UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministérial : 7 août 2006

Présentée par

Fernando Akira ENDO

Thèse dirigée par **Henri-Pierre CHARLES**

préparée au sein **du Laboratoire Infrastructures Atelier Logiciel pour Puce, CEA Grenoble**
et de **l'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Génération dynamique de code pour l'optimisation énergétique

Thèse soutenue publiquement le **18 Septembre 2015**,
devant le jury composé de :

M. Frédéric PÉTROT
Professeur au Grenoble Institute of Technology, Président
M. Florent DE DINECHIN
Professeur à l'INSA de Lyon, Rapporteur
M. Paul KELLY
Professeur à l'Imperial College London, Rapporteur
M^me^ Karine HEYDEMANN
Maître de conférences à l'Université Pierre et Marie Curie, Examinatrice
M. Henri-Pierre CHARLES
Directeur de recherche au CEA LIST, Directeur de thèse
M. Damien COUROUSSÉ
Ingénieur chercheur au CEA LIST, Grenoble, Co-Encadrant de thèse

Dedico esta tese a minha família, que sempre me apoiou.

Agradecimentos

Gostaria de agradecer a todo pessoal do laboratório, que me acolheu durante esses três anos de doutorado, e também ao CEA pelo financiamento desta tese, assim como aos integrantes da banca examinadora, em especial aos examinadores da dissertação, Dr. Florent de Dinechin e Dr. Paul Kelly, por suas sugestões e correções.

Gostaria de agradecer especialmente aos meus amigos Fayçal Benaziz e Thibault Cattelani, que me ajudaram a revisar e corrigir o resumo em francês da tese, e também Alexandre Aminot, Ivan Llopard, Laurentiu Trifan, Thierno Barry, Tiana Rakotovao e Victor Lomüller, que revisaram e corrigiram meus artigos, que por sua vez foram integrados à tese.

Gostaria também de agradecer à UNICAMP, ao programa BRAFITEC e ao INSA de Lyon, sem os quais não teria tido a oportunidade de realizar um intercâmbio na França e obter um diploma francês, que facilitou minha candidatura ao doutorado.

Finalmente, gostaria de agradecer à Phi Innovations, pois grande parte do conhecimento que aprendi nessa empresa, incluindo a BeagleBoard-xM que recebi de presente, foram úteis ao desenvolvimento técnico e científico de meus trabalhos.

Contents

Part I

Thesis

Chapter 1

Introduction

Since the past decade, the energy consumption in high-performance processors is limiting the performance growth expected from transistor scaling. During three decades, reducing the size of transistors also reduced their energy consumption, resulting in constant power densities and exponential growth of performance per watt. Transistor scaling under these conditions is known as Dennard scaling [53]. Today, even if transistors are still becoming smaller in new generations of integrated circuits (ICs), their energy consumption is almost not scaling down anymore. With an almost constant energy consumption per transistor, the increasing number of transistors results in an exponential growth of the total power dissipations of a chip. Figure 1.1 illustrates this phenomenon. Under such conditions, processor designers must limit the power budget of chips to avoid excessive dissipations. Because of this power restriction and its consequences in the performance of computing systems, it is known as the power wall. As a consequence, a decade ago, the "free" performance growth obtained by increasing CPU clock reached a limit and processor designers had to switch from single-cores to (homogeneous) multi-cores.

Today, homogeneous multi-cores in server-class processors are facing another issue: the dark silicon [102]. Because of high power densities and thermal problems, only a fraction of transistors in ICs

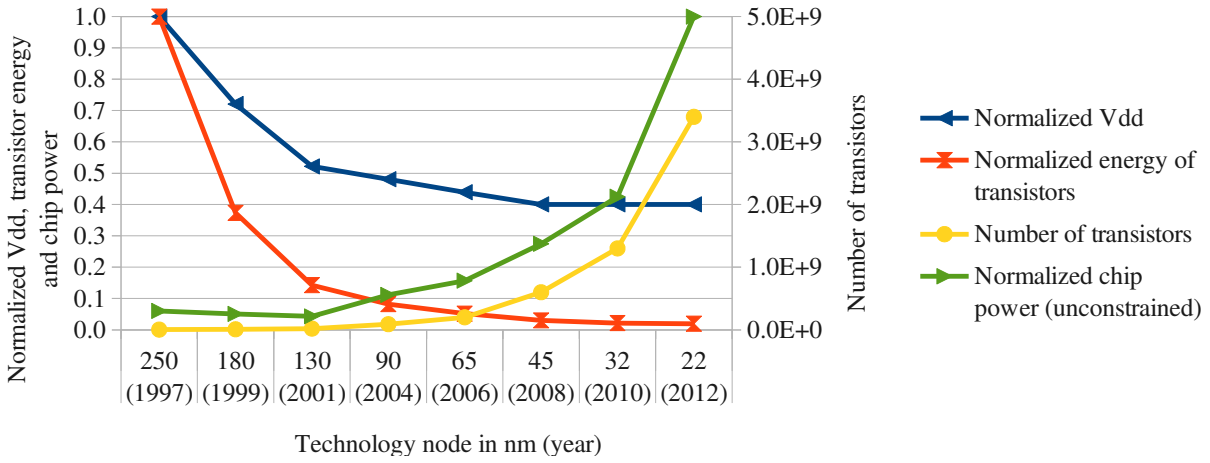


Figure 1.1: Trends of transistor technologies and impact on the total power dissipation of chips. Data from Dreslinski et al. [56] and HiPEAC Vision 2015 [59].

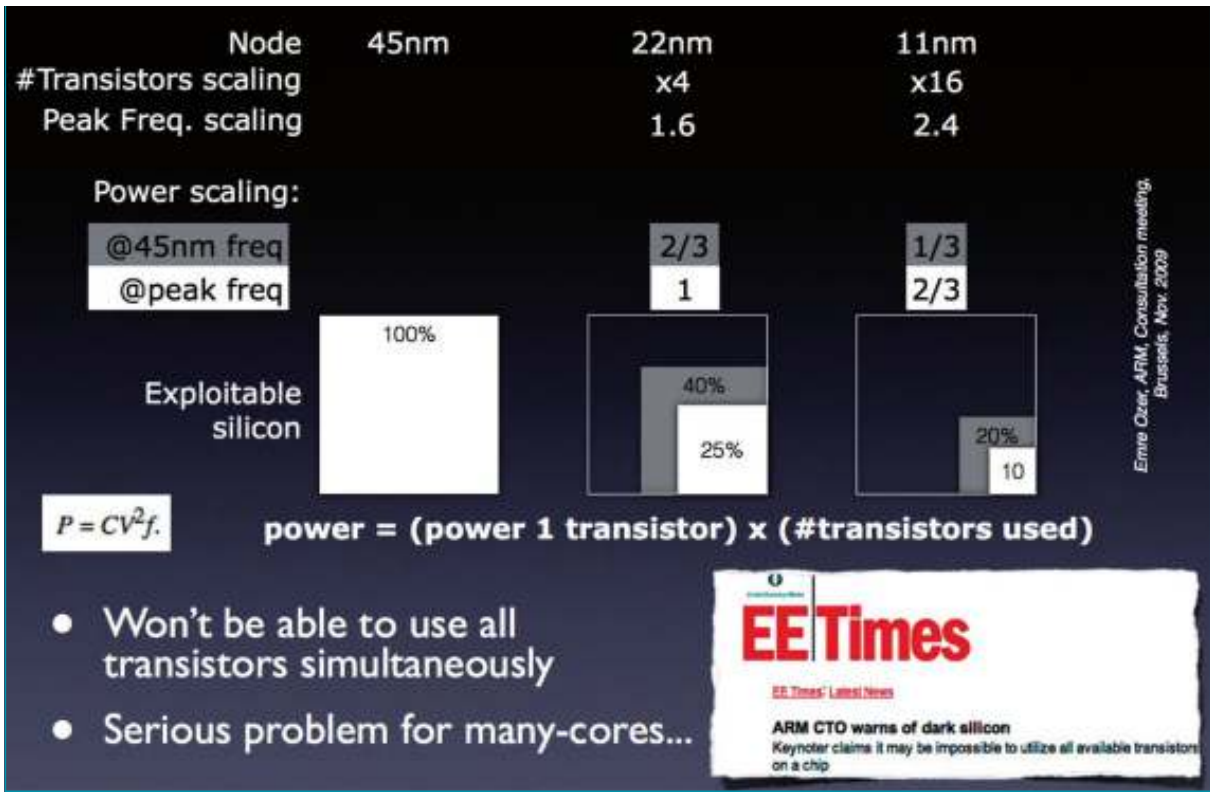


Figure 1.2: Dark silicon: the fraction of transistors that can be powered simultaneously reduces as transistor technologies advance. From The HiPEAC Vision for Advanced Computing in Horizon 2020 [58].

can be powered simultaneously, and this fraction is becoming smaller in each new generation. Figure 1.2 illustrates this problem. It is expected that dark silicon will dominate in CPU-like ICs between 2016 and 2021 [65]. Given that relying only on transistor scaling will not allow to sustain the performance improvement of computing systems, new architectural and micro-architectural designs are needed if we want to avoid a performance stagnation in the next decades, before new transistor technologies become mainstream [82]. Studies suggest that heterogeneous multi- or manycores coupled to accelerators are one of the solutions to keep the performance growth expected from transistor scaling [32]. Traditionally, processor designers invested the maximum number of the transistors to 90 % of the workload and the remaining to special cases. This approach is called 90/10 optimization. Now on, with dark silicon, hardware specialization is need. Instead of the 90/10 approach, investing 10 % of transistors to accelerate 10 % of cases, and another 10 % of transistors to another 10 % of cases, and so on, is more interesting, because the improved energy-efficiency of specialized hardware allows to increase the total throughput of the system. This approach is called 10×10 [46]. By 2020, processors will likely have hundreds to thousands of heterogeneous cores, possibly specialized to different tasks [73].

High-performance embedded cores have already a considerable degree of heterogeneity, for example the ARM architecture defines small in-order cores as the Cortex-A5 and A7, mid-size ones as the A8, A9, A12 and A17, up to big out-of-order cores as the A15, not to mention some 64-bit counterparts as the A53, A57 and A72 respectively. These base designs can be synthesized with different transistor technologies and have varying sizes and types of resources (pipeline buffers, number of cache levels and sizes, to name a few), and completely customized core implementations also exist. As high-performance embedded processors are also suffering from the power wall, and dark silicon may soon com-

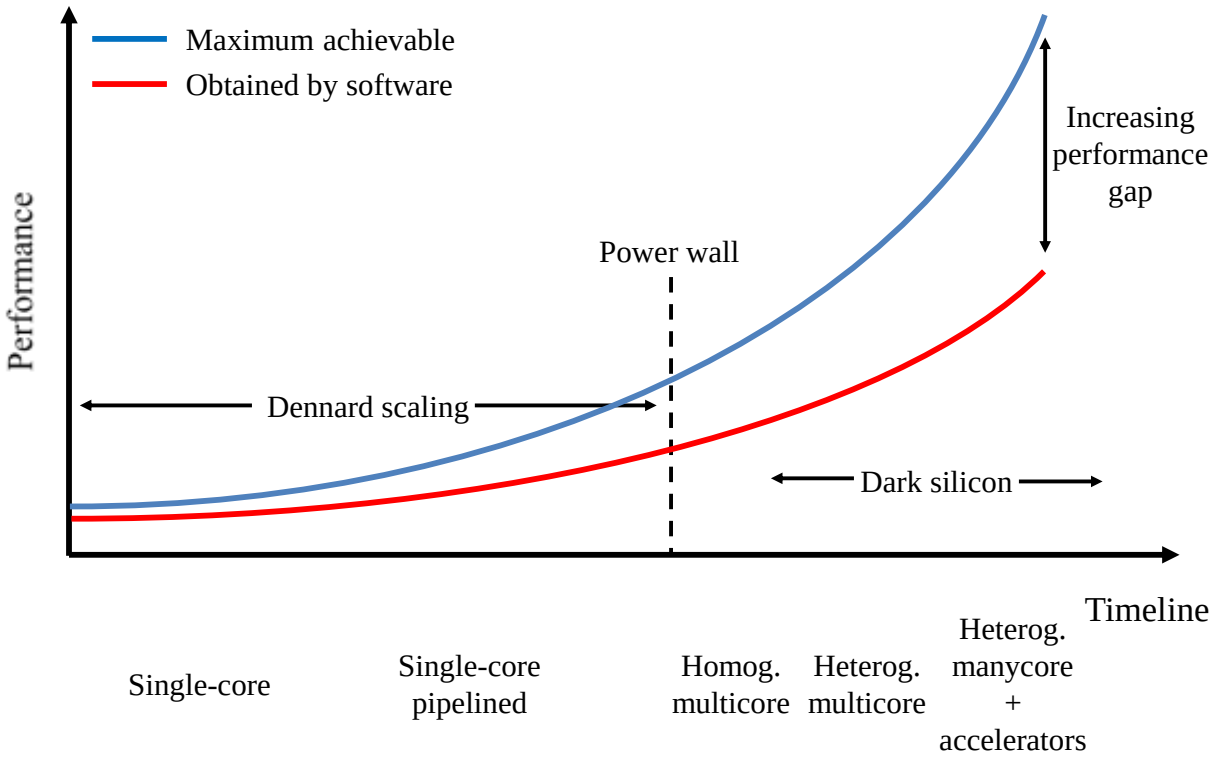


Figure 1.3: Increasing gap over time between the performance obtained by software and the maximum achievable performance in computing systems.

pletely change embedded architectures, we may expect increasing complex designs of heterogeneous multi/manycores. Recently, ARM released a heterogeneous multi-core system (the big.LITTLE design), where applications can be switched between two ISA-compatible cores, with different power and performance trade-offs [71]. Although it improves energy efficiency at low performance demand, the core heterogeneity solution inside a system on chip (SoC) creates new performance optimization challenges. For instance, if an application is compiled and optimized to a target core A, when the application is scheduled to run in a core B, the performance may not be as good as if the application was optimized to the core B, because of the differences in the pipeline implementations.

The performance gap between what software can extract from the hardware and the maximum achievable performance will increase in the heterogeneous multi/manycore era, and in the long term run-time approaches may be the only way to improve energy efficiency [32]. Figure 1.3 illustrates this increasing performance gap. The programming and performance portability challenges in such complex systems get further complicated with different cores in a processor having different ISAs and accelerators.

Static auto-tuning has been employed to increase performance portability and to extract near-optimal hardware performance comparable to manually tuned code. This approach has been successfully used in the domains of linear algebra and signal processing to cope with the architectural complexity of modern processors, by exploring a space of algorithm implementations [73]. However, whenever the target core or running conditions change, the code should ideally be auto-tuned again to the new environment, because statically auto-tuned code has usually poor performance portability when migrating between micro-architectures [3]. When the execution environment is not fixed at compile time, online auto-

tuning is a possible solution to improve performance portability. Nevertheless, only few work addressed online auto-tuning [24] and pushing auto-tuning to run time is very challenging [73], because evaluating the performance of various program versions by directly executing them in the target machine is unlikely to scale to large numbers of cores or to large programs.

This thesis studies machine code adaption to different micro-architectures, through run-time auto-tuning. Current trends in compiler research go toward the parallelization of programs to better use the growing number of available cores in processors. The proposal of this thesis is rather complementary to this trend. In other words, this work addresses the run-time code optimization of programs (multi-threaded or not), with the aim of improving the performance portability between different processors or various cores with different micro-architectural implementation inside a future heterogeneous many-core. It is likely that in such a future manycore, processes may be scheduled in any core or cluster of cores, depending on the application phases, the location of the data to be processed and dynamic system behaviors, as resource allocation, among other scheduling optimizations from the operating system.

One interesting question that this thesis tries to answer is if run-time auto-tuning in simpler and energy-efficient cores can obtain similar performance of statically compiled code run in more complex and hence power-hungry cores. The aim is to compare the energy and performance of in-order and out-of-order designs, with the same pipeline and cache configurations, except for the dynamic scheduling capability. This study would tell us at what extent run-time auto-tuning of code can replace out-of-order execution. However, given that commercial in-order designs have less resources than out-of-order ones (e.g., smaller caches, branch predictor tables), we need a simulation framework to perform such an experiment.

The objective of run-time auto-tuning is to perform machine- and input-dependent code optimizations at run-time, which otherwise would not be possible or convenient with a static approach.

Towards this objective, I analyzed the impact of code optimizations in ARM processors, such as instruction scheduling, vectorization, code specialization and loop unrolling. These experiments gave me good ideas of auto-tuning possibilities, because of the observed performance asymmetries of code optimizations in two ARM processors.

I studied the pipeline implementations of in-order and out-of-order cores present in ARM and Alpha processors and I also studied the pipeline models from two micro-architectural simulators, gem5 and McPAT. These studies permitted me to understand the behavior of machine code interacting with pipeline components, to improve some modeling aspects of both simulators, and to develop a simulation framework to study ARM core heterogeneity.

The simulation framework can estimate the area, energy and performance of ARM CPUs. It was employed to analyze the suitability of the proposed run-time auto-tuning tool to adapt machine code to various pipeline designs, including in-order and out-of-order cores, with single-, dual- and triple-issue capabilities, with varying pipeline depths and number of functional units.

1.1 Thesis contribution

The main contribution of this thesis is the methodology and the proof of concept that run-time auto-tuning of short-running computing kernels is viable in general purpose embedded-class processors.

1.1.1 Run-time code generation and auto-tuning for embedded systems

Towards the objective of implementing a run-time auto-tuning framework for embedded-class processors, I ported deGoal, a run-time code generator, to the ARM Thumb-2 ISA, including the FP and single instruction, multiple data (SIMD) extensions, and extended it with a configurable execution stage model and dynamic selection of code generation options. I performed a preliminary validation by evaluating eight kernel configurations with deGoal.

The proposed run-time auto-tuning approach is demonstrated in two benchmarks, and both scalar and vectorized codes are compared. The benchmarks were executed with three different input sets and run in several real and simulated ARM cores.

In order to prune the search space, a technique that divides the auto-tuning process into two phases is presented.

This thesis demonstrates that online auto-tuning of a computing kernel can considerably speedup a CPU-bound application, which runs during hundreds of milliseconds to a few seconds in embedded cores, even compared to manually vectorized code. I also show that the run-time overhead of code generation and space exploration is very low, and that the performance of the dynamically auto-tuned kernel implementations were very close to those of the best versions obtained by an extensive search in a off-line setting.

1.1.2 Micro-architectural simulation of ARM cores

To evaluate the capability of micro-architectural code adaption of the proposed run-time auto-tuning approach, I developed a simulator that can estimate the performance and energy consumption of heterogeneous ARM cores. Most of the previous work studying core asymmetry employed other ISAs instead of ARM. In the embedded context the ARM ISA is widely used.

However, in the beginning of this thesis, a simulation framework of power and performance of in-order and out-of-order ARM cores did not exist. The gem5 and McPAT simulators were taken as the starting point for the framework. Therefore, this thesis contributes by describing in detail a simulation platform for heterogeneous in-order and out-of-order ARM cores. I enhanced gem5 to better model cores from the ARM Cortex-A series and McPAT to better model core heterogeneity. The simulation framework was validated in two phases: the performance estimations were compared to real ARM hardware, and the energy estimations were validated in a relative way by simulating the energy and performance trade-offs of big.LITTLE CPUs.

This thesis demonstrates that gem5 can simulate ARM cores with considerable lower timing errors than similar micro-architectural simulators, and that a simple McPAT model extension provides accurate area estimation and acceptable energy/performance trade-offs estimations of big.LITTLE CPUs, when coupled to gem5.

1.2 Thesis organization

This thesis begins by presenting the state of the art in Chapter 2, which surveys the following topics: the energy reduction techniques integrated into compilers; micro-architectural simulators of embedded processors; run-time code generation, optimization and specialization; and online auto-tuning.

Chapter 3 details the development and validation of a micro-architectural simulator of heterogeneous ARM cores, based on the simulation frameworks gem5, for performance, and McPAT, for power and area estimation. First, both simulators are presented, including the output conversion of gem5 to input for McPAT. Then, the performance, area and relative energy/performance validations against real ARM cores are described. Finally, an example of architectural/micro-architectural exploration is presented, just before discussing the scope and limitation of the proposed simulation framework and the conclusion.

Chapter 4 presents deGoal, a run-time code generator, and details its porting and validation to the ARM Thumb-2 ISA and the new features developed during this thesis. It begins by presenting the deGoal language with an example of a deGoal code compared to equivalent C source codes. Then, the porting to ARM and the new features are introduced, mostly focusing on the implementations that are needed by a run-time auto-tuner. A preliminary validation is presented, followed by experiments of dynamic code specialization. At the end of this chapter, the scope and limitations of deGoal and its validation are discussed, before the conclusion.

Chapter 5 describes the methodology and proof of concept of a run-time auto-tuning framework for embedded systems. First, I argue why run-time auto-tuners should be developed for embedded systems, following by a motivational example. Then, the methodology and experiments around two case studies in two ARM cores and 11 simulated cores are detailed. Finally, the scope, limitations and conclusion of this study are discussed.

Chapter 6 summarizes the achievements and contributions, and discusses future prospects of this thesis.

Chapter 2

State of the art

This chapter surveys the state of the art of the following topics: sources of energy consumption in ICs, energy reduction techniques integrated into compilers, micro-architectural embedded processor simulation and run-time code optimizations.

2.1 Sources of energy consumption in ICs

The power dissipation of complementary metal-oxide-semiconductor (CMOS) circuits can be divided in two parts: the static and dynamic powers. The static or leakage power corresponds to the power that is always dissipated whenever transistors are powered even if there is no activity in the circuit. On the other hand, the dynamic power is the dissipation produced whenever the state of transistors switches between logical zero and one.

2.1.1 Static or leakage power

Static or leakage power has several sources, the two main ones being the sub-threshold and the gate leakages. The sub-threshold leakage originates from the current that passes through the channel even if the transistor is off, and it is the main contributor to the total static power. Eq. 2.1 describes its behavior [114], where V is the source voltage, I_0 is a constant which depends on the geometrical dimensions and technology of the transistor, q and k are respectively the elementary charge and the Boltzmann constant, α is a number greater than 1, V_{th} is the threshold voltage and T the temperature.

$$P_{sub} = V I_0 \exp\left(\frac{-qV_{th}}{\alpha kT}\right) \tag{2.1}$$

Eq. 2.1 shows us that as the temperature rises, the sub-threshold leakage grows in magnitude, which in turn further increases the temperature of the circuit. This dangerous phenomenon is known as thermal runaway.

The gate leakage, as the name says, comes from the current that passes through the gate toward the body of the transistor. This current depends on the thickness of the insulator at the gate.

Historically, the static power had a negligible contribution to the total power dissipation of CMOS circuits. Only recently, with deep sub-micron transistor technologies, it became a trouble to IC designers [114].

2.1.2 Dynamic power

The dynamic power is mainly described by the charging and discharging cycles of load capacitances, whenever the transistor switches between logical zero and one. Eq. 2.2 describes the dynamic power behavior, where a is the activity factor of the circuit, C is the total capacitance, V is the source voltage and f is the clock frequency.

$$P_{dyn} = aCV^2f \tag{2.2}$$

The dynamic power has also a small dependence on the temperature. Spacial temperature gradients results in small delays between signals, for example, that are the inputs of a logic gate, causing then useless transistor switches inside the component. Nonetheless, those events have a small contribution to the total dynamic power [114].

The dynamic power comes from various components inside a computing system, which have different energy costs per operation. For example, in nVIDIA chips at 28 nm, an integer and floating-point operation consumes 1 and 20 pJ, while accesses to the DRAM cost 16 nJ [104]. These numbers depend on the architecture of the processor and its transistor technology. In addition, depending on the application behavior, the CPU may be required more than the memory, or vice-versa.

2.2 Energy reduction techniques integrated into compilers

The energy consumption can be reduced by software in several layers: drivers, operating systems, libraries, compilers and applications. This thesis is interested in dynamic approaches at the compiler level. This section begins by presenting power and energy reduction techniques in software, and then techniques integrated into compilers. Dynamic compilation historically addressed performance, its usage to reduce the energy consumption of processors is very recent [134]. As a consequence, this section mainly presents the state of the art of techniques employed in static compilers and dynamic techniques not necessarily embedded into compilers.

2.2.1 Energy reduction in software

The energy consumption can be reduced in software by directly acting in the dynamic or static power dissipations. As the energy consumption depends on the power and the elapsed time, speeding up the execution can also reduce the energy. First, power techniques are presented, then I argue why energy consumption was historically addressed by reducing execution time.

2.2.1.1 Dynamic power

From Eq. 2.2, software techniques have four ways to decrease the dynamic power by reducing: the active load capacitance, transistor switches, the clock frequency and the source voltage.

The classic power reduction technique is the dynamic frequency and voltage scaling (DVFS). If the source voltage is reduced in a processor, the result is a slowdown in the transmission of signals. By doing so, it may cause wrong signaling, then the frequency of circuit operation must also be reduced to avoid errors. DVFS is very advantageous, roughly degrading the performance proportionally to the cubic reduction of dynamic power consumption [81]. Thanks to its effectiveness, the DVFS is also employed to avoid thermal problems in processors. Notwithstanding, power and thermal reduction policies may sometimes be conflicting [81].

The activity of the transistors in a processor component depends on successive voltage switches at the gate of transistors placed in the input of the circuit. The set of those voltages is called the input set of a circuit. The input set of pipeline components usually have a direct relationship with the encoding of instructions. Therefore, some compiler techniques try to reorder instructions in order to reduce the switching of encoded bits between neighbor instructions [122]. I will call this technique *switching minimization*.

Two other techniques can reduce the dynamic power by reducing the circuit activity: the strength reduction or pattern matching, and the ISA choice. In the first, costly operations are replaced by simpler ones (e.g., replacing multiplications by left-shifts). In the second, choosing a more compact ISA, such as the variable instruction size ARM Thumb-2 over the fixed ARM 32-bits, can sometimes reduce the activity and hence the power dissipation in the instruction cache, because fewer bytes need to be fetched to execute a code. However, the ISA choice has complex trade-offs, because some operations in a compact ISA may need to be decompressed [83], require more instructions, slower down the processor and even slightly increase branch mispredictions [7, Section 16.4.3], which in turn could increase the power consumption.

2.2.1.2 Static power

Eq. 2.1 tells us that there are four ways to reduce the static power consumption of circuits: reduce the supply voltage, shut down and cool down circuits, and change transistor parameters.

The source voltage can be reduced by two means: putting parts or the whole processor into sleep, or using DVFS. In the context of an application, usually only parts of the processor can be put to sleep or be completely shut down. The latter technique is known as power supply gating or simply power gating.

The temperature of the processor and its components can also be managed by software, for instance by: executing cool loops in simultaneous multithreading (SMT) processors [48], balancing load [101, 105] and inserting NOP instructions [116].

The adaptive body biasing (ABB) is a technique that allows to dynamically change the threshold voltage. While the DVFS mostly acts in the dynamic power, the combined DVFS and ABB can further reduce the static power [2], because changes in the body bias voltage result in exponential variations of the static power (Eq. 2.1).

2.2.1.3 Energy

The energy consumption is obtained by integrating the instantaneous power consumption over a period of time. Even though the energy depends directly on the power, energy and power reduction policies have different objectives and may be conflicting. Some policies reduce the average instantaneous power dissipation over a period of time, but they may increase the energy consumption.

To further complicate the matter, battery discharge is not proportional to the energy consumed in the draining circuit. Pedram and Wu showed that the battery duration is maximized if the variance of the discharging current is minimized [110]. In simulation, they observed that the battery duration could be shortened by 20 % with a white noise distribution for the discharging current compared to the optimal case (constant current).

Considering constant power dissipation, the faster a program runs, the less energy it requires. This conclusion is usually simplified by removing the constant power restriction, and is commonly accepted as a rule in software development. There are some reasons that could explain this common belief: energy measurement is usually expensive or impractical, and usually there is a good correlation between energy and execution time, hence only execution time is usually taken as metric.

Embedding power sensors into SoCs is still a challenging task. At the level of cores, there are problems of calibration and performance degradation caused by the integration of current sensors [113]. At lower levels, practical limitations of the supply network make it difficult to directly measure run-time power consumption of individual blocks [114]. Recently, Intel introduced energy counters in some of its processors. They allow to estimate the energy consumption at intervals of one millisecond of the SoC or all cores together [77]. While these counters can be activated in desktop- and server-class commercial products, in ARM processors, such estimations are only possible in prototyping configurations, such as the energy counters provided by the ARM Versatile boards. Special energy measurement hardware may be externally employed, but this approach is not cost-effective even in the high-performance computing (HPC) domain [27]. Historically, this lack of energy measurements led run-time energy reduction techniques in software to be regrouped in four not necessarily disjoint classes:

- **Explicit energy reduction**: Techniques such as power/clock gating, low-power modes (e.g., DVFS) and switching minimization explicitly reduce the energy consumption, usually at the expense of some performance degradation.
- **Static/dynamic profiling**: Because of the lack of run-time energy measurements, static profiling is employed to feed and train energy models to the run-time phase. Dynamic profiling (energy or performance counters) can take into account the run-time environment and application phases to drive the decisions, but with the extra cost of the profiling overhead.
- **Energy-efficient hardware selection**: Switching program execution to a hardware with fewer resources can reduce the energy consumption, with some performance degradation as well.
- **Run as fast as possible**: Thanks to the usually good correlation between run-time and energy consumption and the availability of real-time clock (RTC) or cycle counters, reducing execution time is a commonly accepted way of reducing energy consumption in a fixed hardware configuration.

Examples of these techniques integrated into compilers are presented in the next section.

2.2.2 Compiler techniques

This section presents energy reduction techniques proposed to or integrated into compilers.

2.2.2.1 DVFS

Hsu and Kremer [76] were one of the first to evaluate DVFS algorithms with physical measurements. They presented the conception and implementation of a DVFS algorithm for compilers. The aim was to identify a source code region through static profiling, where DVFS could be applied with negligible performance loss.

Wu et al. [137] pushed intra-task DVFS to run-time. The main advantage of their dynamic approach was that the actual run-time environment is considered, instead of relying on static profiling. The main code characteristic searched is the memory boundedness, which varies depending on the architecture configuration, input set and application phases, which in turn may vary over time. Wu et al. estimated that their dynamic approach is two times better than a static one.

2.2.2.2 Switching minimization

Chang et al. [42] presented an energy model for a micro-controller (ARM7TDMI) that characterized the dependence between the energy consumption and the encoding of instructions, and relationships with pipeline stages. They found out that the energy consumption of pipeline stages is proportional (or inversely) to the number of '1' bits in the instructions. Differently from previous work, the opcode encoding did not play a very important role in the energy, but instead the number of different bits (Hamming distance) of register encodings and immediate values of neighbor instructions is proportional to the energy consumption. The proposed energy model could for example be employed in compilers to reallocate registers, relocate addresses and re-schedule instructions, in order to minimize the energy consumption.

Cooper et al. [51] proposed a compiler infrastructure that optimized code respecting an objective function. The aim was to explore various compiler phase ordering possibilities and find the order that minimizes the objective function. By setting the switching minimization as objective, they observed between 6 and 7% of energy saving compared to the default compiler sequence, in their preliminary results.

2.2.2.3 Clock/power gating and sleep modes

Clock and power gating are usually used in large structures of the processor, to maximize the energy savings. Example of such structures are the caches, register banks and functional units (FUs). A typical trade-off of these techniques is the energy saving versus the performance degradation, because once the circuits are gated, they need a given number of cycles to be operational again. Special instructions are usually needed by the compiler to power off or put structures into sleep.

Zhang et al. [141] presented a technique to reduce the leakage of instruction caches. A cache line can be powered off or put into sleep if the compiler determines that the line will not be accessed anymore or

only after a long time, respectively.

Ayala et al. [21] proposed a power-aware reconfiguration mechanism in the register file driven by a compiler. In their approach, after the register allocation the compiler inserts special instructions to put portions of the register file in a low-power mode (content is not lost).

You et al. [139] proposed a compiler analysis to determine when FUs could be turned off and to insert power gating instructions. The approach was implemented in a compiler and tested in an Alpha simulator.

Only few researchers tried to push those techniques to run time. Unnikrishnan et al. [133] were among the first to investigate dynamic recompilation for energy reduction. The reduced energy came from memory banks that were turned off depending on the current energy budget. To do so, they employed a specific compiler to generate various versions of functions that traded-off memory usage and performance. The compiler also integrated an energy estimator, in order to classify the functions by their energy consumptions. At run-time, the function versions were swapped following the changes of the energy budget.

Ukezono and Tanaka [131] proposed a run-time technique to reduce the leakage of L2 caches. They used a tool called Hybrid Dynamic Optimization System, which implements a hardware component that allows users to define event traps (the UDT — User Definable Trap). In their approach, the UDT was set up to analyze memory accesses. The trap routine was responsible for detecting L2 cache lines rarely accessed and to replace the corresponding load or store instructions by special ones that will only load the L1 cache and turn off the corresponding L2 cache line (inclusive caches are supposed).

2.2.2.4 Energy-efficient hardware selection

Since single-core processors hit the power wall in the last decade, researchers started to look for energy-efficient multi-core designs. Kumar et al. were one of the first to show the energy benefits of having different types of cores in a SoC, sharing the same ISA [84, 85].

In current big.LITTLE systems, the core switching is decided by hardware or the operating system (big.LITTLE task migration and big.LITTLE MP use models, respectively). Compiler hints and code structuring can also help and guarantee correct execution in some applications [118,123]. Core switching in heterogeneous-ISA multi-cores are even more challenging, and compiler support may be needed to keep the program state in an architecture-neutral form to reduce the switching overheads [55].

Current embedded platforms have heterogeneous computing units, such as cores, micro-controllers and digital signal processors (DSPs). Usually, only CPU resources are directly exposed to programmers, external ones are accessed through library interfaces. Chandramohan and O'Boyle proposed a compiler framework to automatically partition and map applications in the different computing resources of a MPSoC [41]. In their work, the partitioning is statically found through profiling. It would be even more challenging to push such approach to run-time, dynamically adapting a task partitioning as the system load varies.

2.3 The ARM architecture

The ARM architecture is used in a broad range of embedded systems, from low-power micro-controllers, such as the cores defined by the Cortex-M series, to high-performance embedded processors with memory management unit (MMU), as the cores defined by the Cortex-A series, which includes 32- and 64-bit processors.

Currently, there are three main instruction sets: the A64 (for 64-bit processors), A32 (a 32-bit ISA, once called ARM 32-bits) and T32 (a 32-bit ISA, with variable instruction widths). The T32 ISA, also known as Thumb-2, defines instructions whose widths are 16 or 32 bits, and extends a previous ISA called Thumb-1, which defined a 16-bit wide instructions. The A64 ISA has 31 general purpose registers and 32 SIMD registers, accessed as 128-, 64-, 32-, 16- or 8-bit registers [14, Section B1.2.1]. The A32 and T32 ISAs have 13 general purpose registers, and 32 SIMD registers of 64 bits, also aliased as 16 SIMD registers of 128 bits, which can be viewed as 32-, 16- or 8-bit elements. Thumb-1 instructions have access to only 8 general purpose registers and support neither FP nor SIMD instructions.

High performance cores from the Cortex-A series can embed FP and/or SIMD extensions, called VFP and Advanced SIMD, respectively. The Advanced SIMD architecture, implementations and supporting software is referred as NEON.

2.4 Embedded processor simulation

Simulation is mainly employed for three reasons:

- **The real experimental setup is very expensive or impossible**: For example in high-performance processors, obtaining reliable measurements of the behavior of core components without perturbing the circuit is impossible.
- **Experimental or theoretical research**: Some experiments suppose ideal conditions to simplify the analysis or to focus on the main problem. Other examples are the proof of a concept and the simulation of modified versions of existing hardware.
- **Prototyping**: Before synthesizing a complex circuit, prototyping through simulation shortens the process, avoiding simple project errors.

First, this section introduces the abstraction levels of processor simulation. Then, the state of the art of micro-architectural ARM simulators for performance and power estimation is presented.

2.4.1 Abstraction levels

Depending on the accuracy and complexity of the simulated software, six main abstraction levels are typically identified and described in the following.

Transistor The transistor is the basic unit of ICs. Simulation at this level produces very accurate results, but the simulation speed is prohibitive. Usually, only small circuits or components of processors are simulated, in order to model their behavior and calibrate a simulator with a higher abstraction level. For example SPICE was used by PTscalar [92] and eCACTI [98] to model the dependence of static power on temperature and to calibrate empirical models, respectively.

Logic gate and register-transfer level (RTL) Logic gates are composed of a few transistors, while combinatorial logic (such as counters, shifters and adders) and registers are typically composed of several logic gates. The abstraction level of a circuit implemented with combinatorial logic and registers is called RTL. The simulation speed at the gate level is around 1 and 100 Hz, and at the RTL, between 1 and 100 kHz [124]. Hardware description languages as VHDL and Verilog are used to describe circuits at both logic gate level and RTL. Low-level simulators whose abstraction goes up to the RTL are commonly called *detailed simulators*, and they usually simulate small processors or small pieces of code.

Architecture and micro-architecture Further increasing the abstraction level, micro-architectural simulation models the components of cores, going down to structures of pipeline stages, as instruction buffers, queues, register files and FUs. This level of abstraction is usually chosen to modify or to implement new processors or its components. For example, some of the compiler techniques presented in Section 2.2.2.3 were validated through simulation at this level. Micro-architectural simulation is typically mixed with architectural simulation, which models SoC components like caches, networks on chips (NoCs) and peripherals. This level of abstraction is used in cases where a large number of cores and their communications should be accurately modeled, but intra-core behavior can be neglected.

Instruction Instruction set simulators (ISSs) do not model processor components, only the instruction behavior is modeled. Usually, at this level of abstraction, a conversion table indicates the performance or energy consumed when each instruction is executed. Each entry in table should be calibrated based on measurements in a given hardware. Instructions may also be regrouped to speed up the simulation at the cost of being less accurate. Some ISSs are mixed with architectural modeling, for example in the work of Fournel, the energy of an instruction is decomposed in a base energy cost, the energies coming from the instruction itself, and other dissipations as bus/memory accesses and peripherals [66].

2.4.2 Micro-architectural performance simulation

This section presents two micro-architectural simulators of the ARM ISA: SimpleScalar and gem5.

2.4.2.1 SimpleScalar

SimpleScalar [35,36] is one of the first micro-architectural simulators to be proposed. It is a dynamic trace-driven simulator, in other words a simulation trace is firstly generated before feeding the models, but those traces are not saved into the disk. In the first version, the Portable Instruction Set Architecture (PISA) model based on MIPS was simulated. SimpleScalar has a simple design and a very fast simulation

speed. Nevertheless, because of its trace-driven design, SimpleScalar can not accurately simulate multi-cores [36].

There are various simulation models in SimpleScalar, the main six are:

- **`sim-fast`** and **`sim-safe`**: Both are functional simulators, but `sim-safe` verifies memory alignment and access permissions.
- **`sim-cache`** and **`sim-bpred`**: They estimate the micro-architectural statistics of caches and branch predictor, respectively, taking as input the functional simulation traces. These models do not estimate the timing of accesses, being adapted for instance to studies that evaluate miss rates.
- **`sim-outorder`**: Simulates an out-of-order processor, with load/store queues and the Register Update Unit [120], a structure that combines the physical register file, reorder buffer (ROB) and issue window [54]. Six pipeline stages are modeled: fetch, dispatch (decode and rename), issue, execute, writeback and commit. The issue width and in-order issue are configurable options.
- **`sim-alpha`**: Models the Alpha 21264, an out-of-order processor. It is more accurate than `sim-outorder`, but as a result it lacks flexibility and is difficult to extend.

`sim-outorder` and **`sim-alpha`** were compared to the Alpha 21264 processor [54], running ten benchmarks of the SPEC2000 suite. The absolute average errors measured in the simulation of the instructions per cycle (IPC) were 36.7 and 18.2 %, respectively.

In the 3.0 release and later, SimpleScalar supported the PISA, Alpha AXP, PowerPC and ARMv4 instruction sets. Timing models for the SA-1 pipeline were added, which permitted researches to simulate for example the Intel StrongARM SA-1110 processor [36].

2.4.2.2 gem5

gem5 [29] is a performance simulator, resulting from the merging of two previous projects: GEMS [99], for memory timing, and M5 [30], for accurate CPU modeling. Differently from a trace-driven simulator, gem5 focuses on timing accuracy and hence its micro-architectural models only execute instructions after all dependencies have been solved. The main ISAs supported are Alpha, ARM and x86, which can boot Linux in a full-system simulation.

Originally, four CPU models were supported, but a fifth and probably temporary model also exists:

- **AtomicSimple**: It is a functional single IPC model and is usually used to fast forward the simulation until the region of interest. Memory accesses are atomic.
- **TimingSimple**: As AtomicSimple, it tries to execute one instruction per cycle, but memory timing is simulated. This model is adapted to simulate simple in-order CPUs, and is used to warm caches for more complex models.
- **InOrder**: Models a configurable number of pipeline stages of a in-order core, as well as hardware threads. It is currently a deprecated model and will probably be replaced by the Minor model.

- **Minor**[1]: This is a temporary four-stage in-order model, currently supporting the ARM and Alpha ISAs, including the Linux boot.
- **O3**: Implements a seven-stage out-of-order pipeline (fetch, decode, rename, issue, execute, write-back, commit), based on the physical register file implementation from Alpha designs [88]. It also supports SMT.

Two kinds of simulation are supported:

- **System-call Emulation (SE)**: In this mode, most of the system calls are emulated by transferring them to the host operating system. Current limitations include the lack of translation lookaside buffer (TLB) statistics and multi-threading. Multi-cores can be simulated, but in this case each core should have only one workload statically assigned to it.
- **Full-System (FS)**: In this mode, a bare-metal environment is simulated. Interruptions, exceptions, privilege levels and I/O devices are simulated. It can boot unmodified Linux kernels [37], which can then support multi-threading libraries.

There are two models to simulate the cache hierarchy:

- **Classic**: This model comes from M5 and is easily configurable. It is adapted when the simulation focuses on system behaviors other than cache coherence. The MOESI protocol is used.
- **Ruby**: It is a heritage from GEMS and defines a domain specific language (DSL) for cache coherence protocols, called SLICC. Currently, the ARM ISA does not support this model.

gem5 is very flexible and parametrizable, it has around 70 core and 10 cache (Classic model) parameters. The ISA description is implemented through a DSL and is completely separated from the CPU models. A current limitation of gem5 is that the simulator runs as a single-threaded program. In a typical PC, the O3 model in FS simulation can run PARSEC workloads with an average rate of 280 kIPS (between 160 and 870 kIPS). It is estimated that the TimingSimple and Minor models are 9 and 2 times faster than the O3 model, respectively.[2]

The timing accuracy of gem5 was evaluated against real hardware. The TimingSimple model was configured and compared to a Cortex-A9, showing absolute errors between 1.4 and 18 % [37]. An enhanced O3 model was compared to a Cortex-A15, showing average absolute errors of 13 and 16 % when simulating SPEC CPU2006 and PARSEC benchmarks, respectively [72].

2.4.3 Micro-architectural energy simulation

This section presents not only energy simulators related to SimpleScalar and gem5, but also hardware accelerated models at the micro-architectural level.

[1] `http://www.mail-archive.com/gem5-dev%40gem5.org/msg11667.html` [Accessed: 17 April 2015]
[2] `http://www.mail-archive.com/gem5-dev%40gem5.org/msg11667.html` [Accessed: 17 April 2015]

2.4.3.1 Wattch

Wattch [33] was one of the first micro-architectural energy simulators. The energy consumption of processor components is estimated through the multiplication of their energy cost per access by the number of accesses to the component, produced by SimpleScalar. The estimations of the energy cost per access were based on the work of Wilton and Jouppi [136], and Palacharla et al. [109].

Processor component are divided in four main models:

- **Array structures**: Such as caches, register files, branch predictors.
- **Fully associative content-addressable memories (CAMs)**: Instruction queue, TLBs.
- **Combinatorial logic and wires**: FUs.
- **Clocking**: clocking buffers and wires.

Wattch originally only modeled the dynamic power. In addition, the 0.8 µm technology node was taken as reference, which is not adequate anymore to estimate the performance of current processors [90].

The peak power estimation were compared to a Pentium Pro and an Alpha 21264 processor. The absolute average errors found for the power breakdown per structure were between 10 and 13 %. On average, Wattch underestimates by 30 % the total peak power of the Pentium, Alpha and a MIPS R10000.

2.4.3.2 PTscalar

PTscalar [92] is an extension to SimpleScalar 3.00b, which also takes into account the influence of temperature on leakage power. An Alpha 21264 floorplan at 350 nm was taken as reference and rescaled to 65 nm. Although PTscalar was based on the Alpha design, the temperature and leakage models are architecture-independent.

The static power modeling depends on the dynamic voltage and temperature. The two main leakage sources were modeled (sub-threshold and gate leakage) taking as reference the MOSFET BSIM4 transistor model [40, 130]. The static power model was validated by comparing their estimations to results from SPICE. The average absolute error was less than 4 %.

2.4.3.3 Sim-Panalyzer

Sim-Panalyzer [132] is a porting of the `sim-outorder` model in SimpleScalar to the StrongARM SA1100. The power is estimated in a similar way as in Wattch.

2.4.3.4 CACTI-D

CACTI [128] is a tool that models the dynamic power, timing and area of caches. Both SRAM and logic process based DRAM (LP-DRAM) memory types are supported. CACTI-D extended its models to

include the commodity DRAM (COMM-DRAM) type, commonly known as main memories, allowing the simulation of a complete memory hierarchy [127]. Micro-architectural energy simulators directly or indirectly use CACTI models.

2.4.3.5 CAMP

CAMP (*Common Activity-based Model for Power*) [113] is an estimation technique of the activity and power of micro-architectural structures. By taking as input only nine measured activities in the circuit, it can estimate the power of most structures with an accuracy within 5 % of those obtained by a detailed micro-architectural simulator (the Asim framework [61]). The core power (from an Intel Core-like processor) was estimated with an error of 8 %.

The nine activity factors were chosen by analysing their correlation to other activity factors in the core. Then, a simple linear regression model is built. CAMP is fast enough to be integrated into hardware and provide run-time power estimations, or to accelerate design-time analysis.

2.4.3.6 PrEsto

PrEsto (*PoweR ESTimatOr*) [124] is a power modeling methodology capable of automatically generating power models for processor components. A detailed simulator is used to calibrate the models, which are then integrated into a FAST (FPGA-Accelerated Simulation Technologies) simulator [47].

A linear regression model is employed to estimate the power of components from activity factors in the core, as shows Eq. 2.3, where c_i are coefficients that capture the CV^2f factor of dynamic power, and s_i represent the activity factors of selected signals.

$$P = c_0 + c_1 s_1 + c_2 s_2 + \ldots + c_n s_n + c_{n+1}\left(s_1 s_2\right) + c_{n+2} s_1 s_3 + \cdots + c_{2^n}\left(s_1 s_2 \ldots s_n\right) \qquad (2.3)$$

The complexity of Eq. 2.3 is reduced through architectural modeling simplifications and by limiting the number of factors per cross term and the total number of terms in the model.

The methodology was validated by modeling a LEON3 (SPARC V8, 130 nm) and a Cortex-A8 (65 nm) core. An RTL simulator provided the activity factors, and the power results for training runs and for the validation itself. PrEsto was able to estimate the total core power with errors of only 2.6 % for LEON3 and 1.7 % for the A8, being 700 000 times faster than the reference simulator and reaching 7 MIPS.

2.4.3.7 McPAT

McPAT (*Multi-core Power Area and Timing*) [90,91] is a modeling framework that estimates together power, timing and area of multi- and manycores. In-order, out-of-order cores and the SMT technology are supported. The user specifies the system configuration and activity factors of components in a input

XML file. Then, McPAT builds an internal chip model, characterizing the area and energy cost per access of components, mostly through analytical models.

McPAT is composed of three main modules:

- **Hierarchical power, area and timing**: The chip is successively subdivided into smaller components. The hierarchical levels are:
  - **Architectural level**: The chip is for instance divided into cores, NoCs, caches, memory controllers and clocking circuitry.
  - **Circuit level**: Each architectural element is identified as one of the four basic circuit structures:
    * Hierarchical wires.
    * Arrays.
    * Complex logic.
    * Clocking network.
  - **Technological level**: Represents devices and wires specified by physical parameters (resistance, capacitance and current densities). In the McPAT version 1.0, transistors are modeled up to the 16 nm technology node, based on the International Technology Roadmap for Semiconductors (ITRS) [117] until 2019.
- **Optimizer**: This module tries to estimate non-specified parameters.
- **Internal chip representation**: Stores all system parameters in a hierarchical representation. The information from this module, most of which come from the input file, feeds the analysis of power, timing and area.

The power, timing and area models are described in the following:

- **Power**:
  - **Dynamic**: The capacitance of the circuit level elements are estimated from analytical equations, and activity factors are estimated from access statistics and the properties of circuits.
  - **Short-circuit**: Temporary short-circuits between pull-up and pull-down circuits dissipate energy. Such dissipations are estimated through the equations derived by Nose et al. [106].
  - **Leakage**: The two main sources (sub-threshold and gate leakage) are estimated through MASTAR [117] and data from Intel [20].
- **Area**:
  - **Regular structures and basic logic gates**: For example, memory arrays, interconnects and regular logic are mostly modeled as in CACTI [127], with some improvements, and as in the work of Palacharla et al. [109].
  - **Complex structures**: Complex logic such as FUs are better modeled empirically. In McPAT 1.0, the ALU and FPU models are based on Intel [100] and Sun [89] designs.

- **Timing**: The system is divided into small and basic components or devices. For each basic structure, McPAT uses its resistance and capacitance to compute its timing through RC delay equations.

McPAT was validated against real in-order, out-of-order, as well as desktop- and embedded-class processors: two SPARC (Niagara at 90 nm, Niagara 2 at 65 nm), the Xeon Tulsa (65 nm), the Alpha 21364 (180 nm), the dual-core Atom (45 nm), and the dual-core Cortex-A9 (40 nm) [91]. The total peak power estimation error varies from only 2.3 for the A9 to 22.6 % for the Xeon. The peak power breakdown per processor component showed errors between 11 and 23 % in the SPARC, Xeon and Alpha processors. The area estimation errors oscillate between 2.7 for the A9 and 20.8 % for the Niagara.

Xi et al. evaluated the McPAT models against empirically tuned micro-architectural models of a POWER7 server multi-core chip [138]. Accordingly to their study, McPAT models only between 20 and 55 % of the power dissipation sources of the POWER7 cores. In term of area, the McPAT models of the EXE stage encompass between 85 and 100 % of respective POWER7 circuits, but this fraction is reduced to only between 30 and 60 % of components from the remaining pipeline structures. In their experiments, in spite of the incomplete models in McPAT, the area of five main pipeline units (fetch, issue, load/store, fixed and FP/SIMD) were overestimated by factors between two and almost nine (and similarly the power dissipations), even if McPAT was configured with detailed (privileged access) parameters of the POWER7 core.

I asked Xi if he tried to estimate the areas without the McPAT flag `opt_local`, which optimizes the component areas to meet timing constraints given by the transistor technology (usage recommended if no detailed timing is known, which is not the case in one of their experimental setups), but he did not consider this possibility and argued that McPAT is presented as a framework to model together power, area and timing, and sometimes researchers use it to estimate the clock speed of hypothetical processors.

2.4.3.8 SST

The Structural Simulation Toolkit [75] is a set of tools integrating the simulation of power, performance and temperature of processors, among others. The aim is to provide an integrated, scalable and parallel simulation framework at multiple levels of detail for supercomputing systems composed of a single multi/manycore processor, or multiple computing nodes in a network.

The SST core is very modular and based on the Message Passing Interface (MPI) to build a parallel and discrete event simulator. It interacts with instances of the class Components, which represent for example the processors, memories and network switchers.

The Technology Interface is the core of the power, temperature and reliability simulation. The usage statistics of each component are received by the interface at a configurable rate. This data is then used, for instance by McPAT and Hotspot [119], to compute the dynamic and static power with temperature feedback.

In the version 2.2.0, the gem5, McPAT and Hotspot libraries worked for x86 only. The support of the other architectures in gem5 was planned to be functional in the version 5.0.0,[3] but the gem5 support will probably be removed from SST.[4]

[3] `https://sst-simulator.org/ticket/91` [Accessed: 17 April 2015]

[4] `https://sst-simulator.org/wiki/SSTmicroReleaseV5dot0dot0ReleasePlan` [Accessed: 17 April

2.5 Run-time code optimizations

In this section, the state of the art of run-time code optimizations through specialization or auto-tuning is presented. This work focuses on high-performance run-time techniques comparable to statically compiled languages, therefore just-in-time (JIT) compilers that for example accelerate interpreted languages or ahead-of-time compilers are not presented. HotSpot for the Java language and V8 for JavaScript are examples of JIT compilers and Android Runtime is an example of ahead-of-time compiler, which are not surveyed here. Four fields are briefly surveyed: run-time code specialization, dynamic binary optimizations, run-time recompilation and online auto-tuning.

2.5.1 Run-time code specialization

Run-time code specialization is implemented by two main techniques: code generation and instruction insertion into code templates.

2.5.1.1 Run-time code generation

Dynamic code generators define specific or supersets of languages for run-time code generation. Program or data specialization are implemented by special language constructs that capture dynamic information.

Fabius [87] was proposed to dynamically generate and specialize machine code to accelerate ML programs, achieving comparable or even faster speedups than C-optimized versions. The optimizations included constant propagation, loop unrolling and strength reduction. The compiler needed programmer annotations to indicate which function arguments were going to be processed and specialized at run time. Whole functions could also be annotated to be inlined at run time. Fabius achieved a very fast code generation speed because no support for register allocation and instruction scheduling was implemented. The lack of mutable data structures in the ML language also contributed to the code generation speed, because memory aliasing analysis is not necessary.

‘C [112] and tcc implemented an ANSI C superset and compiler that allowed high-level code to be dynamically translated into machine instructions. ‘C provided two run-time systems: a one-pass code generator for generation speed and a more elaborated back-end with a simple intermediate representation (IR) for improved register allocation and code quality. The backquote or tick character (‘) indicates the beginning of a dynamically generated statement, including whole functions with dynamic number of parameters and types. Dynamically assigned variables are indicated by a preceding “$” sign, inside backquote expressions, allowing the incorporation of run-time values as constants in the dynamically generated code. Static and dynamic types are allowed through special keywords.

With more abstraction and optimizing possibilities, TaskGraph [26] implements a C-like sub-language for dynamic code generation, and is integrated into applications as a C++ library. It is very portable, any TaskGraph application and its dynamic code are generated by any standard C compiler. Internally, TaskGraph uses the SUIF-1 representation to perform optimizing passes. TaskGraph is highly portable, but suffers from high run-time overheads, being only adapted to long-running workloads.

deGoal [43] implements a DSL for run-time code generation of computing kernels. It defines a pseudo-assembly RISC-like language, which can be mixed with standard C code. The machine code is only generated by deGoal instructions, while management of variables and code generation decisions are implemented by deGoal pseudo-instructions, optionally mixed with C code. The language is statically source-to-source converted to standard C, producing calls to deGoal back-end functions. The dynamic nature of the language comes from the fact that run-time information can drive the machine code generation. Chapter 4 presents this tool.

2.5.1.2 Run-time template-based specialization

Some run-time specialization approaches try to perform as many computations as possible at compile time, reducing to the maximum extent the run-time overhead. To achieve this objective, selected variations of source code are statically compiled with empty places (called templates), where instructions or data are going to be filled at run time.

Tempo [49, 50] performs program and data specialization for C programs. Both compile-time and run-time specialization are possible. The specialization opportunities have to be identified and annotated by a programmer. In the run-time mode, parts of the code are statically compiled with gcc and stored into a template file. In the specialization process, a dedicated run-time specializer evaluates the static constructs, copies the selected template into a buffer and instantiates with static values and branch offsets. Tempo has been used in several applications, mainly in operating systems and compiler generation. Beyond its usage as a specializer in specific applications, it has been used as a back-end specializer for C++ and Java.

DyC [68,69] is a declarative, annotation-based data specialization system. Polyvariant specialization and division are supported, i.e., multiple templates may be produced for a given run-time constant variable, and a piece of code can be analyzed with different combinations of variables being considered as run-time constants, respectively. DyC targets larger programs than Tempo and ‘C. Calpa [103] is a tool that automatically finds specialization opportunities and generates annotations for DyC.

Khan [78] proposed a feedback-directed specialization method, based on templates. The source code of an application is statically profiled and good candidate variables for specialization are identified. Not only run-time constants are specialized, but also variables that remain a reasonable period with the same value. A cache of recently instantiated templates with given values is implemented to reduce run-time overheads. The optimal cache of instantiated templates is statically found through simulation. Khan observed speedups between 0.97 and 1.58 in a Pentium 4 and an Itanium II processor running SPEC benchmarks, however no analysis was shown using different input sets from those of the static profiling.

2.5.2 Dynamic binary optimizations

Dynamic binary optimization (DBO) frameworks instrument running applications with opportunistic machine code optimizations.

Dynamo [23] was one of the first DBO systems, supporting HP PA-8000 workstations. Even compared to aggressively optimized code from a static compiler, Dynamo showed speedups from 0.97 to 1.22 in eight applications. Applications running less than one minute seemed to not be suitable for DBO.

The observed speedups came mainly from removing indirect branches, which are always mispredicted in the studied platform.

DynamoRIO [34] is the IA-32 version of Dynamo. Instrumented optimizations include redundant load removal, strength reduction, indirect branch dispatch, and procedure call inlining. Employed in SPEC2000 benchmarks running in a Pentium 4 Xeon, DynamoRIO showed an average speedup of 1.12 over native execution for FP benchmarks, but combining integer ones, the performance matched the native execution.

The ADORE [95] and COBRA [79] frameworks implemented DBO systems for Itanium 2 processors. By addressing the problem of data cache pre-fetching at run-time, these tools showed speedups up to 1.68 compared to static compilation.

2.5.3 Run-time recompilation

Run-time recompilation is a technique that tries to take into account run-time information to produce more optimized executable code.

Kistler and Franz [80] proposed a system for continuous program optimization. While a program runs, the recompilation system tries to find better versions running in the background. Two new optimizations were proposed, and considerable speedups were observed when they were applied. Nonetheless, the dynamic recompilation system suffered from high overheads, and the time required to pay off was at least two minutes in a PowerPC 604e.

Recently, Nuzman et al. proposed a JIT technology for C/C++ [108]. Their approach consists in delivering executable files along with an IR of the source code. In other words, an initial binary code is provided to avoid the overhead of generating it at run-time. Then, a JIT compiler is charged to profile the running code and recompile hot functions in idle cores. They claim that their work is the first to obtain performance gains from recompilation of regular/short-running programs. In their main experiment, the benchmarks were compiled with moderate optimization levels, arguing that this is a realistic context in the real world. On average their approach provided speedups of 1.07 in CINT2006 benchmarks, running in a POWER7 processor.

2.5.4 Online auto-tuning

Auto-tuning has been used to address the complexity of desktop- and server-class systems. This approach tries to automatically find the best compiler optimizations and algorithm implementations for a given source code and target CPU. Usually, such tools need long space exploration times to find quasi-optimal machine code. Only few work addressed auto-tuning at run-time.

ADAPT [135] was one of the first frameworks to dynamically optimize a running application, without prior user intervention. The dynamic compilation was performed in an external processor or in a free processor inside the computing system. In three experiments, ADAPT was only able to speedup workloads that run during more than one hour on an uniprocessor.

Active Harmony [129] is a run-time compilation and tuning framework for parallel programs. By

defining a new language, programmers can express tunable parameters and their range, which are explored at run time. In experiments performed in cluster of computers, the code generation is deployed in idle machines, in parallel to the running application. Active Harmony provided average speedups between 1.11 and 1.14 in two scientific applications, run with various problem sizes and in three different platforms.

IODC [45] is a framework of iterative optimization for data centers, which is transparent to the user. The main idea is to control the intensity of space exploration (recompilations and training runs) accordingly to the savings achieved since the beginning of the execution, because the execution time of the workload is assumed to be unknown. For compute intensive workload, IODC achieved an average speedup of 1.14.

SiblingRivalry [4] is a auto-tuning technique in which half of cores in multi-core processors are used to explore different auto-tuning possibilities with online learning, while the other half run the best algorithm found so far. It can both adapt code on-the-fly to changing load in the system, and to migrations between micro-architectures. In this latter case, on average this approach both speeded up by 1.8 and reduced the energy consumption of eight benchmarks by 30 %, compared to statically auto-tuning code for Intel Xeon and running in AMD Opteron, and vice-versa (reference versions were allowed to use all cores).

2.6 Conclusion

This chapter presented the five main topics related to this thesis. The study of energy dissipation in processors and techniques of energy reduction integrated into compilers, as well as run-time code optimizations, allowed us to understand the role that a dynamic code generator can play to reduce the energy consumption of embedded processors. The survey of existing micro-architectural simulators for ARM helped us to better choose a simulation framework apt to model core heterogeneity in modern embedded processors.

Most of the existing work that addressed the performance and energy improvement through dynamic code generation was proposed for desktop- and server-class processors. Furthermore, to my knowledge, no work addressed online auto-tuning in embedded systems. Existing tools are only adapted to workloads that run for minutes or hours, as those of server and scientific computation. General purpose embedded processors may frequently run tasks for a few seconds to a few minutes, and because of constraining limitations, specially energy consumption, recompilation should be performed with very low overheads.

This thesis proposes a methodology and implements a proof of concept to address the challenge of pushing online auto-tuning to embedded systems. The aim is to explore input-dependent and adapt code to pipeline features during the execution of a program, in order to speedup and increase its energy efficiency, overcoming static compilation. A run-time approach is suitable, because in hand-held devices, the input data from the user and the processor are usually not known at compile time. In addition, ARM processors may be ISA-compatible, but with varying pipeline implementations from different SoC designers. Chapter 5 presents the proposed online auto-tuning approach.

In order to study embedded pipeline designs, a micro-architectural simulator was developed during this thesis, and it can estimate the energy and performance of in-order and out-of-order ARM designs, currently present in smartphones and tablets. This simulation framework was used to compare the en-

ergy and performance of in-order and out-of-order CPUs and to show that online auto-tuning can virtually replace hardware out-of-ordering in some cases, by auto-tuning the code in simpler in-order CPUs. Chapter 3 presents the simulation framework for ARM.

Chapter 4 presents a run-time code generator and its porting for ARM. This tool was used as a parametrizable code generator in the online auto-tuner, being responsible for generating various versions of machine code, but functionally equivalent.

Chapter 3

Micro-architectural simulation of ARM processors

Heterogeneous multi-core systems have gained momentum, specially for embedded applications, thanks to the performance and energy consumption trade-offs provided by in-order and out-of-order cores.

Simulation has been used to model and predict the performance and energy trade-offs of core heterogeneity. It allows researchers to explore design possibilities, simulate variations of existing hardware and test ideas that otherwise would not be possible in real experiments.

A simulator should model configurable pipeline structures to allow the design exploration of core heterogeneity. As discussed in Chapter 2, at the architectural and instruction level, intra-core components are not modeled, and hence these simulation levels are not adapted for studies of core heterogeneity. The remaining abstraction levels comprise the detailed (transistor, gate, RTL) and micro-architectural simulators. Detailed simulation is only practical to simulate small pieces of code or to calibrate simulators with higher abstraction levels. Commercial emulators such as the Veloce2 [70] can accelerate RTL simulation and provide comparable simulation times as those obtained by micro-architectural simulators. However, those emulators require considerable computing power. In addition, given that RTL models represent actual hardware, they lack flexibility and abstraction, only allowing quick design explorations around the original RTL model. Therefore, the micro-architectural level is the most adapted to explore various core designs.

Several research studies of different forms of core heterogeneity were carried out in micro-architectural simulators [22, 85, 96, 118, 123]. Those studies employed x86 and Alpha simulators, even for embedded core simulations. Nowadays, the ARM ISA is more relevant in embedded computing ecosystems. Even if experimental measurements suggest that power and performance of modern processors are independent of ISA [31], the advantage of using an ARM simulator is that researchers can evaluate their ideas in real embedded software environments and use up-to-date toolchains.

The micro-architectural performance simulation of modern ARM cores is dominated by the gem5 community [29], to which ARM itself contributes and internally employs in some projects. For energy estimation, regression models, such as those used by CAMP [113] and PrEsto [124], show very ac-

curate estimations, but they need a reference hardware model (e.g., RTL simulators). Extending such approaches to estimate power of an arbitrary model would require advanced technology and micro-architecture knowledge. On the other hand, McPAT [91] is a relatively easier tool to use. With simple high-level specification, an user can obtain area and average power estimations in seconds. Of course, this easiness of deployment may result in rough estimations, because a lot of low-level information is hidden in the source code through hardware implementation assumptions.

In the beginning of the thesis, we were searching for a micro-architectural simulator, at the same time capable of simulating in-order and out-of-order cores, as well as estimating energy and performance. Such simulator did not exit, but gem5 coupled to McPAT seemed to be the best choice. We decided not only to feed McPAT with the results from gem5, but also to implement an in-order model for ARM in gem5.

This framework would allow us to study the impact of code generation strategies on various micro-architecture configurations. It would also allow to simulate similar in-order and out-of-order cores, except for the dynamic scheduling capability, and evaluate at what extent micro-architectural code adaption to in-order pipelines can "replace" out-of-order execution, and hence reduce the energy consumption. This evaluation can not be performed with commercial ARM cores, because in-order designs have less resources than out-of-order ones.

The remainder of this chapter first details both simulators and describes the modifications and improvements in Sections 3.1 and 3.2. The conversion of the gem5 output to input for McPAT is presented in Section 3.3. Then, validation results of performance are shown in Section 3.4 and of combined energy/performance estimations in Section 3.5. An architectural/micro-architectural exploration is exemplified in Section 3.6. Finally, Section 3.7 discusses the scope of application and limitations of the proposed simulation framework, and Section 3.8 concludes this chapter by summarizing the contributions.

3.1 gem5

gem5 [29] is a micro-architectural simulator. It can estimate the timing and utilization statistics of core and cache components. It is famous for precisely modeling pipeline dependencies at each stage, before computing the result. This emphasizes the instruction timing and simulation accuracy (called *execute-in-execute* modeling), differently from trace-driven simulators such as SimpleScalar [36]. Section 2.4.2.2 briefly detailed gem5. This section focuses on the ARM core simulation, and is based on the stable version of June 2012 [67], if not otherwise stated.

For micro-architectural simulation, only the InOrder and O3 CPU models originally produce activity statistics of in-order and out-of-order pipelines, respectively. The Minor CPU model was added in July 2014 to provide an in-order pipeline more adapted to modern implementations. Two types of simulation environment can be used: System-call Emulation (SE), and Full-System (FS). In the SE mode, most of the system calls are emulated by passing them to the host operating system. On the other hand, the FS mode simulates a bare-metal machine, then an operating system must run to boot the machine and to support the running applications.

For micro-architectural simulation with the ARM ISA, the Minor and O3 CPU models are currently functional. However, when this thesis began, there was no in-order model functional for ARM. The following sections describe the O3 CPU model and how I modified it to mimic an in-order pipeline.

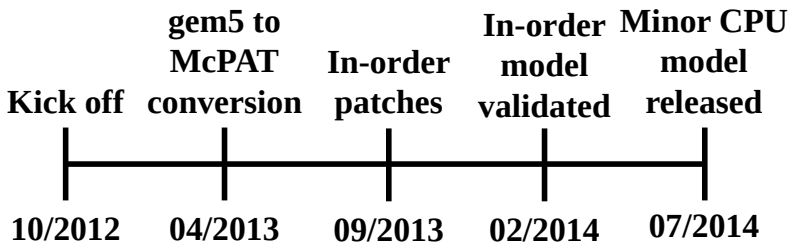


Figure 3.1: gem5 development timeline in the thesis before an in-order model for ARM (Minor CPU model) was released in gem5.

Figure 3.1 shows the gem5 development timeline during this thesis before the Minor CPU model was released.

3.1.1 The `arm_detailed` configuration

The released `arm_detailed` configuration simulates a multi-core system connected to a simple memory hierarchy. Figure 3.2 presents the main components of this system.

This configuration allows us to simulate the out-of-order designs from the ARMv7-A architecture, which comprises 32-bit processors from the Cortex-A family, widely deployed in ARM-based smartphones and computer tablets.

3.1.1.1 O3 CPU model

The O3 CPU model simulates a generic out-of-order pipeline based on the physical register-file architectures, like the DEC Alpha design [88]. Seven pipeline stages are modeled: fetch, decode, rename, issue, execute, writeback and commit, but the timing of deeper pipelines can be effectively simulated by changing the signal delays between stages. One exception is the EXE stage, whose depth is indirectly modeled through the latencies of instructions.

The O3 model is highly configurable: for example, each parameter of the branch predictor (BP) (*tournament*, based on Alpha implementations [88]) can be configured. Other parameters include pipeline stage widths, number of physical registers, and the size of buffers such as the branch target buffer (BTB), return address stack (RAS), instruction queue (IQ), ROB and load and store queues (LSQs).

Table 3.1 shows the configuration of the EXE stage. It is very flexible with variable number of FUs. For example: simple ALUs, complex ALUs (MUL/DIV), FP/SIMD, load and store units (LSUs). Each unit accepts one or more operation classes (*opClass* in gem5) with configurable latency and a pipelined flag[1]. This model does not simulate operation forwarding. Each operation class regroups one or more instructions. The system configuration can be modified in python scripts. For example, they allow to create FUs, attach operation classes to it, and configure the latencies and pipelined flags. Configurations that are not present in the python interface, such as creating a new operation class, require modifications and recompilation of the gem5 source code (C++).

[1]In this version, the issue latency of operation classes is set to 1 to indicate that the FU is pipelined, or greater than 1,

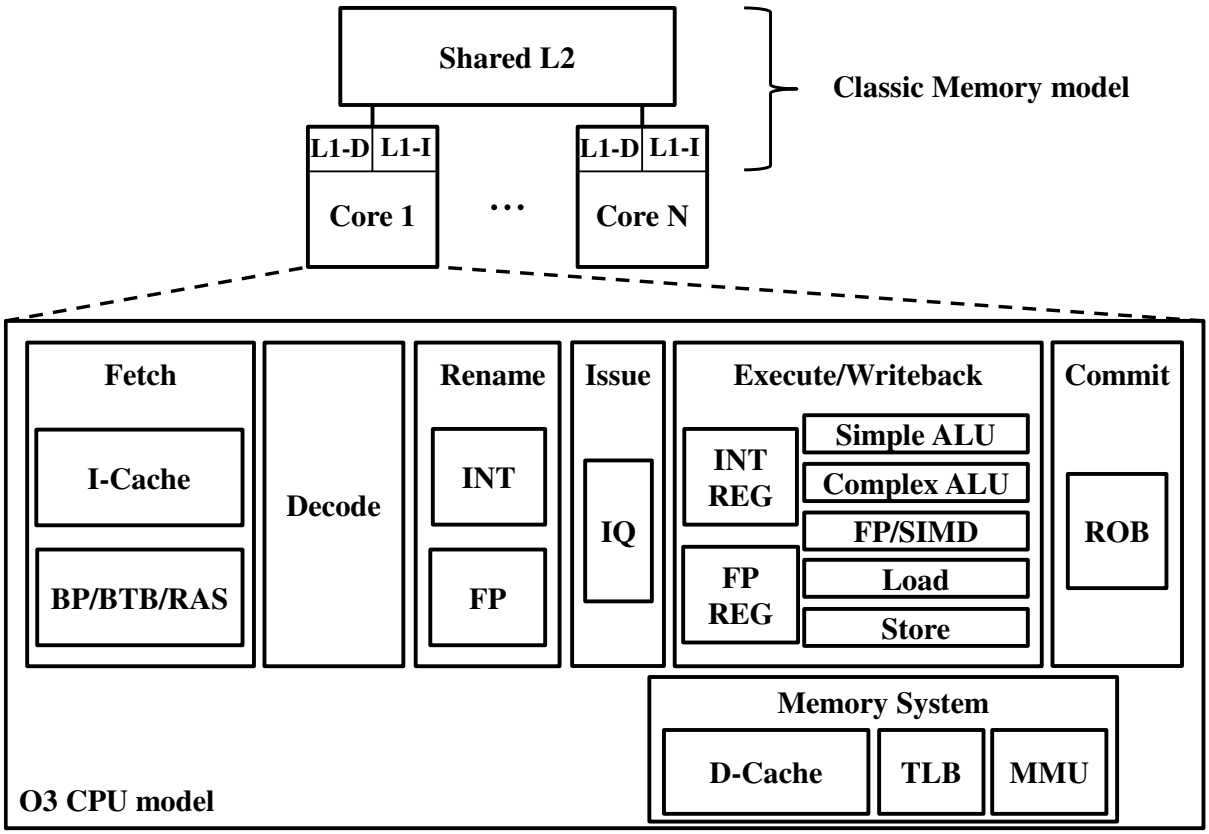


Figure 3.2: The `arm_detailed` configuration of gem5.

Table 3.1: Configuration of the EXE stage in gem5

Example of FU	gem5 opClass	Example of instructions
Simple ALU	IntAlu	MOV, ADD, SUB, AND, ORR
Complex ALU	IntMult	MUL, MLA
	IntDiv	UDIV, SDIV
FP/SIMD Unit[1]	SimdFloatAdd	VADD, VSUB
	SimdFloatMult	VMUL, VNMUL
	SimdFloatMultAcc	VMLA, VMLS, VNMLA, VNMLS
Load Unit	MemRead	LDR, VLDR
Store Unit	MemWrite	STR, VSTR

[1] VFP and Advanced SIMD instructions are regrouped under the SimdFloat* operation classes. Only a few examples are shown here.

3.1.1.2 Memory hierarchy

The `arm_detailed` model uses the Classic Memory model. Two private caches (data and instruction) form the first cache level. A shared L2 cache is the last level, connected to the Simple-

otherwise.

Memory[2] model. Cache parameters include latencies, miss status and handling registers (MSHRs) and write buffers (WBs), associativity and size. The SimpleMemory model has a latency and bus frequency, among others. Stride prefetchers (SPs) can be configured and added to any cache level.

The Classic Memory model puts the focus on the pipeline simulation, being adapted to the study of micro-architectural adaption of code.

3.1.2 Modeling improvements

I extended and enhanced some simulation behaviors in the source code of gem5. For energy estimation with McPAT, a few more statistics were created. I also improved: the LSQ stall modeling to better simulate small cores, missing configurations of SIMD load and store micro-operations and a LSU bandwidth configuration.

3.1.2.1 Statistics

For better energy estimation, I inserted the following statistics required by McPAT:

- Number of BTB updates (BPredUnit.BTBLookups).
- Number of instructions that wake consumers, by producing an integer value (iew.wb_int_producers).
- Number of integer and floating-point renamed operands (rename.int_rename_operands and rename.fp_rename_operands).

3.1.2.2 Rename and dispatch stalls regarding the LSQ

I enhanced[3] the modeling of the rename and issue stages to better simulate pipelines such as the small cores from the Cortex-A series. Embedded cores with limited resources need a more detailed modeling of resource management than cores with plenty of resources. For instance, consider the simulation of a pipeline buffer that is purposefully not perfectly modeled, creating unrealistic stalls when it is almost full. In a core with a small buffer, the "almost full" condition is more frequent, therefore the abstraction error is greater in small cores than in a core with a larger buffer.

When the load or store queue is full, the rename or dispatch stages must stall the front-end and issue stage of the pipeline. In this version of gem5, the rename and dispatch stages considered an unified signal of full LSQ entries. In addition, these two stages stalled the pipeline whenever the load or store queue was full, independently from the instruction type. I modified this behavior by creating two separated signals, one for full load queue (FLQ) and the other for full store queue (FSQ). Moreover, I changed

[2] Butko et al. observed faster behaviors in the SimpleMemory than in real DDR memories. Since then, more accurate models have been added and evaluated [72]

[3] Since then, a similar modification has been implemented in newer gem5 versions (patches 10240:15f822e9410a and 10239:592f0bb6bd6f of 21 June 2014).

Table 3.2: Number of cycles[1] of some NEON load instructions from: then original gem5 model, the improved model, and Cortex-A8 and A9.

Instruction	gem5 original	gem5 improved	ARM Cortex[2]	
			A8 [7]	A9[3][11]
`VLD1.16 {D0}, [R0]`	5	3	1-2	1-2
`VLD1.16 {D0[0]}, [R0]`	9	3	2-3	3-5
`VLD1.16 {D0[]}, [R0]`	9	3	1-2	1-3
`VLD2.16 {D0, D1}, [R0]`	9	6	1-2	1-3
`VLD2.16 {D0[0], D1[0]}, [R0]`	9	3	2-3	3-5
`VLD2.16 {D0[], D1[]}, [R0]`	9	3	1-2	1-3
`VLD3.16 {D0, D1, D2}, [R0]`	10	8	4	4-6
`VLD3.16 {D0[0], D1[0], D2[0]}, [R0]`	9	4	5	8
`VLD3.16 {D0[], D1[], D2[]}, [R0]`	9	4	3	4
`VLD4.16 {D0, D1, D2, D3}, [R0]`	11	11	3-4	4.5-6.5
`VLD4.16 {D0[0],D1[0],D2[0],D3[0]}, [R0]`	9	4	4-5	6.5-8.5
`VLD4.16 {D0[], D1[], D2[], D3[]}, [R0]`	9	4	2-3	2.5-4.5

[1] Number of cycles until the result is ready, considering a cache hit.

[2] The latency estimation technique is the same as that given in Section 3.4.2.2 for INT instructions in the A8. The range of cycles represents memory accesses aligned or not. Fractions correspond to the average cycles of situations in which the result registers are available at different pipeline stages.

[3] Number of cycles when results are bypassed, otherwise at least 4 more cycles are needed to writeback.

the stall behavior by not allowing the rename stage to stall the pipeline because of the LSQs. Only the dispatch stage can produce a stall when a load instruction need to be dispatched and there is a FLQ signal, or the same for a store instruction and the FSQ signal.

3.1.2.3 SIMD load and store micro-operations

I observed considerable slowdown in some NEON load and store instructions. The source of the problem was that some SIMD micro-operations had no specific operation class, being then assigned to the default one (FloatAdd).

I assigned 21 missing operation classes to better model NEON loads and stores that interact with sub-elements in a SIMD register. Examples of such instructions are `VLD1.16`, `VLD2.16`, `VLD3.16` and `VLD4.16`, which load 1 to 4 consecutive 16-bit elements from the memory into vector registers. Table 3.2 shows the timing of some instructions with and without the improvement, and timings from the Cortex-A8 and A9 [7, 11]. Of course, one could create customized micro-operations per instruction, being more accurate at the cost of less flexibility. I decided to keep the original micro-operation implementation, because I have no information of how those NEON macro-instructions should be decomposed.

3.1.2.4 L1-D cache bus width to the pipeline

In gem5, there is no core/cache parameter that allows us to set the L1-D cache bus width to the pipeline. For example, in the Cortex-A8 (32-bit core), this bus in the NEON unit is 128-bit wide, allowing to load and store up to four words per cycle [7, Section 7.2]. In gem5, only one word per cycle can be processed by a load or store unit, therefore instructions that load or store multiple registers could be considerably slowed down, such as LDM* and STM* instructions.

The number of micro-operations and hence the number of cycles required by a load or store instruction (considering cache hit) depends on its description in the ISA support of the simulator. Currently, there is no way to easily set the number of words accessed per cycle, because for each possible value the description of memory instructions should be modified.

I addressed this problem with an approximate solution: bursts of micro-operations are allowed to be processed at once, if they are part of a load or store multiple registers. The number of micro-operations in a burst is limited to respect the maximum number of L1-D cache accesses per cycle. The drawback of this approach is that the front-end bandwidth may limit the flow of micro-operations that in reality should be merged, also increasing the power dissipation.

With this modification, an auto-tuner can explore the performance trade-offs of generating code with instructions that load/store a single or multiple registers.

3.1.3 In-order model based on the O3 CPU model

One of the hypotheses of this thesis is that run-time code adaption to a target pipeline can improve the performance and reduce the energy consumption. For energy efficiency, in-order pipelines are preferred over out-of-order designs. Hence, it would be interesting to compare the performance gap and energy benefits of the proposed run-time technique run in an in-order pipeline to a statically compiled code run in a similar out-of-order pipeline.

To this aim, first I argue why I chose to modify the O3 model to mimic an in-order pipeline. Next, I explain how the O3 model was modified, then a qualitative explanation of the simulation precision of this approach is presented. Its quantitative validation is presented in Section 3.4.

3.1.3.1 InOrder vs modified O3 model

The InOrder model has a common simulation engine and architecture-dependent supporting code, which is partially ported to ARM. In my opinion, the InOrder model may not be adapted to simulate current high-end microprocessors, due to the lack of structures such as the IQ, LSQ or store buffer, present in modern embedded microprocessors. In addition, FP and SIMD instructions are apparently not supported yet.

On the other hand, the O3 model is based on high-performance processors. In my opinion, modifying the O3 model to simulate in-order cores of the Cortex-A series was faster and more stable to implement, compared to bring-up and enhance the InOrder model. However, the modifications provide a cycle-approximate in-order pipeline, because out-of-order structures were not removed.

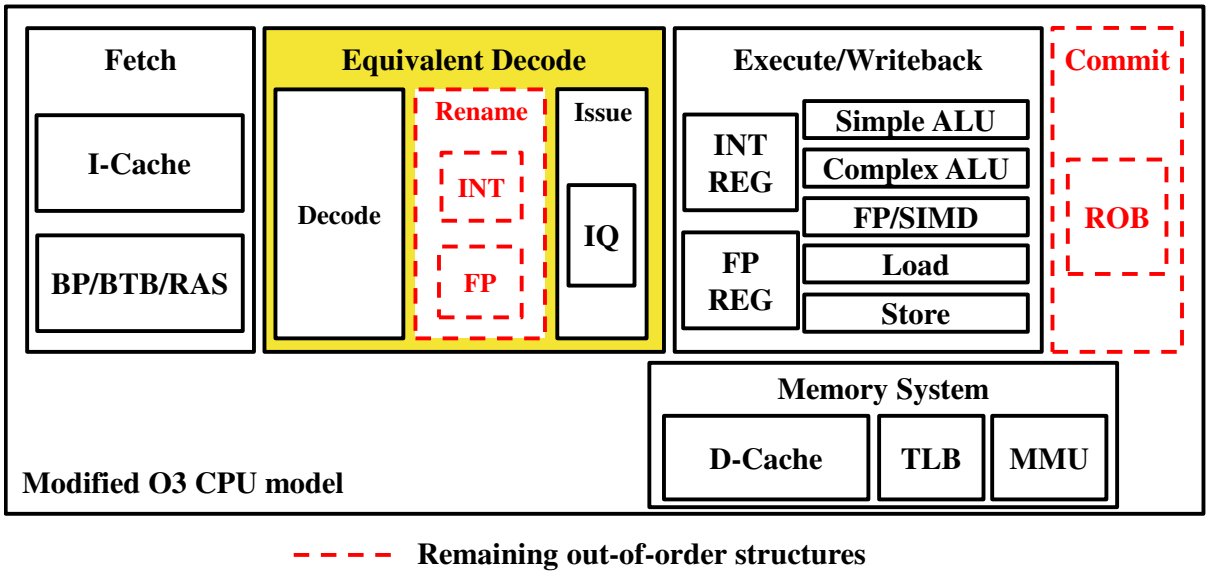


Figure 3.3: The proposed modifications in the O3 model to mimic an in-order pipeline.

3.1.3.2 Modification of the gem5 O3 model

Ideally, to simulate an in-order pipeline using an out-of-order model, structures supporting out-of-order execution should be completely removed. However, instead of removing these structures, which can be complex, time-consuming and error prone, the original pipeline stages were kept intact. In summary, the two following modifications were implemented over the original out-of-order model:

- Issue in dispatch order.
- An extra register scoreboarding to disable register renaming effects.

These two modifications were implemented at the issue and writeback stages, as explained in the following.

Figure 3.3 depicts the proposed modifications.

Issuing in program order I modified the instruction scheduler so that only the oldest instruction in the instruction queue is allowed to issue if it is ready. The only exceptions to this rule are instructions that already requested a memory access and are re-issuing. In this case, I allow them to be issued again even if they are not the oldest instructions. This happens, for example, with load instructions of uncacheable data.

Disabling the rename stage The register renaming must be disabled, because it would incorrectly allow the concurrent execution of instructions that had register dependencies avoided by renaming their destination registers. Then, to cancel the effect of register renaming, an extra register scoreboarding was implemented over the original model. As a consequence, ready instructions are only issued if their

source and destination registers are not being modified by any issued instruction under execution. My register scoreboard only locks destination registers, because I considered that source registers are read during the first cycle of execution. The destination registers of an instruction are then unlocked in the writeback stage.

Out-of-order parameters The parameters that only exist in out-of-order pipelines should be configured properly to mimic an in-order core. Table 3.3 lists and describes these parameters.

3.1.3.3 Functionality

Considering that I modified the instruction scheduler to only issue ready instructions respecting two extra restrictions (issue in-order and respect register dependencies), there is no reason that this approach would lead to wrong results. In other words, every ready instruction from the in-order pipeline point of view is always a ready instruction from the out-of-order point of view. I tested the correct execution of the model, by comparing the results produced by benchmarks. No errors were detected.

3.1.3.4 Influence of remaining structures

My modification of the gem5 O3 model to simulate an in-order pipeline does not remove out-of-order structures. These remaining structures impact the accuracy of the simulator, albeit in a negligible way:

- An "empty" rename stage exists. Nevertheless, we can consider the rename stage as an additional decode sub-stage, replacing one cycle of decoding delay. This was acceptable in the simulated ARM cores, whose instruction decoding takes more than one cycle.
- The commit stage continues managing the correct execution in the pipeline, even if the vast majority of the correct execution is ensured by the issue in program-order modification.

3.2 McPAT

McPAT is a micro-architectural power and area estimator for multi- and manycores [91]. Given system parameters, McPAT estimates the area, peak and average power of SoC and pipeline components. It supports in-order and out-of-order core designs, and multi/manycores can be homogeneous or heterogeneous.

First, an overview of the McPAT models is given. Then, I present a simple modification and methodology to better model core heterogeneity.

3.2.1 Overview

Here, I briefly describe the McPAT user interface, energy models and ARM support.

Table 3.3: Configuration of gem5 O3 model parameters[1] when simulating a dual-issue in-order pipeline

O3 model parameter	Ideal value	Chosen value	Description
*ToRenameDelay renameTo*Delay iewToCommitDelay commitTo*Delay renameToROBDelay	0	1	The rename and commit (which includes the ROB) stages do not exist in an in-order pipeline, then signal delays to and from them should be as small as possible.
numROBEntries commitWidth squashWidth	infinite	512	The ROB is an out-of-order structure that holds the information of instructions eventually executing out-of-order. An in-order pipeline does not need such buffer, because correct program execution must be ensured before instructions writeback. To avoid the ROB stalling the pipeline, ideally there would be an infinite number of entries and an infinite commit and squash bandwidth.
renameWidth	infinite	numROBEntries	With an infinite bandwidth, the rename stage would not stall the pipeline due to instructions waiting to be renamed. In the O3 model, the renaming throughput is only limited by structures that receive the renamed instructions in the following stages, then I set its bandwidth as the number of entries in the largest structure.
dispatchWidth	infinite	numROBEntries	Same reasoning as renameWidth. The dispatch is between the rename stage and the following structures, then it has to be configured with the same bandwidth as the rename stage to not stall the pipeline.
numPhysIntRegs numPhysFloatRegs	infinite	$44 + \text{numROBEntries}$ $72 + \text{numROBEntries}$	In order to avoid stalls in the rename stage due to lack of physical registers, these parameters should be set as large as possible. For a finite ROB, the maximum number of physical registers used is equal the number of architectural registers plus the number of entries in the ROB.[2]

[1] To work properly, the out-of-order model needs two patches: issue in program order and disable register renaming.

[2] In the worst case, each entry in the ROB could have instructions with two destination registers, then $44 + 2 \times \text{numROBEntries}$ and $72 + 2 \times \text{numROBEntries}$ are better values.

3.2.1.1 User interface

The user interface with the simulator is provided through an XML file, which describes system parameters and utilization statistics of components. Given the processor parameters, McPAT builds an internal chip model, modeling and estimating the area and peak power of architectural elements like caches, NoCs and cores. Intra-core elements include each pipeline stage of in-order or out-of-order designs. For average power estimation, the XML file should contain the utilization statistics of core and other SoC components. These statistics come from gem5, through a modified version of a publicly available parser [121].

3.2.1.2 Energy and power models

For most micro-architectural elements and caches, the energy cost per access is estimated through analytical models. The energy consumption of these elements is then calculated multiplying the number of accesses by their energy cost. At a first glance, this technique based on the energy cost per access gives the impression that dynamic energy is independent of the instruction timing. However, it is exactly the role of a performance simulator to estimate the micro-architectural accesses in a dynamic environment, taking into account the timing of instructions. For example, in an out-of-order core, the speculation and hence the accesses to pipeline structures may vary depending on cache latencies. For McPAT, what matters is the total number of accesses (including speculative ones).

The energy consumption of FUs is not modeled per access, but per cycle. In consequence, multiple cycle instructions should be counted as multiple accesses. The area and energy cost of FUs are empirically modeled.

McPAT also estimates the leakage power at a given temperature. The temperature feedback on leakage power is not modeled.

Five main pipeline stages are modeled, with the following structures in out-of-order cores based on physical register file:

- **Instruction fetch unit (IFU)**: L1-I cache, BTB, tournament branch predictor (based on Alpha implementations [88]), RAS, instruction buffer and instruction decoder.
- **Renaming unit**: Two structures of front-end register alias table (RAT), free list and retire RAT, for INT and FP instructions.
- **Load/store unit (LSU)**: L1-D cache and LSQs.
- **Memory management unit (MMU)**: I-TLB and D-TLB.
- **Execution unit (EXE)**: INT and FP register files, INT and FP instruction schedulers, INT ALUs, complex ALUs (MUL/DIV), FP/SIMD units and result broadcast bus.

In in-order cores, there is an unified LSQ, and no renaming unit. An unified instruction scheduler is only present in in-order SMT processors.[4]

[4] Apparently, multi-issue in-order pipelines should also have this structure, as noted in the McPAT source code itself. However, the source code only models an instruction scheduler when SMT is enabled.

McPAT models the three transistor types described by ITRS, including the low operating power (LOP) and high performance (HP) configurations usually employed in high-performance embedded processors.

3.2.1.3 ARM support

The pipeline models are based on Intel, Alpha and SPARC designs, but low-power embedded SoCs are also supported. For example, when the flag "Embedded" is activated in the input file, McPAT models the VFP/NEON unit of the Cortex-A9.

3.2.2 Better modeling core heterogeneity

The area and energy dissipation of FUs are empirically modeled in McPAT. The area and energy costs change with the technology node, but they are the same for any size of core. For instance, in the original McPAT model, the small in-order single-issue Cortex-A5 would have the same FU parameters as those of bigger out-of-order cores, such as the dual-issue A9 and triple-issue A15. I estimated that the VFP/NEON unit in the A5 is 6.5 times smaller than that in the A9, with areas of 0.15 and 0.98 mm^2 at 40 nm, respectively [15, 16, 91]. This example justifies the need of a better FU model.

The area ratio of FUs roughly follows the rule in Eq. 3.1, taken from Intel area scaling studies of out-of-order cores [118].

$$FU_{area} \propto issue\_width^2 \tag{3.1}$$

In Eq. 3.1, I considered the sustainable issue width or front-end width. In McPAT, the sustainable issue width is simply called *issue width*, in contrast to the *peak issue width*, which represents the maximum number of instructions that can be issued together in one cycle. In gem5, the previous two terms are better described by the *decode width* and *issue width*, respectively. I also considered that the energy cost per access also follows the same scaling rule.[5]

In McPAT, I developed a patch to allow the user to specify energy and area ratios for each FU, taking as reference those for the Cortex-A9, which seems to be the embedded core taken as reference in McPAT.

A concrete example and the validation are presented in Section 3.5.

3.3 Parameters and statistics conversion from gem5 to McPAT

This section explains the conversion of the gem5 output data to McPAT. I updated and modified the parser written by Richard Strong [121]. I chose this parser because it was developed by one of the authors

[5] After experimenting with this model, I found out that the FU heights should also be a function of this scaling rule. A FU height is unidimensional, while the area factor is a bi-dimensional metric. Then, the FU height should be multiplied by the square root of the area scaling factor. This missing factor explains the high and low dissipation contributions of the bypass buses in the A7 and A15, respectively, in Section 3.5.5.

of McPAT, but the explanations in this section are independent from the technique employed to translate gem5 parameters and statistics into the McPAT input file.

The McPAT input file is divided into a hierarchical list of components:

- **System parameters**: Configuration of cache levels, presence of peripherals, transistor technology, temperature, among others. System components are:
  - **Cores**: Pipeline parameters and statistics, some core components modeled as caches are instantiated as sub-components:
    - **Branch predictor**.
    - **BTB**.
    - **I-TLB** and **D-TLB**.
    - **L1-D** and **L1-I caches**.
  - **Cache directories**.
  - **L2** and **L3 caches**.
  - **Peripherals**: NoCs, memory and flash controllers, network and PCIe interfaces

Most of the system parameters are straightforward to be configured. This thesis focused on the core and cache simulation, with average power estimation. The average power is then simply multiplied by the execution time to obtain the energy consumption of components.

gem5 is very actively developed. Not only bugs are frequently fixed, but the user interface is also frequently modified to improve comprehensibility, including files with system configuration and statistics. The gem5 and McPAT equivalence presented in this section is based on the stable gem5 version of June 2012 [67], and McPAT 1.0 [74].

Tables A.1 and A.2 present the equivalence of core parameters and statistics. Some parameters and statistics in McPAT can not be easily represented or identified by gem5 equivalent expressions, in those cases, I presented a detailed description. Fixed parameters are the same found in the example of input file for a Cortex-A9[6].

Following the previous remarks, Tables A.3 and A.4 present the model equivalence of intra- and extra-core components: branch predictor, BTB, TLBs, L1 and L2 caches.

For most parameters and statistics, there is a straightforward conversion between gem5 and McPAT, for example `fetchWidth` in gem5 is `fetch_width` in McPAT. In other parameters, different synonyms are used in the two simulators, as “fetch buffer” for “instruction buffer” and “instruction queue” for “instruction window”. When the gem5 and McPAT modeling differs, I derived simple approximations. For instance, McPAT models a IQ for INT and another for FP instructions, while gem5 has a shared IQ. As a rough approximation, I equally divided the number of IQ entries in gem5 between the INT and FP queues in McPAT.

[6] ARM_A9_2000.xml.

3.4 Performance validation

I decided to evaluate the timing accuracy of gem5 for two reasons. First, to verify that the proposed in-order model in gem5 can simulate the behavior of real in-order cores. Secondly, to measure the timing accuracy of the O3 model in gem5 compared to a real out-of-order ARM core.

I configured the original O3 model of gem5 and the proposed in-order model for ARM as Cortex-A9 and A8, respectively. Then, the execution time of 10 PARSEC 3.0 [28] benchmarks were compared to measure simulation errors. To ensure a fair comparison, the same binaries and dynamic libraries were run in all platforms, and core frequencies were fixed at 800 MHz.

First, this section details the reference and simulation models. Then, the benchmarks and the compilation environment are presented. Finally, the simulation results are shown, the behavior of the proposed in-order model is analyzed and an enhancement of its EXE stage is proposed to better simulate the Cortex-A8.

3.4.1 Reference models

The out-of-order reference model is the Snowball SDK, equipped with a dual Cortex-A9 processor [38]. The board runs the Linaro 11.11 distribution with a Linux 3.0.0 kernel.

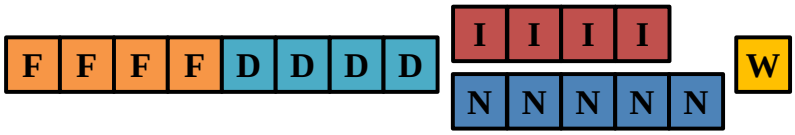
For the in-order reference model, I chose the BeagleBoard-xM SDK, which has a processor with only one Cortex-A8 core [25]. The board runs a Linux 3.9.11 kernel with the Ubuntu 11.04 distribution released by the gem5 website. I ensured that the NEON and VFP extensions access data in the L1 data cache (by default they access the L2).

3.4.2 Simulation models

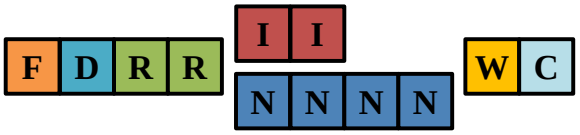
The Cortex-A9 is a dual-issue out-of-order pipeline with 8 to 11 stages [18], while the Cortex-A8 is a dual-issue in-order pipeline with 13 stages [8, Table 2-3]. The main parameters of the two models are summarized in Table 3.4. Figure 3.4 depicts both pipeline stage depths in gem5. The core clock and cache sizes were based on the development kits Snowball SKY-S9500-ULP-CXX series [38] and BeagleBoard-xM [25], respectively. The cache and memory latencies are typical values [5, Section 1.1]. The cache line length is 32 bytes in the Cortex-A9 [12, Section 7.1], but this version of gem5 only accepts 64 as length. Most of the other parameters were taken from manuals and from the ARM website [6–8, 12, 18, 125]. Unfortunately, there is no official information about the following parameters (in parenthesis, the sources if any, otherwise I made an educated guess). For the Cortex-A9 model:

- L1 cache buffer sizes: MSHR and WB (gem5 default values).
- BP and BTB parameters (McPAT).
- I-TLB and D-TLB sizes.
- Number of IQ and ROB[7] entries (default values).

[7] Accordingly to Li et al. [91], the Cortex-A9 implements a out-of-order pipeline without a traditional ROB.



(a) Cortex-A8.



(b) Cortex-A9.

F = Fetch, D = Decode, R = Rename/Dispatch/Issue, I = INT ALU, N = FP/SIMD, W = Writeback, C = Commit

Figure 3.4: Configuration of pipeline stage depths for the Cortex-A8 and A9 in gem5.

- Number of LSQ entries.
- Depth of pipeline stages (gem5 mailing-list[8]).

And for the Cortex-A8 model:

- L2 cache: MSHR and WB (default values).
- BP parameters.
- Separated depth of the fetch and decode stages.

3.4.2.1 The EXE stage

The FUs were configured as shown in Table 3.5, which only includes the VFP extension. Table 3.1 presents examples of instructions that each gem5 opClass can execute. The latency of instructions were obtained from ARM manuals [7, 10, 12]. While in the Cortex-A9, the VFP extension is pipelined for most instructions, the VFP extension in the A8 is not pipelined. The Advanced SIMD unit in the A8, which is pipelined, can execute some single-precision instructions faster than the VFP, but I did not test this feature.

The EXE stage of the in-order and out-of-order pipelines are modeled with two and three execution ports, respectively. Figure 3.5 shows the distribution of the FUs in their EXE stages.

[8] A suggestion is setting equal number of sub-stages, except the rename stage, which is two times longer. It is likely that in my gem5 configuration the fetch stage is too shallow.

Table 3.4: Parameters of the gem5 Cortex-A9 (out-of-order) and Cortex-A8 (in-order) models

Parameter[1]			Out-of-order (Cortex-A9)	In-order (Cortex-A8)
Core clocks			800 MHz	800 MHz
DRAM	Size / clock / latency		256 MB / 400 MHz / 65	256 MB / 166 MHz / 65
L2	Size / assoc. / lat. / MSHRs / WBs		512 kB / 8 / 8 / 11 / 9	256 kB / 8 / 8 / 16 / 8
L1-I	Size / assoc. / lat. / MSHRs		32 kB / 4 / 1 / 2	32 kB / 4 / 1 / 1
L1-D	Size / assoc. / lat. / MSHRs / WBs		32 kB / 4 / 1 / 4 / 16	32 kB / 4 / 1 / 1 / 1
	Stride pref.	Degree / buffer size	1 / 8	N/A
BP	Global History entries / bits		4096 / 2	512 / 2
	BTB / RAS entries		4096 / 8	512 / 8
I-TLB / D-TLB entries			64 each	32 each
Issue width			2[2]	2
gem5 effective EXE stage depth[3]			8	6
Pipeline stages			8	13
Physical INT / FP registers			62 / 256[4]	556[5] / 584[5]
IQ / LSQ / ROB entries			32 / 8 each[7] / 40	16[6] / 12 each[6] / 512[5]

[1] Latencies in core clock cycles.

[2] I assumed that the Cortex-A9 is dual-issue [18], although it may issue 4 instructions in some conditions [8, Section 4.4.1].

[3] In gem5, `wbDepth` multiplied by the issue width represents the maximum allowed number of in-flight instructions in the EXE stage. I considered that it should be set as the longest latency between pipelined instructions.

[4] The Cortex-A9 does not rename FP registers [8, Table 2-3]. The choice of the number of physical FP registers is explained in section 3.4.2.4.

[5] These structures do not exist in an in-order pipeline and the chosen values are explained in section 3.1.3.2.

[6] I considered the corresponding structures in the NEON/VFP unit: one 16-entry instruction queue, one 12-entry load queue [7, Section 16.5]. I assumed that it has a store queue with 12 entries too.

[7] The Cortex-A9 has a store buffer with 4 slots and probably at least a 4-entry load queue (i.e., support for four data cache line fill requests) [6]. Without precise information, these parameters were hence tuned in the simulator.

3.4.2.2 Setting instruction latencies

For the A9, the latencies were directly copied from the manuals, considering the most common and simple instructions. For example, simple ALU instructions without shift take only 1 cycle, fast forward load and store word instructions take 2[9] and 1 cycle, respectively [12, Appendix B].

For FP instructions in the A8, the instruction latencies are explicitly shown in its manual. However, for the INT instructions in the A8, a detailed EXE pipeline model is given [7, Section 16.2], then I had to estimate average result latencies by summing the number of issue cycles[10] to an average result-use stall.

[9] in gem5, there is one cycle of instruction latency plus one cycle to access the L1-D cache.

[10] Also called execution cycles or throughput.

Table 3.5: Configuration of the gem5 Cortex-A9 and A8 FUs for integer and VFP instructions

gem5 FU	gem5 opClass	Out-of-order (Cortex-A9)		In-order (Cortex-A8)	
		Latency	Pipelined	Latency	Pipelined
Simple ALU	IntAlu	1	Yes	1	Yes
Complex ALU	IntMult	4	Yes	5	Yes
FP Unit	SimdFloatAdd	4	Yes	10	No
	SimdFloatCmp	1	Yes	6	No
	SimdFloatCvt	4	Yes	7	No
	SimdFloatDiv	15	No	43	No
	SimdFloatMisc	1	Yes	4	No
	SimdFloatMult	5	Yes	14	No
	SimdFloatMultAcc	8	Yes	22	No
	SimdFloatSqrt	17	No	40	No
Load/Store Unit	MemRead	1	Yes	1	Yes
	MemWrite	1	Yes	1	Yes

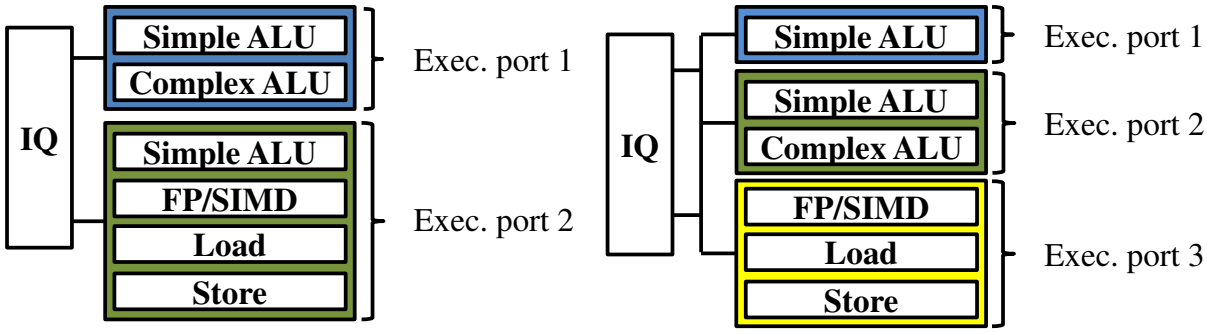


Figure 3.5: The EXE stage configuration of gem5 for the Cortex-A8 (left) and A9 (right).

This average result-use stall (and hence the average result latency) computation is needed because gem5 does not model EXE sub-stages and because several instructions may be regrouped to share the same configuration, while in pipelined processors the result-use stall depends on which sub-stage the result is produced and consumed, which in turn depends on the instruction being executed and the next dependent instruction. I considered that INT instructions on average read their sources at E2, the second EXE stage. Then, the average result latency is $\mathrm{Cycles} + \mathrm{R} - 2$, where Cycles is the number of execution cycles and R is the EXE stage where the result is produced. For instance, simple data-processing instructions with a destination have 1 cycle of execution and produce the result at E2, hence on average the result is produced after $1 + 2 - 2 = 1$ cycle. Simple MUL instructions take 2 cycles to execute and produce their result at E5, then on average their result latency is $2 + 5 - 2 = 5$ cycles.

3.4.2.3 Configuring the tournament branch predictor

gem5 models the tournament branch predictor. A local and a global predictor are updated in parallel, and for each branch instruction, a third predictor (the chooser) decides which one will be used. Given that both Cortex-A9 and A8 have only the global history branch predictor [6, 7], the configuration of the tournament model was adapted. I set the local predictor with the smallest possible size, to reduce its influence, but the chooser was kept intact. The aim is that the local predictor, with a minute size, will loose every prediction to the global one.

3.4.2.4 Micro-architectural differences

Cortex-A9 Compared to the O3 model, the two main differences of the Cortex-A9 are the absence of FP register renaming [8, Table 2-3] and the out-of-order implementation without a traditional ROB [91]. Given that we have no detailed information about its pipeline implementation, I assumed that these micro-architectural differences have a small impact in the execution time. I considered that the Cortex-A9 has a balanced pipeline, i.e., that its structures are large enough to sustain the pipeline width in the absence of stalls. For example, to configure the FP renaming in gem5, I chose a sufficient large pool of physical FP registers (256).

Cortex-A8 Compared to the proposed in-order model, the Cortex-A8 has a considerable more complex micro-architecture. While gem5 only simulates one pipeline for all types of instruction (integer, FP and SIMD), the integer and SIMD/FP pipelines are completely separated in the Cortex-A8. SIMD and FP instructions have to go through the integer pipeline until completion, before being decoded and executed (the NEON unit comprises the FP and SIMD engines) [7, Sections 1.3.6 and 16.5].

EXE stage In gem5, the EXE stage is generically modeled without internal detail. In reality, the latency of each instruction depends on the interaction of previous and concurrent instructions with pipeline structures, not to mention specific hazards, multi-issue restrictions and other complex behaviors that are only captured with detailed models. The average latency technique is a good approximation to estimate the execution time of large portions of code, because local positive and negative timing errors may cancel out each other. If a small number of instructions are executed in loop, gem5 may obtain very poor timing estimations.

3.4.3 Benchmarks

To evaluate the timing accuracy of the models, I selected 10 of the 13 benchmarks of PARSEC 3.0 [28], a modern suite which covers several application domains. Three benchmarks were not tested, because:

- **Canneal**: Does not support ARM yet, because atomic operations need to be ported.
- **Facesim**: The only released input set takes more than one minute to execute, which could take days to be simulated in gem5.

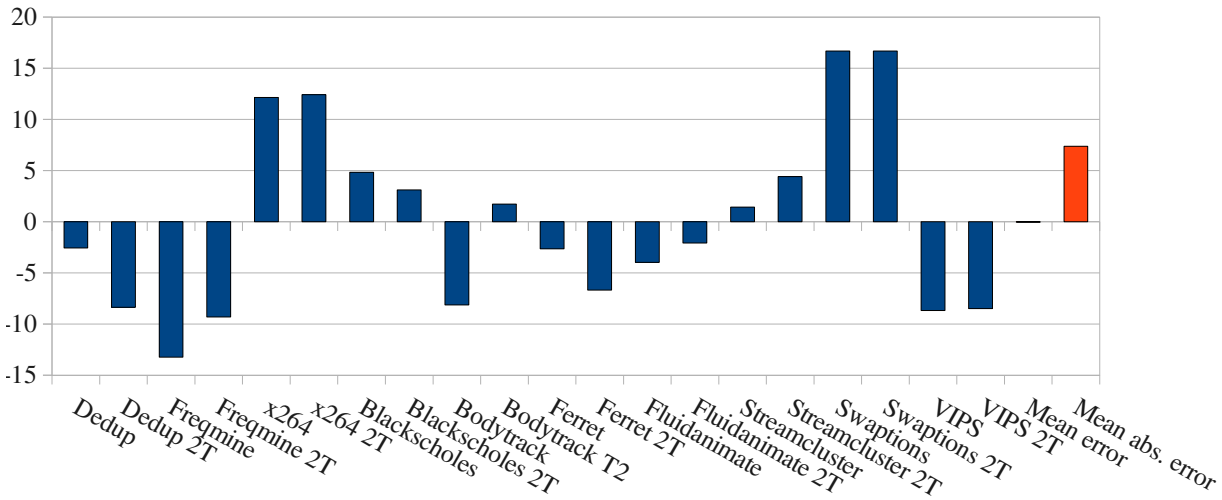


Figure 3.6: The simulation errors (%) of the gem5 Cortex-A9 model.

- **Raytrace**: Depends on X11 development libraries, which were not available in the embedded environment.

To validate the gem5 Cortex-A9 model, I ran the benchmarks with one and two threads (except VIPS with three threads) in order to evaluate the single and dual core behaviors. The A8 model was evaluated only with single-threaded benchmarks.

I used the Ubuntu 11.04 file system released by gem5 to natively compile the benchmarks in the Snowball with gcc 4.5.2. This file system together with the released Linux 2.6.38.8 kernel were used to boot the simulation models in the gem5 FS mode.

The execution time was measured with the built-in `bash` command `time`, whose resolution is the millisecond, and I took the `real` measurements. In gem5, after booting, a script waits 10 s to calm down the initialization processes before running the benchmark under test. This procedure is recommended by a gem5 tutorial [115]. A similar method is carried in the reference boards. The PARSEC suite offers six input sets [28]. I chose the `simsmall`, which is adapted for micro-architectural simulations.

3.4.4 Accuracy evaluation of the Cortex-A models

Table 3.6 details the simulation results of the Cortex-A9 model, while Figure 3.6 shows the simulation errors. Table 3.7 and Figure 3.7(a) show the results of the Cortex-A8 model. The simulations took between 1 and 8 hours. On average, the Cortex-A9 model estimates the execution time with an absolute error of only 7.4 %, ranging from 1 to 17 %. The Cortex-A8 model performs as well as the A9, on average, estimating the execution time with an absolute error of 8.0 %, ranging from 2 to 16 %.

Even considering the generic modeling of gem5, these magnitudes of error can be considered as good results for a micro-architectural simulator. For example, two models of the SimpleScalar simulator were compared to a real Alpha 21264 processor [54]. The `sim-outorder`, which models a generic out-of-order pipeline, showed an average absolute error of 37 %, going up to 77 %. The `sim-alpha`, which models the actual chip, performed better with an average absolute error of 18 %, going up to 43 %. For

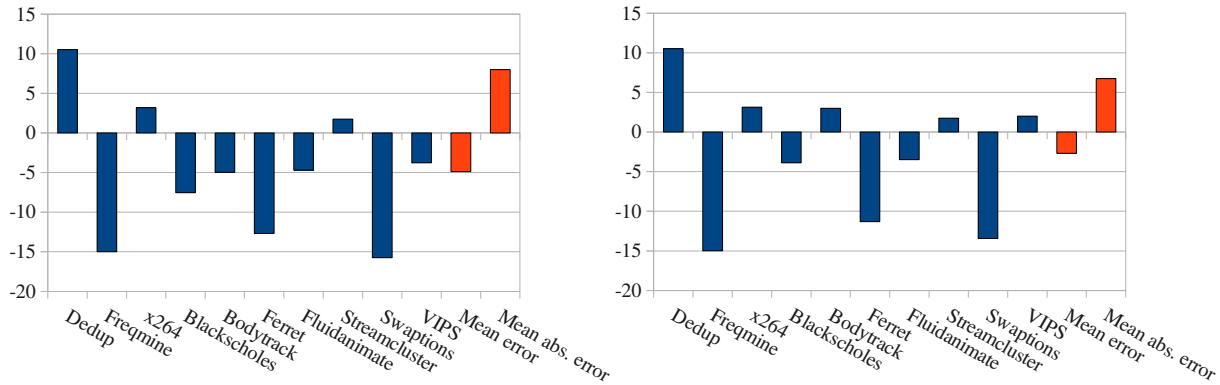
Table 3.6: Comparison of the execution time (seconds) of PARSEC benchmarks. Validation of the gem5 O3 model, configured as a Cortex-A9.

Benchmark		Single thread			Two threads[1]		
		Snowball	gem5	Error (%)	Snowball	gem5	Error (%)
INT	Dedup	2.65	2.58	−2.57	2.02	1.85	−8.36
	Freqmine	4.95	4.30	−13.2	3.46	3.14	−9.30
	x264	6.45	7.23	12.1	3.96	4.45	12.4
INT mean error				−1.22			−1.74
INT mean absolute error				9.31			10.0
FP	Blackscholes	0.600	0.629	4.83	0.353	0.364	3.12
	Bodytrack	2.62	2.41	−8.13	1.62	1.65	1.73
	Ferret	2.80	2.73	−2.64	1.98	1.84	−6.68
	Fluidanimate	3.10	2.98	−3.97	1.94	1.90	−2.07
	Streamcluster	3.48	3.53	1.44	1.76	1.84	4.42
	Swaptions	3.14	3.67	16.7	1.61	1.88	16.7
	VIPS	6.73	6.15	−8.66	3.76	3.44	−8.49
FP mean error				−0.07			1.24
FP mean absolute error				6.62			6.17
Overall mean error				−0.41			0.35
Overall mean abs. error				7.43			7.33

[1] For VIPS, I ran the benchmark with three threads in both platforms.

Table 3.7: Comparison of the execution time (seconds) of PARSEC benchmarks. Validation of the proposed in-order model, configured as a Cortex-A8.

Benchmark		BeagleBoard-xM	gem5	Error (%)
Integer	Dedup	3.40	3.75	10.5
	Freqmine	5.76	4.90	−15.0
	x264	6.50	6.71	3.18
INT mean error				−0.43
INT mean absolute error				9.56
Floating-point	Blackscholes	1.86	1.72	−7.54
	Bodytrack	7.78	7.40	−4.97
	Ferret	7.60	6.64	−12.7
	Fluidanimate	8.33	7.94	−4.68
	Streamcluster	10.3	10.5	1.74
	Swaptions	10.1	8.51	−15.8
	VIPS	14.5	13.9	−3.76
FP mean error				−6.81
FP mean absolute error				7.30
Overall mean error				−4.89
Overall mean abs. error				7.98



(a) Original EXE stage.

(b) Improved EXE stage.

Figure 3.7: The simulation errors (%) of the gem5 Cortex-A8 model.

x86 platforms, PTLSim showed only 5 % of timing error compared to the real AMD's K8 architecture, but the only benchmark considered was the `rsync` command found in Linux systems [140]. Zesto was preliminarily validated against Intel's Merom micro-architecture, showing average absolute errors between 5 and 6 %, although, as the authors noted, the selected micro-benchmarks are too simple to fully evaluate the timing accuracy of their simulator [93]. A more recent x86 simulator, McSimA+, was validate against an Intel Xeon E5540. By comparing the result of a large number of Splash-2 and SPEC CPU2006 benchmarks, McSimA+ achieved an average IPC absolute error of 15.1 %, going up to almost 40 % [1].

In my opinion, most of the timing errors come from the generic purpose of gem5 and the parameters uncertainty. I purposely chose the Cortex-A8 and A9 because they are the only two microprocessors of the Cortex-A series whose EXE stage parameters are publicly available. Such official information is absolutely useful to reduce the parameters uncertainty, but as explained in Section 3.4.2.4, these modern processors have complex micro-architectures, and it is not easy to configure a generic model to simulate them.

3.4.5 In-order model behavior and improvement for a Cortex-A8

First, I analyze the proposed O3 model configuration to mimic an in-order pipeline. Then, I present and evaluate an enhancement of the gem5 modeling of a Cortex-A8.

3.4.5.1 O3 model parameters for in-order simulation

To ensure that the modified O3 model behaves as an in-order pipeline, I verified that the values chosen in Table 3.3, except the delays, do not produce unwanted stalls in the pipeline. For example, by looking at the gem5 statistics, I confirmed that the rename stage does not lack physical registers, because the lack of physical register counter (`FullRegisterEvents`) was virtually zero[11] in all benchmarks.

[11] Only in the Freqmine benchmark, the `FullRegisterEvents` counter was not zero. However, only 22 events out of 4.2 billion renamed operands is negligible. This behavior indicates that I underestimated the maximum number of physical

Table 3.8: FP benchmark results of the improved gem5 Cortex-A8 model with the EXE stage modeling data transfer penalties from NEON to the ARM pipeline.

Benchmark	Error (%)		Abs. error reduction (%)
	Original model	Improved model	
Blackscholes	−7.54	−3.88	48.6
Bodytrack	−4.97	2.99	39.8
Ferret	−12.7	−11.3	10.9
Fluidanimate	−4.68	−3.48	25.6
Streamcluster	1.74	1.74	0[1]
Swaptions	−15.8	−13.4	14.8
VIPS	−3.76	2.02	46.2
Mean	−6.81	−3.62	26.6
Absolute Mean	7.30	5.55	

[1] Streamcluster had no improvement because only a negligible number of VMOV instructions transferring data from NEON to the ARM pipeline are executed.

In addition, I also inserted warning messages in the code, for example, to tell if instructions are issued out-of-order. In more detail, I compared the fetch sequence number of the issuing instructions to check that this number never decreases (except instructions re-issuing, which are verified separately). All benchmarks issued all instructions in-order.

3.4.5.2 Modeling FP data transfer penalties of the Cortex-A8

The validation of the Cortex-A8 model shows that an out-of-order model can be adapted to approximately simulate an in-order pipeline with good precision. However, as Table 3.7 shows, on average, the gem5 Cortex-A8 model is faster than the BeagleBoard-xM. Among FP benchmarks, 6 of 7 are estimated with a faster execution time than the real board. In order to explain this behavior, I therefore focused the analysis on the FP benchmarks. I discovered that VMOV instructions have latency penalties when data is transferred from NEON/VFP to the ARM pipeline [7, Section 16.5.2]. This means that the original EXE stage model in gem5 tends to be faster, because such penalties are not modeled.

To confirm this hypothesis, gem5 was modified with the VMOV instructions that move data from NEON to ARM regrouped in a separate operation class. This new class has a latency of 20 cycles and instructions can be issued back-to-back (i.e., the FU is pipelined to those operations) [7, Section 16.5.2]. Table 3.8 compares the simulation error of the original gem5 model and the improved model, executing FP benchmarks, and Figure 3.7(b) shows the new simulation errors. On average, this modification reduced the simulation error of FP benchmarks by 27 %. The graph in Figure 3.8 shows the reduction of the absolute percentage error obtained by the improved model correlated with the number of such VMOV instructions. The correlation is not perfect, because the slowdown and in consequence the error reduction depends on the relative place of such VMOV instructions and the critical paths, for example. This improved Cortex-A8 model, on average, estimates the execution time of the ten benchmarks with an absolute error of 6.8 %, ranging from 2 to 15 %.

registers used (Table 3.3). Indeed, in the worst case, each instruction in the ROB could have two destination registers.

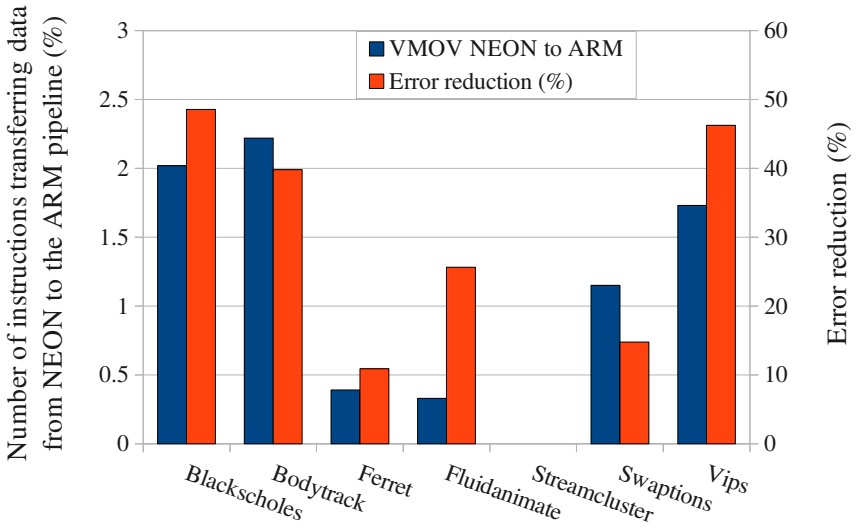


Figure 3.8: Correlation between the number of VMOV instructions transferring data from NEON to the ARM pipeline and the error reduction obtained by the introduction of delay penalties is such instructions (Cortex-A8 model).

3.5 Area and relative energy/performance validation

This section presents the area and relative energy/performance validation of the estimations made by the proposed simulation framework.

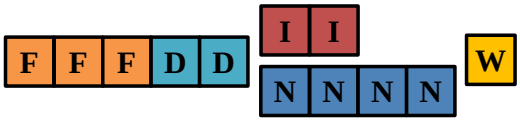
I decided to perform the area evaluation because there are published data and floorplans, and the energy cost per access and leakage are proportional to the size of components. Hence, the area comparison provides a partial validation of the energy models.

This thesis is interested in the relative energy and performance of different code versions executed in varying pipeline configurations taking big.LITTLE systems as reference. Therefore, I decided to compare the simulation framework to real big.LITTLE CPUs.

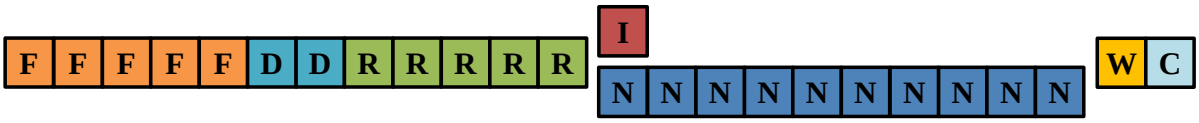
First, this section describes the hardware, simulation platforms and benchmarks, then it presents the validation results of area and relative energy/performance.

3.5.1 Reference models

The system parameters of this experiment are based on the ODROID-XU3 board, which embeds an Exynos 5244 Application Processor. This processor is a big.LITTLE system with two clusters of four cores each, one with Cortex-A7 cores, and the other with Cortex-A15.



(a) Cortex-A7.



(b) Cortex-A15.

F = Fetch, D = Decode, R = Rename/Dispatch/Issue, I = INT ALU, N = FP/SIMD, W = Writeback, C = Commit

Figure 3.9: Configuration of pipeline stage depths for the Cortex-A7 and A15 in gem5.

3.5.2 Simulation models

Table 3.9 shows the main system and CPU parameters, taken from the board specifications and ARM manuals [8, 9, 13, 86]. For both cores, instruction timing in the EXE stage is not publicly available. To configure them in gem5, I proceeded as follows. In both cores, integer instructions have the same latencies of one, four and twelve cycles for ALU, multiply and divide, respectively. For floating-point, I estimated their timing taking as reference the gem5 configuration for the Cortex-A9. For the A7, I subtracted one cycle (two for floating-point multiply-accumulate (FMAC)), because it has a tightly integrated VFP/NEON to the ARM pipeline, compared to the rectangular design of VFP/NEON in the A9. For the A15 I multiplied the A9 latencies by $10/4 = 2.5$, which is the ratio of VFP pipeline depths. Table 3.10 shows the latencies of main instructions. Table 3.1 presents examples of instructions that each gem5 opClass can execute. Based on published ARM diagrams [71], I configured the pipeline stage depths and EXE stages as depicted in Figures 3.9 and 3.10. The branch unit is not simulated, resulting in four and seven execution ports for the A7 and A15, respectively. In consequence, the partial dual-issue in the A7 and the peak issue width of eight instruction in the A15 are not simulated.

I assumed that cores and caches are synthesizable with the low operating power (LOP) transistors, then McPAT was configured accordingly. The technology node was set to 28 ns and the temperature fixed at 27 °C. With this configuration, both L2 caches and the A7 respect the timing constraints of the transistor technology[12], but the L1 caches in the A15 does not. The maximum frequency in the A15 to respect their timing constraints is 1.5 GHz, and hence I am overestimating this limit by 33 %. Table 3.11 summarizes the normalized FU area and energy costs of the reference and studied cores (following the equation proposed in Section 3.2.2).

For energy estimations I did not consider the snooping unit, because the clusters were compared with only one core activated.

[12]This information was obtained by temporarily activating the `opt_local` flag.

Table 3.9: Parameters of Cortex-A7 and A15 CPUs

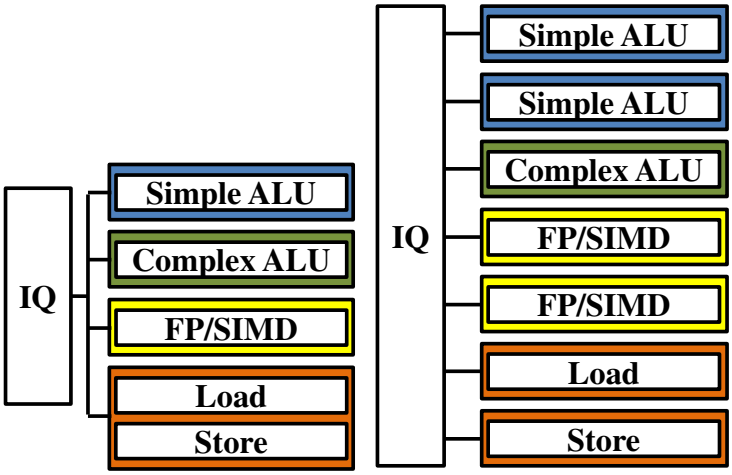
Parameter[1]		Cortex-A15 (out-of-order)	Cortex-A7 (in-order)
Core clocks		2.0 GHz	1.4 GHz
DRAM	Size / clock / latency (ns)	256 MB[2] / 933 MHz / 81[3]	256 MB[2] / 933 MHz / 81[3]
L2	Size / assoc. / lat. / MSHRs / WBs	2048 kB / 16 / 8 / 11 / 16	512 kB / 8 / 3 / 8 / 16
L1-I	Size / assoc. / lat. / MSHRs	32 kB / 2 / 1 / 2	32 kB / 2 / 1 / 2
L1-D	Size / assoc. / lat. / MSHRs / WBs	32 kB / 2 / 1 / 6 / 16	32 kB / 4 / 1 / 4 / 4
SP	Cache level / degree / buffer size	2 / 1 / 16	1 / 1 / 8[3]
BP	Global / Local History entries (bits)	4096[4] (2^4) / 1024[4] (3^4)	256 (2^3) / N/A
	BTB / RAS entries	4096[4] / 48	256[5] / 8
I-TLB / D-TLB entries		128 each[6]	32 each[6]
Front-end / back-end width		3 / 7[7]	1[7] / 1[7]
INT / FP pipeline depth		15 / 24	8 / 10
Physical INT / FP registers		90[3] / 256[3]	1068[8] / 1096[8]
IQ / LSQ / ROB entries		48 / 16 each / 60	16[3] / 8 each[3] / 512[8]

[1] Latencies in core clock cycles, except for DRAM.
[2] gem5 FS mode limitation.
[3] Educated guess.
[4] Based on Alpha 21264.
[5] The A7 implements a small BTB and an equivalent structure holding a few instructions.
[6] Both A7 and A15 have two levels of TLB. Here, I-TLBs and D-TLBs are over-dimensioned to compensate the absent second level.
[7] The A7 is partial dual-issue, while the A15 has a peak issue width of eight instructions. Here, in both cores the branch unit is not simulated.
[8] These structures do not exist in an in-order pipeline and the chosen values are explained in section 3.1.3.2.

Table 3.10: Configuration of the gem5 Cortex-A7 and A15 FUs for integer and VFP instructions

gem5 FU	gem5 opClass	Cortex-A15 (out-of-order)		Cortex-A7 (in-order)	
		Latency	Pipelined	Latency	Pipelined
Simple ALU	IntAlu	1	Yes	1	Yes
Complex ALU	IntMult	4	Yes	4	Yes
	IntDiv	12	No	12	No
FP/SIMD Unit	SimdFloatAdd	10	Yes	3	Yes
	SimdFloatMult	12	Yes	4	Yes
	SimdFloatMultAcc	20	Yes	6	Yes
Load/Store Unit[1]	MemRead	3	Yes	1	Yes
	MemWrite	2	Yes	1	Yes

[1] The A15 can issue one load and one store per cycle. In gem5, this is simulated by separated LSUs. Here, I regrouped them for simplicity.



(a) Cortex-A7 (b) Cortex-A15

Figure 3.10: The EXE stage configuration of gem5 for the big.LITTLE CPUs.

Table 3.11: Normalized FU area and energy costs for a Cortex-A7, A9 and A15.

Core	Width	Area		Energy per access (EPA)		Inst. lat.		Energy per cycle	
		INT	FP/SIMD	INT	FP/SIMD	INT	FP/SIMD	INT	FP/SIMD
Generic	W	$\propto W^2$		$\propto W^2$		L		EPA/L	
A7	1	0.25	0.25	0.25	∼0.25	1	∼1	0.25	∼0.25
A9	2	1	1	1	1	1	1	1	1
A15	3	2.25	2.25	2.25	2.25	1	2.5	2.25	∼1

3.5.3 Benchmarks

Dhrystone is a very simple benchmark. However, processor manufacturers still employ it to compare relative energy and performance. In addition, results of big.LITTLE CPUs running this benchmark were published [71]. Nonetheless, because the detailed environment was not described, I used Dhrystone 2.1 and assumed similar CPU configurations. The benchmark is compiled with Code Sourcery gcc 4.7.2 with the flags `-static -mthumb -O3 -mcpu=cortex-a15`.

To better evaluate the energy and performance trade-offs of the simulated CPUs, the 10 PARSEC benchmarks presented in Section 3.4.3 were also used.

3.5.4 Area validation

Table 3.12 shows the core, L2 and cluster area estimation compared to published data. For core area, the estimations showed an error of only 3.6 % for the A15 and exactly matched the area of the A7. In the cluster estimations, I did not model the snooping unit, which can explain the underestimations of -13 and -1.4 % for the A7 and A15 clusters, respectively. Based on the McPAT example configuration for the A9, the snooping unit may represent less than 1 % of the A15 cluster and around 4 % in the A7.

Table 3.12: Area validation (mm^2 at 28 nm) of core and cluster estimated with McPAT 1.0

Cluster	Comp.	McPAT	Publ.	Error (%)	Ref.
A7	Core	0.45	0.45	0.0	[17]
	L2	1.52	-	-	-
	Total[1]	3.32	3.8	−13	[60]
A15	Core	3.21	3.1	3.6	[126]
	L2	5.88	-	-	-
	Total[1]	18.7	19	−1.4	[60]

[1] In McPAT, it is the area of four cores plus the L2 cache, the snooping unit was not accounted.

Table 3.13: Validation of relative area estimations compared to published data for the Cortex-A7 [19]

Component(s)		Area (%)		Error (%)	Weighted error (%)
McPAT	ARM A7 floorplan	McPAT	ARM A7		
Instruction fetch unit	PFU, I-cache, ICU	24.6	25.6	−3.63	−0.93
Execution unit	DPU	34.7	38.2	−9.40	−3.60
Memory management unit	TLB	1.77	4.45	−60.2	−2.68
Load/store unit	STB, D-cache, DCU	29.7	24.4	21.8	5.31
Other	BIU	9.25	7.36	25.7	1.89

The floorplan of the Cortex-A7 at 28 ns was published [19]. Based on that information, I evaluated the area estimated by McPAT configured as a A7. Table 3.13 presents the relative areas of five main structures in the core. The greatest difference is in the TLB, where McPAT underestimates its area by 60 %. This can be explained by the lack of second level of TLB in the McPAT model. However, if we weight the errors by the relative core area of each structure, the LSU has the greatest error of only 5.3 %.

Figure 3.11 shows the relative area of core and structures of both modeled cores. The A7 is seven times smaller than the A15. The structures of the big core are not as balanced as those in the small one: the EXE stage in the A15 occupies 82 % of the core, while in the A7 it represents only 35 %. Unfortunately, the A15 floorplan was not published to validate the results. Nonetheless, if we look at the Cortex-A9 floorplan [15], its EXE stage represents around 60 % of the core area. If we consider that the A15 embeds two FP/NEON units instead of one, the estimations are reasonable for the A15. Another possible explanation is that McPAT underestimated core structures other than the EXE stage (as commented in Section 2.4.3.7), and this latter is overestimated because of the rough FU area estimations provided by Eq. 3.1.

The area validation presented here demonstrates that McPAT can correctly estimate the area of a Cortex-A7 and provide acceptable estimation for an A15. By composing core and L2 estimated areas, the area estimations of big.LITTLE clusters were partially validated. These estimated areas are consistent with published data and insure that McPAT may also make coherent energy estimations, given that energy costs per access and leakage are proportional to the area of components.

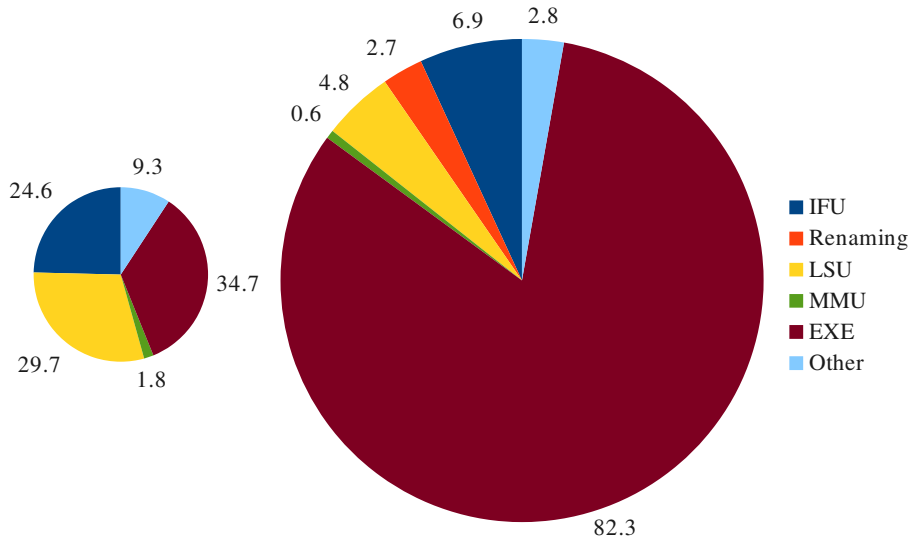


Figure 3.11: Relative core and structure areas of the modeled Cortex-A7 (left) and Cortex-A15 (right).

3.5.5 Relative energy/performance validation

As a relative energy and performance validation of the simulation framework, I present here trade-offs between Cortex-A7 and A15 CPUs. To highlight the energy and performance differences of those cores, I simulated only one active core in each cluster, running single-threaded benchmarks. The energy comparisons take into account the active core and the L2 cache.

Table 3.14 shows the relative energy and performance of the A7 and A15 clusters. In the Dhrystone benchmark, the A15 provided a speedup of 1.84, while the A7 consumes 3.69 times less energy. These results are very close to those published by ARM [71]: 1.9 and 3.5, respectively. In the PARSEC benchmarks, we observe varying degrees of trade-offs. Ferret has a speedup of only 1.11 in the A15, with an energy efficiency of 5.47 in the A7. x264 showed the greatest speedup of 2.46 in the A15, exchanged by an energy efficiency of 3.08 in the A7. On average, the A15 is 1.5 times faster, but the A7 consumes 4.1 times less energy.

It is believed that A15 cores provide speedups of 2-3x and that A7 cores are 3-4x more energy efficient, while the simulation results showed that the A15 provide only an average speedup of 1.5. One explanation is that the benchmarks are compiled with high optimization levels, compared to moderately optimized binaries and libraries or legacy-applications found in real-life, which out-of-order pipelines can accelerate. Another explanation comes from the nature of the PARSEC suite, which focus on thread-level parallelism instead of instruction-level parallelism (ILP), what superscalar pipelines are designed for. Indeed, x264 and VIPS that showed the greatest speedups have higher ILP compared to the others.

Figures 3.12 and 3.13 show the average energy consumption of PARSEC benchmarks per structure in the Cortex-A7 and A15, respectively. If we divide the CPUs into five main pipeline structures plus the L2 cache, in the A7, the L2 cache is the main energy consumer, mostly contributing with leakage. In the A15 cluster, the EXE stage is the most energy hungry, because of its high energy cost. In the in-order core, L1 caches and the FP/SIMD unit are large structures frequently accessed, and hence contribute both with dynamic and leakage energy, while small components with high activity comprise the decoder, load/store queue, register files and bypass buses, which contribute mainly with dynamic energy. In the out-of-order

Table 3.14: Relative energy and performance of Cortex-A7 and A15 CPUs (one core active)

Benchmark	A15 speedup	A7 energy eff.
Dhrystone	1.84	3.69
Blackscholes	1.14	4.42
Bodytrack	1.45	4.09
Dedup	1.46	4.29
Ferret	1.11	5.47
Fluidanimate	1.15	5.29
Freqmine	1.45	4.32
Streamcluster	1.32	3.88
Swaptions	1.34	4.12
VIPS	1.89	3.56
x264	2.46	3.08
Geometric mean	1.47	4.15

core, the EXE stage consumes most of the energy, with the most energy-hungry component in the core being the FP/SIMD units, followed by the instruction scheduler, integer register renaming, registers files, instruction buffer and decoder, which are also examples of small structures highly accessed.

3.6 Example of architectural/micro-architectural exploration

To illustrate the design space exploration capabilities of the simulation platform, I simulated the performance of two hypothetical dual Cortex-A8 processors (the Cortex-A8 does not support multi-core configurations [8, Table 2-3]). In both processors, I configured two Cortex-A8 cores using the in-order model of Section 3.4.5.2 connected to a shared L2 cache. The difference resides in the FP pipeline: one model has the original non-pipelined VFP unit, while the other has a pipelined version with the same parameters as the Cortex-A9 VFP. This latter model offers a more fair comparison between almost equivalent in-order and out-of-order pipelines.

Table 3.15 and Figure 3.14 show the speedup of the dual over the mono A8 and the speedup of the dual A9 over the dual A8.[13] On average, the dual Cortex-A8 allows a speedup of 1.76 over its mono-core version, achieving an almost perfect speedup in the Swaptions and Streamcluster benchmarks. The comparison of the out-of-order dual A9 and the in-order dual A8 is more interesting: for integer benchmarks, there is no significant speedup (between 1 and 6 percent is beyond the simulation precision), which may be explained by the fact that small-latency integer operations benefit less from out-of-order pipelines than long-latency floating-point ones. For FP benchmarks, as expected, the original non-pipelined VFP in the A8 leads to poor results compared to the A9 equipped with a pipelined VFP, on average executing FP benchmarks 2.5 times slower. On the other hand, an A8 equipped with the same VFP as in the A9 shows relatively better performance with an average slowdown factor of 1.24, for FP benchmarks. This performance difference comes not only from the dynamic scheduling capability of the Cortex-A9, but may also come from differences in their memory systems: for example, slower memory frequency,

[13] Some results presented here differ from those presented at SAMOS XIV [63], because I fixed a bug in the in-order model, which only impacted FP pipelined instructions.

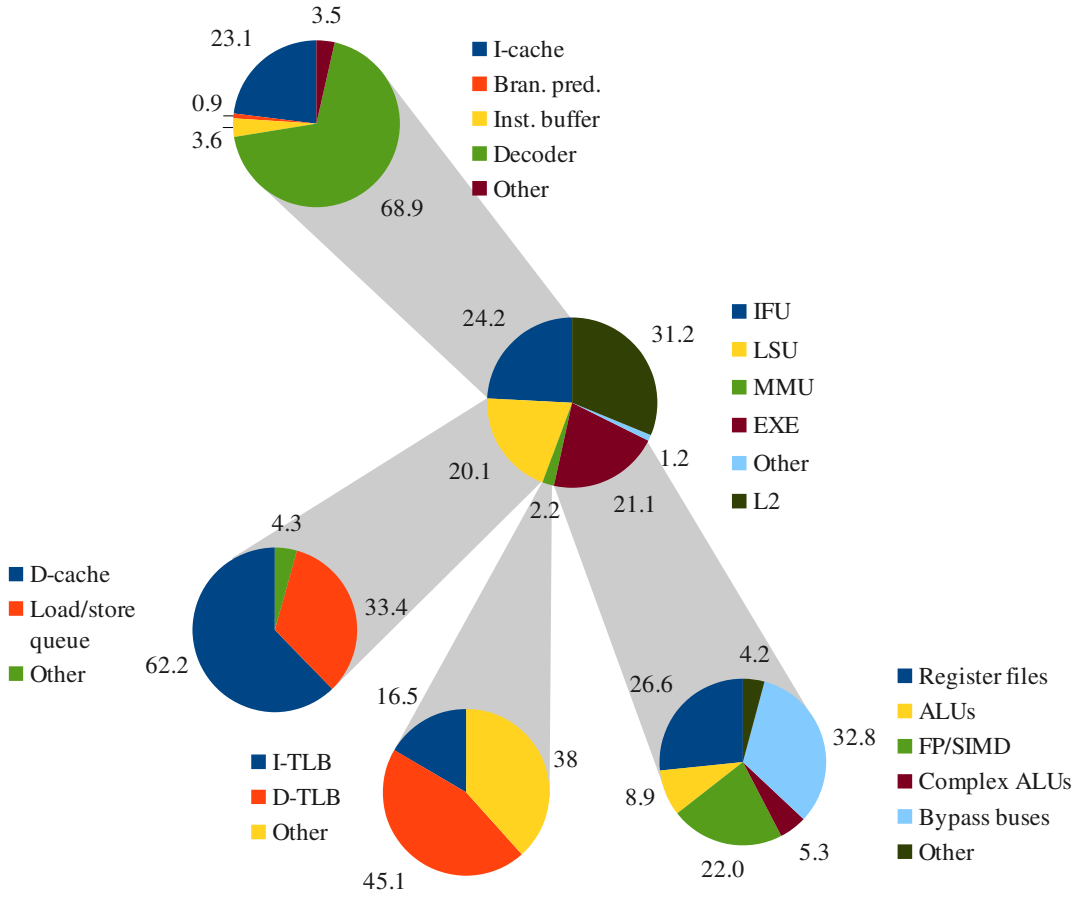


Figure 3.12: Average energy consumption of PARSEC benchmarks per structure in the Cortex-A7.

smaller L2 cache and the blocking L1 caches in the Cortex-A8.

This experiment shows how useful micro-architectural simulators can be. They allow researchers to explore the architectural and micro-architectural design space, to test new ideas and to simulate hypothetical or emerging hardware implementations.

3.7 Scope and limitations

Any simulator has a scope of validity and limitations. Previous work reported modeling, specification and abstraction errors in gem5 [37, 72, 107] and McPAT [138]. Instead, this section discusses the usage scope and the limitations of the simulation framework.

3.7.1 Scope

Micro-architectural simulators based on analytical models are adapted to simulate new or hypothetical configurations of processors. The flexibility of analytical models permits to easily modify the

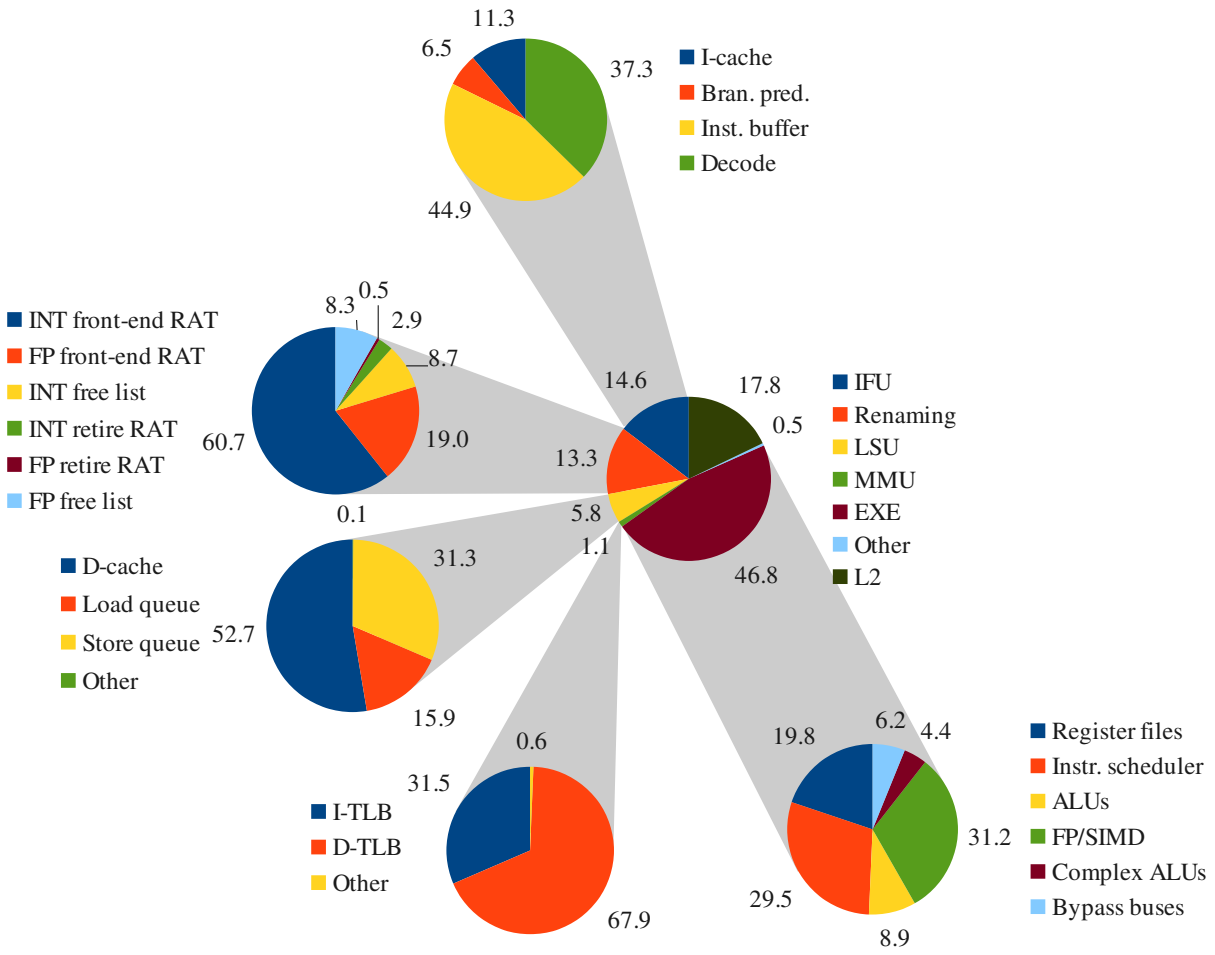
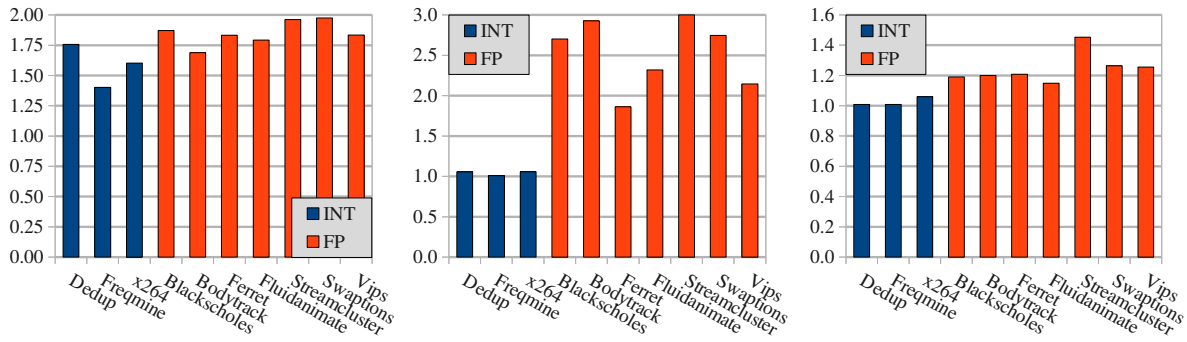


Figure 3.13: Average energy consumption of PARSEC benchmarks per structure in the Cortex-A15.



(a) Speedup of dual A8 vs. mono A8. (b) Speedup of dual A9 vs. dual A8 (original non-pipelined VFP). (c) Speedup of dual A9 vs. dual A8 (pipelined VFP as A9).

Figure 3.14: Speedups of hypothetical dual Cortex-A8 processors running PARSEC benchmarks.

behavior of the simulator to quickly evaluate new ideas, or to explore a design space.

Table 3.15: Performance of PARSEC benchmarks executed in hypothetical dual Cortex-A8[1] processors, compared to the single Cortex-A8 model[1] and the dual Cortex-A9 (Snowball).

Benchmark[2]		Speedup of dual over mono A8 (Original VFP[3])	Speedup of dual A9 over dual A8	
			Original VFP[3]	Same VFP as A9[3]
INT	Dedup	1.76	1.06	1.01
	Freqmine	1.40	1.01	1.01
	x264	1.60	1.06	1.06
INT geometric mean		1.58	1.04	1.03
FP	Blackscholes	1.87	2.70	1.19
	Bodytrack	1.69	2.93	1.20
	Ferret	1.83	1.86	1.21
	Fluidanimate	1.79	2.32	1.15
	Streamcluster	1.96	3.04	1.45
	Swaptions	1.97	2.75	1.26
	VIPS	1.83	2.15	1.26
FP geometric mean		1.85	2.50	1.24
Overall geometric mean		1.76	1.92	1.17

[1] Based on the improved Cortex-A8 model of Section 3.4.5.2.

[2] The mono-core A8 ran the benchmarks with one thread, while the dual-core processors ran with two threads, except VIPS with three.

[3] The original VFP extension in the A8 is not pipelined, while in the A9 the VFP is pipelined for most instructions.

3.7.2 Limitations

For energy and performance estimations of existing processors, unless the desired measurement can not be performed in a real platform, measuring directly in the hardware is the best choice. In addition, calibrated ISSs are considerably faster than micro-architectural simulators and can usually provide estimations with better accuracy.

gem5 and McPAT are generic simulation frameworks. They were configured as close as I could to simulate ARM cores, but they can not represent the real pipeline behavior. Although we have seen acceptable global performance and relative energy estimations, the simulated models probably show very different behaviors at the micro-architectural scale and cycle-level timing.

The proposed in-order model for gem5 is not cycle-accurate. Studies targeting specific processor components should employ a cycle-accurate model. My cycle-approximate model could be only used if the impact of the model approximation over the studied component is negligible.

This thesis studies the heterogeneity of cores. Indeed, the proposed in-order CPU model in gem5 was not integrated as a new CPU model and hence it is not possible to simulate a multi-core composed of in-order and out-of-order ARM cores. It is likely that the O3 and the new Minor CPU models may be already used to simulate such a system in gem5.

3.8 Conclusion

In this chapter, I presented a micro-architectural simulation framework based on gem5 and McPAT. I not only improved some modeling aspects in both simulators, but also proposed a cycle approximate in-order model for gem5. A detailed description of the output translation of gem5 to feed the power and area models in McPAT was presented. The framework allows to estimate the relative energy and performance of real or hypothetical ARM cores, as I demonstrated through validation experiments and an example of architectural and micro-architectural exploration. Hence, this simulation framework is a good starting point to study new hardware implementations or to explore the design space of embedded processors.

This chapter contributed by showing that:

- The O3 model in gem5 has considerably lower timing errors than similar micro-architectural simulators. In 20 configurations of PARSEC 3.0 benchmarks, I demonstrated that gem5 has average absolute errors of only 7.4 %, oscillating between 1 and 17 %. In contrast, similar micro-architectural simulators also extensively validated show average absolute errors greater than 15 %.
- McPAT can estimate the area of big.LITTLE cores with good accuracy (within 4 %), given a few modifications in the source code and a generic area scaling rule for FUs.
- My simulation framework provides acceptable estimations of the relative energy and performance trade-offs of big.LITTLE CPUs. For the Dhrystone 2.1 benchmark, the relative estimations are within 6 %.

The simulation framework presented in this chapter allowed to simulate several pipeline configurations, used to evaluate the capability of the proposed auto-tuning system to adapt code to different micro-architectures, and to estimate the energy and performance differences of similar in-order and out-of-order designs, running statically compiled codes or online auto-tuned ones (Chapter 5).

The work developed in this chapter has already been used by other research teams in CEA. We also received around 10 requests by external researchers, which asked for the source code or configuration files used in the experiments. As a future prospect, it would be interesting to contribute back to the gem5 and McPAT communities, because participating in open source projects can be an effective way to promote the work developed during this thesis.

Chapter 4

Run-time code generation

This chapter describes deGoal [43], a run-time code generator of computing kernels, mostly used in embedded systems. This tool is developed at CEA[1] in the LIALP laboratory, where this thesis was prepared.

deGoal was designed considering the constraints of embedded systems: energy, memory and computing power. Typical JITs and dynamic compilers require too much memory and permanent storage space, which are not available in most small embedded environments.

In this thesis, I ported deGoal to high-performance ARM processors of the Cortex-A series. Although such processors, which are present in smartphones and tablet computers, have enough resources to run complex software stacks, including typical JIT frameworks, run-time code optimization systems[2] are usually developed for desktop and server computers. In consequence, these optimization systems are mostly adapted to platforms that have a lot of computing resources and run relatively heavier workload than embedded systems. Therefore, the objective was to develop deGoal for ARM and demonstrate that run-time auto-tuning could be pushed to embedded processors running small workload.

Toward this objective, this chapter first presents in Section 4.1 the deGoal front-end, which is mostly architecture independent. Then, Section 4.2 details the new features added to deGoal and the porting of its back-end to the ARM ISA. Section 4.3 shows a preliminary performance evaluation of computing kernels in ARM processors. Finally, Section 4.4 discusses the application scope and limitations of deGoal and the study developed in the chapter, before the conclusion in Section 4.5.

4.1 deGoal: a tool to embed dynamic code generators into applications

deGoal implements a DSL for run-time code generation of computing kernels. It defines a pseudo-assembly RISC-like language, which can be mixed with standard C code. The machine code is only generated by deGoal instructions, while the management of variables and code generation decisions are

[1] The French Atomic Energy Commission (Commissariat à l'énergie atomique et aux énergies alternatives).

[2] In the context of this thesis, we are interested in run-time code optimization systems whose performance is comparable to statically compiled code. Therefore, interpreters and virtual machines are not considered.

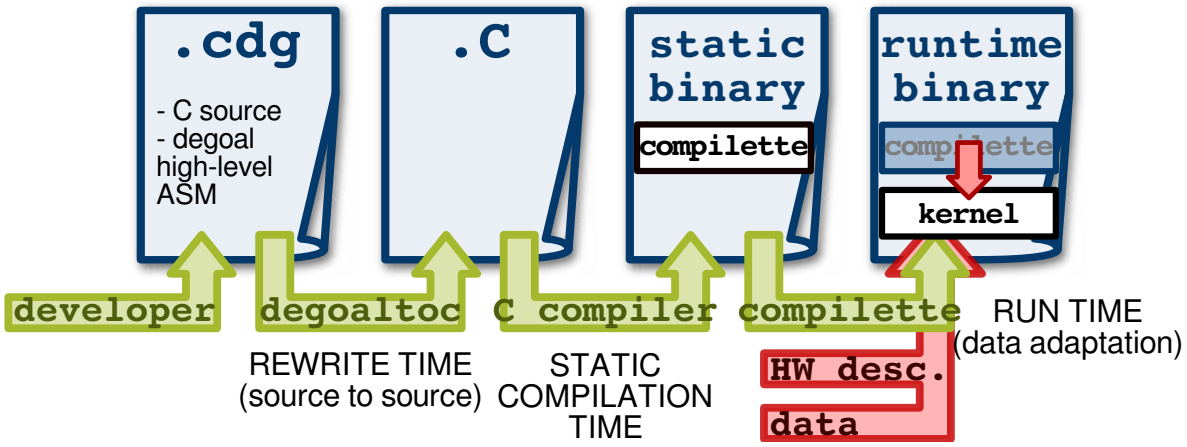


Figure 4.1: deGoal workflow: from source code to run-time code generation.

implemented by deGoal pseudo-instructions, optionally mixed with C code. The dynamic nature of the language comes from the fact that run-time information can drive the machine code generation, and help to generate highly optimized kernels.

Because no high-level language nor IR is processed at run-time, in an ARM processor deGoal generates code four orders of magnitude faster than the LLVM Execution Engine jitting bitcode with the `-O3` optimization level [44, 94].

4.1.1 Utilization workflow

Figure 4.1 presents the deGoal usage workflow. deGoal code can be embedded into any C source file with the extension ".cdg", to describe a kernel generator, also called *compilette* in the deGoal jargon. A python program (`degoaltoc`) is then called to translate .cdg files to standard C code calling deGoal functions. In the current implementation, the target ISA is statically defined through an input parameter passed to `degoaltoc`. Then, the application is linked to the deGoal back-end (library or C headers) and compiled by any standard C compiler. Except one gcc intrinsic used to flush an address range from the cache hierarchy, deGoal is not dependent on a specific compiler. When the application is running, the compilette generates optimized machine code, thanks to the run-time information, such as hardware description and input data, which drives instruction and data specialization.

4.1.2 Example of kernel implementation: C with and without SIMD intrinsics and deGoal versions

To better illustrate the language definition, an example of (squared) euclidean distance computation from the Streamcluster benchmark (PARSEC 3.0 suite [28]) written in C and implemented with deGoal is presented. The same code implemented with SIMD compiler intrinsics (PARVEC suite [39]) is shown, because deGoal transparently supports vector instructions.

Figure 4.2 shows the original C implementation. `Point` is a structure that contains the `dim` coordinates of a point, among others. In lines 12 and 13, the difference of respective coordinates of two points

```
1  typedef struct {
2    float weight;
3    float *coord;
4    long assign;
5    float cost;
6  } Point;
7
8  float dist(Point p1, Point p2, int dim)
9  {
10     int i;
11     float result = 0.0;
12     for (i = 0; i < dim; i++)
13         result += (p1.coord[i] - p2.coord[i])*(p1.coord[i] - p2.coord[i]);
14     return(result);
15 }
```

Figure 4.2: Euclidean distance computation in Streamcluster from PARSEC.

```
1  float dist(Point p1, Point p2, int dim)
2  {
3      float ret;
4      int i;
5      _MM_TYPE result, _aux, _diff, _coord1, _coord2;
6
7      result = _MM_SETZERO();
8      for (i=0;i<dim;i=i+SIMD_WIDTH) {
9          _coord1 = _MM_LOADU(&(p1.coord[i]));
10         _coord2 = _MM_LOADU(&(p2.coord[i]));
11         _diff = _MM_SUB(_coord1, _coord2);
12         _aux = _MM_MUL(_diff,_diff);
13         result = _MM_ADD(result, _aux);
14     }
15     ret = (_MM_CVT_F(_MM_FULL_HADD(result, result)));
16     return ret;
17 }
```

Figure 4.3: Euclidean distance computation in Streamcluster from PARVEC. SIMD_WIDTH $= 4$ for ARM.

is squared and accumulated.

Figure 4.3 presents the same kernel implemented in PARVEC. Here, the vectorization is manually implemented by using SIMD compiler intrinsics. In this implementation, the dimension of the points is considered as a multiple of 4 (not shown in the code).

Figure 4.4 details the compilette implemented with deGoal, in this case equivalent to PARVEC. Data processing, load, store and data pre-fetching instructions in deGoal have the following formats (optional operands are between brackets):

```
add dest, src1[, src2]
lw dest, base[, offset[, stride]]
sw base[, offset[, stride]], src
pld base[, offset]
```

The operands may not only be registers and vectors, but also immediate or dynamically assigned

```
1  Begin code Prelude out = weight1, coord1,
       assign1, cost1, weight2_, coord2_,
       assign2_, cost2_, dim_
2  Type int_t int 32
3  Alloc int_t i, coord2
4  Type fpvec_t float 32 4
5  Alloc fpvec_t Vc1
6  Alloc fpvec_t Vc2
7  Alloc fpvec_t Vdiff
8  Alloc fpvec_t Vaux
9  Alloc fpvec_t Vresult
10     mv coord2, coord2_
11     mv i, dim_
12     mv Vresult, #(0)
13     while_ge i, #(4)
14         lw Vc1, coord1
15         lw Vc2, coord2
16         sub Vdiff, Vc1, Vc2
17         mul Vaux, Vdiff, Vdiff
18         add Vresult, Vresult, Vaux
19
20         add coord1, coord1, #(4*4)
21         add coord2, coord2, #(4*4)
22         sub i, i, #(4)
23     whileend
24 Type fp_t float 32
25 Alloc fp_t result
26     add result, Vresult
27     mv out, result
28     rtn
29 End
```

Figure 4.4: Euclidean distance computation (PARVEC-like) with deGoal.

values, further described in Section 4.1.5. For clarity, vector variables start with a capital V in deGoal codes presented here.

Still in Figure 4.4, lines 13, 22 and 23 represent the C-equivalent `for` instruction, but here the loop index is reversed for simplicity. Lines 14 to 18 perform the loads, difference, squaring and sum of coordinates, exactly as lines 9 to 13 in the PARVEC code (Figure 4.3). In both cases, the computation is performed over vectors of 4 elements, then the final result is the sum of the elements of the result vector, represented in the lines 15 in Figure 4.3 and 26 in Figure 4.4.

4.1.3 The `Begin` and `End` commands

The dynamic generation of a function is delimited by the deGoal `Begin` and `End` commands. The definition of `Begin` is the following:

```
Begin BUFFER GEN_OPT OUT_REGS = IN_REGS
```

`BUFFER` is a C pointer to a user allocated memory space, where the function will be generated. `GEN_OPT` is a code generation option to indicate if function prologue and epilogue should be generated or not. It can be `Prelude`, to automatically manage the stack and registers, or `Leaf`, otherwise. Optionally, `OUT_REGS` is a list of the output register names, followed by the '=' sign and `IN_REGS`, a list

of input registers that names the function parameters.

The `End` command does not receive any parameter.

Example of `Begin` command The code from Figure 4.4 generates executable instructions in a memory buffer `code`, defined in line 1. Continuing in this line, the input and output parameter of the function are declared. The output value will be in the `out` register, while input registers are listed as comma-separated tokens after the sign equal. For while, the procedure call standard must be statically defined. In the case of Figure 4.4, it is the EABI standard for ARM, which allows up to four integer registers to be used to pass integer parameters. The rest must be passed through the stack. For clarity, I named stack-passed parameters with an ending underline (`weight2_` to `dim_`). These inputs are considered as virtual registers and they must be moved to allocated physical registers before any manipulation (e.g., lines 10-11).

4.1.4 Register allocation

Register types are defined through the `Type` command and physical register allocation/deallocation are defined by `Alloc` and `Free`. The definition of `Type` is as follows:

```
Type TYPE_NAME ARITH BITS VECTOR_SIZE
```

`TYPE_NAME` is any valid name to identify the defined type. `ARITH` indicates the used arithmetic to process the register content. Currently, `int`, `float` and `complex` are valid options. `BITS` represents the width of the data type definition. `VECTOR_SIZE` is an optional argument to set the length of vector of registers.

The `Alloc` and `Free` commands are defined as:

```
Alloc TYPE_NAME VAR_NAME ARRAY_SIZE
Free VAR_NAME
```

`TYPE_NAME` must be an already defined type and `VAR_NAME` is the name of variables to be allocated/freed. `ARRAY_SIZE` is an optional argument to allocate an array of registers or an array of vector of registers. Array of registers and vector of registers are different: in the first each element must be accessed through an index (like a C array) to be accepted by an instruction, whilst the vector of registers is directly used as an operand and results in a vector-like instruction. The `Free` command is optional if no new variables will be allocated.

Example of register allocation In Figure 4.4, in the lines 2 and 24, 32-bit integer and FP registers are declared respectively as `int_t` and `fp_t`, while line 4 declares a four-element 32-bit FP vector named `fpvec_t`. Registers are allocated in the respective lines following the type declaration. The allocation of an array `A` of two vectors with type `fpvec_t` is obtained with the following statement:

```
Alloc fpvec_t A 2
```

Then, each vector in the array can be accessed through an index:

```
lw A[#(i)], rbase
```

Where, `i` is any C expression that results in a valid index, in this example `i` must result in 0 or 1, during program execution.

4.1.5 Code generation decisions: deGoal mixed to C code

Dynamic code generation decisions are implemented directly in C. There are two ways to mix the languages: instructions written in C can be mixed with deGoal instructions, and values in deGoal statements can be written with C expressions.

Mixing C and deGoal instructions In order to differentiate C from the dynamic language, a block of deGoal instructions are preceded by the sign '`#[`' and succeeded by '`]#`'. For instance, a conditionally generated ADD instruction can be implemented with the C instruction `if`, and a loop can be unrolled with a `for`:

```
if (generate_add) {
#[  add rdest, rsrc1, rsrc2 ]#
}

for (i = 0; i < unrollFactor; ++i) {
#[  lw rdest, rbase ]# // Load "rdest"
#[  add rdest, #(1) ]# // Increment "rdest"
#[  sw rbase, rdest ]# // Store "rdest"
#[  add rbase, #(4) ]# // "rbase" points to the next element
}
```

C expressions in deGoal statements Another important symbol in the language is the triplet sign '`#()`'. Any C expression placed inside the parenthesis will be dynamically evaluated, allowing the specialization of run-time constants, such as variable sizes, vector lengths and operands of instructions. They are generically referred as "run-time constants". Of course, immediate values can also be inserted in the code. For instance, a run-time constant can be inlined into the code as an operand:

```
add rdest, #(userInput)
```

In another example, instead of a fixed vector type declaration:

```
Type fpvec_t float 32 4
```

the declaration of a vector with a dynamically defined length `vectLen` would be:

```
Type fpvec_t float 32 #(vectLen)
```

This kind of construction allows for instance to dynamically instantiate vectors whose lengths are multiples of the underlining hardware SIMD-width, specially useful if it was not known prior compilation.

4.1.6 Branches and loops

Branch instructions are implemented as follows:

```
    bCC LABEL, CMP_VAL, REF_VAL
    (...)
LABEL:
    (...)
```

`CC` is a condition code, such as `eq` (equal), `ge` (greater than or equal). `LABEL` is a valid name that identifies the target of the branch. It can be put before of after the branch instructions using it. `CMP_VAL` and `REF_VAL` are the value under comparison and the reference value, respectively. They can be registers, or a run-time constant for `REF_VAL`.

Loop instructions were created to alleviate the programming effort and improve code readability. Two kind of loops are supported. The `loop/loopend` pair and the `while/whileend`, defined in the following:

```
loop NUM_ITER
    (...)
loopend
```

`NUM_ITER` indicates the number of iterations to perform. It can be a register or a run-time constant.

```
while_CC CMP_VAL, REF_VAL
    (...)
whileend
```

`CC`, `CMP_VAL` and `REF_VAL` have the same definitions as in the case of branch instructions. The loop body will be iterated while the condition applied to the comparison and reference values is true.

4.2 Thesis contribution: New features and porting to ARM processors

deGoal was originally designed for run-time code specialization in constrained embedded systems, such as DSPs and micro-controllers. In contrast, the ARM Thumb-2 ISA is used in a broad range

Table 4.1: Number of source code lines of deGoal: architecture-independent and ARM architecture.

Module		Number of lines	Description
Architecture-independent		4516	Source-to-source compiler (`degoaltoc`) and ISA description to C header converter (`isatocdg`).
ARM architecture	Common	3036	Generic supporting code for all the other modules.
	Instruction scheduler	1525	Instruction scheduler for all types of instructions.
	Core	2930	Register allocation and instruction generation of ARM Core (INT) instruction.
	FP	1655	Idem for FP instructions (VFP extension).
	SIMD	3337	Idem for SIMD instructions (NEON extension).
	ISA	2244	ISA description files.
	Total	14727	

of computing units, from low-power micro-controllers, such as the Cortex-M0, to high-performance embedded processors, as those from the Cortex-A series.

This thesis is mainly interested in code optimization for high-performance embedded processors. Hence, I focused on the ARMv7-A architecture (the "Application profile"), which supports FP and SIMD instructions. This architecture regroups the 32-bit processors of the Cortex-A series.

The ISA-dependent source code in deGoal is a highly tuned library written in C. This part of the code implements the system that manages the stack, allocates registers, selects, schedules and generates instructions at run-time. For ARM this part corresponds to around 14.7 thousand lines of source code, while the architecture-independent code in deGoal has around 4.5 thousand lines. Table 4.1 details these numbers, further dividing the ARM source code in six main modules.

This section presents the new features and the porting of deGoal to the Thumb-2 ISA. First, it overviews the new language constructs and code generation support for ARM. Then, it further details the main features that allow deGoal to generate code to high-performance embedded cores.

4.2.1 Overview of contributions

The new features added to deGoal are summarized in the following:

- Language constructs:
  - **The `while` instruction**: The ARM port introduced this new instruction in deGoal. The definition of this instruction was presented in Section 4.1.6.
  - **Stack-passed parameters**: Currently, this feature is only supported for ARM. It was needed to implement computing kernels from complex benchmarks. An example of usage was presented in Section 4.1.3.
  - **Two dimensional arrays**: An array of vectors is useful to unroll loops that process vectors. Using different vectors to process each group of unrolled instructions can avoid pipeline stalls, because of register reuse. An example was given in Section 4.1.4.

- Back-end for ARM:
  - **Support for ARM Core instructions**: The basic pipeline implementation of any core is called ARM Core. The basic ISA comprises INT and special instructions to control coprocessors. The register allocator and instruction generators for the ARM Core were ported in deGoal.
  - **FP and SIMD support**: I also ported in deGoal the FP and SIMD instructions (VFP and NEON extensions, respectively), including the register allocator, which was extended to support the FP/SIMD register aliasing. Both are transparently supported in the language through the type definition. Single instruction, single data (SISD) or SIMD machine instruction generation for both INT and FP is selected through a code generation option. FP and SIMD code generation is indispensable in high-performance computing. Section 4.2.2 presents an example of source and generated codes processing INT and FP vectors, with SISD or SIMD code generation.
  - **Configurable instruction scheduler**: I ported the instruction scheduler to ARM and extended it to model a configurable EXE stage, needed to generate optimized code to various pipeline configurations. Section 4.2.3 details its implementation.
  - **Static and dynamic configuration**: The deGoal back-end for ARM can be compiled to a specific target core and with fixed code generation options (static configuration) or with dynamically configurable hardware support and code generation options (dynamic configuration). In the latter case, I developed a set of C functions that can be called in the host application or inside a compilette to configure the code generation. Section 4.2.4 describes the implemented configurations.

The next sections detail the main implementations and give examples of generated code for ARM.

4.2.2 SISD and SIMD code generation

The code generation for the basic Thumb-2 ISA and the VFP/NEON extensions was ported in deGoal. Figure 4.5(a) shows an example of source code that loads an eight-element INT vector starting from the memory address `addr`, multiplies each element by 10 and stores the result. Figure 4.5(c) shows the generated code with ARM Core instructions. While load and store multiple registers exist and are used in lines 2 and 12, the multiplications are performed by eight instructions between lines 4 and 11. On the other hand, Figure 4.5(d) shows the generated code when the NEON extension is activated. Given that SIMD instructions can process up to four 32-bit elements, the multiplications are performed by two instructions in the lines 6 and 7.

Figure 4.5(b) shows the same source code except that the vector type was changed to `float`. Figures 4.5(e) and 4.5(f) show the generated codes without and with SIMD support, respectively.

Because deGoal is a one-pass code generator, some simple optimizations as removing NOP instructions (used as placeholders, but eventually are not replaced by useful instructions) can not be performed. This explains the four NOPs between lines 2 and 5 in Figures 4.5(e) and 4.5(f), used to reserve place to eventually save the context of FP registers.

```
1 Begin code Prelude addr
2 Type  vec_t int 32 8
3 Alloc vec_t v
4     lw v, addr
5     mul v, v, #(10)
6     sw addr, v
7     rtn
8 End
```

(a) Source code: INT.

```
1 Begin code Prelude addr
2 Type  vec_t float 32 8
3 Alloc vec_t v
4     lw v, addr
5     mul v, v, #(10)
6     sw addr, v
7     rtn
8 End
```

(b) Source code: FP.

```
1  stmdb    sp!, {r4, r5, r6, r7, r8, r9}
2  ldmia.w  r0, {r1, r2, r3, r4, r5, r6, r8, r9}
3  movs     r7, #10
4  muls     r1, r7
5  muls     r2, r7
6  muls     r3, r7
7  muls     r4, r7
8  muls     r5, r7
9  muls     r6, r7
10 mul.w    r8, r8, r7
11 mul.w    r9, r9, r7
12 stmia.w  r0, {r1, r2, r3, r4, r5, r6, r8, r9}
13 ldmia.w  sp!, {r4, r5, r6, r7, r8, r9}
14 bx       lr
```

(c) Generated code with INT SISD: ARM Core.

```
1  stmdb    sp!, {r7}
2  mov      r7, r0
3  vld1.32  {d28-d31}, [r7]!
4  movs     r7, #10
5  vmov.32  d2[0], r7
6  vmul.i32 q14, q14, d2[0]
7  vmul.i32 q15, q15, d2[0]
8  mov      r7, r0
9  vstmia   r0!, {d28-d31}
10 mov      r0, r7
11 ldmia.w  sp!, {r7}
12 bx       lr
```

(d) Generated code with INT SIMD: NEON.

```
1  stmdb    sp!, {r7}
2  nop
3  nop
4  nop
5  nop
6  vldmia   r0, {s8-s15}
7  movw     r7, #0
8  movt     r7, #16672      ; 0x4120
9  vmov     s4, r7
10 vmul.f32 s8, s8, s4
11 vmul.f32 s9, s9, s4
12 vmul.f32 s10, s10, s4
13 vmul.f32 s11, s11, s4
14 vmul.f32 s12, s12, s4
15 vmul.f32 s13, s13, s4
16 vmul.f32 s14, s14, s4
17 vmul.f32 s15, s15, s4
18 vstmia   r0, {s8-s15}
19 ldmia.w  sp!, {r7}
20 bx       lr
```

(e) Generated code with FP SISD: VFP.

```
1  stmdb    sp!, {r7}
2  nop
3  nop
4  nop
5  nop
6  mov      r7, r0
7  vld1.32  {d28-d31}, [r7]!
8  movw     r7, #0
9  movt     r7, #16672      ; 0x4120
10 vmov     s4, r7
11 vmul.f32 q14, q14, d2[0]
12 vmul.f32 q15, q15, d2[0]
13 mov      r7, r0
14 vstmia   r7!, {d28-d31}
15 ldmia.w  sp!, {r7}
16 bx       lr
```

(f) Generated code with FP SIMD: VFP + NEON.

Figure 4.5: Example of vector multiplication with INT and FP data types in deGoal, showing the source and generated codes with and without the VFP and NEON extensions.

4.2.3 Configurable instruction scheduler

deGoal embeds one-pass architecture-specific instruction schedulers. For ARM, the EXE stage is modeled with a configurable number of execution ports, which can contain one or more FUs. Each assembly instruction is mapped to one FU and is modeled with an issue and result latency. This model is based on the EXE stage modeling in gem5 (Section 3.1.1.1), but in deGoal instructions can have their own parameters, or parameters shared with a group of instructions by employing C macros to set the

latency and issue cycles.

The common scheduling implementation mainly consists of an instruction buffer (IB), in which each entry holds the machine instruction encoding (the payload) and also register and FU dependencies (scheduling information).

When an instruction should be generated, at first the payload is produced by joining together opcode and operands. Then, the instruction scheduler is called, receiving the information needed to fill one entry in the IB. The entry is found by regressively overtaking previously inserted instructions if register and FU dependencies are respected. This simple and fast heuristic reduces result-use cycle stalls and tries to maximize multi-issues in superscalar pipelines. Because overlapping loads and stores are not verified, deGoal does not currently support memory aliasing when the scheduling is enabled.

The oldest entry in the buffer is forced to write its payload into program memory if there is no space left in the buffer. Otherwise, the buffer accepts new entries without flushing. Some instructions, such as branches, are considered as scheduling barriers and force all entries to be flushed, because no subsequent instruction is allowed to execute before them.

4.2.4 Static and dynamic configuration

Static configuration When the target core is fixed at compile time, which is usually the case in micro-controllers, a very lightweight run-time system and a small library can be obtained if the target configuration is statically fixed and unused features, as FP/SIMD support, are not compiled. This is obtained through a set of compilation directives. The ISA selection, i.e., ARM, Thumb-1 and Thumb-2, is partially supported (besides the Thumb-2, the ARM ISA has a minimal support). The FP and SIMD selection is working. The EXE stage model of four ARM cores (Cortex-A7, A8, A9 and A15) are available options. I also ported the emulation of divide instructions for the A8 and A9, which lack an INT division unit.

Dynamic configuration On the other hand, some binaries may execute in different target cores, not necessarily known at compile time. Hardware virtualization and software packages[3] are two examples. Therefore, in this case the dynamic target configuration is useful. When it is enabled, a C function can be called prior to the code generation of a kernel to configure the target core and set code generation options. The dynamic target configuration includes FP, INT/FP SIMD support, divide emulation and activation of the instruction scheduler. Dynamic configuration is also useful for run-time auto-tuning, allowing the evaluation of the performance impact of various code generation options.

Experimental code generation options A few experimental code generation options were implemented. The most interesting are the stack minimization and the instruction decompression. In the first option, non-scratch FP registers are removed from the allocable list of registers. The aim is to reduce the stack management overhead in the function prologue/epilogue, which may slowdown small kernels. Of course, the positive or negative performance result depends on the register reuse in the code and also on the throughput of the LSU. The option of instruction decompression was created because most Thumb-1 (16-bit) data-processing instructions always update the status register. This register dependence may

[3] Software released through a compiled code.

create bubbles in the pipeline, slowing down the execution. One trade-off is to replace them by Thumb-2 (32-bit) instructions, which usually have a bit in their opcode to update or not the status register, at the cost of increasing the code size. The instruction decompression option can also decompose macro-instructions into simpler instructions, e.g., some load/store multiple registers and FMAC instructions. The trade-off in this case is the code size vs a better instruction scheduling in in-order pipelines (an example is given in Section 4.3.6.3).

4.2.5 Further improvements and discussion

Some interesting features are not supported yet by deGoal. This section lists and discusses possible improvements and new features in deGoal.

Data size support In the original specification, the data type and size of operations is defined by variables, not by instructions. However, in the current implementation, load and store operations define the data size to process, for instance `lw`, `lh` and `lb`, to load a word, half-word or byte, respectively. Such instructions reduce the code portability.

Dynamic configuration of parameter passing Currently, the number of function parameters and the calling convention are statically configured in deGoal. Supporting dynamic configuration of the calling convention is required to allow run-time code generation in different computing architectures.

The `rtn` instruction and `OUT_REGS` In the current implementation, the `rtn` instruction inserts code for context saving and restoration. Actually, this task should be performed by the `End` command, which indicates the end of a function context. Instead, the `rtn` instruction should receive an optional register to return a desired result. This would make the list of output registers `OUT_REGS` unnecessary in the `Begin` command and also fix the following misbehavior. Indeed, in architectures like ARM, the register R0 defines at the same time the first register holding (part of) an input (let's say `IN0`) and/or an output parameter (let's say `OUT0`). In consequence, in some cases writing into `OUT0` before reading `IN0` would overwrite the first function parameter.

High-level strength reduction Currently, the instruction selection is directly implemented in the back-end. Simple strength reduction rules could be created in the deGoal source-to-source compiler, for instance to replace deGoal multiply instructions with a run-time constant operand by a series of conditional calls to multiply or shift instruction generators, depending on the value of the immediate operand. Doing such simple tasks in the architecture-independent side would simplify the architecture-dependent code of all supported architectures.

Configurable multi-pass code generation The very fast one-pass code generation is responsible for the low run-time overhead of the tool. However, the fixed one-pass strategy can not perform simple but efficient optimizations, such as removing redundant or dead instructions (e.g., NOP instructions in Section 4.2.2).

4.3 Performance analysis

This section details the experimental environment and the performance evaluation of deGoal in ARM processors.

4.3.1 Evaluation boards

The evaluation boards used in the experiments are the BeagleBoard-xM (Cortex-A8) and Snowball (Cortex-A9), already described in Section 3.4.1. In both platforms, I enabled the RunFast mode: in this mode, commonly used VFP instructions are executed in the NEON unit, which makes the performance comparison of NEON and VFP instructions fairer. Nevertheless, when the RunFast mode is enabled FP instructions can not bypass results in the BeagleBoard-xM [7, Section 16.7.2].

4.3.2 Benchmarks and deGoal kernels

I implemented with deGoal four kernels of two benchmarks, VIPS and Streamcluster, from the PARSEC 3.0 suite [28] and its manually vectorized version PARVEC [39]. PARSEC 3.0 is a modern parallel benchmark suite. In total, I implemented and compared eight kernels: half generate SISD instructions and the other half SIMD instructions. I chose VIPS and Streamcluster because they are kernel-based benchmarks and have been manually vectorized, providing reliable SIMD implementations.

In the VIPS benchmark, three kernels were evaluated and the execution times were measured per kernel with performance counters. In the Streamcluster benchmark, one kernel was evaluated, and the measured performance corresponds to the execution time of the benchmark, given that the considered kernel accounts for more than 90 % of its execution time. I ran the benchmarks using the `simlarge` input set. The execution time of the kernels is between a few seconds in VIPS and a few minutes in Streamcluster.

I compiled the eight reference kernels and the equivalent deGoal versions in an Ubuntu 11.04 distribution for ARM with gcc 4.5.2. All kernels are compiled to the Thumb-2 ISA and the ARMv7-A architecture, and all 32 single-precision registers can be used. Other important compiler options are (default PARSEC flags): `-O3 -g -funroll-loops -fprefetch-loop-arrays`. The generic instruction scheduler in deGoal is parametrized with the Cortex-A8 instruction latencies (the A9 has similar latencies). To ensure fair comparisons, deGoal does not model specific instruction behaviors of the chosen cores and deGoal did not generate special instructions rarely used by compilers. Special instructions can speedup program execution and are potential usage cases for a low-level tool as deGoal.

4.3.3 Raw performance evaluation

This section presents a preliminary performance evaluation of deGoal for ARM processors. The objective is to evaluate the raw performance of machine code generated by deGoal, by translating kernels found in benchmarks into the deGoal language and comparing their performance. Hence, deGoal is simply used like a static code generator, but code is generated at run time. Because deGoal generated

Table 4.2: Speedup of deGoal reference-like kernels compared to PARSEC and PARVEC (run-time overheads of deGoal are negligible).

Benchmark	Kernel	BeagleBoard-xM		Snowball	
		PARSEC	PARVEC	PARSEC	PARVEC
VIPS	Linear transformation	1.61	1.06	1.00	1.00
	Interpolation	1.14	0.59	0.87	0.68
	Convolution	1.03	1.98	1.04	1.60
Streamcluster	Euclidean distance	1.18	1.00	1.00	0.95
Geometric mean		1.22	1.05	0.98	1.01

functionally equivalent codes as the reference implementations, I called them *PARSEC-like*, *PARVEC-like* or *reference-like* deGoal kernels/versions.

Table 4.2 and Figure 4.6 present the speedup of deGoal versions compared to the reference kernels. Table 4.3 also presents the kernel execution times in the BeagleBoard-xM and Snowball. On the BeagleBoard-xM, deGoal is on average 22 % faster than PARSEC, thanks to simple and efficient optimizations such as loop unrolling and better ILP, explicitly coded in a low-level language. Compared to PARVEC, deGoal is 5 % faster on average. In the Snowball, the dynamic scheduling pipeline probably speeded up under-optimized code, and the average deGoal performance virtually matched those of PARSEC and PARVEC.

Table 4.4 shows the speedups when the instruction scheduler in deGoal is enabled (default option) compared to the code generation without scheduling. We observe speedups from -2 to 11 %, with positive speedups on average, even in the Snowball which has an out-of-order pipeline. Because of energy efficiency constraints, the Snowball has limited out-of-order resources[4], which explains why it benefits from the instruction scheduling.

In the following, I detail the code and performance differences of each kernel.

4.3.3.1 Linear transformation

This kernel applies a linear transformation on a buffer. The original code is implemented with two nested loops, the first iterating over the buffer elements and the second over the image layers.

In the PARSEC version, the inner loop is partially unrolled for different factors, while with deGoal I partially unrolled three times (common assumption of a three-layer image being processed). The excess instructions generated by the compiler to partially unroll for different factors explain why deGoal is 61 % faster in the BeagleBoard-xM. On the other hand, in the Snowball, there is no performance difference between the PARSEC and deGoal versions. It is likely that the out-of-order execution could hide the latency of the extra instructions by executing them during cycles that otherwise would be stalled in an in-order pipeline.

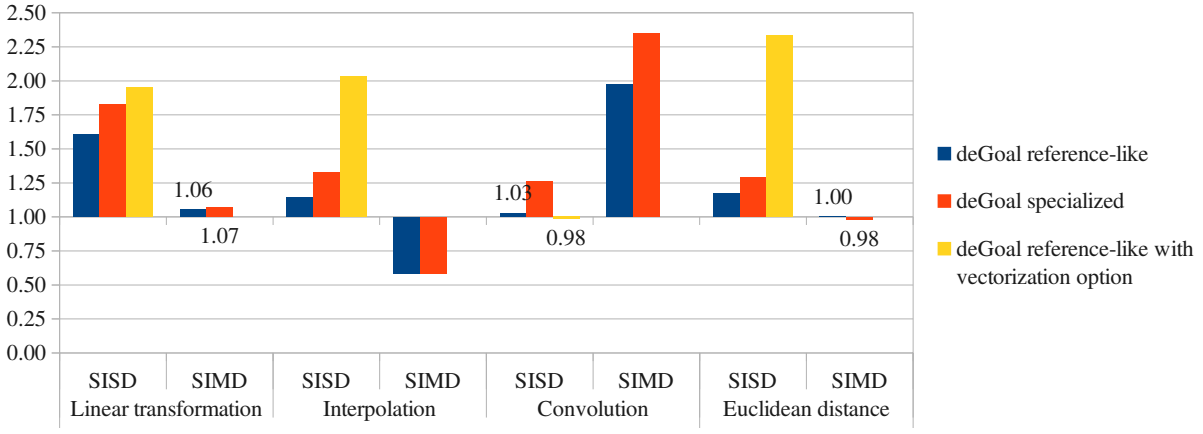
The PARVEC version specializes the number of image layers to completely unroll the inner loop

[4]Because the Cortex-A9 does not implement a ROB [91], the instruction window and hence its reordering capability is very limited, compared to a classic out-of-order pipeline.

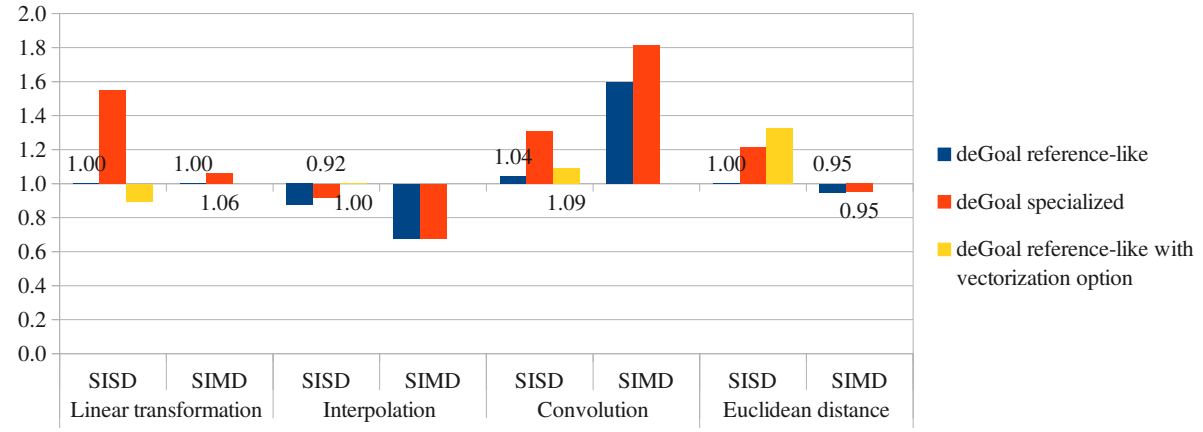
Table 4.3: Execution time (seconds) of reference kernels, deGoal reference-like and dynamically specialized kernels (all run-time overheads included).

Benchmark	Kernel	Version	BeagleBoard-xM				Snowball			
			Ref.	deGoal RL	deGoal S	deGoal RLV	Ref.	deGoal RL	deGoal S	deGoal RLV
VIPS	Linear transf.	SISD	8.17	5.08	4.48	4.18	3.37	3.36	2.17	3.77
		SIMD	2.38	2.25	2.22	N/A	2.44	2.43	2.30	N/A
	Interpolation	SISD	5.51	4.82	4.15	2.71	2.26	2.58	2.45	2.26
		SIMD	1.27	2.17	2.17	N/A	1.20	1.78	1.78	N/A
	Convolution	SISD	5.13	4.98	4.07	5.21	5.22	5.02	3.98	4.79
		SIMD	10.4	5.24	4.41	N/A	7.84	4.91	4.31	N/A
Streamcluster	Euclidean dist.	SISD	147.8	125.7	114.3	63.2	90.2	90.4	74.3	68.1
		SIMD	63.4	63.1	64.6	N/A	64.5	68.1	67.8	N/A

deGoal RL: Reference-like deGoal kernel.
deGoal S: Specialized version with deGoal.
deGoal RLV: Reference-like deGoal kernel with the vectorization option enabled.



(a) BeagleBoard-xM (Cortex-A8).



(b) Snowball (Cortex-A9).

Figure 4.6: Speedup of deGoal reference-like and dynamically specialized kernels over the reference codes (all run-time overheads included).

Table 4.4: Speedup of deGoal reference-like kernels with the instruction scheduler enabled. The reference is code generation without scheduling.

Benchmark	Kernel	BeagleBoard-xM		Snowball	
		PARSEC	PARVEC	PARSEC	PARVEC
VIPS	Linear transformation	1.00	1.01	1.01	1.03
	Interpolation	1.03	1.00	1.05	1.03
	Convolution	1.11	1.04	1.07	0.98
Streamcluster	Euclidean distance	1.01	1.00	1.02	1.00
Geometric mean		1.04	1.01	1.04	1.01

three times. Even comparing to a manually unrolled loop, deGoal is 6 % faster in the BeagleBoard-xM and matches the PARVEC performance in the Snowball.

4.3.3.2 Interpolation

The interpolation kernel is a small piece of code that performs a bilinear interpolation of an image point from four neighbor pixels. There is only one loop that iterates over the image layers. Although this kernel is called a sufficient number of times to be a good candidate for deGoal, given that only a few instructions are executed during each call, I observed that not inlining the function can produce considerable overheads. Differently from gcc, deGoal can not inline machine code in an existing function yet.

In the PARSEC version, the loop is also partially unrolled for different factors, which explains the 14 % of speedup of deGoal in the BeagleBoard-xM. However, in the Snowball, deGoal slows down the execution by 13 %, which is explained by the function inlining performed by gcc. If the no-inlining of the this small kernel is forced, deGoal matches the performance of PARSEC in the Snowball.

Regarding the PARVEC comparison, besides the performance degradation from function inlining, the manually vectorized version uses a trick to improve performance: in some vectors of three elements, it loads one extra element out-of-bound. This optimization reduces the number of load instructions.[5] Given that deGoal can not perform such optimization yet, it contributes to a slowdown of 41 and 32 % in the BeagleBoard-xM and Snowball, respectively.

4.3.3.3 Convolution

The convolution kernel is a separable integer convolution. Instead of performing a full 2D convolution, this technique first applies a 1D mask and then the same mask rotated by 90°. In the experiments, only the first step was compared, which is a kernel by itself with two nested loops.

In PARSEC, the inner loop is unrolled by using a technique called *Duff's Device* [57]. It partially unrolls a loop 16 times by using a switch-case statement in C. The main advantage of employing this technique is that there is no need to process leftover iterations in a separate loop. deGoal does not support such implementation, then I had to implement a main (partially-unrolled) and a left-over loop. The only advantage over the Duff's Device is that instructions may be better scheduled in the main loop. The results show that deGoal is only 3 to 4 % faster than the C version.

In PARVEC, the technique of partially unrolling and leftover processing is used. Surprisingly, deGoal is 1.60 to almost 2 times faster than the manually vectorized code, but this performance comes from an under-optimized vectorization: this kernel processes integer elements, however in PARVEC the actual computation is performed in floating-point precision, which results in performance degradation because of integer to FP conversions, and vice-versa. I questioned the main author of PARVEC about this apparent bug, but he declared that in Intel platforms there is no performance difference if INT or FP instructions are used. Also, he wanted to have the same SIMD implementation for all supported platforms, then because the AVX extension did not have the needed INT instructions he had to implement the INT/FP conversions. This is an example of the better performance portability provided by deGoal, because either the INT instructions, or the INT/FP conversions and the FP instructions could be transparently generated depending on the target platform.

[5] Two NEON instructions are needed to load three consecutive words in a quadword register, but one NEON instruction can load four consecutive words in a quadword register.

4.3.3.4 Euclidean distance

The PARSEC implementation of this kernel is shown in Figure 4.2. As explained in Section 4.1.2, it computes the euclidean distance of two points using FP precision. The only loop iterates over the dimension of the points which is a benchmark parameter.

As in the previous kernels, gcc partially unrolls the loop up to eight times. Without any information about the dimension of the points, with deGoal I decided to partially unroll the loop four times, in order to match it with the ARM SIMD width. This value will allow to demonstrate the capability of the transparent vectorization provided by deGoal, explored in the Section 4.3.4. Another difference is that the vector-like processing in deGoal increases the ILP. While gcc generates a code that reuses one register to accumulate the partial result with a chain of dependent FMAC, with deGoal I accumulated those results in different registers. In the BeagleBoard-xM, this produces a speedup of 18 %, specially because the increased ILP compensates the impossibility to bypass FP results, which creates a result-use stall of at least seven cycles [7, Section 16.7.2][6]. In the Snowball no speedup was observed, because the NEON pipeline in the Cortex-A9 has a special multiply-accumulator forwarding [11, Section 3.4.2], which reduces the result-use stall to only three or four cycles, compensating the lack of ILP in the PARSEC code.

Comparing with PARVEC, deGoal shows no speedup in the BeagleBoard-xM and a slow-down of 5 % in the Snowball. In the latter case, the generic loop unrolling performed by gcc shows its advantage: with deGoal I did not unroll the loop, given that no information about the dimension was assumed, while gcc partially unrolls the loop up to eight times.

4.3.4 Transparent vectorization: SISD vs SIMD code generation

In this section, the transparent vectorization enabled by deGoal is demonstrated. Since I implemented compilettes with vector-like processing, it is possible to generate SISD or SIMD instructions just by toggling a code generation flag. In other words, the PARSEC-like kernels written with deGoal in Section 4.3.3 can transparently generate SIMD instructions.

Table 4.5 shows the speedup of the SIMD generated kernels compared to the SISD generated ones, with deGoal. In this table, I compared to the SISD versions because there was no need to rewrite the code. Figure 4.6 also shows the speedups compared to the reference kernels and Table 4.3 presents the kernel execution times in the BeagleBoard-xM and Snowball. On average, the SIMD versions are 43 % faster in the BeagleBoard-xM, and only 9 % in the Snowball. The observed speedup differences and slowdowns are discussed in the Section 4.3.6.1.

4.3.5 Dynamic code specialization

One of the main usages of deGoal is as a fast dynamic code specializer [43]. To demonstrate this capability, I used deGoal to dynamically specialize input parameters in the four studied benchmarks.

[6] Although ARM states that all VFP instructions that execute in the NEON pipeline take seven cycles to execute, it is likely that the FMAC takes at least ten cycles, as its SIMD version, because it has to pass first through the multiply and then through the add pipeline.

Table 4.5: Speedup of deGoal PARSEC-like kernels when generating SIMD code compared to SISD code generation.

Benchmark	Kernel	BeagleBoard-xM	Snowball
VIPS	Linear transformation	1.22	0.89
	Interpolation	1.78	1.14
	Convolution	0.96	1.05
Streamcluster	Euclidean distance	1.99	1.33
Geometric mean		1.43	1.09

Table 4.6: Parameters dynamically specialized by deGoal in four kernels of VIPS and Streamcluster.

Bench.	Kernel	Parameter	Optimizations
VIPS	Linear transformation	Number of image layers.	Completely unroll the inner loop and partially unroll the outer loop.
	Interpolation	Number of image layers.	Completely unroll the main loop.
	Convolution	Convolution mask size.	Depending on the number of registers available, try to keep mask coefficients in registers.
Streamcluster	Euclidean distance	Dimension of points.	Uses as many registers as possible to unroll loops and conditionally generates a leftover loop, if needed.

Those parameters depend on the input set of the benchmarks, but they are constants during the whole execution. Table 4.6 describes the specialized parameters and the optimizations performed.

The implemented kernel generators are parametrizable in two aspects: they not only specialize code given a kernel parameter, but also maximize vector sizes and loop unrolling factors, by taking into account the number of available registers. For example, in the euclidean distance kernel (shown in Figure 4.2), I specialized the dimension of the points, which is 128 in the `simlarge` input set. At least three vectors are needed: two to hold the coordinates of the two points and one to hold the partial sum. The BeagleBoard-xM and Snowball have 16 quadword (total of 64 single-precision elements) and 32 single-precision registers. Then, in this case, when SISD code generation is activated, the length of the allocated vectors is $32/3 \approx 10$, while for the SIMD code generation, it is $64/3 \approx 20$[7]. These lengths are automatically calculated depending on the dimension of the points.

The speedups of the dynamic specialized versions compared to the reference-like deGoal versions and the reference kernels are shown in Table 4.7. Table 4.3 also presents the kernel execution times and Figure 4.6 summarizes the speedups of all versions studied in this chapter. On average, the dynamic code specialization provides speedups of 1.41 and 1.09 in the BeagleBoard-xM for SISD and SIMD codes, respectively. In the Snowball, the mean speedups are 1.23 (SISD) and 1.06 (SIMD). SIMD specialized codes with deGoal show on average modest speedups because most reference kernels from PARVEC are already specialized and work for a limited range of input set. The advantage of using deGoal in this case is that it can dynamically adapt the code to the input set, generating valid code to a broader range than the static specialization in PARVEC.

[7]Up to two quadword registers can be reserved as scratch or to hold FP input arguments.

Table 4.7: Speedup of deGoal dynamically specialized kernel versions compared to reference-like deGoal versions and reference kernels (from PARSEC and PARVEC).

Benchmark	Kernel	Speedup over reference-like				Speedup over reference[1]			
		BeagleBoard		Snowball		BeagleBoard		Snowball	
		SISD	SIMD	SISD	SIMD	SISD	SIMD	SISD	SIMD
VIPS	Linear transf.	1.14	1.88	1.55	1.64	1.82	1.07	1.55	1.06
	Interpolation	1.16	1.25	1.05	1.27	1.33	0.58	0.92	0.68
	Convolution	1.22	1.18	1.26	1.11	1.26	2.35	1.31	1.82
Streamcluster	Euclidean dist.	1.10	0.98	1.22	1.00	1.29	0.98	1.21	0.95
Geometric mean		1.15	1.28	1.26	1.23	1.41	1.09	1.23	1.06

[1] Most SIMD references (PARVEC suite) are already specialized. This explains why dynamically specialized code with deGoal show on average modest speedups compared to the SIMD references.

Table 4.8: deGoal run-time overhead (% of kernel run-time) to generate the dynamic specialized kernels.

Benchmark	Kernel	BeagleBoard-xM		Snowball	
		SISD	SIMD	SISD	SIMD
VIPS	Linear transformation	0.0061	0.0092	0.0103	0.0084
	Interpolation	0.0085	0.0147	0.0121	0.0171
	Convolution	0.0064	0.0053	0.0058	0.0048
Streamcluster	Euclidean distance	0.0004	0.0006	0.0005	0.0005

The run-time code generation overhead is negligible for all kernels, as Table 4.8 shows. deGoal spent less than 0.02 % of the kernel execution times to generate the specialized versions. The observed speedups come from the reduced number of instructions generated, better ILP and instruction scheduling, thanks to loop unrolling.

4.3.6 Run-time auto-tuning possibilities with deGoal

In this section, I discuss and illustrate the possibilities to use deGoal to auto-tune kernels at run-time. As shown in Table 4.8, deGoal has a very low run-time overhead. For instance, if we limit the code regeneration to 1 % of the kernel execution time, in the worst case, it would be possible to generate 58 different interpolation kernel versions (this kernel executes during only 1.8 second). For Streamcluster, the most favorable case, deGoal could generate 2500 versions. It is worth observing that these measured overheads are per kernel and per core, which highlights the scalability of this approach to multi-threaded applications in heterogeneous multi-core systems.

In the following, I discuss and give examples of code generation options and implementations that deGoal could explore.

4.3.6.1 SIMD vs SISD processing

ISA-compatible cores must ensure instruction compatibility, but how instructions are implemented and what performance they deliver may not be available to programmers and compiler developers. For instance, the performance of vectorized codes compared to non-vectorized ones depends on various factors, including the target architecture and machine-specific code optimizations. For example, in Table 4.5 the SIMD version of the linear transformation kernel has a speedup of 22 % in the BeagleBoard-xM, but a slowdown of 11 % in the Snowball. SIMD performance depends heavily on micro-architectural features, such as the bus and memory bandwidth of the SIMD unit, the throughput of instructions and register file accesses, among others. In the BeagleBoard-xM, the memory bus is 128-bit wide in the NEON and 64-bit in the ARM pipeline, while in the Snowball both are 64-bit wide. Furthermore, in the BeagleBoard-xM the FP SISD instructions can not bypass results in the RunFast mode. These factors combined explain why the BeagleBoard-xM shows greater SIMD speedups on average. The only exception is the convolution kernel, which had a slowdown of 4 % in the BeagleBoard-xM, but a speedup of 5 % in the Snowball. In this case, it is possible that some SIMD instructions are less efficiently implemented in the BeagleBoard-xM.

4.3.6.2 Loop unrolling strategy

A loop that performs computations can be unrolled in three ways: reusing registers by correctly replicating a code pattern in the loop body; using a different register to process each element by allocating vectors of registers; or mixing the two previous approaches.

There are trade-offs in those three loop unrolling strategies. Reusing registers may reduce the stack and register management overhead, but may also create pipeline stalls in in-order or resource constrained out-of-order pipelines. To reduce such pipeline stalls, one solution is to maximize the usage of the available registers, which in turn may increase the stack and register management overhead. Finally, allocating vectors of registers to process data may increase the ILP, but also expand the code before and after the loop to prepare the vectors.

4.3.6.3 Compressed vs non-compressed instructions

Instructions may generate more than one micro-operation in the decode stage. In an in-order core, the execution order of micro-operations must be preserved, while in an out-of-order core, they are dynamically scheduled. In in-order cores, this means that in some situations, it is better to generate non-compressed instructions rather than compressed ones, because the instruction scheduling performed by the compiler may increase the ILP.

For example, in the specialized convolution kernel, with the default code generation options in deGoal, the machine code has 32 instructions (96 bytes). With the option of instruction decompression, deGoal generates a machine code with 64 instructions (208 bytes). In the BeagleBoard-xM, this expanded version is 2.7 % faster, while in the Snowball, it has a slowdown of 7.5 %.

4.4 Scope and limitations

This section discusses the application scope and limitations of deGoal and the study presented in this chapter.

4.4.1 Scope

Run-time data and program specializer deGoal enables run-time information to drive the code generation. If the input data or hardware platform is only known at run time, deGoal is a tool adapted to perform run-time data or program specialization, with a very low overhead.

Optimization of computing libraries Given that dynamically linked applications have to jump into the library code[8], the no-inlining restriction in deGoal will not be a performance issue.

4.4.2 Limitations

Pre-identified small kernels Because deGoal defines a pseudo-assembly instruction set, as any low-level language, its usage is limited to small pieces of code. Although the code generation overhead is low, in order to pay off the programming effort, deGoal should be employed in pre-identified computing kernels.

Small kernels, but not too small Static compilers may inline small pieces of code. A good candidate kernel to be implemented with deGoal should be a small piece of code frequently called, but it should also perform a reasonable amount of computation during each call in order to avoid performance degradations when compared to a statically inlined kernel code. Modifying deGoal to inline code may produce a considerable code generation overhead, because most of the performance degradation do not come from the kernel call, but from the compiler optimizations allowed by function inlining. Therefore, performing similar optimizations directly in the machine code is not an easy task.

Complete performance evaluation In this chapter, I presented a preliminary performance evaluation of deGoal for ARM. Comparing only eight kernel configurations is not enough to evaluate the performance of the tool and draw generic conclusions. In addition, the kernels should ideally be written by programmers that do not know the internal technical details of deGoal. A more complete evaluation may be the subject of a future work.

[8] DBO systems may inline hot function codes from the dynamic library into the application.

4.5 Conclusion

This chapter presented deGoal, a lightweight run-time code generator for embedded systems. This thesis is interested in the energy and performance optimization of high-performance embedded processors. Therefore, I ported deGoal to the ARM Thumb-2 ISA and enhanced its infrastructure with a configurable EXE stage model for instruction scheduling. As FP and SIMD extensions are widespread in high-performance computing, I also implemented in deGoal the code generation support for the VFP and NEON extensions.

To validate the ARM porting, I evaluated the performance of four kernels found in modern benchmark suites (PARSEC 3.0 and PARVEC). Both the SISD and SIMD versions were implemented with deGoal in a way to fairly compare them to the reference codes. In this evaluation we observed on average equal or better performance in out-of-order and in-order ARM cores, respectively. Based on those same four kernels I implemented dynamic code specializers with deGoal, obtaining on average speedups of 1.41 and 1.23 compared to the SISD references in the Cortex-A8 and A9, respectively, and 1.09 and 1.06 compared to SIMD references, which were already completely or partially specialized. We observed less speedups in the A9, because differently from the A8, it is an out-of-order pipeline, which can unroll loops and accelerate underoptimized codes.

As in any language, specially in those without compiler optimizing passes, programmers are tempted to iteratively optimize their code implementation by feedbacking execution measurements in a given platform. However, as I argue and illustrate in Section 5, code optimizations may be machine- and input-dependent. We also observed performance asymmetries in the two studied ARM platforms and measured negligible run-time code generation overheads. These observations allowed to argue that deGoal can be used to implement a run-time auto-tuning framework, to provide better performance portability of computing kernels. In consequence, I also implemented in deGoal APIs that enable the target core configuration and the code generation options to be dynamically modified.

The development and enhancement of deGoal for ARM presented in this section allowed to implement a prototype of auto-tuning system for short-running kernels. The description of this system and its proof-of-concept are presented in the next section.

Chapter 5

Online auto-tuning for embedded systems

High-performance general-purpose embedded processors are evolving with unprecedented growth in complexity. ISA back-compatibility and energy reduction techniques are among the main reasons. In the past, embedded applications were compiled to only one target architecture, with simpler ISAs compared to current ones. Now, applications do not necessarily run in only one target, one binary code may run in processors from different manufacturers and even in different cores inside a SoC.

Iterative optimization and auto-tuning have been used to address the complexity of desktop- and server-class processors (DSCPs). They show moderate to high performance gains compared to non-iterative compilation, because default compiler options are usually based on the performance of generic benchmarks executed in representative hardware. These tuning approaches have been used to automatically find the best compiler optimizations and algorithm implementations for a given source code and target CPU. Usually, such tools need long exploration times to find quasi-optimal machine code. Previous work addressed auto-tuning at run-time [4, 45, 129, 135], however their techniques are only adapted to applications that run for several minutes or even hours, such as scientific or data center workloads, in order to pay off the space exploration overhead and overcome static compilation.

While previous work proposed run-time auto-tuning in DSCPs, no work focused on general-purpose embedded-class processors. In such devices, applications usually run for a short period of time, imposing a strong constraint to run-time auto-tuning systems. In this scenario, a lightweight tool should be employed to explore pre-identified optimizations in computing kernels.

There are several motivations that explain why we should develop run-time auto-tuning tools for embedded processors:

- **Embedded core complexity**: The complexity of high-performance embedded processors is following the same trend as the complexity of DSCPs evolved in the last decades. For example, current 64-bit embedded-class processors are sufficiently complex to be deployed in micro-servers, eventually providing a low-power alternative for data center computing. In order to address this growing complexity and provide better performance portability than static approaches, I propose online auto-tuning.
- **Heterogeneous multi/manycores**: The power wall is affecting embedded systems as it is affecting DSCPs, although in a smaller scale. Soon, dark silicon may also limit the powerable area in

embedded SoCs. As a consequence, heterogeneous clusters of cores coupled to accelerators are one of the solutions being adopted in embedded SoCs. In the long term, this trend will exacerbate software challenges of extracting the achievable computing performance from hardware, and run-time approaches may be the only way to improve energy efficiency [32].

- **ISA-compatible processor diversity**: In the embedded market, a basic core design can be implemented by different manufacturers with different transistor technologies and also varying configurations. Furthermore, customized pipelines may be designed, yet being entirely ISA-compatible with basic designs. This diversity of ISA-compatible embedded processors facilitates software development, however because of differences in pipeline implementations, static approaches can only provide sub-optimal performance when migrating between platforms. In addition, contrary to DSCPs, in-order cores are still a trend in high-performance embedded devices because of low-power constraints, and they benefit more from target-specific optimizations than out-of-order pipelines.

- **Static auto-tuning performance is target-specific**: On average, the performance portability of statically auto-tuned code is poor when migrating between different micro-architectures [3]. Hence, static auto-tuning is usually employed when the execution environment is known. On the other hand, the trends of hardware virtualization and software packages in general-purpose processors result in applications underutilizing the hardware resources, because they are compiled to generic micro-architectures. Online auto-tuning can provide better performance portability, as previous work showed in server-class processors [4].

- **Ahead-of-time auto-tuning**: In recent Android versions (5.0 and later), when an application is installed, native machine code is generated from bitcode (ahead-of-time compilation). The run-time auto-tuner proposed in this chapter could be extended and integrated in such systems to auto-tune code to the target core(s) or pre-profile and select the best candidates to be evaluated in the run-time phase. Such approach would allow auto-tuning to be performed in embedded applications with acceptable ahead-of-time compilation overhead. In the experiments presented later in this chapter, on average deGoal spent only 370 µs to generate one kernel in a Cortex-A9 (800 MHz).

- **Interaction with other dynamic techniques**: Some powerful compiler optimizations depend both on input data and the target micro-architecture. Constant propagation and loop unrolling are two examples. The first can be addressed by dynamically specializing the code, while the second is better addressed by an auto-tuning tool. When input data and the target micro-architecture are known only at program execution, mixing those two dynamic techniques can provide even higher performance improvements. If static versioning is employed, it could easily lead to code size explosion, which is not convenient in embedded systems.

To built a prototype of an online auto-tuning tool, deGoal, the run-time code generator presented in Chapter 4, was taken as the starting point. deGoal is adapted to this task, because its dynamic language allows to parametrize the code generation and hence to explore the implementation space of a computing kernel.

In order to evaluate the proposed run-time auto-tuning approach, not only two ARM platforms were used in the experiments, but also the simulation framework presented in Chapter 3 was employed to explore the design space of embedded ARM cores. Small single-issue in-order cores up to big triple-issue out-of-order cores were simulated, totalizing in 11 different micro-architectures. The objective was not only to compare statically compiled code to online auto-tuned ones, but also to estimate at what

extent run-time code optimizations in in-order pipelines can achieve the same performance of statically compiled binaries in out-of-order pipelines, and then improving the energy efficiency.

This chapter begins by presenting in Section 5.1 a motivational example illustrating the potential performance gains if auto-tuning is pushed to run time. Next, Section 5.2 details the proposed methodology to create online auto-tuning kernels. The proof-of-concept of the online auto-tuning system is presented through the implementation and analysis of two case studies in real and simulated CPUs, in Sections 5.3 and 5.4. Finally, the scope of application and limits of the proposed tool and study are discussed in Section 5.5 and Section 5.6 concludes this chapter.

5.1 Motivational example

This section presents a motivational example illustrating potential performance improvements if run-time code specialization and auto-tuning are mixed.

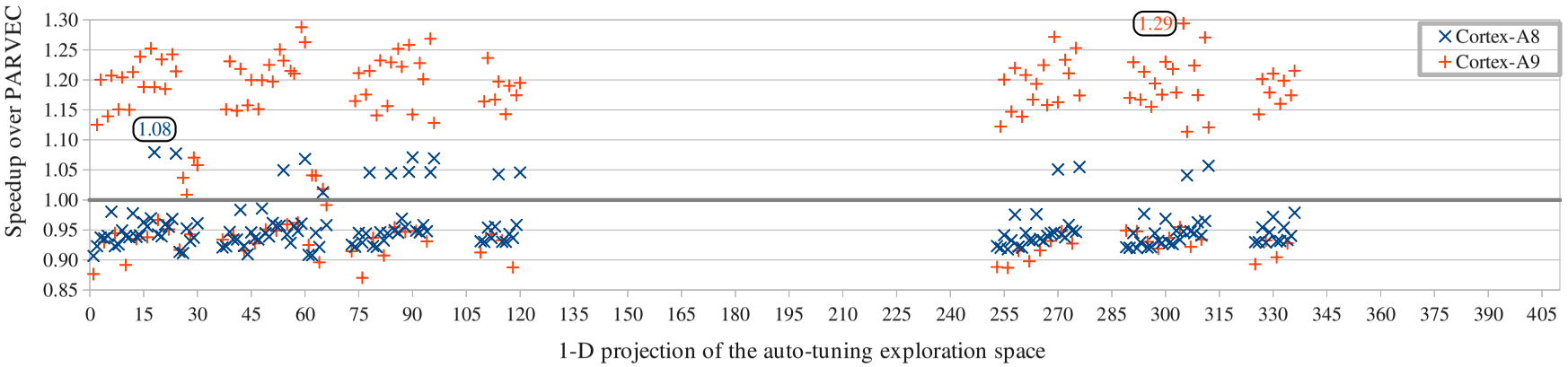
I studied the SIMD version of the euclidean distance kernel implemented in the Streamcluster benchmark, manually vectorized in the PARVEC [39] suite (originally from the PARSEC 3.0 suite [28]).

In the reference kernel, the dimension of points is a run-time constant, but given that it is part of the input set, compilers cannot optimize it. In the following comparisons, I purposefully set the dimension as a compile-time constant in the reference code to let the compiler (gcc 4.9.3) generate highly optimized kernels.

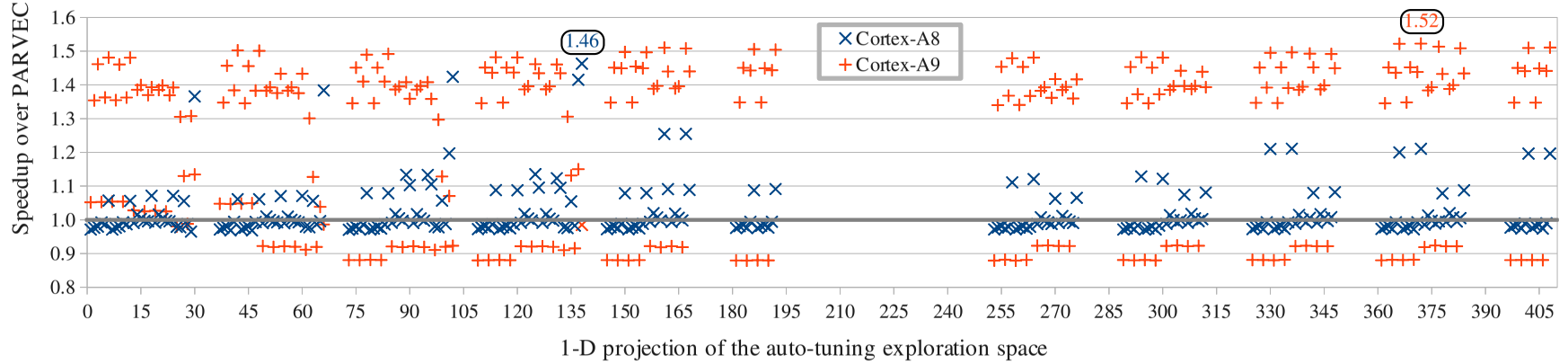
To evaluate the auto-tuning capabilities of deGoal, I statically generated various kernel versions, by specializing the dimension and auto-tuning the code implementation for a Cortex-A8 and A9. The auto-tuned parameters mainly affect loop unrolling and pre-fetching instructions, and are detailed in Section 5.2.1.

Figure 5.1 shows the speedups of various kernels generated by deGoal in two core configurations. By analyzing the results, we can conclude that:

1. **Code specialization and auto-tuning provide considerable speedups even compared to statically specialized and manually vectorized code**: In the studied cases deGoal found optimized kernel implementations with speedups going up to 1.46 and 1.52 in the Cortex-A8 and A9.
2. **The best set of auto-tuned parameters and optimizations varies from one core to another**: In both cases in Figure 5.1, there is a poor performance portability of the best configurations between the two cores. For example, in Figure 5.1(b), when the best kernel for the Cortex-A8 is executed in the A9 it shows a slowdown of 35 %, compared to the best kernel for the latter. Conversely, the best kernel for the A9 when executed in the A8 shows a slowdown of 17 %, compared to the most performing kernel.
3. **There is no performance correlation between the sets of optimizations and input data**: The main auto-tuned parameters are related to loop unrolling, which depends on the dimension of points (part of the input set). In consequence, the exploration space and the optimal solution depend on input data. For example, the configurations of the top five peak performances for the A8 in Figure 5.1(b) (configurations 30, 66, 102, 137 and 138) have poor performances in Figure 5.1(a) or simply can not generate code in a lower dimension (unrolling factors depends on the dimension).



(a) *Dimension* = 32



(b) *Dimension* = 128

Figure 5.1: Speedups of euclidean distance kernels statically tuned with deGoal for two ARM platform and two input sets. The reference is a hand vectorized kernel (from PARVEC) compiled with gcc 4.9.3 and highly optimized to each target core (`-O3` and `-mcpu` options). Both deGoal and reference kernels have the dimension of points specialized (i.e. set as a compile-time constant). Each number in the x-axis represents a possible combination of tuning parameters. The tuning space goes beyond 600 configurations, but here it was zoomed in on the main region. The peak performance of each core is labeled. Empty results in the tuning space correspond to configurations that could not generate code.

The results suggest that, although code specialization and auto-tuning provide high performance improvements, they should ideally be performed only when input data and target core are known. In the released input sets for Streamcluster, the dimensions are 32, 64 and 128, but the benchmark accepts any integer value. Therefore, even if the target core(s) was (were) known at compile time and the code was statically specialized, auto-tuned and versioned, it could easily lead to code size explosion.

The most important feature of deGoal is that it is fast enough to enable the specialization of run-time constants combined with online auto-tuning, allowing the generation of highly optimized code for a target core, which may not be known prior compilation.

The optimized kernels shown in this motivational example were statically auto-tuned. The run-time auto-tuning approach proposed in this thesis successfully found optimized kernels whose performance is on average only 6 % away from the performance of the best kernels statically found (all run-time overheads included). It is worth observing that the static space exploration in the experiments took several hours per dimension and per platform, even if the benchmark runs for a few seconds.

5.2 Methodology

This section describes the methodology of the proposed approach. Figure 5.2 presents the architecture of the framework that auto-tunes a function at run-time. At the beginning of the program execution, a reference function (e.g., C compiled code) is evaluated according to a defined metric (execution time in the experiments presented here). This reference function starts as the active function. In parallel to the program execution, the auto-tuning thread periodically wakes up and decides if it is time to regenerate and evaluate a new version. The auto-tuning thread will replace the active function by the new one, if its score is better. This approach is applicable to computing kernels highly called.

In the experiments presented latter in this chapter, only the execution time was considered as the score to decide if a new function performs better than the active one. Our auto-tuning architecture, presented in Figure 5.2, could easily take other metrics into account, as the energy consumption, given that stable measurement are available in the hardware.

In the following sections, I describe the implementation of each block from the main loop of Figure 5.2.

5.2.1 Auto-tuning with deGoal

To illustrate some auto-tuning possibilities, Figure 5.3 presents the deGoal code to auto-tune the euclidean distance implementation, focusing on the main loop of the kernel. This is the code used in the motivational example presented in Section 5.1, and also in the run-time auto-tuning experiments later in the chapter. The compilette in Figure 5.3 can generate different machine codes, depending on the arguments that it receives. In line 1, the first argument is the dimension, which is specialized in this example. The four following arguments are the auto-tuned parameters:

- **Hot loop unrolling factor (`hotUF`)**: Unrolls a loop and processes each element with a different register, in order to avoid pipeline stalls.

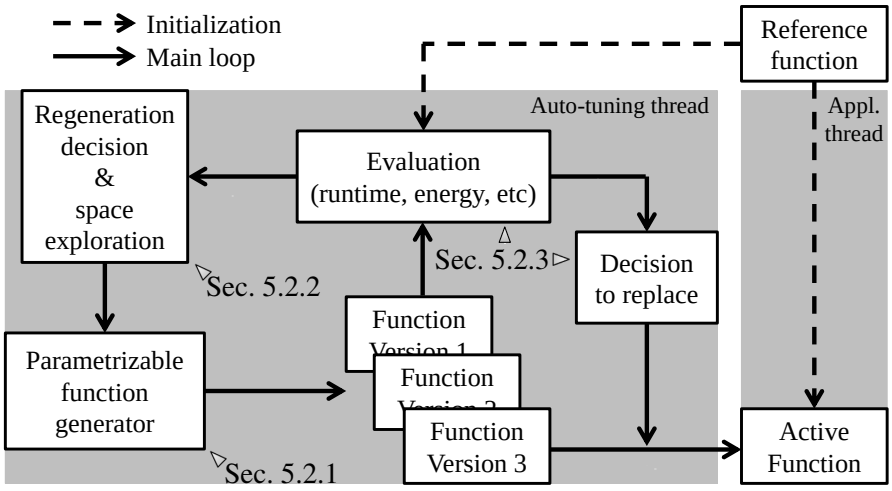


Figure 5.2: Architecture of the run-time auto-tuning framework.

- **Cold loop unrolling factor (`coldUF`)**: Unrolls a loop by simply copy-pasting a pattern of code, using fewer registers, but potentially creating pipeline stalls.
- **Normalized vector length (`vectLen`)**: Defines the length of the vector used to process elements in the loop body, normalized to the SIMD width when generating SIMD instructions (four in the ARM ISA). Longer vectors may benefit code size, because instructions that load multiple registers instructions may be generated.
- **Data pre-fetching stride (`pldStride`)**: Defines the stride in bytes used in hint instructions to try to pre-fetch data of the next loop iteration.

Given that the dimension is specialized (run-time constant), we know exactly how many elements are going to be processed in the main loop. Hence, between the lines 5 and 21, the pair of deGoal instructions `loop` and `loopend` can produce three possible results, depending on the dimension and the unrolling factors:

1. No code for the main loop is generated if the dimension is too small. The computation is then performed by a leftover code (not shown in Figure 5.3).
2. Only the loop body is generated without any branch instruction, if the main loop is completely unrolled.
3. The loop body and a backward branch are generated if more than one iteration is needed (i.e. the loop is partially unrolled).

Between the lines 6 and 20, the loop body is unrolled by mixing three auto-tuning effects, whose parameters are highlighted in Figure 5.3: the outer `for` (line 6) simply replicates `coldUF` times the code pattern in its body, the inner `for` (line 7) unrolls the loop `hotUF` times by using different registers to process each pair of coordinates, and finally the number of elements processed in the inner loop is set through the vector length `vectLen`.

```
1  dist_gen(int dim, int vectLen, int hotUF, int coldUF,
       int pldStride)
2  {
3      numIter = function(dim, vectLen, hotUF, coldUF);
4      (...)
5  #[  loop #(numIter)                                    ]#
6      for (j = 0; j < coldUF; ++j) {
7        for (i = 0; i < hotUF; ++i) {
8  #[      lw Vc1[#(i)], coord1                          ]#
9  #[      lw Vc2[#(i)], coord2                          ]#
10         if (pldStride != 0) {
11 #[       pld coord1, #((vectLen-1)*4 + pldStride)     ]#
12 #[       pld coord2, #((vectLen-1)*4 + pldStride)     ]#
13         }
14 #[      sub Vc1[#(i)], Vc1[#(i)], Vc2[#(i)]           ]#
15 #[      mac Vresult, Vc1[#(i)], Vc1[#(i)]             ]#
16
17 #[      add coord1, coord1, #(vectLen*4)              ]#
18 #[      add coord2, coord2, #(vectLen*4)              ]#
19       }
20     }
21 #[  loopend                                           ]#
22     (...)
23 #[  add result, Vresult                               ]#
24     (...)
25 }
```

Figure 5.3: Main loop of the deGoal code to auto-tune the euclidean distance kernel in the Streamcluster benchmark. The first function parameter is the specialized dimension, and the other four are the auto-tuned parameters (highlighted variables).

In the lines 10 to 13, the last auto-tuned parameter affects a data pre-fetching instruction: if `pldStride` is zero, no pre-fetching instruction is generated, otherwise deGoal generates a hint instruction that tries to pre-fetch the cache line pointed by the address of the last load plus `pldStride`.

In this example, the computation can be divided into two phases, whose parameter choices have negative performance impact on each other: a highly parallel part thanks to loop unrolling and longer vectors; and an almost serial phase, because of the reduction of the partial sums from vector elements (line 23 in Figure 5.3). In other words, longer vectors benefit parallelism in the first phase, but increase the execution time of vector elements reduction in the second one. Given that the right balance between them depends on the target pipeline features (e.g., pipeline width, latency of instructions), auto-tuning can play a important role to find the best implementation.

Besides the auto-tuning possibilities, which are explicitly coded with the deGoal language, a set of C functions can be called to configure code generation options. In this work, three code optimizations were studied:

- **Instructions scheduling (IS)**: Reorders instructions to avoid stall cycles and tries to maximize multi-issues.
- **Stack minimization (SM)**: Only uses FP scratch registers to reduce the stack management overhead.
- **Vectorization (VE)**: Generates SIMD instructions to process vectors.

Most of the explanations presented in this section were given through examples related to the Streamcluster benchmark, but partial evaluation, loop unrolling and data pre-fetching are broadly used compiler optimization techniques that can be employed in almost any application.

5.2.2 Regeneration decision and space exploration

The regeneration decision takes into account two factors: the regeneration overhead and the average speedup achieved at the moment. The first one allows to keep the run-time overhead of the tool at acceptable limits if it fails to find better kernel versions. The second factor acts as an investment, i.e. allocating more time to explore the tuning space if previously found solutions provided enough speedups. Both factors are represented as percentage values, for example limiting the regeneration overhead to 1 % and investing 10 % of gained time to explore new versions.

To estimate the gains, the instrumentation needed in the evaluated functions is simply a variable that increments each time the function is executed. Knowing this information and the measured run-time of each kernel, it is possible to estimate the time gained at any moment. However, given that the reference and the new versions of kernel have their execution times measured only once, the estimated gains may not be accurate if the application has phases with very different behaviors.

The kernel run-times and the auto-tuning overhead are estimated by using performance counters. The procedure employed to filter measurement oscillations is described in the Section 5.2.3.

Most of the following configurations should be empirically defined through a pre-profiling in a representative platform. There is no assurance that the pre-profiled configurations will always work in all processor configurations. This limitation is further discussed in Section 5.5.

Given that the whole tuning space can have hundreds or even thousands of kernel versions, I divided it in two phases:

- **First phase**: Explores auto-tuning parameters that have an impact on the structure of the code, namely, `hotUF`, `coldUF` and `vectLen`, but also the vectorization option (VE). The previous list is also the order of exploration, going from the least switched to the most switched parameter. In the first phase, the default configuration is with IS active and `pldStride` at its maximum value.

- **Second phase**: Fixes the parameters found in the previous phase and explores the combinatorial choices of remaining code generation options (IS, SM) and `pldStride`.

For now, the range of `hotUF` and `vectLen` were defined by the programmer, but these tasks can be automated and dynamically computed by taking into account the code structure (static) and the available registers (dynamic information). Compared to `coldUF`, their ranges are well bounded for online auto-tuning, providing an acceptable search space size.

The range of `coldUF` depends on the number of processed elements in the loop, which in turn depend on the input data (e.g., the dimension in the euclidean distance kernel). However, unrolling a loop beyond a given threshold provides only negligible incremental speedup. Therefore, I empirically limited the range of `coldUF` to 64.

The last auto-tuned parameter, `pldStride`, was explored with the values 32 and 64, which are typical cache line lengths in ARM processors.

Finally, to optimize the space exploration in the first phase, in a first stage the tool searches for optimal kernel implementations, i.e., which have no leftover code. After exhausting all possibilities of this stage, this condition may be softened in the next stages by gradually allowing leftover processing. The first phase ends if at least 32 kernel versions were evaluated since the beginning and after exhausting all solutions of the current stage.

5.2.3 Kernel evaluation and replacement

In order to evaluate a new kernel version, the input data (i.e., processed data) used in the first and second phases can be either:

- **Real input data only**: Evaluates new kernel versions with real data, performing useful work during evaluation, but suffering from measurement oscillations between independent runs (empirical observation). These oscillations can sometimes lead to wrong kernel replacement decisions.

- **Training & real input data**: Uses training data with warmed caches in the first phase and real data in the second one. A training input set with warmed caches results in very stable measurements, which ensure good choices for the active function. Since no useful work is performed, using training data is only adapted to kernels that are called sufficient times to consider the overhead of this technique negligible, and to kernels that have no side effect. In the second phase, the usage of real data is mandatory, because the adequacy of pre-fetching instruction depends on the interaction of the real data and code with the target pipeline.

When the evaluation uses real data, the performance of the kernel is simply obtained by averaging the run-times of a pre-determined number of runs.

When the kernel uses a training input data, the measurement differs. To better illustrate the approach, in the following I present the technique used in the Streamcluster benchmark, but different parameters could be employed in other benchmarks. The performance of the kernel is obtained by running it 15 times in loop (after 10 runs to warm caches) with the training input set. First, the measurements are divided in three groups and the best run-time of each group is identified. Finally, the worst execution time among the three is taken. This technique filters unwanted oscillations caused by hardware (fluctuations in the pipeline, caches and performance counters) and software (interruptions). In the studied platforms, stable measurements were observed, with virtually no oscillation between independent runs (in a Cortex-A9, I measured oscillations of less than 1 %).

The decision to replace the active function by a new version is taken by simply comparing the measured run-times. The active function is defined by a global function pointer, and its modification is done in the auto-tuning thread, protected in a critical section.

5.3 Experimental setup

This section presents the experimental setup. First, I detail the hardware and simulation platforms used to evaluate the capabilities of the proposed tool to dynamically auto-tune computing kernels to different micro-architectures. Then, the chosen applications for two case studies are described.

5.3.1 Hardware platforms

The evaluation boards used in the experiments are the BeagleBoard-xM and Snowball, already described in Section 3.4.1. Given that this work studies auto-tuning in single-cores and code regeneration is performed in a separated thread, I forced the execution of the benchmarks in one core through the Linux command `taskset`, in order to include the code regeneration overhead into the measured execution times.

The fact that auto-tuning in performed in single-cores is not a limitation of the approach. The proof of concept is developed around single-threaded benchmarks in single-cores, but if a multi-threaded benchmark was run in a heterogeneous multi-core, the online approach developed here can be adapted to locally auto-tune the code of a kernel in each heterogeneous core. For example, a kernel function pointer may be a global variable (which is the case in the experiments of this chapter), but each thread can access its "own" global variable through thread-local storage (TLS) methods.

5.3.2 Simulation platform

I configured the simulation framework presented in Chapter 3 to simulate 11 different core configurations. Table 5.1 shows the main configurations of the simulated cores. The 11 configurations were obtained by varying the pipeline type (in-order and out-of-order cores) and the number of FP/SIMD units of one-, two- and three-way basic pipelines. The single-issue core is in-order and has only one FP/SIMD unit. Dual-issue cores can have one or two FP/SIMD units, while triple-issue cores can have one, two or three units. The single- and triple-issue basic cores were based on the gem5 Cortex-A7 and A15 models (Section 3.5.2), respectively, while the dual-issue core is a hypothetical intermediate design close to the gem5 Cortex-A9 model (Section 3.4.2). For power estimation, the temperature was fixed at 47 °C and other McPAT configurations are the same as described in Section 3.5.2.

Table 5.2 shows the abbreviations used to identify each core design and their CPU area.

5.3.3 Benchmarks

I chose two kernel-based applications as case studies to evaluate the proposed online auto-tuning approach. To be representative, one benchmark is CPU-bound and the other is memory-bound. Indeed, the auto-tuned parameters explored by the proposed approach are not suitable for a memory-bound kernel. However, this kind of kernel was also evaluated to show that this approach has negligible overhead in unfavorable situations. In both applications, the evaluated kernels correspond to more than 80 % of execution time.

Table 5.1: Main parameters of the simulated cores.

Parameter		Single-issue	Dual-issue	Triple-issue
Pipeline type		In-order only	In-order or out-of-order	In-order or out-of-order
Core clock		1.4 GHz	1.6 GHz	2.0 GHz
DRAM	Size / clock / latency (ns)	256 MB / 933 MHz / 81	256 MB / 933 MHz / 81	256 MB / 933 MHz / 81
L2	Size / assoc. / lat. / MSHRs / WBs	512 kB / 8 / 3 / 8 / 16	1024 kB / 8 / 5 / 8 / 16	2048 kB / 16 / 8 / 11 / 16
L1-I	Size / assoc. / lat. / MSHRs	32 kB / 2 / 1 / 2	32 kB / 2 / 1 / 2	32 kB / 2 / 1 / 2
L1-D	Size / assoc. / lat. / MSHRs / WBs	32 kB / 4 / 1 / 4 / 4	32 kB / 4 / 1 / 5 / 8	32 kB / 2 / 1 / 6 / 16
Stride prefet.	Cache level / degree / buffer size	1 / 1 / 8	1 / 1 / 12	2 / 1 / 16
Branch pred.	Global / local history entries (bits)	256 (2) / N/A	4096 (2) / N/A	4096 (2) / 1024 (3)
	BTB / RAS entries	256 / 8	4096 / 16	4096 / 48
ITLB / DTLB entries		32 each	64 each[1]	128 each[1]
Front-end / back-end width		1 / 1	2 / 4	3 / 7
INT / FP pipeline depth (+ extra out-of-order stages)		8 / 10	8 / 12 (+3)	9 / 18 (+6)
Physical INT[2]/ FP[2] registers		N/A	82 / 256	90 / 256
IQ / LSQ / ROB[2] entries		16 / 8 each / N/A	32 / 12 each / 40	48 / 16 each / 60
INT units	ALU / MUL execution ports	1 / 1	2 / 1	2 / 1
	ADD / MUL cycles	1 / 4	1 / 4	1 / 4
FP/SIMD	Execution ports	1	1 or 2	1, 2 or 3
	VADD / VMUL / VMLA cycles	3 / 4 / 6	4 / 5 / 8	10 / 12 / 20
Load/store	Execution ports	1 shared	1 shared	1 for each
	Load / store cycles	1 / 1	2 / 1	3 / 2

[1] Over-dimensioned to compensate the lack of L2-TLB.

[2] For out-of-order only.

Table 5.2: Abbreviation of the 11 simulated core designs and CPU areas (mm^2).

Abbreviation	Description	Core	L2	Total
SI-I1	Single-issue in-order with one FP/SIMD unit.	0.45	1.52	1.97
DI-I1	Dual-issue in-order with one FP/SIMD unit.	1.00	3.19	4.19
DI-I2	Dual-issue in-order with two FP/SIMD unit.	1.48	3.19	4.67
DI-O1	Dual-issue out-of-order with one FP/SIMD unit.	1.15	3.19	4.34
DI-O2	Dual-issue out-of-order with two FP/SIMD unit.	1.67	3.19	4.86
TI-I1	Triple-issue in-order with one FP/SIMD unit.	1.81	5.88	7.70
TI-I2	Triple-issue in-order with two FP/SIMD unit.	2.89	5.88	8.78
TI-I3	Triple-issue in-order with three FP/SIMD unit.	3.98	5.88	9.86
TI-O1	Triple-issue out-of-order with one FP/SIMD unit.	2.08	5.88	7.97
TI-O2	Triple-issue out-of-order with two FP/SIMD unit.	3.21	5.88	9.10
TI-O3	Triple-issue out-of-order with three FP/SIMD unit.	4.35	5.88	10.2

The first kernel is the euclidean distance computation in the Streamcluster benchmark from the PARSEC 3.0 suite. It solves the online clustering problem. Given points in a space, it tries to assign them to nearest centers. The clustering quality is measured by the sum of squared distances. With high space dimensions, this benchmarks is CPU-bound [28]. In the compilette, the dimension is specialized (run-time constant). I used the `simsmall` input set, evaluating the dimensions 32 (original), 64 and 128 (as in the native input set), which are referred as small, medium and large input sets, respectively.

The second kernel was taken from VIPS, an image processing application. We apply a linear transformation to an image by executing it with the following command line:

```
vips im_lintra_vec MUL_VEC input.v ADD_VEC output.v
```

Here, `input.v` and `output.v` are images in the VIPS XYZ format, and `MUL_VEC`, `ADD_VEC` are respectively FP vectors of the multiplication and addition factors for each band applied to each pixels in the input image. Given that pixels are loaded and processed only once, it is highly memory-bound. In the compilette, two run-time constants, the number of bands and the width of the image, are specialized. I tested three released input sets: `simsmall` (1600 x 1200), `simmedium` (2336 x 2336) and `simlarge` (2662 x 5500).

The benchmarks were compiled with gcc 4.9.3 2014.11 (gcc 4.5.2 for Streamcluster binaries used in the simulations, because of simulation stalls[1]) and the default PARSEC flags (`-O3 -fprefetch-loop-arrays` among others). I set the NEON flag (`-mfpu=neon`) to allow all 32 FP registers to be used. I also specified the target core (`-mcpu` option) for the real platforms and the ARMv7-A architecture (`-march=armv7-a`) for binaries used in the simulations. The deGoal library was also compiled for the ARMv7-A architecture, because the compilettes are supposed to generate code in any ARMv7-A compatible host.

[1] gcc 4.9.3 2014.11 is a pre-release, while the file system for gem5 has a gcc 4.5.2. I could not identify the reasons of the simulation stalls.

5.3.4 Evaluation methodology

The proposed online approach can auto-tune both SISD and SIMD codes during the application execution. In order to allow a fair comparison between the proposed approach and the references, the auto-tuning internally generates and evaluates both SIMD and SISD code, but when comparing to the SISD reference, SIMD generated codes are ignored and only SISD kernels can be active in the application, and vice-versa. In a real scenario, the performance achieved by the proposed approach is the best among the SISD and SIMD results presented in this chapter. I also set the SISD reference as initial active function, because this is a realistic scenario.

In the real platforms, I also statically explored the tuning space to find the best kernel implementation per platform and per input set. In order to limit prohibitive exploration times, I only searched for optimal solutions (no leftovers) for Streamcluster and explored at least 1000 points in the search space for VIPS, because some input sets had only few optimal solutions. Between 3 and 20 measurements were collected depending on observed oscillations. Part of this exploration is shown in Figure 5.1. Similar measurements were carried out in the online explorations.

5.4 Experimental results

This section presents the experimental results of the proposed online auto-tuning approach in a CPU- and a memory-bound kernels. First, the results obtained in real and simulated platforms are presented.

5.4.1 Real platforms

Table 5.3 presents the execution times of all configurations studied of the two benchmarks in the real platforms.

Figures 5.4(a) and 5.4(b) show the speedups obtained in the Streamcluster benchmark. On average, run-time auto-tuning provides speedup factors of 1.12 in the Cortex-A8 and 1.41 in the A9. The speedup sources come mostly from micro-architectural adaption, because even if the reference kernels are statically specialized, they can not provide significant speedups. Online auto-tuning performance is only 4.6 % and 5.8 % away from that of the best statically specialized and auto-tuned versions, respectively for the A8 and A9.

We can also note that in the A9, the SIMD reference versions are on average 11 % slower than the SISD references, because differently from the SISD reference pre-fetching data instructions are not generated by gcc in the SIMD code. My online approach can effectively take advantage of SIMD instructions in the A9, providing on average a speedup of 1.41 compared to the SISD reference code and 1.13 compared to dynamically auto-tuned SISD code.

Figures 5.4(c) and 5.4(d) show the speedups obtained in the VIPS application. Even with the hardware bottleneck being the memory hierarchy, on average the proposed approach can still speed up the execution by factors of 1.10 and 1.04 in the A8 and A9, respectively. Most of the speedups come from SISD versions (SIMD performances almost matched the references), mainly because in the reference code run-time constants are reloaded in each loop iteration, differently from the compilette implementa-

Table 5.3: Execution time (seconds) of the benchmarks with the original and specialized reference kernels, and with the online auto-tuned and the best statically auto-tuned kernels, in the real platforms (all run-time overheads included).

Benchmark	Input	Version	Cortex-A8				Cortex-A9			
			Ref.	Spec. Ref.	O-AT	BS-AT	Ref.	Spec. Ref.	O-AT	BS-AT
Streamcluster	Small	SISD	9.75	10.2	9.26	9.06	3.26	4.00	2.66	2.47
		SIMD	3.84	3.79	3.74	3.51	3.33	2.90	2.51	2.24
	Medium	SISD	19.9	21.8	17.9	17.8	7.54	11.1	5.87	5.68
		SIMD	7.13	7.05	6.59	5.93	9.09	8.86	5.09	4.84
	Large	SISD	46.8	46.1	41.0	40.8	14.8	14.7	12.0	11.3
		SIMD	15.1	15.0	11.1	10.2	17.2	15.1	10.1	9.84
VIPS lintra	Small	SISD	0.841	0.842	0.676	0.640	0.502	0.504	0.456	0.443
		SIMD	0.556	0.563	0.584	0.510	0.455	0.454	0.471	0.442
	Medium	SISD	2.30	2.29	1.76	1.73	1.47	1.41	1.37	1.24
		SIMD	1.47	1.48	1.40	1.36	1.31	1.41	1.31	1.26
	Large	SISD	26.6	24.3	25.1	22.9	10.1	9.63	9.88	9.49
		SIMD	24.7	24.9	24.0	22.2	10.4	10.0	9.94	9.54

Spec. ref.: Reference kernel specialized as in the auto-tuned versions.
O-AT: Online auto-tuned kernel.
BS-AT: Best statically auto-tuned kernel.

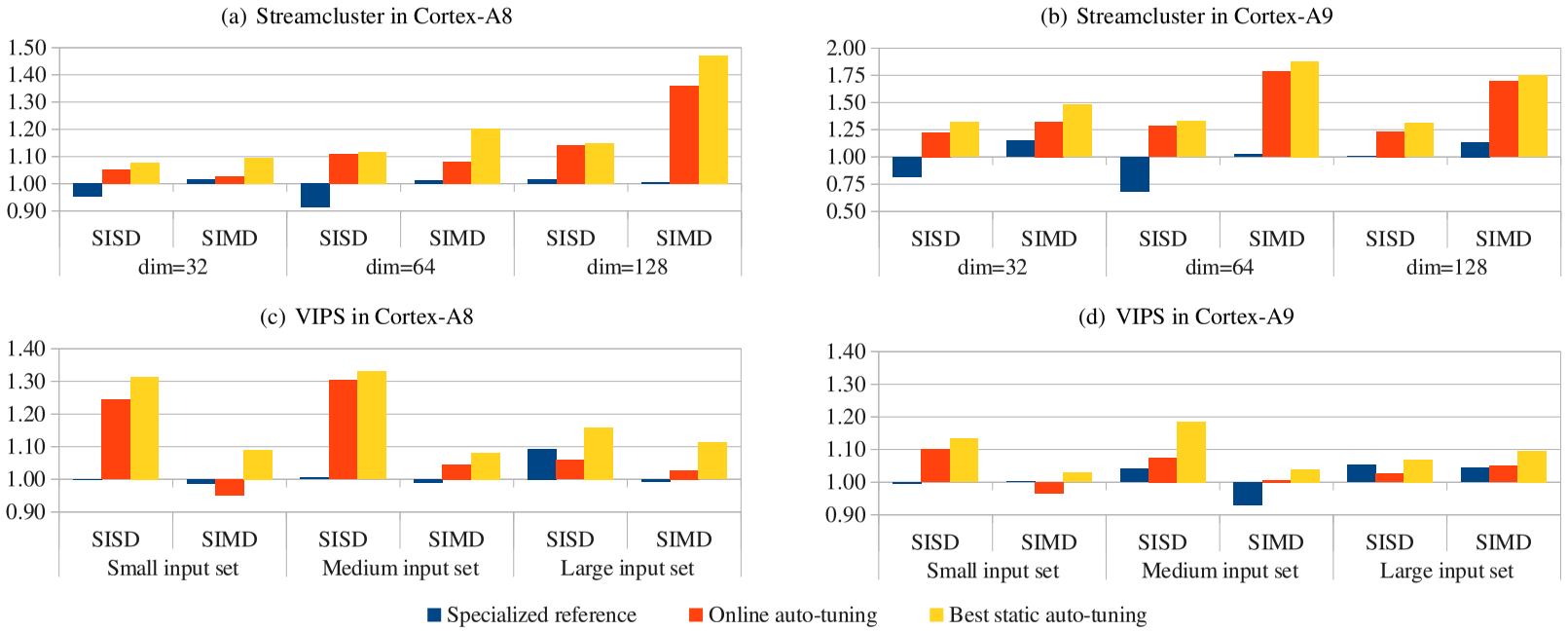


Figure 5.4: Speedup of the specialized reference and the auto-tuned applications in the real platforms (normalized to the reference benchmarks).

Table 5.4: Statistics of online auto-tuning in the Cortex-A8 and A9 (SISD / SIMD separated, or average if minor variations).

Bench.	Input set	Explo-rable versions	Exploration limit in one run	Run-time regeneration and space exploration						
				Kernel calls	Explored		Overhead to bench. run-time		Duration to kernel life	
					A8	A9	A8	A9	A8	A9
Stream-cluster	dim=32	390	43-49	5315388	49	49	0.2 % (11 ms)	0.4 % (9.2 ms)	13 / 4.4 %	32 %
	dim=64	510	55-61		58	61	0.2 % (17 ms)	0.3 % (15 ms)	6.3 / 2.7 %	22 %
	dim=128	630	67-73		67	73	0.2 % (30 ms)	0.2 % (26 ms)	5.6 / 1.8 %	15 %
VIPS	Small	858	106-112	1200	44	28	4.2 % (26 ms)	2.5 % (12 ms)	100 %	100 %
	Medium	330	39-45	2336	40	42	0.9 % (14 ms)	1.0 % (14 ms)	18 %	66 %
	Large	596	73-79	5500	75	71	0.3 % (71 ms)	0.8 % (78 ms)	28 %	86 %

tion. On average, online auto-tuning performances are only 6 % away from the best static ones.

Table 5.4 presents the auto-tuning statistics in both platforms. For each benchmark and input set, it shows that between 330 and 858 different kernel configurations could be generated, but in one run this space is limited between 39 and 112 versions, thanks to the proposed two phase exploration (Section 5.2.2). The online statistics gathered in the experiments are also presented. In most cases, the exploration ends very quickly, specially in Streamcluster, in part because of the investment factor. Only with the small input in VIPS, the auto-tuning did not end during its execution, because it has a large auto-tuning space and VIPS executes during less than 700 ms. In the experiments, the overhead of the run-time approach is negligible, between only 9 and 80 ms were spent to generate and evaluate from 28 to 75 kernel versions.

5.4.2 Simulated cores

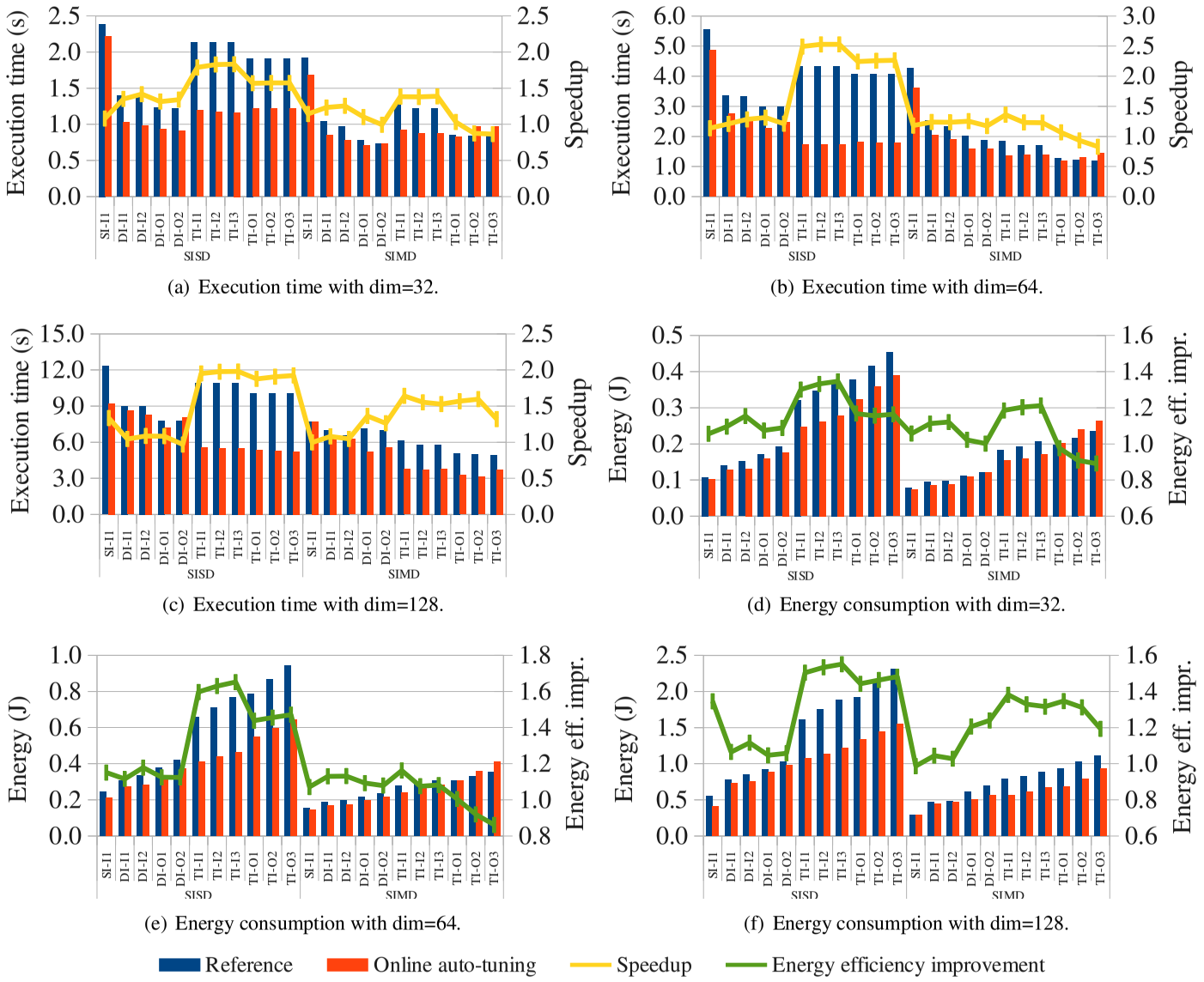
Figure 5.5 shows the simulated energy and performance of the reference and online auto-tuning versions of the Streamcluster benchmark. In the SISD comparisons, run-time auto-tuning can find kernel implementations with more ILP than the reference code, specially remarkable in the long triple-issue pipelines. The average speedup is 1.58. In the SIMD comparisons, the reference kernel naturally benefits from the parallelism of vectorized code, nonetheless online auto-tuning can provide an average speedup of 1.20. Only 6 of 66 simulations showed worse performance, mostly in big cores that quickly executed the benchmark.

In terms of energy, in general, there is no surprise that pipelines with more resources consume more energy, even if they may be faster. However, there are interesting comparisons between *equivalent* in-order and out-of-order cores. Here, the term *equivalent* means that cores have similar configurations, except the dynamic scheduling capability. Figure 5.6(d) shows the area overhead of out-of-order cores compared to equivalent in-order designs.

Still analyzing Streamcluster, when the reference kernels execute in equivalent in-order cores, on average their performance is worsened by 16 %, yet being 21 % more energy efficient, as Figure 5.6(a) shows. On the other hand, online auto-tuning improves those numbers to 6 % and 31 %, respectively (Figure 5.6(b)). In other words, the online approach can considerably reduce the performance gap between in- and out-of-order pipelines to only 6 %, and further improve energy efficiency.

It is also interesting to compare reference kernels executed in out-of-order cores to online auto-tuning versions executed in equivalent in-order ones. Despite the clear hardware disadvantage, on average the run-time approach can still provide speedups of 1.52 and 1.03 for SISD and SIMD, and improve the energy efficiency by 62 % and 39 %, respectively, as Figure 5.6(c) illustrates.

In the simulations of VIPS, the memory-boundedness is even more accentuated, because the benchmark is only called once and then Linux does not have the chance to use disk blocks cached in RAM. The performance of the proposed approach virtually matched those of the reference kernels. The speedups oscillate between 0.98 and 1.03, and the geometric mean is 1.00. Considering that between 29 and 79 new kernels were generated and evaluated during the benchmark executions, this demonstrates that the proposed technique has negligible overheads if auto-tuning can not find better versions.



(a) Execution time with dim=32.

(b) Execution time with dim=64.

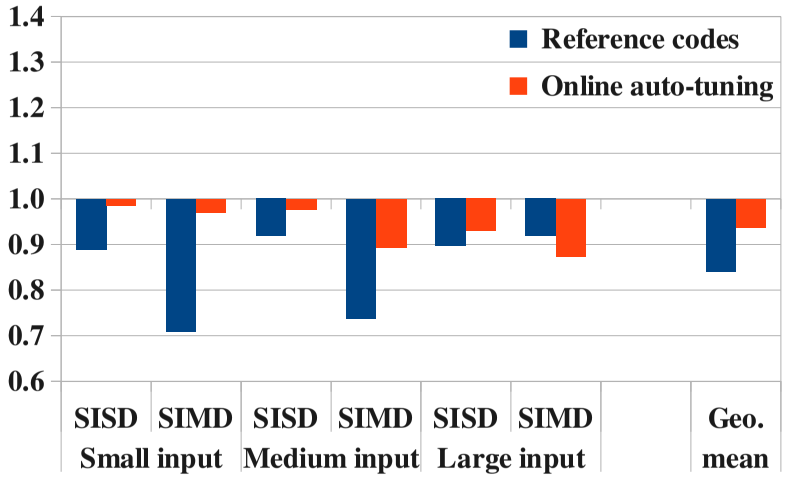
(c) Execution time with dim=128.

(d) Energy consumption with dim=32.

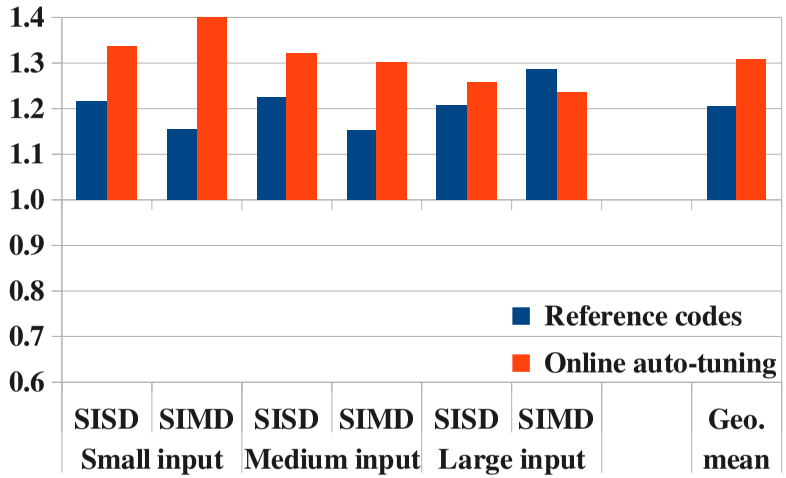
(e) Energy consumption with dim=64.

(f) Energy consumption with dim=128.

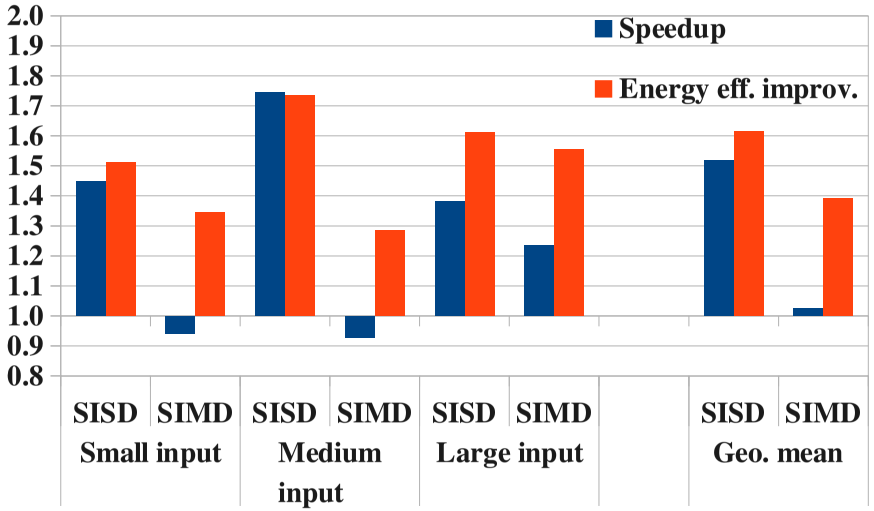
Figure 5.5: Energy and performance of the Streamcluster benchmark in the 11 simulated cores. Core abbreviations are listed in Table 5.2.



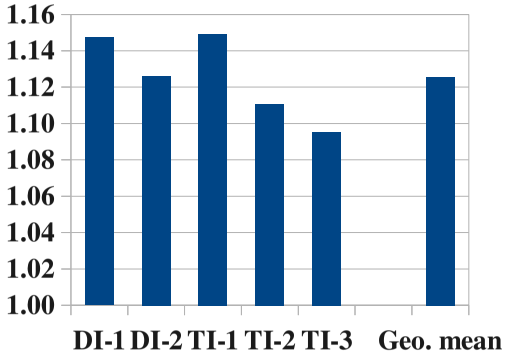
(a) Speedup of in-order vs out-of-order cores.



(b) Energy efficiency of in-order vs out-of-order cores.



(c) Energy and performance improvement of online auto-tuning in in-order vs reference in out-of-order cores.



(d) Area overhead of out-of-order vs in-order (excluding L2 caches).

Figure 5.6: Energy, performance and area of in-order vs out-of-order core designs, simulating the Streamcluster benchmark.

5.4.3 Analysis with varying workload

To better illustrate the behavior of the online auto-tuning framework, I further analyzed its behavior in the CPU-bound benchmark, with varying size of input set and hence execution time. The dimension was varied between 4 and 128 (the native dimension), and the workload through the number of points between 64 and 4096 (that of `simsmall`). The other parameters were kept fixed as those in the `simsmall` input set.

Figures 5.7 show all the results in the two real platforms. Globally, the online auto-tuning framework can find the right balance of space exploration for SISD code, on average obtaining speedups between 1.05 and 1.11, in applications that run for tens of milliseconds to tens of seconds. On the other hand, for SIMD auto-tuning, average speedups go from 0.80 to 1.29, with considerable slowdowns in the A8 when the applications run during less than one second.

Figures 5.7(a) and 5.7(c) show the speedups obtained in the Cortex-A8. We observe that SISD auto-tuning has almost always positive speedups, but SIMD auto-tuning shows considerable slowdowns with small workloads. There are two reasons for these slowdowns: the initial active function is the SISD reference code, and in the Cortex-A8, SISD FP instructions execute in the non-pipelined VFP extension, but the SIMD ones execute in the pipelined NEON unit. These two facts combined explain the observed slowdowns with small dimensions and workloads, because the benchmark starts executing SISD code from PARSEC, while the reference run-time comes from the SIMD code from PARVEC.

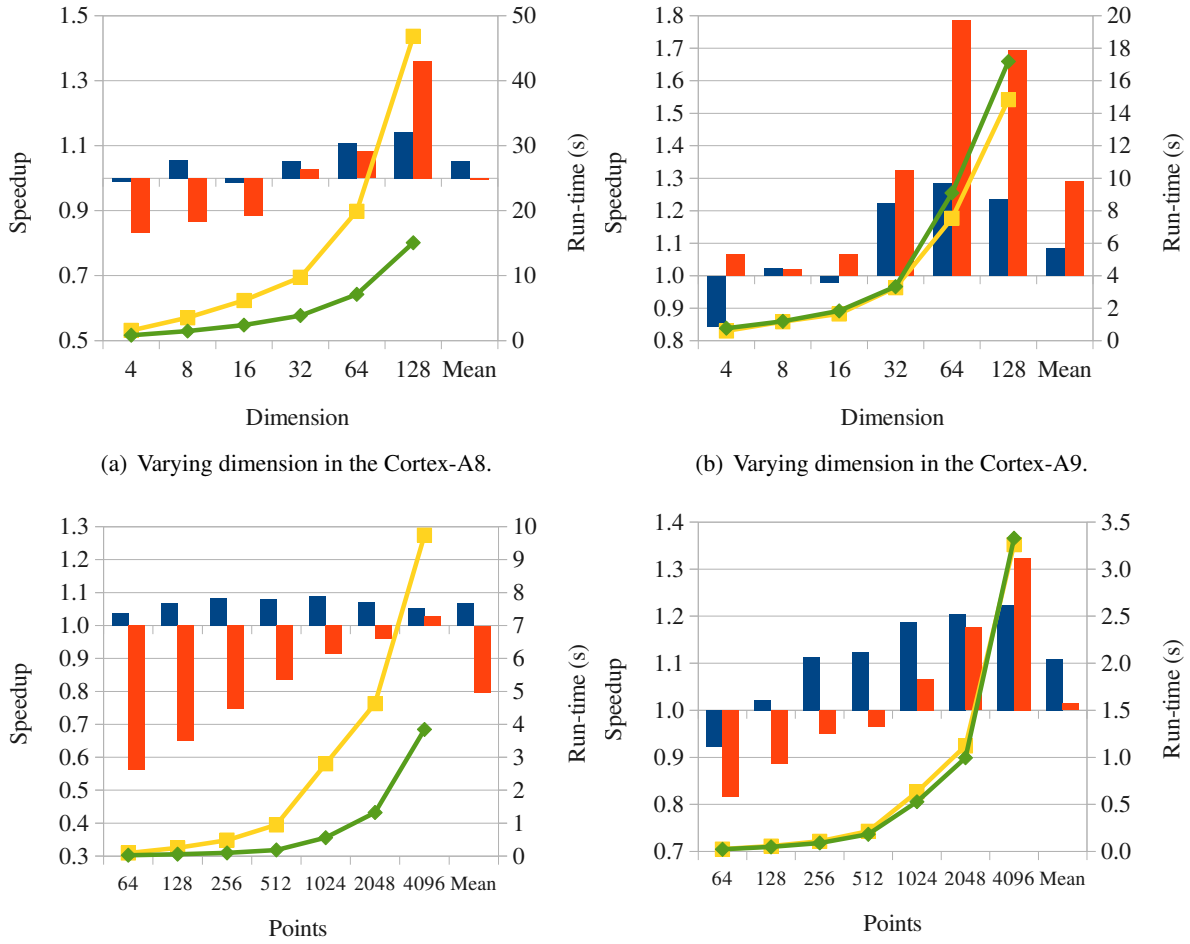
Figures 5.7(b) and 5.7(d) show the same analysis in the Cortex-A9. In this core, the VFP and NEON units are both pipelined. In consequence, the considerable slowdowns observed in the A8 with SIMD auto-tuning does not happen. On average, run-time auto-tuning provides positive speedups. As in the A8, SISD auto-tuning almost always results in positive speedups. SIMD auto-tuning shows slowdowns with small workloads, but after its crossover around 500 ms, high speedups are obtained, up to almost 1.8.

5.4.4 Analysis of correlation between auto-tuning parameters and pipeline designs

Table 5.5 shows the average auto-tuning parameter values of the best kernel configurations dynamically found in the 11 simulated cores, running the Streamcluster benchmark. Figure 5.8 presents them in the normalized range from 0 to 1. By analyzing the statistics, it is possible to correlate some of the most performing parameters to the pipeline designs:

- **`hotUF`**: This parameter loosely correlates with the dynamic scheduling capability of the pipeline. Given that it corresponds to the loop unrolling factor without register reuse, balanced out-of-order cores do not benefit from it, because register renaming does the same in hardware and allocating more registers can increase the stack and register management. Then, 3 of the 4 cores where `hotUF` was not 1 are in-order designs.

- **`coldUF`**: This parameter correlates with pipeline depth. It corresponds to the loop unrolling factor with register reuse, and benefits shallow EXE stages. This happens probably because of three factors: the dynamic instruction count (DIC) is reduced when a loop is unrolled, `coldUF` is the only parameter that allows aggressive unrolling factors, and deeper pipelines need more ILP



(a) Varying dimension in the Cortex-A8.

(b) Varying dimension in the Cortex-A9.

(c) Varying workload in the Cortex-A8.

(d) Varying workload in the Cortex-A9.

Figure 5.7: Analysis of the online auto-tuning speedups in Streamcluster with varying dimension and workload, compared to the static references. Other parameters are those from the `simsmall` input set.

than reduced DIC. In consequence, higher `coldUF` values are found in single- and dual-issue designs.

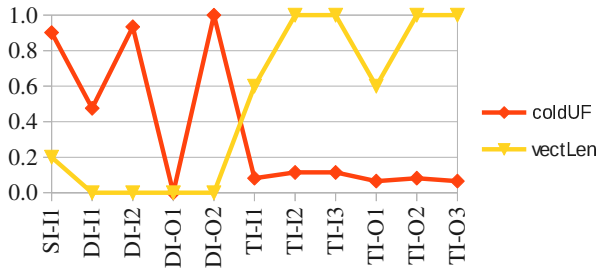
- **`vectLen`**: It correlates with the pipeline width. This parameter defines the length of processing vectors in the loop body, and enables higher ILP. That is why triple-issue designs have `vectLen` $\geq$ 3, while narrower pipelines have it around 2.

- **`pldStride`**: It has no clear correlation, possibly because all cores have stride prefetchers and the same L1 cache line length.

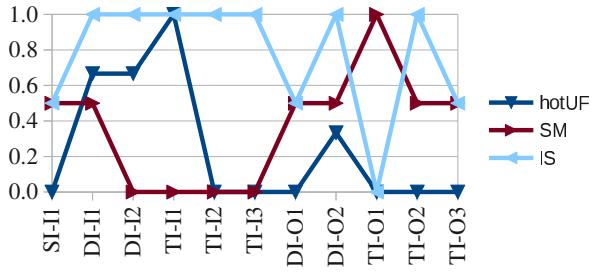
- **Stack minimization (SM)**: This code generation option has a loose correlation with the dynamic scheduling capability of the pipeline. Even with fewer architectural registers available for allocation, out-of-order designs can still get rid of false register dependencies by renaming architectural registers. The reduced stack management can then speed up execution. However, I found out that in most cases the generated code with SM is the same as that generated without SM. The observed results may be only a coincidence.

Table 5.5: Average of the best auto-tuning parameters for the Streamcluster benchmark, in the 11 simulated cores. Between parenthesis, the parameter ranges are shown.

Core	`hotUF` (1-4)	`coldUF` (1-64)	`vectLen` (1-4)	`pldStride` (0, 32, 64)	SM (0, 1)	IS (0, 1)
SI-I1	1.0	11.7	2.3	21	0.2	0.8
DI-I1	1.3	7.3	2.0	43	0.2	1.0
DI-I2	1.3	12.0	2.0	37	0.0	1.0
DI-O1	1.0	2.5	2.0	27	0.2	0.8
DI-O2	1.2	12.7	2.0	37	0.2	1.0
TI-I1	1.5	3.3	3.0	48	0.0	1.0
TI-I2	1.0	3.7	3.7	48	0.0	1.0
TI-I3	1.0	3.7	3.7	43	0.0	1.0
TI-O1	1.0	3.2	3.0	37	0.3	0.7
TI-O2	1.0	3.3	3.7	21	0.2	1.0
TI-O3	1.0	3.2	3.7	27	0.2	0.8



(a) Abscissa: from simpler to more complex cores.



(b) Abscissa: from simpler to more complex cores, in-order first.

Figure 5.8: Normalized values of the averaged best auto-tuning parameters for the Streamcluster benchmark, in the 11 simulated cores. `pldStride` is not shown. Core abbreviations are listed in Table 5.2.

- **Instruction scheduling (IS)**: All types of pipeline benefit from instruction scheduling. Out-of-order designs may sometimes not need scheduling at all, as we observe in 3 of 5 out-of-order cores, whose average utilization of IS was not 1.

This study corroborates the capability of the auto-tuning system to adapt code to different micro-architectures. As the study performed in the real platforms, a more detailed study would compare the results of run-time auto-tuning to the best statically found configuration. Nevertheless, to perform such an experiment, huge simulation resources are required, given that one benchmark run takes from 2.5 to more than 15 hours to be simulated.

5.5 Scope, limitations and future work

This chapter presented a study and proof-of-concept of a run-time auto-tuning system for short-running kernels in embedded systems. This section discusses the limitations, scope of application and future work.

5.5.1 Scope

Input- and machine-dependent optimizations Run-time auto-tuning is interesting in situations that static auto-tuning approaches fail. Poor performance portability over changing micro-architecture and input set are two examples studied in this work.

Small kernels, but not too small The proposed online auto-tuning approach uses deGoal as run-time code generator. Hence, the same limitations related to deGoal discussed in Section 4.4.2 applies to this auto-tuning approach.

Short-running kernels in general purpose embedded processors General purpose embedded processors are widely used in smartphones and computer tablets. Simple and common tasks as sharpening a picture may take only a few seconds in these devices. The prototype of auto-tuning tool presented in this chapter is adapted to these short-running applications based on computing kernels, whose code can be better optimized only when a user inputs a given parameter (e.g., sharpening factor or image size).

5.5.2 Limitations

Comparison to state-of-the-art auto-tuners I measured considerable speedups obtained from the proposed run-time approach when compared to statically compiled codes from the reference benchmarks, and demonstrated that the best auto-tuning choice depends on the target core and that performance from run-time auto-tuning is only 6 % away from the best static ones. However, a more complete study would compare the run-time auto-tuned kernels to statically auto-tuned ones from state-of-the-art tools, in order to verify if variations of micro-architecture features and input set effectively impose a poor performance portability on state-of-the-art compile-time auto-tuners.

Pre-profiling Given the run-time constraint, most auto-tuned parameters, their initial values and space exploration ranges were obtained through a pre-profiling phase. Therefore, these pre-profiled parameters may not be adapted to all dynamic execution environments.

Measurement stability and execution context adaptability Obtaining stable execution time measurements of an application in a very short period is not an easy task. Given the application scope of the online auto-tuning tool, the evaluation of new kernel versions should be very fast in order to allow as many versions as possible in the tuning space to be evaluated. Hence, new kernels should be ideally

evaluated in the duration range of microseconds or less. Because of natural hardware and software oscillations, one of the strategies adopted was to use a fixed training input set with warmed caches in the first auto-tuning phase, with the aim of finding the best kernel configuration for the underlying micro-architecture. Despite of stable measurements, this technique can not take into account the execution context of the kernel. In some circumstances, complex interactions between code, pipeline and caches can create recurrent stalls and slow down the application. Such interactions may not be captured with fixed input set and warmed caches.

Bypasses not simulated In real ARM pipelines, successive FMAC instructions that depend on the destination register (the accumulator) may use a special bypass path to effectively reduce the result-use stall [97]. For example, the Cortex-A9 implements this bypass [11, Section 3.4.2], then such FMAC sequence can be issued with a throughput of one instruction each three or four cycles, which is probably the latency of the FP ADD pipeline, instead of eight cycles, which is the latency of FMAC instructions. Such special bypasses are quite useful in long pipelines as the simulated triple-issue cores, but gem5 does not model them.[2] In those deep pipelines, the result-use stalls equal the latency of the FMAC instruction (20 cycles), instead of the latency of the FP ADD pipeline (10 cycles). In the Streamcluster kernel, gcc generates a dependent chain of FMAC. In consequence, the observed SISD speedups obtained by simulation may be overestimated.[3]

5.5.3 Future work

Automatic pre-profiling or adaptive sampling Two possible solutions could be employed to address the pre-profiling currently required to limit the exploration space:

- Divide the auto-tuning process into two phases, a static one (ahead-of-time) in which the pre-profiling is automatically performed to build a simple model of the target(s) core(s), through micro-benchmarking for instance, and the dynamic one in which the auto-tuning process itself is performed and the exploration ranges are determined from the target model.

- Embed an adaptive sampler [52] to dynamically explore the entire implementation space and to possibly speed-up the process of finding good kernel versions.

Continuous adaptive auto-tuning The execution environment of an application changes depending on system load and the behavior of concurrent processes. The applications themselves may have various different phases. In consequence, the best kernel version may also vary. In such contexts, an auto-tuning system could periodically and continuously revisit old kernel versions to re-adapt the kernel code to the changing application phases.

[2] The EXE stage model in gem5 can be viewed either as an always or never bypass model. Intermediate situations, i.e., partial bypassing is not modeled.

[3] Considering that the FP SIMD pipelines in the A8 and A9 do not have that special bypass, the observed SIMD speedups are more trustworthy.

5.6 Conclusion

In this chapter, I presented a methodology to implement run-time auto-tuning kernels in short-running applications. This work advances the state of the art of online auto-tuning. To the best of my knowledge, this work is the first to implement and demonstrate that online auto-tuning of computing kernels can considerably speed-up short-running kernel-based applications in embedded systems. My approach can both adapt a kernel implementation to a micro-architecture unknown prior compilation and dynamically explore auto-tuning possibilities that are input-dependent.

I demonstrated through two case studies in real and simulated platforms that the proposed approach can considerably speedup a CPU-bound kernel-based application up to 1.79 and 2.53, respectively, and has negligible run-time overheads when auto-tuning does not provide better kernel versions. In the second application, even if the bottleneck is in the main memory, we observed speedups up to 1.30 in real cores, because of the reduced number of instructions executed in the auto-tuned versions.

Energy consumption is the most constraining factor in current high-performance embedded systems. By simulating the CPU-bound application in 11 different cores, I showed that run-time auto-tuning can reduce the performance gap between in-order and out-order cores from 16 % (static compilation) to only 6 %. In addition, I demonstrated that online micro-architectural adaption of code to in-order pipelines can on average outperform the hand vectorized references run in similar out-of-order cores. Despite the clear hardware disadvantage, online auto-tuning in in-order cores obtained an average speedup of 1.03 and an energy efficiency improvement of 39 % over the SIMD reference in out-of-order cores.

Chapter 6

Conclusion and prospects

According to studies of computer architecture trends, the main hypothesis of this thesis is that future computing SoCs will embed hundreds or thousands of heterogeneous clusters of cores, each one adapted to a given task. Eventually, the cores may also embed heterogeneous pipelines (i.e., clustered cores). All this heterogeneity will be needed to increase the energy efficiency of hardware, but software development will face increasing difficulties to obtain the energy and performance promised by heterogeneous hardware. Run-time techniques will likely be required to address the complexity and to adapt software to the dynamic behavior of the system.

Considering the previous hypothetical scenario, this thesis proposed a methodology and implemented a proof of concept of online auto-tuning framework for embedded processors. A simulation framework of ARM cores was developed to study the capability of the proposed auto-tuning approach to adapt code to various micro-architectures.

6.1 Achievements

This thesis was developed around two main axes: embedded core simulation, and run-time code generation and auto-tuning for embedded systems.

6.1.1 Embedded core simulation with gem5 and McPAT

Micro-architectural simulation is used to evaluate new ideas of hardware, explore a design space or to simulate hypothetical hardware implementations. For embedded core studies, gem5 and McPAT are among the best choices of micro-architectural simulators.

However, in the beginning of this thesis, gem5 lacked an in-order model for ARM. Therefore, **I developed a cycle-approximate in-order model** by modifying the functional out-of-order one. **I validated both the proposed in-order and the original out-of-order model in gem5**, configured as Cortex-A8 and A9 and compared to real hardware. My experiments showed that both models have **average absolute timing errors around 7 %** when running 10 complex PARSEC 3.0 benchmarks. These results demon-

strate that gem5 is considerably more accurate than similar simulators extensively validated, which have mean absolute errors greater than 15 %.

To obtain energy estimations, **I analyzed the pipeline models in gem5 and McPAT** to create conversion rules of parameters and statistics from the first to the latter. By analyzing the estimated area of ARM cores, **I proposed a simple modification in McPAT and a methodology** to better estimate the empirical area and energy cost of FUs of heterogeneous cores. **I validated the area estimation of McPAT** by configuring big.LITTLE CPUs: **within 4 % for cores** and **up to 13 % for clusters**. Given that ARM released the floorplan of the Cortex-A7, I also compared the area estimation of five main structures of its pipeline. Although the total core area exactly matched, I observed errors from 3.6 to 60 %, but the structure that impacts the most the core area error contributes with only 5.3 % to this error. As a relative energy/performance validation of the proposed simulation framework, I simulated 11 benchmarks in big.LITTLE cores. The **relative energy/performance estimations** are **within 6 %** of published data for the Dhrystone 2.1 benchmark.

This simulation framework permitted to study embedded core heterogeneity and to evaluate the code adaptivity of the proposed run-time auto-tuning approach to various micro-architectures.

6.1.2 Run-time code generation and auto-tuning for embedded systems

Existing run-time code generation and auto-tuning systems are mostly developed and mature for non-embedded systems. To my knowledge, **this thesis is the first work to address run-time auto-tuning in embedded systems**.

The methodology and proof of concept of **a run-time auto-tuning tool for embedded systems** was developed around **two case studies**: a CPU- and a memory-bound kernel-based applications. Each application was compared in their **SISD and SIMD versions** and run with **three different input sets**. In these studies, I focused on micro-architectural adaption of code, thanks to loop unrolling, vector length and pre-fetching auto-tuning.

In order to accelerate the auto-tuning process, **I proposed a two phase exploration**, based on empirical observations.

The experiments were carried out in the **Cortex-A8 and A9**, and also in **11 simulated heterogeneous ARM cores** with varying issue widths, FP/SIMD units and static or dynamic scheduling capabilities.

In the CPU-bound, we observed **average speedups of 1.26 and 1.38** in real and simulated cores, respectively, **going up to 1.79 and 2.53** (all run-time overheads included). We also observed that the proposed approach produces **negligible slowdowns** when auto-tuning does not provide better kernel versions.

By analysing the simulation results of the CPU-bound application in in-order compared to similar out-of-order cores, we observed that the proposed **run-time auto-tuning** approach can **reduce the performance gap from 16 to 6 %** and **increase the energy efficiency from 21 to 31 %**.

In addition, I demonstrated that the **online micro-architectural adaption of code to in-order pipelines** can on average **outperform** the **hand vectorized references run in similar out-of-order cores**: despite the clear hardware disadvantage, the proposed approach applied to the CPU-bound appli-

cation obtained an **average speedup of 1.03** and an **energy efficiency improvement of 39 %**.

In real hardware, the **run-time auto-tuning performance** is on average **only 6 % away** from the performance obtained by the best statically found kernel implementations.

To develop this auto-tuning system, **I ported and extended deGoal**, a lightweight run-time code generator, to ARM processors. To validate the code quality produced by deGoal, **I implemented and evaluated** SISD and SIMD versions of four kernels found in the PARSEC 3.0 and PARVEC benchmark suites.

The results obtained in Cortex-A8 and A9 cores demonstrated that **on average deGoal generates equivalent or higher quality machine code** even compared to the hand vectorized kernels from PARVEC.

Experiments of **dynamic code specialization** with the same kernels resulted in **average speedups of 1.32 and 1.07** compared to the PARSEC and PARVEC references, respectively, with negligible **run-time overheads of less than 0.1 %** of the kernel execution times. Because of this very low run-time overhead and of observed performance asymmetries, I argued that deGoal could be used to auto-tune computing kernels in short-running applications.

6.1.3 Summary of achievements

The following list summarizes the achievements:

- Embedded core simulation with gem5 and McPAT:
  - Proposal and development of a cycle-approximate in-order model for ARM in gem5.
  - Validation of the proposed in-order and the original out-of-order model in gem5 against a Cortex-A8 and an A9.
  - Modeling enhancements of gem5 to better simulate embedded cores.
  - Study of pipeline models and development of rules to convert parameters and statistics from gem5 to McPAT.
  - Modeling enhancement of McPAT to better estimate the area and energy consumption of FUs in heterogeneous cores.
  - Area and relative energy/performance validation of the proposed simulation framework against big.LITTLE CPUs.
- Run-time code generation and auto-tuning:
  - Porting of deGoal to the ARM Thumb-2 ISA, including FP and SIMD extensions.
  - Preliminary validation of SISD and SIMD code generation in ARM processors.
  - Study of run-time specialization and auto-tuning possibilities with deGoal.
  - Proposal of methodology, proof of concept and study of run-time auto-tuning in real and simulated embedded ARM cores.

6.1.4 Amount of work

An estimation of the time spent in each stage of the thesis is given in the following.

- **State of the art**: 3 months.
- **gem5/McPAT development**: 4.9 months.
  - Kickoff: 1 month.
  - Study of gem5 and McPAT models, parameters and statistics conversion: 2 months.
  - Cycle-approximate in-order model: 1.2 month.
  - McPAT enhancement and big.LITTLE area validation: 0.7 month.
- **gem5/McPAT experimentations and paper writing**: 3.3 months.
  - Benchmark compilation environment, in-order and out-of-order model validation against A8 and A9: 2.6 months.
  - Energy estimation and simulation analysis of the A8: 0.2 month.
  - Configuration of big.LITTLE CPUs and simulations: 0.6 month.
- **deGoal development**: 4.9 months.
  - Kickoff: 1 month.
  - Porting to the ARM Thumb-2 ISA: 1.5 month.
  - Enhancements: 2.4 months.
- **deGoal experimentations**: 3.3 months.
  - Compilation environment, performance counter measurement and experimentations with the convolution kernel: 1.6 month.
  - Linear transformation kernel: 0.6 month.
  - Interpolation kernel: 0.4 month.
  - Euclidean distance kernel: 0.4 month.
  - Paper writing: 0.3 month.
- **Methodology and proof of concept of run-time auto-tuning**: 2.9 months.
- **Thesis writing**: 2.6 months.

6.2 Prospects

Because of the power wall and dark silicon, there is a growing trend of embedding heterogeneous computing units into SoCs. This growing complexity will affect both architectural design and software development. In order to avoid a performance stagnation of computing systems in the next decades, considerable efforts will be needed to change the current architectural and programming paradigms [82].

In light of the performance and energy efficiency challenge faced by the next generations of computing systems, architectural and micro-architectural studies can provide potential new insights to increase hardware performance with the same energy and power budgets. We have seen the trend of computing with CPUs and general-purpose computing on graphics processing units (GPGPUs), heterogeneous cores and more recently the mix of fixed and reconfigurable computing logic. The next trends in micro-architectural designs may not only include special FUs and accelerators integrated into generic pipelines for specific computing tasks and for approximative computation, but also specialized cores to specific tasks and constraints, such as near-threshold computing cores for high energy-efficiency and the EOLE architecture, which mitigates out-of-order complexity through value prediction to accelerate the single-threaded regions of applications [111]. The simulation framework developed during this thesis is a good starting point for micro-architectural studies, because acceptable area, energy and performance estimations of current embedded core designs have been demonstrated through several experiments.

Considering the growing performance portability problem, this thesis studied the feasibility of pushing run-time auto-tuning to embedded systems. In future heterogeneous manycores, run-time approaches may be the only solution to further increase the energy efficiency in dynamic environments. This thesis studied run-time auto-tuning of computing kernels in single-cores. Given that no extra cores were used for code regeneration and training, and that the run-time code generation cost is low, therefore the proposed approach is scalable to asymmetric multi/manycores. In such context, online auto-tuning can provide better performance portability compared to statically compiled binaries. The very fast auto-tuning capability observed in my experiments would also allow to perform auto-tuning in ahead-of-time compilers with small compilation time overheads. Another future prospect would be the definition and implementation of a higher-level language for auto-tuning, which would be automatically translated to deGoal compilettes.

Part II

Appendix

Appendix A

gem5 to McPAT conversion tables

Table A.1: Core parameter translation from gem5 to McPAT.

McPAT parameter	gem5 equivalent	Description
clock_rate		Clock rate, in gem5 the clock period is specified in picoseconds.
opt_local		0: Core configuration is known to be feasible, 1: Locally optimizes timing constraints given by the technology node and warns if timing is not met.
instruction_length	32	Instruction length in bits (16-bit Thumb instructions are decompressed into 32-bit ones).
opcode_width	7	Effective number of opcode bits. Describes the average pipeline complexity, is used for example to estimate the number of pipeline registers.
x86	0	Only for x86-like designs.
micro_opcode_width	-	Only for x86-like designs.
machine_type		0: out-of-order, 1: in-order.
number_hardware_threads	0	For SMT only.
fetch_width	$\mathrm{fetchWidth}$	Number of instructions fetched per cycle.
number_instruction_fetch_ports	1	Number of R/W ports in various L1-I cache structures.
decode_width	$\mathrm{decodeWidth}$	Number of instructions decoded per cycle.
issue_width	$\mathrm{decodeWidth}$	Sustainable issue width: sustainable number of instructions issued per cycle.
peak_issue_width	$\mathrm{issueWidth}$	Maximum number of instructions issued per cycle.
commit_width	$\mathrm{commitWidth}$	Number of instruction committed per cycle.
fp_issue_width		gem5 has a shared IQ for INT and FP. This is the number of FP instructions issued per cycle. Equals the minimum number between FP execution ports and $\mathrm{issueWidth}$.
prediction_width	1	Number of simultaneous branch predictions per cycle. Number of read and write ports in branch-related predictors.
pipelines_per_core	$(1, \mathrm{X})$	Number of (INT, FP) pipelines. X is 0 if no FP support, 1 otherwise.
pipeline_depth	(X, Y)	(INT, FP) pipeline depth, i.e., number of pipeline stages. $\mathrm{X} = \mathrm{BASE} + \mathrm{INT\_EXE}$, $\mathrm{Y} = \mathrm{BASE} + \mathrm{FP\_EXE}$, where $\mathrm{BASE} = \mathrm{fetchToDecodeDelay} + \mathrm{decodeToRenameDelay} + \mathrm{renameToIEWDelay} + \mathrm{iewToCommitDelay} + \max(\mathrm{commitToDecodeDelay}, \mathrm{commitToFetchDelay}, \mathrm{commitToIEWDelay}, \mathrm{commitToRenameDelay})$ is the shared pipeline stages between INT and FP, $\mathrm{INT\_EXE}$ and $\mathrm{FP\_EXE}$ are the depths of the INT and FP EXE stage (indirectly modeled in gem5 through instruction latencies). Typical values for ARM are 1-2 and 4-10, respectively.

continued on next page

McPAT parameter	gem5 equivalent	Description
ALU_per_core		Number of Simple ALUs.
MUL_per_core		Number of Complex ALUs, containing one multiplier and one divider.
FPU_per_core		Number of FP/SIMD units.
instruction_buffer_size	fetchBufferSize	Fetch buffer size in bytes. Only present in recent gem5 versions.
instruction_window_scheme	0	0: Physical register file, 1: Reservation station.
decoded_stream_buffer_size	-	Not used yet.
instruction_window_size	numIQEntries/2	gem5 has a shared IQ for INT and FP. Hence, dividing by two.
fp_instruction_window_size	numIQEntries/2	gem5 has a shared IQ for INT and FP. Hence, dividing by two (the FP queue may be larger than the INT one).
ROB_size	numROBEntries	Number of ROB entries.
archi_Regs_IRF_size	32	Number of architectural INT registers.
archi_Regs_FRF_size	32	Number of architectural FP registers.
phy_Regs_IRF_size	numPhysIntRegs	Number of physical INT registers.
phy_Regs_FRF_size	numPhysFloatRegs	Number of physical FP registers.
rename_scheme	0	0: RAM-based, 1: CAM-based.
register_windows_size	0	For Sun processors.
LSU_order	-	Not used yet.
store_buffer_size	SQEntries	Number of store queue entries.
load_buffer_size	LQEntries	Number of load queue entries.
memory_ports		Number of R/W ports in the L1-D cache and other LSU structures. I considered that it equals the number of execution ports for the load unit.
RAS_size	RASSize	Number of entries in the return address stack.
number_of_BPT	-	Not used yet.

Table A.2: Core statistics translation from gem5 to McPAT.

McPAT statistics	gem5 equivalent	Description
total_instructions	decode.DecodedInsts	Number of instructions crossing the fetch and decode stages.
int_instructions	-	INT micro-operations executed. Only for In-order SMT processors.
fp_instructions	-	FP micro-operations executed. Only for In-order SMT processors.
branch_instructions	BPredUnit.condPredicted	Predicted branches.
branch_mispredictions	BPredUnit.condIncorrect	Mispredicted branches.
load_instructions	iew.iewExecLoadInsts	Executed load instructions.
store_instructions	iew.exec_refs − iew.iewExecLoadInsts	Executed store instructions.
committed_instructions	-	Committed instructions, not used yet.
committed_int_instructions	commit.int_insts	For out-of-order model with reservation station.
committed_fp_instructions	commit.fp_insts	For out-of-order model with reservation station.
pipeline_duty_cycle	ipc_total/issue_width	Equals the IPC divided by the peak IPC (in-order only).
total_cycles	numCycles	Number of cycles during which the core was powered.
idle_cycles	idleCycles	Number of cycles during which the core was idling.
busy_cycles	numCycles − idleCycles	Number of cycles during which the core was busy.
ROB_reads	rob.rob_reads	Number of ROB reads accesses.
ROB_writes	rob.rob_writes	Number of ROB writes accesses.
rename_reads	rename.int_rename_lookups	Number of read accesses to INT renaming structures.
rename_writes	rename.int_rename_operands	Number of write accesses to INT renaming structures.
fp_rename_reads	rename.fp_rename_lookups	Number of read accesses to FP renaming structures.
fp_rename_writes	rename.fp_rename_operands	Number of write accesses to FP renaming structures.
inst_window_reads	iq.int_inst_queue_reads	Number of read accesses to the INT issue IQ.
inst_window_writes	iq.int_inst_queue_writes	Number of write accesses to the INT issue IQ.
inst_window_wakeup_accesses	iq.int_inst_queue_wakeup_accesses	Number of instructions that waked up depending INT instructions.

continued on next page

McPAT statistics	gem5 equivalent	Description
fp_inst_window_reads	iq.fp_inst_queue_reads	Number of read accesses to the FP issue IQ.
fp_inst_window_writes	iq.fp_inst_queue_writes	Number of write accesses to the FP issue IQ.
fp_inst_window_wakeup_accesses	iq.fp_inst_queue_wakeup_accesses	Number of instructions that waked up depending FP instructions.
int_regfile_reads	int_regfile_reads	Number of read accesses to the INT register file.
float_regfile_reads	fp_regfile_reads	Number of read accesses to the FP register file.
int_regfile_writes	int_regfile_writes	Number of write accesses to the INT register file.
float_regfile_writes	fp_regfile_writes	Number of write accesses to the FP register file.
function_calls	commit.function_calls	Number of function calls. In non-SMT processors, BPredUnit.usedRAS is more accurate.
context_switches	-	Not used yet.
ialu_accesses		Number of Simple ALU accesses per cycle and per stage. Roughly iq.FU_type_0 :: IntAlu multiplied by the IntAlu opClass latency.
fpu_accesses		Number of FP/SIMD accesses per cycle and per stage. Roughly the sum of all iq.FU_type_0 :: Float and iq.FU_type_0 :: Simd multiplied by the respective opClass latencies.
mul_accesses		Number of Complex ALU accesses per cycle and per stage. Roughly the sum of iq.FU_type_0 :: IntDiv and iq.FU_type_0 :: IntMult multiplied by the respective opClass latencies.
cdb_alu_accesses	$\frac{\text{iew.wb\_int\_producers} * \text{iq.FU\_type\_0::IntAlu}}{\text{iq.FU\_type\_0::IntAlu} + \text{iq.FU\_type\_0::IntMult}}$	Number of produced results by the Simple ALUs.
cdb_mul_accesses	$\frac{\text{iew.wb\_int\_producers} * \text{iq.FU\_type\_0::IntMult}}{\text{iq.FU\_type\_0::IntAlu} + \text{iq.FU\_type\_0::IntMult}}$	Number of produced results by the Complex ALUs.
cdb_fpu_accesses	iew.wb_producers − iew.wb_int_producers	Number of produced results by FP/SIMD units.
*_duty_cycle	-	Except pipeline_duty_cycle (previously described), these are only for peak power estimation.

Table A.3: BP, BTB, TLB and L1/L2 cache parameter translation from gem5 to McPAT.

McPAT parameter		gem5 equivalent	Description
BP	local_predictor_size	(localHistoryBits, localCtrBits)	Number of bits (i.e., as in the classic n-bit predictor) in each entry of the first and second levels of local branch prediction, respectively.
	local_predictor_entries	localPredictorSize	Number of entries in the first and second level of local history tables (in gem5, each level can have a different number of entries: localHistoryTableSize and localPredictorSize, respectively).
	global_predictor_entries	globalPredictorSize	Number of entries in the global history table.
	global_predictor_bits	globalCtrBits	Number of bits in each entry of the global history table.
	chooser_predictor_entries	choicePredictorSize	Number of entries in the chooser table.
	chooser_predictor_bits	choiceCtrBits	Number of bits in each entry of the chooser table.
BTB	BTB_config	(BTBEntries, 4, 2, 2, 1, 1)[1]	Same meaning as the six first parameters in icache_config/dcache_config.
L1 TLBs	number_entries	size	Number of entries in the L1 TLBs.
L1 caches	icache_config/dcache_config	(size, block_size, assoc, 1, X, Y, 32, Z)	Size, block size or line length in bytes, associativity, number of banks, throughput in core cycles, latency in core cycles, output width (not used yet), cache policy, respectively. X = Y are the ratio of cache latency and core clock period, and Z is 0 for the L1-I cache (no write) and 1 for the L1-D cache (write-back with write-allocate).
	buffer_sizes	(mshrs, mshrs, X, Y)	MSHR, fill buffer, prefetcher buffer and WB sizes, respectively. X is mshrs if no prefetcher is present (zero is not accepted) or the size of the prefetcher buffer. Y is 0 for L1-I caches and write_buffers for L1-D caches.
L2 cache	L2_config		See icache_config/dcache_config.
	buffer_sizes		See buffer_sizes in L1 caches.
	clockrate		Core clock to which the cache is attached in MHz.
	ports	-	Not used yet.
	device_type		0: HP (High Performance Type), 1: LSTP (Low standby power) or 2: LOP (Low Operating Power).

[1] The BTB in gem5 is direct mapped, then an associativity equals to 1 better models it.

Table A.4: BTB, TLB and L1/L2 cache statistics[1] translation from gem5 to McPAT.

McPAT statistics		gem5 equivalent	Description
BTB	read_accesses	BPredUnit.BTBLookups	Number of read accesses.
	write_accesses	BPredUnit.BTBUpdates	Number of write accesses.
L1 TLBs	total_accesses	$*$tb.accesses	Number of read accesses.
	total_misses	$*$tb.misses	Number of misses.
	conflicts	-	Not used yet.
L1 caches[6]	read_accesses	ReadReq_accesses :: total[2]	Number of read accesses.
	write_accesses	WriteReq_accesses :: total[3]	Number of write accesses.
	read_misses	ReadReq_misses :: total[4]	Number of read misses.
	write_misses	WriteReq_accesses :: total $-$ WriteReq_hits :: total[5]	Number of write misses.
	conflicts	-	Not used yet.
L2 cache[6]	read_accesses	ReadReq_accesses :: total[2]	Number of read accesses.
	write_accesses	overall_accesses :: total $-$ ReadReq_accesses :: total $+$ Writeback_accesses :: total[3]	Number of write accesses.
	read_misses	ReadReq_misses :: total[4]	Number of read misses.
	write_misses	overall_misses :: total $-$ ReadReq_misses :: total[5]	Number of write misses.
	conflicts	-	Not used yet.
	duty_cycle	-	Only for peak power estimation.

[1] Branch predictor statistics are informed in the core component.
[2] This is not accurate, overall_accesses :: total $-$ WriteReq_accesses :: total should be used instead.
[3] WriteReq_accesses :: total $+$ Writeback_accesses :: total should be used instead.
[4] overall_misses :: total $-$ WriteReq_misses :: total should be used instead.
[5] Writeback_misses :: total $+$ WriteReq_misses :: total should be used instead.
[6] List of cache signals: `http://www.gem5.org/Packet_Command_Attributes` [Accessed: 3 June 2015].

List of Figures

List of Tables

Bibliography

[1] Jung Ho Ahn, Sheng Li, O Seongil, and Norman P. Jouppi. McSimA+: A Manycore Simulator with Application-level+ Simulation and Detailed Microarchitecture Modeling. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pages 74–85. IEEE, 2013.

[2] Yeter Akgul. *Gestion de la consommation basée sur l'adaptation dynamique de la tension, fréquence et body bias sur les systèmes sur puce en technologie FD-SOI*. PhD thesis, December 2014.

[3] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. PetaBricks: A Language and Compiler for Algorithmic Choice. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 38–49. ACM, 2009.

[4] Jason Ansel, Maciej Pacula, Yee Lok Wong, Cy Chan, Marek Olszewski, Una-May O'Reilly, and Saman Amarasinghe. SiblingRivalry: Online Autotuning Through Local Competitions. In *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '12, pages 91–100. ACM, 2012.

[5] ARM. *CoreLink Level 2 Cache Controller L2C-310 Technical Reference Manual*. Revision: r3p3.

[6] ARM. The ARM Cortex-A9 Processors. *ARM White paper*, 2009. Document Revision 2.0 Sept 2009.

[7] ARM. *Cortex-A8 Technical Reference Manual*, May 2010. Revision: r3p2.

[8] ARM. *Cortex-A Series Programmer's Guide*, June 2012. Version: 3.0.

[9] ARM. *Cortex-A15 MPCore Technical Reference Manual*, July 2012. Revision: r3p2.

[10] ARM. *Cortex-A9 Floating-Point Unit Technical Reference Manual*, June 2012. Revision: r4p1.

[11] ARM. *Cortex-A9 NEON Media Processing Engine Technical Reference Manual*, June 2012. Revision: r4p1.

[12] ARM. *Cortex-A9 Technical Reference Manual*, June 2012. Revision: r4p1.

[13] ARM. *Cortex-A7 MPCore Technical Reference Manual*, April 2013. Revision: r0p5.

[14] ARM. *ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile*, July 2015. Issue: A.g.

[15] ARM website. A9-osprey-hres.jpg. `http://www.arm.com/images/A9-osprey-hres.jpg` [Accessed: 14 April 2015].

[16] ARM website. Cortex-A5 Processor.

[17] ARM website. Cortex-A7 Processor.

[18] ARM website. Cortex-A9 Processor.

[19] ARM website. Single_Cortex-A7_core_layout_image.jpg. `http://www.arm.com/images/Single_Cortex-A7_core_layout_image.jpg` [Accessed: 20 April 2015].

[20] C. Auth, A. Cappellani, J.-S. Chun, A. Dalis, A. Davis, T. Ghani, G. Glass, T. Glassman, M. Harper, M. Hattendorf, P. Hentges, S. Jaloviar, S. Joshi, J. Klaus, K. Kuhn, D. Lavric, M. Lu, H. Mariappan, K. Mistry, B. Norris, N. Rahhal-orabi, P. Ranade, J. Sandford, L. Shifren, V. Souw, K. Tone, F. Tambwe, A. Thompson, D. Towner, T. Troeger, P. Vandervoorn, C. Wallace, J. Wiedemer, and C. Wiegand. 45nm High-k + Metal Gate Strain-Enhanced Transistors. In *VLSI Technology, 2008 Symposium on*, pages 128–129, June 2008.

[21] José L. Ayala, Alexander Veidenbaum, and Marisa López-Vallejo. Power-Aware Compilation for Register File Energy Reduction. *International Journal of Parallel Programming*, 31(6):451–467, December 2003.

[22] R. Iris Bahar and Srilatha Manne. Power and Energy Reduction Via Pipeline Balancing. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, ISCA '01, pages 218–229. ACM, 2001.

[23] 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*, PLDI '00, pages 1–12. ACM, 2000.

[24] Protonu Basu, Mary Hall, Malik Khan, Suchit Maindola, Saurav Muralidharan, Shreyas Ramalingam, Axel Rivera, Manu Shantharam, and Anand Venkat. Towards making autotuning mainstream. *International Journal of High Performance Computing Applications*, 27(4):379–393, November 2013.

[25] BeagleBoard.org. *BeagleBoard-xM Rev C System Reference Manual*, April 2010. Revision 1.0.

[26] Olav Beckmann, Alastair Houghton, Michael Mellor, and Paul H.J. Kelly. Runtime Code Generation in C++ as a Foundation for Domain-Specific Optimisation. In Christian Lengauer, Don Batory, Charles Consel, and Martin Odersky, editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 291–306. Springer Berlin Heidelberg, 2004.

[27] Shajulin Benedict. Energy-aware performance analysis methodologies for HPC architectures–An exploratory study. *Journal of Network and Computer Applications*, 35(6):1709–1719, November 2012.

[28] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[29] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 Simulator. *SIGARCH Computer Architecture News*, 39(2):1–7, May 2011.

[30] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The M5 Simulator: Modeling Networked Systems. *IEEE Micro*, 26(4):52–60, July 2006.

[31] Emily Blem, Jaikrishnan Menon, and Karthikeyan Sankaralingam. Power Struggles: Revisiting the RISC vs. CISC Debate on Contemporary ARM and x86 Architectures. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture*, HPCA '13, pages 1–12, 2013.

[32] Shekhar Borkar and Andrew A. Chien. The Future of Microprocessors. *Communications of the ACM*, 54(5):67–77, May 2011.

[33] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 83–94. ACM, 2000.

[34] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An Infrastructure for Adaptive Dynamic Optimization. In *International Symposium on Code Generation and Optimization*, CGO '03, pages 265–275. IEEE Computer Society, 2003.

[35] Doug Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. *SIGARCH Computer Architecture News*, 25(3):13–25, June 1997.

[36] Doug Burger, Todd M. Austin, and Stephen W. Keckler. Recent Extensions to the SimpleScalar Tool Suite. *SIGMETRICS Performance Evaluation Review*, 31(4):4–7, March 2004.

[37] Anastasiia Butko, Rafael Garibotti, Luciano Ost, and Gilles Sassatelli. Accuracy Evaluation of GEM5 Simulator System. In Indrusiak, LS and Gogniat, G and Voros, N, editor, *2012 7th International Workshop on Reconfigurable and Communication-centric Systems-on-Chip (ReCoSoC)*. IEEE, 2012.

[38] Calao Systems. *SKY-S9500-ULP-CXX (aka Snowball PDK-SDK) Hardware Reference Manual*, July 2011. Revision 1.0.

[39] Juan M. Cebrian, Magnus Jahre, and L. Natvig. Optimized Hardware for Suboptimal Software: The Case for SIMD-aware Benchmarks. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, ISPASS '14, pages 66–75, March 2014.

[40] Anantha P. Chandrakasan, William J. Bowhill, and Frank Fox. *Design of High-Performance Microprocessor Circuits.* New York: IEEE Press, 2000. ISBN: 078036001X.

[41] Kiran Chandramohan and Michael F. P. O'Boyle. A Compiler Framework for Automatically Mapping Data Parallel Programs to Heterogeneous MPSoCs. In *2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '14, pages 9:1–9:10. ACM, 2014.

[42] Naehyuck Chang, Kwanho Kim, and Hyung Gyu Lee. Cycle-Accurate Energy Consumption Measurement and Analysis: Case Study of ARM7TDMI. In *2000 International Symposium on Low Power Electronics and Design*, ISLPED '00, pages 185–190. ACM, 2000.

[43] Henri-Pierre Charles, Damien Couroussé, Victor Lomüller, Fernando A. Endo, and Rémy Gauguey. deGoal a Tool to Embed Dynamic Code Generators into Applications. In Albert Cohen, editor, *Compiler Construction*, volume 8409 of *Lecture Notes in Computer Science*, pages 107–112. Springer Berlin Heidelberg, 2014.

[44] Henri-Pierre Charles and Victor Lomüller. Is dynamic compilation possible for embedded systems ? In *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*, SCOPES 2015, pages 80–83, June 2015.

[45] Yang Chen, Shuangde Fang, Lieven Eeckhout, Olivier Temam, and Chengyong Wu. Iterative Optimization for the Data Center. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 49–60. ACM, 2012.

[46] Andrew A. Chien, Allan Snavely, and Mark Gahagan. 10x10: A General-purpose Architectural Approach to Heterogeneity and Energy Efficiency. *Procedia Computer Science*, 4(0):1987 – 1996, 2011. Proceedings of the International Conference on Computational Science, ICCS 2011.

[47] Derek Chiou, Dam Sunwoo, Joonsoo Kim, Nikhil A. Patil, William Reinhart, D. Eric Johnson, Jebediah Keefe, and Hari Angepat. FPGA-Accelerated Simulation Technologies (FAST): Fast, Full-System, Cycle-Accurate Simulators. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 249–261. IEEE Computer Society, 2007.

[48] Jeonghwan Choi, Chen-Yong Cher, Hubertus Franke, Hendrik Hamann, Alan Weger, and Pradip Bose. Thermal-aware Task Scheduling at the System Software Level. In *Proceedings of the 2007 International Symposium on Low Power Electronics and Design*, ISLPED '07, pages 213–218. ACM, 2007.

[49] Charles Consel, Luke Hornof, François Noël, Jacques Noyé, and Nicolae Volanschi. A Uniform Approach for Compile-Time and Run-Time Specialization. In *Selected Papers from the International Seminar on Partial Evaluation*, pages 54–72. Springer-Verlag, 1996.

[50] Charles Consel, Julia L. Lawall, and Anne-Françoise Le Meur. A tour of Tempo: a program specializer for the C language. *Science of Computer Programming*, 52(1-3):341–370, August 2004.

[51] Keith D. Cooper, Devika Subramanian, and Linda Torczon. Adaptive Optimizing Compilers for the 21st Century. *The Journal of Supercomputing*, 23(1):7–22, August 2002.

[52] Pablo de Oliveira Castro, Eric Petit, Asma Farjallah, and William Jalby. Adaptive sampling for performance characterization of application kernels. *Concurrency and Computation: Practice and Experience*, 25(17):2345–2362, 2013.

[53] Robert H. Dennard, Fritz H. Gaensslen, Hwa-Nien Yu, V. Leo Rideout, Ernest Bassous, and Andre R. Leblanc. Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions. *IEEE Journal of Solid-State Circuits*, SC-9:256–268, October 1974.

[54] Rajagopalan Desikan, Doug Burger, and Stephen W. Keckler. Measuring Experimental Error in Microprocessor Simulation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, ISCA '01, pages 266–277. ACM, 2001.

[55] Matthew DeVuyst, Ashish Venkat, and Dean M. Tullsen. Execution Migration in a Heterogeneous-ISA Chip Multiprocessor. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 261–272. ACM, 2012.

[56] Ronald G. Dreslinski, Michael Wieckowski, David Blaauw, Dennis Sylvester, and Trevor Mudge. Near-Threshold Computing: Reclaiming Moore's Law Through Energy Efficient Integrated Circuits. *Proceedings of the IEEE*, 98(2):253–266, February 2010.

[57] Tom Duff. Tom Duff on Duff's Device, 1983. `http://www.lysator.liu.se/c/duffs-device.html` [Accessed: 4 May 2015].

[58] M. Duranton, D. Black-Schaffer, K. De Bosschere, and J. Maebe. The HiPEAC Vision for Advanced Computing in Horizon 2020, March 2013. `https://www.hipeac.net/assets/public/publications/vision/hipeac-vision-2013.pdf` [Accessed: 7 July 2015].

[59] Marc Duranton, Koen De Bosschere, Albert Cohen, Jonas Maebe, and Harm Munk. HiPEAC Vision 2015. `https://www.hipeac.net/assets/public/publications/vision/hipeac-vision-2015_Dq0boL8.pdf` [Accessed: 7 July 2015].

[60] EETimes. Slideshow: Samsung cagey on smartphone SoC at ISSCC. `http://www.eetimes.com/document.asp?doc_id=1263082&page_number=2` [Accessed: 5 November 2014].

[61] Joel Emer, Pritpal Ahuja, Eric Borch, Artur Klauser, Chi-Keung Luk, Srilatha Manne, Shubhendu S. Mukherjee, Harish Patil, Steven Wallace, Nathan Binkert, Roger Espasa, and Toni Juan. Asim: A Performance Model Framework. *Computer*, 35(2):68–76, February 2002.

[62] Fernando A. Endo, Damien Couroussé, and Henri-Pierre Charles. Towards a dynamic code generator for run-time self-tuning kernels in embedded applications. To appear in the *4th International Workshop on Dynamic Compilation Everywhere*. January 2015.

[63] Fernando A. Endo, Damien Couroussé, and Henri-Pierre Charles. Micro-architectural simulation of in-order and out-of-order ARM microprocessors with gem5. In *2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*, SAMOS XIV, pages 266–273, July 2014.

[64] Fernando A. Endo, Damien Couroussé, and Henri-Pierre Charles. Micro-architectural Simulation of Embedded Core Heterogeneity with gem5 and McPAT. In *Proceedings of the 2015 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, RAPIDO '15, pages 7:1–7:6. ACM, 2015.

[65] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.

[66] Nicolas Fournel. *Estimation et optimisation de performances temporelles et énergétiques pour la conception de logiciels embarqués*. PhD thesis, École Normale Supérieure de Lyon, 2007.

[67] gem5.org. gem5-stable: Summary. Tag: stable_2012_06_28. `http://repo.gem5.org/gem5-stable` [Accessed: 4 May 2015].

[68] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. Annotation-Directed Run-Time Specialization in C. In *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '97, pages 163–178. ACM, 1997.

[69] Brian Grant, Matthai Philipose, Markus Mock, Craig Chambers, and Susan J. Eggers. An Evaluation of Staged Run-time Optimizations in DyC. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 293–304, New York, NY, USA, 1999. ACM.

[70] Mentor Graphics. Veloce Emulation Systems. `http://www.mentor.com/products/fv/emulation-systems/veloce` [Accessed: 22 May 2015].

[71] Peter Greenhalgh. Big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7. *ARM White paper*, 2011.

[72] Anthony Gutierrez, Joseph Pusdesris, Ronald G. Dreslinski, Trevor Mudge, Chander Sudanthi, Christopher D. Emmons, Mitchell Hayenga, and Nigel Paver. Sources of Error in Full-System Simulation. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 13–22, March 2014.

[73] Mary Hall, David Padua, and Keshav Pingali. Compiler Research: The Next 50 Years. *Communications of the ACM*, 52(2):60–67, February 2009.

[74] Hewlett-Packard Development Company, L.P. McPAT. `http://www.hpl.hp.com/research/mcpat/` [Accessed: 16 April 2015].

[75] Ming-yu Hsieh. A Scalable Simulation Framework for Evaluating Thermal Management Techniques and the Lifetime Reliability of Multithreaded Multicore Systems. In *Proceedings of the 2011 International Green Computing Conference and Workshops*, IGCC '11, pages 1–6. IEEE Computer Society, 2011.

[76] Chung-Hsing Hsu and Ulrich Kremer. The Design, Implementation, and Evaluation of a Compiler Algorithm for CPU Energy Reduction. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 38–48. ACM, 2003.

[77] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual Combinated Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C*, June 2013.

[78] Minhaj Ahmad Khan. Feedback-directed specialization of code. *Computer Languages, Systems & Structures*, 36(1):2 – 15, 2010.

[79] Jinpyo Kim, Wei-Chung Hsu, and Pen-Chung Yew. COBRA: An Adaptive Runtime Binary Optimization Framework for Multithreaded Applications. In *Proceedings of the 2007 International Conference on Parallel Processing*, ICPP '07, pages 25–. IEEE Computer Society, 2007.

[80] Thomas Kistler and Michael Franz. Continuous Program Optimization: Design and Evaluation. *IEEE Transactions on Computers*, 50(6):549–566, June 2001.

[81] Joonho Kong, Sung Woo Chung, and Kevin Skadron. Recent Thermal Management Techniques for Microprocessors. *ACM Computing Surveys*, 44(3):13:1–13:42, June 2012.

[82] Christos Kozyrakis. Advancing Computer Systems without Technology Progress. ISPASS Keynote, April 2013. `http://www.ispass.org/ispass2013/2013.advancingsystems.ispass_keynote.pdf` [Accessed: 3 June 2015].

[83] Arvind Krishnaswamy and Rajiv Gupta. Profile Guided Selection of ARM and Thumb Instructions. In *Proceedings of the joint conference on Languages, compilers and tools for embedded systems: software and compilers for embedded systems*, LCTES/SCOPES '02, pages 56–64. ACM, 2002.

[84] Rakesh Kumar, Keith Farkas, Norman P Jouppi, Partha Ranganathan, and Dean M. Tullsen. Processor Power Reduction Via Single-ISA Heterogeneous Multi-Core Architectures. *IEEE Computer Architecture Letters*, 2(1):2–2, January 2003.

[85] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proceedings of the 36th International Symposium on Microarchitecture*, MICRO-36'03, pages 81–. IEEE Computer Society, 2003.

[86] Travis Lanier. Exploring the Design of the Cortex-A15 Processor. `http://www.arm.com/files/pdf/at-exploring_the_design_of_the_cortex-a15.pdf` [Accessed: 4 May 2015].

[87] Peter Lee and Mark Leone. Optimizing ML with Run-time Code Generation. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, PLDI '96, pages 137–148. ACM, 1996.

[88] Daniel Leibholz and Rahul Razdan. The Alpha 21264: A 500 MHz Out-of-Order Execution Microprocessor. In *Proceedings of the 42nd IEEE International Computer Conference*, COMPCON '97, pages 28–. IEEE Computer Society, 1997.

[89] Ana Sonia Leon, Kenway W. Tam, Jinuk Luke Shin, David Weisner, and Francis Schumacher. A Power-Efficient High-Throughput 32-Thread SPARC Processor. *IEEE Journal of Solid-State Circuits*, 42(1):7–16, January 2007.

[90] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 469–480. ACM, 2009.

[91] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. The McPAT Framework for Multicore and Manycore Architectures: Simultaneously Modeling Power, Area, and Timing. *ACM Transactions on Architecture and Code Optimization*, 10(1):5:1–5:29, April 2013.

[92] Weiping Liao, Lei He, and Kevin M. Lepak. Temperature and Supply Voltage Aware Performance and Power Modeling at Microarchitecture Level. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(7):1042–1053, July 2005.

[93] Gabriel H. Loh, Samantika Subramaniam, and Yuejian Xie. Zesto: A Cycle-Level Simulator for Highly Detailed Microarchitecture Exploration. In *ISPASS 2009: IEEE International Symposium on Performance Analysis of Systems and Software*, pages 53–64. IEEE Computer Society, 2009.

[94] Victor Lomüller, Henri-Pierre Charles, Fernando Endo, and Wajdan Rekik. Low Overhead Runtime Code Specializations: A Case Study of the Impact on Speed, Energy and Memory. Presented at the *18th International Workshop on Compilers for Parallel Computing (CPC 2015)*, January 2015.

[95] Jiwei Lu, Howard Chen, Rao Fu, Wei-Chung Hsu, Bobbie Othmer, Pen-Chung Yew, and Dong-Yuan Chen. The Performance of Runtime Data Cache Prefetching in a Dynamic Optimization System. In *Proceedings of the 36th International Symposium on Microarchitecture*, MICRO-36 2003, pages 180–. IEEE Computer Society, 2003.

[96] Andrew Lukefahr, Shruti Padmanabha, Reetuparna Das, Faissal M. Sleiman, Ronald Dreslinski, Thomas F. Wenisch, and Scott Mahlke. Composite Cores: Pushing Heterogeneity Into a Core. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '12, pages 317–328. IEEE Computer Society, 2012.

[97] David R. Lutz. Fused Multiply-Add Microarchitecture Comprising Separate Early-Normalizing Multiply and Add Pipelines. In *20th IEEE Symposium on Computer Arithmetic (ARITH)*, pages 123–128, July 2011.

[98] Mahesh Mamidipaka and Nikil Dutt. eCACTI: An Enhanced Power Estimation Model for On-chip Caches. Technical report, Center for Embedded Computer Systems, University of California, Irvine, CA 92697, USA, September 2004.

[99] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *ACM SIGARCH Computer Architecture News*, 33(4):92–99, September 2005.

[100] Sanu K. Mathew, Mark A. Anders, Brad Bloechel, Trang Nguyen, Ram K. Krishnamurthy, and Shekhar Borkar. A 4-GHz 300-mW 64-bit Integer Execution ALU With Dual Supply Voltages in 90-nm CMOS. *IEEE Journal of Solid-State Circuits*, 40(1):44–51, January 2005.

[101] Andreas Merkel, Frank Bellosa, and Andreas Weissel. Event-Driven Thermal Management in SMP Systems. In *In Proceedings of the 2nd Workshop on Temperature-Aware Computer Systems (TACS'05)*, 2005.

[102] R. Merrit. ARM CTO: power surge could create 'dark silicon', October 2009. `http://www.eetimes.com/news/design/showArticle.jhtml?articleID=220900080` [Accessed: 17 June 2015].

[103] Markus Mock, Craig Chambers, and Susan J. Eggers. Calpa: A Tool for Automating Selective Dynamic Compilation. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 33, pages 291–302. ACM, 2000.

[104] Monolithic 3D Inc. The Dally-nVIDIA-Stanford Prescription for Exascale Computing, November 2011. `http://www.monolithic3d.com/blog/the-dally-nvidia-stanford-prescription-for-exascale-computing` [Accessed: 9 September 2015].

[105] Madhu Mutyam, Feihui Li, Vijaykrishnan Narayanan, Mahmut Kandemir, and Mary Jane Irwin. Compiler-Directed Thermal Management for VLIW Functional Units. In *Proceedings of the 2006 ACM SIGPLAN/SIGBED conference on Language, compilers, and tool support for embedded systems*, LCTES '06, pages 163–172, New York, NY, USA, 2006. ACM.

[106] Koichi Nose and Takayasu Sakurai. Analysis and Future Trend of Short-Circuit Power. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(9):1023–1030, September 2000.

[107] Tony Nowatzki, Jaikrishnan Menon, Chen-Han Ho, and Karthikeyan Sankaralingam. gem5, GPGPUSim, McPAT, GPUWattch, "Your favorite simulator here" Considered Harmful. Presented at the 11th Annual Workshop on Duplicating, Deconstructing and Debunking, June 2014.

[108] Dorit Nuzman, Revital Eres, Sergei Dyshel, Marcel Zalmanovici, and Jose Castanos. JIT Technology with C/C++: Feedback-Directed Dynamic Recompilation for Statically Compiled Languages. *ACM Transactions on Architecture and Code Optimization*, 10(4):59:1–59:25, December 2013.

[109] Subbarao Palacharla, Norman P. Jouppi, and J. E. Smith. Complexity-Effective Superscalar Processors. In *Proceedings of the 24th annual International Symposium on Computer Architecture*, 1997.

[110] Massoud Pedram and Qing Wu. Design Considerations for Battery-Powered Electronics. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, DAC '99, pages 861–866. ACM, 1999.

[111] A. Perais and A. Seznec. EOLE: Paving the Way for an Effective Implementation of Value Prediction. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 481–492, June 2014.

[112] Massimiliano Poletto, Wilson C. Hsieh, Dawson R. Engler, and M. Frans Kaashoek. 'C and tcc: A Language and Compiler for Dynamic Code Generation. *ACM Transactions on Programming Languages and Systems*, 21(2):324–369, March 1999.

[113] Michael D. Powell, Arijit Biswas, Joel S. Emer, Shubhendu S. Mukherjee, Basit R. Sheikh, and Shrirang Yardi. CAMP: A Technique to Estimate Per-Structure Power at Run-time using a Few Simple Parameters. In *IEEE 15th International Symposium on High Performance Computer Architecture, 2009. HPCA 2009*, pages 289–300. IEEE Computer Society, 2009.

[114] Sherief Reda and Abdullah N. Nowroz. Power Modeling and Characterization of Computing Devices: A Survey. *Foundations and Trends in Electronic Design Automation*, 6(2):121–216, February 2012.

[115] Ali Saidi and Andreas Hansson. Simulating Systems not Benchmarks, 2012. `http://gem5.org/dist/tutorials/hipeac2012/gem5_hipeac.pdf` [Accessed: 22 May 2015].

[116] Benjamin Carrión Schäfer and Taewhan Kim. Thermal-Aware Instruction Assignment for VLIW Processors. In *In Proceedings of the 11th Workshop on Interaction between Compilers and Computer Architectures (INTERACT-11)*, February 2007.

[117] Semiconductor Industries Association. *Model for Assessment of CMOS Technologies and Roadmaps (MASTAR)*, 2007.

[118] Eran Shifer and Shlomo Weiss. Low-Latency Adaptive Mode Transitions and Hierarchical Power Management in Asymmetric Clustered Cores. *ACM Transactions on Architecture and Code Optimization*, 10(3):10:1–10:25, September 2008.

[119] Kevin Skadron, Mircea R. Stan, Karthik Sankaranarayanan, Wei Huang, Sivakumar Velusamy, and David Tarjan. Temperature-Aware Microarchitecture: Modeling and Implementation. *ACM Transactions on Architecture and Code Optimization*, pages 94–125, March 2004.

[120] Gurindar S. Sohi. Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers. *IEEE Transactions on Computers*, 39(3):349–359, March 1990.

[121] Richard Strong. m5-mcpat-parser. `https://bitbucket.org/rickshin/m5-mcpat-parser` [Accessed: 5 November 2014].

[122] Ching-Long Su, Chi-Ying Tsui, and Alvin M. Despain. Low Power Architecture Design and Compilation Techniques for High-Performance Processors. In *Compcon Spring '94, Digest of Papers.*, pages 489–498, Feb 1994.

[123] M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt. Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures. *SIGARCH Computer Architecture News*, 37(1):253–264, March 2009.

[124] Dam Sunwoo, Gene Y. Wu, Nikhil A. Patil, and Derek Chiou. PrEsto: An FPGA-Accelerated Power Estimation Methodology for Complex Systems. In *2010 International Conference on Field Programmable Logic and Applications*, FPL '10, pages 310–317. IEEE Computer Society, 2010.

[125] Texas Instruments. *AM/DM37x Multimedia Device Silicon Revision 1.x Technical Reference Manual*, May 2010. Version R.

[126] The Tech Report. AMD's A4-5000 'Kabini' APU reviewed. `http://techreport.com/review/24856/amd-a4-5000-kabini-apu-reviewed/11` [Accessed: 5 November 2014].

[127] Shyamkumar Thoziyoor, Jung Ho Ahn, Matteo Monchiero, Jay B. Brockman, and Norman P. Jouppi. A Comprehensive Memory Modeling Tool and its Application to the Design and Analysis of Future Memory Hierarchies. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 51–62. IEEE Computer Society, 2008.

[128] Shyamkumar Thoziyoor, Naveen Muralimanohar, and Norman P. Jouppi. *CACTI 5.0*. Advanced Architecture Laboratory HP Laboratories, 2007.

[129] Ananta Tiwari and Jeffrey K. Hollingsworth. Online Adaptive Code Generation and Tuning. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 879–892. IEEE Computer Society, 2011.

[130] UC Berkeley Device Group. *BSIM 4 MOSFET Model*. Univ. California, Berkeley, CA, July 2002.

[131] Tomoaki Ukezono and Kiyofumi Tanaka. Reduction of Leakage Energy in Low Level Caches. In *Proceedings of the International Conference on Green Computing*, GREENCOMP '10, pages 537–544. IEEE Computer Society, 2010.

[132] University of Michigan and University of Colorado. DARPA Technical Report v2.7.fm. Technical report. `http://web.eecs.umich.edu/~panalyzer/pdfs/Sim-Panalyzer2.0_ReferenceManual.pdf` [Accessed: 7 February 2013].

[133] P. Unnikrishnan, G. Chen, M. Kandemir, and D. R. Mudgett. Dynamic Compilation for Energy Adaptation. In *Proceedings of the 2002 IEEE/ACM international Conference on Computer-aided Design*, ICCAD '02, pages 158–163. ACM, 2002.

[134] Vasanth Venkatachalam and Michael Franz. Power Reduction Techniques For Microprocessor Systems. *ACM Computing Surveys*, 37(3):195–237, September 2005.

[135] Michael J. Voss and Rudolf Eigenmann. ADAPT: Automated De-Coupled Adaptive Program Transformation. In *Proceedings of the Proceedings of the 2000 International Conference on Parallel Processing*, ICPP '00, pages 163–. IEEE Computer Society, 2000.

[136] Steven Wilton and Norman P. Jouppi. An enhanced access and cycle time model for on-chip caches. Technical report, DEC WRL, 1994.

[137] Qiang Wu, V.J. Reddi, Youfeng Wu, Jin Lee, Dan Connors, David Brooks, Margaret Martonosi, and Douglas W. Clark. A Dynamic Compilation Framework for Controlling Microprocessor Energy and Performance. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '05, pages 271–282. IEEE Computer Society, 2005.

[138] Sam (Likun) Xi, Hans Jacobson, Pradip Bose, Gu-Yeon Wei, and David Brooks. Quantifying Sources of Error in McPAT and Potential Impacts on Architectural Studies. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 577–589, Feb 2015.

[139] Yi-Ping You, Chingren Lee, and Jenq Kuen Lee. Compilers for Leakage Power Reduction. *ACM Transactions on Design Automation of Electronic Systems*, 11(1):147–164, January 2006.

[140] Matt T. Yourst. PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 0:23–34, 2007.

[141] W. Zhang, J. S. Hu, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Compiler-Directed Instruction Cache Leakage Optimization. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO-35, pages 208–218. IEEE Computer Society, 2002.

Personal bibliography

Conferences

Henri-Pierre Charles, Damien Couroussé, Victor Lomüller, Fernando A. Endo, and Rémy Gauguey. deGoal a Tool to Embed Dynamic Code Generators into Applications. In Albert Cohen, editor, *Compiler Construction*, volume 8409 of *Lecture Notes in Computer Science*, pages 107–112. Springer Berlin Heidelberg, 2014

Fernando A. Endo, Damien Couroussé, and Henri-Pierre Charles. Micro-architectural simulation of in-order and out-of-order ARM microprocessors with gem5. In *2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*, SAMOS XIV, pages 266–273, July 2014

Fernando A. Endo, Damien Couroussé, and Henri-Pierre Charles. Micro-architectural Simulation of Embedded Core Heterogeneity with gem5 and McPAT. In *Proceedings of the 2015 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, RAPIDO '15, pages 7:1–7:6. ACM, 2015

Fernando A. Endo, Damien Couroussé, and Henri-Pierre Charles. Towards a dynamic code generator for run-time self-tuning kernels in embedded applications. To appear in the *4th International Workshop on Dynamic Compilation Everywhere*. January 2015

Unpublished papers

Victor Lomüller, Henri-Pierre Charles, Fernando Endo, and Wajdan Rekik. Low Overhead Runtime Code Specializations: A Case Study of the Impact on Speed, Energy and Memory. Presented at the *18th International Workshop on Compilers for Parallel Computing (CPC 2015)*, January 2015

Glossary

ABB adaptive body biasing. 11

ALU arithmetic logic unit. 21, 31, 39, 44, 52, 97

BP branch predictor. 31, 42–44, 53, 124

BTB branch target buffer. 31, 33, 39, 41, 42, 44, 53, 97, 124, 125

CAM content-addressable memory. 19, 121

CMOS complementary metal-oxide-semiconductor. 9, 10

COMM-DRAM commodity DRAM. 20

compilette run-time kernel or function generator. 64, 65, 71, 80, 91, 98, 99

DBO dynamic binary optimization. 24, 25, 84

DIC dynamic instruction count. 105

DSCP desktop- and server-class processor. 87, 88

DSL domain specific language. 18, 24, 63

DSP digital signal processor. 14, 69

DVFS dynamic frequency and voltage scaling. 11–13

EXE execution or execute (unit or stage). 17, 18, 22, 31, 32, 39, 42–46, 49, 50, 52, 54–57, 71–73, 85, 105, 109, 120

FMAC floating-point multiply-accumulate. 52, 74, 80, 109

FPGA field-programmable gate array. 20

FPU floating-point unit. 21

FS Full-System. 18, 30, 47, 53

FU functional unit. 13, 14, 16, 19, 21, 31, 32, 39, 40, 43, 45, 50, 52–55, 61, 72, 73, 112, 113, 115

GPGPU general-purpose computing on graphics processing units. 115

Hamming distance number of different bits between two binary numbers. 13

HPC high-performance computing. 12

IC integrated circuit. 3, 4, 9, 10, 16

IFU instruction fetch unit. 39, 55

ILP instruction-level parallelism. 56, 76, 80, 82, 83, 102, 105, 106

IPC instructions per cycle. 17, 49, 122

IPS instructions per second. 18, 20

IQ instruction queue. 31, 35, 41, 42, 44, 53, 120–123

IR intermediate representation. 23, 25, 64

ISA instruction set architecture. 5, 7, 8, 11, 14–18, 29, 30, 35, 63, 64, 69–71, 73, 75, 83, 85, 87, 88, 92, 113, 114

ISS instruction set simulator. 16, 60

ITRS International Technology Roadmap for Semiconductors. 21, 40

JIT just-in-time (compiler). 23, 25, 63, 143

LP-DRAM logic process based DRAM. 19

LSQ load/store queue. 31, 33–35, 39, 43, 44, 53, 97

LSU load/store unit. 31, 33, 39, 53, 55, 73, 121

MMU memory management unit. 15, 39, 55

MOESI a cache coherency protocol that uses all possible states commonly used in other protocols. 18

MPI Message Passing Interface. 22

MPSoC multiprocessor system-on-chip. 14

MSHR miss status and handling register. 33, 42–44, 53, 97, 124

NoC network on chip. 16, 21, 39, 41

NOP no operation (instruction). 11, 71, 74

Pipeline back-end issue, execute, writeback and commit stages of a pipeline. 53, 97

Pipeline front-end fetch, decode, rename and dispatch stages of a pipeline. 33, 35, 40, 53, 97

PISA Portable Instruction Set Architecture. 16, 17

RAS return address stack. 31, 39, 44, 53, 97

RAT register alias table. 39

RISC reduced instruction set computing. 24, 63

ROB reorder buffer. 17, 31, 38, 42, 44, 46, 50, 53, 76, 97, 121, 122

RTC real-time clock. 12

RTL register-transfer level. 16, 20, 29, 30

SE System-call Emulation. 18, 30

SIMD single instruction, multiple data. 7, 15, 31–35, 39, 46, 53, 54, 56, 57, 64, 65, 69–73, 75, 79–83, 85, 89, 92, 93, 96–99, 101, 102, 105, 109, 112, 113, 121, 123

SISD single instruction, single data. 71, 72, 75, 80–83, 85, 99, 101, 102, 105, 109, 112, 113

SMT simultaneous multithreading. 11, 18, 20, 39, 120, 122, 123

SoC system on chip. 5, 12, 14, 16, 26, 37, 39, 40, 87, 88, 111, 114

SP stride prefetcher. 33, 53

TLB translation lookaside buffer. 18, 19, 39, 41, 42, 44, 53, 55, 97, 124, 125

WB write buffer. 33, 42–44, 53, 97, 124

Résumé étendu

Chapitre 1 : Introduction

Depuis la dernière décennie, la consommation énergétique élevée de processeurs haute performance limite l'amélioration de la performance attendue par la miniaturisation des transistors. Même si les transistors sont de plus en plus petits et nombreux à chaque nouvelle génération de circuits intégrés, leur consommation ne réduit presque plus, ce qui entraîne une croissance exponentielle de la dissipation d'une puce. Cet effet est appelé « power wall ». La Figure 1.1 illustre ce phénomène. En conséquence, il y a dix ans, l'amélioration de performance « gratuite » obtenue grâce à l'augmentation de la fréquence d'horloge a atteint sa limite et les concepteurs de processeurs ont dû changer du design monocœur aux multicœurs homogènes.

Aujourd'hui, les multicœurs homogènes dans les processeurs du type serveur font face à un autre problème : le « dark silicon » [102]. À cause de problèmes thermiques et de densité élevée de puissance, seulement une fraction des circuits intégrés peut être alimenté en courant simultanément, et cette fraction devient plus petite à chaque nouvelle génération. La Figure 1.2 illustre ce problème. On s'attend à ce que le dark silicon domine dans les circuits intégrés du type CPU entre 2016 et 2021 [65]. Des études suggèrent que des multi- ou manycœurs hétérogènes couplés à des accélérateurs sont une des solutions pour suivre l'amélioration de la performance attendue par la miniaturisation des transistors [32]. En 2020, les processeurs auront sans doute des centaines à des milliers de cœurs hétérogènes, probablement spécialisés pour différentes tâches [73].

Les cœurs embarqués haute performance ont déjà un degré considérable d'hétérogénéité, par exemple, l'architecture ARM définit des petits cœurs d'exécution dans l'ordre (en anglais, in-order cores) comme le Cortex-A5 et A7, des cœurs de taille moyenne comme les A8, A9, A12 et A17, jusqu'à des gros cœurs d'exécution dans le désordre (en anglais, out-of-order cores) comme l'A15, sans parler de quelques équivalents 64-bits comme les A53, A57 et A72, respectivement. Ces designs de base peuvent être synthétisés avec différentes technologies de transistors et avoir une variété de taille et de type de ressources (tampons dans le pipeline, niveaux et taille des caches, pour en nommer quelques-uns), et des cœurs complètement personnalisés existent aussi. Étant donné que les processeurs embarqués haute performance sont aussi soumis au power wall, et que le dark silicon bientôt changera complètement les architectures embarquées, on doit s'attendre à une complexité croissante du design de multi/manycœurs hétérogènes. Récemment, ARM a révélé son système multicœur hétérogène (le design big.LITTLE), où des applications peuvent être déplacées entre deux cœurs qui supportent le même jeu d'instructions, avec différents compromis d'énergie et de performance [71]. Malgré qu'il améliore l'efficacité énergétique sous faible exigence de performance, la solution d'hétérogénéité de cœurs intégré dans un système sur puce crée de nouveaux défis d'optimisation de performance. Par exemple, si une application est compilée

et optimisée pour un cœur cible A, lorsque l'application est ordonnancée dans un cœur B, sa performance pourra ne pas être aussi bonne que si elle avait été optimisée pour le cœur B, à cause de différences dans l'implémentation du pipeline.

L'écart de performance entre celle que les logiciels peuvent extraire du matériel et les performances maximales atteignables va augmenter dans l'ère des multi/manycœurs hétérogènes, et à long terme des approches dynamiques pourront être le seule moyen d'améliorer l'efficacité énergétique [32]. La Figure 1.3 illustre cet écart croissant. Le défi de programmation et de portabilité de performance dans ces systèmes complexes augmente encore plus avec les différents jeux d'instructions ou accélérateurs supportés par chaque cœur d'un processeur.

L'auto-tuning statique est utilisé pour améliorer la portabilité du code source et pour extraire des performances matérielles quasi-optimales, comparables aux codes optimisés manuellement. Cette approche a été utilisée avec succès dans les domaines de l'algèbre linéaire et du traitement de signal pour faire face à la complexité architecturale des processeurs modernes, en explorant un espace d'implémentation algorithmique [73]. Par contre, à chaque fois que le cœur cible ou les paramètres d'exécution changent, le code doit idéalement passer de nouveau par le processus d'auto-tuning dans le nouvel environnement, parce que l'auto-tuning statique produit souvent du code avec une mauvaise portabilité de performance entre différentes micro-architectures [3]. Lorsque l'environnement d'exécution n'est pas défini au moment de la compilation, l'auto-tuning à la volée est une option pour améliorer la portabilité de performance. Néanmoins, très peu de travaux ont traité l'auto-tuning à la volée [24] et faire de l'auto-tuning pendant l'exécution d'un programme est un grand défi [73], parce qu'évaluer la performance de plusieurs versions du programme en l'exécutant directement dans la plateforme cible a peu de chances de passer à l'échelle de grand nombre de cœurs ou de larges applications.

Cette thèse étudie l'adaptation de code machine à différentes micro-architectures, par l'auto-tuning à la volée. Une des tendances actuelles dans le domaine de la recherche liée aux compilateurs va vers la parallélisation de programme pour mieux utiliser les cœurs disponibles dans un processeur. La proposition de cette thèse est plutôt complémentaire à cette tendance. Autrement dit, cette thèse traite l'optimisation de code à la volée d'un programme (avec des multiples tâches – multi-threads – ou pas), avec pour but d'explorer des optimisations qui dépendent des données d'entrée et d'améliorer la portabilité de la performance entre différents processeurs et/ou entre divers cœurs avec différentes implémentations micro-architecturales dans un futur manycœur hétérogène. Il est possible que dans un tel futur, les processus pourront être ordonnancés dans n'importe quel cœur ou cluster de cœurs, selon les phases de l'application, l'endroit où se trouve les données à traiter et d'autres comportements dynamiques du système, comme l'adaptation aux ressources disponibles, entre autres optimisations de l'ordonnanceur du système d'exploitation.

Contribution de la thèse

La principale contribution de cette thèse est la méthodologie et la preuve de concept que l'auto-tuning de noyau de calcul de courte durée est possible dans les processeurs embarqués généralistes.

Cette thèse étudie les techniques d'adaptation de code à diverses configurations de pipeline. L'adaptabilité micro-architecturale de code de l'approche d'auto-tuning à la volée proposée est évaluée grâce à la simulation de la performance de plusieurs implémentations de cœurs. Un autre étude intéressante est la comparaison de l'énergie et la performance de designs de cœurs d'exécution dans l'ordre et dans le dé-

sordre, lorsqu'ils tournent les codes de référence ou les versions implémentées avec l'auto-tuning à la volée.

Génération de code et auto-tuning à la volée pour les systèmes embarqués

Pour but d'implémenter un cadriciel d'auto-tuning à la volée pour les processeurs embarqués, j'ai porté deGoal, un générateur de code à la volée, au jeu d'instructions ARM Thumb-2, y compris aux extensions flottante et vectorielle (SIMD), et je l'ai étendu avec un modèle configurable d'étage d'exécution et avec la sélection dynamique d'options de génération de code. J'ai effectué une validation préliminaire en évaluant huit configurations de noyau de calcul avec deGoal.

Je démontre l'approche d'auto-tuning à la volée en deux tests de performance (en anglais : benchmarks), en comparant des versions scalaires et vectorisées. Les tests de performance sont exécutés avec trois jeux de données d'entrées et sur plusieurs cœurs ARM réels et simulés.

Pour réduire l'espace de recherche, je présente une technique qui divise le processus d'auto-tuning en deux phases.

Cette thèse démontre que l'auto-tuning à la volée de noyau de calcul peut accélérer considérablement une application de calcul intensif, qui s'exécute pendant des centaines de millisecondes jusqu'à quelques secondes sur des cœurs embarqués, même en comparant avec un code vectorisé manuellement. Je montre aussi que les coûts de génération de code et d'exploration de l'espace à la volée sont très réduits, et que les performances obtenues avec l'auto-tuning à la volée sont très proches de celles obtenues avec une recherche extensive dans une configuration statique.

Simulation micro-architecturale de cœurs ARM

Pour évaluer la capacité d'adaptation micro-architecturale de code de l'approche d'auto-tuning à la volée proposée, j'ai développé un simulateur qui est capable d'estimer la performance et l'énergie consommée par des cœurs ARM hétérogènes. La plupart des études traitant l'asymétrie de cœurs ont employé d'autres jeux d'instructions bien que dans le contexte embarqué le jeu d'instructions ARM soit largement utilisé.

Cependant, au début de cette thèse, un cadriciel de simulation de la puissance et de la performance de cœurs ARM d'exécution dans l'ordre et dans le désordre n'existait pas. Les simulateurs gem5 et McPAT ont été choisis comme point de départ du cadriciel. Par conséquent, cette thèse contribue avec une description détaillée d'un cadriciel de simulation de cœurs ARM d'exécution dans l'ordre et dans le désordre. J'ai enrichi gem5 pour mieux modéliser des cœurs de la série ARM Cortex-A et McPAT pour mieux modéliser l'hétérogénéité de cœurs. Le cadriciel de simulation a été validé en deux phases : les estimation de performance ont été comparées avec les résultats mesurés dans des plateformes ARM réelles, et les estimations énergétiques ont été validées d'une façon relative, en comparant la simulation du compromis d'énergie et de performance de CPUs big.LITTLE.

Cette thèse démontre que gem5 peut simuler des cœurs ARM avec des erreurs de timing considérablement plus réduites que d'autres simulateurs similaires, et que d'une simple extension du modèle de McPAT résulte une estimations de surface précise et d'estimations de compromis d'énergie/performance

acceptables de CPUs big.LITTLE, lorsque McPAT est couplé à gem5.

Chapitre 2 : État de l'art

Ce chapitre synthétise l'état de l'art des sujets suivants : sources de consommation énergétique dans les circuits intégrés, techniques de réduction intégrées dans les compilateurs, simulateurs micro-architecturaux de processeurs embarqués et optimisations de code à la volée.

Sources de consommation énergétique dans les circuits intégrés

La dissipation de circuits CMOS (complementary metal-oxide-semiconductor) peut être divisée en deux parties : la dissipation statique et dynamique. La partie statique correspond à la puissance toujours dissipée lorsque les transistors sont allumés, même s'il n'y a pas d'activité dans le circuit. D'autre part, la puissance dynamique correspond à la dissipation produite à chaque fois que l'état des transistors changent entre un et zéro logique. Les équations 2.1 et 2.2 représentent la principale source de dissipation statique et la puissance dynamique, respectivement.

Techniques de réduction intégrées dans les compilateurs

La consommation énergétique peut être réduite par logiciel dans plusieurs couches : pilotes, systèmes d'exploitation, librairies, compilateurs et applications. Cette thèse s'intéresse aux approches dynamiques intégrées dans les compilateurs. Une fois que la compilation dynamique a historiquement traité l'amélioration de la performance, cette section présente aussi des techniques utilisées dans les compilateurs statiques et des techniques dynamiques non nécessairement intégrées dans un compilateur.

À partir de l'équation 2.2, un logiciel peut réduire la puissance dynamique en agissant sur : la capacité électrique active du circuit, le nombre de commutations des transistors, la fréquence de l'horloge et la tension de la source. Selon l'équation 2.1, la puissance statique peut être réduite en agissant sur la tension d'alimentation et les paramètres du circuit, en outre éteindre et refroidir les circuits.

Les techniques de clock et power gating permettent respectivement de réduire les commutations dues à l'horloge ou de complètement éteindre un composant inactif. Un technique moins agressive que le power gating est la mise en sommeil d'un circuit, en réduisant sa tension pendant des périodes inactives. L'intégration de ces techniques dans les compilateurs statiques a déjà été étudiée [21,139,141], et certains travaux ont proposé de les utiliser à la volée [131, 133].

Le DVFS (dynamic frequency and voltage scaling) agit à la fois sur la tension d'alimentation et la fréquence du circuit. Des méthodologies ont été proposées pour faire du DVFS statiquement [76] ou dynamiquement [137] avec l'aide d'un compilateur.

L'ordonnancement des instructions de façon à minimiser les commutations dans les circuits du processeur est aussi une technique qui pourrait être employée dans les compilateurs [42, 51].

Un compilateur peut aussi choisir des instructions (strength reduction ou pattern matching), voire un

jeu d'instructions entier ou des unités de calcul pour réduire la consommation. Les bénéfices énergétiques de multicœurs hétérogènes ont été démontrés par les travaux de Kumar et al. [84, 85]. Malgré que dans un système big.LITTLE l'ordonnancement des processus entre les cœurs hétérogènes est décidé par le système d'exploitation ou par un contrôleur matériel, les compilateurs peuvent aider à assurer l'exécution correcte de certaines applications [118, 123]. Pour les multicœurs qui n'ont pas de compatibilité de jeu d'instructions, le support du compilateurs pourrait être nécessaire pour réduire le coût en performance de la migration d'une application [55].

Simulation micro-architecturale de processeurs embarqués

Plusieurs niveaux d'abstraction existent pour la simulation d'un processeur. Le niveau de transistors est le plus détaillé, mais à cause du temps nécessaire pour les simuler, ce niveau n'est utilisé que pour calibrer des modèles plus abstraits. Au niveau des portes logiques et du RTL (register-transfer level), la vitesse de simulation est typiquement entre 1 Hz et 100 kHz [124]. Ces niveaux sont adaptés pour la simulation de petits processeurs ou petits morceaux de code. Au niveau micro-architectural, les composants du pipeline tel que le banc de registres et les unités fonctionnelles sont modélisés, d'autre part, au niveau architectural, le comportement des composants d'un processeur comme les cœurs et les caches sont modélisés. Le niveau d'instructions ne simule que le comportement d'un jeu d'instructions, aucun détail matériel du processeur n'étant simulé. Cette thèse s'intéresse à l'étude de l'hétérogénéité de cœurs embarqués, étant mieux représenté par les simulateurs micro-architecturaux.

Deux simulateurs micro-architecturaux très connus qui supportent le jeu d'instructions ARM sont SimpleScalar [35, 36] et gem5 [29].

SimpleScalar analyse dynamiquement la trace d'exécution des instructions pour dériver les statistiques d'utilisation des composants du pipeline. Bien que cette technique soit rapide, elle ne peut pas précisément simuler un multicœur. Sa première version supportait l'architecture PISA (Portable Instruction Set Architecture), une implémentation modifiée de MIPS. À partir de sa version 3.0, SimpleScalar a commencé à supporter les jeux d'instructions Alpha AXP, PowerPC et ARMv4. Plusieurs modèles de simulation existent, le plus connu étant le `sim-outorder`, qui simule un processeur d'exécution dans le désordre.

gem5 propose une simulation très précise du pipeline, en vérifiant que toutes les dépendances soient résolues avant l'exécution d'une instruction. Les jeux d'instructions les plus supportés sont Alpha, ARM et x86, lesquels peuvent amorcer le noyau Linux dans un environnement de simulation d'un système complet. Deux modèles micro-architecturaux de CPU existent : InOrder et O3, qui simulent respectivement des pipelines d'exécution dans l'ordre et dans le désordre. Pour ARM, seulement le modèle O3 fonctionne. Mais récemment, un modèle appelé Minor a été introduit, permettant la simulation d'un pipeline d'exécution dans l'ordre avec quatre étages[1].

Pour l'estimation énergétique, plusieurs outils existent, la plupart étant développés autour de SimpleScalar. Wattch [33] a été un des premiers simulateurs énergétiques au niveau micro-architectural. D'abord, les coûts énergétiques d'accès aux composants sont estimés avec des modèles analytiques ou empiriques. Ensuite, l'énergie consommée par le composant est obtenue avec la multiplication de son coût énergétique par le nombre d'accès estimés par SimpleScalar. Un des défauts de Wattch est le modelage autour de la technologie 0,8 µm, qui ne peut plus bien modéliser la consommation de pro-

[1] `http://www.mail-archive.com/gem5-dev%40gem5.org/msg11667.html` [Accédé : 17 April 2015]

cesseurs modernes. PTscalar [92] est aussi une extension de SimpleScalar, mais qui prend en compte l'influence de la température sur la puissance statique. Sim-Panalyzer [132] est le portage du modèle `sim-outorder` pour le processeur StrongARM SA1100, et intègre des modèles d'estimation énergétiques.

Des modèles complètement empiriques d'estimation de la consommation existent aussi. CAMP (*Common Activity-based Model for Power*) [113] et PrEsto (*PoweR ESTimatOr*) [124] utilisent des modèles de régression linéaire calibrés avec des simulateurs RTL pour estimer rapidement la consommation d'un processeur.

McPAT (*Multi-core Power Area and Timing*) [90,91] est un cadriciel de modelage qui estime ensemble la puissance, le timing et la surface de multi- et manycœurs. Comme Wattch, McPAT s'en sert de modèles analytiques pour les composants réguliers et empiriques pour les plus complexes. La technologie des transistors qui va jusqu'à 16 nm est basée sur l'ITRS (International Technology Roadmap for Semi-conductors).

Des outils spécifiques modélisent certains composants du processeurs, comme CACTI-D [127] pour les caches et DRAMs. D'autres outils, comme SST (Structural Simulation Toolkit) [75], regroupent divers outils de simulation, par exemple pour estimer la performance, la puissance, la température, la fiabilité, entre outres caractéristiques des processeurs.

Optimisations de code à la volée

Cette section résume l'état de l'art des techniques d'optimisation de code par la spécialisation et l'auto-tuning à la volée. Cette thèse se concentre sur les techniques à la volée, dont la performance est comparable à celle des codes compilés statiquement. Donc, les compilateurs JIT (just-in-time) qui accélèrent les langages interprétés ne sont pas présentés.

La spécialisation de code à la volée est implémentée par deux techniques principales : la génération de code et l'insertion d'instructions dans un patron de code pré-compilé.

Les générateurs de code à la volée peuvent générer un code spécifique pour une tâche ou une plateforme donnée. Fabius [87] a été proposé pour accélérer des programmes ML, grâce à la spécialisation de code à la volée. 'C [112] et tcc implémentent un sur-ensemble ANSI C et un compilateur pour générer du code machine à partir d'un langage haut-niveau. Deux caractères spéciaux permettent d'indiquer le début d'une expression compilée dynamiquement et d'affecter dynamiquement des valeurs à une variable. Avec un but similaire, TaskGraph [26] définit un sous-langage semblable au C pour la génération de code à la volée. Différemment de 'C, TaskGraph utilise n'importe quel compilateur standard C pour générer dynamiquement le code, ce qui le rend très portable, mais aussi très lent comparé aux outils de génération de code à la volée dédiés. deGoal [43] implémente un langage dédié bas-niveau pour la génération à la volée de noyaux de calcul. La génération de code peut être guidée et paramétrée en mélangeant le langage deGoal avec du code C standard.

Les approches basées sur les patrons de code pré-compilés ont pour but de réduire au maximum le temps de génération d'instructions à la volée. Tempo [49, 50], DyC [68, 69] sont des outils de spécialisation de code basés sur les annotations. Un programmeur doit identifier et annoter sur son code les opportunités de spécialisation dépendantes de données connues à l'exécution. Khan [78] a proposé une méthode de spécialisation orientée par la rétroaction. Non seulement les valeurs qui sont constantes

pendant l'exécution, mais aussi celles qui restent constantes pour une bonne période de temps sont candidates à la spécialisation à la volée. Un cache de codes spécialisés permet de réduire la surcharge dynamique.

Les systèmes d'optimisation dynamique de code binaire (DBO – dynamic binary optimization) sont capables d'instrumenter le code d'une application en exécution et générer des optimisations opportunistes dans son code machine. Dynamo [23] a été un des premiers systèmes de DBO, implémenté dans un machine HP PA-8000. DynamoRIO [34] est la version IA-32 de Dynamo. Des optimisations telles que la suppression de code mort, le remplacement d'instructions et de branchements coûteux et l'extension inline sont appliqués sur le binaire à la volée. ADORE [95] et COBRA [79] sont des systèmes de DBO pour les processeurs Itanium 2, qui traitent le problème de pré-chargement de données à la volée.

La recompilation à la volée de codes compilés statiquement est un sujet assez récent. Le but est de chercher à la volée des versions de fonction plus performantes, comme dans les travaux de Kistler and Franz [80]. Nuzman et al. ont proposé une technologie JIT pour le C/C++ [108]. Pour éviter au maximum la dégradation de la performance, la représentation intermédiaire du code est livré avec le code compilé, et pendant l'exécution du programme un compilateur JIT identifie et recompile les fonctions les plus utilisées.

L'auto-tuning à la volée a été traité dans les systèmes complexes tels que les processeurs du type serveur. Les approches existantes peuvent trouver automatiquement des meilleures options de compilation d'une application [45], ou explorer un espace d'implémentations possibles d'un code [4, 129, 135]. Par contre, pour payer les surcharges dynamiques de recompilation et entraînement, l'auto-tuning à la volée dans ces systèmes existants a besoin de plusieurs minutes voire des heures afin de surpasser la compilation statique.

Conclusion

L'étude de l'état de l'art des sources de dissipation et des techniques de réduction de la consommation intégrées dans les compilateurs permet de comprendre quel rôle un générateur de code à la volée peut jouer pour réduire la consommation d'un processeur embarqué. L'état de l'art de techniques d'optimisation de code à la volée montre qu'il n'existe pas de travaux qui traitent l'auto-tuning à la volée dans les application de courte durée qui s'exécutent dans un processeur embarqué. C'est pour cela que cette thèse propose une méthodologie et montre la preuve de concept d'un système d'auto-tuning à la volée pour les systèmes embarqués.

Chapitre 3 : Simulation micro-architecturale de processeurs ARM

Pour étudier l'hétérogénéité de cœurs embarqués, j'ai implémenté un cadriciel de simulation de la performance et la consommation de cœus ARM. La plupart des études sur l'hétérogénéité de cœurs ont été faits avec des simulateurs x86 ou Alpha, mais le jeu d'instructions ARM est beaucoup plus pertinent dans les systèmes embarqués. De plus, un simulateur ARM permet d'évaluer des idées dans un vrai environnement logiciel embarqué et avec une chaîne de compilation à jour.

gem5

Cette section détaille le simulateur gem5, les améliorations que j'ai apportées et le modèle approximatif d'exécution dans l'ordre que j'ai implémenté. Au moment du développement de ce modèle, gem5 pour ARM ne supportait pas la simulation d'un pipeline d'exécution dans l'ordre, comme montre la chronologie dans la Figure 3.1. La version gem5 prise comme référence est la stable de juin 2012 [67].

La configuration de base pour la simulation micro-architecturale s'appelle `arm_detailed`, qui simule un multicœur homogène avec des caches L1 privés et un L2 partagé (Figure 3.2).

J'ai développé quatre groupes d'améliorations dans gem5 : le premier pour mieux fournir des statistiques à McPAT; le deuxième pour mieux simuler les petits cœurs ARM; le troisième pour corriger les configurations de l'extension NEON; et finalement pour mieux simuler les instructions de load/store de multiples registres.

Finalement, j'explique pourquoi et décris comment j'ai modifié le modèle O3 pour qu'il se comporte approximativement comme un pipeline d'exécution dans l'ordre. Les trois étapes sont : l'imposition de l'issue dans l'ordre, l'annulation du renommage de registres et la correcte configuration des structures d'exécution dans le désordre restantes.

McPAT

McPAT modélise analytiquement la plupart des composants de processeurs d'exécution dans l'ordre et dans le désordre, haute performance ou embarqués. Ses modèles estiment ensemble la puissance, le timing et la surface des composants, sauf des unités fonctionnelles, qui ont un modèle empirique. Pour ce faire, son interface d'entrée reçoit un fichier XML avec les paramètres du système, à partir desquels McPAT bâtit un modèle interne de puce. Les statistiques d'utilisation des composants et la température de la puce, aussi renseignées dans le fichier XML, servent à estimer l'énergie dynamique et la puissance statique de chaque composant. La puissance de crête et la surface des composants sont aussi estimées.

Les modèles de pipeline dans McPAT ont été basés sur les architectures d'Intel, d'Alpha et de SPARC, mais lorsque le drapeau « Embedded » est activé, McPAT modélise des cœurs embarqués avec une unité VFP/NEON basée sur celle du Cortex-A9.

Dans McPAT la modélisation des unités fonctionnelles ne prend en compte que la technologie des transistors, c'est-à-dire un petit cœur A5 dont la taille de l'unité NEON est de 0,15 mm^2 et un A9 dont l'unité NEON a une surface de 0,98 mm^2, dans McPAT tous les deux auront une unité NEON de 0,97 mm^2 (avec une technologie de 40 nm). Cet exemple justifie le besoin d'un meilleur modèle d'unités fonctionnelles dans McPAT. Donc, à partir d'études d'échelonnement de la surface de cœurs Intel [118], j'ai proposé une méthode d'estimation de la surface et du coût énergétique d'accès aux unités fonctionnelles en fonction de la largeur du pipeline et des latences des instructions.

Conversion de paramètres et statistiques de gem5 vers McPAT

gem5 et McPAT ont des modèles génériques de pipeline. Afin de m'assurer de la bonne configuration de McPAT à partir des informations générées par gem5, j'ai étudié les modèles de pipeline des deux

simulateurs. Les Tableaux A.1 et A.2 présentent les équivalence de paramètres et statistiques des modèles de cœur. Les Tableaux A.3 et A.4 présentent celles des prédicteurs de branchements, TLBs (translation lookaside buffers) et caches L1 et L2.

Validation de performance

Pour but de valider le modèle approximatif d'exécution dans l'ordre et de déterminer la précision de timing du modèle O3, je les ai configurés comme un Cortex-A8 et un A9 pour les comparer aux processeurs réels. Ces deux cœurs ont été choisis parce que ARM en fournit une documentation détaillée comprenant entre autre la latence des instructions.

La validation a été faite en comparant le temps d'exécution de 10 tests de performance PARSEC 3.0, mesurés dans une carte BeagleBoard-xM et une Snowball SKY-S9500-ULP-CXX, et simulés avec gem5. Le modèle O3 configuré comme un A9 a été validé en exécutant les tests avec un ou deux fils d'exécution (en anglais, threads).

En moyenne, l'erreur absolue par rapport à l'A8 est de 8,0 %, et de 7,4 % par rapport à l'A9. J'ai proposé une amélioration du modelage de l'étage d'exécution dans gem5 pour l'A8 qui a réduit l'erreur absolue moyenne à 6,8 %.

Validation de la surface et énergie/performance relative

Pour valider la surface et les estimations relatives d'énergie et de performance, j'ai configuré mon simulateur comme les CPUs d'un système big.LITTLE, basé sur la carte ODROID-XU3.

J'ai aussi décidé de valider la surface estimée par mon simulateur, parce que les coûts énergétiques et la puissance statique des composants sont proportionnels à leurs surfaces. De plus les données sur la surface et le floorplan du A7 ont été publiés. Donc, les comparaisons en surface représentent une validation partielle des modèles énergétiques.

La validation des surfaces estimées a donné de bons résultats : les erreurs sont de 3,6 % et de 1,4 % pour le cœur et cluster A15; de 0,0 et de 13 % pour le cœur et cluster A7, respectivement. La surface de cinq structures principales du pipeline du A7 ont été comparées avec son floorplan [19] en 28 nm. Les erreurs varient de seulement 3,6 % jusqu'à 60 % pour la TLB, probablement car McPAT ne modélise pas un TLB de second niveau. Par contre, si l'on pondère l'erreur avec la surface de la structure, l'unité load/store est celle qui contribue le plus à l'erreur de la surface du cœur avec seulement 5,3 %.

Pour la validation des estimations relatives d'énergie et de performance, 11 tests de performance ont été simulés (Dhrystone 2.1 plus les 10 tests de PARSEC 3.0) avec un seul cœur actif dans chaque cluster. Les résultats de la simulation de l'efficacité énergétique du A7 et de l'accélération apportée par l'A15 sont à 6 % près pour le test Dhrystone 2.1 (en supposant un environnement d'exécution similaire à celui publié par ARM [71]). Globalement, les résultats de simulation montrent que l'A7 consomme 4,1 fois moins d'énergie, mais que l'A15 est 1,5 fois plus rapide, autant qu'il est généralement dit que l'A7 est entre 3 et 4 fois plus efficace en énergie et que l'A15 est entre 2 et 3 fois plus rapide. Par contre, les tests simulés ont été compilés avec une option d'optimisation très élevée (gcc -O3), différemment des applications peu ou modérément optimisées que l'on trouve dans la vie réelle et qu'un pipeline

d'exécution dans le désordre comme l'A15 peut accélérer.

Exemple d'exploration architecturale/micro-architecturale

Dans cette section, je présente un exemple d'exploration architecturale/micro-architecturale. Le Cortex-A8 ne supporte pas des implémentations multicœurs [8, Tableau 2-3], mais avec le simulateur développé, il est possible de simuler un processeur avec un dual cœur Cortex-A8. Le Tableau 3.15 et la Figure 3.14 montrent les performance d'un tel processeur hypothétique comparé au mono A8 (modèle gem5) et au dual A9 (modèle réel). Par contre, l'A9 a une unité flottante pipelinée, autant que l'A8 non. Cela explique pourquoi le dual A9 est 2,5 fois plus rapide que le dual A8 en exécutant les tests flottants. En simulant la même unité flottante du A9 dans l'A8, l'écart en performance des tests flottants diminue à seulement 1,24.

Portée et limitations

N'importe quel simulateur a une portée de validité et des limitations. Des travaux précédents ont déjà discuté des erreurs d'abstraction, de modelage et de spécification dans gem5 [37, 72, 107] et dans McPAT [138]. Différemment, cette section discute la portée et les limitations de mon cadriciel de simulation.

- Un simulateur micro-architectural basé sur modèles analytiques ne doit pas être utilisé pour estimer la consommation de processeurs existants, ils sont plutôt adaptés à la simulation de nouvelles idées ou de configurations hypothétiques de processeur. À moins que la valeur cherchée n'est pas mesurable, mesurer directement dans le matériel est la meilleure option. De plus, les simulateurs de jeu d'instructions calibrés sont plus rapides voire plus précis.
- gem5 et McPAT ont des modèles génériques de pipeline et donc ils ne simulent pas le vrai comportement de cœurs ARM. À l'échelle du cycle de l'horloge, les comportements du simulateur et des modèles réels peuvent être très différents.
- Le modèle d'exécution dans l'ordre proposé n'est pas cycle-accurate. Des études micro-architecturales avec ce modèle doivent être faits à bon escient.
- Il n'est pas encore possible la simulation d'un multicœur hétérogène, puisque le modèle d'exécution dans l'ordre proposé n'est pas encore intégré comme un nouveau modèle. À noter que les versions récentes de gem5 sans doute supportent déjà une telle configuration.

Conclusion

Dans ce chapitre, j'ai présenté un cadriciel de simulation micro-architecturale basée sur gem5 et McPAT. J'ai non seulement amélioré certains aspects de modelage dans les deux outils, mais aussi proposé un modèle approximatif d'exécution dans l'ordre. Une description détaillée sur la conversion de paramètres et de statistiques de gem5 vers McPAT a été présentée. Le cadriciel permet d'estimer l'énergie et la performance relatives de cœurs ARM réels et hypothétiques, comme j'ai démontré par plusieurs expérimentations et par un exemple d'exploration architecturale et micro-architecturale.

Le simulateur présenté dans ce chapitre m'a permis de simuler diverses configurations de pipeline, utilisées pour évaluer la capacité du système d'auto-tuning à la volée proposé à adapter du code à différentes micro-architectures.

Chapitre 4 : Génération de code à la volée

Ce chapitre décrit deGoal, un générateur de code à la volée conçu surtout pour les systèmes embarqués. Cet outil est développé au CEA dans le laboratoire LIALP, où cette thèse a été préparée.

Différemment des JITs et des compilateurs dynamiques, deGoal a été conçu en considérant les contraintes d'énergie, de mémoire et de puissance de calcul des systèmes embarqués.

Mon objectif pendant cette thèse a été de développer deGoal pour ARM et de démontrer que l'auto-tuning à la volée est faisable dans un processeur embarqué qui exécute des petites charges de travail.

deGoal : un outil pour embarquer des générateurs de code dans une application

deGoal implémente un langage dédié pour la génération de code à la volée de noyaux de calcul. Il défini un langage pseudo-assembleur semblable à un jeu d'instructions RISC (reduced instruction set computing), qui peut être mélangé avec du code C standard. Les instructions machines sont seulement générés par les instructions deGoal, autant que la gestion des variables et les décisions de génération de code sont implémentées par pseudo-instructions deGoal, optionnellement mélangées avec un code C. La caractéristique dynamique du langage vient du fait que les informations connues à la volée peuvent guider la génération de code machine, et aider à générer du code très optimisé.

Étant donné qu'aucun langage haut-niveau ni aucune représentation intermédiaire n'est traité à la volée, dans un processeur ARM deGoal génère du code quatre ordres de magnitude plus rapide que l'engin JIT dans LLVM avec l'option `-O3` [44, 94]

Une brève description du langage avec des exemples est présentée dans cette section du manuscrit.

Contributions de la thèse : nouvelles fonctionnalités et portage pour ARM

Pendant cette thèse, j'ai porté deGoal pour le jeu d'instructions ARM Thumb-2 (le jeu basique entier et les jeux pour les extensions VFP et NEON) et j'ai créé des nouvelles fonctionnalités nécessaires pour tester deGoal avec des tests de performance et pour l'utiliser dans un système d'auto-tuning à la volée.

L'architecture de deGoal est divisée en deux parties principales : le frontal (front-end) et l'arrière plan (back-end). Le frontal constitue la partie indépendante d'architecture et est responsable pour la conversion source-à-source du langage deGoal vers les appels aux fonctions de l'arrière plan. À son tour, l'arrière plan est une librairie peaufinée écrite en C, étant responsable par la gestion de la pile, l'allocation, sélection, réordonnancement et génération du code machine de chaque architecture supportée par deGoal. Pour ARM, cette partie correspond à environ 15 mille lignes de code source, le frontal quant à lui est composé d'environs 4,5 mille lignes de code. Le Tableau 4.1 détaille ces chiffres.

Les trois principales contributions de la thèse à deGoal ont été : le portage au jeu d'instructions ARM Thumb-2; l'extension de l'ordonnanceur d'instructions pour supporter les instructions flottantes/vectorielles et pour supporter un modèle d'étage d'exécution paramétrable; et la réorganisation de l'architecture de deGoal pour supporter la configuration de la plateforme cible et des options de génération de code de façon statique ou dynamique.

La configuration statique ou dynamique de deGoal permet soit de calibrer statiquement deGoal pour une plateforme bien définie, soit de choisir les options de génération de code dynamiquement si la plateforme n'est pas connue au moment de la compilation ou pour faire de l'auto-tuning à la volée.

Analyse de performance

Dans le but d'évaluer la qualité du code machine produit, j'ai effectué une validation préliminaire de deGoal pour ARM.

J'ai implémenté avec deGoal quatre versions de noyaux de calcul et je les ai comparés à leurs versions de référence scalaire (SISD) et vectorielle (SIMD) des tests de performance Streamcluster et VIPS des suites PARSEC 3.0 et PARVEC. Les comparaisons ont été menées dans la BeagleBoard-xM, qui embarque un Cortex-A8 (exécution dans l'ordre) et dans la Snowball qui a un dual Cortex-A9 (exécution dans le désordre).

Grâce à des optimisations comme le déroulage de boucle et l'augmentation du parallélisme d'instructions, deGoal a obtenu des accélérations moyennes de 1,22 et de 1,05 par rapport aux versions SISD et SIMD respectivement dans la BeagleBoard-xM, et a presque eu la même performance moyenne dans la Snowball. L'explication vient du fait qu'avec deGoal il est possible de générer un code plus compact et efficace surtout pour un pipeline d'exécution dans l'ordre.

Avec les mêmes noyaux, j'ai réalisé des expérimentations de spécialisation de code à la volée. Grâce au déroulage complet de boucles et l'utilisation d'autant de registres que possible, deGoal a obtenu des accélérations moyennes de 1,41 et de 1,23 par rapport aux codes SISD, et de 1,09 et de 1,06 par rapport aux codes SIMD de référence, dans la BeagleBoard-xM et la Snowball respectivement. De nouveau, nous observons qu'une optimisation, dans ce cas la spécialisation de code, bénéficie plus à un processeur d'exécution dans l'ordre, puisque l'exécution dans le désordre du A9 permet de dérouler des petites boucles des codes de référence et donc d'imiter partiellement le déroulage de boucle par logiciel.

Finalement, dans cette section j'explique que deGoal pourrait être utilisé pour faire de l'auto-tuning à la volée, surtout parce que j'ai mesuré des surcharges de génération de code à la volée inférieures à 0,1 % du temps d'exécution des noyaux, qui durent entre quelques secondes à quelques minutes. Je discute les possibilités d'auto-tuning, qui incluent la génération d'instructions SISD ou SIMD, dont la performance dépend du code et du pipeline cible, et des différentes stratégies de déroulage de boucle.

Portée et limitations

Les limitations de deGoal et la portée de mes expérimentations sont discutés dans cette section.

deGoal est adapté pour les noyaux de calcul pré-identifiés, dont les tailles doivent être relativement

petites pour être codés avec un langage bas niveau. Par contre, le code d'un noyau ne doit pas être trop petit, parce que j'ai observé des dégradations de performance importantes lorsqu'un petit noyau de calcul n'est pas inliné. Cela dit, deGoal est adapté pour l'implémentation d'optimisations à la volée du code de librairies dynamiques.

Finalement, je considère que l'évaluation préliminaire de performance n'est pas suffisante pour en tirer des conclusions généralisables, et de plus, les noyaux de calcul devrait être implémentés avec deGoal par un programmeur qui ne connais par les détails internes de l'outil.

Conclusion

Ce chapitre a présenté deGoal, un outil de génération de code à la volée. Cette thèse s'intéresse aux optimisations de code pour améliorer la performance et optimiser la consommation de processeurs embarqués, donc j'ai porté deGoal au jeu d'instructions ARM Thumb-2 et j'ai créé des nouvelles fonctionnalités pour supporter l'auto-tuning à la volée.

Une validation préliminaire a donné de bons résultats sur deux plateformes ARM. Les analyses des performances obtenues et des surcharges dynamiques négligeables m'ont permis d'argumenter que deGoal peut être employé dans un outil d'auto-tuning à la volée pour les systèmes embarqués.

Chapitre 5 : Auto-tuning à la volée pour les systèmes embarqués

Les processeurs embarqués haut performance généralistes évoluent avec une croissante complexité sans précédente. Dans le passé, les applications embarquées étaient compilées pour une seule architecture cible. Aujourd'hui, une application ne s'exécute pas forcément sur une seule cible, un binaire compilé peut s'exécuter sur des processeurs de différents concepteurs et voire même dans différents cœurs dans un système sur puce.

Les techniques d'optimisations itératives et d'auto-tuning sont utilisées pour traiter la complexité de processeurs du type serveur. Ces techniques peuvent trouver des implémentations de code machine quasi-optimales. Par contre, pour cela, elles ont besoin de beaucoup de temps pour explorer un espace d'implémentations possibles d'un algorithme. Les techniques existantes d'auto-tuning à la volée ont besoin de plusieurs minutes voire des heures pour payer les coûts de recompilation et d'entraînement à la volée, étant surtout adaptés pour les charges de travail du type scientifique ou data center.

Différemment, les applications embarqués déployées dans les processeurs généralistes s'exécutent pendant une durée relativement plus courte, imposant une très forte contrainte temporelle à l'auto-tuning à la volée. Dans ce contexte, je crois qu'un outil léger doit être utilisé pour explorer des optimisations pré-identifiées.

Motivation

Dans cette section, je montre un exemple d'application dont la performance de la meilleure implémentation du code dépend du pipeline cible et du jeu de donnée d'entrée de l'application. Le code de

référence a été extrait du test de performance Streamcluster de PARVEC.

La Figure 5.1 présente les accélérations obtenues par l'application selon les différentes implémentations possibles de son principal noyau de calcul, sur deux plateformes ARM, avec deux jeux de donnée d'entrée différents. En analysant les résultats, nous concluons que :

1. Des accélérations importantes peuvent être obtenues grâce à l'auto-tuning, jusqu'à 1,46 et 1,52 dans l'A8 et l'A9 respectivement, même si le code de référence a été vectorisé à la main.
2. Les meilleurs paramètres d'optimisation du code varie d'un cœur à l'autre. Lorsqu'on exécute la meilleure version du code pour un cœur dans l'autre cœur, on obtient des ralentissements allant jusqu'à 35 %.
3. Il n'y a pas de corrélation de performance entre les paramètres d'optimisation et les deux jeux de donnée d'entrée.

Cet exemple montre que la spécialisation de code et l'auto-tuning permettent d'augmenter les possibilités d'optimisation dépendantes de l'architecture et des données d'entrés et accélérer considérablement une application basée sur un noyau de calcul. Néanmoins, si ni le cœur cible ni les données d'entrées ne sont connus à la compilation, ces optimisations ne peuvent se faire que pendant l'exécution de l'application. Le versionnage statique de code pourrait générer une explosion combinatoire de possibilités. C'est pour cela, que dans le contexte des systèmes embarqués, je propose l'auto-tuning à la volée.

Méthodologie

Cette section décrit l'architecture et l'implémentation du cadriciel d'auto-tuning à la volée d'une fonction. La Figure 5.2 illustre l'architecture de l'outil. Les trois principaux composant sont : le générateur de noyaux de calcul paramétrable (deGoal, décrit dans le Chapitre 4), de module de décision de régénération et d'exploration de l'espace, et le module d'évaluation et de décision de remplacement. Les composants d'auto-tuning s'exécutent dans un fil d'exécution en parallèle au fil de l'application.

À n'importe quel moment, une fonction dite active est celle appelée lorsque l'application en a besoin. Au démarrage, une fonction de référence doit commencer comme étant la fonction active. Elle est tout de suite évaluée pour avoir une performance de référence dans le système d'auto-tuning. Au fur et à mesure que l'application tourne, des nouvelles versions sont régénérées, évaluées et vont remplacer la fonction active si de meilleures performances sont obtenues, tout en tenant compte des surcharges dynamiques.

Dans ce travail, je présente quatre paramètres d'auto-tuning avec deGoal. La description se concentre sur un code exemple utilisé dans le test Streamcluster, mais les possibilités d'auto-tuning utilisées sont généralisables à la plupart des applications. Les quatre paramètres d'auto-tuning sont :

- **Facteur de déroulage de boucle à chaud (`hotUF`)** : Déroule une boucle et traite chaque élément avec un différent registre, ce qui évite des bulles dans le pipeline.
- **Facteur de déroulage de boucle à froid (`coldUF`)** : Déroule une boucle en copiant-collant un pattern de code, utilisant peu de registres, mais potentiellement créant des bulles dans le pipeline.

- **Longueur normalisée de vecteur (`vectLen`)** : Définit la longueur normalisée des vecteurs utilisés dans le traitement des éléments dans la corps de la boucle.
- **Pas de pré-chargement de donnée (`pldStride`)** : Définit le pas en octets utilisé dans les instructions de pré-chargement de données de la prochaine itération de la boucle.

En outre, trois options de génération de code peuvent être actives ou pas : ordonnancement d'instructions (IS), minimisation de la pile (SM) et vectorisation (VE).

La décision de régénérer une nouvelle fonction prend en compte deux facteurs : la surcharge de régénération en pourcentage du temps d'exécution de la fonction active et le nombre de cycles gagnés avec l'auto-tuning par rapport à la performance de référence. Par exemple, la surcharge dynamique serait limitée à 1 %, et 10 % du temps gagné serait investi pour chercher des nouvelles versions. Les valeurs nécessaires pour faire ces calculs sont estimées à l'aide des compteurs de performance.

Pour accélérer l'exploration de l'espace d'implémentations, j'ai empiriquement divisé l'exploration en deux phases : pendant la première, les paramètres qui ont un impact sur la structure du code sont explorés (`hotUF`, `coldUF`, `vectLen` et VE), et pendant la deuxième, toutes les combinaisons possibles des paramètres restants sont explorées (`pldStride`, IS, SM). Les limites de l'espace de recherche sont trouvées statiquement par un pré-profilage.

L'évaluation des nouveaux noyaux peut être fait sous deux configurations, avec : le jeu de données réel, ou un jeu de donnée fixe pendant la première phase et le réel pendant la deuxième. L'utilisation du jeu de données réel crée des oscillations de mesure, mais du travail utile est fait pendant l'évaluation du nouveau noyau. Elle est aussi obligatoire pour évaluer certaines optimisations comme le pré-chargement de données. L'utilisation d'un jeu de données fixe rend les mesure plus stables, mais son utilisation n'est pas possible si le noyau a des effets de bord ou si son temps d'exécution est prohibitif.

La nouvelle version de noyau remplacera la fonction active si une meilleure performance est obtenue.

Configuration expérimentale

Deux cartes ARM ont été utilisées dans les expérimentations, la BeagleBoard-xM (Cortex-A8) et la Snowball (Cortex-A9). Pour forcer l'exécution des tests de performance dans un seul cœur j'ai utilisé la commande `taskset` dans Linux.

Le cadriciel de simulation présenté dans le Chapitre 3 a été configuré pour simuler 11 différentes configurations de cœur. Le type de pipeline (exécution dans l'ordre ou dans le désordre), le nombre d'unités flottantes/vectorielles et la largeur du pipeline ont été variés.

Les expérimentations sont menées autour de deux cas d'étude d'application, avec : un noyau de calcul intensif (Streamcluster) et un noyau limité par les accès mémoire (VIPS avec une ligne de commande spécifique). Les versions scalaire (SISD) et vectorielles (SIMD) de deux applications ont été comparées, en exécutant trois jeux de données d'entrée différents. L'exécution des noyaux correspond à plus de 80 % du temps d'exécution des applications.

Pour déterminer l'écart entre la performance de l'auto-tuning à la volée par rapport à la performance des meilleures versions possibles, j'ai fait une exploration statique d'un espace d'implémentations assez

large. Pour chaque plateforme réelle et chaque jeu de données, l'exploration prend plusieurs heures.

Résultats expérimentaux

Dans les plateformes réelles, l'auto-tuning à la volée a obtenu des accélérations moyennes de 1,12 et de 1,41 dans l'A8 et l'A9, respectivement, sur l'application de calcul intensif.

De plus, nous observons que dans l'A9, la version SIMD de référence est 11 % plus lente que sa version SISD, à cause d'instructions de pré-chargement de données qui ne sont pas générés par gcc. Par contre, mon approche dynamique a obtenu avec l'auto-tuning SIMD une accélération de 1,41 par rapport au code SISD de référence et de 1,13 par rapport à l'auto-tuning SISD à la volée.

Dans l'application limitée par les accès mémoire, les accélérations obtenues sont en moyenne de 1,10 et de 1,04 dans l'A8 et l'A9, respectivement. Nous n'observons presque pas de ralentissements, même si avec cette application l'auto-tuning à la volée devrait avoir des problèmes pour trouver des meilleures versions, puisque le point d'étranglement est la mémoire.

L'écart moyen par rapport aux meilleures versions statiques est de seulement 6 % dans les deux cas d'étude, ce qui est un très bon résultat si nous considérons qu'entre 28 et 75 versions de noyaux sont régénérés et évalués à la volée. J'ai mesuré des coûts dynamiques de régénération entre 9 et 80 ms, et seulement dans 2 cas évalués sur 12 la surcharge dynamique est supérieure à 1 % du temps d'exécution de l'application.

Dans l'application de calcul intensif, les résultats de simulation montrent des accélérations de 1,58 pour code SISD et de 1,20 pour le SIMD. Seulement dans moins de 10 % des simulations, l'auto-tuning à la volée a ralenti l'exécution.

Des résultats intéressant sont obtenus lorsque nous comparons les simulations de l'application de calcul intensif dans les cœurs d'exécution dans l'ordre avec leurs cœurs d'exécution dans le désordre *correspondants*. Le terme *correspondant* signifie que tous les paramètres sont égaux, à l'exception de la capacité d'ordonnancer dynamiquement les instructions. L'auto-tuning à la volée a réduit l'écart de performance entre l'exécution dans l'ordre et dans le désordre de 16 à 6 % et a augmenté l'efficacité énergétique de 21 à 31 %, par rapport aux codes de référence. Si nous comparons l'auto-tuning à la volée sur les cœurs d'exécution dans l'ordre aux codes SIMD de référence exécutés dans les cœurs d'exécution dans le désordre correspondants, malgré le désavantage matériel, l'auto-tuning à la volée a donné une accélération moyenne de 1,03 et une amélioration de l'efficacité énergétique de 39 %.

Les résultats de simulation de la deuxième application ont montré des performances similaires entre l'auto-tuning à la volée et les codes de référence. Entre 29 et 79 nouvelles versions de noyaux sont générées, mais vu que l'application est limitée par les accès au disque dur, l'auto-tuning ne trouve pas de meilleure version.

Dans les plateformes réelles, j'ai analysé la performance de l'approche proposée en diminuant la charge de travail jusqu'à quelques millisecondes. En général, l'auto-tuning SISD apporte presque toujours des accélérations, différemment de l'auto-tuning SIMD. L'explication vient du fait que la fonction initiale est le code SISD de référence, donc au début l'auto-tuning SIMD exécute des instructions scalaires, mais je compare sa performance au code de référence vectorisé. J'ai considéré qu'une version de fonction initiale SISD est un cas plus réaliste.

Une analyse de corrélation entre les paramètres d'auto-tuning et les designs de pipeline simulés a été faite. Sur quatre des six paramètres, il y a au moins une corrélation avec la largeur ou longueur du pipeline, ou la capacité d'exécuter dans le désordre.

Portée et limitations

Dans cette section, je discute la portée et des limitations de l'approche proposée et des expérimentations menées.

Les résultats obtenus à la volée devraient être comparés à ceux des approches d'auto-tuning statique, afin de mieux évaluer la pertinence de la technique proposée.

L'approche proposée, ainsi que deGoal, est adaptée aux noyaux de calcul pré-identifiés. De plus, il y a beaucoup de configurations qui dépendent d'un pré-profilage. Dans le manuscrit je discute deux possibilités pour traiter ces limitations.

J'ai observé des problèmes de stabilité pour mesurer le temps d'exécution des noyaux qui durent moins d'une microseconde, avec les données réelles. Pour palier à ce problème, un jeu de données fixe a été utilisé avec les caches chauffés. Par contre, cette technique ne peut pas prendre en compte l'interaction entre le code et le pipeline avec les données réelles.

Pendant mes expérimentations, l'application s'exécute seule dans un processeur et l'auto-tuning s'arrête après les deux phases d'exploration. Par contre, dans un environnement réel, l'application peut passer par différentes phases d'exécution selon la charge de travail dans le système. Donc, une approche d'auto-tuning continue et adaptative serait intéressant dans ce contexte.

Finalement, les accélérations des versions SISD obtenues par simulation pourraient être surestimées, parce que gcc génère une chaîne d'instructions multiply-accumulate flottantes qui dépendent du même registre de destination, autant que gem5 ne simule pas la division de cette instruction en multiplication, bypass et addition.

Conclusion

Ce chapitre a présenté une méthodologie et la preuve de concept d'un système d'auto-tuning à la volée pour les systèmes embarqués. À ma connaissance, ce travail est le premier à montrer la faisabilité de l'auto-tuning à la volée dans une application qui s'exécute pendant quelques secondes sur un processeur embarqué.

L'approche présentée peut à la fois optimiser le code d'un noyau de calcul à une micro-architecture non définie à la compilation et chercher des nouvelles implémentations plus performantes qui dépendent d'une donnée d'entrée du programme. Dans ce scénario, une approche d'auto-tuning statique ne serait pas adaptée à moins qu'un grand nombre de versions de code soient statiquement générées et versionnées.

Chapitre 6 : Conclusion et perspectives

Selon des études sur les tendances d'architecture d'ordinateur, l'hypothèse principale de cette thèse est que les futurs systèmes sur puce embarqueront des centaines voire des milliers de cluster de cœurs hétérogènes, chacun adapté pour une tâche donné. Éventuellement, les cœurs pourront aussi embarquer les pipelines hétérogènes (en anglais, clustered cores). Toute cette hétérogénéité sera nécessaire pour améliorer l'efficacité énergétique du matériel, mais le développement logiciel va faire face à de croissantes difficultés pour obtenir l'énergie et la performance promise par le matériel hétérogène. Des techniques dynamiques seront probablement indispensable pour traiter la complexité et adapter le logiciel au comportement dynamique du système.

En considérant le scénario hypothétique précédent, cette thèse a proposé une méthodologie et a implémenté une preuve de concept d'un cadriciel d'auto-tuning à la volée pour les processeurs embarqués. Un cadriciel de simulation de cœurs ARM a été développé pour étudier la capacité de l'approche d'auto-tuning proposée à adapter du code à diverses micro-architectures.

Simulation de cœurs embarqués avec gem5 et McPAT

La simulation micro-architecturale est utilisée pour évaluer de nouvelles idées d'implémentation matérielle, pour explorer l'espace de conception ou pour simuler des implémentations matérielles hypothétiques. Pour l'étude de cœurs embarqués, gem5 et McPAT sont parmi les meilleurs choix de simulateurs micro-architecturaux.

Toutefois, au début de cette thèse, gem5 n'avait pas de modèle d'exécution dans l'ordre pour ARM. Donc, j'ai développé un modèle approximatif en modifiant le modèle fonctionnel d'exécution dans le désordre. Les deux modèles, le proposé et l'original, ont été configurés comme des Cortex-A8 et A9 et validés par rapport aux modèles réels. Mes expérimentations ont montré que en moyenne les deux modèles de simulation ont des erreurs de timing absolus autour de 7 %, en exécutant 10 tests de performance complexes de la suite PARSEC 3.0. Ces résultats démontrent que gem5 est considérablement plus précis que des simulateurs similaires largement validés, qui ont des erreurs absolues moyens supérieur à 15 %.

Pour obtenir des estimation énergétiques, j'ai analysé les modèles de pipeline dans gem5 et McPAT pour créer des règles de conversion de paramètres et de statistiques du premier vers le deuxième. En analysant la surface de cœurs ARM, j'ai proposé une simple modification dans McPAT et une méthodologie pour mieux estimer la surface et le coût énergétique empiriques des unités fonctionnelles de cœurs hétérogènes. La validation de la surface estimée de CPUs big.LITTLE a montré des bons résultats : des erreurs inférieures à 4 % pour la surface des cœurs et jusqu'à 13 % pour celles des clusters. Une fois qu'ARM a publié le floorplan du Cortex-A7, j'ai aussi pu comparer l'estimation de la surface de cinq principales structures de son pipeline. Bien que la surface estimée du cœur correspond parfaitement, j'ai observé des erreurs de 3,6 à 60 %, mais la structure qui impacte le plus l'erreur de la surface du cœur contribue seulement avec 5,3 % à cette erreur. Afin de valider les estimations relatives d'énergie/performance du cadriciel de simulation proposé, j'ai simulé des cœurs big.LITTLE en exécutant 11 tests de performance. Pour le test Dhrystone 2.1, les estimations relatives d'énergie/performance sont à 6 % près des valeurs publiées.

Ce cadriciel de simulation a permis l'étude de l'hétérogénéité de cœurs embarqués et l'évaluation de l'adaptabilité de code de l'approche d'auto-tuning à la volée proposée à diverses micro-architectures.

Génération de code et auto-tuning à la volée pour les systèmes embarqués

Les systèmes existants de génération de code et d'auto-tuning à la volée sont surtout développés et matures pour des systèmes non-embarqués. À ma connaissance, cette thèse est la première à traiter l'auto-tuning à la volée pour les systèmes embarqués.

La méthodologie et la preuve de concept d'un outil d'auto-tuning à la volée pour les systèmes embarqués ont été développées autour de deux cas d'étude : deux applications basées sur noyau de calcul, une de calcul intensif et l'autre limitée par les accès mémoire. Les versions SISD et SIMD de chaque application ont été comparés, en exécutant trois jeux de données d'entrée. Dans ces études, je me suis concentré sur l'adaptation micro-architecturale de code, grâce à l'auto-tuning du déroulage de boucle, de la taille des vecteurs et du pré-chargement de données.

Pour but d'accélérer le processus d'auto-tuning, j'ai proposé une exploration en deux phases, basée sur des observations empiriques.

Les expérimentations ont été effectuées sur le Cortex-A8 et A9, et aussi sur 11 cœurs hétérogènes ARM simulés avec différentes largeurs de pipeline, nombre d'unités flottantes/vectorielles et capacités d'ordonnancement d'instruction (statique ou dynamique).

Dans l'application de calcul intensif, nous avons observé des accélérations moyennes de 1,26 et de 1,38 sur des cœurs réels et simulés, respectivement, jusqu'à 1,79 et 2,53 (toutes les surcharges dynamiques incluses). Nous avons aussi observé que l'approche proposée produit des dégradation de performance négligeables lorsque l'auto-tuning ne fournit pas des meilleures versions de noyau de calcul.

En comparant les résultats de simulation de l'application de calcul intensif sur les cœurs d'exécution dans l'ordre par rapport à leurs cœurs d'exécution dans le désordre correspondants, nous avons observé que l'approche d'auto-tuning proposée peut réduire l'écart de performance de 16 % à 6 % et augmenter l'efficacité énergétique de 21 à 31 %.

De plus, j'ai démontré que l'adaptation à la volée de code à la micro-architecture des pipelines d'exécution dans l'ordre peut en moyenne surpasser la performance du code de référence vectorisé à la main et exécutés sur des pipelines correspondants d'exécution dans le désordre : en dépit de la désavantage matérielle, lorsque l'approche proposée est utilisée dans l'application de calcul intensif, elle obtient une accélération moyenne de 1,03 et une amélioration de l'efficacité énergétique de 39 %.

Dans les processeurs réels, les performances obtenues par l'auto-tuning à la volée sont en moyenne seulement à 6 % près de celles obtenues par les meilleures versions trouvées statiquement.

Afin de développer ce système d'auto-tuning, j'ai porté deGoal, un générateur de code à la volée léger, aux processeurs ARM. Pour valider la qualité du code produit par deGoal, j'ai implémenté et évalué des versions scalaires (SISD) et vectorielles (SIMD) de quatre noyaux de calcul des suites PARSEC 3.0 et PARVEC.

Les résultats obtenus dans les cœurs Cortex-A8 et A9 ont démontré qu'en moyenne deGoal génère du code machine avec une qualité équivalente ou supérieure, et ce, même comparé aux noyaux vectorisés à la main de PARVEC.

Des expérimentations de spécialisation dynamique de code avec les mêmes noyaux ont produit des

accélérations de 1,32 et de 1,07 par rapport aux versions de référence PARSEC et PARVEC, respectivement, avec des surcharges dynamiques négligeables, inférieures à 0,1 % du temps d'exécution des noyaux de calcul. Grâce à cette petite surcharge dynamique et aux asymétries de performance observées, j'ai expliqué que deGoal pourrait être utilisé pour faire de l'auto-tuning de noyaux de calcul dans des applications de courte durée.

Résumé des réalisations

Cette thèse a été développée autour de deux thématiques principales, qui viennent d'être détaillées. La liste suivante résume les réalisations :

- Simulation de cœurs embarqués avec gem5 et McPAT :
  - Proposition et développement d'un modèle approximatif d'exécution dans l'ordre pour ARM dans gem5.
  - Validation du modèle proposé d'exécution dans l'ordre et du modèle original d'exécution dans le désordre dans gem5 par rapport à un Cortex-A8 et un A9.
  - Enrichissement du modelage de gem5 pour mieux simuler des cœurs embarqués.
  - Étude de modèles de pipeline et développement de règles de conversion de paramètres et statistiques de gem5 vers McPAT.
  - Enrichissement du modelage de McPAT pour mieux estimer la surface et la consommation énergétique d'unités fonctionnelles de cœurs hétérogènes.
  - Validation de surface et validation relative d'énergie et de performance du cadriciel de simulation proposé par rapport à CPUs big.LITTLE.
- Génération de code et auto-tuning à la volée :
  - Portage de deGoal au jeu d'instructions ARM Thumb-2, y compris des extensions flottante et vectorielle.
  - Validation préliminaire de la génération de code scalaire (SISD) et vectoriel (SIMD) pour des processeurs ARM.
  - Étude de spécialisation dynamique de programme et de possibilités d'auto-tuning avec deGoal.
  - Proposition de méthodologie, preuve de concept et étude d'auto-tuning à la volée dans des cœurs embarqués ARM.

Charge de travail

Une estimation du temps consacré à chaque étape de la thèse est donnée à la suite.

- **État de l'art** : 3 mois.
- **Développements avec gem5/McPAT** : 4,9 mois.

  - Prise en main : 1 mois.
  - Étude des modèles dans gem5 et dans McPAT, conversion de paramètres et de statistiques : 2 mois.
  - Modèle d'exécution dans l'ordre approximatif : 1,2 mois.
  - Extensions de McPAT et validation de surface du big.LITTLE : 0,7 mois.
- **Expérimentations et rédaction d'article sur gem5/McPAT** : 3,3 mois.
  - Environnement de compilation de tests de performance, validation des modèles par rapport au A8 et au A9 : 2,6 mois.
  - Estimations énergétiques et analyse de résultats de simulation du A8 : 0,2 mois.
  - Configuration de CPUs big.LITTLE et simulations : 0,6 mois.
- **Développements avec deGoal** : 4,9 mois.
  - Prise en main : 1 mois.
  - Portage au jeu d'instructions ARM Thumb-2 : 1,5 mois.
  - Extensions et améliorations : 2,4 mois.
- **Expérimentations avec deGoal** : 3,3 mois.
  - Environnement de compilation, mise en place de compteurs de performance et expérimentations avec le noyau de convolution : 1,6 mois.
  - Noyau de transformation linéaire : 0,6 mois.
  - Noyau d'interpolation : 0,4 mois.
  - Noyau de distance euclidienne : 0,4 mois.
  - Rédaction d'article : 0,3 mois.
- **Methodologie et preuve de concept d'auto-tuning à la volée** : 2,9 mois.
- **Rédaction de la thèse** : 2,6 mois.

Perspectives

À cause du power wall et du dark silicon, il y a une tendance croissante d'embarquer des unités hétérogènes dans les systèmes sur puce. Cette complexité croissante va affecter aussi bien le design architectural que le développement logiciel. Pour éviter une stagnation de la performance de systèmes informatiques pendant les prochaines décennies, des efforts considérables seront nécessaires pour changer les paradigmes architecturaux et de programmation actuels [82].

À la lumière des défis de performance et d'efficacité énergétique que les prochaines générations de systèmes informatiques feront face, des études architecturales et micro-architecturales peuvent donner de potentielles nouvelles idées pour améliorer la performance matérielle avec les mêmes coûts énergétiques et de puissance. Nous voyons les tendances de faire du calcul générique sur des processeurs graphiques, d'intégrer des cœurs hétérogènes, et plus récemment de mélanger des circuits de logique fixe et programmable. Les prochaines tendances de design micro-architectural pourraient non seulement inclure des unités fonctionnelles spéciales ou des accélérateurs intégrés dans un pipeline générique pour

les calculs dédiés ou approximatifs, mais aussi des cœurs spécialisés pour des tâches et contraintes spécifiques, tel que des cœurs de calcul près de la tension de seuil pour une efficacité énergétique élevée et l'architecture EOLE, qui diminue la complexité de designs d'exécution dans le désordre avec la prédiction de valeur pour accélérer les parties non-parallèles des applications [111]. Le cadriciel développé durant cette thèse est un bon point de départ pour les études micro-architecturales, puisque des estimations acceptables de surface, d'énergie et de performance de designs de cœurs embarqués actuels ont été démontrés par plusieurs expérimentations.

Au regard du croissant problème de portabilité de performance, cette thèse a étudié la faisabilité d'un outil d'auto-tuning à la volée pour les systèmes embarqués. Dans les futurs manycœurs hétérogènes, des approches dynamiques pourraient être la seule solution pour améliorer davantage l'efficacité énergétique dans des environnements dynamiques. Cette thèse a étudié l'auto-tuning de noyaux de calcul qui s'exécutent sur un seul cœur. Donc, étant donné que des cœurs supplémentaire ne sont pas utilisés pour les régénérations et les évaluations, l'approche proposée se met à l'échelle des multi/manycœurs asymétriques. Dans ce contexte, l'auto-tuning à la volée peut fournir une meilleure portabilité de performance comparé à des binaires compilés statiquement. La vitesse d'auto-tuning observée dans mes expérimentations pourrait permettre de faire de l'auto-tuning dans les compilateurs anticipés (en anglais, ahead-of-time compilers) avec une surcharge en temps de compilation négligeable. Une autre perspective de cette thèse serait la définition d'un langage plus haut-niveau pour l'auto-tuning, lequel serait traduit automatiquement en compilettes deGoal.

Abstract:

In computing systems, energy consumption is limiting the performance growth experienced in the last decades. Consequently, computer architecture and software development paradigms will have to change if we want to avoid a performance stagnation in the next decades.

In this new scenario, new architectural and micro-architectural designs can offer the possibility to increase the energy efficiency of hardware, thanks to hardware specialization, such as heterogeneous configurations of cores, new computing units and accelerators. On the other hand, with this new trend, software development should cope with the lack of performance portability to ever changing hardware and with the increasing gap between the performance that programmers can extract and the maximum achievable performance of the hardware. To address this issue, this thesis contributes by proposing a methodology and proof of concept of a run-time auto-tuning framework for embedded systems. The proposed framework can both adapt code to a micro-architecture unknown prior compilation and explore auto-tuning possibilities that are input-dependent.

In order to study the capability of the proposed approach to adapt code to different micro-architectural configurations, I developed a simulation framework of heterogeneous in-order and out-of-order ARM cores. Validation experiments demonstrated average absolute timing errors around 7 % when compared to real ARM Cortex-A8 and A9, and relative energy/performance estimations within 6 % for the Dhrystone 2.1 benchmark when compared to Cortex-A7 and A15 (big.LITTLE) CPUs.

An important component of the run-time auto-tuning framework is a run-time code generation tool, called deGoal. It defines a low-level dynamic DSL for computing kernels. During this thesis, I ported deGoal to the ARM Thumb-2 ISA and added new features for run-time auto-tuning. A preliminary validation in ARM processors showed that deGoal can on average generate equivalent or higher quality machine code compared to programs written in C, including manually vectorized codes.

The methodology and proof of concept of run-time auto-tuning in embedded processors were developed around two kernel-based applications, extracted from the PARSEC 3.0 suite and its hand vectorized version PARVEC. In the favorable application, average speedups of 1.26 and 1.38 were obtained in real and simulated cores, respectively, going up to 1.79 and 2.53 (all run-time overheads included). I also demonstrated through simulations that run-time auto-tuning of SIMD instructions to in-order cores can outperform the reference vectorized code run in similar out-of-order cores, with an average speedup of 1.03 and energy efficiency improvement of 39 %. The unfavorable application was chosen to show that the proposed approach has negligible overheads when better kernel versions can not be found. When both applications run in real hardware, the run-time auto-tuning performance is on average only 6 % way from the performance obtained by the best statically found kernel implementations.

Title: Online Auto-Tuning for Performance and Energy through Micro-Architecture Dependent Code Generation

Keywords: micro-architecture, simulation, gem5, McPAT, in-order, out-of-order, heterogeneous cores, pipeline, dynamic code generation, deGoal, data specialization, program specialization, online auto-tuning, embedded systems

Résumé:

Dans les systèmes informatiques, la consommation énergétique est devenue le facteur le plus limitant de la croissance de performance. Conséquemment, les paradigmes d'architectures d'ordinateur et de développement logiciel doivent changer si nous voulons éviter une stagnation de la performance durant les décennies à venir.

Dans ce nouveau scénario, des nouveaux designs architecturaux et micro-architecturaux peuvent offrir des possibilités d'améliorer l'efficacité énergétique des ordinateurs, grâce à la spécialisation matérielle, comme par exemple les configurations de cœurs hétérogènes, des nouvelles unités de calcul et des accélérateurs. D'autre part, avec cette nouvelle tendance, le développement logiciel devra faire face au manque de portabilité de la performance entre les matériels toujours en évolution et à l'écart croissant entre la performance exploitée par les programmeurs et la performance maximale exploitable du matériel. Pour traiter ce problème, la contribution de cette thèse est une méthodologie et la preuve de concept d'un cadriciel d'auto-tuning à la volée pour les systèmes embarqués. Le cadriciel proposé peut à la fois adapter du code à une micro-architecture inconnue avant la compilation et explorer des possibilités d'auto-tuning qui dépendent des données d'entrée d'un programme.

Dans le but d'étudier la capacité de l'approche proposée à adapter du code à des différentes configurations micro-architecturales, j'ai développé un cadriciel de simulation de processeurs hétérogènes ARM avec exécution dans l'ordre ou dans le désordre, basé sur les simulateurs gem5 et McPAT. Les expérimentations de validation ont démontré en moyenne des erreurs absolues temporels autour de 7 % comparé aux ARM Cortex-A8 et A9, et une estimation relative d'énergie et de performance à 6 % près pour le benchmark Dhrystone 2.1 comparée à des CPUs Cortex-A7 et A15 (big.LITTLE). Les résultats de validation temporelle montrent que gem5 est beaucoup plus précis que les simulateurs similaires existants, dont les erreurs moyennes sont supérieures à 15 %.

Un composant important du cadriciel d'auto-tuning à la volée proposé est un outil de génération dynamique de code, appelé deGoal. Il définit un langage dédié dynamique et bas-niveau pour les noyaux de calcul. Pendant cette thèse, j'ai porté deGoal au jeu d'instructions ARM Thumb-2 et créé des nouvelles fonctionnalités pour l'auto-tuning à la volée. Une validation préliminaire dans des processeurs ARM ont montré que deGoal peut en moyenne générer du code machine avec une qualité équivalente ou supérieure comparé aux programmes de référence écrits en C, et même par rapport à du code vectorisé à la main.

La méthodologie et la preuve de concept de l'auto-tuning à la volée dans des processeurs embarqués ont été développées autour de deux applications basées sur noyau de calcul, extraits de la suite de benchmark PARSEC 3.0 et de sa version vectorisée à la main PARVEC. Dans l'application favorable, des accélérations de 1,26 et de 1,38 ont été observées sur des cœurs réels et simulés, respectivement, jusqu'à 1,79 et 2,53 (toutes les surcharges dynamiques incluses). J'ai aussi montré par la simulation que l'auto-tuning à la volée d'instructions SIMD aux cœurs d'exécution dans l'ordre peut surpasser le code de référence vectorisé exécuté par des cœurs d'exécution dans le désordre similaires, avec une accélération moyenne de 1,03 et une amélioration de l'efficacité énergétique de 39 %. L'application défavorable a été choisie pour montrer que l'approche proposée a une surcharge négligeable lorsque des versions de noyau plus performantes ne peuvent pas être trouvées. Sur les processeurs réels, la performance de l'auto-tuning à la volée est en moyenne seulement 6 % en dessous de la performance obtenue par la meilleure implémentation de noyau trouvée statiquement, dont le processus d'auto-tuning prend plusieurs heures par processeur et par jeu de donnée d'entrée.

Mots clés: micro-architecture, simulation, gem5, McPAT, exécution dans l'ordre, exécution dans le désordre, cœurs hétérogènes, pipeline, génération dynamique de code, deGoal, spécialisation de donnée, spécialisation de programme, auto-tuning à la volée, systèmes embarqués