on Simulation*, pages 170–176, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press. (document), 3.2, 3.2, 3.5

[JH08] Vania Joloboff and Claude Helmstetter. SimSoC: A SystemC TLM integrated ISS for full system simulation. In *Asia Pacific Conference on Computer Architecture and Systems*, Macao Macao, 2008. (document), 3.1, 3.3

[JWP+05] Philo Juang, Qiang Wu, Li-Shiuan Peh, Margaret Martonosi, and Douglas W. Clark. Coordinated, distributed, formal energy management of chip multiprocessors. In *ISLPED '05: Proceedings of the 2005 international symposium on Low power electronics and design*, pages 127–130, New York, NY, USA, 2005. ACM. 3.3.2

[KD90] David Ku and Giovanni DeMicheli. HardwareC – A Language for Hardware Design (Version 2.0). Technical report, Stanford, CA, USA, 1990. 2.2

[Law] Kevin et al. Lawton. Open SystemC Initiative homepage. http://bochs.sourceforge.net. 4.1.1

[LS04] Jacob R. Lorch and Alan Jay Smith. PACE: A new approach to dynamic voltage scaling. *IEEE Trans. Comput.*, 53(7):856–869, 2004. 3.3

[LTMF95] Mike Tien-Chien Lee, Vivek Tiwari, Sharad Malik, and Masahiro Fujita. Power analysis and low-power scheduling techniques for embedded DSP software. In *ISSS '95: Proceedings of the 8th international symposium on System synthesis*, pages 110–115, New York, NY, USA, 1995. ACM. 1

[MB08] Andreas Merkel and Frank Bellosa. Memory-aware scheduling for energy efficiency on multicore processors. In *HotPower*, 2008. 3.3.2

[MFMB02] Steven M. Martin, Krisztian Flautner, Trevor Mudge, and David Blaauw. Combined dynamic voltage scaling and adaptive body biasing for lower power microprocessors under dynamic workloads. In *ICCAD '02: Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pages 721–725, New York, NY, USA, 2002. ACM. 2.6, 6.5

[MPM+07] Marius Monton, Antoni Portero, Marc Moreno, Borja Martinez, and Jordi Carrabina. Mixed SW/SystemC SoC Emulation Framework. In *Industrial Electronics, 2007. ISIE 2007. IEEE International Symposium on*, pages 2338–2341, 2007. (document), 3.1, 3.2, 4.7.1

[MVR07] S. Miermont, P. Vivet, and M. Renaudin. A Power Supply Selector for Energy- and Area-Efficient Local Dynamic Voltage Scaling. In *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation*, pages 556–565, 2007. 6.4.1, 8.4.2.1

[NBS+02] Achim Nohl, Gunnar Braun, Oliver Schliebusch, Rainer Leupers, Heinrich Meyr, and Andreas Hoffmann. A universal technique for fast and flexible instruction-set architecture simulation. In *DAC '02: Proceedings of the 39th annual Design Automation Conference*, pages 22–27, New York, NY, USA, 2002. ACM. 3.1

[NC05] Siva G. Narendra and Anantha Chandrakasan. *Leakage in Nanometer CMOS Technologies (Series on Integrated Circuits and Systems)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005. 2.6

[PG03] Frederic Petrot and Pascal Gomez. Lightweight implementation of the posix threads api for an on-chip mips multiprocessor with vci interconnect. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 20051, Washington, DC, USA, 2003. IEEE Computer Society. 4.8.1.1

[PGR+08] Katalin Popovici, Xavier Guerin, Frederic Rousseau, Pier Stanislao Paolucci, and Ahmed Amine Jerraya. Platform-based software design flow for heterogeneous MPSoC. *ACM Trans. Embed. Comput. Syst.*, 7(4):1–23, 2008. 2.3.2

[PJ07] Katalin Popovici and Ahmed Amine Jerraya. Simulink based hardware-software codesign flow for heterogeneous MPSoC. In *SCSC: Proceedings of the 2007 summer computer simulation conference*, pages 497–504, San Diego, CA, USA, 2007. Society for Computer Simulation International. 2.3.1

[PPB02] Pierre Paulin, Chuck Pilkington, and Essaid Bensoudane. Stepnp: A system-level exploration platform for network processors. *IEEE Des. Test*, 19(6):17–26, 2002. 3.1

[SBR05] Jurgen Schnerr, Oliver Bringmann, and Wolfgang Rosenstiel. Cycle Accurate Binary Translation for Simulation Acceleration in Rapid Prototyping of SoCs. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 792–797, Washington, DC, USA, 2005. IEEE Computer Society. 3.1

[SCK+93] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. Binary translation. *Commun. ACM*, 36(2):69–81, 1993. 2.4

[She04] Ethier Sheridan. Application-Driven Power Management for In-car Telematics and Infotainment Devices. Technical report, 2004. 3.3.3

[SHR03] Jurgen Schnerr, Gunter Haug, and Wolfgang Rosenstiel. Instruction Set Emulation for Rapid Prototyping of SoCs. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 10562, Washington, DC, USA, 2003. IEEE Computer Society. (document), 3.1, 3.1

[SLM06] Louis Scheffer, Luciano Lavagno, and Grant Martin, editors. *EDA for IC System Design, Verification, and Testing*. CRC Taylor & Francis, 1 edition, March 2006. Section II, System Level Design. 2.3

[soc] Soclib project. http://soclib.lip6.fr. 4.8.1.2

[SR03] Saowanee Saewong and Ragunathan (Raj) Rajkumar. Practical voltage-scaling for fixed-priority rt-systems. In *RTAS '03: Proceedings of the The 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, page 106, Washington, DC, USA, 2003. IEEE Computer Society. 3.3

[SRS98] John A. Stankovic, Krithi Ramamritham, and Marco Spuri. *Deadline Scheduling for Real-Time Systems: Edf and Related Algorithms*. Kluwer Academic Publishers, Norwell, MA, USA, 1998. 2.7.1

[SYS] Open SystemC Initiative homepage. http://www.systemc.org. 2.2, 2.3

[Tan84] Andrew S. Tanenbaum. *Structured Computer Organization*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1984. 2.1.2

[tha] Thales Group. http://www.thalesgroup.com/. 4.8.5.3

[vdWdKH+04] Pieter van der Wolf, Erwin de Kock, Tomas Henriksson, Wido Kruijtzer, and Gerben Essink. Design and programming of embedded multiprocessors: an interface-centric approach. In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 206–217, New York, NY, USA, 2004. ACM. 2.3.2

[VGS+03] Ankush Varma, Brinda Ganesh, Mainak Sen, Suchismita Roy Choudhury, Lakshmi Srinivasan, and Bruce Jacob. A control-theoretic approach to dynamic voltage scheduling. In *CASES '03: Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 255–266, New York, NY, USA, 2003. ACM. 3.3.1

[VKS00] Diederik Verkest, Joachim Kunkel, and Frank Schirrmeister. System level design using C++. In *DATE '00: Proceedings of the conference on Design, automation and test in Europe*, pages 74–83, New York, NY, USA, 2000. ACM. 2.3.1

[vM96] Vojin Živojnovic and Heinrich Meyr. Compiled HW/SW co-simulation. In *Proceedings of the 33rd annual conference on Design automation*, pages 690–695, Las Vegas, Nevada, United States, 1996. 4.4.2, 8.2.2.2

[WHPZ87] L.-T. Wang, N. E. Hoover, E. H. Porter, and J. J. Zasio. SSIM: a software levelized compiled-code simulator. In *DAC '87: Proceedings of the 24th ACM/IEEE Design Automation Conference*, pages 2–8, New York, NY, USA, 1987. ACM. 3.2, 3.2

[WKL+04] Andreas Wieferink, Tim Kogel, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, Gunnar Braun, and Achim Nohl. A System Level Processor/Communication Co-Exploration Methodology for Multi-Processor System-on-Chip Platforms. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 21256, Washington, DC, USA, 2004. IEEE Computer Society. 3.1

[WM90] Zhicheng Wang and Peter M. Maurer. LECSIM: a levelized event driven compiled logic simulation. In *DAC '90: Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 491–496, New York, NY, USA, 1990. ACM. 3.2, 3.2

[WWDS94] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for Reduced CPU Energy. In *Proceedings Symposium on OS Design and Implementation*, pages 13 – 23, 1994. 3.3.1, 3.1, 3.3.1, 6.1

[XMM05] Rubin Xu, Daniel Mosse, and Rami Melhem. Evaluating a dvs scheme for real-time embedded systems. In *Second International Power-Aware RT Systems Workshop, PARC '05*, 2005. 3.3

[YBB+03] Sungjoo Yoo, Iuliana Bacivarov, Aimen Bouchhima, Yanick Paviot, and Ahmed A. Jerraya. Building Fast and Accurate SW Simulation Models Based on Hardware Abstraction Layer and Simulation Environment Abstraction Layer. page 10550, 2003. 2.1.2

[YCHK07] Chuan-Yue Yang, Jian-Jia Chen, Chia-Mei Hung, and Tei-Wei Kuo. System-level energy-efficiency for real-time tasks. In *ISORC '07: Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 266–273, Washington, DC, USA, 2007. IEEE Computer Society. 3.3

[YWM+01] Peng Yang, Chun Wong, Paul Marchal, Francky Catthoor, Dirk Desmet, Diederik Verkest, and Rudy Lauwereins. Energy-aware runtime scheduling for embedded-multiprocessor SOCs. *IEEE Des. Test*, 18(5):46–58, 2001. 3.3

Publications

International Conferences and Journals

[1] M. Gligor, N. Fournel, and F. Pétrot. Using binary translation in event driven simulation for fast and flexible mpsoc simulation. In *CODES+ISSS '09: International Conference on Hardware-Software Codesign and System Synthesis*, pages 71–80, New York, NY, USA, 2009. ACM.

[2] M. Gligor, N. Fournel, F. Pétrot, F. Colas-Bigey, A.-M. Fouilliart, P. Teninge, and M. Coppola. Practical design space exploration of an H264 decoder for handheld devices using a virtual platform. In *PATMOS '09: Power and Timing Modeling, Optimization and Simulation*. IEEE Computer Society, 2009.

[3] M. Gligor, N. Fournel, and F. Pétrot. Adaptive dynamic voltage and frequency scaling algorithm for symmetric multiprocessor architecture. In *DSD '09: Euromicro Conference on Digital System Design*, pages 613–616. IEEE Computer Society, 2009.

[4] L. Kriaa, A. Bouchhima, M. Gligor, A.-M. Fouilliart, F. Pétrot, and A.-A. Jerraya. Parallel programming of multi-processor SoC: A HW–SW interface perspective. *International Journal of Parallel Programming*, 36(1):68–92, 2008.

[5] L. Kriaa, A. Houari, M. Gligor, A. Bouchhima, F. Pétrot, and A.-A. Jerraya. Low power oriented hardware dependent software implementation in MPSoC architectures. In *IEEE-NEWCAS '07*. IEEE Computer Society, 2007.

[6] M. Gligor, L. Kriaa, A. Bouchhima, F. Pétrot, A.-A. Jerraya, and A.-M. Fouilliart. Abstract platform models for early SW IP integration in MPSoC design. In *IDT '06: International Design and Test Workshop*. IEEE Computer Society, 2006.

Stratégies de simulation rapides et algorithme adaptatif de contrôle de la tension et de la fréquence pour les MPSoCs basse consommation

Résumé - Les Systèmes sur Puce (SoC) ont vu leurs capacités en constante augmentation ce qui leur permet ainsi qu'aux applications s'exécutant dessus de devenir de plus en plus complexes grâce au pouvoir d'intégration de la technologie. Beaucoup de ces appareils fonctionnent sur batterie, mais puisque la technologie des batteries ne suit pas la même progression que l'intégration, à la fois le logiciel et le matériel de ces appareils doivent être économes en énergie. Nous proposons dans cette thèse un algorithme logiciel qui cherche à reduire la consommation énergétique en modifiant la fréquence et la tension des processeurs lorsque l'utilisation du système le permet. Cet algorithme n'a besoin d'aucune information sur les applications. Afin de tester et de déterminer l'efficacité de l'algorithme d'économie d'énergie proposé, nous avons besoin de plateformes de simulation rapides et précises qui supportent le changement de fréquence pour chaque processeur ou sous-système. Le bon niveau d'abstraction pour estimer la consommation d'énergie par la simulation n'est pas évident. Nous avons premièrement défini une stratégie de haut niveau de simulation qui combine la précision des simulateurs orientés matériel à la vitesse des simulateurs orientés comportement. Lorsque des estimations plus précises sont nécessaires, une simulation cycle accurate/bit accurate doit être utilisée. Toutefois, pour accélérer la simulation, des stratégies d'ordonnancement statique non compatibles avec le DVFS sont utilisées. Nous avons défini deux nouvelles approches supportant le DVFS dans ce contexte.

Mots clés : MPSoC, codesign, Simulation, Traduction binaire, Ordonnancement statique, Fréquences multiples, Algorithme d'économie d'énergie, DVFS

Fast Simulation Strategies and Adaptive DVFS Algorithm for Low Power MPSoCs

Abstract - SoC (System on Chip) devices have seen their capabilities increasing continuously allowing these devices and the applications running on them to become more and more complex thanks to the integration technology. Many of these devices operate unplugged, but as the battery technology does not scale with integration, both the software and the hardware of these devices must be energy efficient. We propose in this thesis a software algorithm that tries to save energy by modifying the processors frequencies and voltage when the system utilization permits. This algorithm does not need any input from applications. In order to test and determine the effectiveness of the proposed energy saving algorithm we need fast and accurate simulation platforms that support individual frequency change for each processor or subsystem. The right level of abstraction for estimating power consumption by simulation is not obvious. We firstly defined a high level simulation strategy that combines the accuracy of the hardware focused simulators with the speed of the behavior focused simulators. When more accurate estimations are required, a cycle accurate/bit accurate simulation must be used. However, to accelerate simulation, static scheduling strategies not compatible with DVFS are used. We defined two new approaches for supporting DVFS in this context.

Keywords: Multiple Processors System on Chip, Codesign, Simulation, Binary translation, Static scheduling, Multiple frequencies, Energy saving algorithm, DVFS

Adresse : Laboratoire TIMA, 46 Avenue Félix Viallet, 38031 Grenoble Cedex, France
ISBN : 978-2-84813-157-3