Native simulation of MPSoC : instrumentation and modeling of non-functional aspects

Oumaima Matoussi

Communauté UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES

Spécialité : Informatique

Arrêté ministériel : 25 mai 2016

Présentée par

Oumaima MATOUSSI

Thèse dirigée par **Frédéric PETROT**

préparée au sein du **Laboratoire Techniques de l'Informatique et de la Microélectronique pour l'Architecture des systèmes intégrés (TIMA)**
dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Native Simulation of MPSoC: Instrumentation and Modeling of Non-Functional Aspects

Thèse soutenue publiquement le **30 novembre 2017**,
devant le jury composé de :

Monsieur FREDERIC PETROT
PROFESSEUR, GRENOBLE INP, Directeur de thèse
Madame CHRISTINE ROCHANGE
PROFESSEUR, UNIVERSITE TOULOUSE-III-PAUL-SABATIER, Rapporteuse
Madame FLORENCE MARANINCHI
PROFESSEUR, GRENOBLE INP, Présidente
Monsieur FRANÇOIS PECHEUX
PROFESSEUR, UNIVERSITE PIERRE ET MARIE CURIE, Rapporteur
Monsieur BENOÎT DUPONT DE DINECHIN
DIRECTEUR DE LA TECHNOLOGIE, KALRAY S.A. - MONTBONNOT-SAINT-MARTIN, Examinateur
Monsieur ABDOULAYE GAMATIE
DIRECTEUR DE RECHERCHE, CNRS DELEGATION LANGUEDOC-ROUSSILLON, Examinateur

Acknowledgement

I am particularly grateful to my thesis advisor Prof. Frédéric Petrot for his continuous support of my research, his immense knowledge and his guidance.
I would like to thank my jury members, Prof. Florence MARANINCHI, Prof. Abdoulaye GAMATIE, Prof. Christine ROCHANGE, Prof. François PECHEUX and Dr. Benoit DUPONT DE DINECHIN for their insightful comments.
I wish to acknowledge the help of Prof. Frédéric Rousseau, Dr. Guillaume Sarrazin, Dr. Liliana Lilibeth Andrade Porras and all those whose assistance proved to be a milestone in the accomplishment of this thesis.
To all my labmates, especially the SLS Team, it has been great sharing the laboratory with you during these past three years.
Finally, I owe my deepest gratitude to my family who has provided me with moral and emotional support.

Résumé

Les systèmes embarqués modernes intègrent des dizaines, voire des centaines, de cœurs sur une même puce communiquant à travers des réseaux sur puce, afin de répondre aux exigences de performances édictées par le marché. On parle de systèmes massivement multi-cœurs ou systèmes many-cœurs. La complexité de ces systèmes fait de l'exploration de l'espace de conception architecturale, de la co-vérification du matériel et du logiciel, ainsi que de l'estimation de performance, un vrai défi. Cette complexité est généralement compensée par la flexibilité du logiciel embarqué. La dominance du logiciel dans ces architectures nécessite de commencer le développement et la vérification du matériel et du logiciel dès les premières étapes du flot de conception, bien avant d'avoir accès à un prototype matériel.

Ainsi, il faut disposer d'un modèle abstrait qui reproduit le comportement de la puce cible en un temps raisonnable. Un tel modèle est connu sous le nom de *plateforme virtuelle* ou *de simulation*. L'exécution du logiciel sur une telle plateforme est couramment effectuée au moyen d'un simulateur de jeu d'instruction (ISS). Ce type de simulateur, basé sur l'interprétation des instructions une à une, est malheureusement caractérisé par une vitesse de simulation très lente, qui ne fait qu'empirer par l'augmentation du nombre de cœurs.

La simulation native est considérée comme une candidate adéquate pour réduire le temps de simulation des systèmes many-cœurs. Le principe de la simulation native est de compiler puis exécuter la quasi totalité de la pile logicielle directement sur la machine hôte tout en communiquant avec des modèles réalistes des composants matériels de l'architecture cible, permettant ainsi de raccourcir les temps de simulation. La simulation native est beaucoup plus rapide qu'un ISS mais elle ne prend pas en compte les aspects non-fonctionnels, tel que le temps d'exécution, dépendant de l'architecture matérielle réelle, ce qui empêche de faire des estimations de performance du logiciel.

Ceci dresse le contexte des travaux menés dans cette thèse qui se focalisent sur la simulation native et s'articulent autour de deux contributions majeures. La première s'attaque à l'introduction d'informations non-fonctionnelles dans la représentation intermédiaire (IR) du compilateur. L'insertion précise de telles informations dans le modèle fonctionnel est réalisée grâce à un algorithme dont l'objectif est de trouver des correspondances entre le code binaire cible et le code IR tout en tenant compte des optimisations faites par le compilateur. La deuxième contribution s'intéresse à la modélisation d'un cache d'instruction et d'un tampon d'instruction d'une architecture VLIW pour générer des estimations de performance précises.

Ainsi, la plateforme de simulation native associée à des modèles de performance précis et à une technique d'annotation efficace permet, malgré son haut niveau d'abstraction, non seulement de vérifier le bon fonctionnement du logiciel mais aussi de fournir des estimations de performances précises en des temps de simulation raisonnables.

Abstract

Modern embedded systems are endowed with a high level of parallelism and significant processing capabilities as they integrate hundreds of cores on a single chip communicating through network on chip. The complexity of these systems and their dedicated software should not be an excuse for long design cycles, even though the design space is enormous and the underlying design decisions are critical. Thus, design space exploration, hardware/software co-verification and performance estimation need to be conducted within a reasonable amount of time and early enough in the design process to avoid any tardy detection of functional or performance deficiencies.

Co-simulation platforms are becoming an increasingly important part in design and verification steps. With instruction interpretation-based software simulation platforms being too slow as they model low-level details of the target system, an alternative software simulation approach known as native simulation or host-compiled simulation has gained momentum this past decade. Native simulation consists of compiling the embedded software to the host binary format and executing it directly on the host machine. However, this technique fails to reflect the performance of the embedded software and its actual interaction with the target hardware. So, the speedup gained by native simulation comes at a price, which is the absence of non-functional information (such as time and energy) needed for estimating the performance of the entire system and ensuring its proper functioning. Without such information, native simulation approaches are limited to functional validation.

Yielding accurate estimates entails the integration of high-level abstract models that mimic the behavior of target-specific micro-architectural components in the simulation platform and the accurate placement of the obtained non-functional information in the high-level code. Back-annotating non-functional information at the right place requires a mapping between the binary instructions and the high-level code statements, which can be challenging particularly when compiler optimizations are enabled.

In this thesis, we propose an annotation framework working at the compiler intermediate representation level to accurately annotate performance metrics extracted from the binary code, thanks to a dedicated mapping algorithm. This mapping algorithm is further enhanced to deal with aggressive compiler optimizations, such as loop unrolling, that radically alter the structure of the code. Our target architecture being a VLIW processor, we also model at a high level its instruction buffer to faithfully reproduce its timing behavior.

The experiments we conducted to validate our mapping algorithm and component models yielded accurate results and high simulation speed compared to a cycle accurate ISS of the target platform.

Keywords: native simulation, performance estimation, back-annotation, intermediate-level simulation (ILS), intermediate representation (IR), control flow graph (CFG), mapping algorithm, compiler optimizations, very long instruction word (VLIW), instruction cache model, instruction buffer model

Contents

List of Figures

List of Tables

Chapter 1

Introduction

To cater for the continuously increasing requirements for better performance; high computational capabilities, low power consumption, flexibility, programmability, short time to market and a reasonable cost, the VLSI (Very Large Scale Integration) technology allows IC (Integrated Circuit) designers to integrate billions of transistors into a single chip. As a result, groundbreaking hardware architectures composed of multiple processor cores, the so-called Multi-Processor System on Chip (MPSoC), have become ubiquitous since 2001, the release date of the first general purpose multicore processor POWER4 of IBM that integrated two cores on a single silicon die.

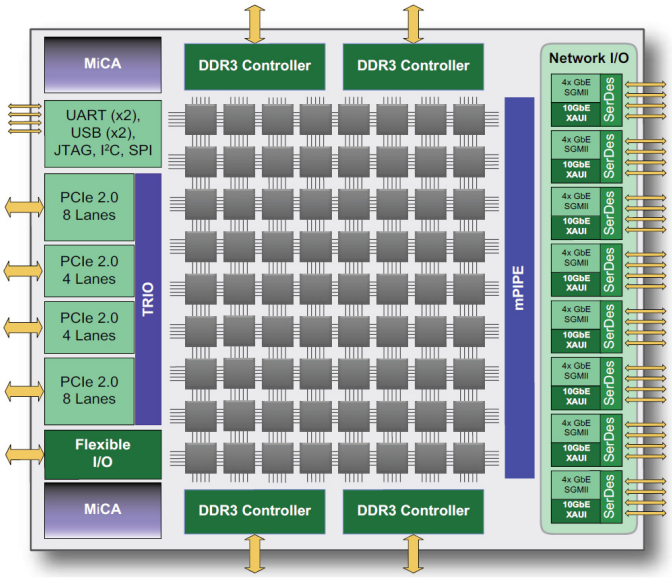


Figure 1.1: A block diagram of Tilera's Gx8072 Tile [Til]

1.1 Many-Core SoC: The Need for Higher Degrees of Parallelism

MPSoCs with a small number of cores, although relatively recent, are outstripped by their many-core successors. MPSoCs containing dozens or even hundreds of cores have become in vogue these last years (Kalray MPPA-256 [DdDAB+13], Intel SCC [HDH+10] ,Tilera TileMx [Til] fig 1.1) offering more parallelism and more processing capabilities. These cores

are usually organized in clusters (a.k.a. tiles). As depicted in fig 1.1, the Tilera device is structured as a 2D array of tiles and it includes 72 tiles in total. Each tile is composed of a 64-bit processor core and a L1 data and instruction caches.

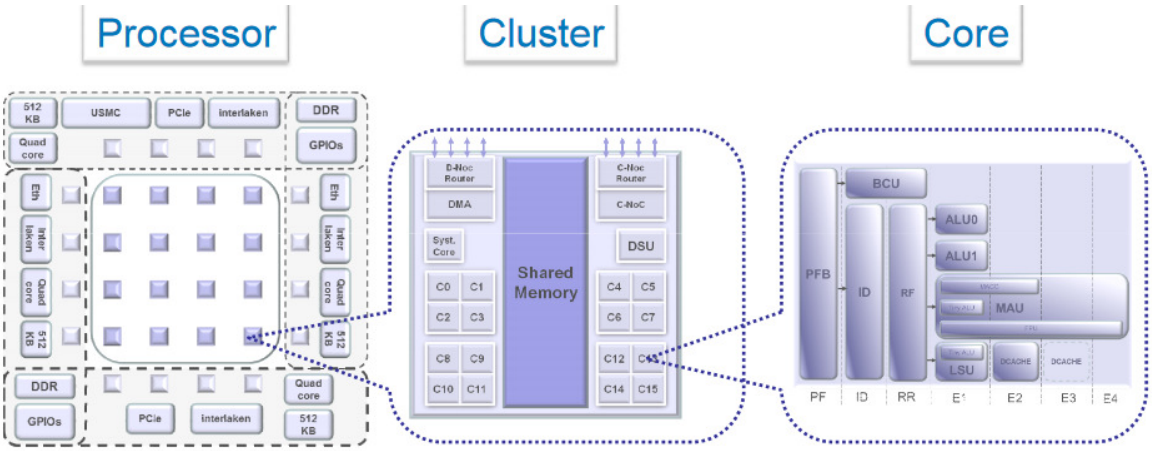


Figure 1.2: Hierarchical architecture of Kalray MPPA-256 manycore processor [DdDAB+13]

Kalray MPPA [DdDAB+13] also offers a clustered architecture (fig 1.2), but with more parallelism inside individual clusters by incorporating 16+1 cores in each one of its 16 compute clusters. Each tile (fig 1.2-middle part) is also equipped with a shared memory, a DMA (Direct Memory Access) unit, a DSU (Debug Support Unit) and interfaces for the interconnect.

Having multiple clusters on a single chip entails inter-cluster communication. In the first generation of multi-core processor systems, the communication was usually performed by a conventional bus shared between the different cores. Due to the need for more bandwidth, Netwok on Chip (NoC) has emerged 15 years ago as a bus replacement.

For more performance gains, parallelism does not stop at the process level (fig 1.2-left side) or thread (fig 1.2-middle part) level, but also reaches the instruction level (fig 1.2-right side). VLIW (Very Long Instruction Word) and superscalars bear witness to the evolution of architectures from scalar processors that issue one instruction per cycle to processors that have the capabilities to exploit ILP (Instruction Level Parallelism). The current breed of processors, e.g. MPPA manycore by Kalray [DdDAB+13], ST200 series by ST microelectronics in platform 2012 [MBF+12], the Tilera processor in Tile64 [TKM+02], etc., makes use of VLIW architectures to achieve high execution speed at low energy. Fig 1.2-right side illustrates an example of a VLIW core. It is the Kalray 5-issue VLIW core with five execution units: two arithmetic and logic units (ALU0, ALU1), a multiply-accumulate and floating-point unit (MAU), a load/store unit (LSU), and a branch and control unit (BCU).

1.2 Hardware/Software Co-Simulation

The unquenchable thirst for better performance led to the advent of these massively-parallel and complex architectures featuring a large number of clusters with many sophisticated processors, which benefit from ILP design techniques and are able to communicate through advanced interconnects. These complex architectures are deployed to run equally complex software. So, the performance of many-core architectures is governed not only by hardware characteristics, such as the processor type (VLIW, Superscalar, out-of-order, etc.), the

number of processors and the NoC type, but also by the software executed on top of these hardware components. Thus, the design space of such systems is enormous. Many design alternatives should be analyzed and many decisions involving both software and hardware need to be taken. Scheduling strategies and task mapping need to be examined at early design stages and a large amount of functionality and performance properties need to be validated without having to wait until the hardware chip is manufactured. Consequently, early design space exploration, HW/SW co-verification and performance analysis of massively parallel MPSoCs create the need for fast yet accurate modeling and simulation tools.

A virtual prototype (VP) is basically a software code that models and simulates the behavior of a hardware system including processors, memories, peripherals and the interconnect. VPs can be used by software engineers to debug and validate software applications if the real hardware is not available yet or if early prototypes are not affordable. VPs are also used by hardware engineers for design space exploration and performance evaluation of micro-architectural models. VPs enable the software development process to start early on and simultaneously to hardware design. The closer the models are to reality, the slower the simulation is, and the more accurate the results are. However, the competitiveness of MPSoC vendors cannot afford long design cycles and delayed time-to-market, which entails efficient simulation tools; short simulation time and accurate simulation results.

A trade-off between simulation accuracy and simulation speed is imposed on system designers. Raising the abstraction level will definitely increase the simulation speed but will penalize the accuracy of the results as models will contain less implementation details. This is obviously true for both hardware models and software simulation tools.

1.3 Scope of the Thesis

Simulation of many-core architectures is a contradicting problem as it entails accurate simulation results at a fast simulation speed. A renowned software simulation approach used for conducting performance estimation of the target software is cycle accurate Instruction Set Simulation (ISS). ISS is a software interpretation approach that consists of transforming target machine instructions into host instructions. The interpretation process is organized in a loop where the target processor simulator sequentially performs instruction fetching, decoding and execution, at run time, based on the semantics described in the target ISA. Thus, ISS yields accurate performance estimates to the detriment of simulation speed, which is inadequate for early system-level design space exploration (DSE).

Native simulation, a.k.a. host-compiled simulation, techniques have emerged as an alternative that aims at abstracting the target low-level micro-architectural details in order to achieve significantly higher simulation speed compared to conventional cycle-accurate ISS. Host-compiled simulation consists of compiling the target software code on the host machine and natively executing the generated host instructions on the host computer to achieve the fastest possible functional simulation. Thus, native approaches have always been considered as functional simulation approaches with little to no focus on performance estimation because of the lack of target-specific details in such approaches.

The gap in performance estimation of many-core architectures between cycle accurate but very slow ISS and very fast but coarse-grained native simulation could be narrowed by *back-annotating* the target software with target-specific performance metrics obtained from abstract performance models of the target architecture. When annotated with accurate non-functional information (e.g. execution cycles, execution time, power consumption, etc.), the natively simulated software will yield accurate performance estimates while achieving a

speed higher than cycle accurate ISS. This way, host-compiled simulation is leveraged to reproduce not only the functional behavior of the target code but also its non-functional (e.g. temporal) behavior at the system-level.

Software back-annotation raises two orthogonal issues. The first one concerns the computation of non-functional information. Some information could be determined statically (e.g. the instruction count) by merely analyzing the target binary code. Other performance metrics (e.g. number of cache misses) are highly-dependent on the target micro-architectural components (memory caches, branch prediction mechanism, pipeline, etc.). So, modeling the impact of these components at a high abstraction level is key in obtaining accurate estimates. With the ever evolving architectures of MPSoCs and their distinguishing features (VLIW, super-scalar, out-of-order, etc.), using generic performance models is too inaccurate. So, new methodologies are required for performance estimation of such complex systems that take into account their distinctive features while maintaining a reasonable simulation speed.

The second issue concerns the coupling of the performance models and the functional model, i.e. the target software. In other words, where do we insert the performance metrics in the software code? It should be noted that the functional model could be employed at different representation levels: source level, compiler intermediate representation (IR) level or binary level. Since the compiler performs multiple advanced optimizations, the structure of the original code changes from one compilation stage to another. So, the choice of an adequate representation level is a crucial step in the back-annotation process. Inserting non-functional information into the right place in the functional model is not always a straightforward task because this information is extracted from the target binary code, which is the result of the final compilation stage. Establishing a mapping strategy between the target binary and the functional model is pivotal in injecting the annotations at the right place in the software and obtaining accurate performance estimates.

1.4 Outline

The remainder of this dissertation is organized as follows:

- **Chapter 2:** We lay the groundwork by explaining the concept of hardware/software co-simulation and presenting the different approaches of SW execution, as well as the different abstraction levels of virtual prototyping. We focus especially on native execution of software on top of a virtual platform and we describe its key issues, which we look to address in this thesis.

- **Chapter 3:** We review state-of-the-art approaches that deal with performance estimation using native simulation and we establish the fundamental concepts of native simulation.

- **Chapter 4:** We present our first contribution, which is an annotation framework that aims at inserting non-functional information into a functional model for the purpose of performance estimation. We explain our choice of the representation level of the functional model and we detail the proposed mapping approach.

- **Chapter 5:** We describe our second contribution, which is a cache performance model. Firstly, we present a conventional data cache model used as an example to explain the modeling process and the combination of a performance model with a functional

model. Secondly, we explain the proposed performance estimation model of an instruction cache and an instruction buffer of a VLIW architecture.

- **Chapter 6:** We conduct a set of experiments to validate our contributions and we discuss the obtained results.
- **Chapter 7:** We conclude the dissertation by summarizing the key contributions and proposing directions for future work.

Chapter 2

Problem Definition and Motivations

The complexity of MPSoCs is increasing as the integration of a large number of cores in a single silicon die is practically achievable and it is also considered an efficient architectural solution to support the execution of intensively parallel applications. Such complexity entails an onerous design process. At a very early design stage, analytical models are generally used. However, to facilitate design space exploration and early software development, simulation models are adopted instead. The simulation speed and the accuracy of simulation results are the two most seeked requirements when it comes to taking pertinent design decisions in a reasonable period of time. To meet the different design and precision needs, several abstraction levels are available for both hardware and software simulation. We will describe these abstraction levels, highlight the level that we are interested in and define the problems that we wish to address in this thesis. But first, we will start by briefly explaining the concept of Hardware/Software co-simulation of Multi-processor systems on chip.

2.1 HW/SW Co-Simulation of MPSoCs

During the early years of microprocessor utilization in embedded systems, software development was delayed until a hardware prototype was fabricated. This sequential design workflow leads to late discovery of HW/SW integration problems, which can be very expensive (money-wise and time-wise) to solve. However, the advances made in the area of hardware modeling and simulation, by means of virtual prototyping, enables the concurrent development of hardware and software.

Virtual prototyping consists in creating a software model that replicates the target architecture and is used for functional validation, performance evaluation, early software development, non-intrusive debugging and architecture exploration. The availability of a virtual platform allows a good visibility of the hardware models and their interaction with each other as well as valuable information about software execution and its impact on the overall system performance. Thus, HW/SW co-simulation speeds up the design process and is fundamental for a successful HW/SW co-design methodology as it enables the early validation of both functional and non-functional properties of the simulated system.

Fig. 2.1 shows a simplified HW/SW co-simulation environment. The virtual platform (right side of fig. 2.1), which we will alternatively refer to as simulation platform, is composed of models of the hardware components found in the target architecture (left side of fig. 2.1) such as, the memory system, the core (EU: Execution Unit in the simulation platform), the peripherals, etc. These components communicate with each other via a communication medium (e.g. bus or NoC), which is represented as an abstract interconnect model

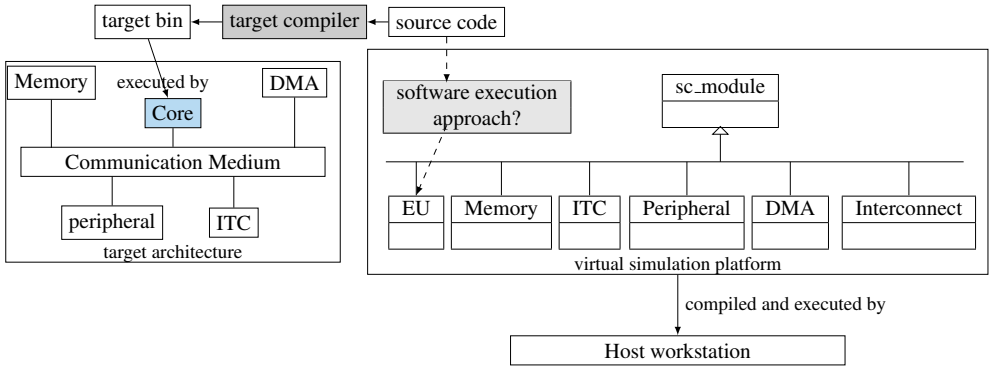


Figure 2.1: A simplified co-simulation platform

in the virtual platform. Modeling these hardware components is performed using HDLs (e.g. SystemC), which provide different levels of precisions ranging from RTL to TLM. As for the software that is intended to run on the target machine, it is executed by the EU in the simulation platform according to a chosen software execution approach (ISS, native simulation, etc.). Once fully developed, the simulation platform is compiled and executed on the host machine (e.g. X86 desktop machine) as a software application.

2.2 Software Execution Approaches in a Virtual Platform

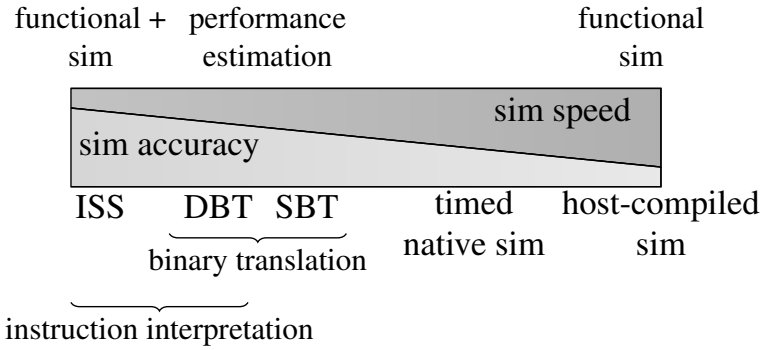


Figure 2.2: Rough classification of the different abstraction levels of software simulation (adapted from [PFG+11])

Given the key role played by co-simulation in the design process of embedded systems, increasing the simulation performance has been the focal point of HW/SW co-design research, which led to the emergence of different abstraction levels of hardware modeling and various approaches of software execution on top of these hardware models.

Different strategies with various abstraction levels [PFG+11] (fig. 2.2) have been proposed to deal with software execution on top of virtual platforms. These propositions differ in their simulation speed and their simulation accuracy and they can be categorized into instruction interpretation methods and native execution methods [PFG+11].

In fig. 2.2, the simulation speed decreases from right to left while the simulation accuracy increases from right to left. Software interpretation approaches depend on the target architecture as they simulate, more or less, internal low level details of processor operations, such as dependencies between intsructions, pipeline stages, instruction latencies and delay slots,

etc., which makes them suitable for evaluating both the functionality and the performance of the system. However, each additional detail implies more simulation time.

On the other end of the spectrum, host-compiled simulation (a.k.a. native simulation) is independent of the target architecture, which makes this simulation approach limited to functional verification. Efforts have been made in order to upgrade host-compiled simulation approaches from simple functional simulators to performance estimation tools as well (e.g. timed native simulation).

2.2.1 Interpretive Simulation Techniques

The basic idea behind software interpretation is to transform instructions of the cross-compiled target binary code into instructions of the host processor.

Instruction Set Simulation

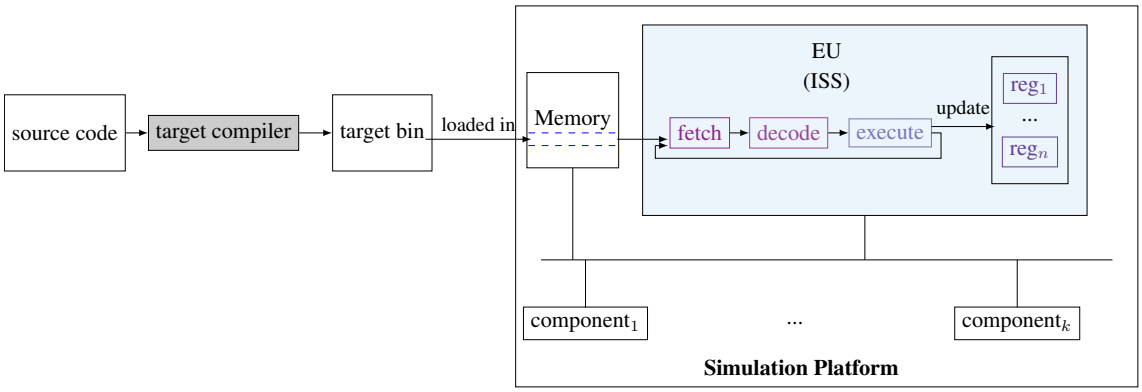


Figure 2.3: Overview of an ISS platform

One of the most mature interpretive simulation techniques is Instruction Set Simulation (ISS), which incorporates a detailed model of the target processor micro-architecture in order to closely mimic its behavior and yield accurate performance estimations. ISS allows the execution of software at a detailed instruction level and simulates the target machine state for each interpreted instruction. Such low-level interpretive simulation approach can be very accurate but extremely slow.

Usually virtual prototypes simulate the behavior of the target processor by performing computations using an ISS (SimpleScalar [ALE02], gem5 [BBB+11] formerly M5 [BDH+06]). As shown in fig. 2.3, target instructions are loaded in the memory component model. The execution of software is organized in an infinite loop. During each iteration of this loop, the processor model sequentially fetches the next instruction from the memory model (possibly from the instruction cache if memory hierarchy is modeled), decodes the fetched instruction and performs the operations (e.g. load, store, arithmetic and logical operations) according to the semantics of the instruction while updating the simulated registers and memory.

The amount of functional and non-functional features of many-core SoCs is too extensive to be verified and validated by ISS as the simulation would take a considerable amount of time. To improve the simulation performance, a simulation technique known as *compiled simulation* works at a coarser granularity by translating a block of instructions instead of a single instruction at a time. Two approaches of compiled simulation can be distinguished: Dynamic Binary Translation (DBT) and Static Binary Translation (SBT), which is not based on instruction interpretation.

Dynamic Binary Translation

Just like the previously mentioned interpretive simulation technique, the key idea of dynamic binary translation is to translate the target code into a behaviorally-equivalent code that is executable on the host machine running the simulation platform.

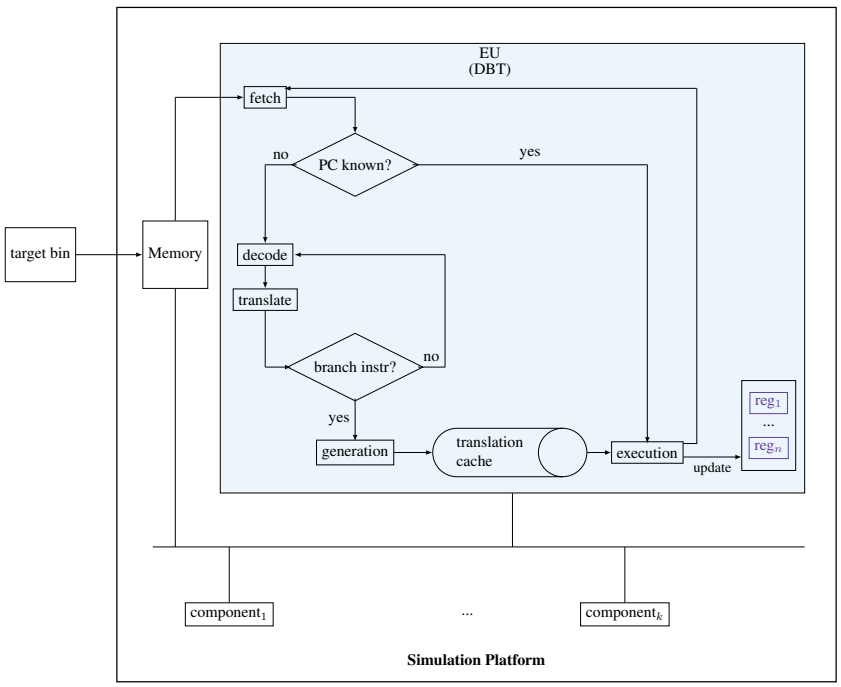


Figure 2.4: Overview of a simulation platform based on DBT

As illustrated in fig. 2.4, the target binary is decomposed into blocks, called translation blocks (TB) in the DBT terminology. A TB starts at the current PC and ends with a branch instruction. The most intuitive way of dynamic translation is to directly find the equivalent of the target instructions in the host ISA (Instruction Set Architecture). This implicates that for each new target and/or host machine, a new translator should be devised. For a retargetable dynamic translator, an additional step is performed. This step consists of translating the target binary into an intermediate representation IR that is independent of both the host and target processors (QEMU, [CM96]). The IR is then translated to a host code (*generation* step in fig. 2.4).

As shown in fig. 2.2, DBT is placed after ISS and before SBT in the simulation-speed slow-fast continuum, but dynamic translation requires a strenuous development effort, which might not be the perfect solution for MPSoC simulation at early design stages.

2.2.2 Static Binary Translation

SBT is a simulation technique that eases the burden of instruction interpretation by moving the frequent decoding task from run time to compile time. The idea behind SBT is to translate the complete target binary code to a functionally-equivalent intermediate representation before simulation time. Thus, the decode stage is separated from the fetch and execute stages. The intermediate representation is then compiled and executed on the host machine. At run-time, the EU executes the decoded and translated instructions without any translation overhead.

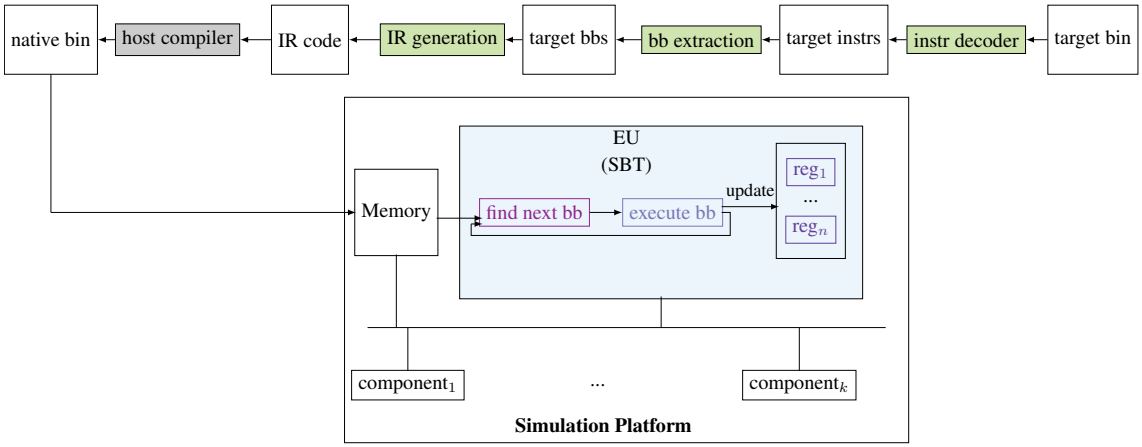


Figure 2.5: Overview of a simulation platform based on SBT

Fig. 2.5 showcases the different translation steps. First, the target instructions are decoded and the corresponding information (instruction type, operands, etc.) is stored in memory objects for later use in translation. Then, the target instructions are scanned for basic block (*bb*) extraction. A basic block is detected when a statically known branch target instruction is encountered or after the exit point (branch/jump) of a previous basic block. After target basic block construction, target instructions are converted into an *Intermediate Representation* (IR) that is independent of both host and target machine architectures. The IR is then compiled to a host binary code. At simulation time, the EU fetches the next basic block, which is already translated, and executes its behavior.

Alhtough faster than ISS, SBT has its own set of limitations, which are related to certain aspects that cannot be translated statically, such as self-modifying code and indirect branches.

2.2.3 Native Simulation

On the other extreme of the simulation speed continuum (fig. 2.2), there is native simulation (a.k.a. source-level simulation or host-compiled simulation). This simulation approach is much faster than the interpretive methods that we briefly described (ISS,DBT) because it eschews instruction decoding and interpretation by directly compiling and executing the software on the host machine.

The first endeavors to natively simulate the target software appeared in [GCM92], [GYNJ01], [CHB09b]. These primitive approaches consist of encapsulating the software in a bare-metal hardware module. These initial proposals are simple but suffer from limited parallelism. Moreover, since the software tasks are executed in the context of a hardware module, all software data allocations are made in the simulator address space instead of the simulated target memory, therefore data allocated by software is inaccessible by the target platform components.

To enable the execution of a more complex software and to support concurrency, a high-level model of a light-weight operating system can also be encapsulated in a hardware module of the simulation platform [GYG03], [MPC04] (fig. 2.6). The OS model implements services such as Inter-Process Communication (IPC), task creation, event handling, delay modeling, interrupt handling, etc., but it remains very restricted and lacks many details (such as library calls including memory management routines). As for the scheduling, the OS model

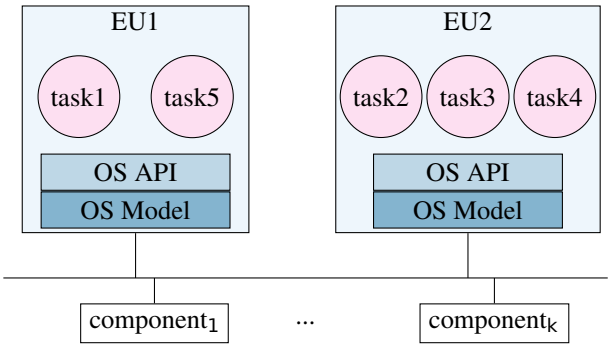


Figure 2.6: Software encapsulation in a hardware module equipped with an OS model

relies on the primitives offered by the simulator scheduler rather than implementing the actual OS scheduler.

The common limitation of software encapsulation is that software is highly tied to its encompassing hardware module and uses its underlying communication interface to interact with the other software tasks that reside in other hardware modules. Also, dynamic task creation and migration are not supported.

Hybrid approaches where specific portions of the software are executed using an ISS, while other portions are executed natively, emerged [MRRJ05], [KGW+07]. The challenges of such approaches lie in the selection process (which part of the software should be executed by which simulator?), the definition of the execution context and the synchronization between the two simulation environments, while keeping the run-time overhead caused by switching between the two simulators in check.

To maximize the amount of software that could be natively simulated, without the help of an ISS and independently of hardware modules but with the possibility to interact with the event-driven simulation environment, layering the software into different abstraction levels starting from the Hardware Abstraction Layer (HAL) to the functional abstraction layer is a requisite. Thus, native execution of the target code on the virtual platform is achievable, using specific software APIs to interact with the simulation models of the target hardware components.

As depicted in fig. 2.7, the software stack consists of a high-level application, standard libraries , the OS, and a HAL [YJ03]. In addition, the software relies on two different APIs: the hardware-dependent software API (HDS), which offers OS services, and the HAL, which provides the processor subsystem services to the HDS. The number of software layers is determined according to the adopted software simulation approach. In case of target code interpretation, no abstraction is needed as the software is loaded in its entirety (all the software layers) in the memory. In native simulation, the execution unit (EU) uses software APIs to abstract either the OS layer [TRKA07], [BBY+05] or the HAL [BYJ04], [YBB+03] (fig. 2.7). Abstracting the OS layer is a tedious task because it requires the implementation of all the OS and library (Math, C, Communication, etc.) functionalities by the EU.

Our simulation approach relies on the definition of a thin HAL used for all hardware-related accesses [Sar16]. The declaration of HAL functions are present in the software stack but the implementation of these functions are made in the hardware platform, namely in the EU. This results in a hardware-independent software stack that only interacts with the hardware when a HAL API function is solicited.

Although very fast and suitable for early DSE, native simulation is not free from chal-

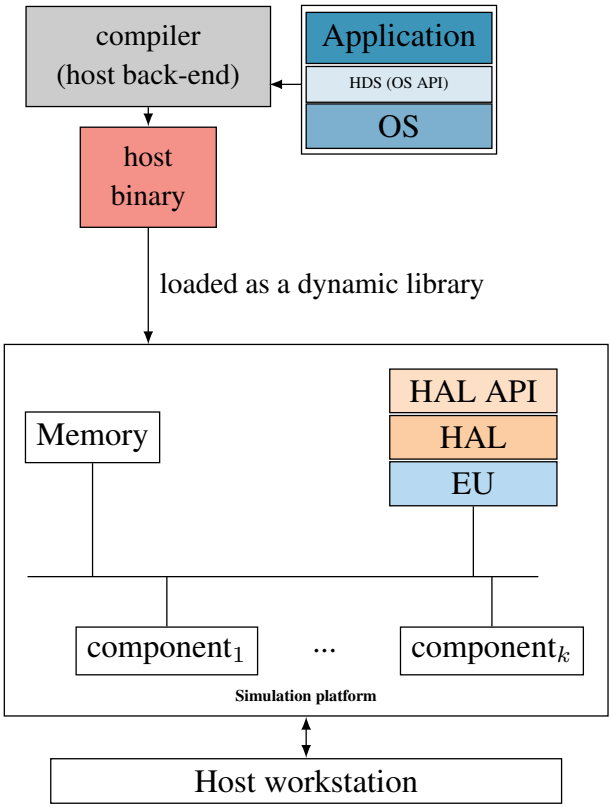


Figure 2.7: Overview of a native simulation platform

lenges. One of the main issues is the different host and guest address spaces. The software, on the one hand, is compiled for the host machine and loaded into the host memory as a dynamic library. So, the software is only aware of the host address space. The hardware platform, on the other hand, simulates the target hardware components whose addresses are predefined by the target system designers. Thus, the hardware models are only aware of the target address space. The presence of these two different and possibly conflicting or overlapping address spaces is emphasized when the software wants to access a hard-coded address of a hardware component or, in the opposite case, when a hardware component (such as a DMA) wants to access an address of a dynamically allocated variable by the natively-compiled software. This heterogeneity between address spaces prohibits certain interactions between the hardware models and the software.

Moreover, since the natively-simulated software is completely dissociated from the target ISA and executes unknowingly of the virtual hardware components thanks to the abstraction layer, the simulation is bereft of accuracy. In fact, the simulation accuracy depends on both the correctness of the functional model and the precision of the performance model. The functional model simulates the functionality of the target software and it can be represented at the source level, the intermediate level (IL) or the binary level.

The simulation of the functional representation on the host machine does not provide any insight on the performance of the software, such as its temporal behavior (i.e. the computation delays caused by the target platform). The software is executed in the context of the EU, which is a simulation thread, in zero time. As we mentioned above, the only time the software interacts with the virtual platform is when a HAL API function is invoked. In between two HAL functions the notion of time does not exist. The abscence of non-functional

information restricts native simulation to functional verification.

Target-specific performance metrics depend on the target ISA, the effects of the target processor micro-architecture and the effects of compiler optimizations. To be able to carry out performance estimation in a native simulation platform, a performance model is indispensable. It should take into account the important target effects on the performance of the system and provide the corresponding performance estimates. This performance model is coupled with the functional model in order to generate a high-level simulation model capable of running on the host processor, while providing performance estimates of the target processor. This coupling is carried out by *back-annotating* the functional model with the results obtained by the performance model. However, the abstraction of the target processor details and the difference between the host-compiled functional model and the target binary get in the way of accurate performance estimation.

The difference between the target binary code and the host-compiled code, even though they are both generated from the same software, is not only caused by the possibly different ISAs but especially by compiler optimizations, which may have different effects on the structure of the code depending on the platform architecture. In addition, micro-architectural components that have no impact on the functional behavior of the system are usually dismissed in order to alleviate the modeling effort and speedup the simulation. Instead, host machine resources are leveraged. For instance, using directly the memory hierarchy of the host computer during the simulation does not change the functionality of the software but may yield inaccurate performance estimates because these estimates are highly influenced by the dynamic behavior of these micro-architectural components and their interaction with software. For example, the miss ratio may be different from one platform to another as it depends, among other factors, on cache memory configuration. Architectural features such as VLIW, which is common in MPSoC systems but not as much in desktop machines, is another example of target characteristics that affect the execution time but does not change the functional behavior of software.

2.3 Hardware Simulation: Abstraction Levels of Virtual Prototyping

In a full-scale simulation platform, the target software is executed on top of a virtual hardware platform. A virtual platform is a collection of models of the target hardware components including its processors, peripherals and interconnect mechanism (fig. 2.1). These models can be described at different abstraction levels, which enables progressive refinement of a given specification to the final implementation.

Hardware description languages (HDLs) allow us to model hardware components; their parallel semantics, their structural and timing behavior and their communication interfaces at different abstraction levels. Each level allows the simulation of full or part of the system with a certain degree of architectural and timing accuracy. Thus, depending on the design stage, designers are able to choose the most suitable abstraction level. Such a choice is always a game of balancing the trade-off between performance and accuracy of a potential simulation model.

The two extreme ends of the SoC design flow, as portrayed by fig. 2.8, are: the system-level at the highest end of the flow and the circuit level at the other end of the flow. We will discuss the top levels of abstraction.

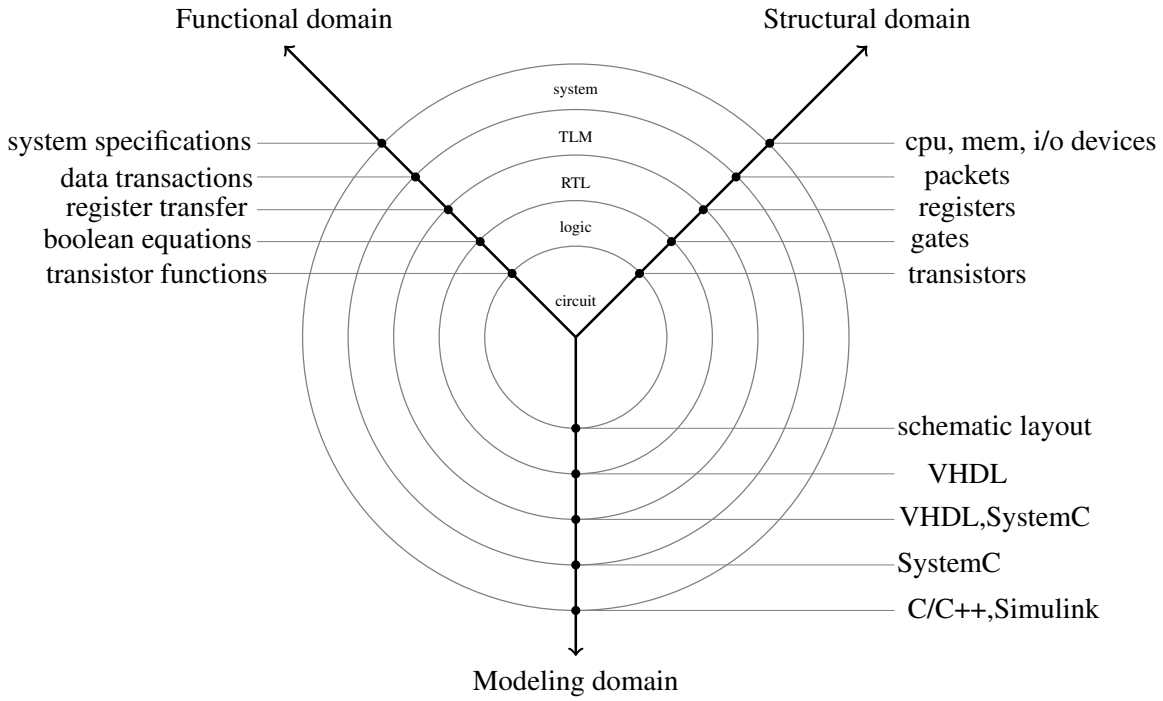


Figure 2.8: Y diagram of hardware simulation abstraction levels (adapted from the Gajski-Kuhn Y-chart)

Register Transfer Level (RTL)

RTL is a very low-level design abstraction where the circuit's behavior is defined in terms of data transfer or the flow of signals between hardware registers and the operations performed on these signals. RTL models written in hardware description languages, such as VHDL and Verilog, describe explicitly the registers that define the internal state of the target system components. The internal architecture described in a synthesizable RTL model is rigorously identical to the real hardware. A RTL description has an explicit clock. All operations are scheduled to occur in specific clock cycles. A RTL description is usually converted to a *gate-level* description of the circuit by a logic synthesis tool. Placement and routing tools are then applied on the synthesis results to create a physical layout.

Evidently, the benefit of RTL simulation is its *fidelity* to the real hardware implementation allowing accurate functional and performance analysis of the SoC. However, the lengthy RTL simulation time and development phase, that keep getting worse lately due to the high SoC complexity, is a price too expensive to pay. It is, thus, too long to wait for the development of RTL hardware models before the HW/SW co-verification can start. Moreover, at this level of the design flow, the breadboard is almost ready, which makes any system modification very costly. Thus, raising the abstraction level above RTL was deemed necessary for early design space exploration and co-verification.

Cycle Accurate Level

The cycle accurate level, which is not represented in fig. 2.8, and the RTL are usually used to designate the same level of abstraction. The CA level , however, is a little faster than the RTL one. Although it provides accurate description of the data transfer on the components interfaces at each clock cycle and it implements the same communication protocol as in

real hardware (all of the signals are represented exactly as they are in a real platform), a CA model can use a representation at a higher level of abstraction than the real hardware realization. Thus, a CA representation cannot be directly synthesized.

The simulation speed of a cycle accurate model is still too slow to run a significant amount of software and the development cost is also too high to make up for the inconsequential benefits of cycle-accurate models (merely an order of magnitude faster than the equivalent RTL models).

System Level

On the other end of the *Y* diagram, we find the most abstract level: the system level (also referred to as the algorithmic level). This level offers high performance, but is purely functional and could be implemented using SystemC, Simulink or even *C/C++*. Starting with initial requirements, designers can model such specifications in highly abstracted descriptions leading to a well defined executable specification model that will serve as a reference. This reference models neither a clock nor platform specific details. It is an untimed functional model with only causal ordering between tasks. Since this abstraction level does not include any information about the target architecture or its temporal behavior, it is only used to verify certain functional properties of the software application. As a result, an intermediate modeling level that offers a reasonable simulation speed, is detailed enough to run the embedded software stack and to provide accurate results, and is relatively quick to develop with a considerably lightweight modeling effort, is required.

Transaction Level Modeling TLM

TLM has been put forward as the best bid to balance the trade-off between simulation accuracy and speed. TLM is a transaction-based abstraction level founded on object oriented programming languages such as *C++* and considered as the doorway to early MPSoC exploration. TLM resides between the cycle accurate model and the untimed algorithmic model (a.k.a. system level) in the design flow. In order to facilitate the integration and compatibility of transaction-level models, the industry-wide standard, a.k.a. SystemC, is used to develop such models. The Open SystemC Initiative (OSCI) [OSC] was the first contributor to the creation of TLM standards.

TLM focuses on communication abstraction to improve the simulation performance. Communication architectures are modeled as channels that provide interfaces to functional units. Thus, signal-based communication interfaces are replaced with transaction-level function calls. In this modeling style, communication and computation can be modeled separately.

The term *transaction* in the TLM context refers to the exchange of data or/and control information between two hardware component models. The term *transaction level* does not designate a single level of abstraction, but rather a continuum of levels that differ in the degree of functional or temporal details they incorporate. For instance, simulation models that aim at achieving high simulation speed, abstract the details of the communication protocols used to convey the transactions. Moreover, time modeling in communication interfaces can be ruled out. Such completely untimed models are known as *Programmer's View (PV)* models [CMMC08]. To improve the simulation accuracy, detailed communication protocols could be implemented and communication interfaces could be annotated with time information. Timed models are commonly known as *Programmer's View with Time (PVT)* or *TLM with time (TLM-T)* [HSAG10].

In this thesis, we use a native simulation platform that consists in natively compiling the software and executing the generated host binaries on top of a virtual hardware platform modeled using SystemC/TLM-T.

2.4 Conclusion and Key Questions

Native simulation of software, although it might seem simple compared to other software simulation approaches, has its own set of problems. One of these problems is the address-space differences between target and host machines. The other problem, which is the focus of this thesis, is the lack of target-specific performance information in host-compiled simulation. In order to be able to perform software performance estimation, native simulation should be supplemented with a performance model. Obtaining accurate performance estimates relies mainly on two key factors:

- The accurate coupling of the performance model with the functional model.
- The accuracy of the performance model.

The first factor depends on the level of abstraction of the functional model:

1. Which software representation (source code, compiler intermediate representation or target binary code) is the most suitable for fast and accurate performance estimation?

If a high-level representation is selected as a functional model, the difference between its structure and the binary structure creates some challenges:

2. How to correctly insert non-functional information computed using the target binary code into the high-level code when they could have different control flow graphs (CFGs) due to compiler optimizations? In other words, how to find correspondences between the target binary code and the high-level code CFGs when:
    - common compiler optimizations are enabled (e.g. *gcc -O2*)?
    - aggressive compiler optimizations that radically change the CFG are turned on (e.g. *gcc -O3*)?

The second factor consists of the ability of the performance model to *faithfully* re-create the non-functional behavior of the target SoC:

3. How to develop a performance model that takes into consideration the target MP-SoC's micro-architectural components and their advanced features (e.g. the instruction cache and instruction buffer of a VLIW processor)?

We intend to address these key questions in the following chapters.

Chapter 3

Preliminaries and Prior Work: On Native Execution of SW on Top of a Virtual Platform

Native execution of software has become the go-to functional simulation approach at early design stages of MPSoCs thanks to its high simulation speed. Early techniques were based on the encapsulation of software in a hardware module of the virtual platform with little to no distinction between a software module and a hardware IP, which brings about unsolicited concurrency between software tasks. More advanced techniques rely on a hardware abstraction layer HAL to validate as many software layers as possible. However, both these techniques suffer from the heterogeneity of the host and target address spaces making the interactions between native software and target hardware models very tricky. Different approaches have been proposed to solve this problem, which will be explained in the first section with a special focus on the retained approach. Performance estimation is another challenge facing native simulation because of the lack of target architecture information in native software. Overcoming this challenge has led to the appearence of intelligent modeling approaches that, if correctly coupled with the functional model, help preserve the simulation accuracy. A run down of software annotation techniques for performance estimation will be presented in the second section and examples of performance models will be described in the third section.

3.1 Target vs. Host Address Spaces

The hardware components of the target platform (DMA, ADC, ITC, RAM, etc.) have hard-coded addresses, i.e. their address mappings are statically defined by the target system designers. Consequently, the address decoder in the hardware simulation platform uses these mappings to ensure correct communication between the hardware models. The software application, on the other hand, is natively compiled and loaded in the host memory as a dynamic library, which means that software addresses are resolved at runtime and are allocated in the host virtual address space. To resolve the problem of these two incoherent address spaces, two main approaches have been proposed.

3.1.1 Using a Unified Address Space

The unification of host and target address spaces was suggested in [Ger09]. The idea is to use the host address space as a uniform memory representation for both the software and the hardware models. To do so, a number of modifications are performed on the hardware models. Dynamically allocated host addresses are used for the software accessible memory resources. Each hardware component in the virtual platform should provide a list of *Symbols* and information about its *mappings* (memory region: size, base address and name). This information is leveraged dynamically during the elaboration phase of the simulation in order for the communication network model to construct the address decoding table. It no longer uses the statically known target addresses but the dynamically allocated host addresses. Not only does this approach require the modification of the hardware platform components but it also restricts the software as hard-coded addresses cannot be used and the *new* hardware addresses can only be resolved by software until the start of the simulation.

3.1.2 Using Hardware Assisted Virtualization

Authors of [SHP12] exploited hardware extensions available for the Hardware Assisted Virtualization (HAV) technology, which has been introduced a decade ago in the majority of high-performance processors (x86, SPARC, PowerPC, ARM) to provide a second set of page tables, in order to solve the problem of the two address spaces in the context of native simulation. While researchers have attempted to solve this problem by merging the host and target address spaces into a joint address space used for the software stack (application and embedded OS), the simulated hardware components and the host system, Shen et al. proposed to keep these two address spaces separate in order to avoid any kind of conflict or overlap. They rely on a hardware-defined address translation layer based on the HAV feature of the host processor. Using this layer makes address translation completely transparent to the entire software stack.

The HAV technology provides a new operating mode, called *guest mode* (fig. 3.1), where a new address space, called *guest address space*, can be fully customized to cater for the target address space. The address spaces of both the memory as well as the memory-mapped input/output (MMIO) modeled devices can stay the same as the ones of the target platform. This way, the host and target address spaces can be kept intact thanks to the two distinct execution contexts of the host machine: user space and target space.

In [SHP12], Shen et al. used the HAV technology by integrating a hypervisor (or a Virtual Machine Monitor (VMM)), more precisely the Linux hypervisor called Kernel Virtual Machine (KVM), in an event-driven transaction level simulation environment. KVM is constituted of a linux kernel driver used in kernel mode and a library used in user mode. The KVM library is encapsulated in the SystemC environment in user mode. This library allows the instantiation and the configuration of a VMM that is in charge of its associated Virtual Machine(s) (VM) (fig. 3.1). It also enables interactions between user mode and guest mode.

At the begining of the simulation, a memory zone is allocated in the virtual memory in user mode and is passed to KVM. This memory zone corresponds to the physical memory in guest mode where the host-compiled software will be loaded and then executed by a VM. Each EU is equipped with an interface with KVM allowing it to control its corresponding VM. Simulating a core consists in executing its dedicated VM. A VM reads the program instructions loaded in guest memory space and executes these instructions natively in the host processor guest mode. Once a VM starts executing the host binary, it does not stop unless the guest mode has to be exited for a particular reason (synchronization time is reached,

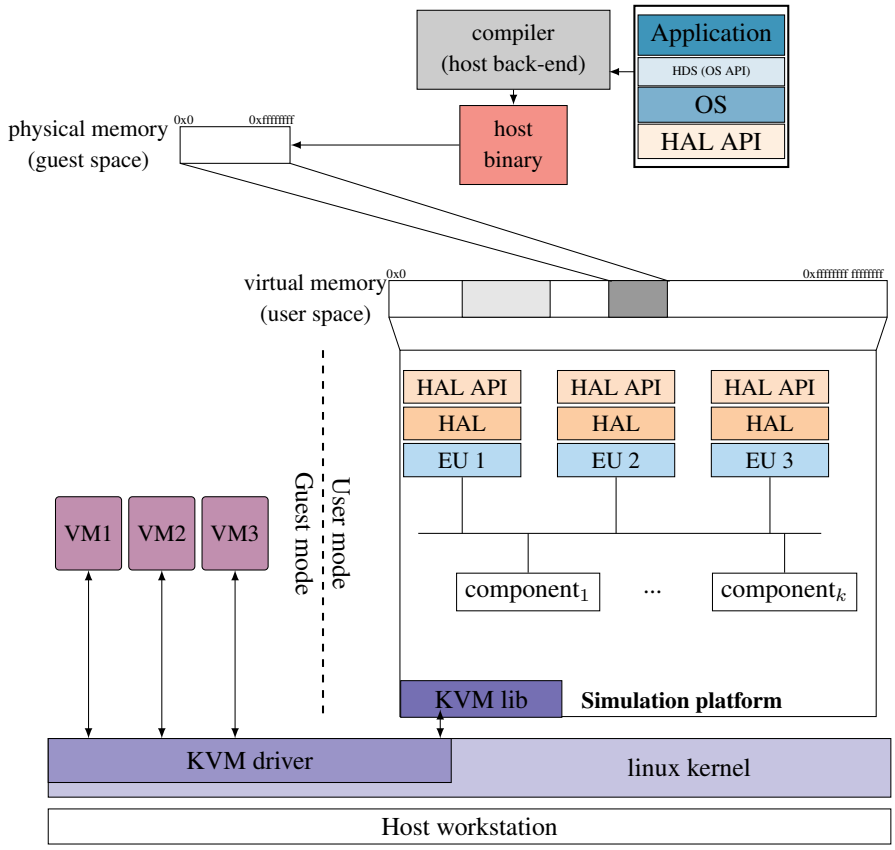


Figure 3.1: Native simulation using HAV

annotation function, page fault caused by a memory access to an address outside the previously allocated memory zone, access to a memory-mapped input/output component, etc.). The KVM driver examines the reason behind exiting the guest mode and handles it, if possible (e.g. page fault), in kernel mode. Otherwise, the host processor switches to user mode and it is up to the simulator to handle the exit reason. After the exit reason is handled, the host processor returns to the guest mode via the KVM kernel driver.

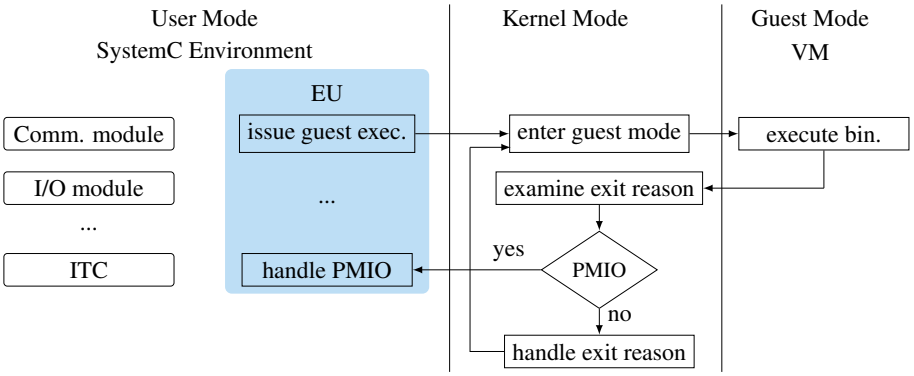


Figure 3.2: User, guest, kernel transition flow in case of a PMIO request in HAV-based native simulation

In this thesis, we use a HAV-based native simulation platform developed by Shen et

al. [SHP12] and revamped by Sarrazin [Sar16] because it is characterized by a fast simulation speed and it efficiently handles the problem of the two address spaces. We will leverage this platform to conduct performance estimation. Thus, we will only describe the underlying support provided by this platform to enable performance estimation. Functional aspects such as timer handling, interrupt handling, memory mapping, I/O accesses, etc., will not be detailed and we assume a correct functional simulation.

Time annotations, among other target performance metrics, are inserted in the software, which is executed by the VM in guest mode. This time information is used to advance the simulated time, which requires the use of SystemC timing mechanism. This entails a communication mechanism between the code executed by the VM and the simulation platform in user mode. So, to exit the guest mode, the annotation function used in the software stack is mapped to an I/O port and the timing data is sent to a port-mapped I/O (PMIO). This PMIO request (fig. 3.2) shifts the control to the EU in the SystemC environment in user mode, which uses the time information to update the simulated time by calling the SystemC *wait()* function. When the requested amount of time is consumed by the core model, control is given back to the target code in guest mode.

The transition from guest mode to kernel mode and then to user mode (in case of a MMIO or PMIO request for instance) takes thousands of cycles. On top of that, handling the exit reason by linux kernel, choosing the SystemC component concerned with the request and finally simulating the request, takes additional time. This slows down the simulation time significantly. To reduce this overhead, Sarrazin et al. [Sar16] proposed an approach to handle the I/O requests in guest mode as much as possible, thus avoiding mode switches. This necessitated the extension of the HAL in the software stack as well as the addition of a memory region in guest mode.

3.2 Software Annotation for Performance Estimation

Research regarding software simulation is focused on enhancing the simulation speed by setting aside low level details of the target architecture and using the host machine resources instead. Raising the abstraction level has given rise to fast simulation techniques such as SBT, DBT and native simulation. However, the higher the abstraction level is the more prominent the loss of accuracy becomes. Obtaining accurate performance estimation results, such as an approximation of the execution time of software, using native simulation is not a trivial task. Efforts have been made to enhance the accuracy of the results by introducing target-specific metrics into the natively executed functional model. This technique is known as software annotation ([CHB09a], [LLT10], [MSVSL08a], etc.). Obtaining precise estimates depends on both the functional model, where the annotations will be inserted, and the accuracy of the annotations themselves. First, it is necessary to decide at which stage of the software compilation process the information is back-annotated. There are three possibilities: in the original source code, in the host binary code, or in the compiler intermediate representation (IR). In the current section, we will give a rundown of the three abstraction levels (source-level, intermediate-representation level and binary-level) used in literature for the functional model, as illustrated in Table 3.1. We will highlight the challenges that come with each level and explain our inclination for one specific level of abstration.

Table 3.1: Classification of previous works according to the adopted functional model

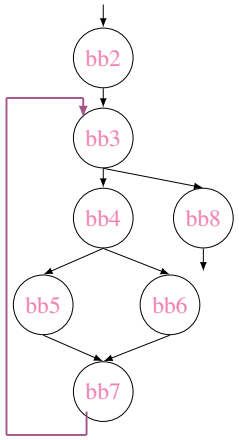
Source-level simulation (SLS)	Intermediate-level simulation (ILS)	Binary-level simulation (BLS)
[LLT10]	[ZH09]	[LBH+00]
[Wan10]	[GCK12]	[PWH12]
[WH12]	[GCZ13]	[ZM96]
[LMGS12]	[BGP09]	
[MGLS11]		
[SBR11b]		
[SBR11a]		

3.2.1 Source-Level Simulation (SLS)

To obtain performance estimates of the target platform, while natively executing a host-compiled functional model, a feasible solution is to insert pre-estimated target-specific performance metrics directly into the source code (e.g. written in *C* language). Annotating at the source code level is referred to as Source Level Simulation (SLS). This approach is abundantly adopted in performance estimation techniques for its high level of abstraction and the simplicity gained from working with a source code.

```
int odd=0;
int even=0;
for(i=N−1;i>=0;i−−){
   if(T[i]%2)
      odd++;
   else
      even++; }
printf("odd=%d",odd);
```

(a) source code



(b) source-level CFG

Figure 3.3: An example of a source-level CFG

The general approach is to extract non-functional information from the target binary code, usually at a basic block level granularity, and back-annotate it into the source code from which the binary was generated. A basic block is a maximal sequence of consecutive instructions that the program control flow can only enter at its first instruction and leave at its last instruction, i.e. there is no possibility of branching in a basic block except at its last instruction. A basic block is considered as the ideal estimation unit because it is the smallest unit used by the compiler for optimization purposes. Moreover, the program is usually represented by a control flow graph (CFG), a directed graph, where the nodes V are the basic blocks of the program and each directed edge indicates the flow of control between the nodes (fig. 3.3-(b)). $CFG = (V, E), E \subseteq V \times V$. In fig. 3.3, a simple C code and its CFG are delineated. A CFG has a unique entry node from which the execution of a program starts.

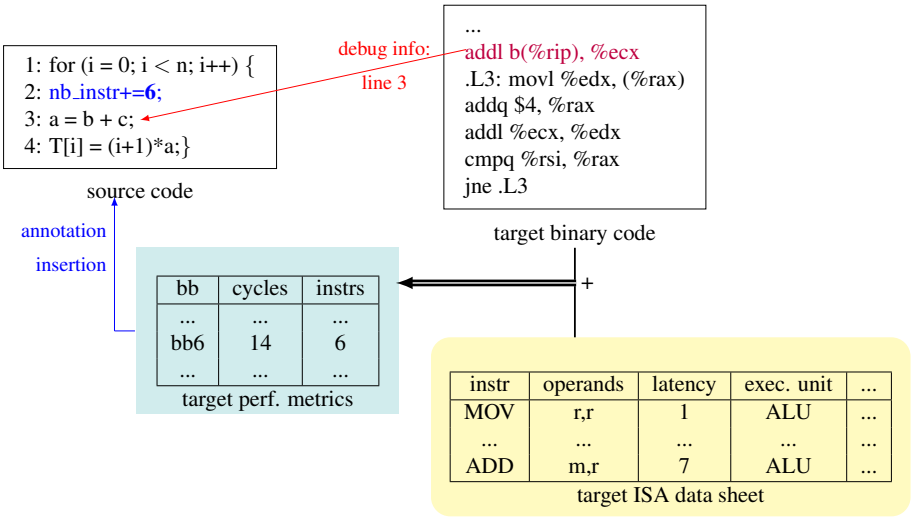


Figure 3.4: Source code annotation

As shown in fig. 3.4, performance metrics (e.g. number of instructions and number of cycles) are statically extracted from the target binary with reference to the target ISA data sheet. A more elaborate analysis would also include a detailed study of the pipeline and the effect of the other micro-architectural components in order to accurately compute the number of cycles. After this non-functional information is determined, it has to be inserted at the right place in the source code (i.e. the source code portion that corresponds to the binary basic block from which this information was extracted). Accurate placement of annotations requires a mapping between the source code and the binary code. It goes without saying that finding correspondences between both codes is very difficult because of all the compiler optimizations. These optimizations introduce many transformations to the original source code leading in all but the most simple cases to a binary code with a control flow graph (CFG) different from the original one. As a result, a one-to-one correspondence between the source code CFG and the binary code CFG is typically broken by compiler optimizations, while establishing a precise mapping between the two CFGs is crucial to inject the annotations at the right place in the high-level code and to obtain accurate performance estimates.

The simple example in fig. 3.4 showcases a compiler optimization called *loop-invariant code motion*. The value of variable a in line 3 of the source code does not change with each iteration of the loop. So, the addition instruction should be performed only once and thus it would be logical to hoist it outside of the loop body. In the machine code, the effect of this compiler optimization is obvious (red instruction in machine code corresponds to line 3 in the source code, but it no longer resides in the loop body). This optimization, among many others, makes it hard to associate binary basic blocks to their source code counterparts. To address this issue, mapping binary code to source code approaches have been propounded.

In [LLT10], the source code is represented by a control flow graph. Each node of the CFG, i.e. basic block, is considered as the estimation unit where timing information is placed. The source code is cross-compiled and the generated target binary code is analyzed. The cycle count of each target binary basic block is estimated statically and annotated in the corresponding source code basic block. This static analysis is coupled with run-time corrections once the actual execution path of the program is known. The authors claim that the source and binary CFGs are identical, even though they use an optimized binary code,

and they conduct the mapping between the two CFGs using "compiler pragmas or symbols" without giving any further explanation.

In [Wan10], a Source code instrumentation based Simulation (*SciSim*) is proposed and more details about the mapping process using debug information is given. The authors extract performance metrics from the binary code at a basic block level and insert the annotations of each basic block before its corresponding source line. For complex C statements like *loop* constructs for instance, they devised special instrumentation rules that define the placement of the annotation code, i.e. before or/and after the corresponding source line, with respect to the different loop parts (loop count initialization, loop condition and loop count update). Other complex C constructs like *While loop*, *Do-While loop* and a function's prologue and epilogue have their own set of instrumentation rules. However, these rules are tightly dependent on the syntax of the C programming language.

line		@1	@2
3	—>	0x13360	0x13360
4	—>	0x13364	0x13364
5	—>	0x13368	0x1337c

Figure 3.5: Mapping information

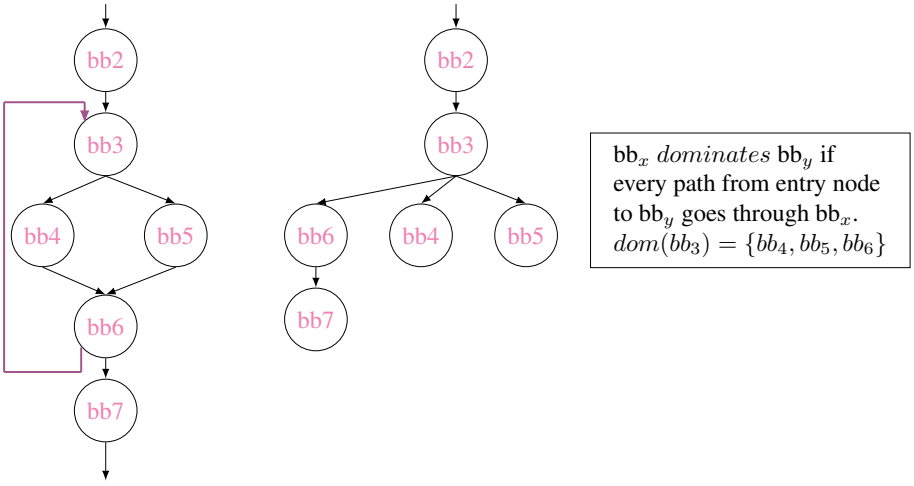
As for the mapping between binary code basic blocks and source code statements, it is established with the help of debug information dumpded automatically during cross-compilation. Fig. 3.5 shows some mapping information excerpted from the DWARF's line table, which describes the correspondence between source lines and machine code instructions. According to fig. 3.5, the entry "$5 \longrightarrow 0x13368 - 0x1337c$" indicates that the instructions stored in memory region "$0x13368 - 0x1337c$" are produced from line 5 in the source code. It is a seemingly efficient and elegant way to conduct an accurate mapping between source and binary codes, however, when compiler optimizations are enabled (e.g. *gcc -O2/-O3/-Os*), debug information might fail in providing correct mapping information leading to estimation errors. Sometimes, even with correct debug information, if the annotations are straightforwardly inserted in the source code according to the DWARF line table, estimation errors will occur. This problem is depicted in fig. 3.4. Although debug information denotes that the hoisted machine instruction (in red) corresponds to line 3 in source code, which is correct, performance metrics related to this instruction should not be inserted inside the loop body. Thus, the number of instructions (for example) inside the loop should exclude the hoisted instruction ($2 : nbinstr+ = 5$). If this particularity is not taken into account and we only rely on debug information, although correct, the number of executed instructions after natively simulating the software will be $nb\_hoisted \times nb\_loop\_itr = 1 \times 100$ (for example) times over what they should be.

Unsurprisingly, *SciSim* [Wan10] led to large mapping errors when used for optimized (*gcc -O2*) code, especially when control flows were drastically changed by compiler optimizations. Consequently, the authors of [Wan10] admitted that *SciSim* is restricted to performance estimation of unoptimized (*gcc -O0*) software.

The limited usability of *SciSim* motivated Zhonglei et al. to propose a new mapping approach that takes into consideration compiler optimizations. So, in their work [WH12], the idea behind the mapping consists of pinpointing loops in binary and source codes and attributing levels to these loops. These loops are scrutinized in order to find out the effects of compiler optimizations. In case the compiler did not alter the structure of the code and only performed code motion optimizations, mapping problems between source code and binary

code caused by these optimizations are addressed using a method called fine-grained flow mapping, which also relies on debug information as in [Wan10], but with consideration to these code motions. In case of aggressive compiler optimizations, flow mapping fails to match source code portions to their equivalent binary basic blocks because of the heavily modified code structure (which is usually the case with loops). So, their fall-back solution is to identify the highly-altered loops of the source code and replace them with their optimized IR-level counterparts. The result is a mixture of the original source code and the optimized IR code. However, the IR structure is not always identical to the binary structure, as will be explained in subsection 3.2.2, which leads us to believe that only replacing source code with IR code is not sufficient in obtaining a CFG identical to the binary one. Further mapping efforts are needed to account for the dissimilarities between IR loops and binary loops caused by aggressive compiler optimizations.

A control flow mapping algorithm, based on the analysis of loop and control dependency properties of source code and binary code, is presented in [MGLS11]. Line references provided by debug information are used, along with the analysis information, to match branch edges and loops in the source code CFG and binary CFG. Standard transformations activated at *gcc -O2* optimization level, like *while to repeat loop transformation* and *function inlining*, are reflected back into the source code CFG when no matching is found using line references. *Partial loop unrolling* is not supported.



(a) bin-level CFG (b) dominator tree

Figure 3.6: Dominator relation

In [LMGS12], the CFGs of the source code and binary code are matched using the dominance principle (fig. 3.6), which is bound to fail as stated by the authors, due to compiler optimizations. So, the source and binary codes are divided into sub-graphs of loop regions and branch regions. Each binary subgraph is matched to its source code counterpart in a top-down manner by matching their root nodes using debug information, which is yet again prone to errors. The problem is that due to compiler optimizations, there is no guarantee that the number of regions in the source code equals those of the binary code (in case of aggressive compiler optimizations, such as *complete loop unrolling*, the number of loops of the binary code becomes less than the one of the source code) or that the dominance principle still holds, which may cause the matching process not to go as smoothly as described by the authors. In case no proper mapping is found between two regions, which is mainly

caused by loop optimizations, the authors resort to worst case estimation and the result is annotated outside the source code loop.

In [SBR11b], debug information is used to relate the source code and the binary code. Markers are inserted in the source code to indicate which portions correspond to which binary basic blocks. The binary-level control flow is reconstructed at the source level using these markers and a path simulation code generated from the binary level CFG. The authors assert that this technique of path simulation helps consider structural differences between source code and binary code caused by compiler optimizations, such as loop unrolling and function inlining. Only the case of loop unrolling where the CFG remains unaltered is briefly explained and handled through manual identification and annotation of unrolled loops. To overcome the inaccuracies caused by using debug information, the work in [SBR11a], which is a continuation of [SBR11b], is based on a technique similar to [LMGS12] that compares the execution order of source code statements and machine instructions using dominator homomorphism. This comparison is used to reconstruct and disambiguate compiler-generated debug information. The authors claim that their approach is evaluated with respect to aggressive compiler optimizations, such as loop unrolling, but they do not give any explanation on how these optimizations are managed.

Mapping the source code CFG to the binary code CFG can be very complicated when the two CFGs are dissimilar. SLS mapping techniques usually succeed with unoptimized code. However, in practice, program developers usually compile their programs while enabling compiler optimizations (at least *gcc -O2*). These optimizations can radically change the structure of the source code, which may impede the mapping process. So, using low-level functional models have been sought to account for compiler optimizations and thus facilitate the mapping.

3.2.2 Intermediate-Level Simulation (ILS)

IR-level simulation approaches use the compiler intermediate representation (IR) as a functional model where non-functional information will be back-annotated. The IR code is generated by the compiler after certain optimization passes. So, the IR encompasses several compiler optimizations. The discrepancy in the CFG structure between the source code and the binary is avoided by working on the IR code whose structure is close to the binary CFG and that allows for a simulation as fast as SLS.

Fig. 3.7 shows the intermediate representation corresponding to the same source code featured in fig. 3.4. The *invariant code motion* optimization is present in the IR. In fact, instruction $pretmp\_19 = pretmp\_2 + pretmp\_12$ in the IR, which corresponds to the loop invariant code $a = b + c$ in the C code, is placed before the loop body, i.e. basic block 4 ($bb4$) in the IR.

The presence of high-level, i.e. architecture-independent, compiler optimizations in the IR made ILS popular in the performance estimation field [ZH09], [GCK12], [BGP09], etc. An approach called *intermediate source code instrumentation based simulation (iSciSim)* is proposed in [ZH09]. The authors convert the IR to a C code called *intermediate source code (ISC)*. In fact, their annotation flow (fig. 3.8) goes through three compilation steps and the source code undergoes some modifications. The source code is cross compiled to generate an IR. The IR is then transformed to an intermediate source code (ISC) (fig. 3.8-1). The ISC is cross-compiled to generate a binary code from which debug information, as well as time measures, are extracted (fig. 3.8-2). Time information is determined prior to simulation using static pipeline analysis. This step in particular makes us question the accuracy of the

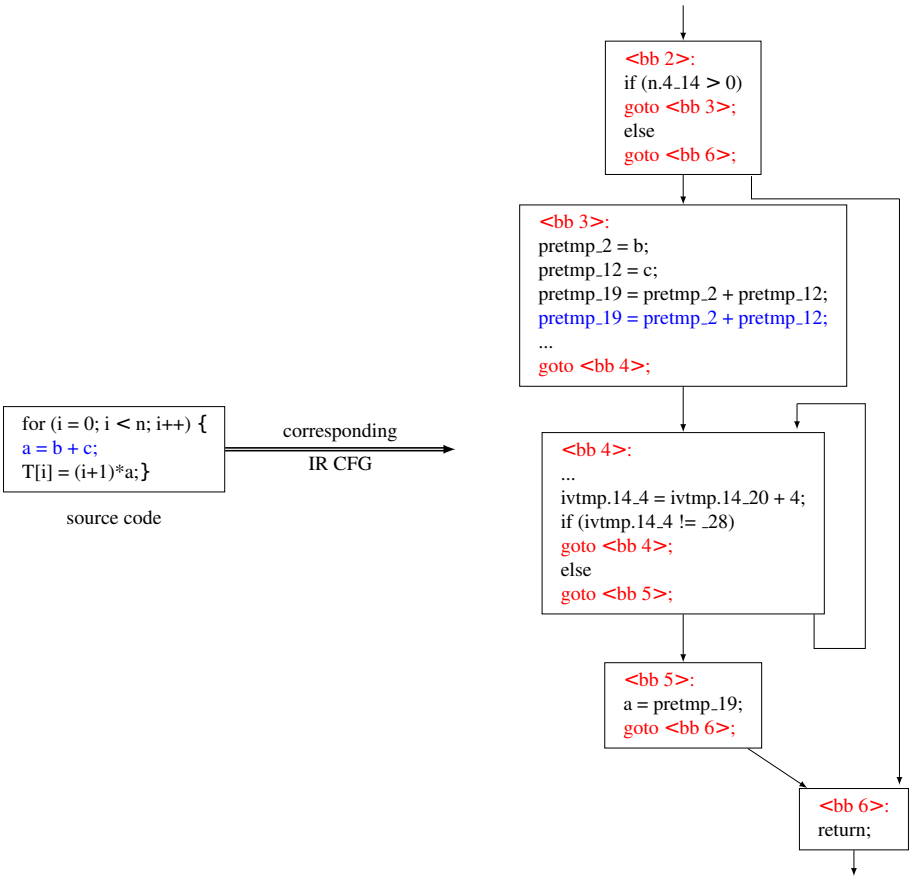


Figure 3.7: An IR example

method because the ISC is different from the original source code, which means that the binary generated from the ISC and from which they extract time information is definitely different from the one generated from the source code. Besides, the mapping between the binary code and the ISC is based on debug information, which is not reliable due to compiler optimizations. Finally, a third compilation step takes place (fig. 3.8-3) where the annotated ISC is compiled for the host machine and coupled with an on-line performance model (e.g. cache memory and branch prediction models) and then it is natively simulated.

The work in [GCK12] is similar to [ZH09] in that debug information is used to map addresses of assembly instructions to the optimized source line numbers. To obtain the desired debug information, the binary is generated from the IR instead of the original source code, which raises the same problems as before. To avoid the discrepancies caused by debug information, the authors resorted to turning off compiler optimizations. They enhanced their approach in a later work [GCZ13] by improving their mapping scheme with a more elaborate Binary-to-IR mapping algorithm using a heuristic subgraph matching scheme. However, this algorithm may lead to several possible matches for a single basic block, which they dealt with using debug information. In their approach, they especially focus on CFG structure changes caused by branch optimizations. Loop optimizations, on the other hand, have a drastic impact on the CFG structure. When they are poorly-handled, they may lead to far more erroneous estimations than "out-of-loop" mismatched blocks, in that the time spent by a program on executing a loop usually outweighs the time spent on code portions outside a loop.

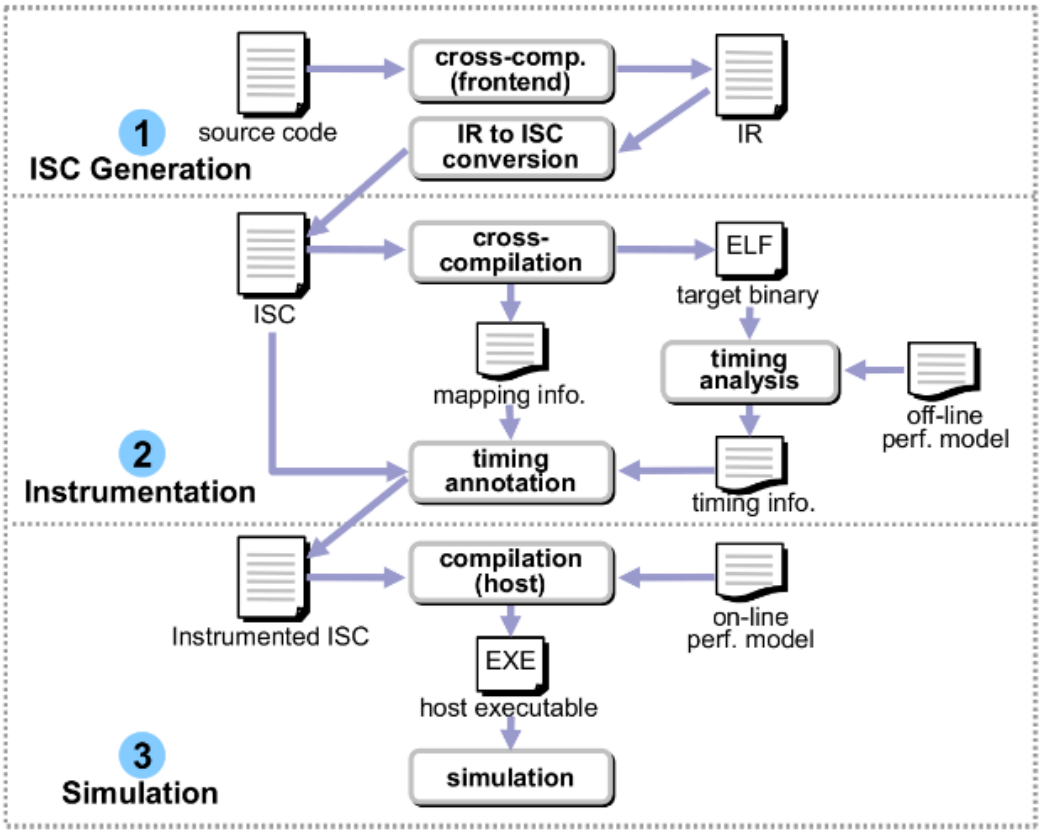


Figure 3.8: The iSciSim Approach [ZH09]

In [BGP09], the annotation scheme is based on the low level virtual machine (LLVM). The authors extend the compiler by adding passes to the back-end (fig. 3.9) in order to keep track of all compiler optimizations and reverberate them to a high-level IR that they call *cross-IR*. The distinguishing feature of the *cross-IR* is that it contains both front-end and back-end compiler optimizations. To do so, for all the target-dependent optimizations they had to find their equivalents in the LLVM's processor-independent ISA. Thus, the obtained *cross-IR* CFG is equivalent to the target binary CFG, while being independent of the target architecture. However, finding such equivalences is not a sure-fire process, which may lead to unmatching CFGs. Adding to the fact that this approach is compiler intrusive, it also relies on the target-specific instructions in order to find their match in the LLVM's target-independent ISA. This implies that whenever a new target is simulated, the matching pass needs to be changed to cater for the new architecture.

Compiler optimizations are well-known for improving the run-time performance of programs. However, for approaches that aim at finding a relation between source code statements and optimized binary code, these optimizations are perceived as a hindrance because they significantly alter the structure of the code. Aggressive optimizations, like loop unrolling, make it impossible to match source-level statements to binary-level instructions only relying on debug information. Even when IR code is leveraged instead of source code for its optimized structure, there is no guarantee that a straightforward mapping between IR and binary codes exists. For this reason, the majority of existing approaches either turn off all compiler optimizations [SB08], [MSVSL08a] or rule out optimizations that heavily alter the

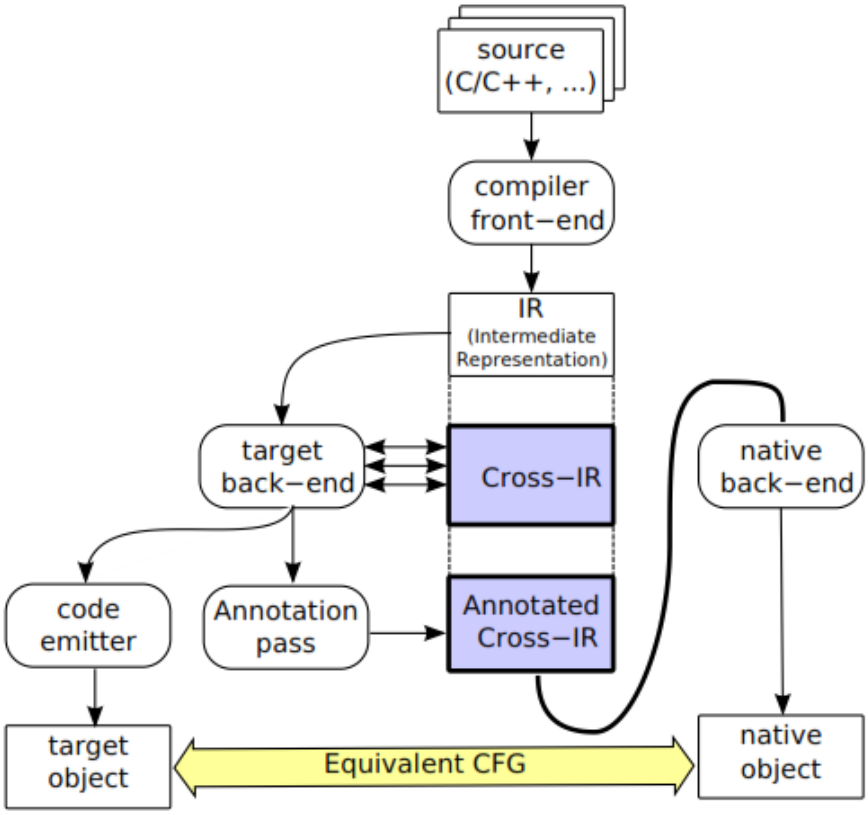


Figure 3.9: IR-level annotation technique using an extended compiler [BGP09]

control flow graph [MGLS11], [ZH09], [GCZ13].

3.2.3 Binary-Level Simulation (BLS)

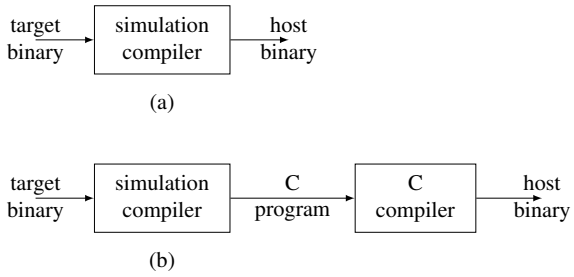


Figure 3.10: Two approaches of binary-to-binary translation taken from [ZM96]

To avoid the matching problems caused by compiler optimization, the target binary-level code is used as a functional model [ZM96], [LBH+00], [PWH12]. This code contains all the optimizations made by the target compiler. The target binary cannot be executed on the target machine if the host and target ISAs are different, which is usually the case. So, to be able to natively execute the target binary code, it is statically converted to either functionally-equivalent host machine instructions (fig. 3.10-(a)) or a high level language code written in C or C++ (fig. 3.10-(b)). The translation is carried out by a so-called *simulation compiler*.

```
while(i<10){
  c[i]=a[i]*b[i];
  i++;
}
```

(a) Source Code

```
0x1800098 lbz     r0,0(r8)
0x180009c addi    r8,r8,1
0x18000a0 lbz     r9,0(r11)
0x18000a4 addi    r11,r11,1
0x18000a8 mullw   r0,r0,r9
0x18000ac stbx    r0,r10,r7
0x18000b0 addi    r10,r10,1
0x18000b4 bdnz+   1800098
```

(b) Target Binary

```
r[0]=MEM_READ_BYTE(r[8]+0);
r[8]=r[8]+1;
r[9]=MEM_READ_BYTE(r[11]+0);
r[11]=r[11]+1;
r[0]=(sword_t)((sdword_t)((sword_t)r[0]*
     (sword_t)r[9])&0x00000000ffffffff);
MEM_WRITE_BYTE(r[10]+r[7], r[0]);
r[10]=r[10]+1;
PC=0x18000b8;
CTR=CTR - 1;
if(CTR != 0)
{
  PC=0x01800098;
}
```

(c) Binary Level Representation in C

Figure 3.11: An example of C code generated from target binary code [Wan10]

Non-functional information is inserted in the re-constructed code in a straightforward manner because both the functional model and the performance model are extracted from the same target binary code. So, the simulation accuracy only depends on the performance model.

BLS is very similar to SBT and presents the same set of limitations caused by the presence of indirect jumps and potentially run-time self-modifying code. The authors of [ZM96] use interpretive simulation, in addition to the compiled simulation, as a fallback mechanism in case of self-modifying code, which slows down the simulation speed. To deal with indirect branch instructions, authors in [LBH+00] consider every instruction as a possible indirect branch target and thus they add a label in front of each one of them. The existence of such labels in the translated code hinders the compiler optimization process during host code generation.

In this thesis, we chose to use the compiler intermediate representation as a functional model, where target-specific performance metrics will be inserted.

3.3 Modeling Micro-Architectural Components: Lack of Consideration for Complex Architectures

In addition to the precise placement of annotations in the functional model (section 3.2), the accuracy of the estimates also depends on the accuracy of the annotation data itself. The retrieval of precise non-functional information and the careful abstraction and modeling of target components play an important role in accurate and fast native simulation. For clarity reasons, we will take the cycle count, the instruction count and the miss count as examples of non-functional information in the remaining of this thesis.

MPSoCs tend to increase the level of parallelism by integrating VLIW processors with instruction buffers, out-of-order processors, high-performance branch predictors, advanced prefetch mechanisms, etc. Taking into account the impact of target-specific features on the performance of the system in native simulation while maintaining a reasonable simulation time has been addressed, to an extent, in previous research.

Estimating the execution time of a program can be handled by associating each instruction with its corresponding latency [MSVSL08b] using information provided by the target

processor manual and simply tallying up the latencies of single instructions. This naive approach is unrealistic and inaccurate because it completely blacks-out low-level timing effects of the processor micro-architecture such as pipeline effects, caching, branch prediction, network congestion, superscalarity, etc. More in-depth approaches perform static and/or dynamic analysis of the target code in order to yield accurate estimates.

Some effects such as pipeline stalls due to certain dependencies between instructions can be determined statically, whereas the execution time of memory instructions or a pipeline flush delay, caused by a conditional branch at the end of a basic block, are highly dynamic and context-related and cannot be accurately determined prior to simulation. So, joining static analysis to dynamic execution can be advantageous.

3.3.1 Estimation of Pipeline Effects

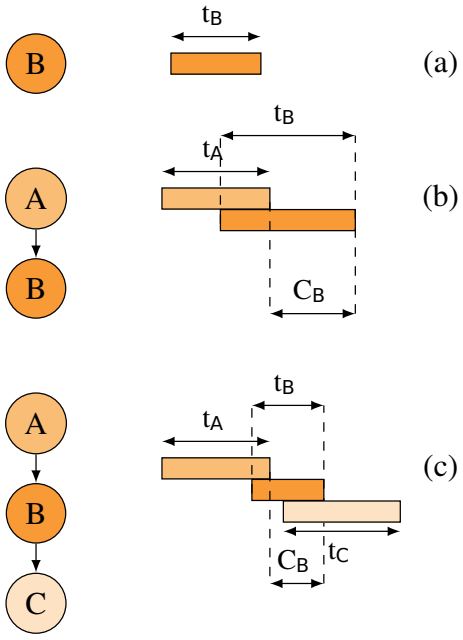


Figure 3.12: Execution cost of a basic block

Most modern MPSoCs use a pipeline, which makes the execution time of a basic block dependent on the sequence of its predecessors and also successors because the execution of successive basic blocks may overlap. As a consequence, the execution time of a sequence of basic blocks may be different from the sum of execution times of individual basic blocks, as illustrated in fig. 3.12. The execution cost of a basic block is the difference between its completion time and the completion time of its predecessor. The execution cost of the first executed basic block (entry basic block) is equal to its execution time. So, the execution time of a sequence of basic blocks can be computed as the sum of their execution costs (not their execution times): $Exec_{time}(B1 \rightarrow B2 \rightarrow ... \rightarrow Bn) = \sum_{i=1}^{n} C_{B_i}$.

As shown in fig. 3.12-(b), the execution time of basic block B depends on its predecessor A. In fact, its execution time is longer than when B is executed in an empty pipeline fig. 3.12-(a). This can be explained by the fact that some instructions of basic block A are still in the pipeline and might stall some instructions of B because of data dependencies or resource conflicts (pipeline stages, functional units, register values, etc.). The execution

time of a basic block can also be affected by its successors in case the processor features a superscalar pipeline with dynamic instruction scheduling or if it exhibits out-of-order execution capabilities. In fig. 3.12-(c), the execution time of B is affected by both its succesor and predecessor and it is shorter than t_B in fig. 3.12-(a). Since the sequence of instructions in a basic block is known at compile time, pipeline effects such as data and structural hazards can be studied statically.

In [RS09], a pipeline analysis is conducted by building execution graphs, also employed in [LRM06] to determine an upper bound of block execution time, to model the execution of basic blocks and the way they are processed through the pipeline. The authors take into consideration all possible contexts by representing each context as a set of parameters representing the availability of pipeline resources. They derive block costs from relative start and finish times and as a function of the modeled parameters. Although the latency of instructions also depends on the content of instruction and data caches and the branch predictior table, Rochange et al., do not address this issue and assume perfect (always hit) caches and oracle branch predictor.

In [GHP09] and [CHB09a], the stalling of the pipeline is reflected by statically determining interlocks between instructions and accordingly adding extra dependency cycles. These dependencies are detected by examining the registers used in machine instructions. For instance, if a register is loaded by an instruction and it is immediately used by the following instruction, a dependency is pinpointed. The second instruction cannot start execution unless its input register has been loaded by the previous instruction. However, modern processors allow the execution of an instruction even if its data is not yet available. The authors of [GHP09] advocate the use of the approach in [RS09] for superscalar processors.

3.3.2 Estimation of Cache Effects

Almost all modern MPSoCs are equipped with data caches to improve their performance and to tighten the gap between their fast processing units and slow memory accesses by taking advantage of the locality principle offered by these fast small memories. Therefore, non-functional properties like execution time are ruled by the interaction between the software and the cache. A cache hit for instance takes less time (and consumes less power) than a memory access that has to cross the memory hierarchy to reach main memory. Consequently, the number of hits and misses have a consequential impact on the execution time of the software. So, estimating the performance of MPSoCs calls for taking into consideration cache effects. These effects have been considered in static and dynamic techniques.

The Worst Case Execution Time (WCET) has been adopted in many researches in order to estimate the software performance and guarantee that the execution time of the software does not surpass a certain bound. This makes the WCET technique more convenient with real-time embedded systems. Integer linear programming (ILP) has been proposed in [LSA95] to estimate a tight bound on the WCET taking into account the instruction cache effect. The idea of the ILP scheme is to represent the execution time of a program as a cost function. Program structural constraints, program functionality constraints and instruction cache constraints are determined then passed to an ILP solver in order to maximize the cost function. Another approach based on computing the WCET for static instruction cache analysis is proposed in [PLH11]. They resorted to a fixed-point free analysis technique to avoid the high consumption of memory and time. Their method offers a twofold estimation: a mandatory basic analysis involving an intra basic block, as well as a loop analysis and an optional analysis, which entails an inter basic block analysis and an inter call analysis.

In [PJ15], an analytic method based on the computation of the reuse distance distribution is employed to estimate cache performance. Different models are tailored to suit various cache configurations (fully associative and N-way set associative with Random/ LRU/PLRU/bit-PLRU replacement policy). The main idea of these models is to keep track of the cache behavior between two consecutive accesses to the same cache line using traces of memory access sequences. The models are represented by a Markov chain in which transitions between different states depend on the reuse distribution. By analyzing the Markov chain, a prediction of a miss ratio is determined.

In [SGCB12], a method to model memory accesses in a source level simulation was proposed. This method relies on static cache analysis using the concept of "must" and "may" states (as not every memory access can be statically determined to be either a hit or a miss). To estimate these potential cache states, memory addresses are needed. However, not all addresses can be known statically. So, interval analysis is used to determine the possible address range of a memory access. This static analysis is then coupled with dynamic path simulation. The result is a range of cache hits and misses; some accesses remain unclassified.

In [LMGSB13], the authors avoid the simulation overhead that may be caused by cache modeling (also referred to as in-place caches) in a host compiled simulation by conducting a cache-conflict aware annotation. For instruction caches, a cache conflict analysis is conducted for all loops before simulation. So, no instruction cache simulation needs to be performed within loop bodies. For data caches, aggregated data cache simulation is used. Instead of annotating every single memory access, a large data block with a large address range (e.g. an array) is annotated at once. They also use data locality (stack, heap and data section) to determine an address range, which can be used to estimate the number of data cache misses. In this case, an over estimation of cache misses may occur.

Static approaches fall short of providing accurate performance estimation results because of the dynamic nature of cache memories. The state of the cache is highly dependent on the execution context of the program, which can only be known at run-time. For this reason, dynamic estimation techniques ([KMGS13], [WH13], [DPE11], [PCG09],[YMH+14]) rely on abstract models that imitate the behavior (performance-wise and not necessarily functionality-wise) of the target micro-architectural components. Instruction and data cache models are used during simulation in order to recreate the dynamic behavior of the real caches. These models are triggered during simulation via appropriate functions, called *annotation functions*, inserted in the software. The interaction between the software and such architectural models help determine the statically-unpredictible number of cache misses and their delay.

However, these approaches are valid for simple architectures, such as scalar or in-order processors, where instructions are executed sequentially and where memory accesses are considered to be blocking. However, in more complex architectures like superscalar, VLIW or out-of-order processors, a more complex dynamic behavior is exhibited. Out-of-order processors, for example, are capable of executing instructions in advance, i.e. the processor does not stall on a cache miss, instead it continues executing independent instructions. Moreover, load/store operations may be reordered by an out-of-order processor. So, the above-mentioned approaches are not suitable for complex architectures.

An approach consisting of computing the stack distance histogram using memory traces is used in [DAP15]. The main contribution was to propose an approximation of the number of misses and execution cycles with respect to different cache setups in an out-of-order processor.

The authors of [PWH12] propose a new performance estimation technique dealing with

dynamic out-of-order effects of the instruction queue and non-blocking caches. Since out-of-order execution occurs at the instruction level, they model the out-of-order effects using BLS as it allows the representation of the target code at the granularity of instructions. The code is annotated with time information measured with a reference ISS. They make use of the dependency chains generated by the cycle-accurate simulator to determine dependency between load/store instructions and to perform static reordering of these instructions for the purpose of simulating a non-blocking cache.

One of the contributions of this thesis is to propose a performance estimation approach of an instruction cache taking into consideration the particularities of a VLIW architecture.

3.3.3 Branch Penalty

Modern pipelined microprocessor architectures make use of *branch predictors* to enhance their performance. The role of a *branch predictor* is to speculate about which path a branch will follow before the branch is even evaluated. An example of a branch prediction policy is the static *not-taken* policy of ARM9. A conditional jump can either be *taken* and jumps to its target in a different memory region, or *not-taken* and continues execution with the basic block whose first address immediately follows the address of the branch instruction. In the *not-taken* policy, the instructions of the consecutive block are always fetched and speculatively executed. If, at run time, it turns out that it is the other block that has to be executed, the pipeline has to be flushed and a new fetch (of the correct instructions) is performed. This *branch misprediction* incurs a non negligeable number of additional clock cycles.

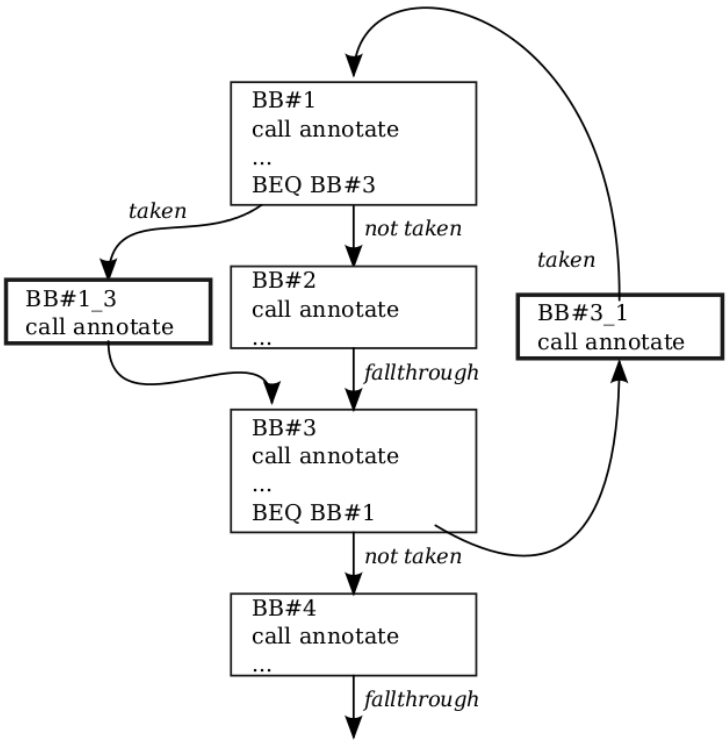


Figure 3.13: Annotation of branch prediction effects [GHP09]

In order to take into account the misprediction delay, auhtors of [GHP09] annotate the arcs of the target processor CFG by inserting additional basic blocks that contain the mispre-

diction penalties on the corresponding arcs of the host CFG (fig. 3.13). A similar approach is employed in [CHB09a]. More complex branch prediction techniques, such as bimodal, local, global, two-level, TAGE [SM06], etc., may require the integration of a prediction model to the annotation scheme and possibly the annotation of both *taken* and *not-taken* arcs. However, embedded processors usually still use static branch predictors.

The more the micro-architectural effects are accounted for, the higher the precision is and the lower the simulation speed is. It is a compromise the developer has to make based on the aspects of the MPSoC that need to be validated and whether the time allotted to design space exploration allows exhaustive simulation or not.

3.4 Conclusion

In this chapter, we presented the solutions provided by previous works to the key issues in native simulation. These issues include the difference between host and target address spaces, which was addressed by the use of the HAV technology, and the inability of host-compiled simulation to provide information about the performance of the system. An overview of performance estimation techniques dealing with the latter issue was given. We classified these techniques according to the abstraction level of the employed functional model: source level, intermediate level and binary level. We also highlighted the advantages and limitations of each approach insisting on the adopted mapping strategy between the target binary and the high-level functional model for the purpose of annotation insertion. As for the computation of performance metrics, previous approaches focused on the effects of certain micro-architectural components (e.g. pipeline and instruction and data caches) on the performance of the system and made a number of simplifications, which are not valid for complex systems. Complex architectures that incorporate superscalar or VLIW processors were not considered.

In the following chapters, we will present the key contributions of this thesis, which include an IR-based annotation framework that aims at accurately placing non-functional information into the IR code using a loop-oriented mapping algorithm and that computes target-specific performance metrics by recreating the behavior of the real instruction cache and instruction buffer of a VLIW architecture.

Chapter 4

IR-Level Annotation Framework for Performance Estimation

Native simulation is one of the most suitable candidates to speed up the architecture space exploration and early design validation steps. However, it lacks time information, which is crucial in software performance estimation. To cater for the absence of time, a plenitude of approaches that aim at annotating the software with time information have emerged. These approaches (be it SLS, BLS or ILS) tend to either rely on debugging information when mapping the binary code to the high-level code, which can be misleading due to compiler optimizations, disregard (some or all) compiler optimizations or at best consider all compiler optimizations while being dependent on the target architecture.

In this chapter, we propose an IR-level (GIMPLE-CFG) annotation framework that is architecture independent and that reflects compiler optimizations through a mapping scheme. This mapping scheme is conducted at a basic block level, between the binary and the high-level IR with a special focus on loop structures, as they are the most challenging part of the mapping process.

We will give an overview of the proposed annotation framework in section 4.1, followed by a detailed description of the espoused intermediate representation in section 4.2. The mapping approach, which is the core of this chapter, will be discussed in section 4.3. A mapping scheme devised to tackle common compiler optimizations (*gcc -O2*) will be explained and applied to a toy example, followed by a description of an overhauled version of the mapping approach that deals with aggressive compiler optimizations (*gcc -O3*).

4.1 Annotation Framework Overview

The proposed ILS approach brings about two orthogonal issues. The first one concerns the way of extracting non-functional information and the second is about where to place this information in the IR. Thus, a two-fold strategy is required. Retrieving precise performance metrics of the target processor is achieved by incorporating a performance model that takes into account the effects of the target micro-architectural components. Inserting the obtained information into the IR code requires a mapping scheme between the IR and binary CFGs. As compilers perform many optimizations to enhance software performance, the high-level code and the binary code structures may be radically different. Consequently, this mapping process, which is key to correct performance estimation, is challenging to conduct, especially when aggressive compiler optimizations are enabled.

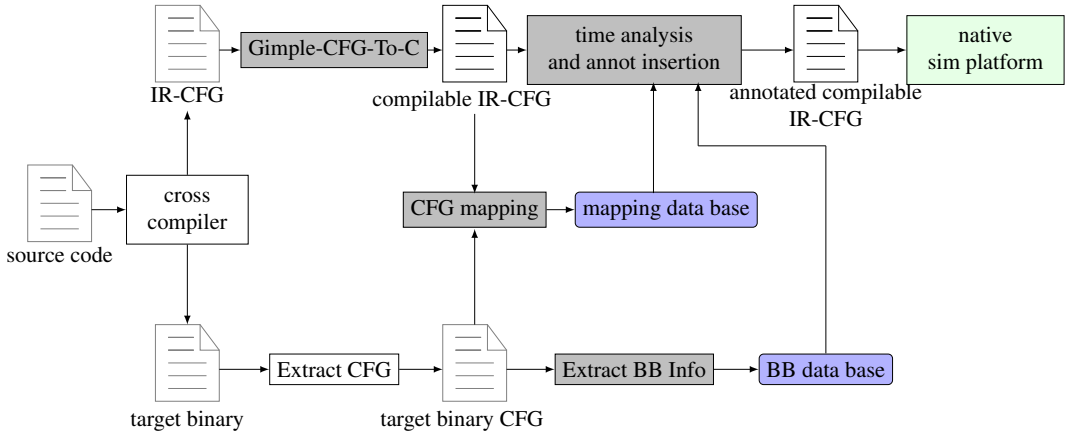


Figure 4.1: The IR-level annotation framework

We developed an IR-level annotation framework (fig. 4.1) that computes performance metrics of the target processor and places them in their correct positions in the IR code through a loop-oriented mapping scheme.

The workflow of the framework is outlined in fig. 4.1. The source code is percolated through the framework leading to the generation of an optimized high-level code: *Annotated compilable IR-CFG*.

First, the source code is cross-compiled generating both the intermediate representation (IR-CFG), after high-level compiler optimization passes are performed, and the target binary containing both front-end and back-end compiler optimizations. Despite its easy-to-understand syntax, Gimple CFG is not compilable. So, we converted it into a compilable and optimized C code that maintains the CFG structure (uppermost flow). To do so, we developed a tool called *GIMPLE-CFG-To-C* that carries out the transformation of the IR to a particular compilable C code. This code preserves the functionality of the original source code but has an optimized structure (inherited from *Gimple CFG*) that exposes the basic block boundaries and accounts for the high-level optimizations. As for the binary, a CFG is recovered from it and non-functional information available at compile time (number of instructions, first and last addresses, number of cycles, etc.) is extracted from each basic block of the binary CFG and stored in a *basic block data base* (lower-most flow). A full-blown performance model would also include dynamic timing effects caused by components like instruction and data caches.

As we mentioned earlier, extracting precise non-functional information from the target binary is not enough in producing accurate performance estimates. This information has to be inserted at the right place in the IR, hence the need for a precise mapping between the binary CFG and the IR CFG (the middle flow fig. 4.1). The mapping algorithm results in a *Mapping data base*, which will come in handy during the *time analysis and annotation insertion*. Finally, annotations are inserted in the IR-like C code according to the mapping algorithm.

4.2 Choice of the Intermediate Representation

Due to compiler optimizations (branch optimizations, function inlining, loop optimizations, etc.), accurate annotation of the source code, with low-level target-specific information, is challenging. The discrepancy in the CFG structure between the source code and the binary code, which may lead to estimation errors, is avoided by working on the IR code that

encompasses machine-independent optimizations and whose structure is very close to the binary CFG. One of the perks of using a high-level IR as a functional model is that it is architecture-independent, unlike the BLS approach, which requires knowledge of the target ISA. This makes our annotation approach retargetable.

4.2.1 GCC's Intermediate Representations and IR to C Conversion

The GCC compiler is composed of three main parts: a *front-end*, which is language dependent, a *middle-end* (a.k.a. *Tree optimizer*), which is language and target machine independent, and a *back-end*, which is target dependent. The *front-end* and the *middle-end* are usually considered as one part. Fig. 4.2 depicts two decoupled parts, namely the *front-end* (and *middle-end*) and the *back-end*. GCC offers a variety of options to dump intermediate representations at different optimization stages. As illustrated by fig. 4.2, the final optimized file generated

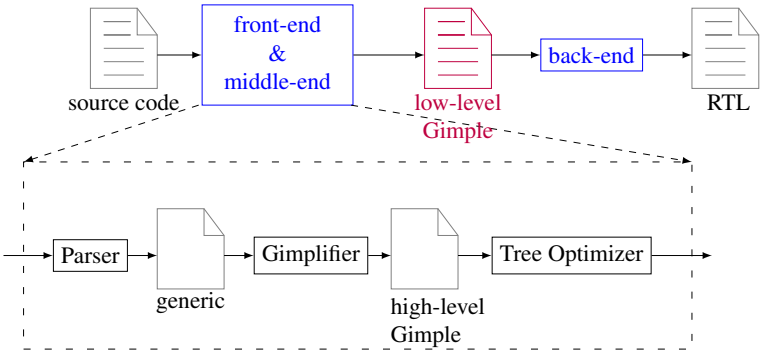


Figure 4.2: GCC's intermediate representations

by the back-end right before assembly code generation is *gcc-RTL* (Register Transfer Level: not to be confused with the RTL used in hardware description languages, is the final optimization pass in GCC). Thus, the RTL format, which could be output using *GCC*'s dump option *-fdump-rtl-pass*, is the closest to machine code as it encompasses machine-dependent optimizations. However, RTL is a very low-level IR that highly depends on the target architecture. One of the optimization passes that takes place at the RTL level is *register allocation*. This pass aims at replacing the pseudo registers with hard registers, which requires knowledge of the target processor.

On the other hand, there are the GCC Tree Optimizer IRs, which are generated before the RTL. These IRs contain front-end optimizations and can be dumped in a *pretty print* format, which is a format that *resembles* (but not identical to) C code. The front-end parses the source code and checks for syntactical errors. It converts the input source code into a tree representation. An *AST (abstract syntax tree)/GENERIC* IR, which is independent of the programming language, is then generated from this tree. Later, it is converted to a *high-level Gimple* representation, which is a three-address representation. *Gimple* is a family of intermediate representations (IR) based on the tree data structure. The gimplifier, which is a compiler pass that is in charge of converting GENERIC into *Gimple*, allows simplifying complex source code expressions, like *for*, *while*, *do-while*, *switch*,etc., by re-expressing them using *if* and *go-to* statements and breaks statements into simpler code with three operands each. To do so, it creates new temporary variables to keep the value of the sub-expressions. These temporaries are called *expression temporaries*. The *Tree Optimizer*, then, performs several optimization passes on *high-level Gimple* after lowering it to a CFG representation leading to the

generation of *low-level Gimple*.

What interests us is the last pass before RTL, which is *Gimple CFG* generation (low-level Gimple in fig. 4.2) because it includes all machine-independent optimizations. *Low-level Gimple* can be dumped using *gcc -fdump-tree-optimized*. Gimple CFG is both source-language and target-machine independent.

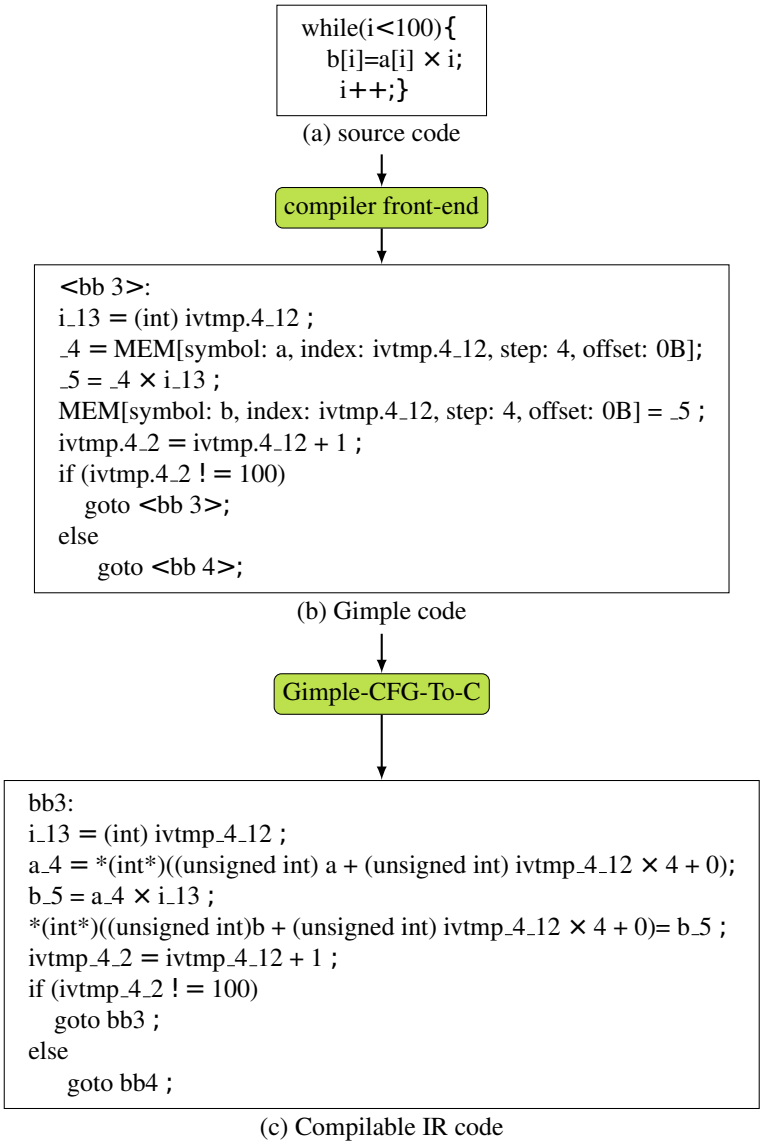


Figure 4.3: Generation of a compilable IR

Although the high-level IR has a structure close to the binary code while retaining high-level information of the original source code, such as variable names and high-level operations, Gimple is not compilable because of some naming rules of its temporary variables and a few Gimple operations that do not comply with C syntax.

Fig. 4.3 outlines the generation process of a compilable IR code from a C code. First, an intermediate representation is generated from the source code (fig. 4.3-(a)) and dumped using *gcc*'s front-end. This IR (fig. 4.3-(b)) encompasses all architecture-independent optimizations. As mentioned earlier, the *gimplifier* pass converts complex C statements into

sets of simpler expressions. All control structures in the C code are lowered to conditional jumps in Gimple. For instance, the *while* loop in the source code is broken down to *if-else* statements in the IR. These conditional branches are followed by *jump* statements to the desired targets using *goto* with the label of the target basic block (the borders of basic blocks in the IR are delimited with labels). Basic block labels are also not C-friendly. We can also notice the presence of new variables i.e. the temporaries, (e.g. $ivtmp.4\_12$) to hold intermediate values in complex expressions. In addition, some memory accesses (dealing with arrays in particular) are performed using the *MEM* syntax (fig. 4.3-(b)). This *MEM* syntax has the following structure: $MEM[symbol : s, base : b, index : i, step : st, offset : of]$ (where all parameters default to zero except for st, it defaults to one) and it refers to a memory address($s + b + i \times st + of$). This structure also does not conform to C syntax.

Other differences that are not represented in the example may be observed in function definitons, pointer arithmetic operations and tree structures like *ABS_EXPR*, *MIN_EXPR* and *MAX_EXPR*.

These differences between IR and C syntax, although small, prohibits the IR from being compilable. So, a tool was devised (*Gimple-CFG-To-C*) to convert the IR into a compilable C code. This tool handles the above-mentioned discrepencies between IR and C syntax and generates a compilable IR (i.e. optimized C code) that maintains the same functionality as the original source code and has the same structure and optimizations as Gimple (fig. 4.3-(c)). As shown in the compilable IR example, variables and labels are renamed. The *MEM* structure is transformed by extracting the address to which the structure refers and the data type (e.g. integer) of the value accessed by this memory address. The address is cast as a pointer to the determined data type (e.g. (int *)) and a dereferencing operator (*) is added to access the value located at that address: *(int *)(($unsigned\ int$)s + ($unsigned\ int$)b + ($unsigned\ int$)i $\times$ st + of). Once all conversions have been performed, the IR becomes compilable and is used as a functional model.

4.2.2 Compiler Optimizations and Code Structure

Compilers offer a multitude of advanced optimizations that improve software performance (common wisdom suggests up to 30%) and/or code size by exploiting instruction-level parallelism (ILP), data level parallelism, data locality, etc., [BEP04]. Enabling optimizations comes at the expense of compilation time and debugging capabilities.

During the different optimization phases, the compiler uses the IR under the CFG format because it facilitates the analysis and optimization process. Optimizations can be classified in two categories:

- Machine-dependent optimizations: They exploit specific target machine features and they are mainly based on register allocation and the use of special instructions of the target machine (e.g. peephole optimizations, register allocation, hardware loops, loop unrolling, if-conversion, etc.). These optimizations are performed by the compiler back-end.

- Machine-independent optimizations: They improve the target code by operating on abstract programming concepts (structures, loops, objects, etc.) that do not require any consideration of the target machine properties (e.g. hoisting, renaming of temporary variables, constant propagation, dead code elimination, loop distribution, etc.). These optimizations are performed by the compiler front-end.

Compiler optimizations can have a local effect (local to a vertex of the CFG, i.e. a basic block) or a global effect (the scope of the entire CFG). Locally-scoped optimizations only look at the instructions inside a basic block (e.g. instruction combining, interchange of two independent adjacent instructions, etc.), so they do not require a lot of information and they are easier to perform compared to global optimizations that necessitate a global view of the program and complex computations (e.g. code block reordering, cross jumping, function inlining, dead code elimination, vectorization, etc.). Also, some optimizations preserve the structure of the code (e.g. common sub-expression elimination, dead code elimination, constant folding and propagation, hoisting, instruction combining, etc.), others introduce many transformations to the original source code leading in all but the most simple cases to a binary code with a control flow graph (CFG) different from the original one (e.g. if-conversion, loop distribution, loop unswitching, loop fusion, vectorization, jump threading, loop unrolling, etc.). Optimizations are enabled if a *Ox* ($x = 1, 2, 3$) level is specified on the command line when *gcc* is invoked. With *O0*, no optimization is performed and the source code is directly transformed to machine instructions. Fig. 4.4 shows two different binary codes generated from the same source code. Fig. 4.4-(b) corresponds to the unoptimized binary and fig. 4.4-(c) corresponds to the optimized one. With the optimization level *O2* the code size is reduced compared to the unoptimized one.

A higher optimization level is inclusive of all the optimizations enabled at lower levels and turns on additional optimizations. The *O3* level for example, applies a larger number of optimizations compared to the level *O2* including aggressive ones that radically change the structure of the code, like *loop unrolling*. More (aggressive) optimizations does not necessarily mean more performance gain as the code can grow in size (in case of certain optimizations like *loop unrolling* or *function inlining*), which increases memory requirements, and a slowdown in execution time can sometimes be observed when a very small instruction cache is used. Developers usually use level *O2*, where most common compiler optimizations are turned on while maintaining a reasonable code size, or level *Os*, where all *O2* optimizations that do not increase the executable size are enabled.

As we can notice in fig. 4.5, the IR and the binary code have the same CFG structure. So, there is a one-to-one relationship between basic blocks of the CFGs, which facilitates the mapping process. These two CFGs are said to be *isomorphic*. It should be noted that this best case scenario (i.e. isomorphic CFGs), as depicted in fig. 4.5, does not always occur. An *isomorphism* from $CFG_{IR}(V_{IR}, E_{IR})$ to $CFG_{bin}(V_{bin}, E_{bin})$ is a pair of bijections:

$$\phi : V_{IR} \rightarrow V_{bin}, \psi : E_{IR} \rightarrow E_{bin}$$

such that $\forall e \in E_{IR}$ and vertices $u, v \in V_{IR}$, the edge e is incident with u, v *iff* the edge $\psi(e)$ is incident with the vertices $\phi(u), \phi(v)$:

$$e = uv \Leftrightarrow \psi(e) = \phi(u)\phi(v).$$

CFG_{IR} and CFG_{bin} are said to be isomorphic when such pair of bijections exists. In this case, we benefit the most from the IR representation because the code preserves its structure till the end of the compilation. However, once a loop is introduced, which is illustrated in fig. 4.6, the CFGs are not isomorphic anymore. In addition to loops, the compiler back-end may optimize branch statements by using target-specific instructions (*if-conversion)*, which may further change the structure of the code.

As discussed before, *Gimple-CFG* does not take into account target specific optimizations which will lead, in certain cases, to a CFG different from the target binary one. How-

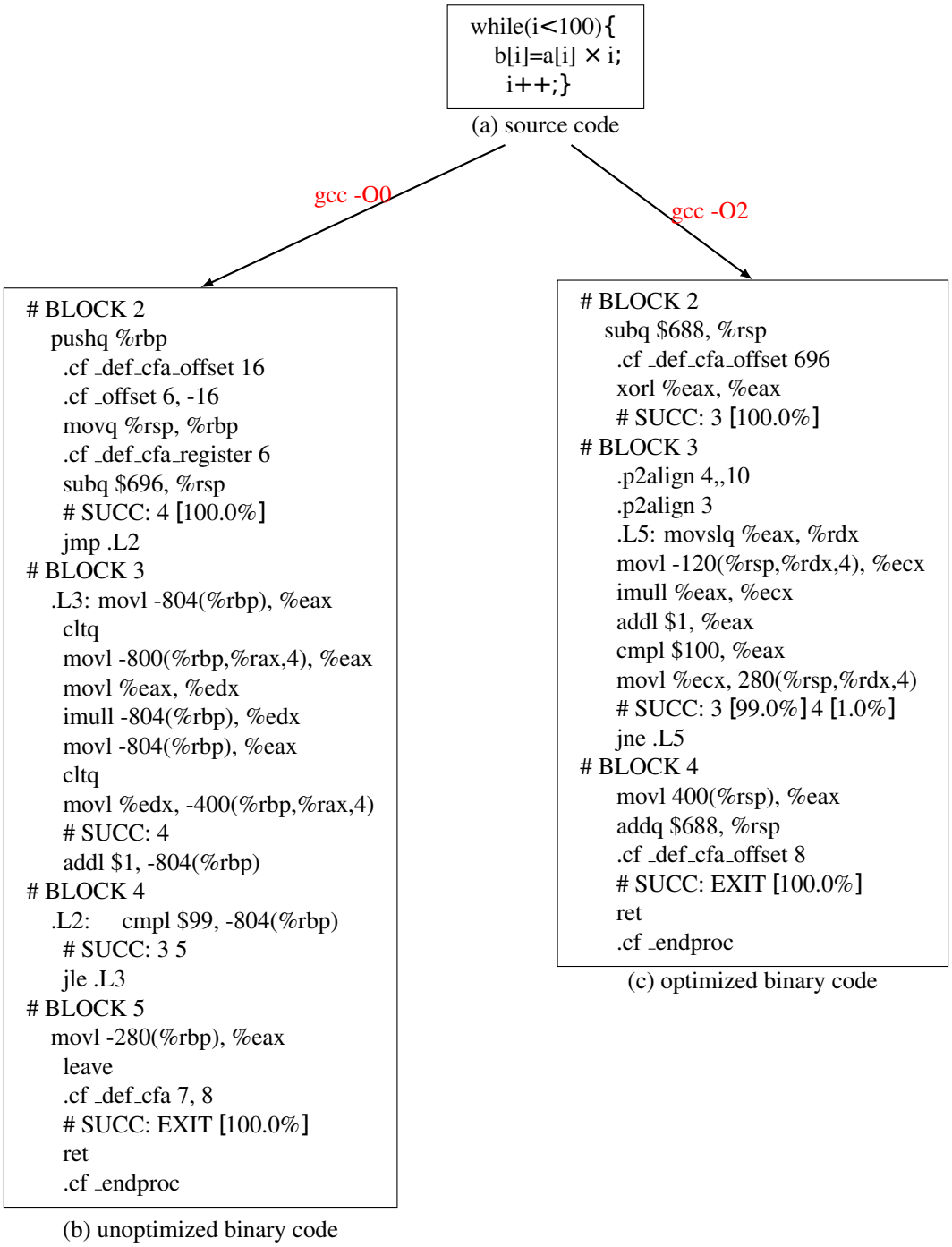
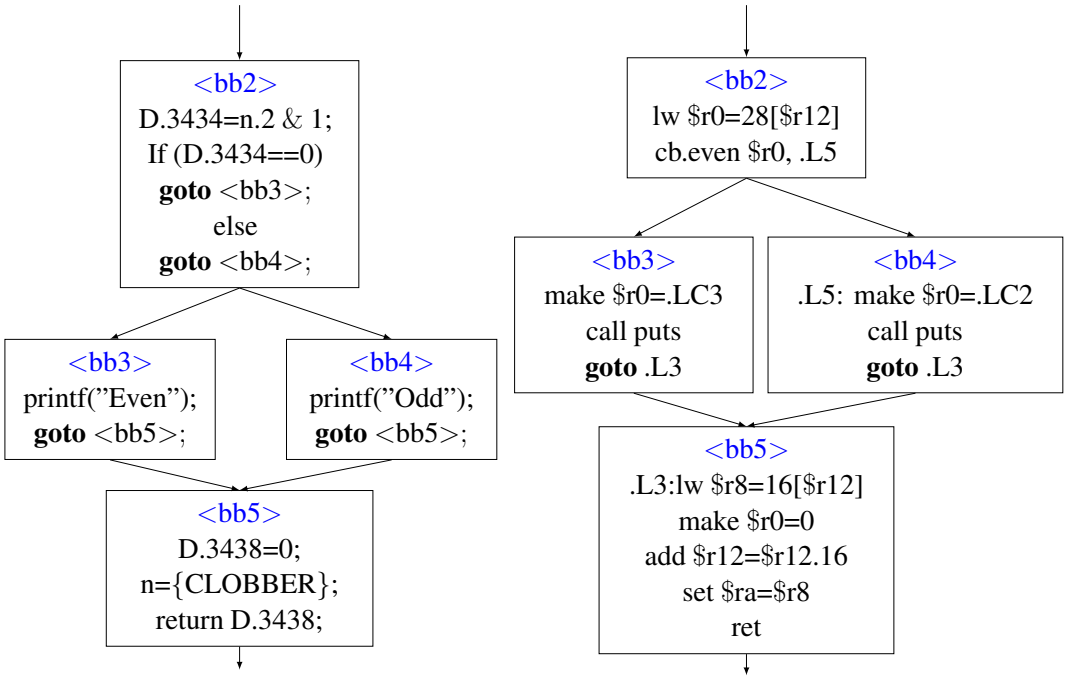


Figure 4.4: An example of two different binary codes generated from the same source code

ever, many optimizations take place in the GCC *Tree Optimizer* as the trend in GCC consists of advancing the RTL optimization passes to the *Tree Optimizer level* [Nov06]. Target-independent loop optimizations are more and more carried out before the back-end. In fact, some tree loop optimizations can be manually activated through specific flags (for eg: *-floop-interchange*, *-ftree-loop-im*, *-ftree-parallelize-loops*, etc.) as they are not automatically activated by just precising an optimization level.

Fig. 4.6 illustrates a simple example of a C code compiled with *gcc*, using the optimiza-

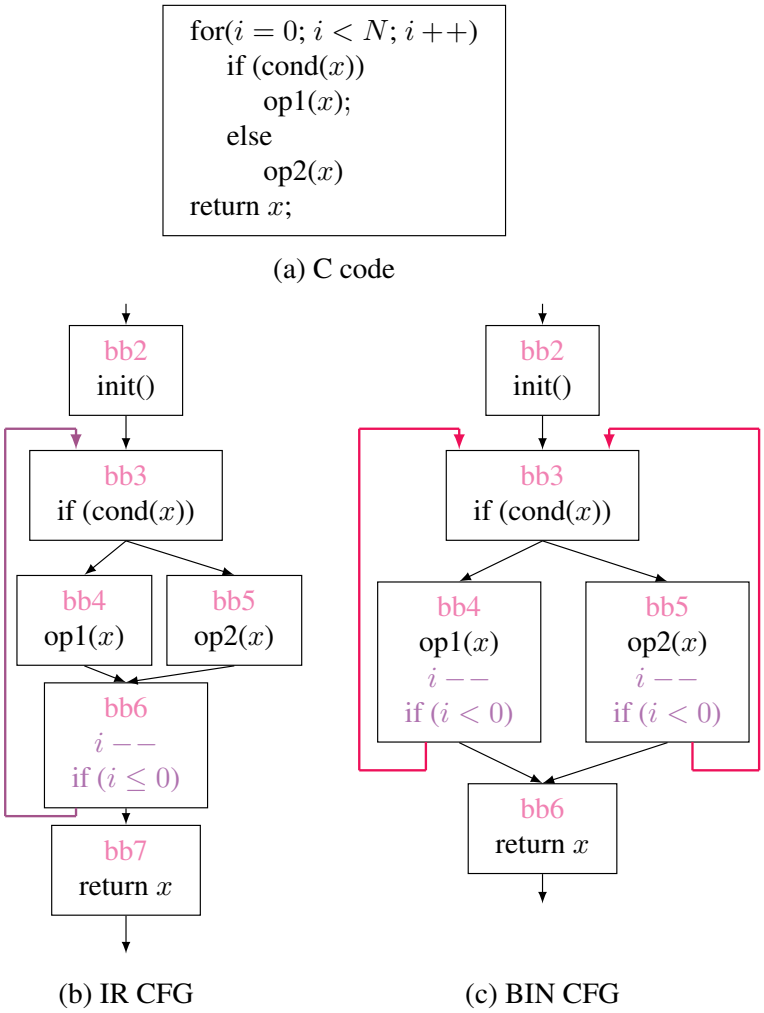


(**a**) Odd_even_IR_CFG (**b**) Odd_even_binary_CFG

Figure 4.5: An example of isomorphic IR and binary CFGs

tion level *O2*, and its corresponding IR and binary CFGs. The IR is generated by the compiler after the front-end so it contains high-level optimizations. The binary, on the other hand, encompasses both front-end and back-end compiler optimizations, hence the difference between the IR and binary CFGs. The C code (fig. 4.6-(a)) underwent several compiler transformations before reaching the binary generation pass. One of the obvious transformations present in both the IR (fig. 4.6-(b)) and binary (fig. 4.6-(c)) CFGs is *Loop Reversal*, which consists of running the loop backward. Moreover, the loop in the IR is not identical to the one in the binary because it went through *tail duplication* and *branch target expansion*. First, the tail ($bb6_{ir}$) is duplicated. Then, $bb6_{ir}$ and its clone ($bb6_{ir}$) are merged into their predecessors ($bb4_{ir}$ and $bb5_{ir}$), which are the targets of the condition in $bb3_{ir}$, leading to $bb4_{bin}$ and $bb5_{bin}$. The expansion of the branch targets often increases code size but by forming larger blocks of instructions, more possibilities of optimization and scheduling are created.

Although, the IR is closer in its structure to the binary than the source code is, disparities can sometimes occur due to optimizations (the loop in fig. 4.6-(b) is composed of four basic blocks including one loop latch while the one in fig. 4.6-(c) has only three basic blocks and two loop latches). An accurate mapping should be able to circumvent such mismatches. For example, mapping the CFG in fig. 4.6-(b) to the one in fig. 4.6-(c), without modifying the IR structure, results in: $bb2_{ir} \Leftrightarrow bb2_{bin}$, $bb3_{ir} \Leftrightarrow bb3_{bin}$, $bb4_{ir} \Leftrightarrow bb4_{bin}$, $bb5_{ir} \Leftrightarrow bb5_{bin}$, $bb6\_ir \leftrightarrow \emptyset$ and $bb7_{ir} \Leftrightarrow bb6_{bin}$.



(a) C code

(b) IR CFG

(c) BIN CFG

Figure 4.6: IR and binary CFGs of a simple C code compiled with *gcc -O2*

4.3 Proposed Mapping Approach Between IR and Binary CFGs

The linchpin of accurate coupling of the performance model and the functional model (IR code) is the mapping process. In order to accurately place the annotations in the IR code, a precise mapping needs to be performed. Although, we decided to meet the binary code halfway by using the IR instead of the source code, a straightforward mapping is not guaranteed because low-level Gimple does not contain back-end optimizations. The compiler back-end may introduce further optimizations to the IR leading to different IR and binary CFGs.

Mapping branch structures is not very complicated as it falls back either to ignoring the extra basic blocks in the IR (*i.e.* extra bbs remain unmapped) or to mapping a basic block in the IR to more than one basic block in the binary. Some cases of branch optimizations and their effect on the structure of the code are addressed in [LMGS12] and [GCZ13]. Further details about mapping branch structures will be provided in subsection 4.3.2. The main focus of our mapping scheme is to match IR basic blocks to their equivalents in the binary CFG with special consideration to loop structures.

Compilers perform several loop transformations, such as loop unrolling, loop permuta-

tion, loop fusion, loop fission, etc., to speed up a program. Due to these optimizations, even a small annotation error will lead to serious repercussions on the overall performance of the software because this insignificant error will be magnified as it will be repeated as many times as the number of loop iterations.

In our mapping scheme, we focus on loops because codes without loop statements do not engender major mapping problems. Indeed, programs spend the majority of their execution time on code inside loops, thus we consider loops as hotspots. On the bright side, target-independent loop optimizations are more and more carried out before the back-end (i.e. before RTL) [PCB06], thus minimizing stucture dissimilarity between the IR and the binary CFGs. As a result, analyzing loop structures at the IR level is very profitable as opposed to source code [WH12]. Finding which loop in the binary code corresponds to which loop in the source code is not a trivial task [LMGS12] as source code and binary code have very different structures.

We will start first, in subsection 4.3.1, by explaining the proposed mapping algorithm that aims at finding a correspondence between basic blocks of IR and binary CFGs. This algorithm tackles compiler optimizations enabled at level *O2*. Then, in subsection 4.3.2, we will describe how we adapted the mapping algorithm to aggressive optimizations enabled at level *O3*, namely *loop unrolling*.

4.3.1 Basic Mapping Scheme: Tackling Standard Compiler Optimizations

Our mapping process aims at matching basic blocks of the IR CFG with their counterparts in the binary CFG using an algorithm based on CFG condensation and the principle of *fixed points*. Before we dive into the details of this algorithm, we will give an insight about two concepts, which are considered to be the pillars of the proposed mapping approach.

Fixed Points

In the context of matching IR and binary CFGs, if two elements, each from a graph, are determined to be equivalent, they are both considered *fixed points*. We designate by element a subgraph $SG = (V_{SG}, E_{SG})$ of a $CFG = (V_{CFG}, E_{CFG})$:

$$SG_{CFG} := (V_{SG} \subseteq V_{CFG} \wedge E_{SG} \subseteq E_{CFG}),$$

that satisfies a property P, which will be descibed hereafter. A basic block is an example of such subgraph. Entry basic blocks of the IR and binary CFGs are considered *fixed points* because control flow can only enter a program through one entry point, which makes these entry basic blocks equivalent. The objective is to find as many *fixed points* as possible.

The idea of *fixed points* is inspired from [DR05] in which the authors conduct a bijective mapping using *fixed points* between two differing executables of the same source code, for the purpose of malware analysis. However, loop optimizations were not considered. Needless to say that finding *fixed points* in our case is more delicate because we are working with two different CFG representations: IR and binary.

Strongly Connected Component (SCC)

A subgraph SG, used in the mapping algorithm, should satisfy property $P : SG$ should be a maximal strongly conncected subgraph, i.e. a strongly connected component (SCC). A SCC of a directed graph is a maximal set of vertices such that for every pair u and v in the set, there is a path from u to v, $p1 = u \rightarrow v$, and a path form v to u, $p2 = v \rightarrow u$, i.e. each vertex of the set is reacheable from every other vertex of that same set. Fig. 4.7 highlights a

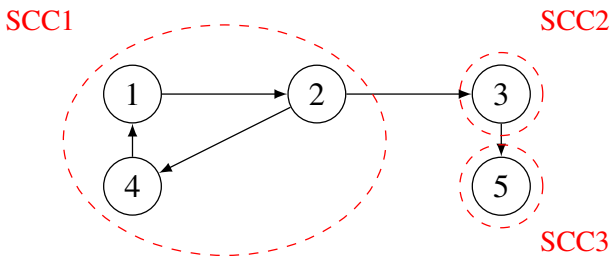


Figure 4.7: An example of SCCs

simple directed graph decomposed into three SCCs. $SCC1$ is composed of three vertices, $SCC2$ and $SCC3$ are composed of one node each.

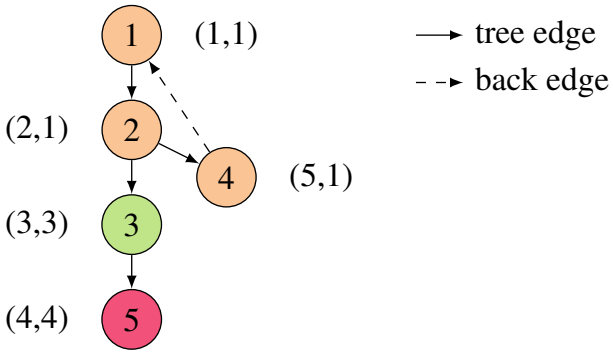


Figure 4.8: SCC decomposition using DFS

To decompose the CFGs into SCCs we rely on Tarjan's algorithm [Tar72], which is $O(n+m)$ where m is the number of vertices and n is the number of arcs of a given graph. Tarjan reduces the problem of finding the strongly connected components of a graph to the problem of finding the roots of the SCCs. To do so, a depth first search (DFS) is conducted leading to the genaration of a tree/forest and two values $(LOWVINE, LOWPT)$ are computed for each node of the tree. $LOWVINE(v)$ is the time/order at which node v was discovered during the DFS. $LOWPT(v)$ indicates $LOWVINE(u)$ such that u is the earliest visited vertex (the vertex with minimum discovery time) that can be reached from subtree rooted with v. Node v is the root of a SCC in Graph G *iff* $LOWPT(v) = LOWVINE(v)$. Nodes with the same $LOWVINE$ value form a strongly connected component.

Fig. 4.8 illustrates the DFS tree of the graph in fig. 4.7 and the $(LOWVINE, LOWPT)$ values of each node. For an in-depth description of the SCC extraction algorithm, as well as its implementation, we invite our readers to peruse [Tar72].

Mapping Algorithm

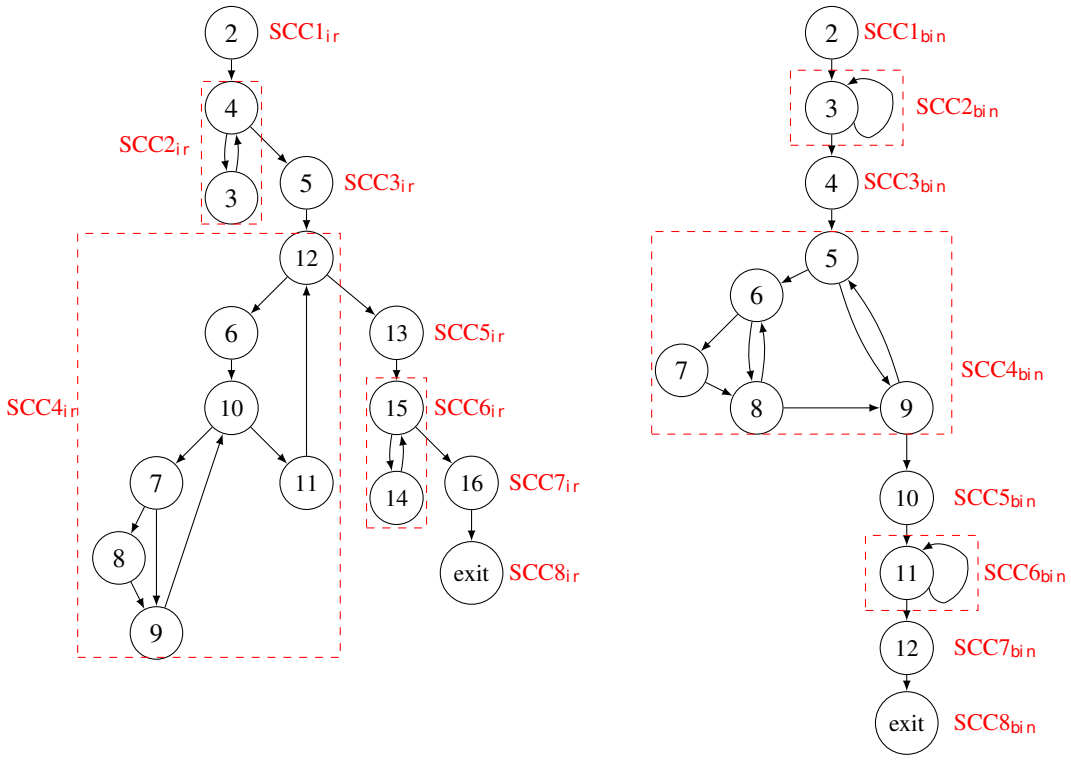
Algorithm 1 presents a high-level view of the proposed mapping approach.
We now give a detailed explanation using the example of fig. 4.9.

i) First, the IR and binary CFGs are both decomposed into SCCs using Tarjan's algorithm [Tar72] (line 2).

Fig. 4.9 shows the Gimple CFG (a) and the binary CFG (b) of the toy example *BubbleSort*. The Gimple CFG and the binary CFG are both composed of 8 SCCs. For example, the IR SCCs

Algorithm 1 MAPCFG(CFG_{ir},CFG_{bin})

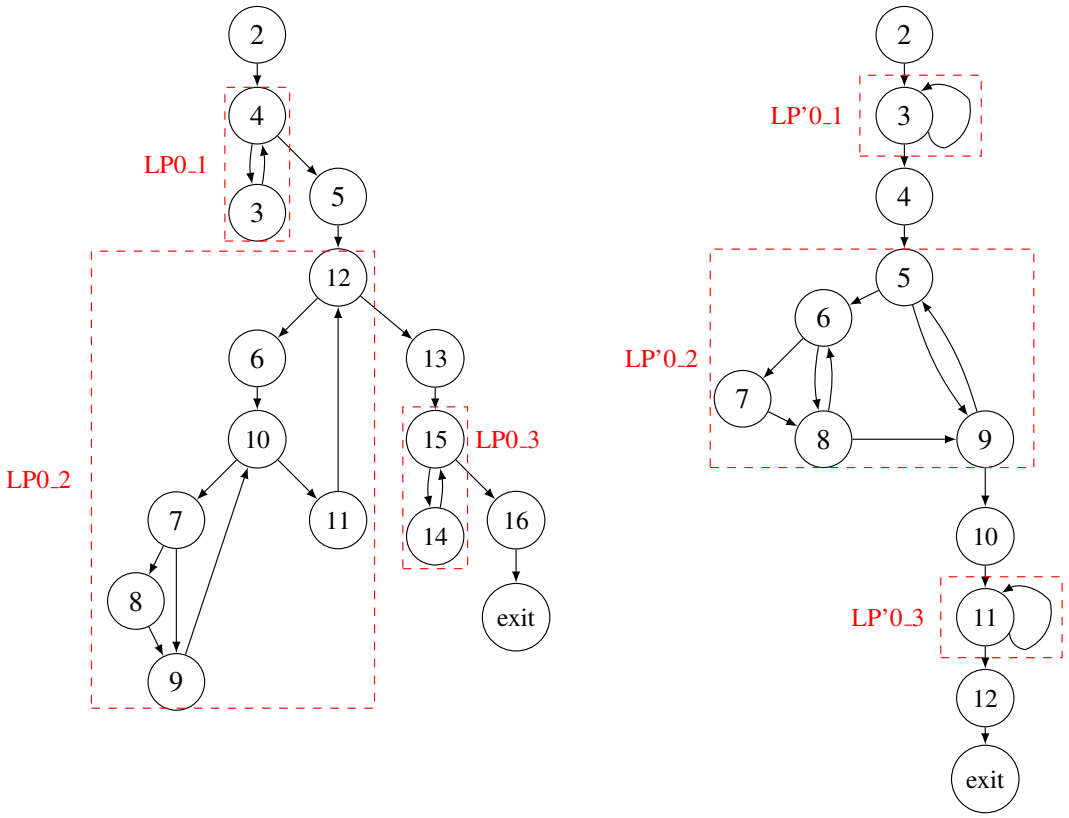
for $x \in \{ir, bin\}$ **do**
  $SCC_x \leftarrow$ EXTRACTSCC(CFG_x)
  ▷ *SCC_x is a set of CFGs, CFG_x is a CFG*
  $SCC_{x\_loop} \leftarrow$ EXTRACTLOOP(SCC_x)
  ▷ *SCC_{x_loop} is a set of CFGs*
  $SCC_{x\_cont} \leftarrow$ CONTRACTSCC(SCC_{x_loop})
  ▷ *SCC_{x_cont} is a set of single-node CFGs*
  $CFG_{x\_cond} \leftarrow$ CONDENSECFG(CFG_x, SCC_{x_cont})
  ▷ *CFG_{x_cond} is a directed acyclic graph*
end for
MATCH($CFG_{ir\_cond}, CFG_{bin\_cond}$)
▷ *builds mapping ($\Leftrightarrow$) between IR and binary SCCs*
for all $\{scc_{ir}, scc_{bin}\} \in (SCC_{ir\_loop}, SCC_{bin\_loop})$ **do**
  if $scc_{ir} \Leftrightarrow scc_{bin}$ **then**
    REMOVEBACKARCS(scc_{ir}, scc_{bin})
    MAPCFG(scc_{ir}, scc_{bin})
  end if
end for



(a) IR CFG (b) Bin CFG

Figure 4.9: The IR CFG (a) and the binary CFG (b) of the BubbleSort example

are: $SCC_1 = \{bb2\}$, $SCC_2 = \{bb3, bb4\}$, $SCC_3 = \{bb5\}$, ..., $SCC_8 = \{bbExit\}$ as indicated in fig. 4.9-(a).



(a) IR CFG (b) Bin CFG

Figure 4.10: Identification of loop blocks in the IR and binary CFGs of the BubbleSort example

ii) Second, we identify SCCs that are loop blocks among the extracted SCCs (line 4). A SCC with at least one arc is a loop block.

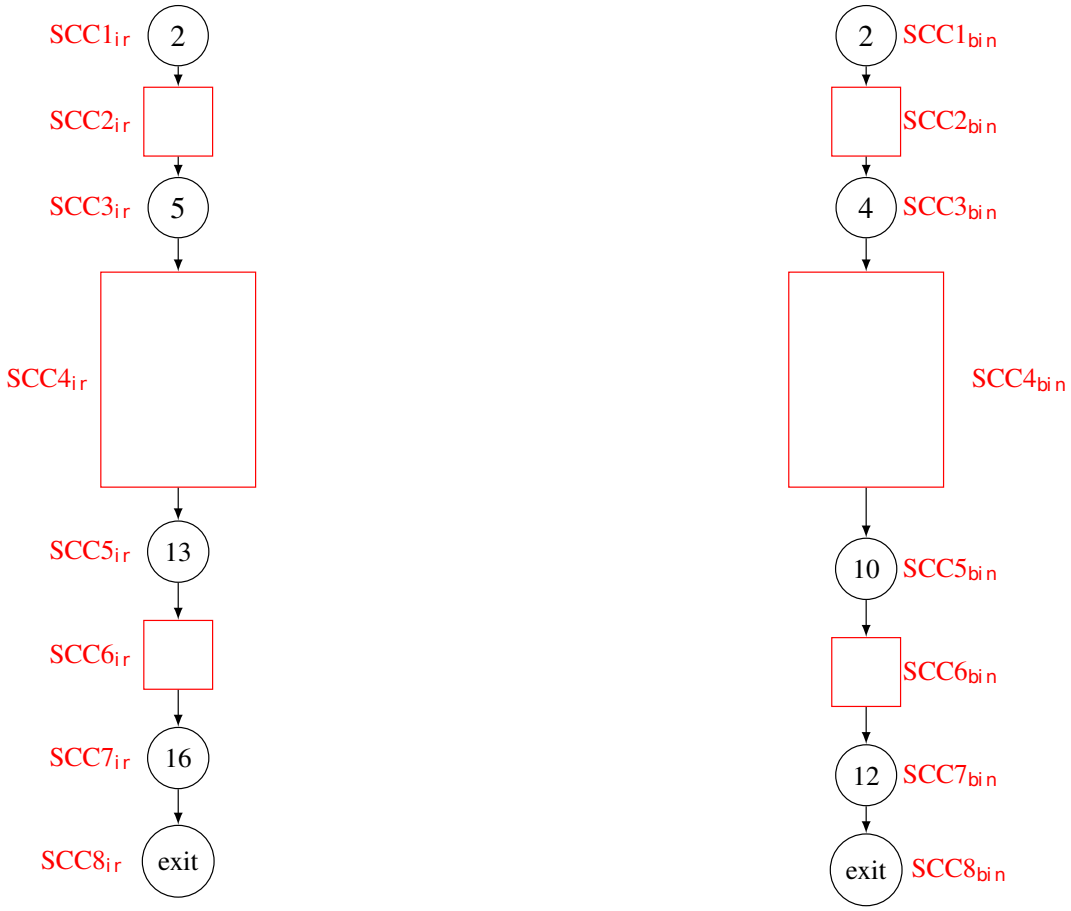
Fig. 4.10 highlights the IR and binary loop blocks after step ii. Among the SCCs extracted at step i, there are three SCCs, in each CFG, that are composed of at least one arc. These loop blocks represent level-0 loops (*i.e.* outermost loops) that we call $LP0\_i$ for the IR and $LP'0\_i$ for the binary.

iii) Third, SCCs with at least one arc (*i.e.* loop blocks) are contracted into a single node (line 6).

At this level of the algorithm, we are not interested in the basic blocks that constitute a loop block, but we are rather interested in the loop block as a whole. That is why we abstract away these basic blocks and consider a loop block as a single node (fig. 4.11).

iv) Fourth, we reconnect the SCCs again forming a new CFG that is the *condensation* of the original CFG (line 8). This resulting CFG is a directed acyclic graph since its nodes are SCCs.

Fig. 4.11 delineates the condensed IR and binary CFGs.



(a) IR CFG (b) Bin CFG

Figure 4.11: Condensed IR and binary CFGs of the BubbleSort example

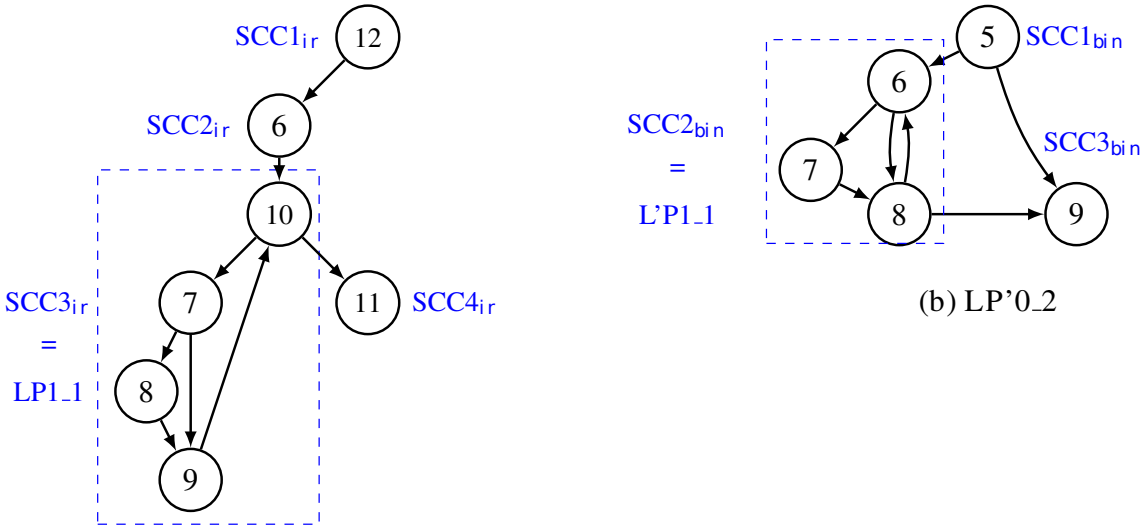
v) In the fifth step, we match the binary and IR condensed CFGs (line 11). The mapping process becomes easier as we contracted the loops into single nodes. At this level, basic blocks outside level-0 loops, as well as level-0 loops, are matched. The way our matching scheme works is to find as many fixed points as possible. A fixed point, as explained before, is an SCC in the IR-CFG that has a surefire corresponding SCC in the binary CFG, which is also considered as a fixed point. Entry SCCs, which are fixed points, are insufficient in propagating fixed points across the CFGs. So, we need to find more of them. Under the assumption that the number of loops in the IR remains the same in the binary (this assumption may not hold when using optimization level $O3$ as several loop optimizations that are susceptible of changing the number of loops are activated, whereas, in this section, we focus on optimization level $O2$), each loop in the IR will map to exactly one loop in the binary. So, if we find out which loop in the IR corresponds to which one in the binary we will have loop blocks as fixed points which will facilitate the mapping process of the whole CFG. To do so, we use debug information to match each loop in the IR code to its corresponding loop in the binary. And since, in our case, there is only one possible match for each loop block in the IR, debug information is conclusive. Thus, loop blocks are matched and considered as fixed points.

   More fixed points can be created iteratively starting with the already-established fixed

points (loop blocks and the entry SCCs) and the relation $PRED(SCC)$ (resp. $SUCC(SCC)$), which returns the immediate predecessor (resp. successor) of a given fixedpoint (eg. $PRED(LP0\_2) = SCC3_{ir} = bb5_{ir}$, $SUCC(LP0\_1) = SCC3_{ir} = bb5_{ir}$, $PRED(LP'0\_2) = SCC3_{bin} = bb4_{bin}$, $SUCC(LP'0\_1) = SCC3_{bin} = bb4_{bin}$ so $bb5_{ir} \Leftrightarrow bb4_{bin}$), until no more fixedpoints can be found.

Some instructions contained in the basic blocks (such as function calls, exit and return statements) along with debug information may be used as clues to curtail the ambiguity that may arise when designating fixed points and thus improve the accuracy of the mapping. This may sound contradictory as we previously denounced the use of debug information. However, our mapping scheme does not rely entirely on debug information. Instead, debugging is only used to solve equivocal cases.

The mapping between the condensed IR and the binary CFGs in our example, considering loop blocks as fixedpoints and using the $PRED/SUCC$ relations, is straightforward: $bb2_{ir} \Leftrightarrow bb2_{bin}$, $LP0\_1 \Leftrightarrow LP'0\_1$, $bb5_{ir} \Leftrightarrow bb4_{bin}$, $LP0\_2 \Leftrightarrow LP'0\_2$, $bb13_{ir} \Leftrightarrow bb10_{bin}$, $LP0\_3 \Leftrightarrow LP'0\_3$, $bb16_{ir} \Leftrightarrow bb12_{bin}$, $bbEXIT_{ir} \Leftrightarrow bbEXIT_{bin}$. To differentiate between basic blocks from the IR CFG and the binary CFG, we will use (respectively) this notation (only when an ambiguous situation arises): bbi_{ir} and bbi_{bin}.



(a) LP0_2

(b) LP'0_2

Figure 4.12: Recursive Mapping of SCCs

In order to take into account basic blocks that constitute the level-0 loop, we have to recursively apply the same process (steps i, ii, iii, iv and v) but for loop blocks instead of the entire CFGs (line 13):

vi) Sixth, we start by removing the back arcs of the loop block, in that if we keep them we cannot decompose it into SCCs, by definition of a SCC (line 15).

We take $LP0\_2$ and $LP'0\_2$ as examples from fig. 4.10, to demonstrate the recursive mapping process. Back arc $bb11_{ir} \rightarrow bb12_{ir}$ is removed from $LP0\_2$. Similarly, back arc $bb9_{bin} \rightarrow bb5_{bin}$ is removed from $LP'0\_2$. The result is shown in fig. 4.12.

vii) Seventh, we apply i, ii and iii on the modified loop block (line 1).

The SCCs of the "modified" subgrapghs $LP0\_2$ and $LP'0\_2$ after removing their back arcs are highlighted in fig. 4.12. $LP0\_2$ and $LP'0\_2$ each enclose one nested loop, $SCC3_{ir}$ and $SCC2_{bin}$ that we contracted to $LP1\_1$ and $LP'1\_1$ (respectively). We will have to apply steps vi and vii on these loops as well. We repeat the same process for each loop level in the CFGs until no more SCCs with at least one arc are found (in other words, no more loops are found).

There are less SCCs in $LP'0\_2$ than in $LP0\_2$ due to compiler optimizations. To reflect these optimizations in the IR, certain nodes in $LP0\_2$ will remain unmapped. To decide which nodes will be unmapped, we start first by matching the loop blocks as they constitute fixed points at this level of the recursion from which we will infer more: $LP1\_1 \Leftrightarrow LP'1\_1$. $PRED(LP1\_1) = bb6, PRED(LP'1\_1) = bb5$. So, $bb6_{ir} \Leftrightarrow bb5_{bin}$ ($bb6_{ir}$ and $bb5_{bin}$ become fixed points). $SUCC(LP1\_1) = bb11, SUCC(LP'1\_1) = bb9$. So, $bb11_{ir} \Leftrightarrow bb9_{bin}$ ($bb11_{ir}$ and $bb9_{bin}$ become fixed points).

As a result, $bb12_{ir}$ remains unmapped. We then, apply vi and vii on $LP1\_1$ and $LP'1\_1$. $SCC(LP1\_1) : \{bb10, bb7, bb8, bb9\}$, $SCC(LP'1\_1) : \{bb6, bb7, bb8\}$.

Here, we also have more SCCs in $LP1\_1$ than in $LP'1\_1$. In this case, we have a branch inside the loop blocks. In the IR (fig. 4.12-(a)), the loop header ($bb10$) and the branch statement ($bb7$) are in two separate basic blocks. However, in the binary (fig. 4.12-(b)), they are grouped together in one block $bb6$. To reflect this optimization, the basic block containing the branch statement in the IR will be mapped to the basic block of the branch and the loop in the binary: $bb7_{ir} \Leftrightarrow bb6_{bin}$. The branch targets in the IR are mapped to their equivalents in the binary: $bb8_{ir} \Leftrightarrow bb7_{bin}$, $bb9_{ir} \Leftrightarrow bb8_{bin}$. $bb10_{ir}$ will stay unmapped.

Table 4.1: The mapping data base of the example of fig. 4.9

IR	bb2	bb3	bb4	bb5	bb6	bb7	bb8	bb9
BIN	bb2	∅	bb3	bb4	bb5	bb6	bb7	bb8
IR	bb10	bb11	bb12	bb13	bb14	bb15	bb16	exit
BIN	∅	bb9	∅	bb10	∅	bb11	bb12	exit

Table 4.1 illustrates the mapping data base resulting from applying mapping algorithm 1 on the CFGs in fig. 4.9.

By following these mapping steps, multi-level loops can be easily matched because the problem of dealing with complicated loop structures always falls back to dealing with a one-level loop thanks to the *contraction* method. The only challenge we may face with a loop is when it encompasses branches (like in our example when the loop header and the branch statement are amalgamated in one block in the binary). Loops with a condition-free body are relatively easy to match: if the number of basic blocks in the loop block is the same in the IR and the binary code then the mapping is straightforward. Otherwise, if the basic blocks in the IR outnumber the ones in the binary code, certain blocks will remain unmapped in the IR. In the opposite case, several basic blocks of the binary code will be mapped to one basic block in the IR.

In this section, we proposed an annotation framework that aims at accurately placing annotations in the high-level IR code using a mapping algorithm that takes into account *O2* compiler optimizations. The algorithm pays particular attention to loop structures as they

considerably affect the accuracy of the estimations. The mapping scheme allows us to reflect the behavior of the optimized binary code onto the IR. There are, however, extreme cases where the mapping cannot take into account all the optimizations in the binary code. These cases occur when a node in the binary code has more (ingoing or/and outgoing) arcs than its corresponding node in the IR. These arcs represent possible execution paths that the code may take. These paths are not represented in the IR as we do not introduce any structural modification to the original IR (we neither add nor reduce any arcs or nodes) at this level.

4.3.2 Upgraded Mapping Scheme: Tackling Aggressive Compiler Optimizations

The mapping algorithm, described in subsection 4.3.1, succeeds in dealing with the majority of structure modifications caused by optimizations enabled at level *O2* without changing the IR CFG. Thus, it provides an accurate mapping between an IR CFG and its corresponding binary CFG compiled with the *gcc -O2* optimization level where the number of loops in the IR is identical to the one in the binary. Experiments on the instruction count, which are detailed in the experimentation chapter (chapter 6, subsection 6.3.1), prove the accuracy of the algorithm by showing a small error value (an average of 2%). However, at the *O3* optimization level, aggressive compiler optimizations, such as loop unrolling, are activated. These optimizations can radically change the CFG and may lead in certain cases to a different number of loops between the IR and the binary code. So, obtaining accurate results by merely applying the proposed algorithm without performing any modification on the IR CFG is simply not feasible.

The Inadequacy of Algorithm 1 to Deal with *O3* Level Optimizations

Fig. 4.13 shows a simple program (consisting of the addition of two arrays inside a loop with an unknown trip count) at three different compilation stages: before any compiler optimization is conducted (fig. 4.13-(a)), after machine-independent optimizations are performed (fig. 4.13-(b)) and after all (i.e. front-end and back-end) compiler optimizations are carried-out (fig. 4.13-(c)). These figures were obtained using *gcc -O3*.

The figure captures the salient effects of compiler optimizations on the different CFGs. In addition to the optimizations automatically enabled at level *O3*, we also enabled an optimization called *Strip-mining* by activating a *Graphite* (a framework that brings more high-level loop optimizations to Gimple) flag. *Strip-mining* affects the structure of the code by transforming a single loop into a nested loop. This transformation is noticeable in the IR code (fig. 4.13-(b)) by the presence of an additional loop compared to the source code (node 6).

This optimization is no exception as several others (loop fusion, loop distribution, complete unrolling of small loops, loop unswitching, just to name a few) can be enabled. These optimizations can radically change the structure of the code by introducing major transformations to loops resulting, in many cases, in the appearance/disappearance of new/existing loops [BEP04].

Fig. 4.13-(b) and fig. 4.13-(c) show the same number of loops in the IR and the binary code after *strip-mining*, but the two CFGs are different due to the unrolling of the inner loop. We will apply algorithm 1 on the example of fig. 4.13 to show its limitations when used on an *O3* optimized code. First, the IR and binary CFGs are decomposed into strongly connected components. For example, fig. 4.13-(b) is composed of 5 SCCs: $SCC1_{ir} = \{bb2\}$, $SCC2_{ir} = \{bb4\}$, $SCC3_{ir} = \{bb5, bb6, bb7\}$, $SCC4_{ir} = \{bb3\}$ and $SCC5_{ir} = \{exit\}$, among which, $SCC3_{ir}$ is a level-0 loop. Fig. 4.13-(c) is also composed of 5 SCCs: $SCC1_{bin} = \{bb2\}$,

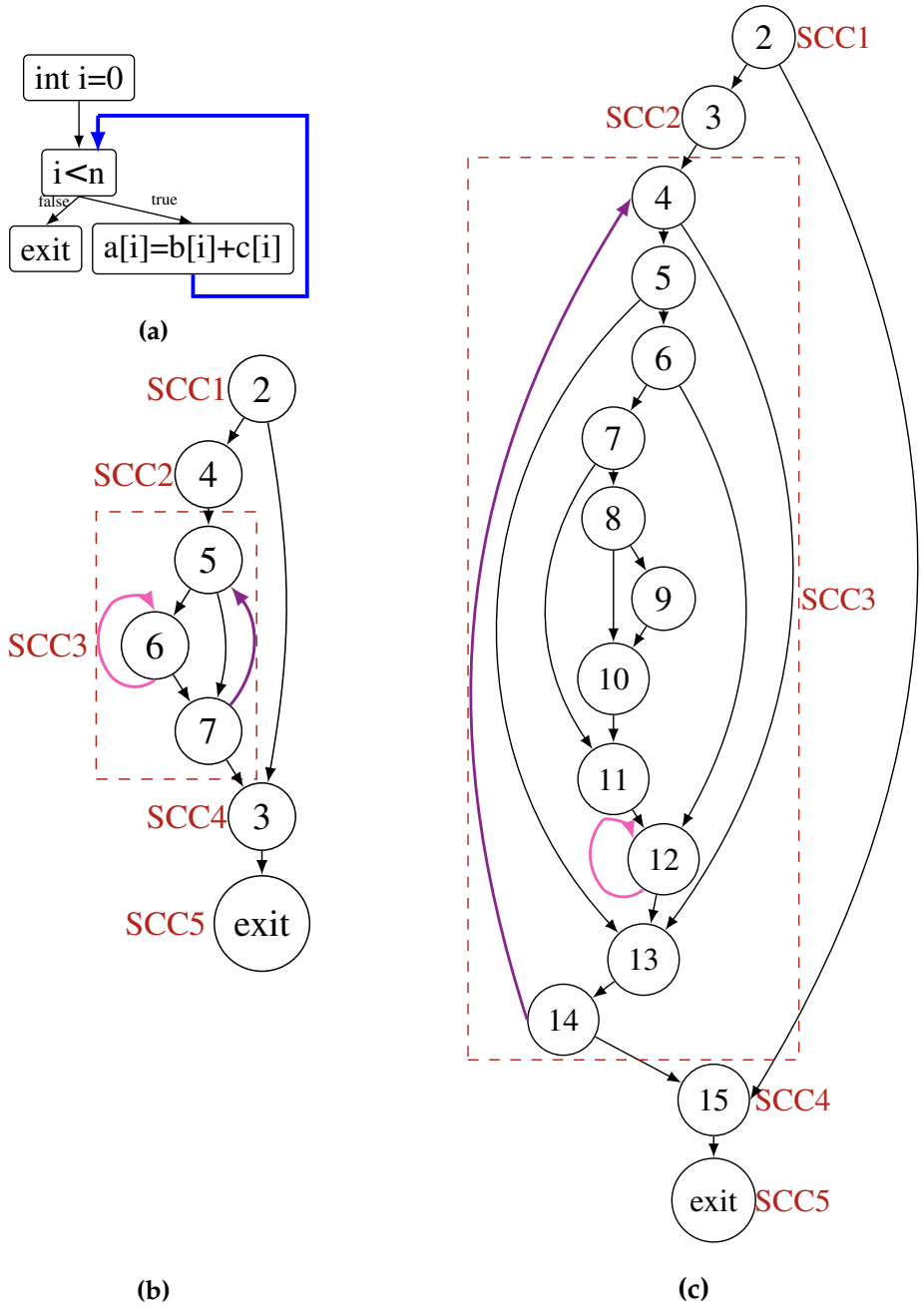


Figure 4.13: Control flow graphs of **(a)** source code, **(b)** intermediate representation (IR) and **(c)** the binary code compiled using the highest level of compiler optimizations (*gcc -O3*)

$SCC2_{bin} = \{bb3\}$, $SCC3_{bin} = \{bb4, bb5, \ldots bb14\}$ (a level-0 loop), $SCC4_{bin} = \{bb15\}$ and $SCC5_{bin} = \{exit\}$.

Then, the SCCs are reconnected forming new acyclic CFGs that are the condensation of the original CFGs. The new CFGs have the extracted SCCs as nodes, which facilitates the mapping process as loops are contracted into single nodes. The condensed IR CFG of fig. 4.13-(b) is: $SCC1_{ir} \Rightarrow SCC2_{ir} \Rightarrow SCC3_{ir} \Rightarrow SCC4_{ir} \Rightarrow SCC5_{ir}$.

After CFG condensation, the different SCCs of both CFGs are matched using fixed points. Entry SCCs of fig. 4.13-a and fig. 4.13-b are fixed-points: $SCC1_{ir} \Leftrightarrow SCC1_{bin}$. Loops are also

considered as fixed points (binary and IR have equal numbers of loops at the *O2* optimization level): $SCC3_{ir} \Leftrightarrow SCC3_{bin}$.

To propagate fixed points throughout the CFGs, the already established fixed points are used along with the immediate predecessor/successor relations. For example: $PRED(SCC3_{ir}) = SCC2_{ir}$, $PRED(SCC3_{bin}) = SCC2_{bin}$, $SUCC(SCC1_{ir}) = SCC2_{ir}$, $SUCC(SCC1_{bin}) = SCC2_{bin}$ so $SCC2_{ir} \Leftrightarrow SCC2_{bin}$ ($SCC2_{ir/bin}$ are considered as fixed-points).

These same steps (SCC extraction, loop block contraction, CFG condensation and fixed points designation) are applied recursively on SCCs that are loop blocks until no more loops can be found and we reach the granularity of basic blocks. Before doing so, back arcs of loop blocks have to be removed ($bb7 \rightarrow bb5$ in loop block $SCC3_{ir}$ and $bb14 \rightarrow bb4$ in loop block $SCC3_{bin}$). Then, these loop blocks are in turn decomposed into SCCs and fixed points are pinpointed ($bb6_{ir}$ and $bb12_{bin}$ are fixed-points).

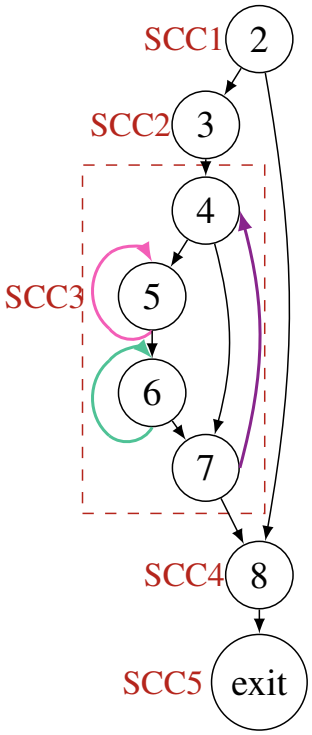


Figure 4.14: Loop unrolling with *LLVM*

However, at the *O3* optimization level, loop unrolling introduces several transformations to the IR ($SCC3_{ir}$ and $SCC3_{bin}$ have very different structures) and can cause the appearance of an additional loop in the binary in case of the *LLVM* compiler (fig. 4.14). Fig. 4.14 shows the binary CFG of the same example in fig. 4.13-(a) compiled with *LLVM -O3*. An additional loop ($bb6_{bin}$) is introduced in the binary CFG because of loop unrolling.

Applying the mapping steps on $SCC3^{gcc}_{ir/bin}$ (fig. 4.13-(b) and fig. 4.13-(c)), which are loop blocks, leads to unmapped basic blocks ($bb5^{gcc}_{bin}$, $bb6^{gcc}_{bin}$,...,$bb11^{gcc}_{bin}$ and $bb13^{gcc}_{bin}$ have no

match in the IR). These unmapped basic blocks are the result of *partial loop unrolling*. A transformation called *peeling* is performed by *gcc* during the loop unrolling optimization. Peeled instructions are placed before the unrolled loop, $bb12^{gcc}_{bin}$, as a *prologue* (fig. 4.13-(c)), which explains the difference between $SCC3_{ir}$ and $SCC3^{gcc}_{bin}$

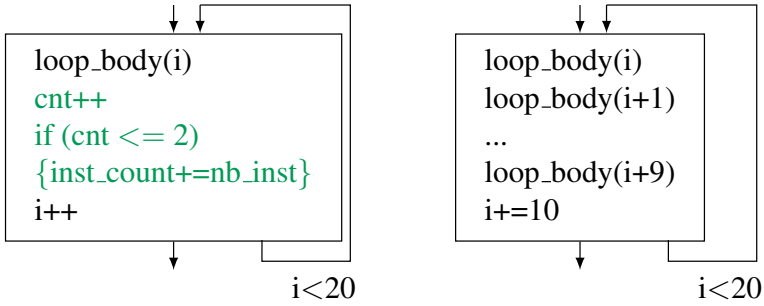
LLVM, on the other hand, gathers the peeled instructions in a loop block and places it as an *epilogue*, $bb6^{llvm}_{bin}$ (fig. 4.14). If we apply the same algorithm on fig. 4.13-(b) and fig. 4.14, we will end up with an unmapped loop block: $bb6^{llvm}_{bin}$ has no match in fig. 4.13-(b).

In addition to this obvious structural CFG transformation, another modification is present inside the unrolled loop. In fact, the unrolled loop has more instructions than the original loop, but it requires less iterations. This is true for *gcc* as well as *LLVM*. In the remainder of this section, we propose an approach to troubleshoot this mapping limitation.

Loop Unrolling

Loop unrolling is a compiler optimization that replicates the loop body UF (unrolling factor) times. Consequently, the unrolled loop requires less iterations than the original loop and the loop counter is modified fewer times. Unrolling also reduces the number of branch instructions and exposes more possibilities of ILP. The UF is chosen by the compiler based on the loop trip count. Loops with a small number of iterations (a maximum of 17 iterations in *gcc*) are fully unrolled (a.k.a. peeled) before the back-end. This transformation leads to the disappearance of the loop structure and is visible at the IR level.

Loops with bigger or unknown number of iterations are partially unrolled by the back-end. Partial loop unrolling leads to different CFG transformations depending on whether or not the loop iteration count is known at compile time. In the best case, the control flow is unchanged, but keeping the same loop bound of the IR leads to erroneous estimates. In most cases, however, IR and binary CFGs become different.



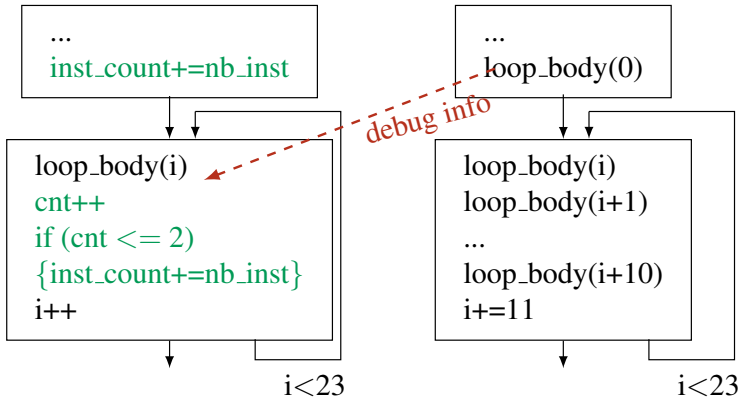
(a) IR loop (max itr bound=20) (b) Unrolled binary loop (max itr bound=2, UF=9)

Figure 4.15: The trip count is a multiple of $(UF + 1)$

• In case the loop trip count is known at compile time, the IR CFG and the binary CFG are identical, as shown in fig. 4.15. So, a graph matching tool would be tempted to apply a straightforward mapping algorithm without considering the modification of the iteration bound, which will inevitably lead to overestimations. Two cases should be taken into account:

a) If the loop trip count is a multiple of $(UF + 1)$ then the loop body is duplicated exactly $(UF + 1)$ times in the binary. In the example of fig. 4.15, the trip count is 20 and the UF is 9. The loop body is thus duplicated 10 times in the binary code, which

reduces the number of loop iterations tenfold. Using the instruction count as a performance metric and assuming a straightforward mapping, if we annotate the IR loop with the number of instructions of the corresponding binary loop, the instruction execution count after natively simulating the IR will be multiplied $(UF + 1 = 10)$ times, which is a considerable error. To fix this problem, we add a counter and a test in the IR loop body. The test is only entered when the counter satisfies the following condition: $cnt <= (loop\_trip\_count/(UF + 1))$.
Placed under this condition, the instruction count will yield accurate results (see chapter 6).



(a) IR loop (max itr bound=23) (b) Unrolled binary loop (max itr bound=2, UF=10, first itr peeled)

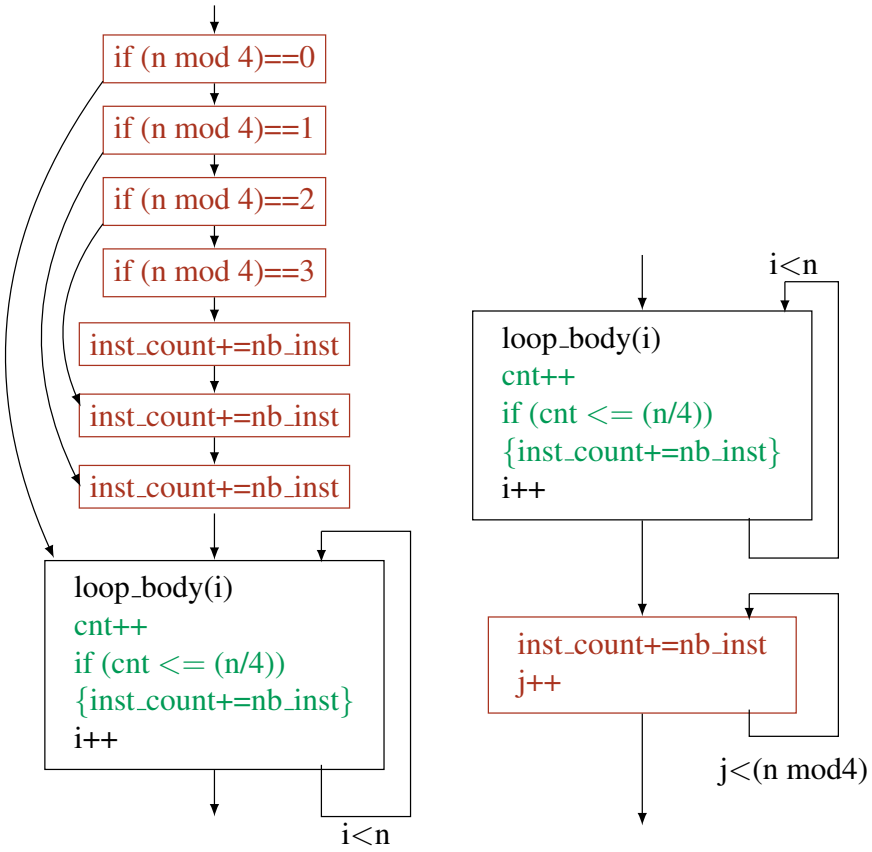
Figure 4.16: The trip count is not a multiple of $(UF + 1)$

b) If the loop trip count is not a multiple of $(UF + 1)$, then the loop body is unrolled $(UF+1)$ times and the leftover iteration(s) $(trip\_count\%(UF+1))$, which might be the few first or last iteration(s), are moved outside of the loop and placed in a prologue or epilogue. In the example of fig. 4.16, only the first iteration of the loop is peeled by the compiler and placed in a prologue.

   Mapping approaches based on debug information, such as line reference, would map the peeled iterations, although placed outside the binary loop, inside the IR loop, which leads to overestimation. Our solution is to proceed as in a) and to deal with the peeled iterations outside the loop body. In the example of fig. 4.16-(a), the instruction count is incremented with the number of peeled instructions in the predecessor of the basic block containing the loop.

• In the very common case in which the loop trip count is unknown at compile time, the backend generates a complex structure preceding the unrolled loop in the binary CFG, as shown in fig. 4.13-(c) (the unrolled loop is the innermost loop, which is $bb12$). In fact, when the trip count is only available at execution time, the compiler cannot decide if it is a multiple of $(UF + 1)$ or not at compile time. As a result, it always adds a prologue/epilogue for the remaining iterations (*gcc*, for instance, adds a prologue).

In fig. 4.13-(c), only the innermost loop is unrolled (which is always the case). The prologue ($bb6 \rightarrow bb11$) is more intricate than the prologue in fig. 4.16-(b) because the compiler does not know the exact number of iterations that should be performed before entering the unrolled loop body. Hence, a few tests (fig. 4.13-(c): $bb6 \rightarrow bb8$), one for each possible excess, are carried out. The number of tests depends on the UF. When the trip count is unknown, *gcc*



(a) adding a prologue with if statements to the IR (b) adding an epilogue with a loop to the IR

Figure 4.17: The trip count is unknown at compile time

sets the UF to 7. For space reason, we changed the UF to 3 for the example of fig. 4.13-(c) and fig. 4.14. So the possible number of the remaining iterations is: $(trip\_count\%(UF+1)) = n\%4 = \{0,1,2,3\}$.

The prologue in the binary code has no equivalent in the IR. So, basic blocks in the prologue cannot be mapped to any basic block in the IR as no match exists. Leaving these basic blocks unmapped may lead to underestimations. The solution is to add a *structurally equivalent* prologue that has no functional purpose (i.e. no peeling is really performed) in the IR (fig. 4.17-(a)). Basic blocks in the added prologue will hold non-functional information extracted from the actual prologue. Fig. 4.17-(a) corresponds to *bb6* in fig. 4.13-(b) to which we added a prologue (represented by the red rectangles). The instruction count is incremented, in the added prologue, according to the number of peeled iterations, which is decided at run time.

LLVM, on the other hand, handles loop unrolling with an unknown trip count in a different way. A new loop is created as an epilogue in the binary to account for the remaining iterations. In this new loop, the last *few* iterations $(trip\_count\%(UF+1))$ of the original loop are executed. As shown in fig. 4.17-(b), we add a new loop in the IR to reflect this modification.

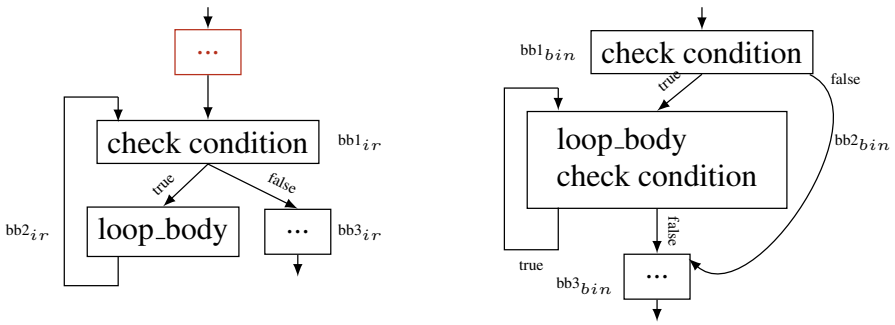
For clarity reasons, the loop example that we use to explain our mapping approach is made of straight-line code (i.e. the loop body is contained in one basic block, there are no branches). Consequently, the peeled iterations are also contained in one basic block each (eg.

the three basic blocks right above the loop block in fig. 4.17-(a)), which makes the structure of the prologue/epilogue relatively simple. However, if the loop body contains convoluted control flow, the epilogue/prologue added by the compiler will not be as simple as in our example since each peeled iteration will have the same control flow as the loop body. Either way, the peeled iterations added in the IR should replicate the structure of those in the binary since our main goal is to facilitate the mapping process by bringing the IR CFG as close as possible (structure-wise) to the binary one.

It should be noted that the additional basic blocks that we introduce in the IR CFG are not aimed at changing the functional behavior of the code.

Miscellaneous Optimizations

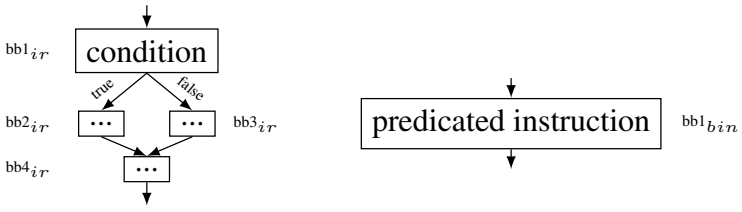
Other optimizations that are enabled at the *O3* level and that change the code structure are discussed in this section. Optimizations that preserve the control flow (peephole optimizations, instruction splitting, instruction scheduling, etc.) do not need a particular mapping strategy.



(a) while loop (b) do-while loop

Figure 4.18: Loop Inversion

Loop inversion (fig. 4.18) is a compiler optimization that transforms a *while* loop to a *do-while* loop (it moves the loop-test from before the loop body to after the loop body) in order to reduce the number of jumps. The *do-while* loop is wrapped in an *if-statement*. In fig. 4.18-(b), the loop is no longer composed of two basic blocks as in fig. 4.18-(a). The loop body and the loop-test are gathered in one basic block. This block ($bb2_{bin}$) is mapped to $bb2_{ir}$. As for $bb1_{ir}$, it remains unmapped and we add a new basic block (the red rectangle) immediately before $bb1_{ir}$ to account for $bb1_{bin}$.



(a) If-then-else in the IR (b) If-conversion in the binary

Figure 4.19: If conversion

If-conversion is a back-end optimization that consists of converting conditional branches into predicated instructions supported by the target instruction set architecture. One of the

features of ARM ISA, for example, is that almost all instructions are predicated. Consider the simple C code:

```
1 if (a > 15) {
2   a = 15;}
3 else {
4   a = a + 1;}
```

This produces the following machine code:

```
1 cmp     r0, #15
2 movhs   r0, #15
3 addlo   r0, r0, #1
```

In fig. 4.19, the *if-then-else* statement composed of three basic blocks in the IR (a *bb* for the conditional statement and one *bb* for each condition outcome) is converted to a single basic block containing predicated operations in the binary code. Mapping these two different structures in fig. 4.19 doesn't entail any modification in the IR. We simply map $bb1_{bin}$ to $bb1_{ir}$ and leave $bb2_{ir}$ and $bb3_{ir}$ unmapped.

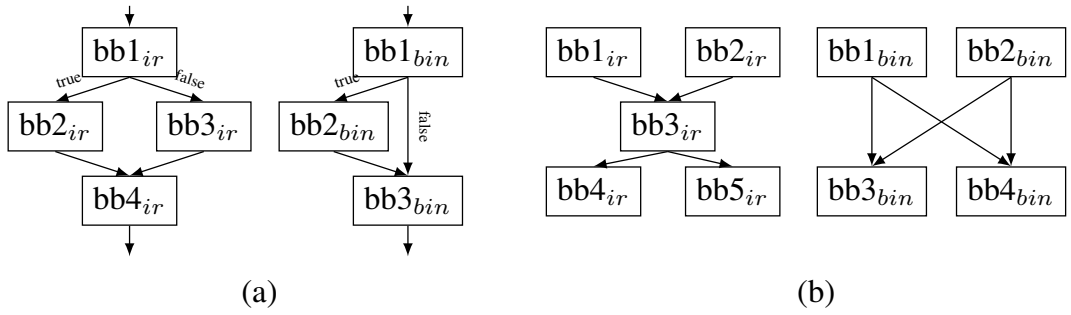


Figure 4.20: Other branch optimizations

Other branch optimizations like the examples in fig. 4.20 may be conducted by the compiler and lead to further mismatches between IR and binary CFGs. These optimizations boil down to basic blocks being added to or removed from the code by the compiler. For example, the mapping breakdown of the two CFG snippets in fig. 4.20-(a) is: $bb1_{ir} \Leftrightarrow bb1_{bin}$, $bb2_{ir} \Leftrightarrow bb2_{bin}$, $bb4_{ir} \Leftrightarrow bb3_{bin}$ and $bb3_{ir}$ remains unmapped. The opposite case where the right side of fig. 4.20-(a) is the IR CFG and the left side is the binary CFG may also take place. To handle this case, a basic block is added in the IR to account for the extra basic block in the binary. As for fig. 4.20-(b), the mapping is: $bb1_{ir} \Leftrightarrow bb1_{bin}$, $bb2_{ir} \Leftrightarrow bb2_{bin}$, $bb4_{ir} \Leftrightarrow bb3_{bin}$, $bb5_{ir} \Leftrightarrow bb4_{bin}$ and $bb3_{ir}$ remains unmapped.

If the hardware loop feature (a.k.a. zero overhead loop) is supported by the target architecture, the compiler chooses loops that are eligible to be executed as hardware loops (only innermost loops with no control transfer except for the loop branch) in order to minimize cycle cost overhead induced by pipeline stalls due to jump instructions. Indeed, the hardware module dedicated specifically to loop execution (a loop counter, a register for the number of instructions, a pointer to next instruction, a loop buffer, etc.) eliminates branch logic in the loop. As for the structural changes, a minor transformation is observed (according to the experiments we conducted using the Kalray processor [DdDAB+13], which supports the HW loop feature). One basic block is added as a pre-header. All incoming arcs to the loop basic block are redirected to the added pre-header block. So, all we need to do is to add an

identical block right before the IR loop header that will hold the non-functional information extracted from the corresponding binary basic block.

After introducing the above mentioned transformations in the IR CFG, algorithm 1 can be efficiently applied, especially that loop unrolling is no longer a hindrance in propagating fixed-points all the way through the CFGs.

4.4 Conclusion

In this chapter, we proposed an IR-level annotation framework that takes into consideration front-end compiler optimizations, thanks to the choice of the IR level, and back-end optimizations with the help of the proposed CFG matching strategy. Although, we decided to meet the binary code halfway by using the IR instead of the source code, a straightforward mapping is not guaranteed, especially when loop optimizations are performed. Hence, the necessity for an accurate mapping approach.

The proposed mapping approach is architecture independent, thus retargetable. It consists of matching the binary and the IR CFGs at a basic block level using *fixedpoints* and propagating them throughout the CFGs. The mapping is conducted recursively between SCCs of the binary and IR CFGs until reaching the basic block level.

We transformed loop blocks from a puzzling issue into a facilitator in the mapping process by considering them as fixedpoints. This is valid under the assumption that the number of loops remains unchanged from IR to binary generation, which is true with *O2* optimizations.

With the presence of aggressive compiler optimizations such as *loop unrolling*, which are activated at the *O3* optimization level, the proposed mapping algorithm fails at generating accurate mapping information. Some optimizations entail many transformations of the control flow leading to a binary CFG completely different from the original code. To circumvent this problem, we proposed to change the structure of the IR (without modifying its functional behavior) merely by replicating the binary CFG portions that have no equivalents in the IR. Applying the proposed mapping algorithm on the binary CFG and the modified IR CFG yields accurate mapping results according to the instruction count experiments presented in chapter 6.

Chapter 5

Modeling the Impact of Cache Memories on the System Performance

MPSoCs continuously incorporate more powerful CPUs and CPU subsystems with advanced features, such as fast instruction and data caches, prefetch mechanisms, branch prediction mechanisms, out-of-order or VLIW capabilities, etc., in order to further increase their speed. Non-functional aspects are tightly influenced by the behavior of the micro-architectural components and their inter-dependency with software. On that account, obtaining sufficiently accurate performance estimates requires accurate and fast modeling techniques of the system components. In this chapter, we will start first by describing the modeling and dynamic performance estimation process using a *classic* data cache model as an example. Then, we will detail the proposed performance model of an instruction cache and an instruction buffer of a VLIW architecture after giving an overview of such architecture and highlighting its distinctive features.

5.1 Data Cache Performance Estimation

Cache behavior is affected by software and its content is dictated by the execution path followed by the program during run time. This dynamic nature of caches makes static approaches not very accurate at estimating cache hit/miss rates and computing the delay they incur. It is hard to know the behavior of the program before its execution and it is quite impossible to delimit all possible paths that might be taken by the program. Given the pivotal role played by caches in enhancing the performance of embedded systems, simulating their behavior has become a key factor to accurate performance estimates.

The purpose of this section is to explain the concepts of a performance model of a micro-architectural component and how it is coupled with the functional model, through an annotation function, in order to perform dynamic performance estimation. The micro-architectural component serving as an example is the data cache, which has been abundantly studied in literature. So, both the data cache model and the annotation function presented in this section are similar to what has been proposed in literature. We will also touch on the major issue with data cache performance estimation, which is the retrieval of data addresses. This issue has already been tackled in previous works leading to various solutions, which we will describe briefly.

Simulation of data caches allows the imitation of the dynamic behavior of the real cache, in terms of hits and misses, without performing any transaction between main memory and

the cache model. In other terms, there is no actual data stored in the cache model as it is only leveraged for performance estimation and not for functional simulation. As for the task of obtaining correct functional results, it is the responsibility of the host machine running the simulation, which natively takes care of the actual data. In order to re-create the data cache behavior of the target platform and yield accurate performance estimates with minimum simulation overhead, several modeling choices need to be made.

5.1.1 Data Cache Model

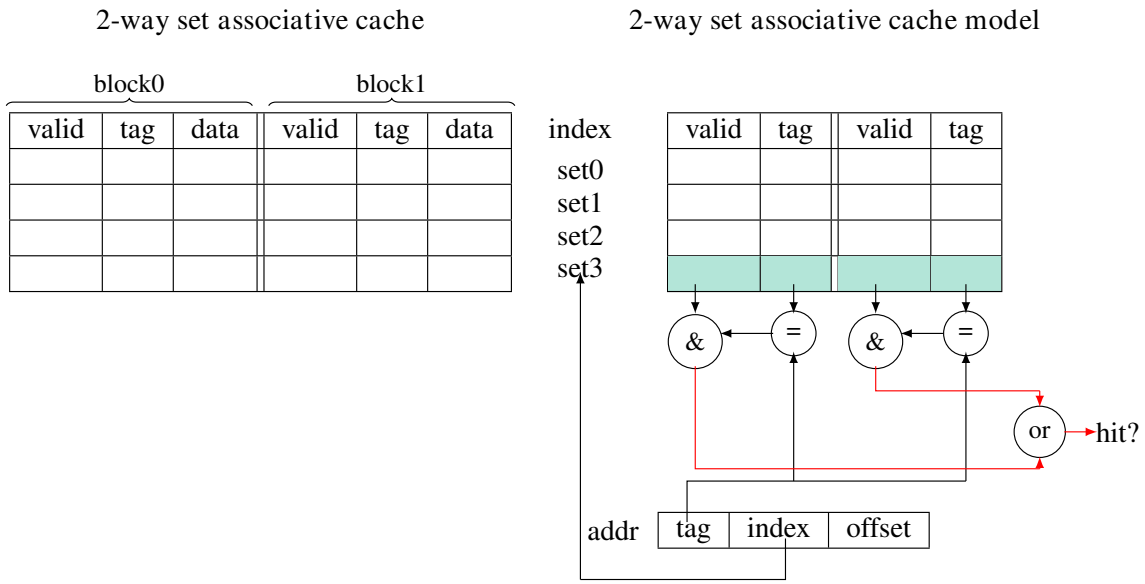


Figure 5.1: An example of a 2-way set associative cache (left) and its corresponding cache model (right)

The cache model is a behavioral model that replicates the same characteristics of the on-chip cache (fig. 5.1), such as associativity, number of lines, line size, the writing policy and the replacement policy. Establishing a behavioral cache model that is a replica of the on-chip cache helps imitate the dynamic behavior of the real cache, which increases the estimation accuracy. Moreover, a behavioral model offers the possibility to explore different areas of optimizations and assist designers in making the most suitable architectural choices. As depicted in fig. 5.1, the cache model is simply a matrix of address tags and valid bits, where the rows represent the number of sets (4 sets) and the columns, in pairs of (valid, tag), represent the number of blocks (2 blocks) per set. Since the cache model is purely behavioral, data is not represented in the matrix.

For more flexibility, the cache model can be easily made reconfigurable. The cache setup can be tuned for the purpose of studying various cache configurations (e.g. a larger/smaller k-way associative cache may be more/less convenient) and whether they can be optimized to yield better performances. With such a model, the user can inspect the state of the cache at any point in time and test different configurations.

5.1.2 Inserting The Annotation Functions In The High-Level Code

The cache model described hereinbefore is triggered whenever a memory access is detected in the high-level code. To do so, annotation functions are inserted in the code for every

encountered access. With data caches, two operations may take place: a read from the cache and a write to the cache. These two operations have different impacts on the cache. So, they need to be modeled with two different annotation functions. When a memory access is located, a call to the appropriate annotation function, depending on the access type, is made and the cache model is activated in order to figure out whether the access is a hit or a miss.

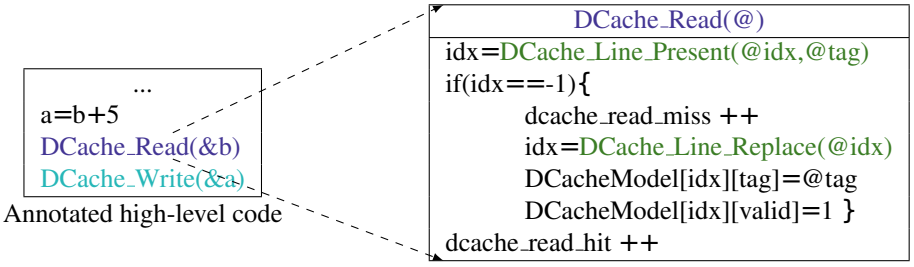


Figure 5.2: Data cache annotation functions

As shown in the example of fig. 5.2, two memory accesses are pinpointed in one instruction. As a result, the instruction is followed by two annotation functions. The first one simulates a read access of variable b. When the value of b is successfully retrieved, the addition operation is performed and the result is stored in variable a. This is illustrated by the second annnotation function, which simulates a write access. Once again, no actual write in the data cache model is conducted, which makes $DCache\_Write()$ and $DCache\_Read()$ functionally similar except for the access type ($dcache\_read\_miss/hit$ and $dcache\_write\_miss/hit$ may have different delays).

The annotation functions take as a parameter the address to which a memory access is made in order to determine whether the data at that memory location is present in the cache model. All the information needed to locate the data in the cache model is deduced from the address. The index is used to find out the cache set the data should reside in. For each block of that set, the tag of the block is compared to the address tag. If they are different, then the data is not in the cache. If they match, then the valid bit of the block where the data was found has to be checked. If it is 1 then the access is a cache hit. Otherwise, it is a miss. These checking steps are summarized by $Dcache\_Line\_Present()$ in fig. 5.2. In case of a miss, a replacement policy is applied to eject an old block from the cache and replace it with the new one. The most common algorithm is the LRU (Least Recently Used) replacement policy and it can be implemented in $Dcache\_Line\_Replace()$. The state of the $DCacheModel$ should always be maintained by updating the tag and $valid$ cells of the matrix when required.

5.1.3 Obtaining Memory addresses

The annotation functions and the data cache model used to accurately track the occurence of misses and hits rely on the data addresses as explained in the previous section. However, in the context of native simulation, two address spaces exist: the host address space and the target address space. The most obvious and simple solution is to directly use the native addresses (eg. $\&a$ and $\&b$ in fig. 5.2) [ZH09], [KKW+06], [GHP09].

In [ZH09] and [KKW+06], the addresses in the host memory space are used for data cache simulation because the authors conjectured similar spatial and temporal localities of the same program running on two different architectures (host and target). Similarly, authors of [GHP09] use directly data addresses from the native software during run time for their cache performance estimation. Their data cache simulation method slightly differs

from [ZH09]. The authors reckon that the number of memory accesses made by the same variable may be different from one architecture to another. So, they apply heuristics in order to associate to each variable the corresponding number of memory accesses in the target platform.

Since the software is natively compiled and executed, it is only aware of the host address space. Host and target address spaces may be different and the way variables are organized in memory may also differ. The layout of data in memory is controlled by the compiler and the operating system. The compiler is responsible for arranging the statically declared variables. The stack and static variables, for instance, can be placed in very different locations for the host and the target machines. As for memory accesses made to dynamically allocated variables, their locations are determined by the operating system. If the target OS is not simulated and the native OS is used instead, then a discrepancy in data locality between the host and the target is expected. Additionally, the two architectures may differ in data sizes (eg. a 64-bit Intel X86 processor for a host vs. a 32-bit Kalray processor for a target). Thus, many factors can contribute to the host architecture's having a different spatial locality from the target's. In that capacity, the outcome of the data cache model can diverge from the real cache, which may lead to substantial loss of simulation accuracy if native addresses are used without any modification.

```
int b[50];
sp=0x800; //sp initialization
fct(){
   int a;
   sp-=60;
   ...
   a=b[i];
   DCache_Read(0x2030+4*i);
   DCache_Write(sp+40);
   ...
   sp+=60;
      }
```

Figure 5.3: Annotating memory accesses to local and global variables

In order to realistically simulate memory accesses and the conflicts in referencing shared resources, as if they were performed by the target architecture itself, target addresses need to be reconstructed with information from the target binary code [KMGS13], [WH13], [DPE11], [PCG09].

In [DPE11], native addresses are transformed to target addresses, but the transformation is based on the assumption that the order of the variables in each memory section is maintained given the same compiler front-end and the same linkage order. But, the authors do not prove in any way the correctness of their assumption. Besides, in their transformation, they disregarded changing the base address because, according to the authors, cache performance depends mainly on spatial locality.

The approach, in [PCG09], is limited to global variables whose base addresses are obtained from the *symbol table* and back annotated in the user code for cache simulation.

[KMGS13] and [WH13] present two similar ways for accurately computing target addresses of the accessed memory in order to perform precise cache simulation in SLS. They simulate the target memory allocation mechanisms and thus they can handle global/static variables, dynamically allocated data and stack data. The idea is to identify memory ac-

cesses at the binary level, after compiler optimizations have been performed, and to determine their addresses. These addresses are then fed to the annotation functions, which are placed in the high-level code. The difficulty of this approach is twofold: first, to accurately place the annotation functions in the high-level code, the target binary-level memory accesses ($LOAD$ and $STORE$ operations) should be painstakingly mapped to their corresponding high-level code instructions, which is the responsibility of the mapping algorithm. Second, the addresses of binary-level memory accesses cannot be all resolved statically. In fact, these accesses are classified in two categories:

- Data accesses whose addresses can be known at compile time: this category encompasses static and global variables, which are allocated statically.
- Data accesses whose addresses can only be known at execution time: this category includes stack data and dynamically allocated variables.

Addresses that belong to the first category can be reconstructed statically using the *symbol table* in the *ELF* file provided by the debugger. Addresses in the second category are more complicated to reconstruct because they change dynamically depending on the execution context. These addresses are handled by simulating the heap allocation mechanism (for dynamically allocated variables) and the state of the stack pointer (for local variables) of the target processor in the natively-simulated code.

The example in fig. 5.3 illustrates the computation of the target addresses of a global variable (b) and a local variable (a) in a high-level code. The base address of b ($0x2030$) is extracted from the *symbol table*. As for the address of a, it is determined by tracing the value of the stack pointer. So, a variable sp is introduced in the code to track the dynamic behavior of the target stack pointer. The initial value of sp ($0x800$) is defined by the boot loader. At the entrance, respectively exit, of the function ($fct()$), sp is increased, respectively decreased, by the stack size (60) of $fct()$ taken from the target binary code. The address of the local variable a is obtained by adding its offset (40) to the stack pointer. The offset is computed by decoding load/store instructions from the target binary.

By exploiting the target memory allocation mechanisms, not only the number of misses and hits are computed, but also their timeline is faithfully re-created. However, for each memory access, address computations are performed adding a large simulation overhead.

5.2 Modeling Instruction Cache and Instruction Buffer for Performance Estimation of VLIW Architectures

Just like data caches, instruction caches are employed to meet stringent run-time and power consumption constraints. Given their impact on the non-functional aspects of the system, simulating the behavior of instruction caches has been the focus of many performance estimation approaches [LSA95], [WH13], [YMH+14].

Simulating an instruction cache has always been considered less challenging than simulating a data cache because instructions' addresses are known at compile time. Most of the existing native simulation techniques use generic tag-search based cache models. Similar to the data cache model, no instruction needs to be stored in the array as the instrcution cache model is only behavioral. In these classical cache models, an instruction is checked, during run time, to be a hit or a miss using its tag.

Reducing tag search [WH13] and minimizing the access frequency to the cache model [YMH+14] are the two axes of improvements that have been tackled when it comes to in-

struction cache modeling in native simulation. However, these generic cache models may work for simple architectures, but will definitely lead to erroneous estimations when used for architectures that exploit ILP, such as VLIW processors. ILP techniques have been increasingly used by processors' manufacturers to improve performance. The current breed of modern processors like Itanium by Intel, SHARC by Analog Devices, MPPA manycore by Kalray, ST200 series by STMicroelectronics, C674 by Texas instruments, TriMedia by NXP, etc., makes use of VLIW architectures to achieve high execution speed at low energy [Let09].

Although simulating micro-architectural components dynamically leads to accurate estimations and a clear view of the target architecture, it introduces a considerable simulation time overhead. This is why, existing approaches that simulate the instruction cache dynamically settle for classic architectures and do not take into account common performance-enhancing features, such as the ones used in VLIW processors.

In this section, we explain our contribution, which consists of demonstrating a software performance estimation approach by presenting a realistic instruction cache model that accurately reflects the impact of a component necessary to handle the instruction parallelism in a VLIW architecture, called the instruction buffer (IB). The behavior of the instruction cache and the IB is dynamically simulated using a native simulation platform while maintaining a reasonable simulation speed.

5.2.1 Overview and Particularities of a VLIW Architecture

In the 1980's, a new style of ILP technology called VLIW emerged as a natural outgrowth of horizontal microcode and have had a huge impact on the computer industry ever since [Let09]. A VLIW processor (a.k.a. static multiple-issue processor) executes n independent RISC instructions in parallel.

Bundles

The difference between a machine that supports ILP and one that doesn't is not the type of the execution units but their number. In a VLIW architecture, many execution units are made available to the program allowing the execution of several operations simultaneously. VLIW CPUs execute operations in parallel according to a dependence analysis between operations and a static schedule determined by the compiler (rather than the hardware), which avoids encumbering the hardware but requires a more complex software (i.e. advanced compiler). So, the role of the compiler is to find enough independent operations to keep the execution units busy. The number of operations issued simulatneously in a VLIW architecture is a function of the output of the dependence analysis performed by the compiler and of the available hardware functional units.

Branch op (BCU)	Integer op1 (ALU1)	Integer op2 (ALU2)	Integer op3 (MAU)	Floating point op (FPU)	Memory op1 (LSU1)	Memory op2 (LSU2)

Figure 5.4: An example of a VLIW instruction format

In a VLIW architecture, a bundle (also called execute packet or VLIW instruction [AFY05]) is a set of independent instructions that can be executed in parallel, each instruction being composed of syllables (32-bit words). Fig. 5.4 shows an example of a VLIW instruction format that might include seven operations: three integer operations, two memory operations, one floating point operation and one branch operation. A VLIW instruction also contains a control field for each of the execution units (ALU, BCU, etc.).

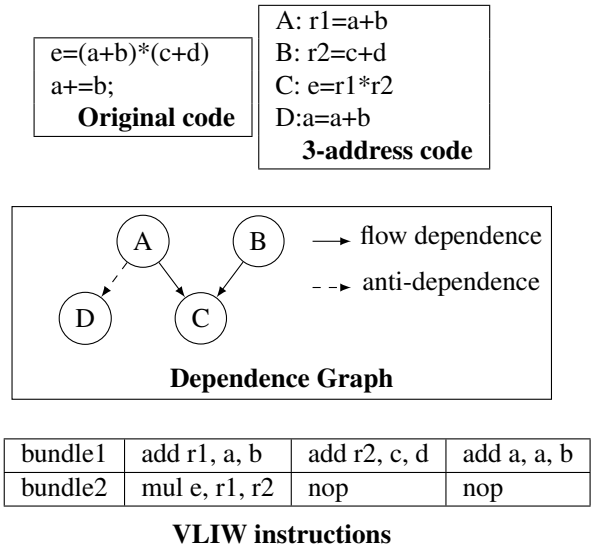


Figure 5.5: A simple example of bundle construction

A simple example of bundle construction is sketched in fig. 5.5. The original code is transformed into RISC-like operations (a 3-address code). Then, a precedence graph based on dataflow analysis (flow dependence, output dependence, anti-dependence, control dependence, etc.) conducted by the compiler is established to expose parallelism. Independent operations are packed into bundles and are executed simultaneously on the available execution units. In fig. 5.5, two bundles are formed. The first one is composed of three *add* operations scheduled for concurrent execution, which requires the availability of at least three *ALU*s. In the second bundle, there are no concurrent operations that can be scheduled with the *mul* operation. So, the compiler pads the unused slots with *nop*s (null operations). Null operations are dummy operations that do nothing but they do occupy the memory, which is undesirable. Some VLIW architectures avoid *nop*s by varying the size of the bundles [DdDAB+13].

Instruction Buffer

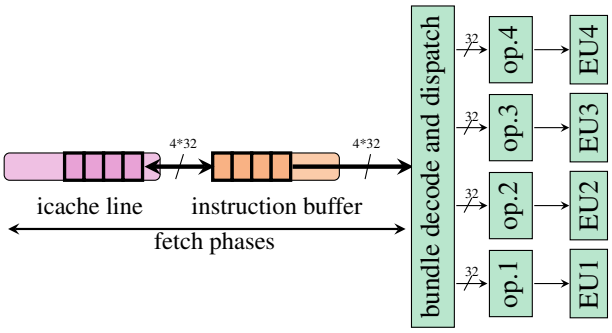
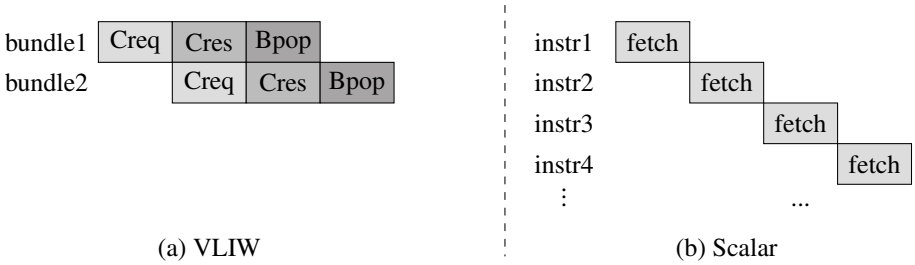


Figure 5.6: Overview of a VLIW architecture

During execution, a contiguous sequence of syllables (a.k.a. a fetch packet), is fetched from the instruction cache and placed in a buffer (referred to as instruction buffer-IB [AFY05], [Int10], [STM04] or prefetch buffer-PFB [DdDAB+13]), as shown in fig. 5.6. A fetch packet

is thus composed of either a subset of a bundle, a single bundle or multiple bundles. In the example of fig. 5.6, four 32-bit instructions (i.e. a fetch packet of 128 bits) are fetched from the instruction cache and pushed in the instruction buffer (IB). Then, during the decode and dispatch stage, the fetch packet is decoded, exposing the constituent bundle(s). The fetch packet, in the example of fig. 5.6, is composed of one bundle, that is, all four instructions of the fetch packet are dispatched to the four existing functional units and executed simultaneously.



(a) VLIW (b) Scalar

Figure 5.7: Comparison of the pipeline's fetch stage between VLIW and scalar

The instruction buffer is the intermediary between the instruction cache and the processor. It is responsible for requesting fetch packets from the cache and issuing bundles to the core. So, the IB does not have an impact on the behavior of the cache in terms of hits/misses, but it certainly impacts the performance in terms of CPI (cycle per instruction), as shown in fig. 5.7. Since we are specifically interested in studying the instruction cache, only the fetch phases of the pipeline are represented.

In the example of fig. 5.7-(a), the VLIW architecture features a three-phase fetch stage: cache request (*Creq*), cache response (*Cres*) and instruction buffer pop (*Bpop*). As for the pipelined scalar architecture (fig. 5.7-(b)), the fetch stage is composed of one phase. Assuming that each pipeline stage takes 1 cycle, bundles are composed of four instructions and that fetch phases are ideal (no stalls), bringing eight 32-bit instructions from the instruction cache to the pipeline takes 4 cycles in case of a VLIW architecture and 8 cycles in case of a scalar architecture. In case of scalar processors, computing the number of hits and misses, using the traditional cache models and annotating the software with their respective time delays, is a feasible approach and it has been abundantly adopted in literature. Yet, a naïve estimation of the number of cycles based solely on the number of cache hits and misses will lead to unreliable results for VLIW architectures because of the complex timing behavior of the fetch phase.

5.2.2 Generic Instruction Cache Modeling

The estimation of instruction cache performance has been intensively researched in literature using both static and simulation-based methods. In [SB08], [LLT10], [YMH+14] and [WH13], the instruction cache behavior is studied at execution time. These works share the same idea, which consists of dividing basic blocks of the high-level code into smaller blocks (called cache analysis blocks in [SB08]). Instructions inside a cache analysis block fit into the same cache line (i.e. these instructions have the same tag). A function at the end of each basic block is added. At run time, this function checks whether the tag of each cache analysis block is found in the instruction cache model or not, and it consequently updates the cache model (valid bit, LRU, tag) if necessary. In [WH13], the authors further enhance cache simulation by reducing tag search.

The proposed instruction cache simulation process

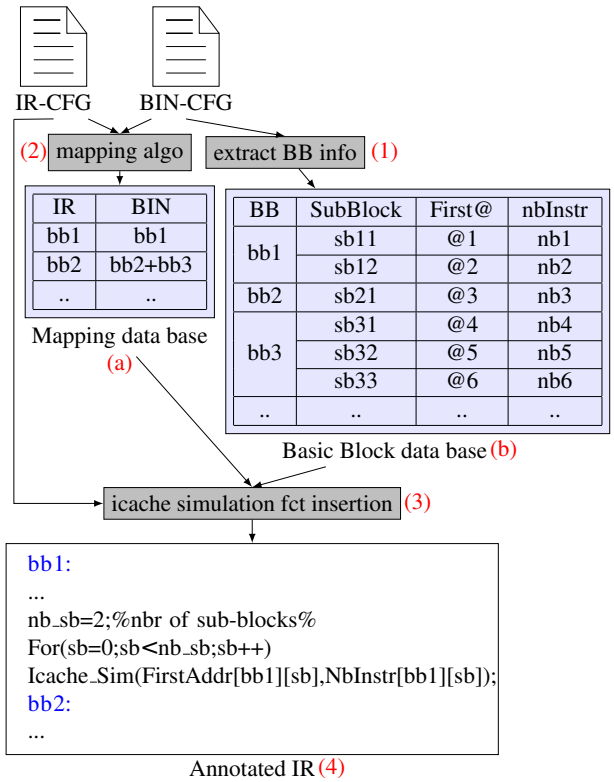


Figure 5.8: The instruction cache simulation process

In order to simulate the instruction cache behavior, a cache model identical to the real cache is used. The instructions' addresses are also required. Unlike data cache simulation, where data addresses are determined at run time, instruction addresses are known at compile time and are extracted from the target binary code (fig. 5.8-(1)).

To mitigate the simulation load, we exploit the accessibility of instruction addresses at compilation time by doing as many computations as possible statically i.e. before simulation. So, to reduce tag search, each basic block is divided into sub-blocks such that all the instructions of a sub-block fit in the same cache line. A basic block data base is created (fig. 5.8-(b)) to hold the statically collected information about each binary basic block: number of sub-blocks, first address and number of instructions of each sub-block.

Since our annotation approach is based on ILS, the annotations are inserted inside the basic blocks of the IR code. As a reminder, the IR and binary control flow graphs are not always isomorphic due to compiler optimizations. So, a mapping (chapter 4) between the binary CFG and the IR CFG is conducted (fig. 5.8-(2)) resulting in a mapping data base (fig. 5.8-(a)). For example, $bb2_{bin}$ and $bb3_{bin}$ both correspond to $bb2_{ir}$, which means that information (for e.g. $nbInstr$) about $bb2_{bin}$ and $bb3_{bin}$ should be inserted in $bb2_{ir}$.

The annotation function $Icache\_Sim()$ is then inserted in the IR basic blocks (fig. 5.8-(3)). Instead of calling the annotation function for each instruction of a basic block, it is only called as many times as the number of sub-blocks inside a basic block (fig. 5.8-(4)).

Algorithm 2 ICACHE_SIM($first\_addr$,nb_instr)

1: $present = Icache\_Line\_Present$(first_addr)
2: **if** $present == -1$ **then**
3: line=$Icache\_Line\_Replace$(first_addr)
4: $Icache\_Update$(line,first_addr)
5: $delay+ = miss\_penalty$
6: **end if**
7: $delay+ = icache\_access\_time * (nb\_instr)$

The Generic Instruction Cache Annotation Function

Algorithm 2 illustrates the instruction cache annotation function. $Icache\_Sim()$ takes as parameters the first address and the number of instructions of a given sub-block. A tag search in the cache model is performed for the first instruction only (line 1). If the first instruction of the sub-block is not in the instruction cache then the replacement algorithm is used (line 3) and the cache model is updated with information about the new line (line 4). Only the tag and the index are needed as no real instructions will be loaded in the cache. Following a cache miss, the delay is augmented with the miss penalty of the first instruction (line 5). Finally, the total instruction cache access time of the sub-block is tallied and added to the delay (line 7).

During simulation, when a basic block is executed, the underlying instruction cache annotation function is called and the instruction cache model is triggered. Even though annotation functions are inserted in all the mapped basic blocks in the IR, only basic blocks that are reached during simulation call their annotation functions. So, a different execution may lead to a different miss ratio, which is expected because the cache behavior tightly depends on the program context, hence the benefits of dynamically simulating the cache as opposed to the static approaches.

In a generic cache simulation approach (algorithm 2), the delay is computed as follows:

$$delay+ = nbr\_misses \times miss\_penalty + nbr\_accesses \times icache\_access\_time \quad (5.1)$$

This formula is based on a simple association of number of hits and misses to their respective delays. Each time a basic block is visited during execution, the delay is incremented with a newly computed value corresponding to the current basic block.

This approach, though efficient with scalar processors, is incapable of accurately reflecting the impact of VLIW architectures on the performance of instruction caches.

5.2.3 The Effect of VLIW on Instruction Cache Performance Estimation

Fig. 5.9 shows a comparison between the execution of two simple operations using a scalar processor and a rudimentary VLIW processor with two load/store units, one multiply and one add unit. The symbol "//" portrays parallelism between instructions.

As can be noticed in the example of fig. 5.9, the VLIW processor executes up to three instructions simultaneously. These concurrently executed instructions are placed in a bundle. In our example, instructions are grouped in 4 bundles. Assuming that each instruction can be completed in one unit of time, then the VLIW processor takes 4 units, which is half the time taken by the scalar processor.

To accommodate ILP, the VLIW architecture offers multiple independent functional units. The processor fetches a bundle from the instruction cache and dispatches the instructions

Operations	Scalar processor	VLIW processor with 2 load/store units 1 add unit,1 mul unit
op1: m=a+b op2:n=c*d	load a r1	load a r1 // load b r2
	load b r2	load c r3 // load d r4 // add r1 r2 r5
	add r1 r2 r5	mul r3 r4 r6 // store r5 m
	store r5 m	store r6 n
	load c r3	
	load d r4	
	mul r3 r4 r6	
	store r6 n	

Figure 5.9: Scalar vs. VLIW

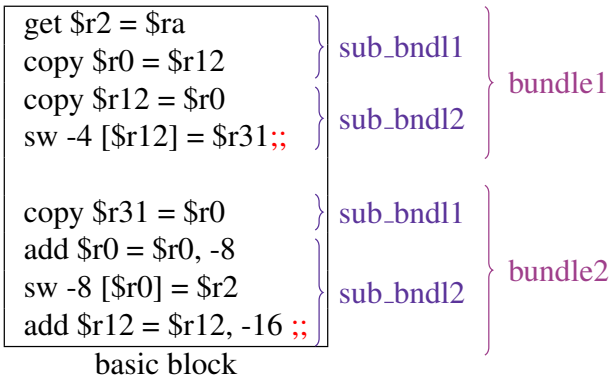


Figure 5.10: Bundles in assembly language

for parallel execution on the different functional units. To keep all of these units busy, the compiler's goal is to gather as many instructions as the number of units in a bundle.

As depicted in fig. 5.10, bundles can be discerned easily at compile time. The basic block shown in fig. 5.10 contains two bundles separated by ";;". Each bundle is composed of four 32-bit instructions.

BB	bundle	sub-bundle	first@
bb1	bndl11	sub-bndl111	@1
	bndl12	sub-bndl121	@2
		sub-bndl122	@3
	bndl13	sub-bndl131	@4
..	..	..	..

Figure 5.11: Basic block data base

At the basic block information extraction step, instead of dividing basic blocks into sub-blocks, like we did in the generic cache simulation method (fig. 5.8-(b)), we divide basic blocks into bundles. Each bundle is then divided into sub-bundles (fig. 5.11). A sub-bundle has the same definition as a sub-block. The cache annotation function is called for each

sub-bundle. If the first instruction of the sub-bundle is in the cache, then

$$delay+ = icache\_access\_time,$$

otherwise,

$$delay+ = miss\_penalty + icache\_access\_time.$$

Since instructions in a sub-bundle are executed synchronously, only the first instruction is considered in the delay computation, as opposed to eq. (5.1). So, there is no need to record the number of instructions of sub-bundles in the basic block data base (fig. 5.11).

5.2.4 Instruction Buffer Impact on Instruction Cache Performance Estimation

At this level, we handled parallelism inside bundles. However, we considered bundles to be executed sequentially. In many architectures [STM04, DdDAB+13, Int10]), the instruction

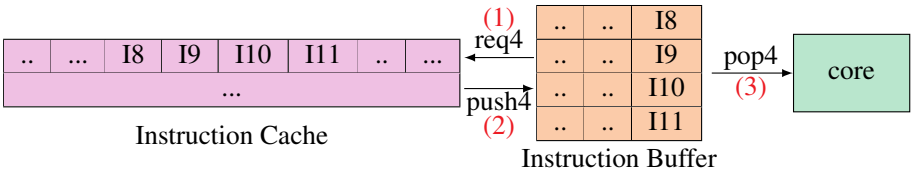


Figure 5.12: Instruction Buffer

cache is further enhanced with an instruction buffer (fig. 5.12). An instruction buffer is in charge of fetching a sequence of instructions (*i.e.* a fetch packet) from the instruction cache and providing it to the core. The interesting aspect of an IB is the ability to fetch instructions ahead of time (before the core even requests them), which offers a sort of parallelism among fetch packets.

In the example of fig. 5.12, the IB is composed of four FIFOs with three stages each. The IB can hold four 32-bit instructions in each one of its three stages. Each clock cycle, the IB requests four instructions from the instruction cache (fig. 5.12-(1)). The cache pushes the four requested instructions into the instruction buffer's FIFOs (fig. 5.12-(2)), which are then popped into the next stages of the pipeline (fig. 5.12-(3)) to be decoded and executed as bundles. In the remaining of this section, the IB example (fig. 5.12) will be used to explain the IB behavior and its influence on the execution time. For clarity reasons and without loss of generality, we assume that a fetch packet is composed of one bundle. Thus, the terms *bundle* and *fetch packet* will be used interchangeably.

Breaking up the execution of bundles into sub-steps enables the bundles to overlap (to be executed partially at the same time) and offers better performances. As can be depicted in fig. 5.13, the sub-steps are: *CacheRequest*, *CacheRespond*, and *IBPop*, which respectively correspond to (1), (2) and (3) in fig. 5.12. These sub-steps constitute the 3 phases of the pipeline's fetch stage.

Given the VLIW architecture and with the instruction buffer in the picture, the parallelism is both intra- and inter-bundles. So, the delay computation is more complicated than explained in the previous section as several cases arise.

Nominal Case

In the nominal case (fig. 5.13), the IB requests 4 instructions from the cache in the first cycle. In the following cycle, assuming that the instructions hit in the cache and they are

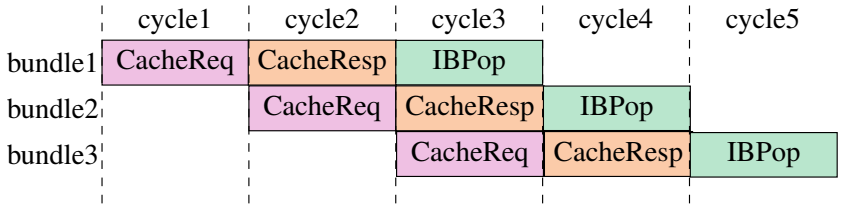


Figure 5.13: Nominal Case

in the same cache line, the cache pushes the requested instructions in the buffer and the buffer initiates a new request during the same cycle. In the third cycle, the core requests a bundle of four 32-bit instructions, which is already in the IB. So, the bundle is popped into the pipeline. During the third cycle, three bundles are handled simultaneously: the first bundle is already in the pipeline, the second bundle is in the IB and the third bundle is fetched from the cache. Starting from the fourth cycle, a steady state is reached where a bundle is executed per cycle.

To be able to reflect the overlapping of bundles in the delay computation, the delay formula of the nominal case (i.e. cache hit, the requested instructions reside in the same cache line and the core requests a 4-instruction bundle) is:

$$delay_{nom} = k + nbr\_bundles - 1, \tag{5.2}$$

where k is the number of phases of the fetch stage (3 in our case), and $nbr\_bundles$ is the number of the executed bundles. For example, the delay of the 3 bundles (fig. 5.13) is 5 cycles. In order to keep track of the number of executed bundles, a bundle counter is introduced in the IR code. During simulation, each time a basic block is visited, the counter is incremented with the number of bundles of the visited basic block.

Branch Case

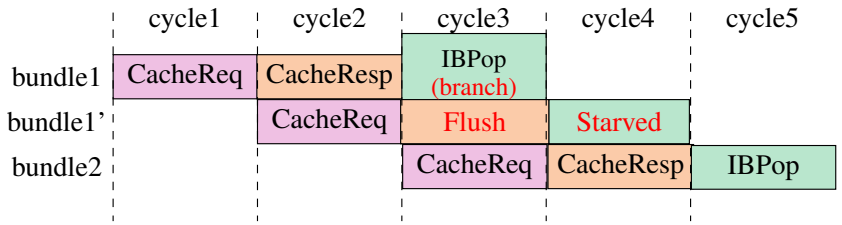


Figure 5.14: Branch Case

When a branch is executed (fig. 5.14), the buffer has to flush its FIFOs because it has already filled them with sequentially fetched instructions that may not be valid anymore. In cycle 3, bundle1 (the bundle that comprises the branch instruction) is popped into the pipeline to be decoded and dispatched to the different execution units. When a branch is executed, the IB flushes its FIFOs and sends a request to the instruction cache in order to retrieve four instructions starting from the address of the branch target. During the fourth cycle, the core attempts to pop a bundle from the buffer, but fails as the buffer is still empty. So, the pipeline is starved for one cycle. In the fifth cycle, the core can finally retrieve a bundle since it is available in the IB.

As a consequence, a bundle that is a target of a branch (a branch can be a function call) takes 1 extra cycle to be brought to the pipeline, under the same conditions as the nominal

case. Accordingly, the starting bundle of each basic block, except the entry basic block, is a target of a branch (by definition of a basic block) and as a result takes 1 extra cycle:

$$delay_{branch} = nbr\_basic\_blocks - 1 \tag{5.3}$$

A Line-Crossing Bundle Case

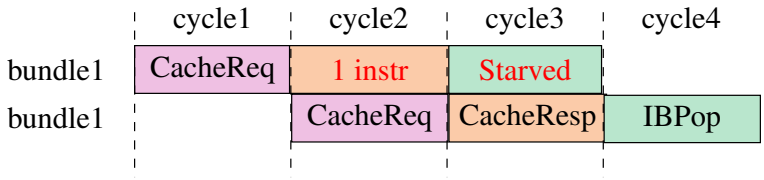


Figure 5.15: A line-crossing bundle case

In case of a line-crossing bundle (fig. 5.15), the cache responds with less instructions than requested by the IB because the bundle crosses the cache line boundary. Thus, more than one cache access is needed. In the example of fig. 5.15, only one instruction of bundle1 is pushed into the buffer in the second cycle. Consequently, the pipeline is starved during the third cycle since it wants a bundle of four instructions and only one instruction is available in the buffer. The three remaining instructions are requested by the buffer in the second cycle, pushed in its FIFOs in the third cycle and popped into the pipeline in the fourth cycle. So, each sub-bundle introduces an extra cycle:

$$delay_{cross} = \sum_{i=1}^{nbr\_bundles} nbr\_sub\_bundles_i - 1 \tag{5.4}$$

For each bundle, the number of added cycles is the number of its sub-bundles minus one. We subtract 1 for each visited bundle because its first sub-bundle is already included in eq. (5.2). Assuming that a cache line can hold more than four instructions, a bundle can span two cache lines at most. So, it can be divided into two sub-bundles, each one resides in a different cache line. Thus, the added delay of a line-crossing bundle is 1 cycle (in case both sub-bundles hit in the cache).

A Miss Case

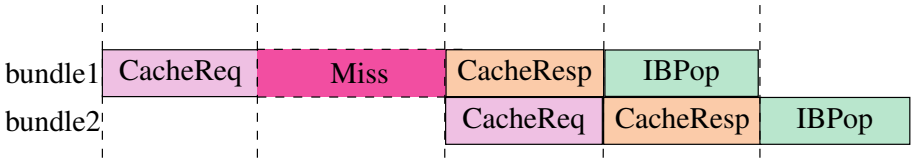


Figure 5.16: A miss case (a blocking cache)

In case a cache miss occurs (fig. 5.16) (in a blocking cache), the cache cannot respond to the ensuing requests issued by the IB until the missing bundle is brought to the cache. So, subsequent requests are delayed by the miss penalty of the missing bundle.

$$delay_{miss} = miss\_cycles \times (nbr\_misses) \tag{5.5}$$

The number of misses is computed during simulation using the cache model and the cache annotation approach explained hereinbefore. The number of misses is recorded in a counter and updated whenever a basic block is visited.

The overall delay (in cycles) caused by the instruction cache and the IB is the sum of equations (5.2), (5.3), (5.4) and (5.5):

$$delay = delay_{nom} + delay_{branch} + delay_{cross} + delay_{miss} \tag{5.6}$$

5.2.5 Limitations: Variable-Sized Bundles

In the different cases discussed in the previous subsection 5.2.4, the core always requests a 4-syllable bundle. However, in reality, the core can pop variable-sized bundles.

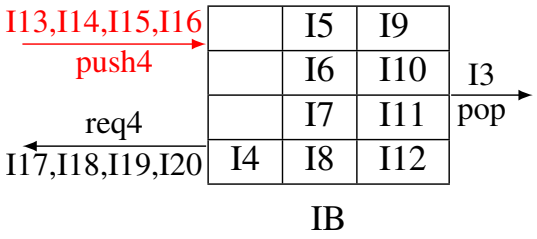


Figure 5.17: Problematic situation: full IB

The instruction buffer should be able to hold instructions sent from the instruction cache at any point in time. The case where instructions are ready to be pushed in the IB but the IB can't receive them because it has no room, should not occur (fig. 5.17). When a 4-syllable bundle is popped each cycle from the instruction buffer, the IB will never face the problem delineated in fig. 5.17. However, when the core decides to pop smaller bundles, the IB may reach a state where it is full and cannot store all the syllables that it requested from the instruction cache, during the previous cycle, in its FIFOs.

To solve this problematic situation, the instruction buffer initiates a fetch request only if it is sure it will be able to hold the requested instructions in its FIFOs:

$$\forall i \in [0,3], \sum_{k=0}^{2} nbFIFO_i[k] + pending_i < k, \tag{5.7}$$

where i designates a specific FIFO (among the four FIFOs in our example), k designates a specific stage (among the three stages in our example) of a given FIFO, $nbFIFO_i$ is the number of syllables (0 or 1) stored in FIFO number i and FIFO stage number k, and $pending_i$ corresponds to the number of syllables (0 or 1) intended for FIFO number i. If this condition is fulfilled, then the IB will always have room for the previously requested instructions, but this solution may lead to the imbalance of the FIFOs, which may cause the pipeline to starve.

At cycle 1, in the example of fig. 5.18, four syllables are pushed in the IB, four syllables are requested from the cache, and two syllables are requested by the core from the IB. At cycle 2, a 2-syllable bundle is popped from the IB. The IB does not perform a fetch request because:

$$nbFIFO_{2,3}[0] + nbFIFO_{2,3}[1] + nbFIFO_{2,3}[2] + pending_{2,3} = 1 + 1 + 0 + 1 = 3.$$

Since the IB did not request any syllable from the instruction cache during the previous cycle, nothing is pushed in the IB at cycle 3. At cycle 3, the core requests a 4-syllable bundle

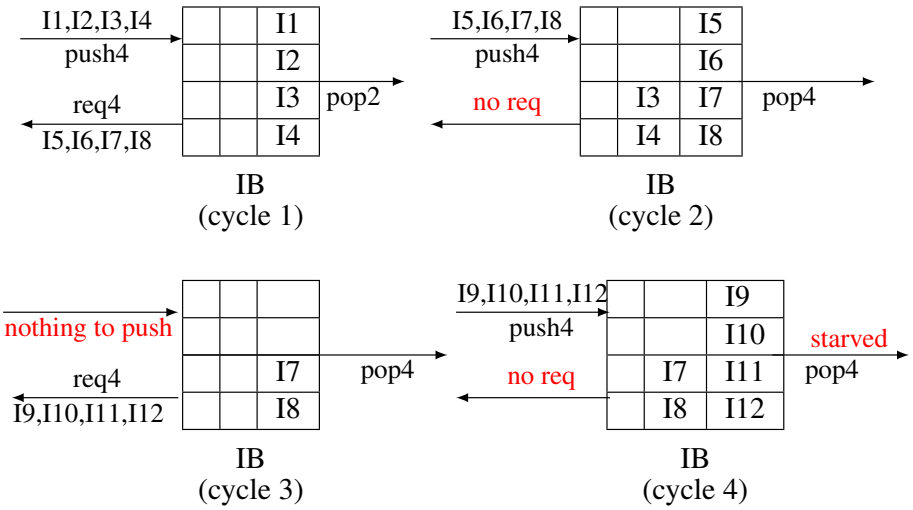


Figure 5.18: Unbalanced FIFOs of the IB

from the IB, but only two syllables are available. The IB launchs a fetch request, as its state fulfills condition (5.7). At cycle 4, nothing is removed from the IB and the pipeline is starved. The pipeline requests a 4-syllable bundle from the IB again and this time the IB has the requested bundle. This bundle is popped at cycle 5. The state of the IB at cycle 4 is the same as in cycle 2. In fact, starting from cycle 4, cycles 2 and 3 will be repeated as long as a 4-syllable bundle is popped each time. The imbalance of FIFOs caused by a smaller bundle (less than 4 syllables) causes the pipeline to starve periodically.

We did not take into account the impact of the irregular bundle size on the instruction cache and instruction buffer performance because different successions of bundles can lead to different outcomes. For instance, if the core in the example of fig. 5.18 requested a 2-syllable bundle at cycle 2, the starvation problem wouldn't have arisen. Alternatively, if the pipeline stalls for some reason (cache miss, data dependencies, etc.) the IB would have more time to bring more instructions to its FIFOs and the problem described above would not occur.

Since the disadvantageous pattern that we described in fig. 5.18 (i.e. the core keeps requesting 4-syllable bundles after it had requested a 3-syllable bundle), is usually broken by requesting a (4-n)-syllable bundle or a (8-n)-syllable bundle, where n is 2 in our example, the approximation that we made does not have a consequential imapct on the accuracy of the estimates. In the worst case scenario, where a bundle mean size is 4 syllables and a single smaller bundle (1, 2 or 3 syllable-bundle) perturbs the balance of the IB, the performance of the code is downgraded by 33% compared to the optimal case (i.e. a series of *exactly* 4-syllable bundles), according to Kalray.

5.3 Conclusion

Micro-architectural components have a consequential impact on the performance of the system. Thus, their effects should be accounted for, in performance estimation. Pipeline and cache behavior have been given a lot of attention in performance estimation approaches. Yet, the ever evolving architectures of MPSoCs are characterized by the incorporation of highly advanced hardware components with superscalar or VLIW features in order to cater for the continuous quest for better computing capabilities. To yield accurate performance

estimates, the distinguishing features of such complex architectures should be accurately modeled.

In this chapter, we presented a generic data cache model used for performance estimation. Then, we gave an overview of a VLIW architecture and we underlined the limitation of a generic instruction cache model in reflecting the behavior of such architecture.

So, we presented an approach to estimate the performance of an instruction cache in a VLIW architecture with an *instruction buffer*. In the instruction cache simulation, bundles were considered instead of instructions. Several cases related to parallelism among bundles were explained and their effects on the delay computation were formalized in separate formulas.

We only addressed fixed-size bundles, whereas in reality, a processor can request variable-sized bundles, which may lead to unbalanced FIFOs of the instruction buffer and consequently additional delay cycles. The irregular aspect of such bundles makes the computation of the delay very challenging as many elements should be factored in the computation.

Chapter 6

Experimental Results

To validate the accuracy of the proposed mapping approach, as well as the instruction cache and instruction buffer performance models, we present and analyze a set of experimental results using two simulation environments: a native simulation platform and an instruction set simulator used as a touchstone. We evaluated the proposed mapping approach and the performance models quantitatively in terms of simulation speed and accuracy. The accuracy of the mapping approach is measured using the instruction count as a metric. As for the precision of the instruction cache and instruction buffer, it is assessed using miss count and cycle count. These performance estimates obtained by native simulation are compared to the ones generated by ISS.

Before we tackle the experimental results of our contributions in sections 6.3 and 6.4, we will start by dissecting the quintessential constituents of the experimental environment.

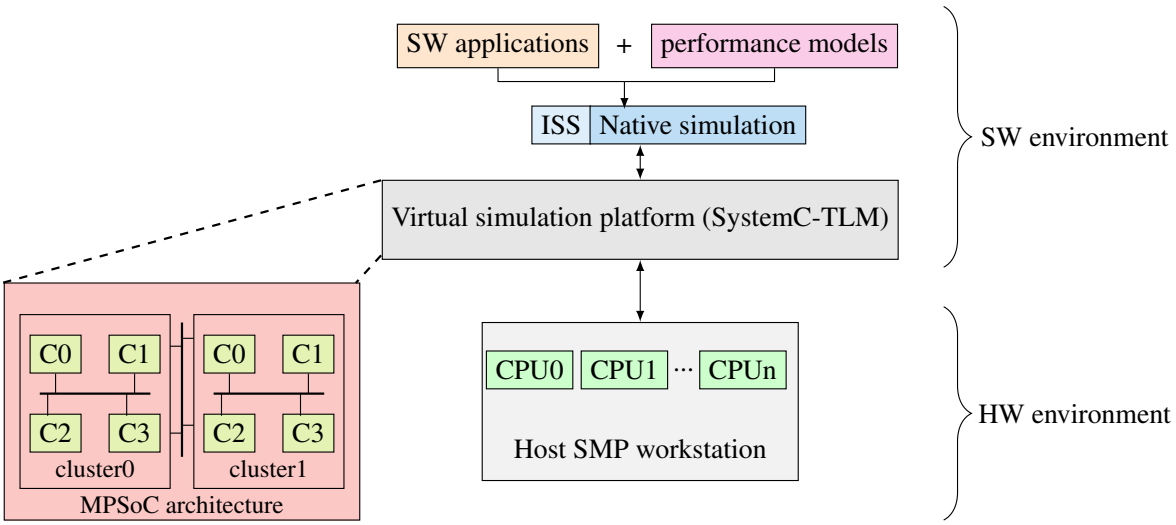


Figure 6.1: A bird's eye view of the experimental environment

As showcased in fig. 6.1, the experimental environment can be roughly divided into two parts:

- a hardware part that includes the host machine, where the simulation is conducted, and the simulated target MPSoC,
- a software part that incorporates the software applications (benchmarks), which are

coupled with the performance models, and the simulation environment, i.e. the software simulation platform (ISS/Native) and the hardware simulation platform (SystemC/TLM).

The hardware environment is decribed in section 6.1 and the software environment is detailed in section 6.2.

6.1 HW Environment

The target platform that we simulate is the Kalray MPPA-256 SoC. The platform on which the simulation of the target SoC is conducted, a.k.a. host machine, is a *x86* desktop.

6.1.1 Target Architecture: Kalray MPPA-256

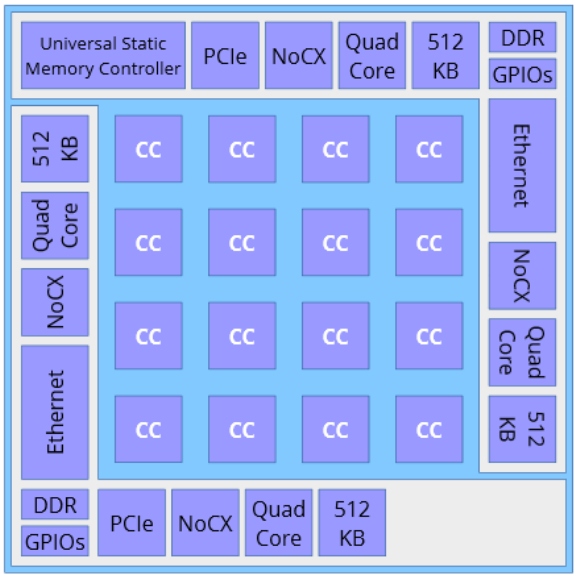


Figure 6.2: An overview of the Kalray-MPPA

MPPA (fig. 6.2) is a family of programmable and massively parallel processor array (MPPA) intended for real-time, low power, compute-intensive applications, such as autonomous vehicles software and storage in data centers.

A clustered architecture

The kalray MPPA-256 architecture integrates 256 user cores and 32 system cores. These cores are organized in 16 compute clusters and 4 quad-core I/O subsystems located at the periphery of the chip (fig. 6.2). This array of compute clusters and I/O subsystems are connected to a toroidal 2D NoC. Each compute cluster (fig. 6.3), which is the basic processing unit of the MPPA architecture, is multicore and is composed of 16 processing elements (PEs) dedicated to application processing and one core referred to as the resource manager (RM), characterized by a connection to NoC interfaces and in charge of controlling the cluster (firmware uploads, system code execution, security functions, etc.).

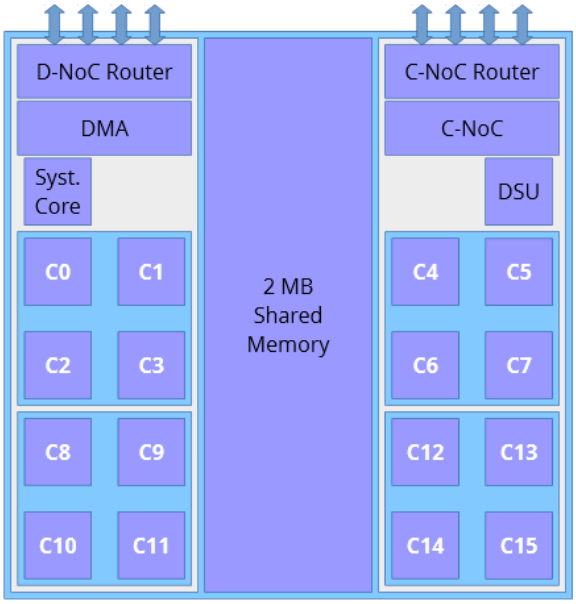


Figure 6.3: An internal view of a cluster from the kalray-MPPA

A VLIW core

Each core (RM or PE) implements a 5-issue VLIW architecture with a 7-stage instruction pipeline and five execution units: two arithmetic and logic units, a multiply-accumulate/floating-point unit, a load/store unit, and a branch and control unit.

Memory hierarchy

A compute cluster is equipped with a 2MB shared memory organized in 16 parallel banks of 128KB each. Kalray-1 VLIW core implements separate instruction and data caches with no hardware cache coherence. When needed, cache coherence has to be maintained by software. Each VLIW core (RM or PE) has its own independent 8K-bytes, 64-byte lines, 2-way set associative instruction cache and 8K-bytes, 32-byte lines 2-way set associative data cache. A prefetch buffer (PFB), i.e. instruction buffer, in charge of issuing instruction bundles to the core is also a part of a the Kalray-1 VLIW architecture. The PFB is composed of four 3-stage FIFOs. Each stage of the PFB can hold up to four 32-bit instructions.

Interconnect

The 16 compute clusters and the 4 I/O subsystems are connected to a NoC with bi-directional links, characterized by its 2D-wrapped-around torus topology and wormhole switching providing a full duplex bandwidth up to 3.2 GB/s between two adjacent clusters. In fact, The MPPA NoC comprises two parallel networks: a data NoC (D-NoC) optimized for bulk data transfer and a control NoC (C-NoC) optimized for small messages at low latency. Each compute cluster is linked to one NoC node. Each I/O subsystem, on the other hand, is associated with four NoC nodes. A NoC node consists of a D-NoC router and a C-NoC router. Both NoCs ensure reliable delivery and messages that take the same route arrive in order.

6.1.2 Host Machine

Table 6.1: Host CPU information

Model name	Intel(R) Xeon(R) CPU	
Architecture	x86-64	
Frequency	3.47GHZ	
Nbr of cores	6	
caches	L1 dcache:	32KB
	L1 icache:	32KB
	L2 cache:	256KB
	L3 cache:	12288KB

The host processor, on which the native simulation platform is executed, is an Intel x86, 64-bit, SMP 6-core that runs at 3.47GHz (table 6.1). During our experiments, only one core is used.

6.2 SW Environment

In the software environment, we will focus on the simulation platforms: the native simulator, which is the testbed for all the experiments we conducted, and a cycle accurate ISS, which is used as a guideline to measure the accuracy of our contributions. We will also give a rundown of the chosen benchmarks.

6.2.1 Simulation Platforms

Native Simulation

The native simulation platform that we adopted is the one proposed in [Sar16], which relies on the integration of KVM into an event-driven simulation environment (SystemC) to solve the address space problem.

The software is compiled to the host machine format. At the start of the simulation, a memory region is allocated in user space and passed to KVM using a dedicated system call (`ioctl(KVM_SET_USER_MEMORY_REGION)`). The host instructions are loaded in that memory zone. Simulating a processor consists of executing the VM associated to it using system call `ioctl(KVM_RUN)`. This VM reads the host instructions and executes them natively in guest mode. One of the reasons for which the guest mode is exited is when a synchronization with the virtual simulation platform (SystemC) in user mode is required. In other words, when a time annotation is encountered while executing the software by the VM, the VM halts and the SystemC platform, more precisely a SystemC module called *native processing unit* (NPU) (fig. 6.4), takes over and advances the execution time by the value indicated in the annotation. A switch from guest mode to user mode happens in two steps: a switch from guest mode to kernel mode and a switch from kernel mode to user mode.

To bridge the gap between the kernel driver and the virtual platform in user mode the NPU provides the interface between hardware SystemC components and KVM kernel driver using the KVM user-space library (fig. 6.4). Among the services requested by the native processor from the KVM driver are: the creation of a new virtual machine

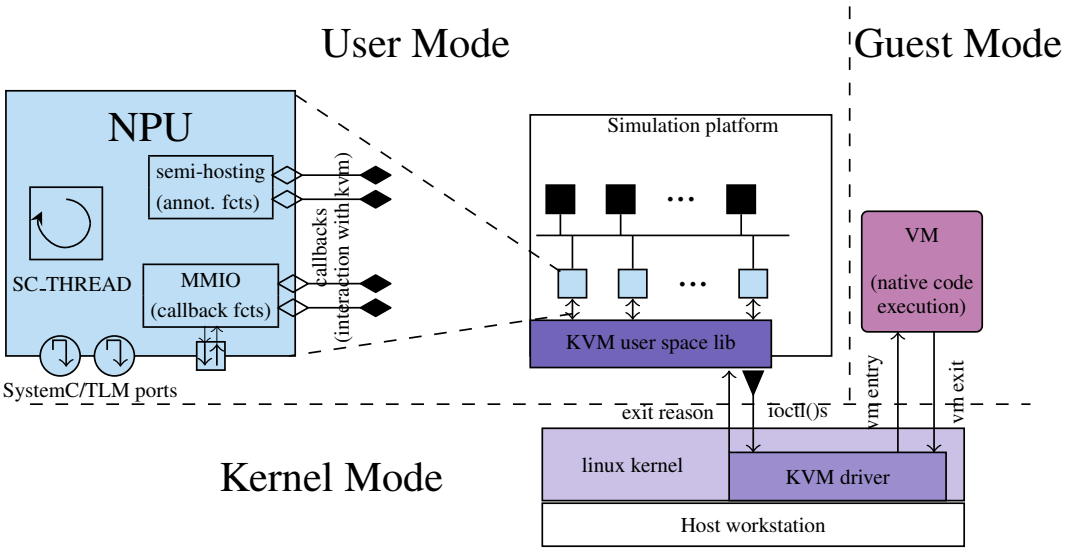


Figure 6.4: Interaction between the NPU and the VM

(KVM_CREATE_VM) with one or more virtual CPUs (KVM_CREATE_VCPU), the creation or modification of a guest memory slot in user space (KVM_SET_USER_MEMORY_REGION), the launch of a VM execution (KVM_RUN). These requests are sent to KVM using the `ioctl()` mechanism. The native processor also provides SystemC interfaces, such as TLM ports (fig. 6.4).

Listing 6.1: Time annotation request

```
1 bb30:
2 ...
3 exec_time += 60;
4 cpu_kvm_io_callback(ANNOTATION_BASEPORT, &exec_time);
5 return D5731;
```

To transmit a time annotation request from the application running in guest mode to the SystemC platform, a *trap* mechanism based on PMIO (port mapped I/O) access is used, as illustrated in listing 6.1. The function that triggers the mode switch (`cpu_kvm_io_callback()`) takes two parameters: the first one is the annotation port, which is defined in KVM (e.g. `#define ANNOTATION_BASEPORT 0x4000`) and the second one is the address of the variable containing the time annotation.

Listing 6.2: Guest to host address transformation

```
1 uintptr_t host_vaddr = (uintptr_t)guest_flat_to_host(kvm_cpu->kvm,
    guest_addr);
2 systemc_annotate_function(sc_kvm_wrapper, kvm_cpu->cpu_id, *((int
    *)host_vaddr));
3 ...
```

This address in guest memory is transformed by a KVM callback function to a virtual address in host memory suitable for the SystemC platform (listing 6.2).

Listing 6.3: Wait function call

```
1 void systemc_annotate_function(void* sc_kvm_wrapper, int cpu_id,
    int exec_time)
```

```
2 {wait(exec_time, SC_NS);}
```

The NPU retrieves the value of the variable `exec_time` and calls the SystemC `wait()` function (listing 6.3).

However, a switch from guest mode to user mode is very costly (almost 10000 cycles to handle the mode switch i.e. go to kernel mode, determine the exit reason, switch to user mode, execute the wait function and return to guest mode). To minimize these mode switches, annotations should be accumulated and the *trap* function should only be called in exit basic blocks (i.e. basic blocks that contain an instruction that could possibly be the last executed instruction of the program, e.g. a basic block containing a *return* statement).

ISS

The kalray-1 toolchain comes with a `k1-cluster` simulation platform that instantiates multiple instances of Kalray-1 processor ISS and peripherals to model a MPPA cluster. The *k1* is an abstract model of the actual processor and it offers two execution modes: a functional simulation mode (`k1-cluster --prog`), with maximum simulation speed at the expense of simulation accuracy, and a *cycle-based* simulation mode (`k1-cluster --cycle-based --prog`), with maximum simulation accuracy and slower simulation time. In this cycle-accurate mode, a simplified processor pipeline is used and additional execution cycles due to pipeline stalls, instruction prefetch mechanism and cache memory accesses are taken into account. In all of our experiments, the *cycle-based* mode is used, even if we do not explicitly mention it in the examples.

The *k1* simulator also provides profiling support, which is extremely useful because it produces detailed profiling files containing valuable information about instruction and cycle counts and cache miss/hit counts. This information is key in evaluating the efficiency of the proposed mapping algorithm and the instruction cache performance model. Profiling is enabled with the `--profile` option. The generated profiling files can be analyzed and converted by external profiling tools in order to produce user-friendly output files in the desired formats. The currently available output files are *disassembly execution traces* generated by the (`k1-disasm`) tool and `callgrind` files generated by the (`k1-callgrind`) tool. In our work, we use the well-known profiling tool `cachegrind` from the `valgrind` tool suite to generate profiling information and we visualize it by the `kcachegrind` tool.

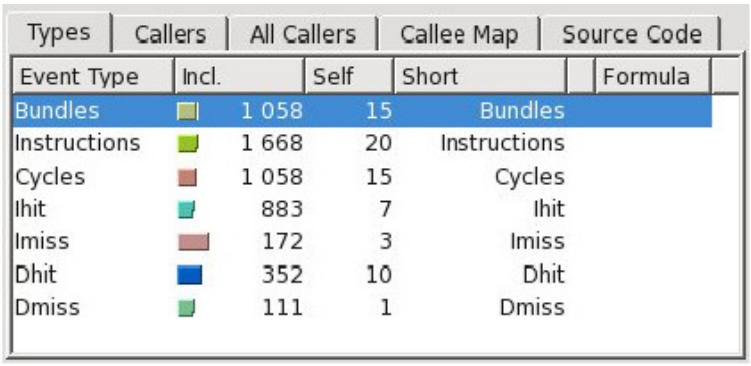


Figure 6.5: A screen shot of the event window of `kcachegrind`

Figure 6.5 gives a view of the `kcachegrind` event window and the cost types (executed bundles, executed instructions, executed cycles, instruction/data cache hit/miss count) supported by the *k1* simulator.

Table 6.2: Benchmarks

Benchmark	Description
Polybench	
covar	Covariance Computation
atax	Matrix Transpose and Vector Multiplication
reg-detect	2-D Image processing
trmm	Triangular matrix-multiply
gemver	Vector Multiplication and Matrix Addition
trisolv	Triangular solver
jacobi-2d	2-D Jacobi stencil computation
syr2k	Symmetric rank-2k operations
3mm	3 Matrix Multiplications ($E = A.B; F = C.D; G = E.F$)
lu	LU decomposition
Splash2	
FFT	complex one-dimensional FFT
radix	Integer Radix sort
Lu	Matrix triangulation
other	
matmult	1 Matrix Multiplication
bubbleSort	Bubble Sort
blowfish	Symmetric-key block cipher
crc	Cyclic redundancy check computation
fft1	Fast Fourier Transform using the Cooley-Tukey algorithm.

6.2.2 Benchmarks

Most of the applications used in the experimentation are taken from benchmark Polybench [Pou] because they are characterized by a significant number of (multi-level) loops, which will help test our loop-based mapping algorithm especially that loop-intensive code is prone to radical CFG transformations by the compiler, as explained before. Three applications (*FFT*, *Lu*, *radix*) from the mature benchmark suite Splash2 [WOT+95] are also selected because they feature a convoluted control flow (an important number of `if-else` and `switch-case` statements) and they are considered relatively big applications in terms of number of instructions and memory accesses. Other applications like *bubbleSort*, *blowfish*, *crc*, *fft1*, *matmult* are chosen because they are highly used by similar approaches in literature [WH12], [SBR11b], [SBR11a]. The different applications used in the experiments are presented in Table 6.2 with brief descriptions.

6.3 Validation of the Mapping Approach

In this section, we aim at validating the proposed mapping approach, which consists of finding correspondences between the target binary basic blocks and the IR basic blocks in order to accurately insert target-specific performance metrics into the functional model.

We will start first by describing the experimental setup and results of the mapping algorithm applied on the selected benchmarks compiled with `gcc -O2`, in subsection 6.3.1.

Then, in subsection 6.3.2, we will show the efficiency of the mapping approach in matching highly-optimized binaries (compiled with `gcc -O3`) to the IRs, at a basic block level, provided that certain transformations (as described in chapter 4, section 4.3.2) are conducted on the IR CFG.

6.3.1 Basic Mapping Scheme

Experimental Setup

Fig. 6.6 illustrates the different steps of the experimental process leading up to the obtained results. We start off by a source code. We took an excerpt of the *C* code of a *Matrix Multiplication (matmult)* application as an example, in fig. 6.6. For space reasons, only snippets of the code at its different compilation and instrumentation stages are displayed.

The source code is cross-compiled (`k1-gcc -O2 -fdump-tree-optimized`) leading to the generation of a target binary code (on the right) and low-level Gimple (the IR on the left). *Cross-compilation* is the process of generating binaries for a processor (Kalray *k1* processor in our case) other than the host processor on which we run the compilation (*x86* in our case). The *Cross-compiler* used in our work (`k1-gcc`) is part of Kalray's design kit. Although we employ gcc-based (cross-) compilers, our approach can be applicable to other compilers that use intermediate representations.

The IR code is transformed into a compilable IR (a.k.a. optimized C code). As for the target binary, it is executed using a cycle-accurate instruction set simulator (`k1-cluster`, also provided by Kalray). The results obtained with the ISS are very close to the ones obtained with the real platform. For this reason, the ISS is used as a reference to which we compare the results generated by native simulation.

The metric we retain to evaluate the accuracy of our mapping approach is the number of executed instructions: indeed, it depends only on the control flow path followed during execution and not on the accuracy of the SystemC models within the system as would timing do. So, the accuracy of the instruction count depends uniquely on the mapping algorithm between the IR and the binary, unlike cycle count for example, which depends both on the mapping and the time analysis strategy. Thus, by simulating the application on the ISS, while enabling the profiling option (`--profile`), we are able to generate the number of executed instructions. We also measure the simulation time of the original code executed on the ISS to which we will compare the simulation time of the original and optimized code (IR) executed on the native simulation platform.

The number of executed instructions of the host-compiled and natively executed IR is not representative of the number of executed instructions of the target code when it is run on the real platform, simply because the host and target machines have different ISAs. So, the target binary and the binary resulting from compiling the IR on the host machine will most likely have a different number of instructions. In order to be able to generate accurate estimates of the executed instructions using native simulation, we extract the number of instructions of each basic block of the binary CFG and store them in a data base (e.g. $nb\_instr(bb6_{bin}) = 4$).

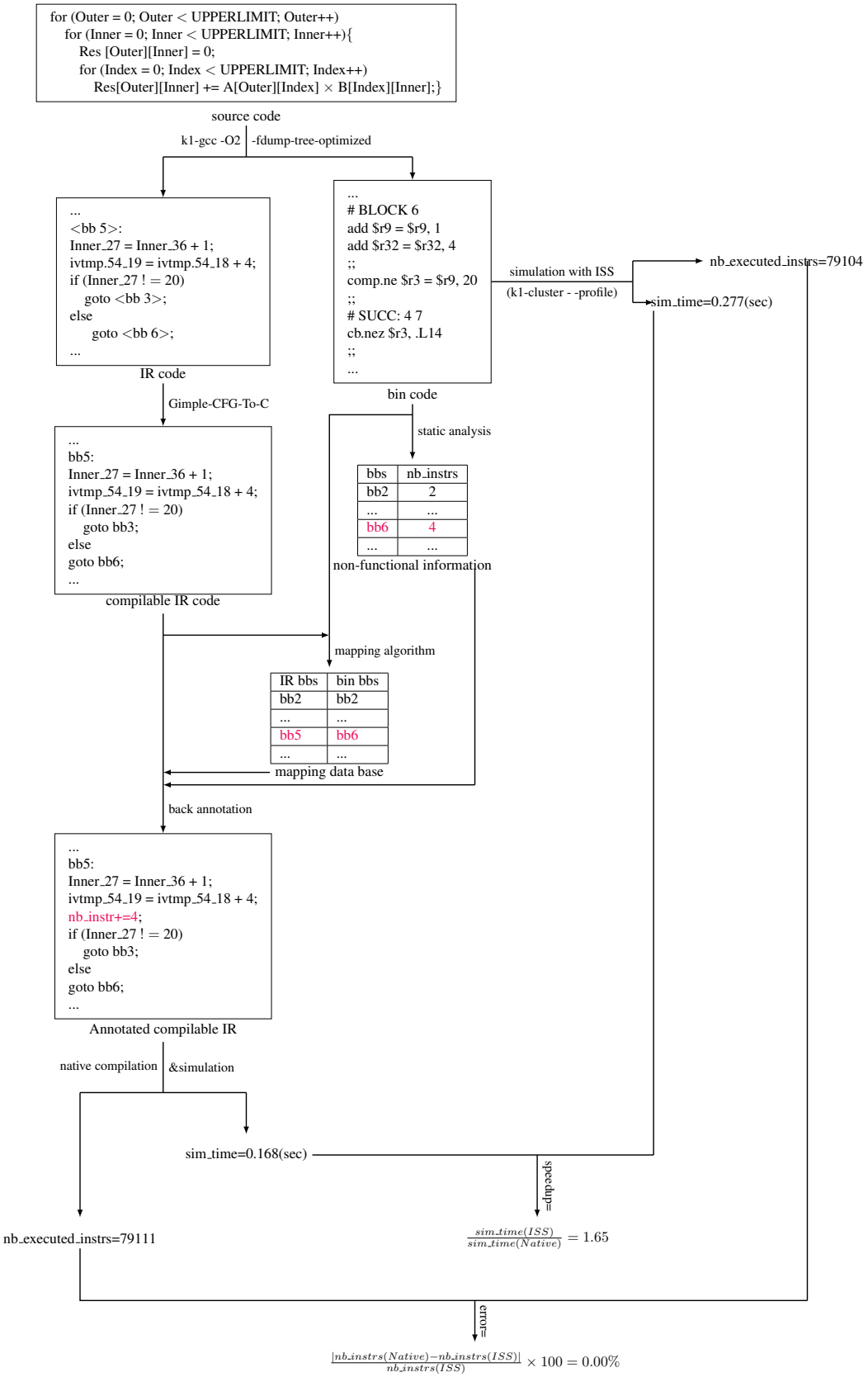


Figure 6.6: Validation of the mapping algorithm at the `gcc -O2` optimization level using the instruction count as a performance metric

Then, the mapping algorithm is applied on both the target binary and the IR CFGs leading to a mapping data base in which basic blocks of the IR are associated to their matching basic blocks in the target binary CFG (e.g. $bb5_{ir} \Leftrightarrow bb6_{bin}$). According to the established mapping information, the instruction count of a binary basic block is accurately inserted in the corresponding IR basic block.

An instruction counter (`nb_instr`) is introduced in the optimized code, which follows the execution path while keeping track of the number of instructions of each visited basic block. Finally, the annotated compilable IR is compiled on the host machine (`k1nsim-gcc -O2 -o annot_ir annot_ir.c`) and is natively executed (`k1nsim-cluster --annot_ir`). At the end of the simulation, the estimated number of executed instructions as well as the simulation time are output.

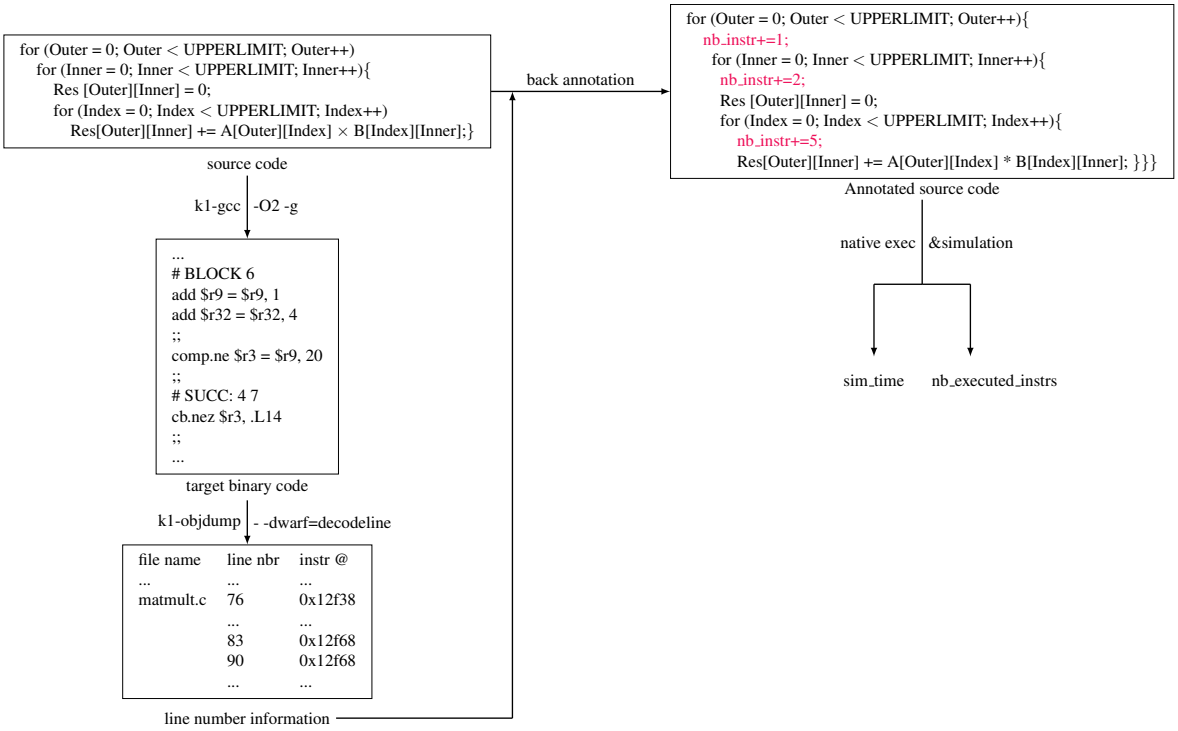


Figure 6.7: Mapping source code to target binary code using debug information for SLS

The speedup gained by native simulation is computed as follows:

$$speedup = \frac{sim\_time(ISS)}{sim\_time(Native)}.$$

The error percentage caused by the mapping algorithm is determined using the following formula:

$$error(\%) = \frac{|nb\_exec\_instrs(Native) - nb\_exec\_instrs(ISS)|}{nb\_exec\_instrs(ISS)} \times 100.$$

The mean absolute error (a.k.a. average error) of N applications is:

$$error_{avg}(\%) = \frac{\sum_{i=1}^{N} error_{i(\%)}}{N}.$$

In addition to our IR-based approach (a.k.a ILS), in which the mapping is conducted using the proposed loop-oriented algorithm, we also implemented the traditional SLS approach, in which the mapping between the source code and the binary code is conducted based on debug information (fig. 6.7). This approach is inspired from [Wan10]. The source code is cross-compiled with the debug option (`k1-gcc -O2 -g`). Debug information is generated from the target binary code (`k1-objdump --dwarf=decodeline`), and a file containing line number information is produced. In this file, source line numbers are coupled with the addresses of their corresponding target machine instructions. Based on this information, we annotate the source code with the instruction count. The annotated source code is then compiled and executed natively. The results of SLS are an estimation of the number of executed instructions, as well as the simulation time. These metrics are compared to the results generated by our approach (ILS) and the cycle accurate simulator (ISS).

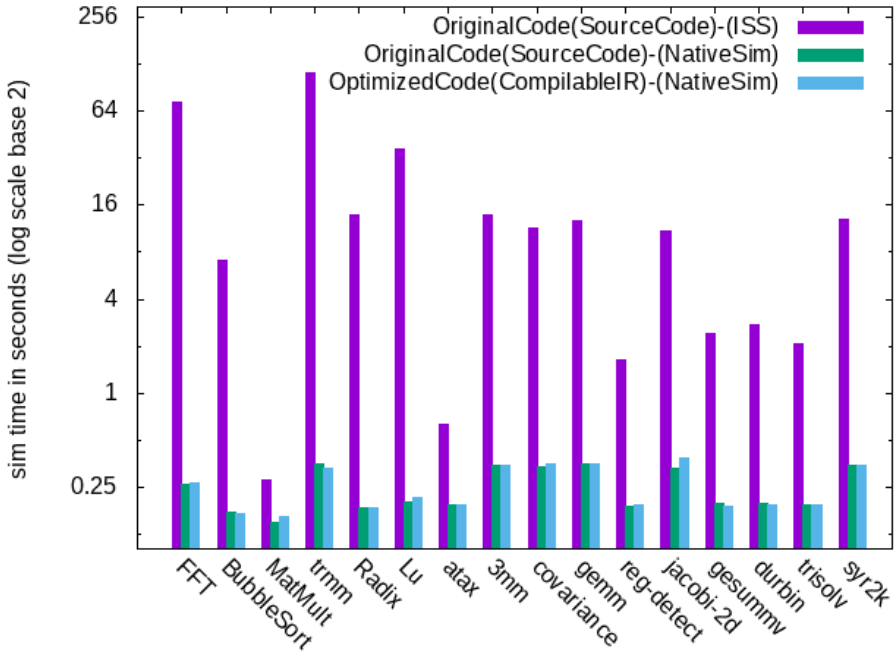


Figure 6.8: Comparison of the simulation time between ISS and native simulation

Analysis of the Results

Table 6.3 lists the main optimizations, in each application, observed at the optimization level *O2*. Optimizations that are enabled by the compiler in a given application are marked with the "×" symbol, otherwise, they are marked with "-". The table also provides information about the number of loops in each application and the degree of loop nests.

Fig. 6.8 shows the simulation time in seconds of the original codes executed using the ISS and the native simulation platform. Native Simulation offers a significant speedup for all the applications. As we can notice, the use of the IR code that we transformed to C code (referred to as optimized code in fig. 6.8) does not introduce any prominent effect on the simulation time compared to the original code, using the native simulation platform. In fact, the slowdown is approximately a factor of 1 for the different applications.

Table 6.4 shows the number of the executed instructions of the natively-executed optimized code (ILS) compared to the number of the executed instructions of the original code

Table 6.3: *O2* optimizations observed for each application

	FFT	bubbleSort	matmult	Radix	Trmm	Lu	atax	3mm
# loops	35	2	3	24	3	23	4	9
# NL*(ND*)	9(4,4,3,2,1,1,1,1,1)	1(1)	1(2)	2(1,1)	1(2)	8(1,1,1,1,1,1,1,3)	1(1)	3(2,2,2)
complete unrolling of small loops	×	-	-	×	-	×	-	-
small fct inlining	×	-	-	×	×	×	×	×
branch optims (mainly cross jumping, if conversion and jump threading)	×	×	×	×	×	×	×	×
other loop optims	×	×	×	×	×	×	×	×

	covar.	gemm	reg-detect	jacobi	gesu.	durbin	trisolv	syr2k
# loops	7	3	10	5	2	4	2	5
# NL*(ND*)	3(1,1,2)	1(2)	1(3)	1(2)	1(1)	1(1)	1(1)	2(1,2)
complete unrolling of small loops	-	-	-	-	-	-	-	-
small fct inlining	×	×	×	×	×	×	×	×
branch optims(same)	×	×	×	×	×	×	×	×
other loop optims	×	×	×	×	×	×	×	×

*NL: nesting loop (outer loop or level 0 loop), *ND: nesting degree (number of inner loops inside a NL)

run on the ISS, and the resulting error (ILS-ERROR). The results of SLS are also compared to ISS, and the error is presented in table 6.4.

As can be noticed, the SLS-ERROR has elevated values. These high values are caused by the unreliable debug information that failed to keep track of compiler optimizations. The impact of these optimizations on the structure of the code is significant, especially in the presence of a large number of loops. On the other hand, the use of our mapping scheme that takes into account compiler optimizations and focuses on loops led to a small percentage of error. This small ILS-ERROR is due to some differences between the binary CFG and the IR CFG (caused by some compiler back-end optimizations, which are target-specific optimizations that are not present in the IR) that could not be handled by our mapping scheme.

Experiments on instruction count, using the IR as a functional model and the proposed mapping algorithm, show, in average, around 2% of error while maintaining a considerable speedup compared to instruction set simulation.

Table 6.4: Comparison of the number of executed instructions (*O2*)

Benchmark	FFT	bubbleSort	matmult	Radix	Trmm	Lu	atax	3mm
ISS	35399998	5451974	79104	6565782	148695	16672930	28251	782941
ILS	35038273	5451998	79111	6572218	151822	16568886	28571	806692
ILS-ERROR	-1.02%	0.00%	0.00%	0.09%	2.10%	-0.62%	1.13%	3.03%
SLS-ERROR	-67.08%	-81.51%	-21.97%	-39.48%	-28.84%	-41.55%	-53.14%	-22.4%

Benchmark	covar.	gemm	reg-detect	jacobi	gesu.	durbin	trisolv	syr2k
ISS	175205	320192	10251	66719	28279	26224	14911	479761
ILS	179861	318642	10310	68022	29066	27232	15160	489156
ILS-ERROR	2.66%	-0.48%	0.58%	1.95%	2.78%	3.84%	1.67%	1.96%
SLS-ERROR	-69.98%	-31.05%	-66.22%	-74.97%	-68.2%	-68.67%	-61.63%	-75.55%

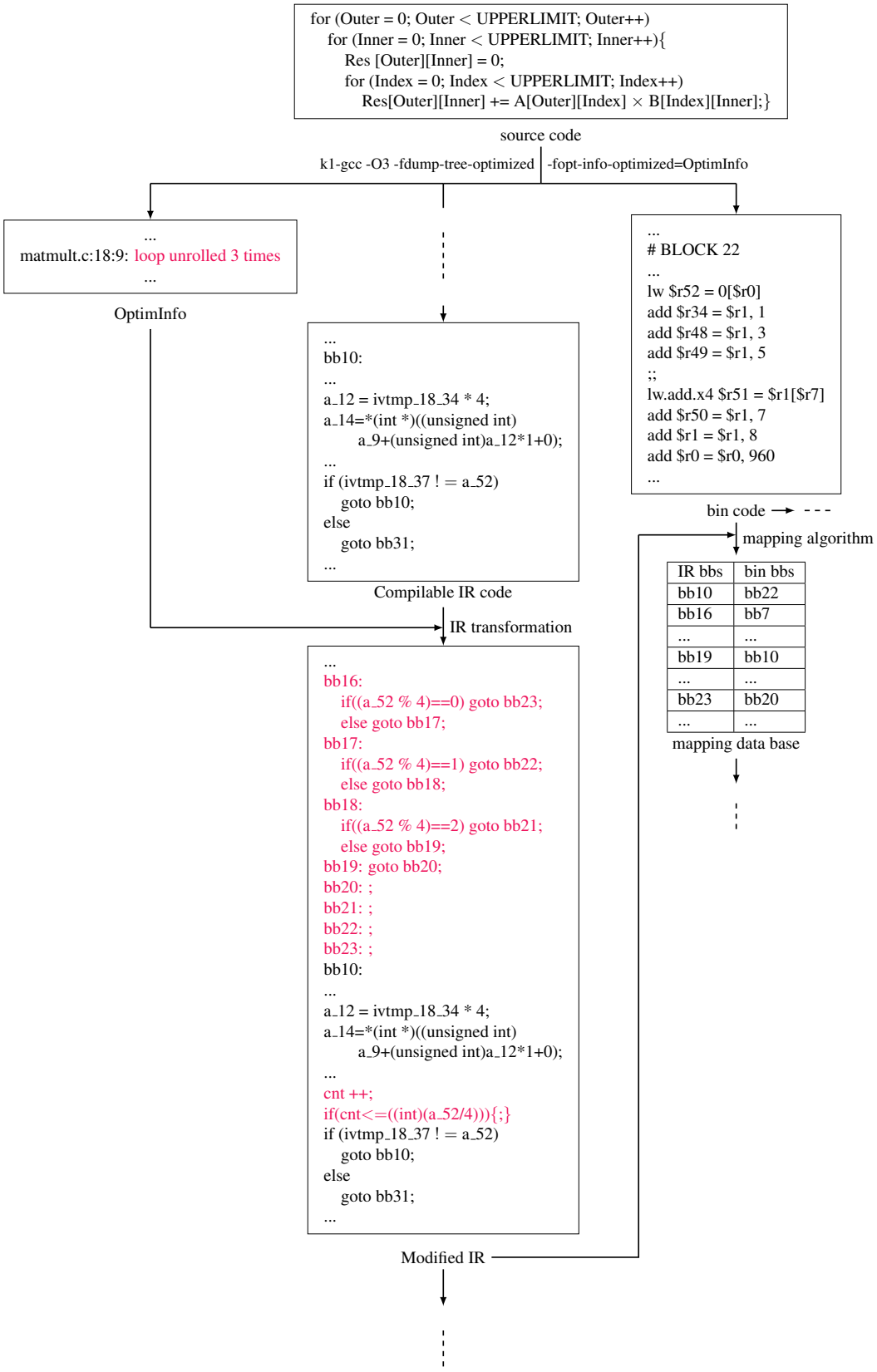


Figure 6.9: Validation of the mapping algorithm at the `gcc -O3` optimization level using the instruction count as a performance metric

6.3.2 Upgraded Mapping Scheme

Applied in isolation, the different cases of loop unrolling and their effect on the mapping method were explained in chapter 4, section 4.3.2, using examples contrived to highlight each case. However, in real applications compiler optimizations are not enabled separately. Instead, they depend on each other and many are only beneficial when others have been applied beforehand to lay the groundwork.

The applications used in this experimentation are compiled with gcc's highest level of optimization `O3`, where aggressive loop optimizations, like loop unrolling along with other code transformations, like branch optimizations, are turned on. We left the compiler in charge of choosing the right combination of optimizations (i.e. which optimizations should be enabled together) as our objective is not to evaluate nor to improve the performance of compiler optimizations. Instead, we are only concerned with providing an accurate mapping scheme, while leaving the task of effectively choosing the optimizations to the compiler.

Experimental Setup

The experimental process of the upgraded mapping scheme involves a few more steps than the basic approach. These steps can be depicted in fig. 6.9. We also use the instruction count as a performance metric to validate the accuracy of the mapping approach.

The source code is cross-compiled, this time, at the `O3` optimization level and with an additional compilation option (`-fopt-info-optimized=OptimInfo`) in order to output a file (*OptimInfo*) containing information about certain optimizations performed by the compiler at the `O3` level. More scpecifically, we are looking for the unrolling factor chosen by the compiler, which we will use later on to introduce a few modifications to the IR code for annotation purposes. The *OptimInfo*, in the example of fig. 6.9, indicates that the unrolling factor of the inner loop is $UF = 3$. Accordingly, we modify the compilable IR by adding a prologue contating *non-functional* basic blocks to account for the peeled instructions in the binary code. The number of *leftover* iterations ($a\_52\%(3+1)$) is determined at run time (because the *trip count* is unknown at compile time in this example), which explains the `if-else` structure preceding the loop body (*bb*10 in the IR code). We also add a counter (`cnt`) and a condition on the counter (as explained in chapter 4, section 4.3.2) inside the loop, in order not to exceed the number of iterations of the unrolled loop when calculating the number of instructions.

After performing the necessary modifications (structure-wise) to the IR, the CFG of the modified IR is closer to the binary CFG. At this stage, we apply algorithm 1 to the binary CFG and the modified IR CFG, leading to the generation of a mapping data base. The remaining steps (binary basic block analysis, back-annotation of the instruction count, execution of the binary with ISS, native simulation of the annotated compilable IR, etc.), which are not represented in fig. 6.9, are identical to the steps in fig. 6.6.

Three mapping methods are tested: the proposed mapping approach *ILS+O3Map* that handles `O3` optimizations, the former mapping approach *ILS+O2Map* (described in chapter 4, section 4.3.1) that yields accurate results with `O2` optimizations, but is not adapted to `O3` optimizations, and finally the former mapping approach readjusted to cater for loop unrolling *ILS+O2Map+*. In the latter approach, we only protected the instruction count in the IR loops with a test (like we did with `cnt` in fig. 6.9), so as to simulate the number of iterations of the corresponding unrolled loop. However, no prologue/epilogue is added to account for the peeled iterations, as the former mapping algorithm does not introduce any modification to the IR control flow.

Table 6.5: O3 optimizations observed for each application

	matmult	bubbleSort	covar	atax	reg-detect	trmm	gemver	trisolv	jacobi-2d
# loops	3	2	7	4	10	3	7	2	5
# NL*(ND*)	1(2)	1(1)	3(1,1,2)	1(1)	1(3)	1(2)	3(1,1,1)	1(1)	1(2)
# UL*(UF)	1(7)	1(7)	3(7,7,7)	2(7,7)	3(7,7,7)	1(7)	4(3,7,7,7)	1(7)	2(3,7)
branch optims	-	×	×	×	-	-	-	×	-
HW loop	×	-	×	×	×	×	×	×	×
other observations				loop-distribute-patterns: loop initialization is distributed (split to 0 loops) and a library call (memset zero) is generated					

	syr2k	3mm	lu	blowfish	crc	fft1
# loops	5	9	4	9	3	11
# NL*(ND*)	2(1,2)	3(2,2,2)	1(2)	3(1,1,1)	0	2(2,1)
# UL*(UF)	2(7,3)	3(7,7,7)	2(7,7)	$3(16_{cmp^*}, 16_{cmp}, 5_{cmp})$	$2(3,9_{cmp})$	$3(8_{cmp}, 8_{cmp}, 6_{cmp})$
branch optims	-	×	-	-	×	-
HW loop	×	×	×	-	×	-
other observations				loop-distribute-patterns: one loop is distributed (split to 0 loops) and a library call (memcpy) is generated	loop unswitching: loop-invariant conditions are moved out of the loop, 2 loops are created	

*NL: nesting loop (outer loop or level 0 loop), *ND: nesting degree (number of inner loops inside a NL), *UL: unrolled loop, *cmp: completely unrolled loop

For each mapping approach, the number of executed instructions is generated at the end of the native execution. It is compared to the number of executed instructions of the original code executed on the ISS provided by Kalray.

Analysis of the results

Table 6.5 lists the main O3 optimizations that are enabled (marked with "×" symbol) by the compiler for each application and that have an impact on the structure of the code. It also provides information about the number of loops in each application, the degree of loop nests, and whether loop unrolling is performed, in which case the unrolling factor is pointed out.

Table 6.6 shows a comparison of the instruction count and the simulation time between the IR, which is simulated natively and mapped to the binary using the three mapping methods (each one at a time), and the target binary code executed on the ISS.

As expected, *ILS+O2Map* causes overestimations in the instruction count, which are proved by the high error values that reached a maximum of 584.75% for application *3mm*. Only by simulating the number of iterations of the unrolled binary loop in its corresponding rolled loop in the IR (*ILS+O2Map+*), we notice a considerable improvement of the error values. The absolute error value dropped from 584.75% (with *ILS+O2Map*) to 11.43% (with *ILS+O2Map+*) for this application. This confirms that inaccuracies come primarily from mapping errors inside loops because even a small error will become critical if it is repeated as many times as the number of loop iterations. As for the negative error values, they can be explained by the non consideration of the peeled iterations, as well as the other O3 optimizations in the mapping approach.

On the other hand, the proposed mapping approach *ILS+O3Map* yields accurate results as it fully handles the different cases of loop unrolling, as well as the other O3 optimizations. Structural transformations are introduced in the IR, so as to have a CFG similar to the binary CFG. As a result, the average instruction count error is 0.59%.

Applications *blowfish* and *fft1* yield the same results with the three mapping approaches because at O3 there are no further structural dissimilarities between the IR and the binary CFGs. In fact, as indicated in Table 6.5, the compiler performs complete loop unrolling in both applications. As mentioned earlier, even though this optimization contributes to the modification of the CFG, it is carried out before the compiler back-end so, it is present in the

IR. Moreover, in *blowfish* another optimization called *loop-distribute-patterns* is performed. This optimizaion is also present in the IR.

As for the simulation time, native simulation is noticeably much faster than ISS. Even with the transformations that we introduced in the IR to deal with compiler optimizations (*ILS+O3Map*), the average speedup is 24.83. It should be noted that the simulation time includes the overhead caused by the instantiation and destruction of the simulation components.

There are, however, certain cases (*matmult, blowfish, crc, fft1*), where the speedup is below 5. In these cases, small speedup values are also observed with the original source codes. In fact, these applications are simple and do not make any *OS* function calls. This leads to fast simulation times even with the ISS (less than $1s$), which may explain the small gains in simulation performance obtained with native simulation.

6.4 Performance Estimation of the Instruction Cache and Instruction Buffer in a VLIW Architecture

In addition to the instruction count, we measure the accuracy of the simulation by taking into account the effects of the instruction cache and instruction buffer in a VLIW architecture. We evaluated our instruction cache model by simulating 8 programs from Polybench [Pou] compiled with `gcc -O2`.

Table 6.6: Comparison of instruction count and simulation time (O3)

		matmult	bubbleSort	covar	atax	reg-detect	trmm	gemver
instr_count	ISS	155993	2646028	151302	25748	9892	136033	40556
	ILS+O3Map	155993	2656128	154561	25684	10011	136321	40809
	error_O3Map	+0.0%	+0.38%	+2.15%	-0.25%	+1.2%	+0.21%	+0.62%
	ILS+O2Map	954293	10498510	902115	109985	18213	862273	176398
	error_O2Map	+511.75%	+296.76%	+496.23%	+327.16%	+84.12%	+533.87%	+334.95%
	ILS+O2Map+	102893	3600010	98327	14625	5741	88129	29686
	error_O2Map+	-34.04%	+36.05%	-35.01%	-43.20%	-41.96%	-35.21%	-26.80%
sim_time(s)	ISS	0.624	2.863	9.006	2.020	1.396	38.086	4.208
	ILS+O3Map	0.180	0.184	0.284	0.196	0.180	0.348	0.196
	speedup_O3Map	3.47	15.56	31.71	10.31	7.76	109.44	21.47
	ILS+O2Map	0.170	0.180	0.282	0.192	0.172	0.348	0.188
	speedup_O2Map	3.67	15.91	31.94	10.52	8.12	109.44	22.38
	ILS+O2Map+	0.176	0.180	0.282	0.194	0.176	0.348	0.188
	speedup_O2Map+	3.56	15.91	31.94	10.41	7.93	109.44	22.38

		trisolv	jacobi-2d	syr2k	3mm	lu	blowfish	crc	fft1
instr_count	ISS	14089	63726	464043	836817	111592	145564	15714	1467
	ILS+O3Map	14135	64020	466986	842961	112680	145823	15834	1467
	error_O3Map	+0.33%	+0.46%	+0.63%	+0.73%	0.97%	+0.18%	+0.76%	0.00%
	ILS+O2Map	25236	179452	1778442	5730076	509272	145823	18375	1467
	error_O2Map	+79.12%	+181.60%	+283.25%	+584.75%	+356.37%	+0.18%	+16.93%	0.00%
	ILS+O2Map+	2836	31561	445450	741148	99898	145823	15823	1467
	error_O2Map+	-79.87%	-50.47%	-4.01%	-11.43%	-10.48%	+0.18%	+0.69%	0.00%
sim_time(s)	ISS	1.940	9.594	11.813	15.684	9.751	0.629	0.536	0.542
	ILS+O3Map	0.192	0.240	0.336	0.332	0.264	0.180	0.188	0.184
	speedup_O3Map	10.10	39.98	35.16	47.24	36.94	3.49	2.85	2.95
	ILS+O2Map	0.184	0.236	0.332	0.328	0.260	0.180	0.184	0.184
	speedup_O2Map	10.54	40.65	35.58	47.82	37.50	3.49	2.91	2.95
	ILS+O2Map+	0.188	0.238	0.336	0.332	0.262	0.180	0.184	0.184
	speedup_O2Map+	10.32	40.31	35.16	47.24	37.22	3.49	2.91	2.95

Experimental Setup

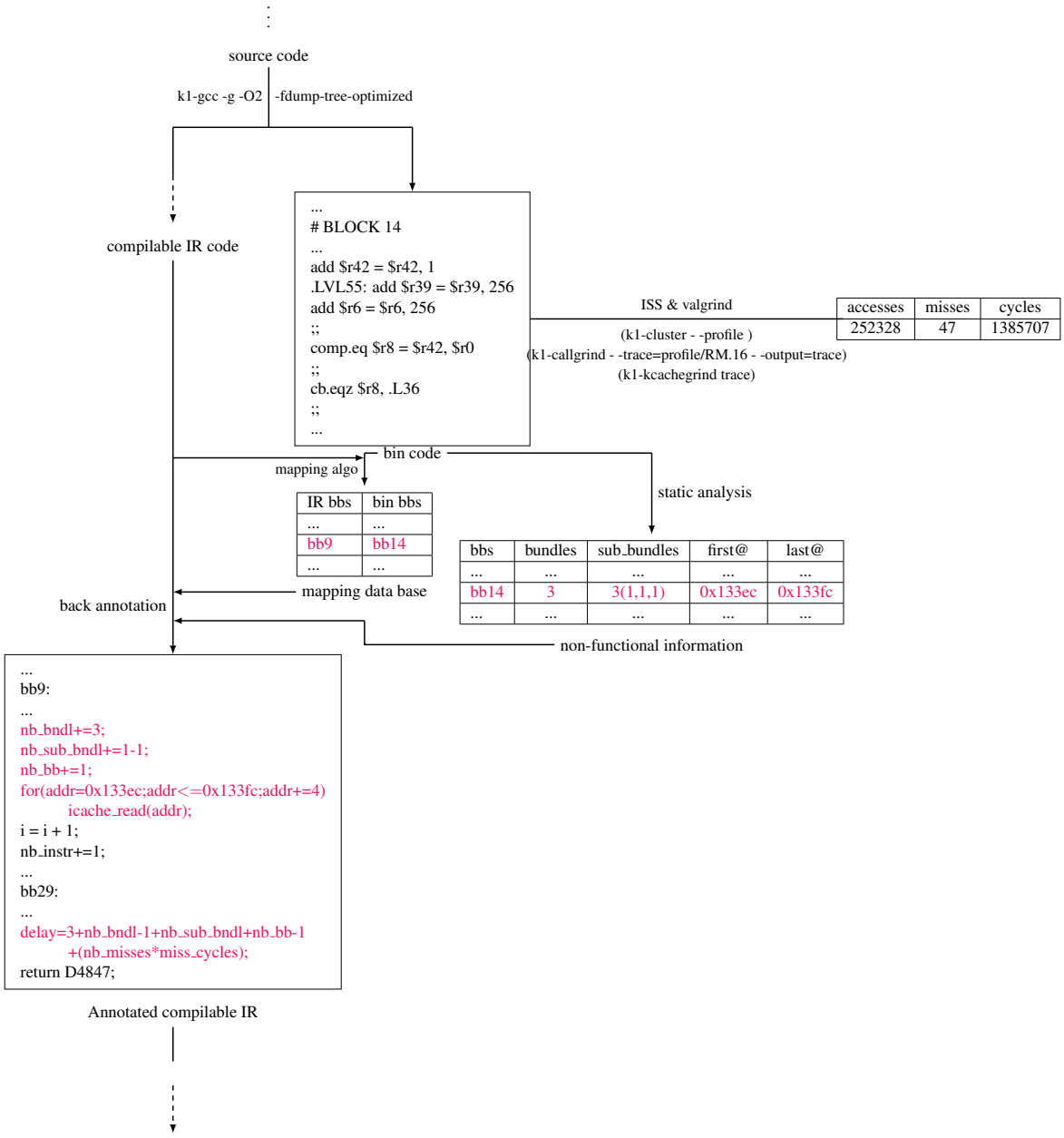


Figure 6.10: IR annotation with instruction cache function calls and performance metrics

To validate our instruction cache model, which is a replica of the real cache, we also used the ISS platform as a reference and we based our comparison on the miss count, cycle count and the simulation time. Thus, the number of accesses to the instruction cache, as well as the number of misses, should be generated for the target binary simulated by the ISS, as well as the natively simulated code.

Listing 6.4: Instruction cache configuration and declaration

```
1 #define  ICACHE_LINES       128 /*number of lines*/
2 #define  ICACHE_ASSOC_BITS    1 /* 2 sets */
```

```
3 #define  ICACHE_LINE_BITS    6 /* 2^6=64 lines per set*/
4 uint8_t  **icache_flags;       /* [cpuid][set_way(target_addr)] */
5 uint32_t **icache_tag;         /* [cpuid][set_way(target_addr)] */
```

Listing 6.4 illustrates the configuration of the instruction cache model, which is identical to the configuration of the hardware cache.

The instruction cache model is implemented using two arrays. One array holds the tags and the other holds the flags, as represented in listing 6.4. The data (i.e. instructions here) is not represented because the model is behavioral, as we explained before. We allocate an instruction cache (a tag array and a flag array) for each processor of the platform. In the experiments we conducted, we used one processor ($nb\_cpu\_cache = 1$).

As portrayed in fig. 6.10, the target binary code is executed on the ISS leading to the generation of a trace file (*RM.16*), which is then interpreted by the profiling tool `callgrind`. The profiling data is displayed by the visualization tool `kcachegrind`. This profiling tool gives us access to non-functional information, such as the number of instruction cache accesses, number of misses/hits and the number of cycles.

To generate such information by native simulation, the functional model (a.k.a. compilable IR) is supplemented with function calls to an instruction cache model, as well as other non-functional information, to account for the instruction buffer effects. To do so, a data base containing information about each target binary basic block is formed (fig. 6.10). In order to compute the delay caused by the instruction cache (formula (5.6)), information like the number of executed bundles, basic blocks, sub-bundles and the number of misses are required. From each basic block of the target binary CFG, we extract the number of bundles, as well as their first and last addresses, which are used to divide the bundles into sub-bundles (instructions inside a sub-bundle belong to the same cache line) and determine the number of sub-bundles (e.g. $bb14$ is made of 3 bundles, each bundle is composed of 1 sub-bundle $3(1,1,1)$, fig. 6.10, "*non-functional information*" table).

Accordingly, three counters are introduced in each basic block of the IR code: `nb_bb`, `nb_bndls` and `nb_subbndls`. Each time a basic block is visited during the native execution of the IR code, the counters are updated with the corresponding values obtained from the basic block data base.

This non-functional information is inserted in the compilable IR, according to the proposed mapping algorithm 1. So, the accuracy of the estimates depends on the instruction cache and instruction buffer models, the non-functional information obtained from the analysis of the binary code, and also on the mapping algorithm.

Regarding the number of misses, it is updated inside the instruction cache annotation function, which is also inserted in every basic block of the IR code. The delay is computed according to formula (5.6), explained in chapter 5-section 5.2.4, in exit basic blocks of the compilable IR (e.g. $bb29$).

Analysis of the Results

Table 6.7 shows the miss count and the cache access count recorded for different programs executed on the ISS platform and natively executed with the ILS approach that we enhanced with an instruction cache model. The model takes into account VLIW and instruction buffer effects. The average miss error has an absolute value of 2.76%.

It should be noted that since the performance metrics and the annotation functions are inserted in the application code, and not in the *OS* and library functions, performance estimates obtained by native simulation are associated with the simulated application only. To make a fair comparison, the results generated by ISS are based on `valgrind`'s *self costs*

Table 6.7: Instruction cache performance

	ISS		ILS+icacheVLIW		
	accesses	misses	accesses	misses	miss_error
gemm	103835	32	104950	30	-6.25%
reg_detect	3687	16	3706	16	0.00%
bubbleSort	3252	5	3205	5	0.00%
atax	15059	89	14897	88	-1.12%
3mm	252328	47	261634	44	-6.38%
jacobi_2d	24245	14	24703	14	0.00%
trisolv	11280	12	11426	13	8.3%
syr2k	158212	14	156864	14	0.00%

(i.e. the sum of all self costs of instructions belonging to a function) and not the *inclusive costs* (i.e. self cost of a function plus inclusive cost of its callees). Thus, we only consider the application and we disregard the effects of the *OS* and library functions on the performance of the instruction cache in both simulators. Performance estimation of the whole software stack (application + *OS* + libraries) with the proposed annotation framework is possible, using the same methodology, which means that *OS* and library functions should also be transformed into IR code, mapped to their binary counterparts and annotated with performance metrics and instruction cache annotation functions.

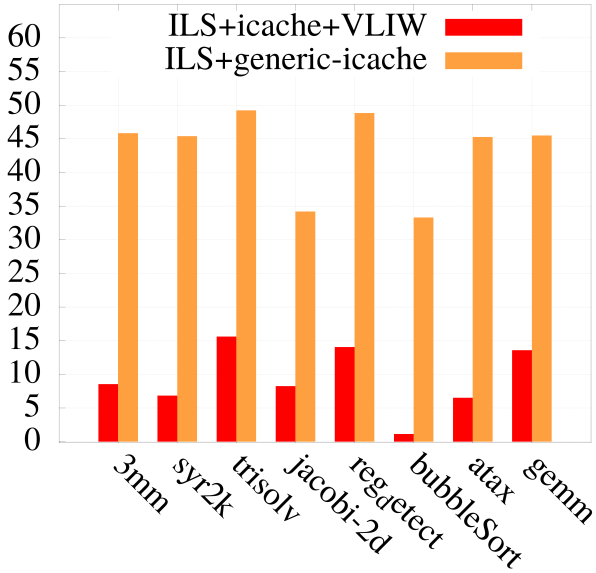


Figure 6.11: Cycle count error (%)

We also used our cache simulation approach to compute the cycle count (table 6.8) and compared it to the ISS and the generic instruction cache simulation approach (i.e. no consideration for the VLIW aspect, instructions are used instead of bundles as described in chapter 5, section. 5.2.2). As depicted in fig. 6.11, the proposed instruction cache simulation approach offers more accurate cycle count estimates than the generic approach for the different simulated programs.

Table 6.8: Comparison of the cycle count

	ISS	ILS+icacheVLIW	
	cycle_count	cycle_count	cycle_error
gemm	595123	675888	13.57%
reg_detect	9511	10847	14.05%
bubbleSort	6220	6290	1.12%
atax	44857	47769	6.49%
3mm	1385707	1503928	8.53%
jacobi_2d	117789	127494	8.24%
trisolv	25494	29470	15.60%
syr2k	1286076	1373793	6.82%

The variation of the miss count error and cycle count error between the different applications is due to compiler optimizations. So, the IR and the binary may have different CFGs. The more different the CFGs are, the less straightforward the mapping process between binary and IR basic blocks becomes and the less accurate the estimates are.

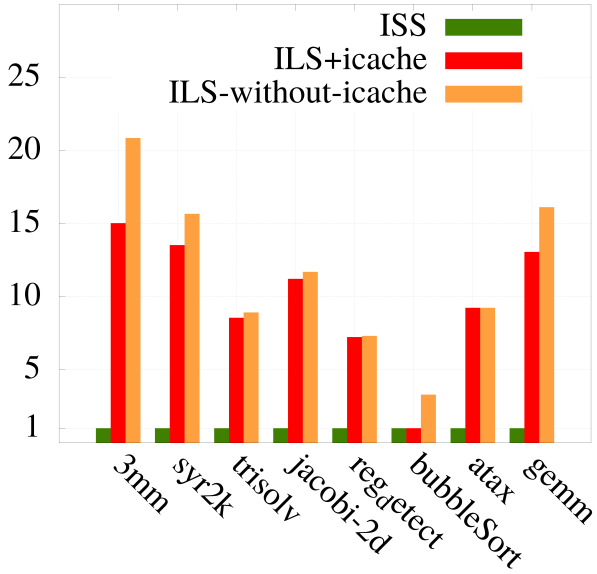


Figure 6.12: Simulation speedup

Fig. 6.12 plots the speedup of ILS-with-icache and ILS-without-icache compared to ISS. Since simulating an instruction cache requires the insertion of annotation functions that trigger the cache model whenever they are reached during simulation, as well as the introduction of several counters in the software code, a simulation overhead is expected compared to ILS-without-icache. The maximum slow down that we have is under 6 for application *3mm* (which is the largest application among the simulated ones in terms of instruction cache accesses).

6.5 Conclusions

This chapter provided experimental results that corroborate the efficiency of our contibutions. These results were obtained from the execution of several applications using a HAV-based native simulation platform that interacts with a TLM-based virtual platform of the target architecture. The native simulation results were compared to the results obtained by the execution of the same applications using a cycle-accurate ISS of the target platform. The simulated target platform is Kalray MPPA-256.

We evaluated the efficiency of both the basic and upgraded mapping approaches using the instruction count as a performance metric. Experimental results of the basic mapping algorithm showed small error values whith applications compiled with `gcc -O2`, but the algorithm reached its limits with applications compiled with `gcc -O3` leading to elevated error values. The upgraded mapping approach that deals with aggressive compiler optimizations yielded accurate results with `gcc -O3` optimizations.

Contributions of this chapter also include a demonstration of the accuracy of the simulation under consideration of a realistic performance model of an instruction cache and instruction buffer of a VLIW architecture. The accuracy was measured in terms of miss count and cycle count. In the proposed performance model, we assumed that the processor requests fixed-size bundles (4-syllable bundles), which is not usually the case in reality. This simplification, along with the possible mapping error, explain the miss count and the cycle count error values. These values are considered low nonetheless. The proposed performance model is useful at early design stages, as it provides reasonable performance estimates at a high simulation speed.

Chapter 7

Conclusions and Perspectives

For design space exploration of complex embedded systems, simulation is key in capturing their dynamic behavior and providing performance estimates. An efficient technique for fast and accurate software performance simulation is necessary in studying the impact of software on the performance of the whole system and making pivotal decisions about task mapping and scheduling and hardware/software partitioning.

In this thesis, we focused on enhancing native simulation, which is able to capture a system's functional behavior, but lacks the capability to estimate its performance. We proposed an IR-level annotation framework that aims at introducing target-specific non-functional information (namely instruction count, cycle count and instruction cache miss count) into a functional model (IR) for the purpose of performance estimation for system-level design (SLD) of embedded systems. For the accurate placement of annotations in the IR, a two-fold mapping approach of the IR and target binary basic blocks that considers compiler optimizations was proposed. We also proposed a performance model of an instruction cache that takes into account the impact of a component necessary to ILP in VLIW architectures, called *instruction buffer*. This performance model is triggered, at simulation time, by the IR, through function calls inserted in the IR's basic blocks according to the proposed mapping algorithm. The conducted experiments prove the accuracy of both the mapping strategy and the instruction cache performance model.

No doubt native simulation is fast as opposed to interpretive methods as it does not depend on the ISA of the target processor, but it suffers from certain shortcomings that hinder the process of performance estimation. The problems raised by native simulation and affecting performance estimation were presented in chapter 2. In this chapter, we will conclude the thesis by recapitulating these problems, underlining the key contributions and presenting the advantages and limitations of the proposed approaches, in section 7.1, and giving an outlook for future work, in section 7.2.

7.1 Conclusions

Although native simulation is one of the most suitable candidates to speed up the architecture space exploration and early design validation steps, it lacks information about the target's performance metrics, which is crucial in software performance estimation. To cater for the absence of such information, target-specific performance metrics are injected into the software code in order to reflect its behavior, as if it was executed on the target platform. This process is referred to as back-annotation and it raises several questions:

• Which representation level of the software (source code, intermediate representation or binary code) is the best candidate for the accuracy/speed trade-off?

The accuracy of the performance estimates not only depends on the accuracy of the devised performance model but also on the software simulation level. This accuracy should not jeopardize one of the main characteristics of host-compiled simulation, which is its speed. Using the source code as a functional model is an abundantly adopted approach in performance estimation techniques for its high level of abstraction, the simplicity of manipulating a source code and its high simulation speed. However, back-annotating a source code with information extracted from the target binary code is very difficult because of the difference between source and binary CFGs. SLS approaches use debug information, which is prone to error because of all the compiler optimizations, to relate source code statements to machine code instructions. So, they resort to turning off compiler optimizations.

Simulation of the target code at the binary level avoids the mapping problems raised by SLS. BLS is always instruction accurate and accounts for all the optimizations of the target compiler. However, the translation of the target binary code into a functionally equivalent high-level code (e.g. C) is a burdensome and time consuming task and it necessitates knowledge of the target ISA. A new target requires a new translation effort.

In our work, we chose the compiler intermediate representation (IR) as our functional model, which offers the best of both worlds (BLS and SLS). The IR is close enough in its structure to the binary code without being dependent on the target ISA. It is also close enough in its syntax to source code, which makes it more readable and easier to manipulate than a binary-level code, while containing compiler front-end optimizations. As we operate on the dump file of the compiler and we do not introduce any modification to its internal structure, unlike some ILS approaches, our approach is compiler non-intrusive and thus retargetable. However, the IR is not compilable, which requires an extra step of compilable-IR generation as opposed to SLS. Also, ILS requires the availability of a source code, whereas BLS allows simulation of target software for which the source code has not necessarily been provided. More importantly, although close to the target binary structure, IR and binary CFGs are not always isomorphic. This issue is expressed by the following question.

• How to establish an accurate mapping between the IR and binary basic blocks in order to accurately place target performance metrics into the functional model? More precisely, how to deal with common compiler optimizations (e.g. `gcc -O2`) and aggressive ones that radically change the structure of the code (e.g. `gcc -O3`)?

The IR and binary CFGs may differ due to the compiler back-end optimizations, which are not present in the IR. Programs spend the majority of their execution time on code inside loops, thus loops are considered as hotspots. In order to reduce the overhead caused by loop statements and thus enhance the speedup of a program, compilers perform several loop transformations. Due to these optimizations, even one annotation misplacement will lead to serious repercussions on the overall performance of the software because this insignificant error will be repeated as many times as the number of loop iterations. Therefore, we dedicated a special focus to loops in our mapping approach.

At the commonly used `gcc -O2` optimization level, most loop optimizations are performed before the back-end. Based on this observation, we proposed a mapping algorithm that aims at matching binary basic blocks to their corresponding IR basic blocks using loops as *fixedpoints*. Loops are pinpointed in both CFGs using the SCC algorithm and are contracted into single nodes to facilitate the mapping process. The algorithm is recursively ap-

plied on loops until the basic block granularity is reached. Experiments on the instruction count prove the efficiency of the mapping approach in dealing with `gcc -O2` optimizations.

At the `gcc -O3` optimization level, aggressive compiler optimizations, such as loop unrolling, that radically change the structure of the code and that can lead to the appearance/disappearance of new/existing loops, are enabled. The proposed algorithm fails at finding accurate mapping at this optimization level. So, we proposed to change the IR CFG so that the mapping becomes feasible. To that aim, we added non-functional basic blocks in the IR that reflect the effect of loop unrolling on the CFG. These basic blocks are only added for annotation purposes and have no impact on the behavior of the code. We also simulated the number of iterations of the unrolled loop in its equivalent IR loop. Other aggressive loop and branch optimizations were taken into account by applying necessary transformations on the IR. These IR transformations, along with the mapping algorithm, yield accurate mapping results at the `O3` optimization level according to the experiments we conducted.

• How to create an accurate performance model that takes into consideration advanced features of modern MPSoCs, such as VLIW architectures?

In native simulation, the reconstruction of the non-functional behavior of the target binary consists of a static part and a dynamic part. Statically computed performance metrics are extracted from the target binary code before simulation (e.g. instruction count). Target micro-architectural components, such as cache memories, have a huge impact on the performance of the system. Their effect cannot be determined at compile time. So, their dynamic behavior is reproduced at simulation time through performance models. The closer the models are to the real hardware component, the more accurate the estimates are.

Modern MPSoCs incorporate cores with complex micro-architectures that support ILP for faster execution of the software. These advanced features call for suitable performance models, capable of reflecting the impact of such features on the performance of the system. In this thesis, we proposed a performance model of an instruction cache and an instruction buffer that considers the effects of a VLIW architecture. We handled the different cases of parallelism among bundles and proposed formulas to compute the delay caused by each case. However, in the proposed model we only considered fixed-size bundles (4 syllables). Despite that we do not account for variable-sized bundles, the average cycle error of the proposed instruction cache and instruction buffer performance model amounts to only 9.3%.

7.2 Perspectives

Different directions for possible future research could be drawn from the limitations of our approach.

• The proposed mapping approach is applied to gcc's intermediate representation and it handles gcc's optimizations. It would be interesting to test the mapping with applications compiled with other compilers (e.g. LLVM).

• Detecting which part of the IR code was optimized in the binary code and what kind of optimization was performed is not a trivial task. Using gcc to ouput a dump file containing information about the performed optimizations can help to an extent. However, the generated file does not provide a comprehensive list. So, we had to further analyze and compare the target binary code and the IR code. Automatizing this step would facilitate and speedup the mapping process.

- Taking into account variable-sized bundles in the instruction cache and instruction buffer performance model, without overloading the model and slowing down the simulation speed, would increase the accuracy of the estimation results

- MPSoC architectures are continuously changing in response to the on going demands for better performance. Complex components, such as advanced caches possibly allowing multiple outstanding misses, highly efficient prefetch mechanisms and branch prediction techniques, etc., may be further implemented in embedded systems. Consideration of these techniques for performance estimation becomes necessary.

- Detailed simulation of massively parallel MPSoCs containing dozens or hundreds of processor cores interconnected through a network on chip (NoC) is afflicted with extensive model development effort and extremely long runtimes. This calls for new modeling and co-simulation methods that preserve simulation accuracy at very high simulation speed. The performance of the co-simulation platform is dictated not only by the software execution approach, but also by the virtual protoypes of the target hardware components. While the software has been successfully back annotated with target-specific performance metrics and simulated by a fast native simulation approach, processor-level effects including the communication aspects should be efficiently modeled, as they contribute significantly to the overall model accuracy. However, modeling and simulating the communication aspects of a many-core architecture can quickly become a hindrance to the simulation performance. Therefore, abstraction should be applied at all layers of the system starting from the software level down to the processor subsystem and communication layers. Communication is usually modeled using TLM, where unnecessary low-level details are ignored, enabling fast MPSoC architecture exploration that would otherwise be almost impossible using low abstraction levels, such as gate-level description or RTL. Despite the rewarding benefits of transaction-level modeling of hardware components, the speedup offered by TLM is not enough for architectures containing hundreds of cores. In fact, traditional sequential simulation kernels (like SystemC) represent the bottleneck of the simulation, as they do not exploit the parallelism offered by modern multi-core SMP workstations. Parallel discrete event simulators (PDES) have emerged as a solution for providing faster simulation of virtual platforms on SMP host machines. In PDES, component models are simulated by different simulation processes, which are executed in parallel by the different cores of an SMP workstation.

So, coupling fast native simulation of annotated software with a parallel virtual simulation platform containing fast and abstract processor subsystem and communication models could be an efficient solution to the extremely long runtimes of complex many-core systems simulations.

Publications

International Conferences

1. Omayma Matoussi and Frédéric Pétrot. Loop aware ir-level annotation framework for performance estimation in native simulation. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 220–225. IEEE, January 2017

2. Omayma Matoussi and Frédéric Pétrot. Modeling instruction cache and instruction buffer for performance estimation of VLIW architectures using native simulation. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 266–269, March 2017

3. Omayma Matoussi and Frédéric Pétrot. IR-level annotation strategy dealing with aggressive loop optimizations for performance estimation in native simulation. In *Work-in-progress at the 2017 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, October 2017

4. Omayma Matoussi and Frédéric Pétrot. A mapping approach between IR and binary CFGs dealing with aggressive compiler optimizations for performance estimation. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, January 2018

Journal

1. Marcos Aurélio Pinto Cunha, Omayma Matoussi, and Frédéric Pétrot. Detecting software cache coherence violations in MPSoC using traces captured on virtual platforms. *ACM Trans. Embedded Comput. Syst.*, 16(2):30:1–30:21, 2017

Glossary

API Application Programming Interface.

BLS Binary Level Simulation.

CA Cycle Accurate.

CFG Control Flow Graph.

CPI Cycle Per Instruction.

DBT Dynamic Binary Translation.

DFS Depth First Search.

DMA Direct Memory Access.

DSE Design Space Exploration.

DSU Debug Support Unit.

DWARF Debugging With Attributed Record Formats.

EU Execution Unit.

HAL Hardware Abstraction Layer.

HAV Hardware Assisted Virtualization.

HDL Hardware Description Language.

HDS Hardware-Dependent Software.

HW Hardware.

IB Instruction Buffer.

IC Integrated Circuit.

IL Intermediate Level.

ILP Integer Linear Programming.

ILP Instruction-Level Parallelism.

ILS Intermediate Level Simulation.

IP Intellectual Property.

IPC Inter-Process Communication.

IR Intermediate Representation.

ISA Instruction Set Architecture.

ISC Intermediate Source Code.

ISS Instruction Set Simulator.

KVM Kernel Virtual Machine.

LLVM Low Level Virtual Machine.

LRU Least Recently Used.

MMIO Memory-Mapped Input Output.

MPPA Massively Parallel Processor Array.

MPSoC Multi Processor System on Chip.

NoC Network on Chip.

NPU Native Processing Unit.

PDES Parallel Discrete Event Simulation.

PE Processing Element.

PFB Prefetch Buffer.

PLRU Pseudo Least Recently Used.

PMIO Port-Mapped Input Output.

PV Programmer's View.

RISC Reduced Instruction Set Computer.

RM Resource Manager.

RTL Register Transfer Level.

SBT Static Binary Translation.

SCC Strongly Connected Component.

SLD System-Level Design.

SLS Source Level Simulation.

SMP Symmetric Multiprocessing.

SoC System on Chip.

SW Software.

TLM Transaction Level Modeling.

TLM-T Transaction Level Modeling with Time.

UF Unrolling Factor.

VLIW Very Long Instruction Word.

VLSI Very Large Scale Integration.

VM Virtual Machine.

VMM Virtual Machine Monitor.

VP Virtual Prototype.

WCET Worst Case Execution Time.

Bibliography

[AFY05] J. A.Fisher, P. Faraboschi, and C. Young. *Embedded Computing, a VLIW approach to architecture, compilers and tools*. Elsevier, 2005.

[ALE02] T. Austin, E. Larson, and D. Ernst. Simplescalar: an infrastructure for computer system modeling. *Computer*, 35(2):59–67, Feb 2002.

[BBB+11] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[BBY+05] A. Bouchhima, I. Bacivarovx, W. Youssef, M. Bonaciu, and A. A. Jerraya. Using abstract cpu subsystem simulation model for high level hw/sw architecture exploration. In *Asia and South Pacific Design Automation Conference*, volume 2, pages 969–972. IEEE, Jan 2005.

[BDH+06] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The m5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, July 2006.

[BEP04] D. Berlin, D. Edelsohn, and S. Pop. High-level loop optimizations for gcc. In *GCC Developers' Summit*, 2004.

[BGP09] A. Bouchhima, P. Gerin, and F. Pétrot. Automatic instrumentation of embedded software for high level hardware/software co-simulation. In *Asia and South Pacific Design Automation Conference*, pages 546–551, January 2009.

[BYJ04] A. Bouchhima, S. Yoo, and A. Jerraya. Fast and accurate timed execution of high level embedded software using hw/sw interface simulation model. In *Asia and South Pacific Design Automation Conference 2004*, pages 469–474. IEEE, Jan 2004.

[CHB09a] E. Cheung, H. Hsieh, and F. Balarin. Fast and accurate performance simulation of embedded software for mpsoc. In *Asia and South Pacific Design Automation Conference*, pages 552–557, Jan 2009.

[CHB09b] E. Cheung, H. Hsieh, and F. Balarin. Memory subsystem simulation in software tlm/t models. In *Asia and South Pacific Design Automation Conference*, pages 811–816, Jan 2009.

[CM96] Cristina Cifuentes and Vishv Malhotra. Binary translation: Static, dynamic, retargetable? In *IEEE International Conference on Software Maintenance*, pages 340–349, Los Alamitos, CA, USA, 1996. IEEE Computer Society.

[CMMC08] J. Cornet, F. Maraninchi, and L. Maillet-Contoz. A method for the efficient development of timed and untimed transaction-level models of systems-on-chip. In *2008 Design, Automation and Test in Europe*, pages 9–14, March 2008.

[CMP17] Marcos Aurélio Pinto Cunha, Omayma Matoussi, and Frédéric Pétrot. Detecting software cache coherence violations in MPSoC using traces captured on virtual platforms. *ACM Trans. Embedded Comput. Syst.*, 16(2):30:1–30:21, 2017.

[DAP15] R. J. Douma, S. Altmeyer, and A. D. Pimentel. Fast and precise cache performance estimation for out-of-order execution. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1132–1137, March 2015.

[DdDAB+13] Benoît Dupont de Dinechin, Renaud Ayrignac, Pierre-Edouard Beaucamps, Patrice Couvert, Benoît Ganne, Pierre Guironnet de Massas, François Jacquet, Samuel Jones, Nicolas Morey Chaisemartin, Frédéric Riss, and Thierry Strudel. A clustered manycore processor architecture for embedded and accelerated applications. In *IEEE High Performance Extreme Computing Conference*, pages 1–6. IEEE, 2013.

[DPE11] L. Díaz, H. Posadas, and E.Villar. Fast data-cache modeling for native co-simulation. *Design Automation Conference (ASP-DAC)*, January 2011.

[DR05] Thomas Dullien and Rolf Rolles. Graph-based comparison of executable objects. In *Actes du Symposium sur la securite des technologies de l'information et des communications*, pages 1–13, 2005.

[GCK12] A. Gerstlauer, S. Charkravarty, and M. Kathuria. Abstract system-level models for early performance and power exploration. In *Asia South-Pacific Design Automation Conference*, pages 213–218, January 2012.

[GCM92] R. K. Gupta, C. N. Coelho, and G. De Micheli. Synthesis and simulation of digital systems containing interacting hardware and software components. In *Proceedings 29th ACM/IEEE Design Automation Conference*, pages 225–230, Jun 1992.

[GCZ13] A. Gerstlauer, S. Charkravarty, and Z. Zhao. Automated, retargetable back-annotation for host compiled performance and power modeling. In *International Conference on Hardware/Software Codesign and System Synthesis*, pages 1–10, September 2013.

[Ger09] Patrice Gerin. *Modéles de Simulation pour la Validation Logicielle et l'Exploration d'Architectures des Systémes Multiprocesseurs sur Puce*. PhD thesis, Institut Polytechnique de Grenoble, Grenoble, France, November 2009.

[GHP09] Patrice Gerin, Mian Muhammad Hamayun, and Frédéric Pétrot. Native mpsoc co-simulation environment for software performance estimation. In *Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, pages 403–412. ACM, 2009.

[GYG03] A. Gerstlauer, Haobo Yu, and D. D. Gajski. Rtos modeling for system level design. In *Design, Automation and Test in Europe Conference and Exhibition*, pages 130–135, 2003.

[GYNJ01] P. Gerin, Sungjoo Yoo, G. Nicolescu, and A. A. Jerraya. Scalable and flexible cosimulation of soc designs with heterogeneous multi-processor target architectures. In *Proceedings of the ASP-DAC*, pages 63–68, 2001.

[HDH+10] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, and T. Mattson. 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 108–109. IEEE, 2010.

[HSAG10] Y. Hwang, G. Schirner, S. Abdi, and D. G. Gajski. Accurate timed rtos model for transaction level modeling. In *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, pages 1333–1336, March 2010.

[Int10] Intel. *Intel Itanium Processor 9300 Series Reference Manual for Software Development and Optimization*. Intel, March 2010.

[KGW+07] S. Kraemer, L. Gao, J. Weinstock, R. Leupers, G. Ascheid, and H. Meyr. Hysim: A fast simulation framework for embedded software development. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 75–80, Sept 2007.

[KKW+06] T. Kempf, K. Karuri, S. Wallentowitz, G. Ascheid, R. Leupers, and H. Meyr. A sw performance estimation framework for early system-level-design using fine-grained instrumentation. In *Proceedings of the Design Automation Test in Europe Conference*, volume 1, pages 6 pp.–, March 2006.

[KMGS13] L. Kun, D. Muller-Gritschneder, and U. Schlichtmann. Memory access reconstruction based on memory allocation mechanism for source-level simulation of embedded software. *Design Automation Conference (ASP-DAC)*, January 2013.

[LBH+00] M. T. Lazarescu, J. R. Bammi, E. Harcourt, L. Lavagno, and M. Lajolo. Compilation-based software performance estimation for system level design. In *International High-Level Design Validation and Test Workshop*, pages 167–172. IEEE, 2000.

[Let09] Richard A. Lethin. How vliw almost disappeared - and then proliferated. *IEEE Solid-State Circuits Magazine*, 1(3):15–23, summer 2009.

[LLT10] K. Lin, C. Lo, and R. Tsay. Source-level timing annotation for fast and accurate tlm computation model generation. *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 235–240, January 2010.

[LMGS12] K. Lu, D. M-Gritschneder, and U. Schlichtmann. Hierarchical control flow matching for source-level simulation of embedded software. *System On Chip International Symposium*, pages 1–5, October 2012.

[LMGSB13] K. Lu, D. Muller-Gritschneder, U. Schlichtmann, and O. Bringmann. Fast cache simulation for host-compiled simulation of embedded software. *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 637–642, March 2013.

[LRM06] Xianfeng Li, Abhik Roychoudhury, and Tulika Mitra. Modeling out-of-order processors for wcet analysis. *Real-Time Syst.*, 34(3):195–227, November 2006.

[LSA95] Y. Steven Li, S.Malik, and A.Wolfe. Performance estimation of embedded software with instruction cache modeling. *International conference of Computer-Aided design,IEEE*, November 1995.

[MBF+12] Diego Melpignano, Luca Benini, Eric Flamand, Bruno Jego, Thierry Lepley, Germain Haugou, Fabien Clermidy, and Denis Dutoit. Platform 2012, a many-core computing accelerator for embedded socs: performance evaluation of visual analytics applications. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1137–1142. ACM, 2012.

[MGLS11] D. Mueller-Gritschneder, K. Lu, and U. Schlichtmann. Control-flow-driven source level timing annotation for embedded software models on transaction level. In *Euromicro Conference on Digital System Design*. IEEE, October 2011.

[MP17a] Omayma Matoussi and Frédéric Pétrot. IR-level annotation strategy dealing with aggressive loop optimizations for performance estimation in native simulation. In *Work-in-progress at the 2017 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, October 2017.

[MP17b] Omayma Matoussi and Frédéric Pétrot. Loop aware ir-level annotation framework for performance estimation in native simulation. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 220–225. IEEE, January 2017.

[MP17c] Omayma Matoussi and Frédéric Pétrot. Modeling instruction cache and instruction buffer for performance estimation of VLIW architectures using native simulation. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 266–269, March 2017.

[MP18] Omayma Matoussi and Frédéric Pétrot. A mapping approach between IR and binary CFGs dealing with aggressive compiler optimizations for performance estimation. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, January 2018.

[MPC04] R. Le Moigne, O. Pasquier, and J. P. Calvez. A generic rtos model for real-time systems simulation with systemc. In *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, volume 3, pages 82–87 Vol.3, Feb 2004.

[MRRJ05] A. Muttreja, A. Raghunathan, S. Ravi, and N. K. Jha. Hybrid simulation for embedded software energy estimation. In *Proceedings. 42nd Design Automation Conference.*, pages 23–26, June 2005.

[MSVSL08a] T. Meyerowitz, A. Sangiovanni-Vincentelli, M. Sauermann, and D. Langen. Source-level timing annotation and simulation for a heterogeneous multiprocessor. In *Design Automation and Test Europe*. ACM, March 2008.

[MSVSL08b] T. Meyerowitz, A. Sangiovanni-Vincentelli, M. Sauermann, and D. Langen. Source-level timing annotation and simulation for a heterogeneous multiprocessor. In *Design, Automation and Test in Europe*, pages 276–279, March 2008.

[Nov06] D. Novillo. Gcc - an architectural overview, current status, and future directions. *In Proceedings of the Linux Symposium*, September 2006.

[OSC] Osci tlm-2.0 language reference manual. http://www.accellera.org/images/downloads/standards/systemc/TLM_2_0_LRM.pdf.

[PCB06] S. Pop, A. Cohen, and C. Bastoul. Graphite: Polyhedral analyses and optimizations for gcc. *In Proceedings of the GCC Developers Summit*, pages 1–18, 2006.

[PCG09] Ardavan Pedram, David Craven, and Andreas Gerstlauer. *Modeling Cache Effects at the Transaction Level*, pages 89–101. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[PFG+11] F. Pétrot, N. Fournel, P. Gerin, M. Gligor, M. Hamayun, and H. Shen. On mpsoc software execution at the transaction level. *IEEE design and test of computers*, 28(3):32–43, 2011.

[PJ15] X. Pan and B. Jonsson. A modeling framework for reuse distance-based estimation of cache performance. *Performance Analysis of Systems and Software (ISPASS), IEEE*, pages 62–71, March 2015.

[PLH11] I. Puaut, B. Lesage, and D. Hardy. Scalable fixed-point free instruction cache analysis. *Real-Time System Symposium (RTSS), IEEE*, December 2011.

[Pou] LN. Pouchet. Polybench benchmark. *http://web.cse.ohio-state.edu/ pouchet/software/polybench/*.

[PWH12] R. Plyaskin, T. Wild, and A. Herkersdorf. System-level software performance simulation considering out-of-order processor execution. In *IEEE International Symposium on System On Chip*, pages 1–7, October 2012.

[RS09] Christine Rochange and Pascal Sainrat. *A Context-Parameterized Model for Static Analysis of Execution Times*, pages 222–241. Springer Berlin Heidelberg, 2009.

[Sar16] Guillaume Sarrazin. *Simulation fonctionnelle native pour des systèmes many-cœurs*. PhD thesis, Institut Polytechnique de Grenoble, Grenoble, France, May 2016.

[SB08] J. Schnerr and O. Bringmann. High-performance timing simulation of embedded software. *Design Automation Conference (DAC)*, June 2008.

[SBR11a] S. Stattelmann, O. Bringmann, and W. Rosenstiel. Dominator homomorphism based code matching for source-level simulation of embedded software. In *International Conference on Hardware/Software Codesign and System Synthesis*. ACM, October 2011.

[SBR11b] S. Stattelmann, O. Bringmann, and W. Rosenstiel. Fast and accurate source-level simulation of software timing considering complex code optimizations. In *Design Automation Conference*. IEEE, June 2011.

[SGCB12] S. Stattelmann, G. Gebhard, C. Cullmann, and O. Bringmann. Hybrid source-level simulation of data caches using abstract cache models. *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, March 2012.

[SHP12] H. Shen, M. M. Hamayun, and F. Petrot. Native simulation of mpsoc using hardware-assisted virtualization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(7):1074–1087, July 2012.

[SM06] André Seznec and Pierre Michaud. A case for (partially) tagged geometric history length branch prediction. *J. Instruction-Level Parallelism*, 8, 2006.

[STM04] STMicroelectronics. *ST200 VLIW Series ST231 Core and Instruction Set Architecture Manual*. STMicroelectronics, March 2004.

[Tar72] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[Til] Tile-gx8072 processor product brief. http://www.tilera.com/files/drim_TILE-Gx8072_PB041-04_WEB_1_22_15_7639.pdf.

[TKM+02] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, et al. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE micro*, 22(2):25–35, 2002.

[TRKA07] W. Tibboel, V. Reyes, M. Klompstra, and D. Alders. System-level design flow based on a functional reference for hw and sw. In *Design Automation Conference*, pages 23–28. IEEE/ACM, June 2007.

[Wan10] Zhonglei Wang. *Software Performance Estimation Methods for System-Level Design of Embedded Systems*. PhD thesis, Technical University of Munich, Munich, April 2010.

[WH12] Z. Wang and J. Henkel. Accurate source-level simulation of embedded software with respect to compiler optimizations. In *Design, Automation and Test in Europe Conference and Exhibition*, pages 382–387, March 2012.

[WH13] Z. Wang and J. Henke. Fast and accurate cache modeling in source-level simulation of embedded software. *Design, Automation and test in Europe Conference and Exhibition (DATE)*, March 2013.

[WOT+95] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. *SIGARCH Comput. Archit. News*, 23(2):24–36, May 1995.

[YBB+03] Sungjoo Yoo, I. Bacivarov, A. Bouchhima, Y. Paviot, and A. Jerraya. Building fast and accurate sw simulation models based on hardware abstraction layer and simulation environment abstraction layer. In *Design, Automation and Test in Europe Conference and Exhibition*, pages 550–555. IEEE, 2003.

[YJ03] Sungjoo Yoo and A. A. Jerraya. Introduction to hardware abstraction layers for soc. In *Design, Automation and Test in Europe Conference and Exhibition*, pages 336–337, 2003.

[YMH+14] R. Yan, D. Ma, K. Huang, X. Zhang, and S. Xiu. Annotation and analysis combined cache modeling for native simulation. *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 406–411, January 2014.

[ZH09] W. Zhonglei and A. Herkersdorf. An efficient approach for system-level timing simulation of compiler-optimized embedded software. *Design Automation Conference (DAC)*, July 2009.

[ZM96] V. Zivojnovic and H. Meyr. Compiled hw/sw co-simulation. In *Design Automation Conference Proceedings*, pages 690–695. IEEE, Jun 1996.