Early Timing Exploration of Embedded Software for Heterogeneous Platforms

Dissertation

der Mathematisch-Naturwissenschaftlichen Fakultät
der Eberhard Karls Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von
M.Sc. Alessandro Cornaglia
aus Venedig/Italien

Tübingen
2022

Gedruckt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Eberhard Karls Universität Tübingen.

Tag der mündlichen Qualifikation:	16.07.2024
Dekan:	Prof. Dr. Thilo Stehle
1. Berichterstatter/-in:	Prof. Dr. Oliver Bringmann
2. Berichterstatter/-in:	Prof. Dr.-Ing. Jeronimo Castrillon

„ Those who can imagine anything, can create the impossible.“

Alan Turing

Abstract

Under growing tasks complexity and time-to-market pressures, modern embedded systems are commonly characterized by large software applications that are collaboratively executed on complex multiprocessing and heterogeneous computation platforms. Furthermore, functional and non-functional requirements drive the design and development of such performance-critical embedded systems. On the one hand, these powerful platforms offer high-computation capabilities, but on the other hand, the complexity of their architectures limits the performance analyzability of the complete system. Non-functional properties, such as the execution time, have to be assessed since the early design stages of a system. For this reason, during the design space exploration activities, the designers and the engineers are commonly interested in evaluating different hardware and software configurations for determining the most suitable one for ensuring that the final implementation can satisfy the strict performance requirements. This evaluation is usually conducted via virtual prototypes that rely on a timing simulator. The effectiveness of the evaluation depends on the speed and accuracy of the underlying simulator. Therefore, a simulation methodology is essential for supporting the early evaluation of multiple hardware and software configurations of a system.

This thesis proposes a novel simulation methodology that allows to rapidly and accurately evaluate the performance of multiple configurations of an embedded system. The definition of the simulation methodology requires facing multiple hard challenges that are common to performance simulators. The proposed methodology is based on the program's intermediate representation that is internal to the compiler and it ensures a beneficial level of abstraction for both the analysis and simulation stages. The intermediate representation is architecture-independent and it allows evaluating multiple configurations in only one simulation. This choice requires an appropriate mapping technique for matching the structures of a program at the intermediate and the binary representations. Their structures may substantially differ due to the effects of aggressive compiler optimizations that make a direct matching impossible. For this reason, this thesis proposes an innovative two-phases algorithm for tackling this hard problem. Relying on these mappings and on an appropriate technique for modeling the timing behavior of programs, accurate performance estimations can be produced via fast simulations. Furthermore, the conducted experimental evaluation results show that the simulation methodology can be applied for early evaluating the design of complex heterogeneous systems. Finally, the elevated simulation speed capabilities enable the definition of a new co-simulation technique for evaluating the performance of embedded systems that are developed directly on a model-driven environment such as the one provided by Simulink.

Zusammenfassung

Unter wachsender Aufgabenkomplexität und Time-to-Market-Druck sind moderne eingebettete Systeme üblicherweise durch große Softwareanwendungen gekennzeichnet, die kollaborativ auf komplexen Multiprozessor- und heterogenen Rechenplattformen ausgeführt werden. Darüber hinaus treiben funktionale und nicht-funktionale Anforderungen das Design und die Entwicklung solcher leistungskritischen eingebetteten Systeme voran. Einerseits bieten diese leistungsfähigen Plattformen hohe Rechenleistungen, andererseits schränkt die Komplexität ihrer Architekturen die Analysierbarkeit der Performance des Gesamtsystems ein. Nicht-funktionale Eigenschaften, wie zum Beispiel die Ausführungszeit, müssen bereits in den frühen Entwurfsphasen eines Systems bewertet werden. Aus diesem Grund sind die Designer und Ingenieure während der Entwurfsraumuntersuchung in der Regel daran interessiert, verschiedene Hardware- und Softwarekonfigurationen zu evaluieren, um die am besten geeignete Konfiguration zu ermitteln, die sicherstellt, dass die endgültige Implementierung die strengen Leistungsanforderungen erfüllen kann. Diese Evaluierung wird normalerweise über virtuelle Prototypen durchgeführt, die auf einem Timing-Simulator basieren. Die Effektivität der Evaluierung hängt von der Geschwindigkeit und Genauigkeit des zugrunde liegenden Simulators ab. Daher ist eine Simulationsmethodik unerlässlich, um die frühzeitige Evaluierung mehrerer Hardware- und Softwarekonfigurationen eines Systems zu unterstützen.

In dieser Dissertation wird eine neuartige Simulationsmethodik vorgeschlagen, die es ermöglicht, die Leistung mehrerer Konfigurationen eines eingebetteten Systems schnell und genau zu bewerten. Die Definition der Simulationsmethodik erfordert die Bewältigung mehrerer schwieriger Herausforderungen, die für Leistungssimulatoren üblich sind. Die vorgeschlagene Methodik basiert auf der Zwischenrepräsentation des Programms, die intern im Compiler ist und ein vorteilhaftes Abstraktionsniveau sowohl für die Analyse- als auch für die Simulationsphase gewährleistet. Die Zwischendarstellung ist architekturunabhängig und erlaubt es, mehrere Konfigurationen in nur einer Simulation zu evaluieren. Diese Wahl erfordert ein geeignetes Mapping- Verfahren, um die Strukturen eines Programms in der Zwischendarstellung und der Binärdarstellung abzugleichen. Deren Strukturen können sich aufgrund der Auswirkungen von aggressiven Compiler-Optimierungen, die einen direkten Abgleich unmöglich machen, erheblich unterscheiden. Aus diesem Grund wird in dieser Dissertation ein innovativer Zwei-Phasen-Algorithmus vorgeschlagen, um dieses schwierige Problem zu bewältigen. Basierend auf diesen Mappings und einer geeigneten Technik zur Modellierung des Timing-Verhaltens von Programmen können durch schnelle Simulationen genaue Leistungsabschätzungen erstellt werden. Darüber hinaus zeigen die durchgeführten experimentellen

Evaluierungsergebnisse, dass die Simulationsmethodik zur frühzeitigen Evaluierung des Designs komplexer heterogener Systeme eingesetzt werden kann. Schließlich ermöglichen die erhöhten Simulationsgeschwindigkeiten die Definition einer neuen Co-Simulationstechnik für die Bewertung der Leistungsfähigkeit eingebetteter Systeme, die direkt auf einer modellgetriebenen Umgebung wie der von Simulink entwickelt werden.

Acknowledgements

I would like to thank my advisors, Prof. Dr. Oliver Bringmann and Prof. Dr.-Ing. Jeronimo Castrillon for their kindness and support in realizing this dissertation.

Furthermore, I would also like to express my sincere thanks to all the former colleagues at the Department of Microelectronic System Design (SiM) with whom I shared a great working experience at FZI Forschungszentrum Informatik in Karlsruhe. I am profoundly grateful to Dr. Alexander Viehl and Dr. Sebastian Reiter for the support I received for achieving my scientific results and their kindness. Among all the former colleagues, I can't forget to thank the wise Frederik Haxel and my darling officemate Leon Hielscher for their patience in always listening to my crazy ideas and for their suffering in reviewing all my publications.

Finally, I would like to thank my lovely wife Giulia for her commitment and all her sacrifices that made possible the dream that we are currently living.

Contents

Dedicated to my father.

CHAPTER 1

Introduction

Nowadays, modern embedded and cyber-physical systems are widely spread across multiple domains covering a copious range of heterogeneous applications that pervade everyday life. Different, and sometimes conflicting, requirements drive the development of the hardware platforms designed for efficiently executing such systems. In this context, non-functional requirements are as important as the functional ones. Non-functional requirements describe the expected behavior of a system considering specific non-functional characteristics. For example, automotive, avionic, railways and medical performance-critical applications are commonly subjected to strict timing requirements. Contrarily, power and energy constraints are common requirements for devices powered by batteries like smartphones and IoT nodes. Both the requirement types can be mixed in complex applications composed of multiple functional units subjected to different levels of criticality.

The hardware manufacturers tend to support the development of new complex systems by offering powerful hardware platforms that include multiple heterogeneous processing units and so-called Multi-Processor System on Chip (MPSoC). Every processing unit is designed and optimized for fulfilling specific tasks and objectives [91]. A first example consists of the Infineon Tricore board [2]. This MPSoC is designed for safety-critical domains, where the functional units of a complex application can be statically mapped to energy efficient or to performance oriented cores. A different example consists in the ARM big.LITTLE platform [93], whose concept is shown in Figure 1.1. Slower battery-saving processors (LITTLE) are interconnected with more powerful but battery-intensive processors (big) in order to optimize the application's usage at run-time.

Unfortunately, the continuously growing complexity of the hardware mechanisms included in modern processors, generally designed for improving the systems performance, reduces instead the timing analyzability of the target systems [169]. This limitation is problematic for the design exploration activities focused on the partitioning and mapping of the software functional units to the available processing units in respect of the non-functional requirements. In order to support the design exploration activities, the definition of a methodology is required for allowing a fast and accurate evaluation of the necessary design decisions.

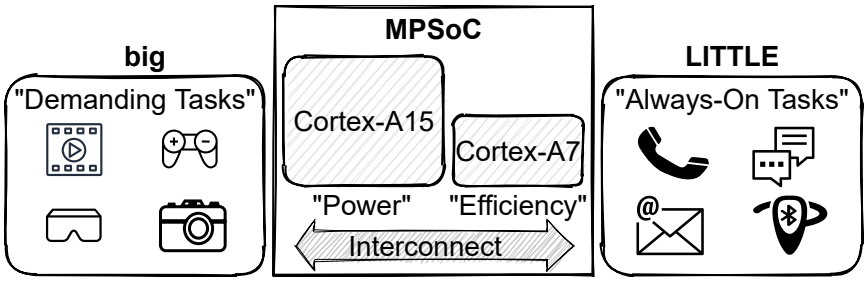


Figure 1.1 – *ARM big.LITTLE processing concept: "Use the right processor for the right core". The ARM big.LITTLE heterogeneous processing combines a "LITTLE" energy saving core with a "big" high-performance core for executing programs that dynamically switch the execution core depending on the performance needs (figure adapted from [93]).*

1.1 Motivation

In the embedded systems domain, it is a common practice that a complex system is initially developed on a powerful development host machine. The host and the target platforms generally implement different hardware architectures. The differences between the architectures imply that, during the development of the software program, some architecture-dependent considerations about the target platform have to be taken into account. Furthermore, in industrial settings, manual coding activities tend to be reduced as much as possible. The source code of an application is commonly automatically generated by relying on model driven development tools and taking advantage of their ensured benefits [5]. Nevertheless, the porting activities, necessary for making the software executable on the designated target platform, are consequent to some initial assessment conducted on the host. Unfortunately, the discrepancies between the development and execution environments make impossible a direct evaluation of both the functional and non-functional properties of the target system on a host machine. An effective evaluation can be conducted only on the target environment but this is commonly too expensive in terms of resources and time. Furthermore, considering that both the hardware resources included in a platform and the program's input data influence the run-time behavior of a system, it is unfeasible to evaluate all their possible combinations directly on the target.

The established approach for overcoming the critical challenges due to the development of a complex embedded system is the Electronic System Level (ESL) design [56]. The enhanced probability of a successful implementation is ensured by adopting an appropriate level of abstraction when concurrently developing hardware and software components. This level of abstraction can be obtained by describing a system relying on one or multiple system level design languages (SLDL) [141]. An example of SLDL is SystemC [121], a language that allows to describe virtual prototypes for efficiently and accurately early evaluating the performance of a system. The different SLDLs often use inaccurate or slow techniques [138, 28] for inferring the performance of the software components that cause an undesired slowdown for the complete process.

Simulators represent a possible solution for tackling this limitation and consequently for enabling faster virtual prototype evaluations [20]. Multiple host-based simulations can be executed for evaluating both the functional and non-functional aspects of a target system directly on the development environment. The simulation objectives consist of two conflicting metrics that are simulation speed and simulation accuracy. Traditional cycle-accurate and gate-level simulators ensure precise simulation results by in-depth modeling the target system. Unfortunately, the complexity of the modern hardware architectures requires simulating a massive number of events implying a drastic reduction of the simulation speed capabilities.

Complex systems can be efficiently simulated only at a higher-level of abstraction. Even the commonly utilized instruction set simulators (ISS) does not reach a sufficient level of abstraction for avoiding the introduction of undesired simulation slowdown when integrated in system-level simulators.

An alternative and more suitable solution for producing the necessary performance estimations consists in host-based simulation techniques. This kind of simulation overcomes the slow speed capabilities of the traditional approaches by relying on the higher performance achievable by executing a host-compiled version of the program on a fast development machine. Therefore, these simulators execute a software binary version that differs from the one compiled for the target. This property ensures a substantial simulation speedup. The simulation can rely on one of the multiple software representations, such as the source code, its compiler intermediate representation or the host binary simulation code. Every representation shows a different level of abstraction. However, accurate results can be achieved only by relying on an accurate mapping technique that matches the chosen simulation code representation to the target binary machine code. This is a hard task because their structures may substantially differ due to the different architecture implementations and aggressive optimizations that cause the compiler to produce highly different executables.

Due to the complexity of the problem, all the existing host-based simulation techniques show some drawbacks. The simulators commonly suffer from slow simulation speed, inaccuracy or scarce retargetability. With the ever evolving architectures of MPSoC platforms and their complexity, it is required a new simulation methodology that allows to fast evaluate the performance of a system, since the early design stage. This methodology has to rely on a sufficient level of abstraction for supporting the fast evaluation of different implementations defined by the design space exploration activities. At the same time, the chosen level of abstraction has to be appropriate for producing accurate and significant results. It is desirable for such methodology to show an adequate level of automation and to support industrial development environments, such as model driven development.

1.2 Contributions

In this thesis, it is proposed a novel simulation methodology designed for the fast and accurate evaluation of embedded systems. The methodology addresses multiple open problems common to the host-based simulation approaches. In particular, the specific contributions proposed in the manuscript are as follows:

- *Fully automatic LLVM IR to binary code matching* - A novel approach, based on a two-phases algorithm, is presented for accurately matching the structure of the intermediate representation of a program to its corresponding target-compiled binary executable. The approach is fully automatic and it even copes with aggressive compiler optimizations without requiring any modification to the LLVM compiler or the need of expert supervision. This approach succeeds in generating accurate mappings even when the compilation of the source code applies aggressive compiler optimizations that can substantially change the structure of a program. Timing simulations based on replaying the measured traces for a set of given highly-optimized programs show an average estimation error that is below 2%.
- *Context-sensitive LLVM IR timing simulation* - A new simulation methodology is proposed for fast and accurately evaluating the performance of an embedded system. The methodology defines a new workflow for executing context-sensitive timing simulations based on the intermediate representation of the given source code. The simulation

methodology ensures to accurately estimate (average estimation error below 2%) the performance of a given target system while showing elevated simulation speed capabilities (average simulation speed slightly below 74 MIPS). The high-level of abstraction lent by the program intermediate representations allows the possibility of evaluating in parallel multiple configurations of a system in only one simulation. This property ensures a significant beneficial speedup for the simulation speed capabilities (the simulation speed largely exceeded the value of 1,000 MIPS by evaluating four different configurations in parallel). The possibility of simulating multiple target configurations in parallel enables the additional definition of a technique for producing early estimations about the execution of a heterogeneous system on an MPSoC platform.

- *Timing-aware Simulink model co-simulation* - The elevated simulation speed ensured by the presented simulation methodology enables the definition of an additional simulation technique useful for analyzing complex systems that are designed in a model driven way via MATLAB Simulink. The proposed co-simulation methodology enables overcoming the Simulink simulation limitations due to the unrealistic lack of architecture-dependent timing considerations. The enhancement of a model with dedicated components permits the co-simulation between Simulink and the LLVM IR context-sensitive timing simulator. As a result, the co-simulation of a model in the Simulink environment enables the system designers to natively evaluate the performance of a system considering the timing effects due to the execution of the software on a given target processor.

1.3 Thesis Organization

The rest of this thesis is organized as follows:

- *Chapter 2* - The chapter introduces the scope of this thesis and it presents the concept of an ideal simulation technique for fast and accurately estimating the performance of an embedded system in support of the design space exploration activities.
- *Chapter 3* - The chapter initially describes multiple background aspects to consider for timing analyzing an embedded system and consequently it presents the related works on performance estimations produced via simulation.
- *Chapter 4* - After describing the problem definition and its related work, the chapter introduces the proposed two-phases algorithm for mapping LLVM IR to binary control flow structures.
- *Chapter 5* - The chapter introduces the proposed LLVM IR context-sensitive simulation methodology and its extension for enabling the timing-aware co-simulation of Simulink models.
- *Chapter 6* - The chapter presents the overall results observed during the conducted evaluation for estimating the accuracy and performance of the complete simulation methodology, including the Simulink co-simulation.
- *Chapter 7* - This final chapter concludes the thesis and it discusses the possible future research directions.

1.4 Related Publications

The content and the concepts presented in this thesis are based on the following publications of the author:

- Alessandro Cornaglia, Alexander Viehl, Oliver Bringmann, and Wolfgang Rosenstiel. “SIMULTime: Context-Sensitive Timing Simulation on Intermediate Code Representation for Rapid Platform Explorations". In: *Proceedings of the 24th Asia and South Pacific Design Automation Conference*. 2019.
- Alessandro Cornaglia, Md. Shakib Hasan, Alexander Viehl, Oliver Bringmann, and Wolfgang Rosenstiel. “MODELTime: Fully Automated Timing Exploration of Simulink Models for Embedded Processors”. In: *AmE 2019-Automotive meets Electronics; 10th GMM-Symposium, VDE*. 2019.
- Alessandro Cornaglia, Md. Shakib Hasan, Alexander Viehl, Oliver Bringmann, and Wolfgang Rosenstiel. “JIT-Based Context-Sensitive Timing Simulation for Efficient Platform Exploration”. In: *Proceedings of the 25th Asia and South Pacific Design Automation Conference*. IEEE, 2020.
- Alessandro Cornaglia, Alexander Viehl, and Oliver Bringmann. “Accurate LLVM IR to Binary CFGs Mapping for Simulation of Optimized Embedded Software”. In: *Embedded Computer Systems: Architectures, Modeling, and Simulation: 21st International Conference, SAMOS 2021*.

CHAPTER 2

Performance Exploration of HW/SW Co-Design

This chapter is intended to introduce the challenges that commonly arise during the design exploration of partitioning mapping decisions of heterogeneous systems. An initial overview of the design exploration process and its objectives supports the importance of the definition of a methodology for assessing the performance estimations of a design iteration. The chapter ends presenting the base ideas of the host-based simulation methodology that is discussed later in the thesis and that allows producing fast and accurate timing estimations of a target system.

2.1 Design Challenges in MPSoC

The shift to the adoption of MPSoC platforms poses many challenges for the system designers [106]. One of the main challenges for the system designers consists in effectively mapping the software functional units to the processors available in an MPSoC platform. The designers' goal is to define a mapping that ensures a high-performance implementation in consideration of the different kinds of hardware resources. The mapping definition is unfortunately not trivial or immediate. In fact, the allocation decision has to consider both the software and hardware properties in regards of the performance requirements. A given processor in an MPSoC platform can result to be more efficient in executing a specific software functional unit compared to the others available on the same chip. This mismatch is also known as implementation gap [52]. An investigation is necessary for defining a mapping that enables to benefit from the heterogeneous capabilities of the different types of cores in an MPSoC. Typically, several design iterations are spent before obtaining a sufficient implementation that meets the performance requirements.

In order to reduce the number of design iterations required for identifying a valid mapping, a systematic approach is desirable for its definition. An appropriate approach should produce accurate performance estimations for the given mappings by considering the execution of a software module in a specific hardware architecture. As a consequence, a systematic approach for supporting the design activities should provide three essential capabilities:

1. Generation of an analyzable model of the software functional units,
2. Generation of a hardware model that captures the performance features of all the heterogeneous cores in an MPSoC platform and,
3. Easy exploration of the performance estimations.

The main focus of this thesis is on the challenges due to the realization of these three essential capabilities. In the thesis is discussed a methodology for the automatic evaluation of the performance of a software functional unit executed on a given processor included in an MPSoC platform. The consequent definition of a way for identifying the ideal mapping between the software and hardware resources instead is out of the scope of this thesis.

2.1.1 Programming Models

One of the main challenges in MPSoC design consists in mapping the application software, composed of multiple functional units, into the effective hardware implementation. The different available MPSoC platforms can be classified in two major classes that are symmetric multi-processing (SMP) and asymmetric multi-processing (AMP).

The SMP platforms include identical processors that share a common view of the complete system. The SMP platforms, commonly called multi-core platforms, are generally designed for reducing the power consumption of a system. At the same time, the SMP suffers from concurrency problems that are out of the scope of this thesis.

Differently from the SMP, the AMP platforms include loosely coupled heterogeneous processors (which may implement different ISAs) equipped with dedicated local memory resources. Every processor in an AMP platform is designed for efficiently solving a specific kind of task. The different processors in an AMP platform communicate with one of the large available variety of shared mechanisms (e.g. shared memory, hardware FIFOs, signaling, etc.).

The choice of adopting an AMP platform implies the hard decision of mapping the mix of software applications to the different heterogeneous processing units. The correct mapping choice can be hard to identify becoming an issue for the system designers during the design space exploration activities [105].

Synchronous Reactive AMP Systems

One of the most common and powerful paradigms in the heterogeneous systems domain is the synchronous paradigm [33]. This paradigm relies on a periodic clock that synchronizes all the connected system components. The kind of synchronization is also known as single-driver rule and the global system behaves like a finite-state machine. Channels connect the different components and the transmitted information cannot be buffered. At every tick, each block checks the available inputs and consequently starts computing the eventual outputs. The value of eventual inputs and outputs in the channels do not vary while a block is computing.

One of the main benefits of synchronous reactive systems is that the software components can be hierarchically designed. In fact, those components that have to be executed on the same processor can be grouped in single software function units. This allows the separate compilation of the different software subsystems. Another important aspect of these kinds of systems is that the performance behavior of every functional unit can be analyzed separately. The analysis has to consider only the internal state of the designated processor due to the software execution history.

The content of this thesis is primarily focused on heterogeneous systems based on MPSoC platforms and designed in respect of the synchronous paradigm. From here on, only synchronous heterogeneous system designs are considered.

2.1.2 Performance Analysis

The design space exploration (DSE) phases for the deployment of an MPSoC-based system commonly rely on the well-known Y-chart approach [80, 7, 78, 51]. This approach allows performing a quantitative analysis of architectural designs, in which architectural design decisions can be exercised. As shown in Figure 2.1, its application begins with two distinct activities. These activities are intended to provide the specification of the software application and the individuation of a suitable MPSoC platform. The software specification and a model of the selected MPSoC are the inputs for a subsequent activity focused on generating a mapping that binds the software functional units to the architectural resources. The mapping goal consists in maximizing the application performance by effectively distributing the software functional units between the different processing units. Given a mapping, an analysis tool is necessary for assessing the performance of the system's configuration. The performance results are essential for eventual further mapping iterations. In every iteration, depending on the performance results, the designer can decide to modify the software application, the MPSoC configuration or the mapping until a satisfactory design is found.

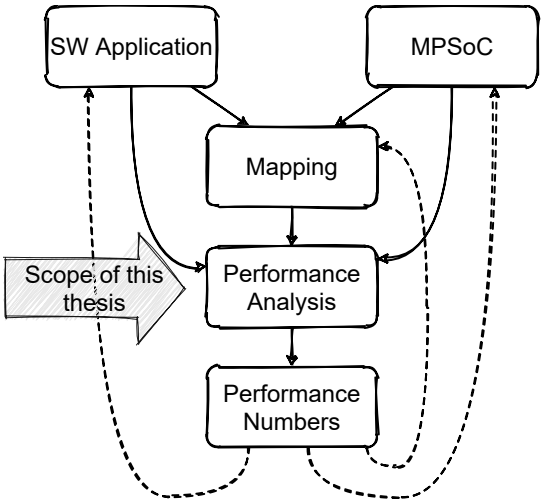


Figure 2.1 – *The Y-chart approach adopted in design space exploration for deploying MPSoC applications: Multiple iterations are performed for exploring different hardware/software configurations and determining the most suitable mapping for the execution of the software functional units of a program to the available heterogeneous processors included in an MPSoC platform (approach initially defined in [80]. Accurate performance estimations represent the feedback that drives the iterations of this exploration.*

Requirements and Objectives

The Y-chart model approach motivates the necessity and the importance of a tool that allows assessing the performance of a system in a cycle-approximate way. A simulator is commonly used for profiling the exploration activities [79]. Usually, the simulator operates on a high level of abstraction, as shown in Figure 2.2. A simulator allows the designer to easily modify the configuration of an MPSoC or to evaluate a different hardware architecture. Furthermore, a higher abstraction level helps in making the evaluation faster. The simulation objectives can be either at the MPSoC level or at a single processing unit. Efficiency is consequently one of the key requirements for a simulator. In fact, the faster an iteration can be performed the

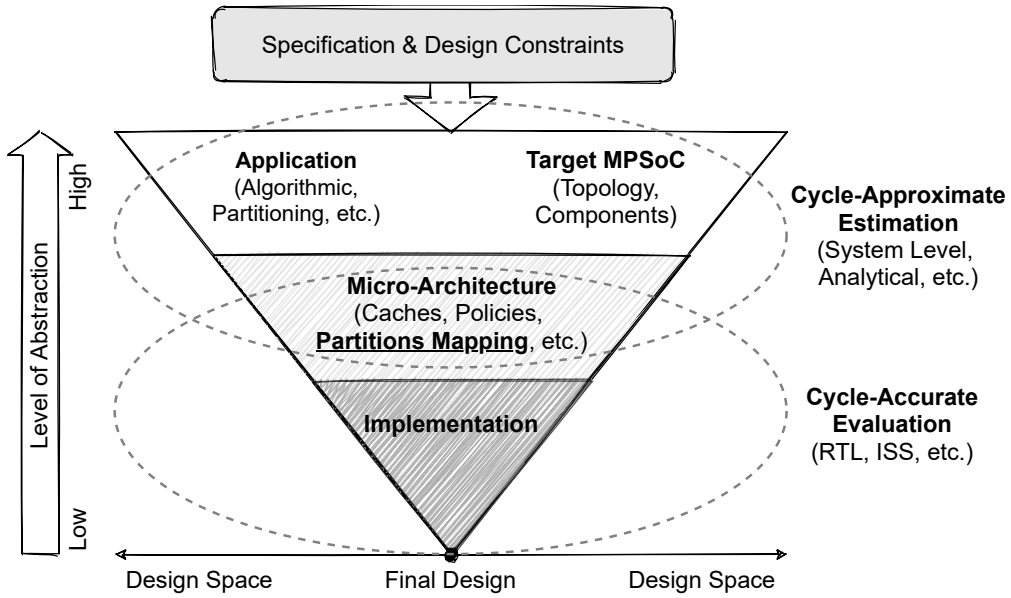


Figure 2.2 – *Design space exploration phases, abstractions and simulation objectives (figure edited from [70]): Given an application composed of software functional units and a target MPSoC platform, a simulator working at an appropriate level of abstraction is desirable for evaluating the performance behavior of every single functional unit by considering their execution in one of the configured cores and producing sufficiently accurate timing estimations.*

more explorations can be evaluated. However, the simulation results have to be sufficiently accurate for appropriately driving the design decisions. Furthermore, the post-simulation visualization of the results is essential for their consequent analysis.

Ideal Solution

Accuracy and efficiency are essential requirements for any tool designed for producing performance estimations in support of the design activities. An ideal solution should provide cycle-accurate results in a very fast way. For example, in the software-in-the-loop simulation scenario [175], it is expected a simulator to simulate the system faster than the amount of time necessary for executing the program on the real target platform. Unfortunately, these two objectives are commonly in contrast. As described in Figure 2.3, several existing approaches show that a trade-off between accuracy and simulation speed is inevitable. For example, register transfer level (RTL) simulations achieve cycle-accurate results by considering in depth information about the target architecture that causes a substantial slowdown. The slowdown due to complex architectures makes this kind of simulations unfeasible for predicting the performance of real-word applications. Instruction set simulators (ISS) instead can speed up the simulation performance by slightly sacrificing the estimation accuracy. Still, the ISSs are not suitable tools for evaluating a real-word application and supporting the design exploration.

Analytical models represent a faster alternative to classical performance estimation approaches by ensuring several order of magnitude speedup. The speedup is ensured by modeling the dynamic behavior of complex hardware mechanisms by static formulas that describe their average behavior. Unfortunately, the problem simplification introduces a significant inaccuracy on the performance estimations. More accurate results can be achieved by high-level simulation techniques like the host-based approaches. Even for these kinds of approaches, the accuracy's improvement requires the consideration of more detailed information about the target architecture, which causes undesired slowdown. Nevertheless, compared to the classic

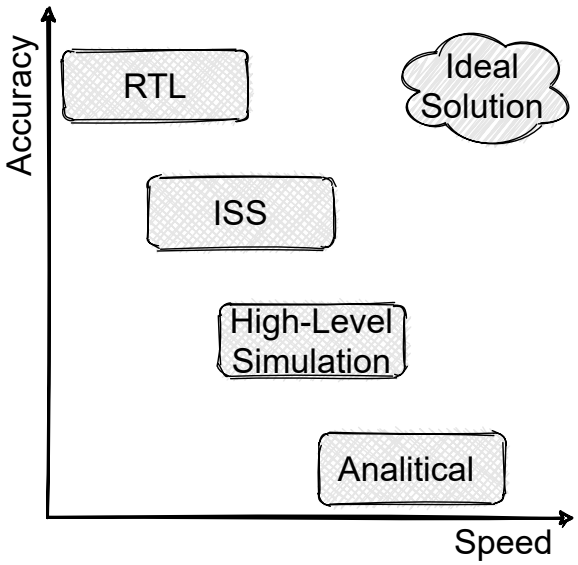


Figure 2.3 – *Concept of ideal solution for performance analysis: It is desirable to produce accurate performance estimations by executing fast simulations. Unfortunately, simulation accuracy and simulation speed are two contrasting quantitative metrics that require the adoption of an inevitable trade-off.*

simulation approaches, both host-based simulations and analytical models can support the analysis of a full and complex system.

In conclusion, an ideal approach should be able to provide at the same time an elevated level of accuracy and fast simulation speed. These two simulation objectives are the focus of the methodology that is described in this thesis. In fact, in the following sections, a novel host-based technique is described for producing fast and accurate performance estimations of embedded systems in support of the design decisions.

2.2 Performance Estimation via Host-Based Simulation

The scope of this thesis is to present a host-based simulation methodology that supports the design exploration activities by producing fast and accurate timing estimations for the execution time of a program considering its execution on a given configuration of a target platform. The estimations have to consider a specific configuration of a given processor (e.g. cache memories enabled or disabled) as well as the given input data that determines the program's execution path. In order to successfully fulfill its objectives, and as shown in Figure 2.4, the methodology is subdivided in two distinct phases that are:

1. *Timing analysis phase* - The goal of this initial phase is the generation of a timing model that describes the performance behavior of a given pair of hardware and software configurations. A hardware configuration is composed of the definition of a given target platform and of a description about the configuration of the hardware resources included in it. Differently, a software configuration is composed of the source code of a program (programmed in a high-level programming language like C or C++) and a set of compiler optimizations to apply at compile-time for producing an efficient target executable. The timing model is the result of a set of performance extraction activities focused on describing the dynamic performance behavior of the software program in consideration of a given input dataset and the hardware/software configuration.

2. *Evaluation phase* - This second phase is responsible for generating the necessary performance estimations requested by the design activities. A specific host-based simulator is responsible for producing these estimations of the target system. Multiple estimations can be quickly generated by executing the simulator with different program's input data. During its execution, the simulator updates the simulation results by considering the information contained in the previously generated timing model.

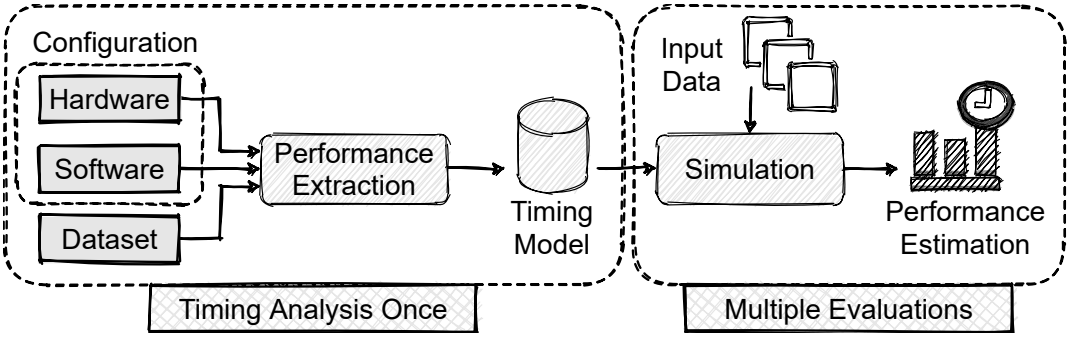


Figure 2.4 – *Proposed concept for the generation of performance estimations: Given a hardware and software configuration, a timing model is generated only once. The timing model is consequently utilized while executing multiple simulations for producing one performance estimation for every different given input data.*

One of the key aspects of the presented methodology is that, for a specific hardware/software configuration, multiple performance estimations can be assessed by varying the program's input data while executing multiple fast simulations that rely on only one timing model. The generation of a valid timing model is essential for producing accurate estimations. During its generation, it is important to ideally consider all its possible different timing behaviors. At the same time, a timing model is fixed for a specific configuration. A new timing model has to be generated every time that a different configuration has to be explored.

2.2.1 Workflow Concept

The proposed two-phases methodology satisfies the requirements of the previously described and commonly adopted Y-chart approach. The more detailed workflow represented in Figure 2.5 shows how the methodology can provide essential support for the design decisions.

Every time a new configuration has to be evaluated, the design exploration process defines the parameters that categorize both the hardware and the software components. The hardware description consists of a target configuration that describes the available hardware resources in the target platform and their configuration. The software component identifies a specific version of the source program and the selected compiler optimizations to apply during the generation of the target executable.

The timing analysis phase starts with the compilation of the source code. This phase has to be performed only once per hardware/software configuration. During the compilation process, the given input program is optimized by applying all the compiler optimizations specified by the software configuration. The compilation process produces two binaries as output. One binary is the result of the cross-compilation for the target architecture and the other one is the binary to be simulated. Depending on the simulator implementation, the two binaries may coincide. The timing analysis phase continues by generating the necessary timing model for the requested configuration. The timing model can be generated by measuring and analyzing the dynamic behavior of the cross-compiled executable executed on the target platform. An

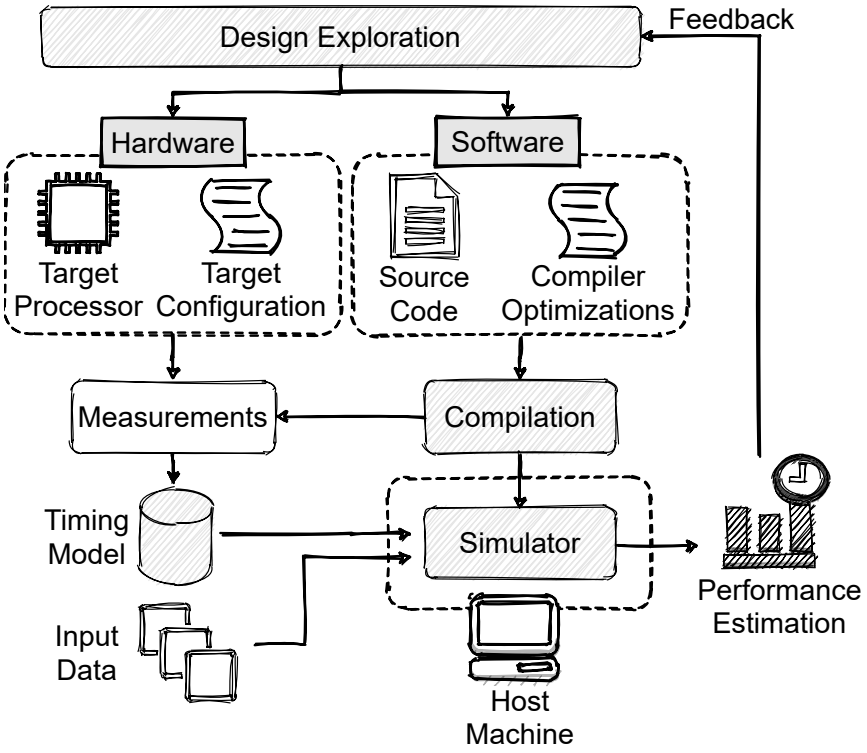


Figure 2.5 – *High-level host-based simulation concept workflow: The flow starts with the definition of both the hardware and software configurations that have to be explored. After the compilation of the program, measurements are extracted directly from the target platform for generating a timing model. The timing model is consequently utilized by a host-based simulator for producing the system's performance estimation in consideration of the specific given input data.*

extensive analysis can be conducted by considering an appropriate input data set that induces the measurements to observe the possible different program's behaviors.

Finally, the performance of the configured target system can be generated by running a simulator in a fast host machine (e.g. a development machine). The simulator requires in input an executable, a timing model and specific input data. Multiple performance estimations can be produced by simulating the target system and varying the input data. The performance estimations produced by the simulator represent the requested feedback for the design iteration.

2.2.2 Possible MPSoC-Oriented Improvements

The workflow previously described in Section 2.2.1 can be extended for providing better support for the design exploration activities focused on the generation of the software partition mapping for MPSoC-based systems. In fact, the simulation idea presented in the workflow in Figure 2.5 can be adapted for defining the concept shown in Figure 2.6. The adaption requires the simulator to expect in input multiple timing models. As a consequence, multiple timing models can be considered while simulating.

The possibility of considering multiple timing models in parallel requires a simulation strategy based on a shared representation of the software program between the configurations.

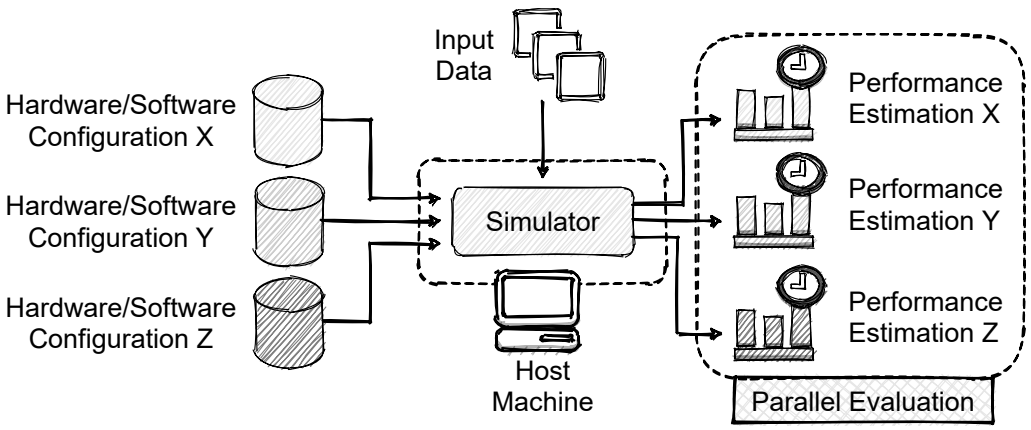


Figure 2.6 – *Parallel evaluation of multiple hardware configurations in only one simulation: Multiple hardware configurations can be analyzed in parallel if the simulator executes a software representation that is shared between multiple hardware architectures and by considering different timing models.*

The consideration of multiple timing models opens two different simulation scenarios:

1. *Parallel evaluation of multiple configurations* - Simulating the same program and taking into consideration different timing models at the same time can allow the parallel evaluation of different hardware/software configurations, like suggested in Figure 2.6. This scenario can offer a valid support for the exploration activities focused on identifying the most suitable processor, in a limited set of processing units, for executing a software functional unit (not necessarily in the context of MPSoC design exploration).
2. *Early evaluation of synchronous heterogeneous systems* - Differently from the previous scenario, the possibility of considering multiple timing models but accessing them one per time depending on a fixed scheme (like the functional software units in a synchronous heterogeneous systems) during a simulation represents the opportunity for producing early performance estimations of a complete heterogeneous system.

2.3 Summary

This chapter presented the base concept of an ideal simulation solution for supporting the performance assessment activities of the design space exploration of complex embedded systems. The goal of an ideal simulator is to provide extremely accurate results in a very fast way. Unfortunately, the simulation speed and the simulation accuracy are two contrasting metrics. This limitation requires a solution to determine an adequate trade-off. For this reason, in the second part of the chapter it has been proposed a high-level workflow for the implementation of an efficient timing simulator. The workflow requires executing a timing analysis phase only once. Thereafter, multiple performance estimations can be produced by fast simulating the system and considering different input data sets. The workflow supports the parallel evaluation of multiple configurations for ensuring a significant speedup in the overall performance analysis stage.

CHAPTER 3

Background and Related Work

The initial scope of this chapter is to provide a set of concepts that are at the base of this thesis and consequently to present the most relevant related work. The chapter starts by giving a general brief introduction about the embedded systems design and their implementation. The implementation part focuses on the model-driven realization of embedded software via MATLAB Simulink. Thereafter, some essential tools, components and concepts from the LLVM Compiler Infrastructure are introduced. Follows a short overview about timing analysis topics and related basic fundamentals. Finally, the chapter ends with a detailed overview of the state of the art in performance estimation of embedded systems with a special focus on approaches that produce timing estimations via simulation.

3.1 Embedded Systems

According to the definition given in [59], an embedded system is *"an engineering artifact involving computation that is subject to physical constraints"*. Like other computing systems, an embedded system is composed of software and hardware components that have to interact with a specific environment. Furthermore, every embedded system interacts with both the physical word and the target platform implementation. These interactions are ruled by constraints that drive the behavior of a system and that are commonly hidden to the final user. For example, some of the implementation requirements define specific bounds for parameters like processor execution frequency, power and others. Therefore, when developing an embedded system, the definition of its design requires the consideration of multiple challenging tasks. The tasks involve both the software and the hardware components. In fact, it is common for modern embedded systems to include a large amount of software that implements complex algorithms. At the same time, both the functional and non-functional properties of a system are influenced by the hardware architecture implemented by the selected target platform where the software is executed. Therefore, both the two components have to be taken into account in the design process for satisfying the imposed requirements and constraints.

3.1.1 System Design

In general, the systems design is a process whose main purpose is to define and generate a model that satisfies a given set of requirements. The process can be performed with different levels of automation and it can be applied to hardware and software components. The process's output model consists in an abstract representation of a target system. Common objectives of systems design are the realization of software programs that can be compiled as well as hardware descriptions that can be utilized for synthesizing a new circuit.

In order to meet a set of given requirements, the process requires the execution of a certain number of refinement iterations. A simplified representation of this design flow is shown in Figure 2.1. More specifically, the goal of the embedded systems design is to define an efficient solution that meets a given set of hardware and software requirements that describe the operation modes of a system that is subject to the environment's physical constraints. Differently from other domains, these kinds of constraints impose the designers to consider together the different hardware and software components.

The multiple iterations required by the system design activities have the goal of increasingly optimizing and refining the finial system performance. More and more precise models are considered after every iteration in the process. These models are essential for early evaluating the design in respect of the given requirements. Relying on the results of these evaluations, the process determines the implementation of an evaluation version of the system. This version is consequently utilized for a more accurate evaluation and analysis of the actual implementation.

As previously declared, the design steps and the consequent decisions required for determining a system's version to be analyzed are out of the scope of this thesis. However, the work presented in this thesis is focused in defining a simulation methodology that enables the early evaluation of the performance of an embedded system. For this reason, this chapter introduces the base concepts and components useful for its definition. In particular, the chapter focuses on introducing previously proposed approaches for generating performance estimations for the execution time of embedded systems. This requires presenting a technique that is commonly utilized in the industrial setting for producing the source code of a system. Thereafter, an overview is given about the standard compilation process for generating an executable binary from an input source code.

3.1.2 MATLAB Simulink

Model-based development is common practice in the design of modern and complex embedded systems [39]. The design of intelligent systems requires implementing more and more complex functionalities and logics. Traditional design methodologies consist in deriving a written text specification that describes the algorithms that the developers have to manually implement. However, the growing complexity of the software requires to limit as much as possible the hand coding activities of the programmers for ensuring a significant speedup of the time to market by simplifying the validation process. In fact, model-based development tools allow reducing the implementation problems by supporting the systematic transformation of high-level problem descriptions to software implementations. In model-based development, models at different levels of abstractions can describe the software of a complex system. The models are consequently automatically translated to programs that can be compiled for the target platforms. The different model's components allow the reutilization of well-structured and trusted code.

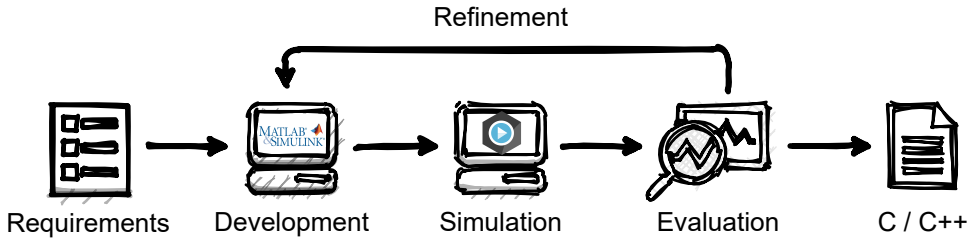


Figure 3.1 – *Standard workflow for model-based development of software in MATLAB Simulink: A system can be realized from the given requirements on a development machine via a model-based environment. On the same machine, multiple functional simulations can be conducted for evaluating the system at any phase of the development. Depending on the simulation results, multiple iterations can be performed before to automatically generate the final program source code.*

In this context, MATLAB Simulink [166] is a widely utilized tool, especially in the automotive domain [42]. Simulink is an extension of MATLAB that offers a graphical modeling environment for easy development of complex control applications abstracting from the target platform. One of the most attractive Simulink peculiarities is that it supports the verification and validation of a system since the early design stages and before that the hardware is available [46]. In fact, in addition to the development capabilities, Simulink offers the possibility of evaluating and simulating complex control applications during the complete development process. Early feedback can be obtained since the early design activities via native simulations.

The standard activities flow for software development in Simulink are shown in Figure 3.1. The process starts with the implementation of a model from specifications or requirements. A model is composed of multiple nested and interconnected components that define the logic of an application. In addition to the standard components, Simulink offers a wide set of specialized toolboxes designed for solving specific tasks. At any of the development stages, the system under development can be evaluated by simulating it. Simulink provides a user-friendly graphic interface for visualizing and evaluating the events produced while simulating. After the evaluation, the process can be reiterated multiple times for further developing. A Simulink model cannot be directly executed on a target platform, it requires first to be translated to a source program and then to be compiled. The code generation, resulting from the translation of the model to a high-level programming language concludes the process. Multiple and different coders can be utilized for solving this task. Between them, the Embedded Coder toolbox [164] allows producing C and C++ code in a highly configurable way. The code appearance can be specified by modifying its target language compiler (TLC), modifiable also via a simple graphic interface. In contrast to other coders, the Embedded Coder produces source code that is ready to be compiled for being executed on a target platform and excluding the superfluous simulation code.

Simulation Limitations

The possibility of simulating the functional behavior of the system under development is one of the most attractive features of Simulink. A system is generally modeled in interconnected components and the interconnections determine their execution order. The components activation is driven by the synchronous reactive paradigm (SRP) [33]. The output generated from a component is given in input to all the connected components. The activation order is fully deterministic. A component can be activated only if the expected input is ready. If the output of a component is connected with more than one consequent component, the execution order of these is statically determined with a priority-based scheduler.

A model is commonly composed of interconnected software and hardware components as shown in Figure3.2. The first ones define the behavior of a controller. The controller (e.g. the software executed by the ECU of a car) has to take actions depending on the input values generated from the plant (e.g. mix of sensors that sample the real word). The plant is formed of only hardware components. These two component types require to be simulated with different time representations. The nature of the plant components forces them to be simulated in continuous time. The controller is updated in discrete time instead.

Unfortunately, the Simulink models are purely functional. The simulation overlooks the timing effects due to the execution of the controller application on a target platform. In fact, the internal computation of every block in the plant component is performed in zero time and the outputs are provided to the connected inputs instantly. The blocks in the controller update their output with a fixed delay (logical time) expressed in simulation steps. This simulation paradigm is not realistic because the execution time of the software depends on both the target platform where it is executed and on the input data that determines the visited control flow path. The Simulink simulations lack in timing considerations about the execution time of the software on a specific target platform.

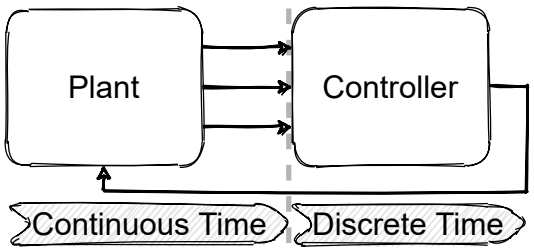


Figure 3.2 – *Simulink system components and simulation time representations concept: A Simulink model is commonly structured in hierarchically interconnected components that belong to one of two possible subsystems that interact with external sensors. The plant is the subsystem that implements the physical world and it is sampled by sensors. The controller is the subsystem that implements the software part of a system and it is designed to process the data that is generated by the input sensors. The plant, for its nature, is simulated in continuous time. Differently, the controller is iteratively activated and it is simulated in discrete time.*

3.1.3 Compilation Process

Programs are commonly coded in a high-level programming language that ensures an adequate level of abstraction from the target architecture. This software representation cannot be directly executed on a target platform. A compiler can be used for translating a given program into semantically equivalent executable machine code. The steps executed during the compilation process are shown in Figure 3.3. A compiler is designated for translating the source code to relocatable machine code. An assembler is consequently required for generating the target assembly code from the machine code. Finally, the designated linker produces the expected target machine code by linking together the target assembly with eventual external libraries and other relocatable object files. The complete compilation process is called cross-compilation if the program is compiled on a different host architecture.

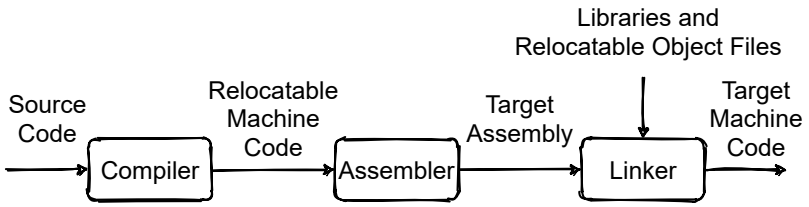


Figure 3.3 – *Standard compilation stages and outputs (edited from [3]): The common process for compiling the given source code of a program requires the execution of three consequent stages. A compiler initially produces the relocatable machine code for the given source code. Consequently, an assembler generates the target assembly code that is finally linked by the linker with eventual libraries for obtaining the requested target machine code.*

On request, a compiler can optimize the structure of a program. Two common optimization objectives in the embedded systems domain are: code size reduction and improvement of the instruction level parallelism (ILP). A compiler achieves the second objective optimizing the code with considerations on the available hardware resources in the target architecture. Improving the ILP requires the modification of a program's structure for ensuring an efficient utilization of the hardware resources. In general, the compilation activities change the original structure of a program.

The preponderance of the different available compilers shares a common structure composed of consequent phases [3]. As shown in Figure 3.4, the compilation phases can be grouped in three main categories:

1. *Analysis* - Phases that are responsible for generating an intermediate representation (IR) which is expected from the synthesis stages. The IR generation requires parsing and verifying the correctness of the given input program.
2. *Machine-independent optimization* - Optional phases focused on optimizing the IR structure for allowing the consequent phases to generate a better executable.
3. *Synthesis* - Code generation phases that translate the IR into target specific executable code.

These three phases are commonly called: front end, middle end and back end. Differently from the front end and middle end, that are always present in a compiler, the middle end is optional.

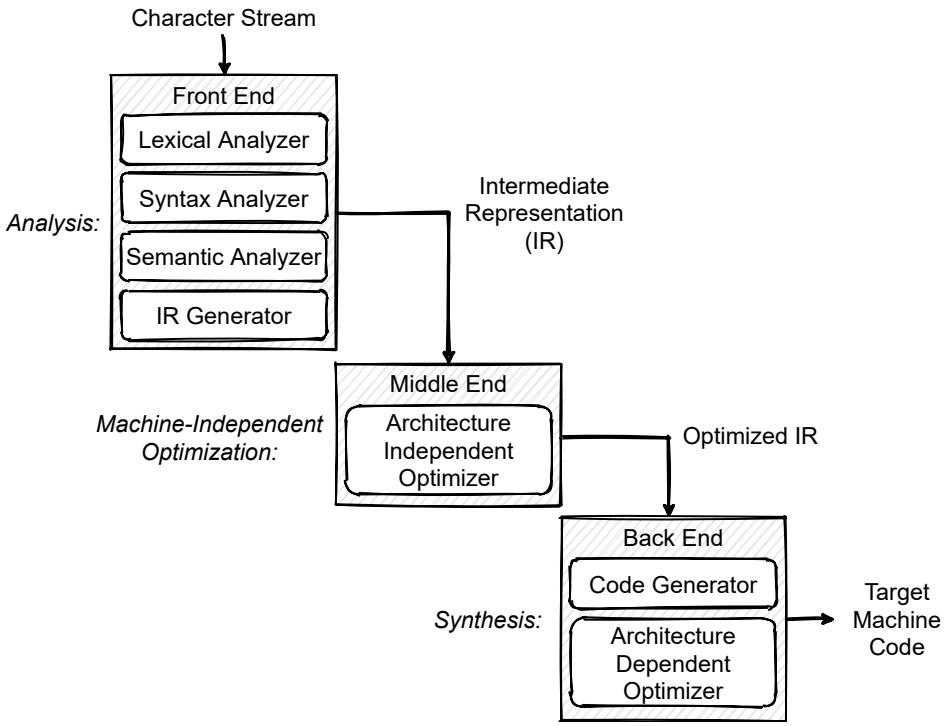


Figure 3.4 – *Detailed workflow of the standard compilation phases (edited from [3]): A standard three-stage compiler produces the requested target binary executable by initially translating the given source code to an intermediate representation and optimizing it. Commonly, the front end is responsible for the translation, the middle end performs the architecture-independent optimizations and the back-end concludes the compilation by applying the architecture-dependent optimizations and producing the executable.*

3.2 The LLVM Compiler Infrastructure

The LLVM Compiler Infrastructure [87] provides a wide set of tools for analysis and both static and dynamic compilation of programs. The compiler is open source, it is supported by a conspicuous number of contributors and it is widely utilized across academic, research and industrial projects. It is organized in a modern object-oriented and extendable framework of libraries that supports many different target architectures.

3.2.1 Static Compilation

The LLVM project implements a standard three-stage compiler. Consequent invocations to the available tools allow an efficient compilation of a given program as shown in Figure 3.5. The front-end, called `clang`, performs all the necessary steps for translating the given source code (originally only supported C and C++ programs) into an intermediate representation (IR). The LLVM IR is also called bitcode and it is internal to the compiler. The bitcode consists in an architecture independent instruction set (similar to assembly) structured in modules and its instructions respect the single static assignment (SSA) form [139]. The compiler bases most of the optimizations on this software representation. The optimizations are applied by executing the necessary middle-end passes. The middle-end, or optimizer, is called `opt` and it can be chained for applying consecutive optimizations at different levels [158]. Multiple bitcode modules can be linked together with the `llvm-link` tool. The compilation, or cross-compilation, terminates with the architecture-dependent optimizations performed by `llc`, the back-end. The last action performed by `llc` consists in lowering the program representation internal to the compiler to machine binary code.

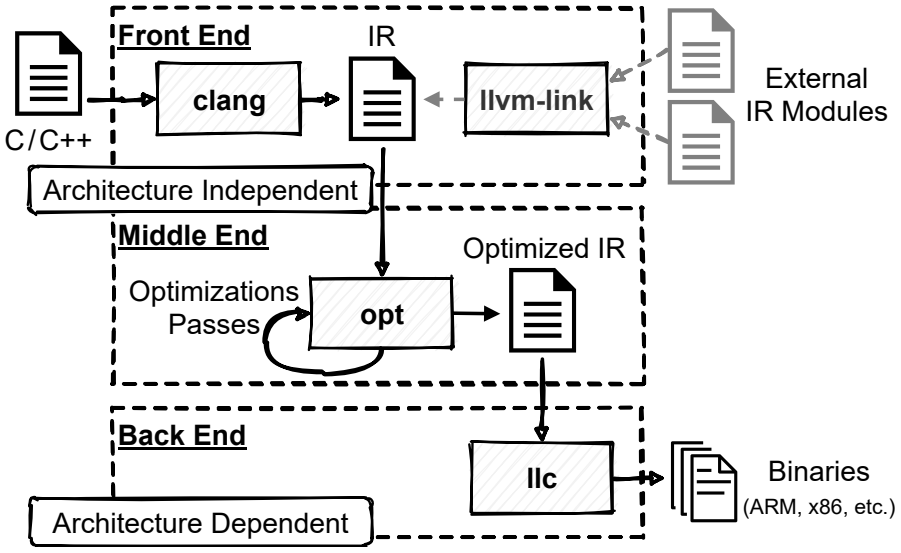


Figure 3.5 – *Static LLVM compilation tools and workflow: The LLVM Compiler Infrastructure implements a standard three-stage compiler. The compilation starts by compiling the source code with* `clang` *and producing the corresponding bitcode. The bitcode is consequently optimized via* `opt` *applying architecture-independent optimizations at the IR level. Finally,* `llc` *optimizes the IR code by applying the necessary architecture-dependent optimizations and producing the requested binary executable.*

3.2.2 LLVM IR

The bitcode represents the cornerstone of the LLVM compiler. The different tools of the framework rely on this intermediate representation for optimizing or analyzing a given input program. The bitcode can be generated via `clang`, if a program is given in input, or it can be directly hand coded by following a well-defined instruction set. An example of the bitcode generated from the simple C program listed in Listing3.1 is shown in Listing3.2.

Listing 3.1 – *A simple example of a C program.*

```
1 int globalVar = 0;
2
3 int main() {
4   const int MAX_VAL = 100;
5   int incr = 1;
6
7   while (globalVar < MAX_VAL) {
8     globalVar += incr;
9   }
10  return 0;
11 }
```

Listing 3.2 – *Structure of the bitcode of a simple program.*

```
1 ; // Module
2 source_filename = "simple_program.c"
3
4 ; // Global Data
5 @globalVar = global i32 0, align 4
6
7 ; // Function
8 define i32 @main() {
9 ; // Basic blocks
10 entry:
11   br label %while.cond ; // Terminator
```

```
12 while.cond:
13   ; // Instructions
14   %tmp = load i32, i32* @globalVar, align 4
15   %cmp = icmp slt i32 %tmp, 100
16   br i1 %cmp, label %while.body, label %while.end
17 while.body:
18   %tmp1 = load i32, i32* @globalVar, align 4
19   %add = add nsw i32 %tmp1, 1
20   store i32 %add, i32* @globalVar, align 4
21   br label %while.cond
22 while.end:
23   ret i32 0
24 }
25
26 ; // Architecture dependent information for llc
27 target datalayout = "..."
28 target triple = "..."
```

The LLVM IR code is structured in modules and every module is identified by a name that typically corresponds to the name of the source file. The structure of a module follows a fixed hierarchical scheme. If the program defines global variables, the global data information is placed in the upper part of the module. Every module contains at least one function for reflecting the original structure of the functions in the source code. Every function contains at least one basic block identified by a label and consisting in one or more instructions. The instructions are architecture independent but they resemble tradition assembly instructions. The basic block instructions can follow eventual PHI instructions that can appear only at the beginning. The last instruction always consists of a basic block terminator. Every module ends with some metadata containing architecture dependent information useful during the back-end activities.

Multiple passes can be executed via `opt` for optimizing or analyzing the bitcode. Additional custom passes can be easily generated in addition to the ones provided by the LLVM framework. Every pass can work on different levels (module, basic blocks or instructions) and eventually can modify the bitcode or the metadata information. This possibility makes straightforward the retargeting of the bitcode to different architectures from the one specified at the beginning of the compilation process.

3.2.3 Dynamic Compilation

In addition to the classic static compilation, the LLVM Compiler Infrastructure allows the dynamic compilation of programs. This capability is enabled by the `lli` tool [160]. In fact, this tool can execute directly the bitcode. Unfortunately, `lli` is not an emulator and it cannot execute LLVM IR code produced for a different architecture. This limitation can be tackled by retargeting the architecture dependent metadata information contained in an IR module. This can be easily achieved by implementing an appropriate modification pass for the `opt` tool.

Interpretation

The `lli` tool offers two different dynamic compilation modes. The first one is based on the interpretation of the bitcode. The interpretation is the simplest but slower mechanism of dynamic compilation. The bitcode given in input is compiled instruction per instruction, introducing a significant slowdown in the compilation process. The steps for interpreting a program are represented in the upper part of Figure 3.6. Initially, the module containing the entry function is loaded and other modules can be loaded later on during the execution, only

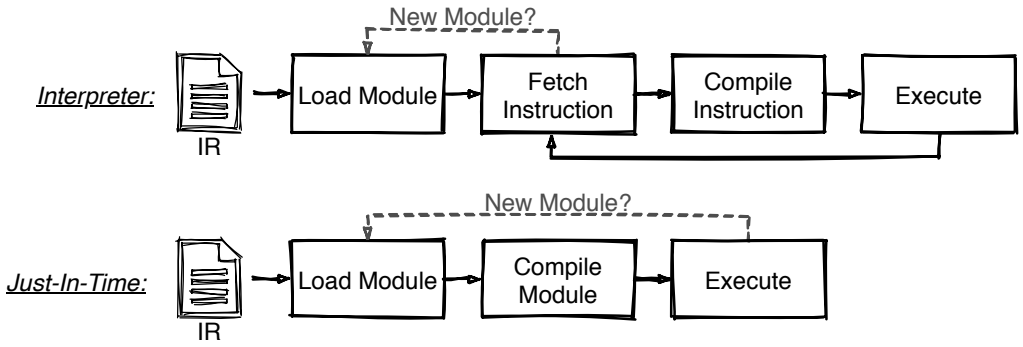


Figure 3.6 – *Interpretation and Just-In-Time execution of LLVM IR: The LLVM Compiler Infrastructure offers two different kinds of dynamic compilation. The first possibility consists of a simple but slow interpreter. Differently, the second possibility of dynamic compilation relies on a fast JIT-compilation technique.*

if requested. The interpretation performs repeatedly the following operations until the end of the program: fetch the next bitcode instruction, compile the instruction to binary code for its execution on the hosting machine, and execute the generated binary code.

The simplicity of the interpretation mechanism makes easier assessing the compilation state during the execution. Eventual extensions can be straightforward implemented. At the same time, the simplicity represents the interpretation drawback. The bitcode instructions are fetched and compiled individually. No instruction caching mechanisms are considered determining redundant compiling operations if the same instructions are executed multiple times.

Just-In-Time Compilation

The second compilation mode supported by `lli` relies on the Just-In-Time (JIT) compiler. Compared with the interpretation, the JIT compiler ensures a substantial speedup during the compilation process [18]. Higher performance can be achieved by disabling the possibility of lazy compilation. As shown in the lower part of Figure 3.6, the speedup is ensured by providing a caching mechanism for the compiled modules. Every module in a program, that has to be executed, is compiled only once and on request. A module compilation implies the compilation of all its instructions. Differently from the interpretation, every compiled instruction will be re-utilized in case of successive executions.

The JIT compilation introduces a slowdown in the program execution cost if compared with the native binary execution. However, the slowdown is minimal. The slowdown effects are minimal when executing large programs. Conversely to the interpretation, the JIT execution mode is particularly efficient when the execution of a program consists of multiple loops.

3.2.4 Other Program Intermediate Representations

The LLVM Compiler Infrastructure utilizes two further internal intermediate representations during the compilation process. These representations are architecture dependent and they support the back end activities of `llc`. The two representations are:

1. *Machine IR (MIR)* [159] - Architecture specific IR resulting from the translation of the bitcode given in input to the back end. This IR is utilized by `llc` for applying the necessary architecture-dependent optimizations.

2. *Machine Code (MC)* [161] - Code representation for an object file that is similar to the binary representation. It does not contain high-level information commonly stored in the bitcode or in the MIR. The MC code is derived from the MIR and it represents the last representation of the program before the generation of the binary code.

The structure of an MIR module resembles the structure of an IR module, as the one previously showed in Listing 3.2. In addition, it also contains a copy of the original bitcode from which it has been generated. At the same time, compared to the bitcode, the MIR structure is closer to the binary representation. In fact, the MIR includes most of the effects caused by the architecture-dependent optimizations. The final MIR module includes all the effects of the architecture-dependent optimizations applied via `opt` from `llc`, including the ones that change the program structure [158]. Therefore, an MIR module includes the effects of aggressive loop optimizations such as loop unrolling, loop-invariant code motion, loop inversion, loop fusion and others.

During the final compilation phases, for every function in the MIR module, the MIR basic blocks are sequentially translated in order into LLVM MC instructions. The structure of the MC code is very close to the structure of the target assembly that is linked for producing the target machine code (see Figure 3.3).

3.3 Timing Analysis

The execution time of a program is a quantitative non-functional property. A quantitative property of an embedded system is a property that can be measured. Other examples of quantitative properties are: power consumption, reaction time, system's weight, and others. The analysis of any quantitative property requires modeling both the software program and the hardware resources included in the target platform. Therefore, the execution time ET of a program P can be modeled by a function f that respects the following equation [89]:

$$ET_P = f(P, i, w) \tag{3.1}$$

Where i represents the program's input and w the state of the stateful hardware resources in the target platform (such as the contents of the cache memories) before the execution of the program's first instruction. Unfortunately, the definition or identification of the function f is a hard problem [178]. This problem can be solved only by using a restricted form of programming that ensures the program termination for any given input data (e.g. avoiding recursion, fixing the loop bounds, and others).

The formula in Equation 3.1 allows defining some interesting timing properties of a program. For a given program, different combinations of the values of the two parameters i and w determine the frequency distribution of execution time values represented in Figure 3.7. For some domain-specific applications, it is requested to determine the best-case or the worst-case execution time (respectively BCET and WCET). In particular, the WCET is a mandatory objective in the verification of safety critical-systems. For example, this is explicitly required by both the DO-178B/C avionic verification guidelines [61] and the ISO 26262 standard for the development of functional-safety vehicles [120]. Determining the WCET value for a program P can be formalized as follows:

$$WCET_P = f(P, i_{max}, w_{max}) \tag{3.2}$$

Unfortunately, the problem space is too large and it is not trivial to identify the program input i_{max} and the initial hardware state w_{max} that determine the WCET for the program P. Measurements can lead in determining only the maximal observed execution time (MOET).

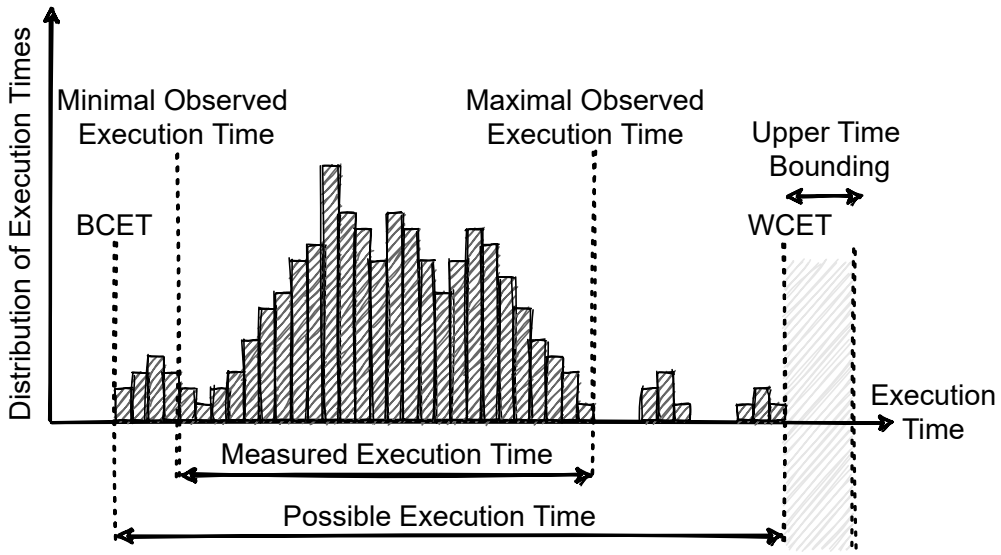


Figure 3.7 – *Interesting timing properties of a program (edited from [178]): Considering different input values and initial hardware resources states it is possible to observe different execution times for the execution of a program. Depending on the analysis goals, different bounds for the execution of a program can be assessed.*

The MOET can differ from the actual WCET. It is not feasible to measure the execution time of a program for all the possible combinations, the problem space is too large to be explored. Therefore, in the domains where the WCET is required, a tight upper bound is used instead. The scope of the WCET analysis tools consists in reducing as much as possible the difference between the produced upper bound and the actual WCET.

3.3.1 Programs as Graphs

The timing analysis activities are often applied to a convenient abstract representation of the source code. In a similar way as it is done internally by the compilers, the source code can be represented as a graph. The graph represents the control flow between different segments of code. All the different program representations that are involved in the compilation process, from the source to the binary code, can be represented as a graph.

Basic Block

A basic block is a sequence of consecutive program instructions that are always executed in order. The execution of a basic block starts from its first instruction and it ends after the execution of the last instruction. There is no possibility of halting, jumping out or terminating before that all the instructions are executed. In the same way, there is no possibility of branching inside a basic block. The last instruction of a basic block always consists of a terminator instruction that can determine a termination or a branch. When representing the program as a graph, the basic blocks are considered as the graph's nodes.

There exist some architectural specific exceptions for the given definition of basic block. For example, the full ARMv7 ISAs supports the utilization of conditional execution instructions [58]. In this specific case, the scope of conditional execution instructions consists in reducing the number of branch instructions in a binary program for improving the run-time performance of a system by reducing the requirement of a large and accurate branch predictor. However, these exceptions are out of the scope of this thesis and they will not be considered.

Edges

The complete execution of a program requires visiting multiple basic blocks. The program's execution flow accesses a specific basic block depending on the terminator instruction of the last executed basic block. Conditional and unconditional branch instructions determine the transfer of control to one of the possible successors of a basic block. In a graph, this information can be represented as directed edges between the nodes. Therefore, a direct edge from a node n_x to a node n_y can be expressed as (n_x, n_y). In this case, node n_x is the tail of the edge and node n_y is its head. The number of incoming heads to a vertex (or node) determines the vertex in-degree. In the same way, the number of outgoing tails determines the vertex out-degree.

The edges of a directed graph can be classified in four types performing a depth-first search [123]. Annotating the nodes during the search with the time of discovery t_x, it is possible to define four types of edges as represented in Figure 3.8:

1. *Tree edges*: Edges that describe the relation between a node a and one of its direct successors b. In this case, $t_a < t_b$.
2. *Back edges*: Edges that connect a node e to one of its ancestors a. Also a self-loop is considered a back edge. In this case, $t_e > t_a$.
3. *Forward edges*: Non-tree edges that connect a node c to a descendant e node. These edges lead from high to low nodes. In this case: $t_e > t_c$.
4. *Cross edges*: Edges that are not part of any of the previous types. In particular, these kinds of edges connect one node c to a node e that belongs to a different depth-first tree. In this case: $t_e < t_c$.

From now on, as already graphically represented in Figure 3.8, dashed edges in graphs will represent back edges of directed graphs.

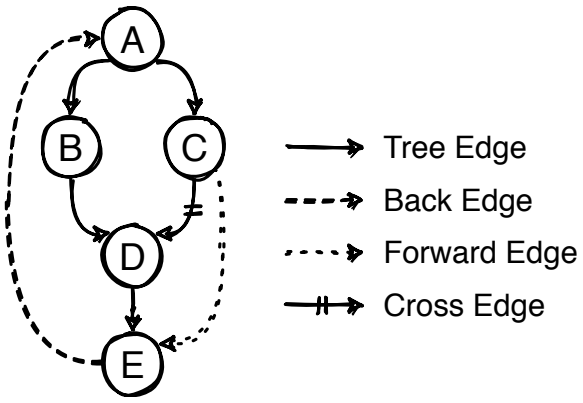


Figure 3.8 – *Classification of the edges in directed graphs: The edges of a directed graph can be of four different types. For the purposes of this thesis, the most relevant two are the tree and back edges. These edges are useful in the analysis of loops and control-flow paths.*

Paths

A sequence of edges that connects consecutive nodes identifies a path of edges. Considering a certain number of direct consecutive nodes $n_0 \rightarrow n_1 \rightarrow ... \rightarrow n_n$, the path P identified by their edges can be expressed as $P_{n_0 \rightarrow n_n} = \{(n_0, n_1), ..., (n_{n-1}, n_n)\}$. Consequently, considering the directed graph in Figure 3.8, the path from node B to node E can be expressed as: $P_{n_B \rightarrow n_E} = \{(n_B, n_D), (n_D, n_E)\}$.

Control Flow Graph

A control flow graph (CFG) is a common way utilized for representing the control flow of a function. Given a specific function F of a program, its CFG can be defined with the following equation:

$$CFG_F = (N, E) \tag{3.3}$$

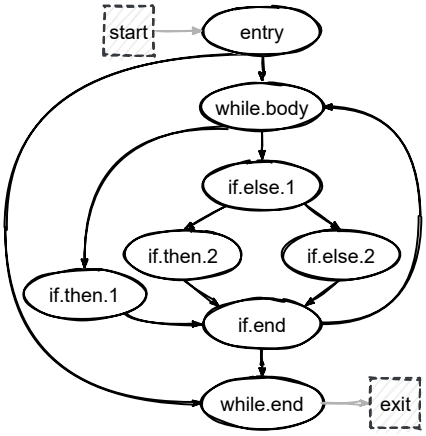
Where the set of nodes N is composed of the basic blocks in function F, and the set of edges in E describes the flow of control between the basic blocks.

Every CFG has a unique entry node that represents the initial basic block that is executed when the function owner is called. It is assumed that a CFG is a connected graph where every node can be reached starting from the entry node. Non-connected nodes are commonly removed during compilation when executing dead code elimination passes. Finally, it is possible to assume that every CFG has a unique exit node. This property can be ensured by simply adding a synthetic node that has to be reachable from all the nodes whose represented basic block ends with a function termination instruction.

The CFG can be considered at all the different levels of abstraction of a program. In fact, the definition in Equation 3.3 is applicable in the same way to the source code, to any of its intermediate representations or to the binary machine code. An example is given in Figure 3.9, the LLVM IR CFG representation of the function listed in Figure 3.9(a) is shown in Figure 3.9(b), where the nodes are identified by the basic block labels.

```
void someFunction( ... ) {
  while ( ... ) {
    if ( ... ) {
      ...  // Do something
    }
    else {
      if( ... ) {
        ...  // Do something
      }      // else
      else {
        ...  // Do something
      }      // different
    }
  }
}
```

(a) *Example function source code.*



(b) *LLVM IR CFG for the given example program.*

Figure 3.9 – *Example program source code and relative LLVM IR CFG: The compilation of the source code for the function listed in (a) with the LLVM compiler produces a bitcode whose IR CFG structure is shown in (b). Every CFG is composed of nodes connected by directed edges that represent respectively the basic blocks and their connections. The structure between the different CFG representations may vary due to the effects of the compiler optimizations.*

Dominator Relation

In a given CFG, a node n_X dominates a node n_Y if all the paths $P_{n_E \to n_Y}$, from the entry node n_E to n_Y contain node n_X. In other words, node n_X dominates node n_Y if:

$$n_X \in P \mid \forall\, P \in \{P_{n_E \to n_Y}\} \tag{3.4}$$

The dominator relation properties of a CFG are commonly preserved during the compilation phases of a program and they can be utilized for further analysis activities [130]. For this

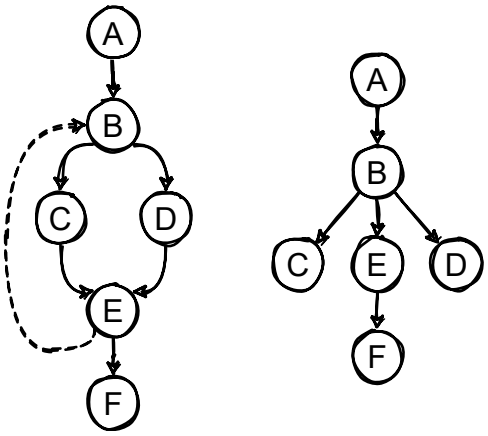


Figure 3.10 – *Dominator tree of a control flow graph: The dominator tree on the right side can be utilized for identifying the back edge (n_E, n_B) of the CFG represented on the left side. Furthermore, all the ancestor nodes of a given node in the dominator tree dominate the node also in the CFG. An interesting property of the dominator relation is that it is propagated through all the different compilation phases.*

specific purpose, the dominator relation can be expressed as a directed graph like the dominator tree shown in Figure 3.10. For example, the relations in a dominator tree can be helpful in identifying the back edges of a CFG. In fact, an edge (n_X, n_Y) in a CFG is a back edge if node n_Y dominates node n_X in the dominator tree.

Call Graph

Another significant graph for the analysis of a program is represented by its call graph. A call graph is a graph that connects calling basic blocks (basic blocks that contains a function call instruction) to the entry basic block of the called function. The edges of a call graph are shown if Figure 3.11. The definition of call graph is essential for introducing the consequent concept of interprocedural CFG.

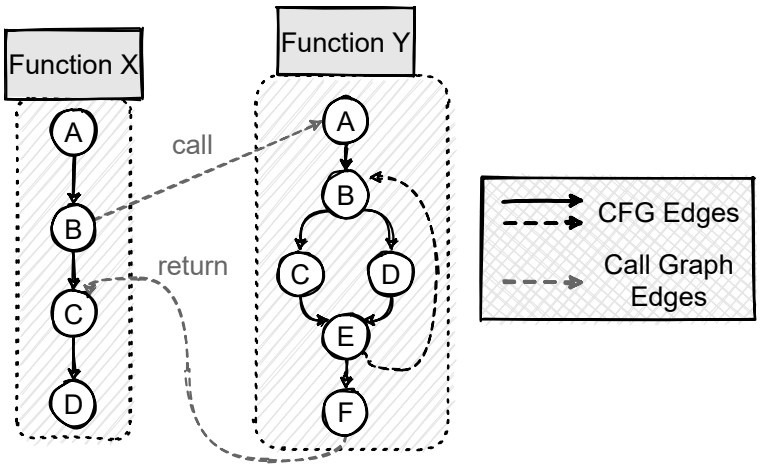


Figure 3.11 – *Interprocedural control flow graph: Graph that describes the complete control-flow of a program by connecting the different CFGs with edges that represent the program's call graph.*

Interprocedural Control Flow Graph

The interprocedural control flow graph (ICFG) is a directed graph that describes the complete control flow of a program [75]. In contrast to the definition of CFG, an ICFG considers the structure of multiple functions of a program. The graph of an ICFG can be considered as a graph of interconnected CFGs. The kind of edges that connect two CFGs, due to a function call or return instruction, describes the call graph of a program. An example of an ICFG is shown in Figure 3.11.

3.3.2 Timing Analysis Approaches Classification

The large majority of the available approaches for timing analysis of programs belongs to one of the following two classes: static or measurements-based methods [178]. The former class of approaches requires an in-depth description of the target architecture for the generation of an accurate model of the target processor. The approaches in the latter class instead tries to tackle the complexity of the static modeling by relying on mere measurements extracted directly from the target. All the approaches in both the classes can be further classified depending on aspects like automation, generality, applicability and others.

In general, the implementation of a timing analysis tool follows a standard scheme based on consequent steps. The common steps performed by a timing analysis tool and their connections is shown in Figure 3.12. The goal of the first step executed by a timing analysis tool usually consists in reconstructing the program ICFG for a given architecture-dependent binary executable. In the next step, this information is consequently processed for extracting all the different CFGs in the program and its flow information. This is possible by considering the target ISA semantic. At this point, the first timing considerations for parts of the program can be made by relying on a given target processor model. The model describes the timing behavior of all the physical resources included in the target processor and it can be equally generated via static analysis or deduced from measurements. The local estimations are finally utilized for computing a global estimation for the execution time of the complete program depending on its ICFG. This estimation can eventually consider a given input data for producing a timing estimation for a specific control-flow path in the ICFG. At the end of the process, a timing analysis tool offers the possibility of visualizing the computed results.

The way of producing of deducing a timing model for the target processor determines to which classification class belongs the approach. For both the classification classes, hereafter are summarized some of the key aspects intrinsic to their available timing analysis tools.

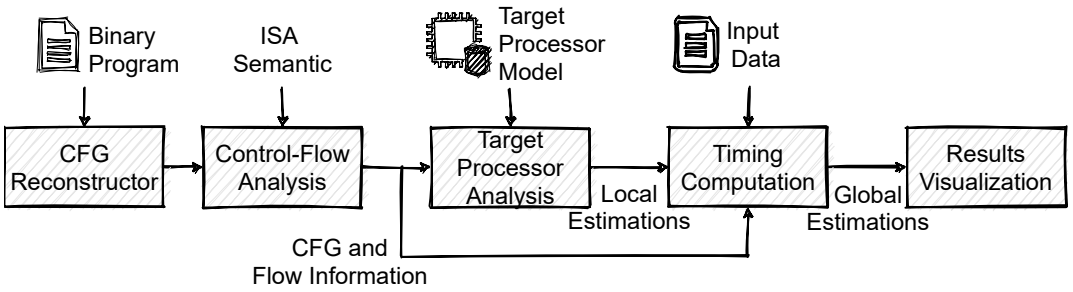


Figure 3.12 – *Common steps performed by a timing analysis tool (edited from [178]): Both the static and measurements-based timing analysis tools performs a similar sequence of steps for computing their estimations. A program is initially analyzed for extracting the information contained in its ICFG. Relying on a timing model of the target processor and on the control-information, the tool compute an estimation for the execution time of the complete program or part of it. A timing analysis tool always shows the computed estimation at its termination.*

Static Analysis

Static method approaches do not execute the software on the target platform or in a simulator. Instead, they require the realization of an accurate and in-depth model of the target architecture that is used for producing results depending on the program control-flow path. The workflow of a standard timing analysis approach, such as the one implemented by the AbsInt's WCET Analyzer aiT tool [35], is composed of three consecutive phases:

1. *Flow analysis* - Analysis required for the reconstruction of the complete program control- and data-flow.
2. *Architectural analysis* - Analysis designed for generating an architectural timing model by considering every minimal hardware component in the target.
3. *Global bound analysis* - Analysis conducted on top of the results of the previous two for determines the appropriate bounds for the execution of the complete program of parts of it.

Depending on the complexity of the hardware components implemented in the target processor, this can be a hard task. For instance, it is not possible to accurately evaluate the performance of a multi-core system via static analysis due to the complexity of the problem [32, 179, 24]. The analysis of multi-core architectures is not the only limitation of the static approaches. In fact, modern single-core architectures including complex hardware mechanisms (such as speculative, superscalar, out of order execution and others) can introduce timing anomalies in the performance behavior of a system [98, 133]. These anomalies are extremely hard to be statically modeled [132].

Measurements-Based

Modern processor architectures appear too complex to be modeled for timing analysis and, in most cases, their documentation is not completely available due to intellectual property restrictions [174]. The main alternative to the different static analysis approaches and their limitations is represented by measurements-based solutions. These kinds of approaches base their timing analysis on measurements about the execution time of a program [174, 12]. The analysis is commonly conducted in two different ways:

1. *Water-mark analysis* - Analysis based on end-to-end measurements of the execution time of a program for the execution of different control flow paths.
2. *Compositional analysis* - Analysis that composes the measured execution times of the single basic blocks of a program, or their aggregations, to times or distributions of times for the complete program.

Measurement-based timing analysis approaches are preferable to the static analysis ones. One of these commercial analysis tool is RapiTime [131]. An analysis based on mere observations allows an implicit modeling of the target system behavior without requiring in-depth modeling of complex hardware mechanisms (which is time-consuming and error prone). However, also the measurement-based approaches show some drawbacks [29, 126]. Most of the drawbacks are due to three different main problems. The first one is the consequence of the possible unavailability of hardware tracing capabilities that are not implemented in the target platform. The second one is due to the difficulty of forcing a complex program to execute a desired control-flow path by relying on a configuration of the input program data set. Finally, especially in case of complex processors, execution times often cannot be measured for all the possible initial states of the stateful hardware resources.

3.3.3 Performance Estimations

The large amount of available timing analysis tools covers different analysis goals. For example, the verification and validation activities for the realization of safety-critical systems require the identification of a safe upper bound for the execution of a program or part of it. This goal is commonly solved by relying on solid WCET timing analysis tools. The quality of these tools depends on the level of pessimism introduced in the WCET estimation. The level of pessimism represents the padding added to the actual WCET value that is due to the analysis approach. Lower is the level of pessimism, better is the the timing analysis tool. Unfortunately, the input data set that causes the observation of the WCET for the execution of a program is unknown. Therefore, these timing analysis tool perform a kind of analysis that is input-independent. However, the goal of producing an accurate WCET value for a given system is out of the scope of this thesis.

This thesis is focused on defining a methodology that allows the evaluation of the performance of a system. A timing analysis tool for evaluating the performance of a system has to produce an accurate timing estimation for the execution of a program for a specific input and considering multiple factors. This kind of analysis tools is essential in the early development stages of a system, such as the design space exploration. In fact, fast and accurate estimations are used for optimizing the design of a system. Therefore, in addition to the input data, the resulting timing estimation has to consider also a given system configuration. This configuration involves both the hardware and software components of a system. Multiple parameters influence the execution time of a program, such as the different compiler optimizations, the chosen memory allocation, the settings of the physical resources include in the target processor and others.

The estimations accuracy has to be supported by a sort of analysis that allows the generation of such results in a very fast way [173]. Differently from the WCET scenario, in every iteration of the exploration of a design, the exploration requires the evaluation of multiple configurations and different input data sets. It is unacceptable to wait for a long time for evaluating every single configuration. In fact, as discussed in Section 2.1.2, an ideal timing analysis tool is supposed to produce high-accurate estimations while preserving elevated analysis speed capabilities. The accuracy metric can be evaluated by comparing the produced estimation with the measurement of the effective execution of the program on the target platform. Differently, an adequate value of simulation speed is quantified in million of instructions simulated per second (MIPS).

3.4 Simulation Approaches for Performance Estimation

The goal of the research in producing performance estimations of a target system via timing simulation is commonly oriented on proposing new simulation methodologies focused on enhancing the simulation speed capabilities compared with the state of the art. Multiple different approaches, as the one proposed in this thesis, ensure elevated simulation speed capabilities by relying on some common properties between them. In fact, faster simulations can be executed by setting aside unnecessary too low-level and complex hardware details and by executing the simulation on a powerful host machine. The level of abstraction at the base of the simulation technique is the main responsible for its simulation speed capabilities. In general, higher is the level of abstraction and higher are the simulation speed capabilities. However, the level of abstraction can highly-influence the simulation accuracy too. In fact, multiple approaches based on an elevated level of abstraction show that their estimations are more inclined to be less accurate than others. Therefore, defining a host-based simulation methodology for efficiently evaluating the performance of a system is a hard task.

The remaining of this section is structured in two parts. The first one gives an initial overview about different simulation techniques that are part of the state of the art but that are not related with the simulation technique proposed in this thesis. Consequently, the second part of the section described different host-based simulation techniques that are inspirational for the simulation methodology later described.

3.4.1 Different Simulation Categories

An initial list of available simulation categories is provided hereafter. All the approaches belonging to one of these categories try to tackle the hard problem of accurately and efficiently evaluating the performance of a system via simulation. Unfortunately, the different levels of abstraction adopted by these approaches cannot ensure them to achieve a sufficient trade-off between the results accuracy and the simulation speed required by an ideal solution (see Section 2.1.2).

Traditional Cycle-Accurate Approaches

Traditional simulation techniques, such as RTL [63, 184] and ISS simulators [187, 34, 116], are too slow to be applicable in the design space exploration activities. In fact, they are not suitable tools for supporting the rapid evaluation of multiple configurations. Their slowdown is mainly due to the very low-level of abstraction that forces the simulators to consider in-depth and unnecessary details about the target architecture to simulate. Recent solutions have been proposed for enhancing the performance of these simulators. For example, the simulation methodology presented in [13] shows a speedup of multiple orders of magnitudes by relying on a hardware accelerator implemented in the RISC-V ISA [4]. Many other methodologies try to speed up the execution of such simulations in a similar way [129, 27, 81]. However, none of these approaches scales in case of large target systems. In fact, they all require a huge amount of in-depth information that is often unavailable. Furthermore, even if they can produce cycle-accurate results, they still require a large amount of compilation time and their simulation speed is not comparable with the one achievable by the simulation approaches later described.

Trace-Driven Simulators

A large number of different simulation methodologies have been proposed for overcoming the limitations of the traditional simulators. Between them, , the so-called trace-driven simulators represent a more scalable and flexible simulation solution [176, 69, 142, 183]. The common methodology at the base of these approaches allows achieving better simulation performance by separating the simulation of functional and non-functional (e.g. timing) aspects of a system. These kind of measurements-based approaches associate the timing behavior previously measured for an execution trace to the functional simulation of the target code. This separation reduces the overhead due to the consideration of fewer components and details in parallel. The timing information represented by the measured traces allows the implicit modeling of the timing behavior of parts of a program considering its execution on the target platform. This is recognized as a beneficial property and it is reflected in the simulation methodology proposed in this thesis too. However, trace-driven simulators can produce inaccurate results. An exact result is produced only if it exists a trace that exactly replicates the simulated control-flow. Depending on the program complexity, this property implies the consideration of very long pre-recorded traces. Furthermore, the traces are often adjusted via slow micro-architecture models that slowdown the complete simulation speed.

Analytical Performance Modeling

Most of the faster simulation approaches in the state of the art are the ones that belong to the analytical performance modeling category [72, 128, 71, 154, 74]. These approaches are commonly utilized in the early design phases. They allow obtaining fast but rough estimations that can be used by the system designers as an early feedback for multiple different configurations. Analytical models are usually composed of complex mathematical models that describe the timing behavior of the complete system component per component. The higher is the complexity of a hardware component, the more complex it is to model it via some formulas. In general, the simulators based on analytical models are structured in two consecutive phases. Initially, during the target profiling phase, specific performance metrics (such as instructions count, cache misses rate, mispredicted branches rate and others) are extracted from the execution of a large suite of programs. This data is utilized for producing the performance models required by the simulation methodology. This data extraction phase is executed only once per hardware configuration. Consequently, during the estimation phase, the previously generated models collaboratively produce the necessary estimations depending on the simulated instructions visited in the dynamically discovered control-flow path. One of the critical points of these approaches consists in the accuracy of the models considered during the simulation. In fact, the complexity of these models directly depends on the complexity of the hardware resources included in the target processor. Therefore, especially for complex processors, and in a similar way to the static analysis approaches, the so-generated fast performance estimations can be extremely inaccurate.

Phase-Based Simulators

The approaches belonging to the phase-based category represent a different fast simulation alternative. These approaches are based on measurements and they can achieve fast simulation speed capabilities. Instead of simulating the complete program from end to end, their speedup is achieved by simulating only a significant part of the program. This is possible according to the program cyclic behavior theory [143, 144, 68, 125, 146]. This theory argues that the execution of a program does not show a steady behavior and that it can be decomposed in periodic phases. These phases are the direct result of the coding process (the coding guidelines often induce the source code to follow some common patterns). In fact, programs

are coded in a modular way that allows functions to contain loops in which other functions can be consequently invoked starting further nested loops. A direct consequence is that after the first initialization phase, the remaining execution of the program shows a periodic phase behavior. From one side, these approaches scale well for large programs like the ones in the industrial settings. On the other side, the main criticality of these approaches is the identification of the different phases. Multiple different techniques propose to identify the program phases by considering the values of the hardware performance counters available on the target platform. These values are proposed to be assessed at different kinds of granularity (e.g. instruction, basic block, set of consecutive basic blocks, etc.). The extracted values can be processed for identifying the program phases. For instance, in [144], the authors propose the utilization of a Fourier analysis on the period representative signal. The identified phases and corresponding values are finally utilized for producing a performance estimation via a partial-simulation that requires the simulation of every phase only once. Unfortunately, the individuation of the different phases can be tricky and misleading. In addition, the performance counters may not be implemented in all the platforms or they can be available in a limited set. Furthermore, the consideration of a single timing value for the simulation of a phase can lead to inaccurate results due to the different execution context of a phase. These three problems together can represent a significant source of inaccuracy for the fast performance estimation provided by these approaches.

Cross-Platform Estimations via Machine Learning Models

A final category of rapid simulation approaches for the evaluation of embedded systems consists of approaches that based their analysis on machine learning techniques [95, 82, 53, 100]. With the coming of innovative and powerful machine learning algorithms, these approaches produce their performance estimations by relying on models generated via complex algorithms that analyze and elaborate a substantial amount of data extracted from measurements or observations. Measurements and observations are commonly based on timing traces and performance counters values extracted directly from the target platform. The machine learning models are complex analytical models that can describe the performance behavior of a complete target platform or parts of it. Every single model requires to be generated only once per hardware configuration and they are program-independent. In fact, they are commonly trained with a wide set of benchmarks which does not include the program that has to be analyzed. Similarly to multiple other methodologies, the machine learning approaches require the execution of two distinguished phases. Initially, they require a training phase for the generation of the models. The training phase does not require any knowledge about the target system to analyze, the target behavior is learned from mere observations. Consequently, in the prediction phase, multiple performance estimations can be produced via fast simulations that rely on the previously generated models. The quality of the training set utilized in the first phase directly influences the accuracy of the predicted estimations. Unfortunately, the definition of a valid training set for ensuring accurate results is not trivial. Neither the auto-generation of synthetic benchmarks to consider during the training phase fully solve the problem [122, 76].

A different kind of approaches still based on machine learning techniques have been proposed for overcoming the limitations of the previously described approaches. In fact, multiple cross-platform machine learning approaches have been proposed with the intent of improving the accuracy results [186, 85, 101, 102]. These kinds of approaches claim that there exists a latent relationship between the performances of a program when executed in different target platforms. Therefore, they sustain that the performance behavior of a program executed on a fast host machine can be correlated to the one observable on a slower embedded target platform. This correlation can be described via machine learning models. During the

host-execution of the program, these models are used for generating the cross-performance estimations. Unfortunately, the overall results show that accurate results can be achieved only in some circumstances and by considering a fine-grained and low-level of abstraction (models based at the binary instructions level). A low-level of abstraction implies the introduction of conspicuous and undesired simulation speed slowdown. Differently, moving the models level to higher code structures (such as basic blocks, functions, phases and others) often determine the generation of inaccurate results.

3.4.2 Control-Flow-Driven Host-Based Simulation

The simulation methodology proposed in this thesis is a host-based simulation approach belonging to the control-flow-driven category. In general, the different control flow-driven timing simulators produce their performance estimations by commonly relying on the simulated CFG and on a priori estimation of the different execution paths of a program. This base concept is shown in Figure 3.13 and it is in accordance with simulation idea previously presented in Section 2.2. An initial phase is required for determining the execution time of the different parts of a program. The analysis can be conducted at different granularity levels. Consequently, multiple simulations can be executed for determining the control-flow path due to a given input data set. Depending on the control-flow path, the performance estimations of a system are produced by accumulating the execution time previously computed for the requested parts of the program.

The different control-flow-driven timing simulation techniques differ according to two main aspects. The first one consists in the technique utilized for ensuring a simulation to accumulate the relative execution times. One possibility is represented by an appropriate instrumentation or annotation technique of the simulation code. The additional code causes the simulation to update automatically the performance estimation during its execution. The modification or enrichment of an existing functional simulator represents a different possibility. In both the cases, the execution time considered for a specific part of a program can be constant or dynamically determined. The constant case represents the simplest solution but, at the same time, it is source of inaccuracy because unrealistic (see Section 5.2.1). Differently, as proposed by the simulation methodology described in this thesis, the dynamic consideration of a different execution time depending on the execution context of the program allows the generation of accurate results.

The second aspect that characterizes a control-flow-driven simulator is the level of abstraction at the base of the simulation. This aspect determines at which code representation the simulation considers the accumulation of the relative execution times extracted during the analysis phase for updating the requested performance estimations. In general, higher is the

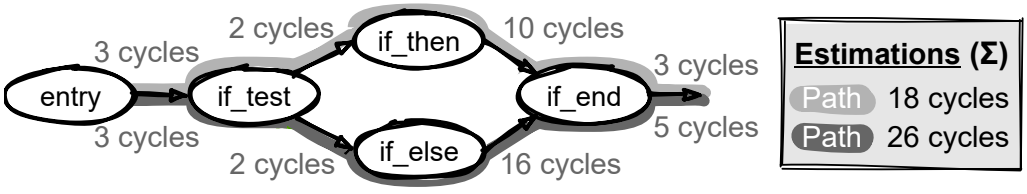


Figure 3.13 – *Control-flow-driven performance simulation concept: The simulation approaches belonging to this category produce the required timing estimations in a similar way. A priori analysis is conducted for determining the execution time of different parts of a program. Consequently, the control-flow path visited during the simulation is used for updating the timing estimation by accumulating a relative execution time previously determined during the analysis phase. The execution of different paths determines the generation of different performance estimations.*

abstraction level and faster capabilities can be achieved by the simulator. Unfortunately, at the same time, higher is the abstraction and harder is to model accurately all the aspects that determines the execution time of a specific part of a program. Therefore, the accuracy of the resulting performance estimations depends on both the accuracy of the modeling technique, utilized for describing the timing behavior of the different parts of a program, and the annotation technique utilized for allowing the simulator to dynamically update the timing estimations.

The literature shows that the different proposed simulation approaches base their simulations on three possible program representations that are: the source code, the executable binary code and the IR code (as the simulation methodology proposed in this thesis). Respectively, the simulation techniques are called in the literature source-level, binary-level and IR-level timing simulations. In the remaining of this section, it is provided an overview about the three different simulation possibilities. For every level of abstraction, multiple exemplary approaches are presented that allow the evaluation and discussion about eventual advantages and disadvantages due to the underlying simulation's code representation.

Source-Level Timing Simulations

The highest level of simulation abstraction is achieved by the source-level timing simulation approaches [94, 171, 97, 149, 150, 109, 185, 44, 147]. These approaches commonly produce the necessary performance estimations of a target system by natively executing on a powerful development machine a host-compiled binary version of the program. The common implemented workflow of these approaches is shown in Figure 3.14. During the host-execution of the program, the performance estimations are incrementally updated by relying on the additional simulation code that is pre-annotated directly on the program source code. The exemplary annotation of a small portion of code is shown in Figure 3.15. Therefore, the code annotation enriches the program with target-specific performance metrics that are computed before the simulation. The high-level of abstraction commonly ensures elevated simulation speed capabilities for these approaches. Furthermore, working at the source code level is commonly simpler than managing lower-level representations of the program.

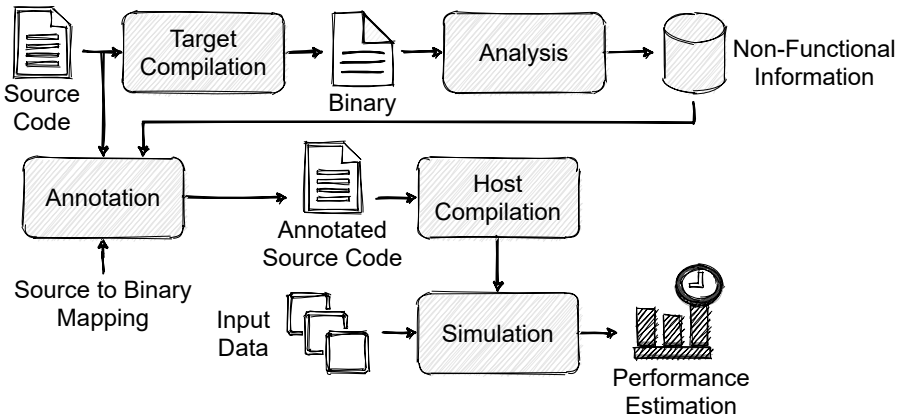


Figure 3.14 – *Common workflow for source-level timing simulation: This kind of approaches can achieve elevated simulation speed capabilities by executing an annotated version of the program natively on a fast host machine. During the simulation, the annotation code enables the consideration of non-functional aspects related to the target platform. Unfortunately, the annotation process requires an accurate mapping between the structures of the source code and the target-compiled binary that is hard to define.*

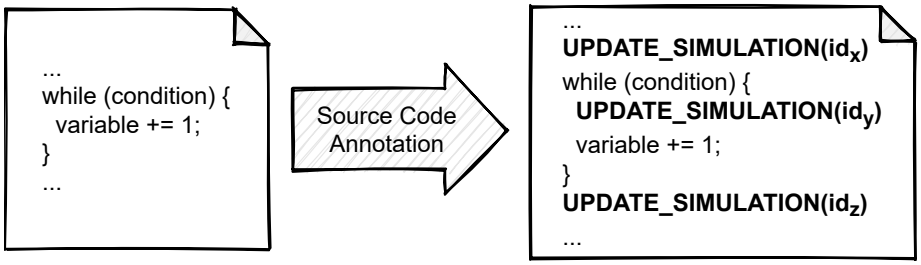


Figure 3.15 – *Exemplary source code annotation for target performance simulation: The source-level timing simulation approaches commonly require annotating the program software with appropriate instructions or target-dependent information that enable the generation of the necessary performance estimations by executing it on a fast host machine.*

Commonly, the target-dependent information (commonly non-functional properties such as timing) are extracted at the binary level directly from the target, from a cycle-accurate simulator or from the available hardware data sheets. In most of the cases, these approaches are based on a granularity that is defined at the basic blocks level of the source code. It is assumed that the instructions of a source-level basic block are always executed in order and consecutively from the first one until the last one. Furthermore, it is also assumed that the basic block is the smallest unit used by the compiler for structural optimization purposes. Unfortunately, this assumption is not valid for all the wide set of available compiler optimizations. Nevertheless, these assumptions imply that the annotation has to be inserted in specific places of the source code. For this delicate purpose, an accurate placement technique requires an appropriate mapping between the structures of the source code and its target-compiled representation. Unfortunately, the definition of such mapping is not trivial (see Chapter 4). In fact, due to the transformations resulting from the compilation process a direct mapping between the two structures is impossible. The mapping problem is often the main drawback of these approaches because its accuracy highly influences the precision of the produced performance estimations.

Different approaches try to overcome the mapping problem in different ways. Unfortunately, most of these techniques are not applicable in industrial settings. For instance, in [94] it is proposed to limit the set of applicable compiler optimizations excluding the ones that can substantially change the structure of a program. Another approach [22] proposes to mitigate the structural changes due to the aggressive compiler optimizations for producing a simple direct mapping. This can be done by inserting appropriate volatile labels that force the compiler to keep the initial structure of a program during all the compilation stages. In a different way, the approach initially presented [41] and consequently improved in [40] proposes a simulation methodology that does not require a mapping between the source code and binary structures. The timing model considered by this approach is not based on the performance of the binary representation but on a novel concept called elementary operations. The authors claim that these operations allow modeling both the target platform behavior and the timing effects due to the compiler optimizations on specific blocks of code. Unfortunately, the presented results show a high-level of accuracy in the produced performance estimations only in case of non-highly optimized programs.

The most inspirational approach for the simulation methodology proposed in this thesis is the one described in [150]. This approach can produce accurate performance estimations relying on a valid mapping technique later described and on an accurate source code annotation technique [153]. The results accuracy are ensured by a timing modeling technique that considers the execution contexts of the basic block of a program. This approach has been successfully applied also in support of WCET estimations [152] and in analyzing the behavior of the cache

memories of a target platform [151]. Highly accurate results can be produced relying on this simulation technique but it still suffers the common drawbacks of the source-level simulation techniques. In fact, the mapping problem is solved relying on the debug information produced by the compiler. This information can be ambiguous or incomplete causing undesired inaccuracy in case of multiple aggressive compiler optimizations.

Binary-Level Timing Simulations

In contrast with the source-level timing simulation approaches, the simulators based on the binary-level consider the lowest level of abstraction. Working at the binary-level can be harder than working at the high-level of abstraction offered by a programming language as C or C++. However, these approaches avoid the mapping problems common to the source-level and IR-level simulation techniques [88, 48, 127, 119, 117, 9, 60]. In fact, the binary code represents the result of the last compilation phase and it contains all the optimizations applied by the compiler. Unfortunately, the binary code compiled for a target platform cannot be directly executed on a host machine. Host and target machines commonly implement different ISAs. Therefore, a mechanism has to be defined for allowing the simulation of the target binary code on the host architecture.

A first possibility consists in a binary-to-binary translation mechanism [188, 170]. This mechanism converts a binary compiled for a target architecture to a functionally equivalent one that can be executed on a host machine. The process can be either dynamic or static [88, 48, 182]. In the latter case, some approaches propose to simplify the process by initially lifting the code to a higher representation (such as at the source code level or at the IR level [18]) and consequently compiling this representation for the host machine. The simulation relies on appropriate annotations for producing the requested performance estimations. The main drawback of these approaches is their complexity. The problem can be simplified by interpreting one instruction per time but this would cause undesired slowdown (similar to the slowdown intrinsic to ISS simulators) and it could be hard to resolve eventual indirect branches or jumps.

A different simulation possibility consists in basing the simulation on an enriched version of a fast emulator for enabling the possibility of context-sensitive timing simulation. For instance the binary-level simulation approach presented in [119] and consequently refined in [117] is based on the well-known and open-source QEMU emulator [11]. This tool emulates the functional behavior of a target architecture via a fast dynamic binary translation based on a JIT mechanism. The emulator enrichment allows QEMU to dynamically query an external database, the simulation's timing model (see Section 2.2) that contains relative execution times for the different part of a program depending on the execution context that is discovered during the simulation. The methodology workflow is shown in Figure 3.16 and it is composed of two phases. An initial analysis phase has to be performed for generating the so-called timing database representing the timing model of a specific system configuration. The timing database is generated by measuring the execution time of different parts of the program by changing the input data set. The measurements can be extracted directly from the target or from an accurate simulator. Thereafter multiple fast and accurate context-sensitive timing simulations can be performed relying on the same timing model. The methodology workflow of this simulation approach is a source of inspiration for the one presented in this thesis. In fact, the workflow later presented resembles the one shown in Figure 3.16. However, this context-sensitive timing simulation methodology shows some of the drawbacks commonly inherent to the binary-level timing simulators. For instance, a simulation is not retargetable and a timing model is specific for only one system configuration. Any minimal hardware or software modification in the system invalidates the timing model. Unfortunately,

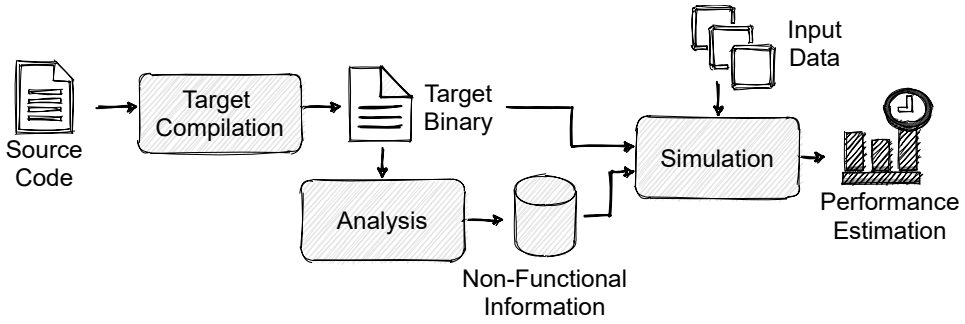


Figure 3.16 – *Common workflow for context-sensitive binary-level timing simulation: The program is directly compiled for the architecture of the target machine. Thereafter, the necessary analysis activities are conducted for extracting the essential non-functional information according to its execution context history. Thereafter the target binary can be directly executed on the host machine via an emulator that generates the performance estimations via simulation and relying on the previously extracted data. A different variant consists in annotating the target binary with the non-functional information before executing the emulator.*

an architecture can be evaluated only if the simulator effectively supports the target ISA. Furthermore, this approach does not consider the possibility of evaluating multiple architectures or configurations in parallel.

An additional improvement has been presented for this simulation methodology [118]. Faster simulation capabilities can be achieved by annotating the timing information previously contained in the database directly on the code with the support of an automaton that distinguishes the program execution contexts. However, this extension speedups the simulation speed capabilities but does not improve the approach flexibility or retargetability.

IR-Level Timing Simulations

On the one side, the very high-level of abstraction ensured by the source-level timing simulations make the mapping task harder, but on the other side, the low-level of abstraction of the binary-level timing simulations make these approaches less flexible and retargetable. As a consequence, basing a simulation approach on the IR code representation is an attractive alternative to the previously two simulation categories. The IR offers a sufficient level of abstraction that makes the code easily retargetable (as for the source code written in C or C++) but, at the same time, it contains a consistent amount of the effects of the compiler optimizations. In fact, the IR code is the result of the execution of multiple architecture independent compilation passes which make the IR structure closer to the binary one. This property implies a simplification of the mapping problem. Working at the IR level, in most of the cases, does not require the consideration of the source code structure at all. Mainly for these reasons, as for multiple other actual approaches in the literature [26, 77, 172, 16, 107, 145, 55, 43, 25], the proposed simulation methodology presented in this thesis is based on the intermediate representation of a program.

The different IR-level simulation techniques commonly implement a similar simulation workflow. This workflow is shown in Figure 3.17. The process starts with an architecture-independent compilation phase. Multiple chained compilation passes are executed for generating the IR code by requesting both the front end and back end to optimize the code. The resulting IR is consequently given in input to two different phases. In an initial phase, the compilation is completed by requesting the back end compiler to perform the architecture-dependent optimizations. The resulting binary is analyzed for producing the source of non-functional information (or timing model). In a second phase, the previously generated IR

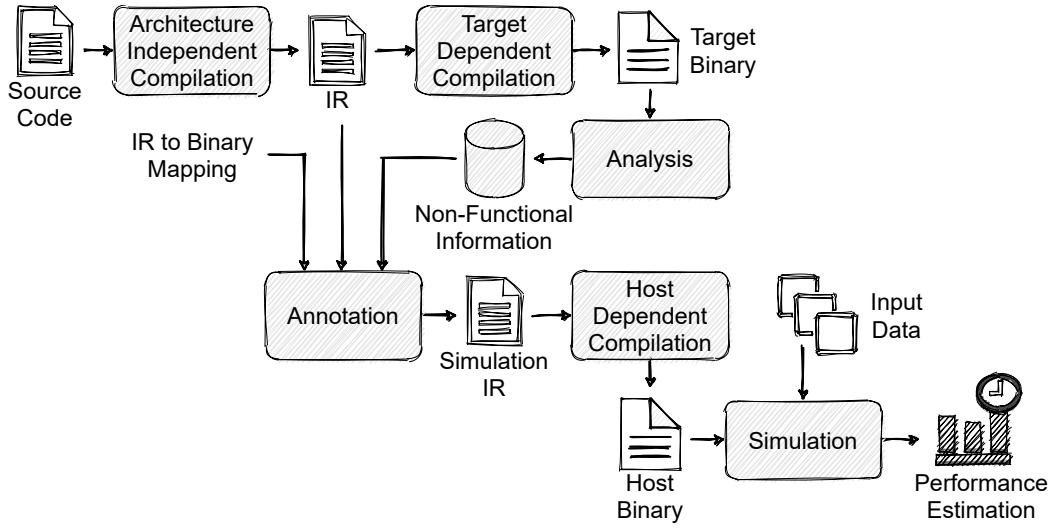


Figure 3.17 – *Common workflow for IR-level timing simulation: The IR code resulting from the architecture independent compilation of the source code is utilized for producing the target binary applying the target-dependent compiler optimizations. The binary is consequently analyzed for extracting the necessary non-functional information. This information is annotated in the IR code according to the previously defined IR to binary mapping scheme. The result of the annotation is an enriched version of the IR code that can be compiled for the host architecture. The execution of the resulting binary allows the rapid generation of the required performance estimations.*

code is given in input to the annotation phase. In addition to the IR code, the annotation phase expects in input also the IR to binary mapping and the previously generated timing model. Depending on the mapping scheme, the annotation inserts the information contained in the timing model directly on the IR code. The result of the annotation phase is an IR program that can be compiled for the host machine. The resulting host-compiled binary can be finally executed considering different input data. Depending on the annotation and on the visited control-flow, the execution of this version of the program produces the requested performance estimations. This kind of simulation workflow can ensure fast and accurate performance estimations in case of an exact mapping scheme. Minor inaccuracies can have major influence on the simulation results. In general, these approaches differ for the mapping generation algorithm and for the kind of analysis utilized for generating the necessary timing model to consider while simulating.

A slightly different simulation workflow is shown by two simulation approaches [26, 172]. In the first case, it is proposed to utilize a modified compiler that directly annotates the IR code with timing information produced via a slow cycle-accurate simulator. Unfortunately, the annotation forces the modified compiler to produce a binary executable whose structure differs from the one obtainable without instrumentation. Therefore, this approach suffers from accuracy problems in case the difference between the two structures is consistent. In the second case, the difference with the simulation workflow shown in Figure 3.17 is more accentuated. This approach requires an additional compilation phase. In fact, after the architecture-independent compilation phase the resulting IR code is not directly annotated. Differently, the IR code is translated to a sort of C-based intermediate source code. Some modifications are consequently applied to this code version for simplifying the mapping problem. Thereafter the code is compiled for both the host and target architecture. The target-compiled executable is utilized for the generation of the timing model. During the host compilation, the non-functional information is annotated in the new resulting IR. Finally, the executable

compiled for the host architecture can be simulated for producing the performance estimations. Even in this case, the main problem in the results accuracy consists in the fact that the source code structure analyzed differs from the original one. On the one hand, this approach simplifies the mapping problem, but on the other hand, it can produce inaccurate results due to the consideration of code whose structure differs from the original one.

The workflow of the timing simulation methodology presented in this thesis is in compliance with the one shown in Figure 3.17. It differs from the other approaches in the state of the art on two main aspects. First, in this thesis is presented a novel mapping algorithm that allows the accurate matching between the IR and binary structures of a program. Second, the simulation is not based on a pre-compiled version of the annotated IR code. In fact, the simulation is based on the dynamic compilation of the IR provided by the `lli` tool. This tool can efficiently compile and execute IR code. Furthermore, the tool offers the possibility of easy debugging and interaction. This property is essential for the definition of a co-simulation methodology that enables the evaluation of Simulink models.

3.4.3 System Performance Considerations in Simulink

The evaluation of the performance of a system requires a timing analysis tool that provides the possibility of visualizing the produced estimations. This represents one of the key requirements for a simulation tool (see Section 2.1.2). Unfortunately, a widely utilized development tool as Simulink, which offers an attractive but limited functional simulator, ignores the concrete system implementation considering only the simulation of architecture-independent models. This lack can cause undesired disparity between the behavior shown by a simulation and the one showed by the consequent final implementation of a physical system. For this reason, different multiple approaches have been proposed in the last years to support the exploration and visualization of the performance of Simulink models that consider the timing effects due to the execution of a model on a real target platform.

A first solution for enhancing the Simulink simulations with timing consideration is described in [57]. This simulation technique allows simulating networked embedded control systems. It is completely based on the native Simulink environment without necessitating the cooperation with any external simulator. This is made possible by providing to the system designers a set of custom and fixed Simulink libraries, to utilize when developing a system, that support the modeling of the system behavior. The simulation can consider the timing behavior of only these libraries. Consequently, it is not possible to directly evaluate an existing Simulink model. This can be done only after that the model is substantially rewritten by utilizing the provided timing-aware libraries. Considering the limited amount of provided libraries compared with the vastness of the components natively available in Simulink, the model translation activities for obtaining a functionally equivalent model can be challenging.

A direct but independent extension of the technique previously presented is described in [31]. The main drawback of the previous approach is tackled by allowing the definition of a Simulink model with the native components and by providing additional blocks for enabling timing considerations while simulating the system. In particular, by properly annotating an existing model, these blocks allow the external timing simulation of the different Simulink components that they are connected to. This methodology allows the timing consideration of elements of a system such as concurrent tasks, messages, signals and others. However, also this approach shows some limitations. In fact, it requires the system designers to appropriately annotate the model with the provided blocks. Unfortunately, the annotation is

completely manual and it is not considered the possibility of adding automatic support. Furthermore, the technique does not consider the possibility of evaluating the system performance directly in Simulink. In fact, the visualization of the simulation results is expected to be shown in the external timing simulator (e.g. SysML).

A different category of approaches is constituted by solutions that propose to enrich a given model for enabling the possibility of timing-aware simulations directly on Simulink [113, 83]. These approaches require an annotation phase in which the model is enriched by adding native Simulink components that force the simulator, at run-time, to mimic the behavior of the system. The annotation is made in a partially automatic way. The base concept of the annotation is that it is possible to model the timing behavior of a Simulink component by connecting a delay block to its output. Therefore, this timing information has to be generated before the functional simulation with an external performance estimation tool. Unfortunately, these approaches do not completely solve the unrealistic Simulink simulation problem described in Section 3.1.2. In fact, the delay blocks consider only one fixed execution time for every block. This assumption is realistic because it does not consider the different observable execution times of a piece of software depending on different execution contexts. Furthermore, compared with the methodology described in this thesis, these approaches are not prone to support the evaluation of different system configurations or of heterogeneous systems.

Finally, an interesting solution is represented by the approaches that tries to execute timing-aware Simulink models via a co-simulation technique [17, 6, 19, 15]. These approaches allow the synchronization between the Simulink simulator and an external timing simulator relying on the canonical synchronization algorithm [45]. The methodology later presented in this thesis belongs to this category. The basic idea of these approaches is straightforward: the different simulators, designed to simulate different properties of a system, are linked together in order to collaboratively produce a final unique result. The simulators are mapped to independent processes that interact with each other. The synchronization between the processes can result extremely complicated. Furthermore, these approaches force the Simulink simulator to consider the execution of the software part of a system only once per iteration. From one side, the simulation is enriched with timing considerations but, on the other side, also this strategy does not completely solve the unrealistic Simulink simulation problem. In fact, internally to the controller subsystem, the execution of the components are still simulated in fixed- or zero-time, even in case the co-simulation is used for validation purposes [136, 137].

3.5 Summary

This chapter presented the fundamental concepts that allow the later definition of the simulation methodology described in this thesis. Initially, the chapter provides a definition for the concept of embedded systems. In this context, the MATLAB Simulink environment tool is briefly introduced by mainly considering the Simulink simulation and code generation capabilities. Thereafter, the chapter describes the standard steps requested for the compilation of an embedded program. This description introduces the consequent presented LLVM Compiler Infrastructure topics. In particular, it is given an overview about the LLVM compiler's internals and the different compilation possibilities. A rough introduction to some basic timing analysis concepts for the evaluation of embedded systems precede an overview of the multiple different timing analysis approaches available in the state of the art. A special focus is given to those approaches in the related work that enable the performance evaluation of a system via simulation.

CHAPTER 4

Mapping IR to Binary Control-Flow Graphs

The scope of this chapter is to present the mapping approach algorithm defined for matching LLVM IR to binary CFGs. This is done by initially contextualizing the problem and the consequent criticality. Thereafter, some pertinent approaches that define the current state of the art are briefly introduced with a special focus on the more inspirational ones. An extension to one of these approaches is proposed before to finally describe the two-phases algorithm utilized for generating the accurate mapping at the base of the proposed simulation methodology.

4.1 Problem Definition

The goal of the multiple and different available host-based simulation approaches is to estimate the performance of a target system directly executing the host-compiled software program on a fast host machine. This means that a target system can be directly evaluated on a standard development machine. The host and target machines commonly, and especially in the embedded systems domain, implement different ISAs. Consequently, the compilation of the same program for the two machines produces different binaries.

In case of IR host-based simulation, and in a similar way as source level simulation, the simulator executes the IR code on a host machine or the IR code is annotated before being compiled for the host machine. An accurate IR to target binary-code mapping is essential because the simulation accuracy strongly depends on the effectiveness of the mapping between different code representations. Relying on the mapping, the simulator makes assumptions on the performance behavior of the target system while executing a different binary version on a host machine. Therefore, a highly accurate mapping is essential for a high estimation accuracy of the target's performance.

The main problem in defining a mapping approach that tries to provide an accurate matching CFGs at different program representations is that, due to aggressive compiler optimization, the CFGs structure can be substantially vary making a direct match between the basic blocks impossible [111].

The mapping accuracy is not the only requirement for a valid approach. In fact, considering the problem complexity and the large size of real industrial applications, it is expected the algorithm to be fully automatic. It is unrealistic to rely on experts for generating or fixing such mappings. Furthermore, it is desirable the mappings to be generated without requiring any compiler modification. Most of the time, modifications are not possible and, when possible, they can break the compiler's internal behavior.

4.1.1 Program Structure Representation

The available host-based approaches work either at the source code level or at the IR level. Working at the source level is beneficial in terms of readability, especially in the case of an expert is required to manually adjust an incomplete or ambiguous mapping. However, a CFG representing the source code structure is usually harder to map to its corresponding binary CFG compared with lower- level software representations. This is due the fact that a CFG at the source code level does not include any of the effects of the eventual compiler optimizations.

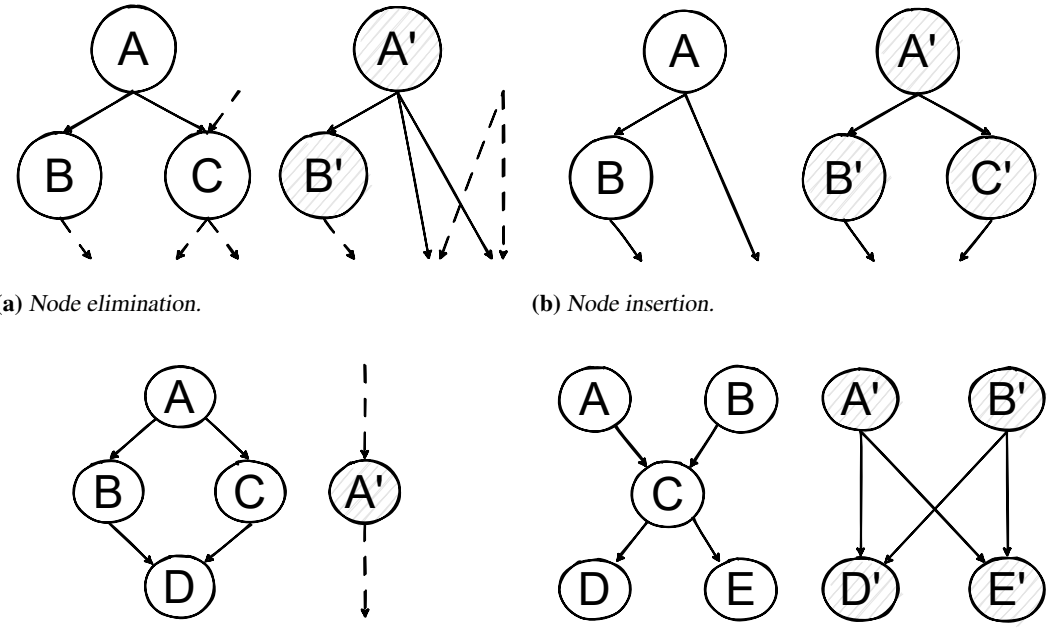
Several approaches prefer instead working at the IR level. The main benefit of working at the IR level is that the IR code includes, at least, all the compiler optimizations performed by the compiler during the front-end phase. This property implies the IR CFGs to have a closer structure to the binary CFGs compared to the source level ones. The CFGs mapping is a hard problem but considering a lower software representation abstraction like the IR ensures a simplification of the problem. The problem simplification usually comports the definition of more accurate mapping between the CFGs.

Considering the benefits of both the two software representations, the proposed mapping approach that is presented later in this chapter is intended to work at the IR level. However, even the structure of IR and binary CFGs may substantially vary due to aggressive compiler back-end optimizations. For this reason, an even lower software representation is taken into account for considering CFG structures that include more optimizations effect due to the middle-end and back-end compilation phases.

4.1.2 LLVM Optimizations and Passes

Compilers implement a wide range of optimizations designed for satisfying different compilation objectives. A valid CFGs mapping algorithm can achieve an adequate level of accuracy only by identifying and modeling all the optimizations that can change the structure of a CFG. Between all the available compiler optimizations, the ones that can change the structure of a CFG are the optimizations that operate at the basic blocks level [3]. The common possible effects of these optimizations can be summarized as basic block insertions or removal on the different CFG representations. Four different practical examples of these effects are shown in Figure 4.1. Multiple conditions can influence the optimizations in adding or removing nodes. For example, a node can be removed when its instructions are moved inside its successors, as shown in Figure 4.1(a). Pushing down instructions inside successor nodes may also determine the insertion of a new node, as represented in Figure 4.1(b). Nodes can be removed if the compiler recognizes that they contain path independent instructions as shown in the examples in Figure 4.1(c) and in Figure 4.1(d).

The proposed host-based simulation methodology is intended for executing timing simulations based on the LLVM IR code representation. Therefore, the solutions described in this thesis are specific for tools and conventions belonging to the LLVM Compiler Infrastructure.



(a) *Node elimination.* **(b)** *Node insertion.*

(c) *Node simplification.* **(d)** *Multiple nodes merging.*

Figure 4.1 – *Common CFG structural changes due to compiler optimizations: Four examples of possible changes to the structure of a CFG due to the effects of different compiler optimizations that can be classified as node additions or removals (figure edited from [25]).*

The standard compiler optimizations, generally available in most of the compiler implementations [3], can be executed in LLVM in the form of compilation passes via tools of the LLVM framework [86, 158]. New passes can be generated for implementing custom optimizations in a straightforward way. In LLVM the different passes are classified in six different categories [162]:

1. *Immutable* - Passes that do not change the structure of a program and that are designed for extracting compilation information from an IR module.
2. *Module* - Passes that operates on the entire program as a single unit. The passes in this category can modify the structure of a program at different levels like at the function, the basic block or the instruction granularity.
3. *Call graph* - Passes that need to traverse the program bottom-up on the call graph. They are not allowed to modify the program structure.
4. *Function* - Passes similar to the module ones with the difference that they perform local optimizations at the function level for a given function. The optimizations applied to a function are independent from the ones applied to other functions in the program. These passes can change the CFG structure of a function.
5. *Loop and Region* - Analysis oriented passes that allow evaluating loops and specific regions of a program without having the possibility of modifying them.
6. *Machine* - Passes utilized during the code generation process and that operate on the machine-dependent representation of a given function in the program. They can change the CFG structure of a function but they are not allowed to modify its structure at the IR level.

Only some of the LLVM passes can modify the CFG structure of a function. In particular, the passes belonging to the Module and Function categories are allowed to modify the IR representation of a CFG. The corresponding MIR CFG structure can be further optimized by changing its structure via Machine passes. These passes consequently influence the final structure of a binary CFG. Furthermore, inside the previously described categories of passes that can modify the structure of a CFG, the LLVM project divides these passes in three additional sub-categories [158]:

1. *Analysis* - Passes designed for computing and extracting information that can be used by consecutive passes or for debugging purposes.
2. *Transform* - Passes that implement the compiler optimizations that can change the structure of a program in some way.
3. *Utility* - Passes that offer some utility that do not fit in one of the previous two categories and that do not modify the structure of a program.

An approach for accurately mapping IR to binary CFGs has to consider all the effects of the compiler optimizations. Considering the LLVM classification of the passes, mapping LLVM IR to binary code requires to model the effects of the passes in the module, function and machine categories that are classified as transform passes.

4.2 Relevant and Inspirational Mapping Approaches

According to the literature, the problem of matching a higher-level software representation to its executable binary representation is a problem of intensive research. A wide number of mapping approaches have been published in the years. In this section, some of the most relevant and inspirational ones are mentioned and briefly illustrated.

4.2.1 Dominator Homomorphism

A compiler independent approach for mapping source code lines to machine instructions, for the purpose of source level simulation, is presented by the authors in [149]. Instead of modifying the compiler, this approach relies on a heuristic that utilizes the DWARF debug line information [30] produced during the compilation process. Thanks to this information, the execution order of the statements of both the software representations allows the definition of an accurate mapping.

The information contained in the dominator trees is essential for the mapping algorithm. As described in Section 3.3.1, a node in a dominator tree is dominated by all its ancestors also in the CFG. The authors claim that the changes in the CFG structure due to the compiler optimizations are also visible in its dominator tree representation. Therefore, the dominator relation contains important structural information about the possible execution order of the basic blocks. As a consequence, the generation of the mapping between the CFGs is produced by analyzing the dominance relation of the two CFG representations and by comparing their dominator trees.

The source code to binary mapping consists in a mathematical function that is intended to define a partial mapping between binary basic blocks and the corresponding basic blocks at the source level in respect of the dominator relation of both the software representations. The function ensures that the execution of a source level basic block b_S, mapped to a binary basic block b_B, always implies the execution of b_B. This property is ensured by mapping basic blocks which most closely resemble their semantic between the two CFGs, like shown in the example in Figure 4.2.

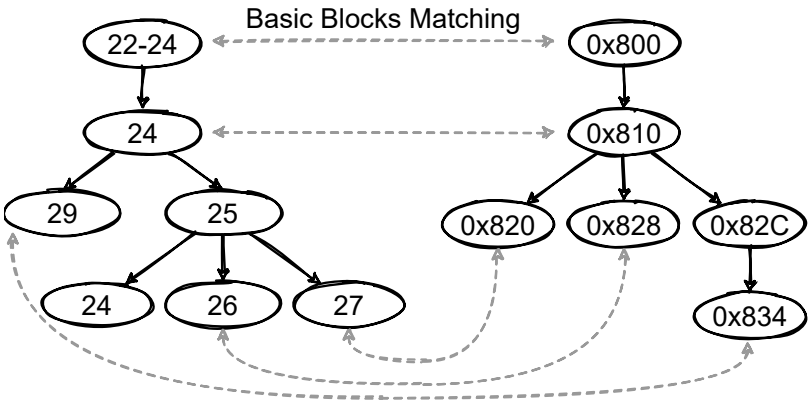


Figure 4.2 – *Example of dominator homomorphism matching algorithm (figure extracted from [149]): The source code lines of a program are mapped to the binary instructions, function per function, by defining a partial mapping between the basic blocks in the two dominator tree representations. The source-level basic blocks, on the left dominator tree, are mapped to the binary ones, on the right dominator tree, according to a most closely resembling criterion that defines the dominator homomorphism relation between the two CFG representations.*

Limitation

This mapping approach requires complex structural analysis of the CFGs and dominator trees in addition to precise debug information that has to be provided by the compiler. Unfortunately, the debug information can be ambiguous or imprecise [10]. For example, multiple binary instructions can be associated to the same line of source code or simple mismatches can happen [97]. Generating precise and complete debug information is not a trivial task. Compiler optimizations can break the full traceability between the source code lines and the compiled binary instructions [171]. Furthermore, the debug information might also be unavailable.

In general, producing a mapping between the source code of a program and its compiled binary version is a hard task. To solve the task it is necessary to systematically handle the full and wide range of available compiler optimizations [111]. Furthermore, eventual mappings produced considering the dominator homomorphism relation might not be unique [10] and consequently not accurate.

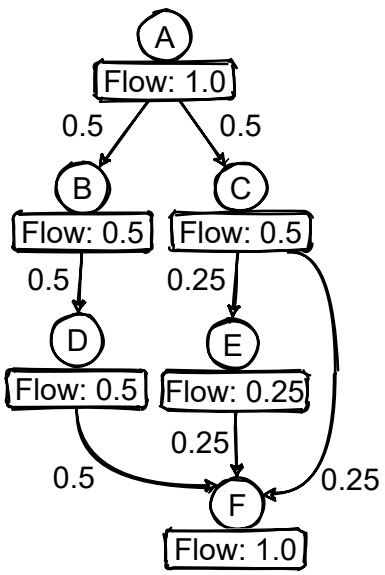
4.2.2 Subgraph Matching Algorithm

An automated approach for accurately mapping IR CFGs to the corresponding binary CFGs has been presented by the authors in [25]. The algorithm implements a heuristic that determines a mapping between the basic blocks of the two CFG representations relying only on the graph structures alone. The authors claim that a unique mapping exists between the basic blocks of an IR CFG and its corresponding binary representation. Furthermore, the unique mapping has to adhere to the overall control flow of the original CFG. This assumption is the key point of the algorithm and it is fundamental also for the proposed two-phases algorithm later presented in Section 4.4.

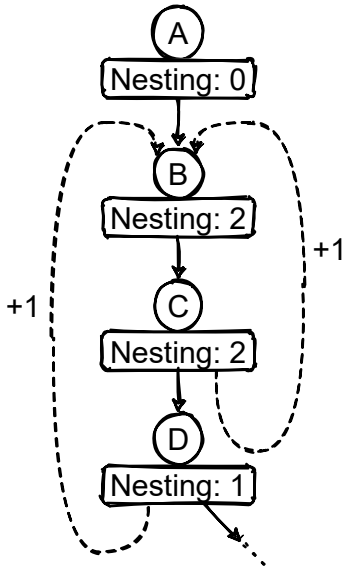
The algorithm requires an initial annotation of all the basic blocks in the two CFG versions. The annotation consists of two numeric metrics that have to be computed and consequently utilized by the algorithm for automatically generating the mapping. As shown in the example in Figure 4.3, the two metrics that have to be annotated are:

1. *Flow value* - Metric that globally describes the branching structure of a CFG (eventual back edges have to be ignored while computing these values). This metric is a real number, whose value is considered between 0 and 1.0. The flow value of the root node of a CFG is 1.0. The flow is consequently equally divided between its successors. Starting from the successors of the root node, and for all the remaining nodes, the flow value of a basic block is equal to the sum of the incoming flows. The outgoing flow of every node is always equally divided between its successors. The example in Figure 4.3(a) shows how these values are computed for a simple CFG starting from the root basic block A.

2. *Nesting level* - Metric that counts the number of looping paths that include a given basic block. The metric consists of an integer number whose minimum value is 0. An example that shows the nesting levels of a simple CFG is shown in Figure 4.3(b).



(a) *Flow value: CFG branching descriptor.*



(b) *Nesting level: Number of nested loops including a basic block.*

Figure 4.3 – *Flow value and nesting level: The two metrics of a CFG that are at the base of the mapping algorithm based on a heuristic presented in [25]. The flow value metric describes the branching structure of a CFG and the nesting level its loops.*

These two metrics are essential for producing a valid mapping. A valid mapping ensures that, for every mapped basic block, the number of traversed execution paths and the number of run-time passing through paths are identical between the two representations.

After computing and annotating the required metrics for all the nodes in both the CFG representations, the algorithm continues with producing the necessary matching. The matching candidates between the nodes are selected by considering nodes of both the CFGs with identical metric values. Starting from the root, and continuing until the end, all the mapping is iteratively generated by matching identical nodes that have not been yet mapped. A simple example of resulting mapping is shown in Figure 4.4.

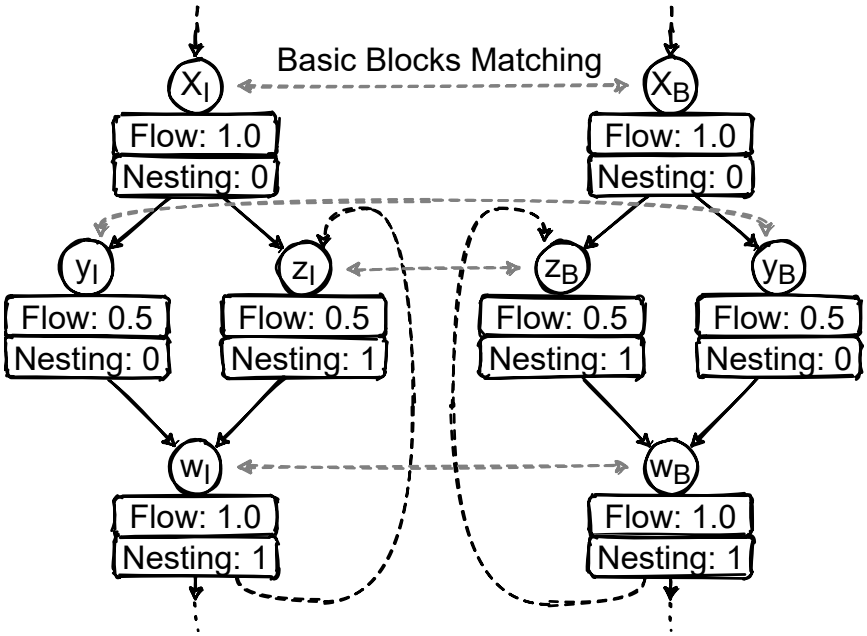


Figure 4.4 – *Example of heuristic subgraph matching algorithm: In order, and starting from the root nodes, the algorithm presented in [25] directly matches IR to binary basic blocks identified by identical values of the two annotated metrics flow value and nesting level.*

Limitation

Unfortunately, the algorithm proposed by the authors is not fully automatic. In fact, the algorithm fails in producing a complete mapping in case of ambiguities. The problem is a direct consequence of the choice of mapping two CFGs by considering their graph structure alone. An ambiguity is caused by the possibility of multiple equal matches. For example, multiple equal matches occur when one representation of a basic block can be mapped indistinguishable to more than one basic block of the other CFG because all of them share the same amount of flow value and nesting level. An example of ambiguity is shown in Figure 4.5 where the algorithm fails in mapping the IR basic blocks n_C and the IR sequence $n_B \rightarrow n_D$ because they can be equally mapped to one of the binary basic blocks $n_{C'}$ and $n_{B'}$.

In case of ambiguities in the mapping, these have to be manually fixed by an expert. The expert can disambiguate the mapping of single basic blocks by matching the basic blocks implementing the same code at the IR and binary level. Still, this can be a hard task depending on the complexity of the program, the number of ambiguities and the similarity between the instructions. In case of large applications, this solution does not scale and it can be tricky to disambiguate very similar operations.

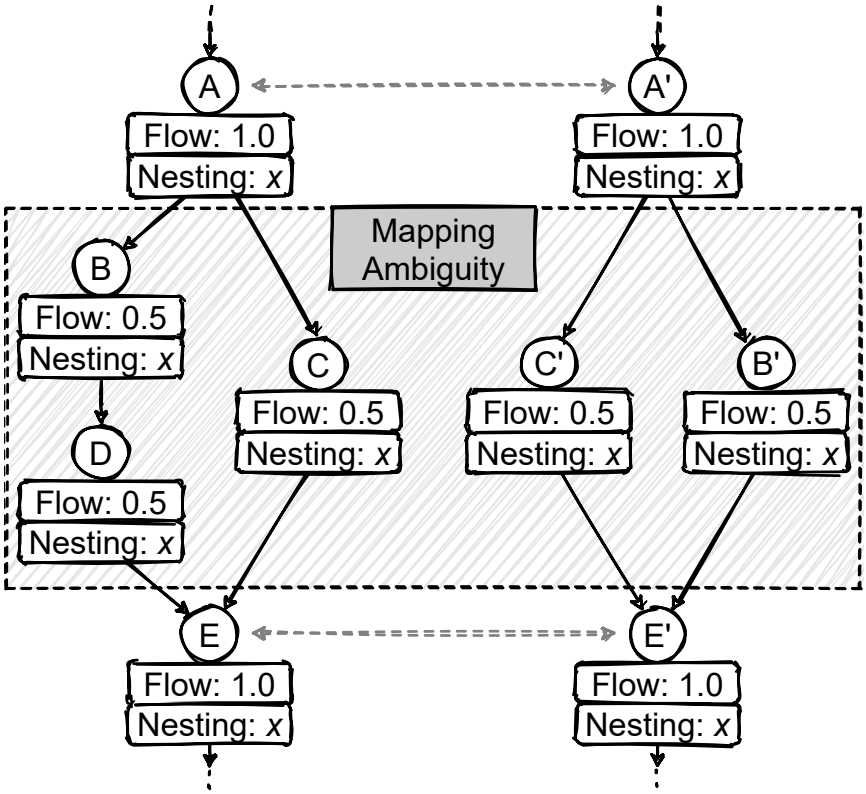


Figure 4.5 – *Example of CFG mapping ambiguity: The algorithm presented in [25] fails in automatically mapping an IR CFG to its corresponding binary CFG in case of multiple equal matches. In this example, the basic block n_C and the IR sequence $n_B \rightarrow n_D$ of the IR CFG on the left side can be equally match to the basic blocks $n_{C'}$ and $n_{B'}$ because they share the same amount of flow value and nesting level.*

4.2.3 Other Approaches

Many other approaches try to solve the mapping problem between different CFG representations. A first interesting approach in the literature is the control flow dependency mapping presented by the authors in [112]. The base idea is to match source code to binary basic blocks that are executed under the same condition. Similar branching in the two software representations are identified and utilized for matching the involved basic blocks. The algorithm relies on the compiler debug information and it suffers from the ambiguity problems described in Section 4.2.1. Even if this approach seems to be more robust to aggressive compiler optimizations than the dominator homomorphism approach (other structural optimizations are considered in addition to loop unrolling), it can still fail in producing an accurate mapping.

A different approach [171] tries to tackle the limitations of the dominator homomorphism approach by simplifying the mapping problem. The simplification consists in considering a higher optimized version of the program when a specific structure of the source code is highly changed by the compiler optimizations. Imprecision and ambiguities in the mapping are reduced by substituting the source code parts subjected to aggressive compiler optimizations with functionally-equivalent optimized IR code. The substitution of the code does not always solve the problem. In most cases, the optimized IR code shows a structure that is closer to the binary representation, helping in producing a valid mapping. In some other cases, mapping IR to binary code can be complex too and therefore the proposed problem simplification does not fully solve the problem.

Considering the difficulties caused by the structural modification of the CFGs due to aggressive compiler optimizations, other approaches [97, 172, 150] prefer to utilize a different technique based on optimizations handling. These approaches try to model the structural changes of the CFGs (especially the loops in a CFG) by mimicking the transformation rules applied by the compiler. This way of approaching the problem is compiler specific and it requires modeling a huge number of optimizations. Furthermore, applying the same compiler optimizations in a different order can cause different structural changes depending on eventual dependent effects between the optimizations. Therefore, an appropriate modeling technique can ensure accurate mappings between two CFGs but this way of tackling the problem does not scale in complexity. In fact, it might be impossible to efficiently model all the possible compiler optimization effects. Differently from other works, the approach presented in[97] proposes a solution for reducing the problem space by recursively considering sub-regions of a CFG. This solution reduces the size of a graph in a scalable way but it still suffers from infeasibility of modeling all the possible effects of the different compiler optimizations.

A final alternative, inspired by the previously mentioned optimization handling approaches, consists in the mapping strategies presented in [16, 107]. These kinds of approaches are focused on mapping IR to binary code. Instead of relying on the initial IR code produced by the front-end compilation phase, these approaches consider the last optimized version of the IR code. In this way, analyzing the IR code that is given in input to the compiler's back-end, the mapping algorithm has to model only the final architecture-dependent optimizations. Furthermore, this version of the IR code is closer to the compiled binary. In this way, it is easier for the algorithm to identify, modeling and mapping the loops in the two different program representations. Better mapping accuracy can be achieved compared to the previous approaches but minor imperfections in the optimizations modeling can introduce significant inaccuracy in the resulting mapping.

4.3 Fully-Automatic Subgraph Matching Algorithm

A first contribution to the state of the art presented in this thesis consists in the proposal of a solution in support of the approach presented by the authors in [25] and previously described in Section 4.2.2. The approach can produce accurate mappings but it is not fully automatic. As discussed, the main limitation of the subgraph matching algorithm is that the heuristic fails in producing a complete mapping in case of ambiguities in the CFGs. It requires the supervision of an expert that manually fixes the gaps. A fully automatic solution is desirable instead. Hereafter is presented a tracing-based solution that can make the approach fully automatic.

4.3.1 Tracing-Based Solution

The idea at the base of this solution consists in replacing the supervision of an expert for solving eventual ambiguities with an automatic mapping disambiguator component as shown in Figure 4.6. This component is intended to complete a partial mapping produced with the algorithm presented in Section 4.2.2 and solving the ambiguities by relying on execution traces of the program. For this purpose, the disambiguator compares the program flow in the binary traces with the one in the IR traces that are generated executing the program with the same input data. The process can be iterated multiple times by varying the input data, for visiting different paths in the CFGs, until a complete mapping is produced.

The workflow shown in Figure 4.6 is perfectly applicable for the ideal simulation workflow of the host-based approach previously presented in Figure 2.5. In fact, the timing model is intended to be generated by measuring and analyzing multiple timing traces that are extracted

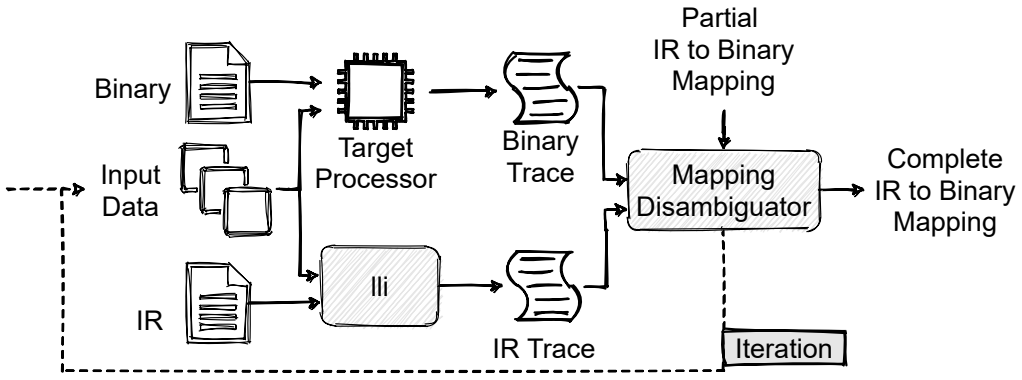


Figure 4.6 – *Proposed fully-automatic subgraph matching solution: An automatic mapping disambiguator can replace the expert supervision for completing eventual only partial mappings. The disambiguator solves the ambiguities in the CFGs mapping by comparing IR traces with binary traces generated with the same input data. The process can be iterated multiple times by varying the input data until all the basic blocks are completely mapped.*

directly from the target processor by varying the input data. The mapping disambiguator can consequently reuse the extracted tracing information. The IR traces instead can be generated by executing the IR code via `lli` and forcing it to print out the label of the visited basic blocks during the program execution. The basic block labels can be easily extracted by implementing an LLVM annotation pass. The annotation consists in adding a specific instruction at the beginning of every basic block that forces the `lli` execution to export the basic block label.

Algorithm

The disambiguator mapping algorithm is explained relying on the exemplary ambiguity mapping problem previously represented in Figure 4.5. The algorithm can be considered as an extension of the one presented in Section 4.2.2. In fact, an initial mapping is required to be generated between the CFGs using the subgraph matching algorithm. In case of the generation of an incomplete mapping, due to some ambiguities, the disambiguator algorithm can be executed for filling the mapping gaps. The pseudo code in Algorithm 1 describes all the necessary essential steps required by the proposed algorithm.

The initial generated partial mapping succeeds in matching the node n_A to node $n_{A'}$ and node n_E to node $n_{E'}$. The other nodes instead cannot be mapped because of an ambiguity problem between them caused by the same metrics of flow value and nesting level in the subgraph. In general, an ambiguity can be expressed as a pair of disjointed sequences of nodes that share the same predecessor. They also share a successor, if any. For example, in the figure, the nodes n_A and n_E are respectively the predecessor and the successor of the ambiguity $(n_B \rightarrow n_D, n_C)$. These two sequences of nodes can be indistinguishably mapped to the nodes of the other CFG representation in the ambiguity $(n_{B'}, n_{C'})$. According to the partial mapping initially generated, a sequence of nodes in an ambiguity can be mapped to only one sequence of nodes in a specific ambiguity of the other CFG representation. This means that, for example, node n_C can be mutually mapped only to node $n_{B'}$ or node $n_{C'}$. The initially generated partial mapping is one of the crucial inputs of the algorithm.

The first step of the algorithm requires the identification of all the ambiguities in two CFGs by traversing them with a depth-first search. Thereafter, the ambiguities have to be sorted in a way of data structure, similar to a dictionary, that associates an ambiguity in one CFG representation to another ambiguity in the other CFG. For example, considering the previously

Algorithm 1 Mapping Disambiguator(CFG, CFG', trace, trace', partialMapping)

```
1: mapping := partialMapping
2: [ambiguities ⇔ ambiguities'] := identfyAmbiguities(CFG, CFG', partialMapping)
3: adjustTraces(trace, trace')
4: while true do
5:     if |[ambiguities ⇔ ambiguities']| = ∅ then
6:         break("Solved all the ambiguities.")
7:     else if trace = ∅ or trace' = ∅ then
8:         break("No more traces to analyze.")
9:     end if
10:    // Map sequences and mutual sequences
11:    sequence := popUntilNextAmbiguousSequence(trace)
12:    sequence' := popUntilNextAmbiguousSequence(trace')
13:    mapping := mapping ⋃ map(sequence, sequence')
14:    mutual := getSibling(sequence)
15:    mutual' := getSibling(sequence')
16:    // Update mapping and mark ambiguity as solved
17:    mapping := mapping ⋃ map(mutual, mutual')
18:    solvedAmbiguity := (sequence, sequence')⇔( mutual, mutual')
19:    [ambiguities ⇔ ambiguities'] := [ambiguities ⇔ ambiguities'] \ solvedAmbiguity
20: end while
21: return mapping
```

described mapping ambiguity, an entry of the data structure has to be $(n_B \rightarrow n_D, n_C) \Leftrightarrow (n_{B'}, n_{C'})$. The operator $\Leftrightarrow$ is utilized for expressing the mutual mapping between the single sequence of nodes in the ambiguity.

The algorithm can start solving possible ambiguities in the mapping considering the flow information contained in both the IR and binary input traces. The information in the traces has to be arranged in a way that a trace consists of a sequence of basic block labels, if the trace is an IR trace, or a sequence of basic block start addresses, in case of a binary trace. A mapping is specific per function, therefore only the information specific for a given function has to be considered. The algorithm sequentially and repeatedly accesses both the traces. This causes the algorithm to follow the program's control flow until a stop condition is satisfied. Three stop conditions cause the algorithm to stop when trying to complete the mapping between two CFGs and these are: an ambiguity is encountered, no more traces are available or all the ambiguities have been solved. The first stop condition allows the algorithm solving an ambiguity problem without causing the algorithm's end. In fact, two sequences of nodes that cause the algorithm stopping represent two sequences that have to be mapped to each other. As a consequence, due to the mutual association property, the two non-visited sequences can be mapped to each other too. For example, as shown in Figure 4.7, a stop caused by visiting nodes n_C and $n_{C'}$ in the respective graphs implies the mapping ambiguity solution that initially maps them to each other and consequently mutual maps the nodes in sequence $(n_B \rightarrow n_D)$ to node $n_{B'}$. The other two conditions instead, force the algorithm to terminate.

The algorithm can be iterated multiple times by extracting new multiple traces by varying the input data for trying to visit different control flow paths and consequently solving as many ambiguities as possible. The iteration should be continued until a full mapping is obtained.

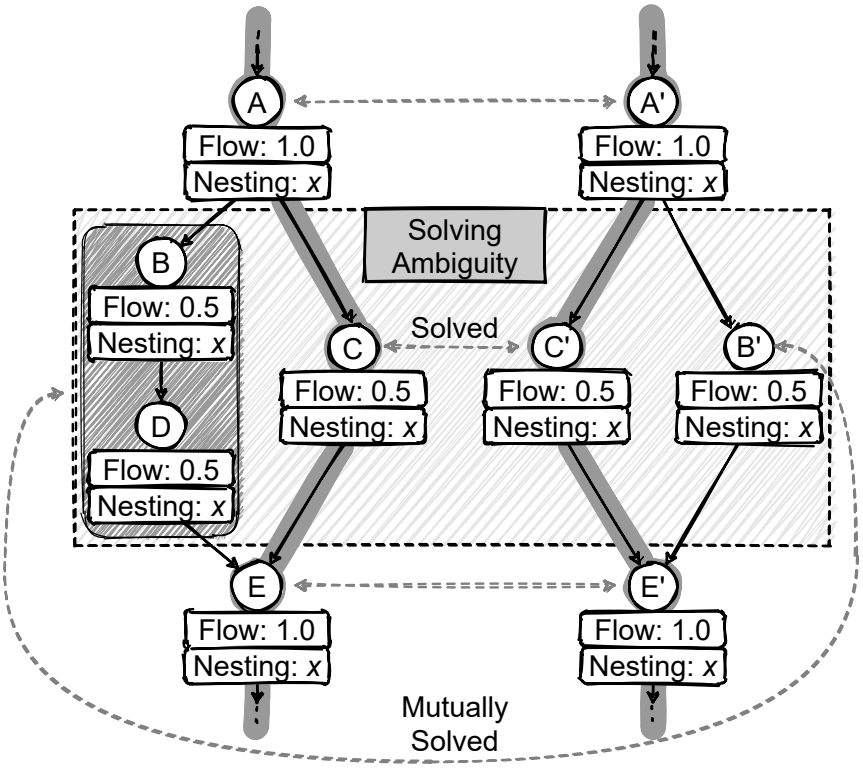


Figure 4.7 – *Fully-automatic subgraph matching disambiguation: A partial mapping can be completed relying on the information extracted by tracing the program execution flow at both the IR and binary representations. Two nodes like n_C and $n_{C'}$ can be directly matched relying on the visited program flow, marked in gray. As a consequence, also the sequence $(n_B \rightarrow n_D)$ can be matched to node $n_{B'}$ for mutual exclusion.*

4.3.2 Limitations

The presented tracing-based algorithm, starting from an incomplete mapping generated as described by the algorithm in [25], allows generating a complete and accurate mapping between two CFG representations in a fully automatic way. The improvement consists in filling the mapping gaps by comparing IR and binary execution paths extracted from the corresponding traces generated giving in input the program the same input data. On the one hand, the solution can fix the ambiguities problem, but on the other hand, it can be hard to identify the exact input data that leads the execution to visit the desired control flow path. Therefore, multiple iterations can be required for solving the problem showing scalability problems in case of large and complex industrial settings. A more flexible and scalable solution is consequently desirable.

4.4 Two-Phases Algorithm

On the one hand, the tracing-based proposed enrichment for the heuristic initially presented in [25] makes the mapping process fully automatic. On the other hand, as previously described, it can be hard to identify all the necessary input data for solving the mapping ambiguities. For this reason, a new and fully automatic methodology has been defined for mapping LLVM IR CFGs to their corresponding binary CFGs.

The new methodology is compiler specific because it relies on tools and functionalities provided by the LLVM Compiler Infrastructure. The bitcode format, that can be produced compiling a program only with the LLVM compiler, is the only IR code representation considered by the methodology.

The two prior mapping approaches presented in [149] and [25] are the main source of inspiration for the new two-phases algorithm. Both of them assume that there exists a mapping between every IR CFG and its corresponding binary representation and that it is unique. They also assume that such mapping exists even when the program is compiled with aggressive compiler optimizations. This assumption is supported from the fact that all the different optimization steps have to preserve the overall control flow of a given input program. This assumption is also at the base of other different approaches [97, 16, 107]. The approach of the two inspirational algorithms in solving the problem is similar: initially, they identify some key nodes that can be directly matched and consequently, they fill the mapping gaps relying on the information extracted from the control flow graphs or dominator tree.

As shown in Figure 4.8, the proposed methodology for mapping LLVM IR to binary machine code is composed of two distinct phases. To the best of the author's knowledge, differently from any other mapping approach in literature, the mapping between two different representations of the same CFG can be produced by executing a two-steps algorithm. The complex mapping problem is tackled by initially mapping the IR to MIR CFGs and subsequently mapping the MIR to binary CFGs. The two steps represent a fundamental simplification of the mapping problem. In fact, the compiler optimizations that can change the structure of a

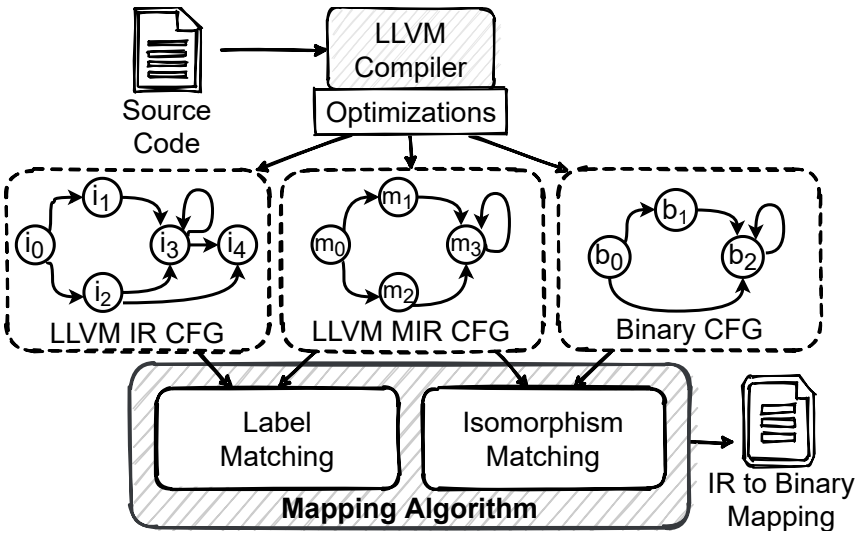


Figure 4.8 – *Two-phases methodology for LLVM IR to binary machine code matching: The structure of a CFG at the bitcode level is initially mapped to its corresponding LLVM MIR representations. Consequently, the structure of the second representation is mapped to the corresponding binary implementation. In the first phase, the mechanism relies on a label matching algorithm. In the second, the two CFG structures are mapped according to an isomorphism matching algorithm.*

program can be performed at different stages of the compilation process. The accuracy of a mapping algorithm strongly depends on the effectiveness in considering all of them. The different intermediate representations can represent a valid support in the mapping definition. For example, the structure of an MIR CFG already includes all the effects of the architecture-independent optimizations because these are performed at the IR level. The IR and MIR CFGs, between them, share some basic block labels. Furthermore, depending on the time of generation of an MIR module, the structure of an MIR module may also include part of the architecture-dependent optimizations. When the MIR is generated immediately before its translation to MC code, its structure resembles the one of the final binary code.

The problem simplification ensured by subdividing the mapping problem in two distinct phases enables the generation of mappings that can accurately match paths in the IR CFGs to paths in the binary CFGs. The approach is fully automatic and it does not require the supervision of an expert or any compiler modification. The next sections describe the two-phases algorithm presenting first the label matching algorithm, utilized in the first step, and the isomorphism matching used in the second and final step.

4.4.1 Label Matching Algorithm

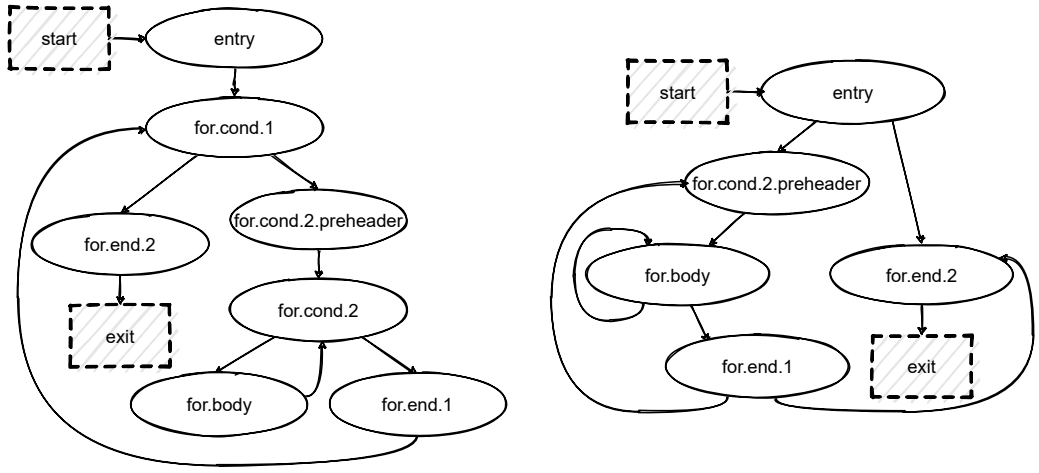
The algorithm for mapping paths of an IR CFG to paths of the corresponding MIR CFG relies on the information provided by the basic block labels of both the two software representations. In fact, this information supports the proposed methodology for automatic mapping IR to MIR CFGs. The basic block labels are completely independent from the debug symbol information, which can be unavailable or eventually imprecise. They are internal to the compiler and they identify in an univocal way the nodes of a CFG.

The graphs in Figure 4.9 show an example of the two types of CFGs annotated with basic block labels information resulting from the compilation with optimization level `-O1` of the `fir` function, extracted from the `edn` Mälardalen benchmark [54] and shown in Listing4.1. In particular, the graph in Figure4.9(a) represents the IR CFG and its corresponding MIR CFG is shown if Figure4.9(b). Generally, the structure of the two graphs is similar and they share multiple basic block labels. Unfortunately, as the example shows, a direct mapping is often not possible because of the effects of the compiler optimizations.

Listing 4.1 – *Source code of the `fir` function extracted from the Mälaralden `edn` benchmark [54].*

```
1  /*****************************************************
2  *                FIR Filter                          *
3  *****************************************************/
4  void fir(const short array1[], const short coeff[], long int output[]){
5     long int i, j, sum;
6
7     for (i = 0; i < N - ORDER; i++) {
8        sum = 0;
9        for (j = 0; j < ORDER; j++) {
10          sum += array1[i + j] * coeff[j];
11       }
12       output[i] = sum >> 15;
13    }
14 }
```

The compiler optimizations focused on the instructions level cannot modify the structure of a CFG. The optimizations that can change the structure of a program operate at the basic blocks level. These optimizations modify the CFG of the different functions in a program by inserting or removing nodes as previously discussed in Section 4.1.2. The label matching algorithm is intended to model all the node insertions and removals applied at the IR level



(a) *IR control flow graph.* **(b)** *MIR control flow graph.*

Figure 4.9 – *Example of IR and MIR CFGs annotated with basic block labels: The two graphs represent the structure of the corresponding IR and MIR CFGs for the* `fir` *function resulting from the compilation of the Mälardalen* `edn` *benchmark when compiled and optimized applying the optimization level* `-O1`.

as well as the ones applied to the MIR level. For example, the two graphs in Figure 4.9 shows the compilation effects due to the removal of two IR nodes (identified by the labels `for.cond.1` and `for.cond.2`) that are not present in the corresponding MIR CFG. Differently from the node removals, the insertion of a MIR node implies the appearance of a new node identified by a synthetic label that is not present in the IR CFG. In either the cases, the insertion or removal of nodes always preserve the overall control flow of the original IR CFG.

Due to the compiler optimizations that can change the structure of a CFG, for the same function, the set of nodes in the IR and in the MIR CFGs may differ. As a consequence, also the set of edges may differ. The differences between the two sets make a direct mapping between the two graphs impossible. The label matching algorithm is intended to map paths between two different CFG representations. If possible, the algorithm directly maps edges between them (sequences of length one). Otherwise, paths composed of sequences of consecutive edges are considered. The steps performed by the complete algorithm are listed in Algorithm2 and they are accurately presented in the following paragraphs.

Initial Partial Mapping

The label matching algorithm starts with the reconstruction of the IR and MIR CFGs. The nodes of both the graphs are consequently annotated with the labels of the corresponding basic blocks. At the end of this initial phase, the CFGs to be mapped appear in a similar way as the two graphs in Figure4.9. The label annotation inserted in the nodes allows defining an initial partial mapping between some of the edges in the two CFGs.

A supporting function is necessary for the next steps. The supporting function Φ is utilized for identifying a specific node n_l in a CFG of a function f that is annotated with a given input label l:

$$\Phi(l, CFG_f) = \begin{cases} \emptyset & n_l \notin N_f \\ n_l & \text{otherwise} \end{cases} \tag{4.1}$$

Relying on this function, and after the initial annotation, the algorithm continues by identifying the IR edges that have been preserved from the compiler optimizations. These edges can be directly mapped and they define the initial partial mapping between IR and MIR CFGs.

The steps required for generating the initial partial mapping are listed in the first part of Algorithm 2. After the initial labels annotation, the algorithm iterates through all the edges of one CFG representation and checks if any edge identified by the same node labels is present also in the other CFG representation. The connection between IR and MIR is possible thanks to the Φ function and the node labels. Coloring in gray the nodes that are present in only one CFG representation and in white the common ones helps in recognizing the common edges. If the edge is present in both the CFGs, the mapping is consequently updated with a direct match. Otherwise, the edge is placed in the list on unmatched edges. At the end, it is necessary to add to a second list of unmatched edges also all the edges in the other CFG representation that have not been considered in the initial mapping. Both the lists of unmatched edges and the partial mapping are the inputs of the next algorithm's phase.

Considering a more suitable example, in Figure 4.10 is shown the partial mapping generated by the algorithm between an IR and its corresponding MIR CFG extracted from the `adpcm` benchmark of the Mälardalen suite compiled with the middle-end optimization `-O2`. For the sake of simplicity, synthetic labels have replaced the original basic block labels. The highlighted edges show the paths that remain to map for the algorithm.

Algorithm 2 Label Matching($CFG^{ir}(N^{ir}, E^{ir})$, $CFG^{mir}(N^{mir}, E^{mir})$)

1: mapping := $\varnothing$
2: unmatched^{ir} := E^{ir}; unmatched^{mir} := E^{mir};
3: *// Initial Partial Mapping*
4: **for all** $(n_x, n_y)^{ir} \in E^{ir}$ **do**
5: **if** $\big(\Phi(x, CFG^{mir}), \Phi(y, CFG^{mir})\big) \in E^{ir}$ **then**
6: *// Direct match between edges*
7: $e^{mir} := \big(\Phi(x, CFG^{mir}), \Phi(y, CFG^{mir})\big)$
8: mapping := mapping $\cup$ $map\big((n_x, n_y)^{ir}, e^{mir}\big)$
9: unmatched^{ir} := unmatched^{ir} $\setminus$ $(n_x, n_y)^{ir}$
10: unmatched^{mir} := unmatched^{mir} $\setminus$ e^{mir}
11: **end if**
12: **end for**
13: *// Complete IR to MIR CFG Mapping*
14: **for all** $(n_x, n_y)^{ir} \in unmatched^{ir}$ **do**
15: **if** $\exists\ p^{mir}_{x \to y} \mid getMinSimilarPath(CFG^{mir}, x, y) \neq \varnothing$ **then**
16: mapping := mapping $\cup$ $map\big((n_x, n_y)^{ir}, p^{mir}_{x \to y}\big)$
17: unmatched^{mir} := unmatched^{mir} $\setminus$ $getEdgesInPath(p^{mir}_{x \to y})$
18: **end if**
19: **end for**
20: **for all** $(n_x, n_y)^{mir} \in unmatched^{mir}$ **do**
21: **if** $\exists\ p^{ir}_{x \to y} \mid getMinSimilarPath(CFG^{ir}, x, y) \neq \varnothing$ **then**
22: mapping := mapping $\cup$ $map\big(p^{ir}_{x \to y}, (n_x, n_y)^{mir}\big)$
23: **end if**
24: **end for**
25: **return** mapping

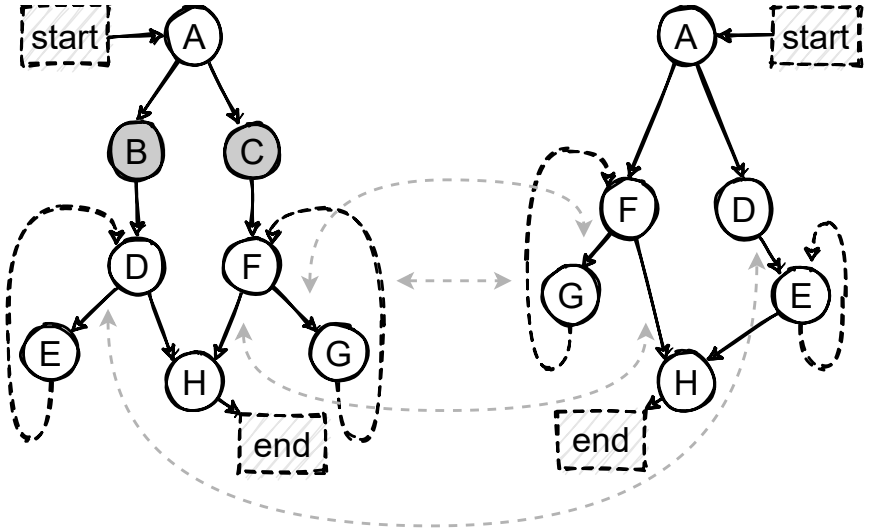


Figure 4.10 – *Initial partial mapping between preserved edges: The first part of the algorithm requires to directly map the preserved edges from the compiler optimizations between an IR CFG, (the one on the left side) and its corresponding MIR CFG (the graph on the right side). In gray are colored the nodes that, due to the compiler optimization effects, are part of only one representation of the CFG. The direct matches define an initial partial mapping between IR and MIR CFGs that will be completed in the next part of the algorithm.*

Complete IR to MIR CFG Mapping

The previously unmatched paths have to be mapped following the steps in the second part of Algorithm 2. The algorithm first tries to iteratively map the IR unmapped paths to some of the MIR paths and consequently, in a similar way, completes the mapping by matching still unmapped MIR paths to IR paths.

The initial and the final nodes of an unmatched path are always two nodes that have been previously colored in white. Furthermore, two paths in the two CFGs are matchable only if they are similar. Two paths are similar if they share, at least, the same nesting level. This means, for example, that if one path includes a back-edge, a back-edge has to be included also in its matching path. Therefore, the algorithm starts iterating through the unmatched IR paths and, for each of them, identifies the corresponding path in the MIR CFG. Another similarity property consists in the node labels. An unmatched IR path $p^{ir}_{A\to D}$, from a node with label A to a node of label D, can be mapped only to the minimal MIR path identified by nodes annotated with the same labels relying on the supporting function Φ. The third and final similarity criteria considers the dominator relation between the nodes in the two path representations. Two matchable paths have to share a similar dominator relation between the two CFG representations. For instance, in the mapping example shown in Figure 4.11, the unmatched IR path $p^{ir}_{D\to H}$ can be mapped only to the unmatched MIR path $\{(D,E),(E,H)\}^{mir}$ and not to $\{(D,E),(E,E),(E,H)\}^{mir}$ because it is the one that respects all the three similarity criteria. Every time a new match between IR and MIR paths is discovered, this is stored in the initial partial mapping structure and both the IR and MIR matched edges are removed from the corresponding list of unmatched edges.

Once completed the iterations through the unmatched IR paths, the algorithm terminates after repeating the same process for all the remaining unmatched MIR paths. After completing, the algorithm returns the mapping structure containing the complete mapping between IR and MIR paths.

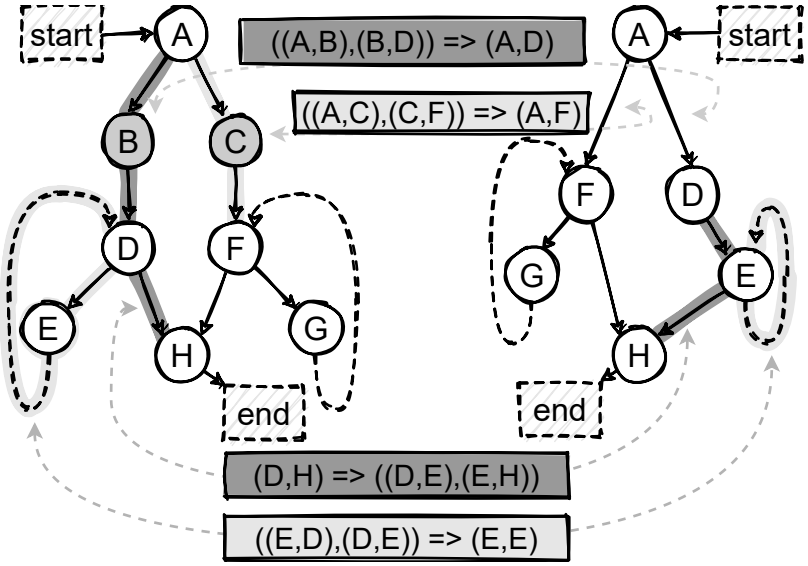


Figure 4.11 – *Complete IR to MIR CFG mapping: The initial partial mapping based on direct matching between preserved edges is completed by the second phase of the label matching algorithm. Paths, highlighted in gray colors, are matched between the two CFGs considering the similarity parameters between the traversed nodes.*

4.4.2 Isomorphism Matching Algorithm

The second step required by the proposed IR to binary mapping approach consists of an algorithm for mapping MIR to binary CFGs. In most cases, this is a not trivial task. As for the IR to MIR mapping generation, due to the effects of the compiler optimizations, a direct mapping between the nodes of a MIR CFG and its corresponding binary CFG is impossible. This task can be tackled by generating a mapping between the CFGs of the two architecture dependent representations executing the proposed isomorphism matching algorithm.

The algorithm specifies the necessary steps for the generation of an isomorphism between the two CFG representations in support of the mapping decisions. In fact, the algorithm describes how to obtain an isomorphism between the CFGs by modifying the structure of the graphs of both the representations while preserving their original control flow paths. Instead of trying to model all the possible effects of the compiler optimizations, that is a hard task, the approach tries to further optimize the CFGs for simplifying their structure and making them similar and easier to map. In fact, similarly to certain compiler optimizations, specific nodes can be removed in order to obtain two graphs with an isomorphic structure. The isomorphism ensures the definition of a unique mapping between the edges and paths of the original CFGs. The unique mapping concept is supported by the two inspirational approaches that are at the base of this algorithm. Consequently, the isomorphic structure of the processed CFGs allows a direct mapping between the graph structures.

The isomorphism matching algorithm is composed of multiple phases that will be described in details hereafter and are initially listed in Algorithm 3. An initial annotation phase for numbering the nodes of the CFGs is followed by a coloring phase that prepares the fundamentals for the isomorphism generation. Depending on the coloring results, the isomorphism is generated. Relying on the adjusted initial annotation, the isomorphism is finally utilized for defining an accurate mapping between MIR and binary CFGs.

Algorithm 3 Isomorphism Mapping($CFG^{mir}(N^{mir}, E^{mir})$, $CFG^{bin}(N^{bin}, E^{bin})$)

1: mapping $\leftarrow \varnothing$
2: *// Node numbering annotation*
3: *sortByAppearance*(N^{mir}); *sortByAddress*(N^{bin})
4: *assignIntegerIds*(N^{mir}, N^{bin})
5: *// Graph coloring*
6: *colorNodes*(N^{mir}, N^{bin})
7: *// Isomorphism generation*
8: CFG^{mir}_{iso} := *remove&Annotate*(CFG^{mir}); CFG^{bin}_{iso} := *remove&Annotate*(CFG^{bin})
9: N^{iso}_{mir} := *sort*(N^{mir}_{iso}) N^{iso}_{bin} := *sort*(N^{bin}_{iso})
10: *updateIntegerIds*(N^{mir}_{iso}); *updateIntegerIds*(N^{bin}_{iso})
11: *// Mapping definition*
12: **for all** $(n_i, n_j)^{mir}_{iso} \in E^{mir}_{iso}$ **do**
13: **if** *getAnnotation*$\left((n_i, n_j)^{mir}_{iso}\right) \neq \varnothing$ **then**
14: p^{mir} := *extractPath* $\left((n_i, n_j)^{mir}_{iso}, \text{getAnnotation } \left((n_i, n_j)^{mir}_{iso}\right)\right.$
15: **else**
16: $p^{mir} := \{(n_i, n_j)^{mir}_{iso}\}$
17: **end if**
18: **if** *getAnnotation*$\left((n_i, n_j)^{bin}_{iso}\right) \neq \varnothing$ **then**
19: p^{bin} := *extractPath* $\left((n_i, n_j)^{bin}_{iso}, \text{getAnnotation } \left((n_i, n_j)^{bin}_{iso}\right)\right.$
20: **else**
21: $p^{bin} := \{(n_i, n_j)^{bin}_{iso}\}$
22: **end if**
23: mapping := mapping $\bigcup$ map(p^{mir}, p^{bin})
24: **end for**
25: **return** mapping

Node Numbering Annotation

The objective of the first step of the isomorphism matching algorithm is to label the nodes of a CFG with unique numerical IDs. The nodes of both the CFGs are consequently annotated. The numerical IDs are essential for ensuring the definition of a direct mapping in the final phase. As discussed in Section 3.2.4, the structure of an LLVM MIR module is sequentially translated at the end of the back-end compilation activities in LLVM MC code. The MC code is the address-independent code representation for an object file and its structure is similar to the structure of the final binary executable. Therefore, the structure of the binary code resulting from the translation of the MIR instructions contained in the ordered basic blocks code always reflects their order of appearance in the MIR module. Considering the information intrinsically contained in the instructions' order of appearance of both the program representations, the algorithm provides a unique integer ID to all the nodes of a CFG for later purposes.

The numbering annotation process differs for the two CFG representations. The nodes of a binary CFG are straightforward annotated with values in the range from 0 to $\mid nodes \mid -1$ by following the ascent order of their first instruction's address reported in the binary file. In a similar way, the MIR nodes are annotated with an integer ID relying on the sequential order of appearance in the MIR function. The two annotated CFG representations are given in input to the next phase.

Graph Coloring

In order to obtain an isomorphism between the structure of a MIR and binary CFGs, it is necessary to remove some nodes from their structures while preserving their control flow. The algorithm utilizes a coloring scheme for initially marking and consequently identifying the necessary nodes that have to be removed for this purpose. In the first stage, all the possible candidate nodes to be removed are identified. The set of candidates is consequently reduced by identifying specific conditions for which, a node removal, would cause a change in the control flow.

The coloring activities start by coloring in a neutral white color all the nodes of both the CFG representations. In the consequent step, all the nodes are analyzed for identifying the ones that are eligible to be removed. A node has to be marked as a possible node to be removed if, ignoring potential back-edges and the unique successor of the synthetic start node, its in-degree is equal to one and its out-degree is at maximum one. All these nodes are colored in black, as shown in the examples in Figure 4.12. Some of the black nodes could be changed in different colors during the next steps.

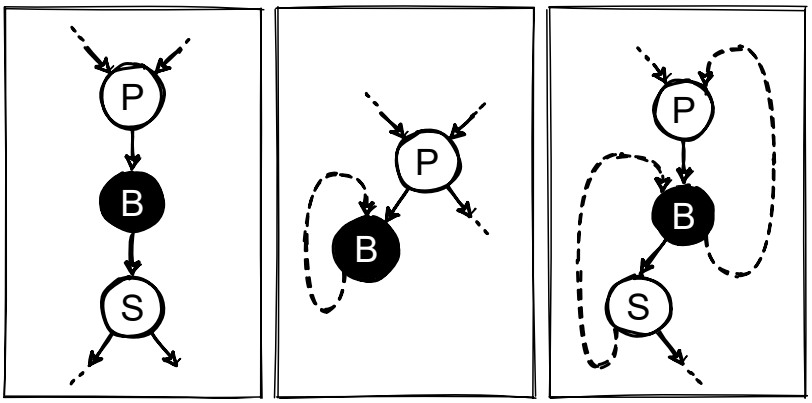


Figure 4.12 – *Initial removal candidate nodes are colored in black: The candidate nodes to be removed for the obtaining an isomorphism between MIR and binary CFGs have to be initially colored in black. A node has to be colored in black if, ignoring eventual back-edges, it has only one predecessor node and at maximum one successor node.*

After that the removal candidate nodes are identified, a subsequent coloring step is necessary for identifying which of the black node removals would not preserve the original structure of the paths in the final isomorphic graph. For this purpose, some nodes in the graphs are colored in dark gray. A node has to be colored in gray if it satisfies one of the following two conditions:

1. *The candidate node is the direct successor of at least two distinct black nodes* - The removal of both its predecessors may cause the change of the control flow by losing the information belonging to two distinct flows.

2. *The candidate node is the direct successor of the last node of an uninterrupted path of black nodes $p_{s\to e}$ and it shares a common direct predecessor n_p with the first node n_s of the path* - The removal of the black nodes' path would cause the loss of the flow of a complete path.

The two conditions that determine a node to be colored in dark gray are graphically shown in Figure 4.13 with the help of some examples.

The dark gray nodes identified in the last step allows the algorithm determining which of the black colored nodes have to be different colored for exiting from the set of candidate

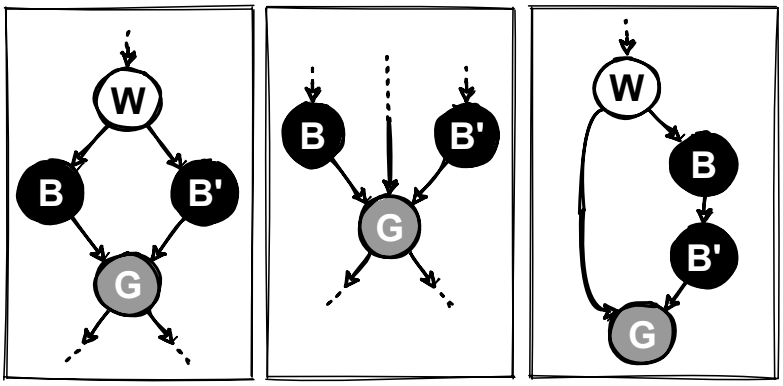
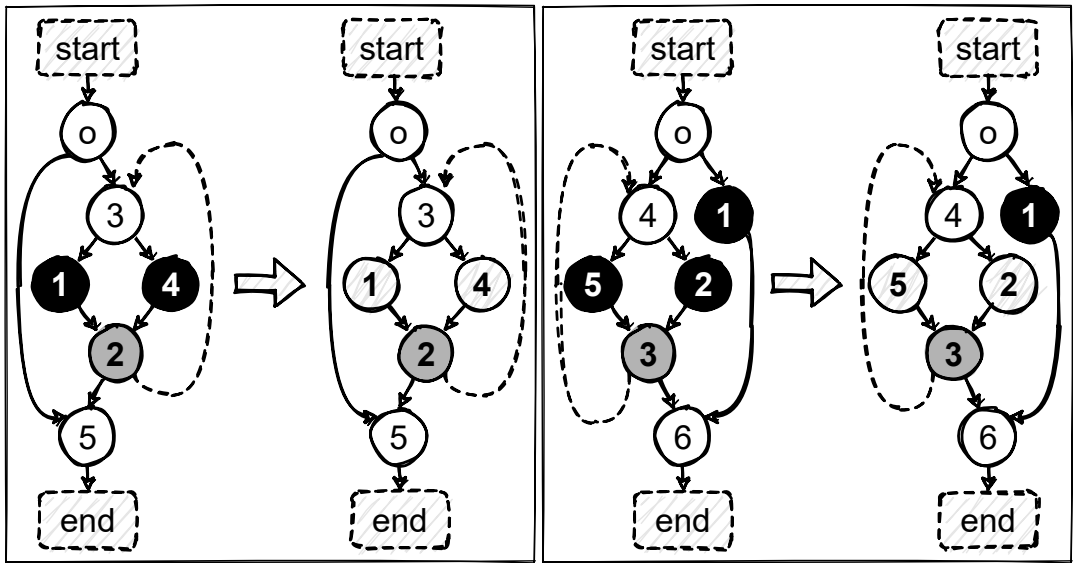


Figure 4.13 – *Conditions for coloring a node in dark gray: Exemplary cases where a node has to be colored in dark gray. These nodes represent a set of supporting nodes used for identifying, in the next step, the initial colored black nodes that must not be removed from the CFG. Therefore, every dark colored node is useful for preserving the structure of a CFG during the generation of the required isomorphism.*

nodes that later will be removed. Any node in both the CFGs has to be colored in light gray if, excluding eventual back edges, it has at least one successor colored in black gray. The coloring transition for some exemplary black nodes is shown in Figure 4.14. More precisely, both the initially black colored nodes in the MIR CFG in Figure 4.14(a) are consequently re-colored in light gray; the node identified by ID number 1 on the binary CFG in Figure 4.14(b) remains colored in black instead. The so colored CFGs are the inputs for the next step designed for generating the isomorphism between them.



(a) *Annotated and colored MIR CFG.* **(b)** *Annotated and colored binary CFG.*

Figure 4.14 – *Full CFG coloring in support of the isomorphism algorithm: The MIR and binary CFGs produced from the IR CFG previously presented in Figure 3.9(b) are utilized for showing the coloring stages required by the isomorphism algorithm. The initial two black nodes in the MIR version are consequently colored in light gray for preserving the flow structure. Differently, in the binary CFG, the node labeled with the number 1 remains colored in black and it will be later removed for making the graph isomorphic with the MIR CFG.*

Isomorphism Generation

The nodes that are still colored in black at the beginning of this step are the nodes that have to be removed from the CFGs. The node removal has to preserve the control flow of a CFG. Iteratively, and for both the CFGs, the algorithm requires the removal of the remaining black nodes by performing the following steps:

1. *Preserving the control flow and nesting levels of a graph* - This can be ensured by updating, after a node removal, any affected original edge or back-edge. Multiple scenarios have to be considered when removing a node n_B that, at the end of the coloring activities, is still colored in black. According to the definition of the coloring scheme, and excluding eventual back edges, a black node has at maximum one predecessor n_P and one successor n_S. The removal of node n_X requires to remove also its incoming and outgoing edges. Therefore, the edges (n_P, n_B) and (n_B, n_S) have to be replaced by a new edge (n_P, n_S) that directly connect the predecessor with the successor. Any incoming back-edge (n_P, n_B) has to be replaced by a new incoming back-edge (n_P, n_S) to the successor. In a similar way, any outgoing back-edge (n_B, n_S) (including self-loops as (n_B, n_B)) has to be replaced by a back-edge (n_P, n_S) starting from the node's successor. Some exemplary cases that show how to update the structure of a CFG after the removal of a black node are shown in Figure 4.15.

2. *Preserving the incremental numerical labeling scheme* - The numeric IDs are essential for the definition of the isomorphism between the CFGs. Therefore the order of the nodes has to be preserved. This can be ensured by updating the initially assigned IDs. The remaining nodes have to be sorted, according to their initially annotated ID, and a new ID (starting again from the value of 0) has to be assigned to each of them by following the ascending sorting order.

3. *Annotating the affected edges* - The flow information contained in the initial version of a CFG can be kept by annotating the new generated edges with information about the removals. Therefore, every time a new edge is created, it has to be annotated by indicating which path it replaced. For example, the removal of a node n_x, which causes a new edge (n_p, n_s) to replace the two consecutive edges (n_p, n_x) and (n_x, n_s), requires the annotation of the removed path $\big((n_p, n_x), (n_x, n_s)\big)$ on the new generated edge.

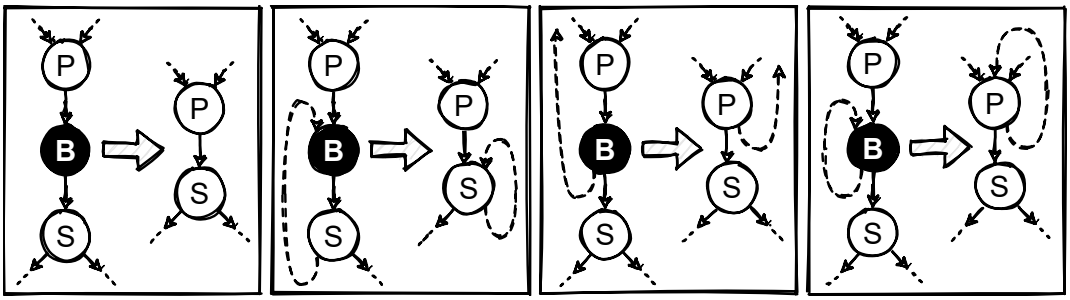


Figure 4.15 – *Control flow and nesting level preservation: The removal of any black node requires preserving the structure of a CFG and consequently to update the edges that are connected to the black node. Four different scenarios show how to properly update the edges of a CFG after the removal of a black node.*

After that all the three steps are correctly performed, the structures of both the MIR and the binary CFGs are isomorphic, like in the example shown in Figure 4.16. The isomorphism allows the generation of an initial direct mapping between the two CFGs that has to be completed by considering the information annotated on some of the edges, as explained in the next final step.

Mapping Definition

The isomorphic structures ensured by the previous step allows generating the final and complete map between corresponding paths in the original MIR and binary CFGs. In fact, thanks to the isomorphic property of the modified CFGs, the mapping can be produced in a straightforward way via direct matching of the graphs' edges (this was not possible before the modifications applied to both the CFG representations). The direct matching assignment is based on the information provided by the updated node IDs and by the eventual annotated edges during the previous step.

When generating the mapping via direct matching between the edges of the isomorphic CFGs, two different kinds of edges have to be considered:

1. *Simple edge* - An edge in an isomorphic CFG is part of the original graph only if it has not been annotated. A simple edge is a non-annotated edge. In this case, the original edge can be directly mapped with the corresponding edge or path in the other graph.
2. *Annotated edge* - This kind of edge is not part of the original CFG because it has been generated during the isomorphism generation step, after the removal of one or more black nodes. If the path composed of the sequence of edges $\{(n_0, n1), \ldots, (n_{n-1}, n_n)\}$ is the annotation assigned to the edge (n_x, n_y), the matching process has to consider the path $\{(n_x, n0), (n_0, n1), \ldots, (n_{n-1}, n_n), (n_n, n_y)\}$ instead of the simple edge.

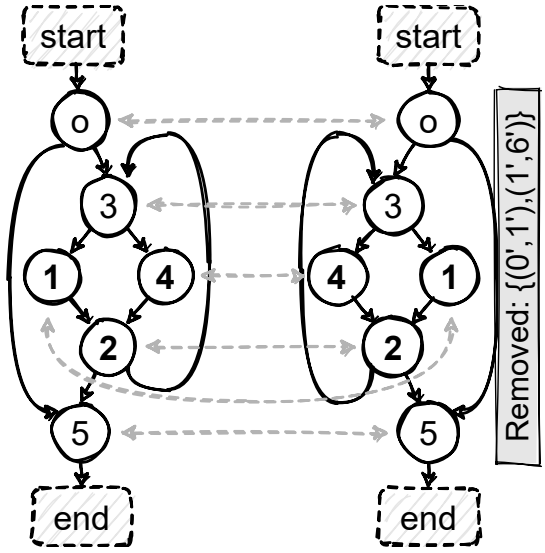


Figure 4.16 – *Direct mapping between two isomorphic CFG structures: The generation of the isomorphism between the MIR and the binary CFGs previously presented in Figure 4.14(b) allows defining a direct mapping between their nodes (or edges). The direct mapping relies on the updated IDs assigned to the nodes of both the graphs. This initial direct mapping and the annotation on the edges of the graphs are the base for the final mapping generation.*

The so generated MIR to binary CFGs mapping in addition to the ones for mapping IR to binary CFGs, generated in the first step, allows to accurately mapping the LLVM IR paths to their corresponding binary paths by first translating them into MIR paths.

4.5 Summary

This chapter presented a novel approach for mapping LLVM IR to binary CFGs. The problem definition and the discussion about the state of the art approaches for addressing this complicate issue have been initially discussed. Consequently, a tracing-based mapping solution has been presented as a fully automatic improvement of a previously defined mapping approach. The limitations of this extension are discussed before presenting the novel solution that takes advantage of the stronger points of the first solution. The proposed LLVM IR to binary CFGs mapping approach presented in this chapter is fully automatic and it is based on some internals of the LLVM compiler. The presented mapping approach proposes to tackle the hard problem by dividing the mapping generation in two distinct phases by implementing two different algorithms and consequently simplifying the problem. The first one produces an initial mapping between the IR and MIR labels of the basic blocks of two corresponding CFGS. The second one maps MIR to binary CFGs after the definition of an isomorphism between them. The two mappings together provide accurate IR to binary CFG mappings in support of host-based timing simulations based on the IR representation of a given input program.

CHAPTER 5

Efficient Performance Estimation via IR-Level Host-Based Simulation

The scope of this chapter is to present the timing simulation methodology at the base of this thesis. The explanation starts with a description and motivation about the kind of timing model that has been chosen for producing performance estimations via simulation. Thereafter, the proposed LLVM-IR context-sensitive timing simulation methodology is described in details. The description starts by describing a flexible simulation technique based on the interpretation of the program to analyze. Consequently, a faster simulation technique based on the JIT-compilation of the program is described. Both the techniques allow to evaluate multiple configurations in parallel in only one simulation. Consequently, it is presented an extension for the simulation methodology that enables the early evaluation of the performance of a heterogeneous system. Finally, the chapter ends by presenting a co-simulation between the proposed simulation methodology and the simulator natively provided by Simulink. This co-simulation technique enables the consideration of the timing effects due to the execution of the Simulink model on a real target platform.

5.1 Sources of Timing Variation

As already discussed in the previous sections, most of the available processors are not designed with a focus on their timing behavior analyzability. They are rather designed for maximizing the throughput of the instructions to execute relying on complex hardware resources and mechanisms. Therefore, the time requested for executing the same execution path on a given processor may vary depending on the internal state of such hardware resources. A key aspect for producing accurate timing estimations is to identify these mechanisms and to be able modeling their different behaviors [134]. Hereafter are discussed the commonly included hardware resources that are a source of timing variation [124].

Pipeline

The pipeline can be a complex hardware resource that is commonly included in all modern processors. The scope of this resource is to increase the average performance of a processor by increasing the utilization of all the other hardware resources and ensuring a higher instructions per cycle (IPC) value. Nevertheless, dynamic hazards can induce stalls on the pipeline stages that may cause a variation in its behavior [99]. Out of order execution pipelines are even harder to analyze. These kinds of pipelines try to reduce the number of stalls allowing the execution of the program instructions in a different order from the one defined by the compiler. A fetched instruction is directly executed as soon as the required input operands and the hardware resources are available without considering the instructions order.

The timing behavior of a pipeline is influenced by its size. Wrong predictions and stalls may cause different penalty cycles that depend on the number of stages. Furthermore, a larger pipeline allows a higher number of hazards that can be performed at the same time.

Branch Predictor

A branch prediction unit is a hardware resource that tries to improve the performance of a system by attempting the resolution of a branch instruction before its time. Every branch predictor utilizes a specific strategy that can be static or dynamic. In both of the cases, wrong branch prediction of not taken paths may have an impact on the timing behavior of the pipeline and consequently cause timing variation on the overall system behavior.

Floating Point Unit

Some of the floating point units included on modern processors can introduce variation on the execution time behavior of a program. In fact, in some cases, a hardware mechanism is implemented that allows to pipeline consecutive floating point instructions. This mechanism can be a source of stalls in the processor pipeline, especially in case of out of order pipelines. Furthermore, hardware manufacturers usually do not disclose timing information for all the kinds of instructions and especially for the floating point instructions (because unpredictable). For instance, ARM does not provide any timing information about the floating point instructions of two out-of-order processors like the Cortex-A9 and Cortex-A15 [115].

Cache Memories

Cache memories are commonly implemented for speeding up the execution of a program relying on the spatial locality and temporal locality properties of both program instructions and data. The capacity of a cache memory is limited and multiple different replacement policies are available. Furthermore, in some cases, processors include nested levels of cache memories.

Unfortunately, on one side the adoption of cache memories improves the average performance of a system but on the other side it reduces its timing analyzability. The realization of a valid cache model can be hard, especially in case of lack of essential data due to intellectual property restrictions. Cache memories are stateful resources and even small model's inaccuracies may invalidate the analysis results. Furthermore, cache memories are subjected to timing anomalies [134].

MMU and TLB

The translation lookaside buffer (TLB) is a small cache memory utilized by the memory management unit (MMU) for speeding up the translation of the virtual addresses. Powerful processors usually include an MMU. The time requested for translating a virtual address may vary depending on the number of operations requested for the translation. Bigger TLBs require less data replacement operations. Nevertheless, TLBs are affected by timing anomalies like the cache memories.

5.2 Context-Sensitive Timing Information

The list of hardware resources presented in Section 5.1 shows only the common available hardware resources that are source of timing variations. This list can be extended by considering vendor specific implementations. Nevertheless, all these hardware resources are stateful. All of them are hard to in-depth model because of their complexity or lack of information. As it will be discussed in the next paragraphs, an approach based on measurements is preferable for modeling their timing behavior at a context-level that is able to consider the program execution history. In this regard, the proposed simulation methodology utilizes a specific implementation of a so-called timing database [118, 119, 117] as source of timing information. Hereafter the base concepts for generating such timing database are introduced.

5.2.1 The Concept of Context

The execution time of a complete program depends on the execution time of all the instructions in the control flow path. Unfortunately, it is unrealistic to assign a fixed execution time for every instruction in the program. A performance estimation based on this assumption may result extremely inaccurate. In fact, the execution time of an instruction can vary depending on the program's execution history and its control flow [177]. Instructions in a loop are commonly subjected to this kind of timing behavior. For example, during the first loop iteration, the instructions can take longer for being executed compared to the consecutive iterations because the information needed is not cached [180, 156]. Therefore, a valid performance estimation approach considers different execution times for the same instruction depending on the visited control flow path. The way of control in a CFG is also known as context.

Contexts are generally defined at the basic blocks granularity because of the sequential execution of their instructions, from the first one until the last one without possibility of branching. A context defined at the CFG-level can consider only local loops. Differently, a program-level context can consider loops due to a sequence of function calls in the ICFG. When unrolled, program-level contexts are more suitable for describing the visited control flow path for reaching a specific basic block.

The concept of context was initially defined in [104, 37] as a support for the analysis of programs. Under the assumption that common programs spend most of their time in loops, contexts provide a formal way of describing this behavior. In the timing analysis domain, the concept of context has been successfully applied in WCET analysis tools [103, 36] as well as for producing accurate timing estimations via context-sensitive timing simulations [152, 119, 118]. In fact, the accuracy of the context-sensitive timing simulations is enabled by a timing granularity that does not require any direct modeling of complex hardware resources' behavior [21].

In the remaining parts of this section, some necessary notations are given before to present the context-based source of timing information selected for producing accurate performance estimations via simulation.

Call String

A call-graph-based definition for the concept of context relies on a finite abstraction of all the possible different executions of a basic block due to its previous call execution histories [168]. Multiple call strings can be utilized for describing all the different execution histories of a basic block. A formal definition for the concept of call string is formulated in [177]. Given a program P containing a set of functions $F = (f_0, f_1, ..., f_n)$ and a set of loops $L = (l_0, l_1, ..., l_n)$, for a basic block b a call string is a word C:

$$C : P \bigcup L \times \mathbb{N} \tag{5.1}$$

if b can be reached starting from the entry basic block of P, by calling all the functions in F, iterating through all the loops in L and by following their order of appearance in C. A more practical way of defining a call string is representing it as a string composed of all the traversed edges from the program start until the basic block execution.

Since the call graph is known, all the possible call strings for every basic block can be ideally statically computed. Unfortunately, this is not realistic. In fact, due to recursion or to unbounded loops, the set of call strings may not be finite. An example of an infinite set of call strings for a simple program is shown in Figure 5.1.

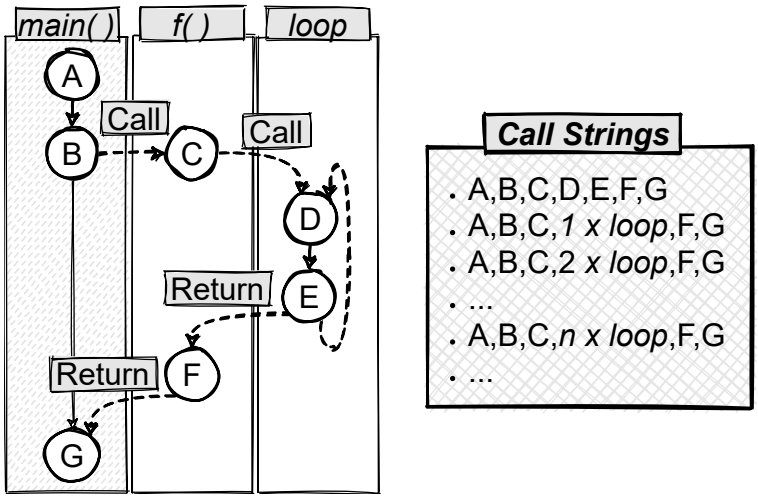


Figure 5.1 – *Possible infinite contexts of a program due to unbounded loop: The length of a call string for the represented ICFG of a simple program can be hypothetically infinite. If unbounded, the loop in the called function can be iterated an infinite number of times. As a consequence, the set of contexts required for describing the timing behavior of all its basic blocks may result infinite too.*

VIVU Mapping

The problem of having a number of infinite number of contexts due to the hypothetical infinite size of the call strings can be tackled by adapting an adequate technique that makes it systematically finite. In this regard, the Virtual Inlining and Virtual Unrolling (VIVU) mapping [104] is a function that enables the generation of a finite set of contexts for a given set of call strings. The VIVU mapping function makes it possible thanks to a well-defined set of operands that allows imposing bounds to any possible call string.

The $VIVU(n,k)$ is a function that, given in input a call string of any size, returns a certain set of contexts arranged depending on two numerical parameters. These two parameters are:

1. *Loop recursion count* - This first parameter, indicated as n, determines the maximum loop recursion count that has to be taken into account for a given call string.
2. *Number of elements* - This second parameter, indicated as k, limits the number of elements that have to be considered in a context.

By varying the values of these two parameters, the VIVU function determines the finite set of contexts that have to be considered for a given call string.

The values of the both the parameters n and k can vary between zero and infinite. The two parameters are independent. Varying the values of the parameters will change the number and the length of the contexts generated by the function. For example, the result of setting both of them to zero, $VIVU(0,0)$, determines the generation of only one context per basic block (or set of consecutive unconditional basic blocks) in the call string that does not depend on the previous execution history. On the opposite, setting both of them to infinite, $VIVU(\infty,\infty)$, imposes the generation of contexts that consider the complete previous execution history in the call string.

5.2.2 Implicit Modeling of the Hardware Timing Behavior

Similar to other previous works [118, 119, 117], the proposed methodology relies on the VIVU mapping for the generation of the essential timing model that is consulted while simulating for the production of the target system's performance estimations. Given an input software program to analyze, the key idea is to associate an execution tim to every context generated with a specific setting of the VIVU mapping function. The execution time associated to any context is extracted via measurements. These values implicitly model the timing behavior of parts of the program when executed on the target platform and considering the initial state of the hardware resources included on the target.

The next paragraphs describe the main activities required for the generation on a timing model based on the VIVU mapping.

Control Flow Reconstruction

One of the activities that is necessary for the generation of the timing model is the control flow reconstruction of the program considering its ICFG at the binary representation level. Unfortunately, differently from the LLVM code representations, which can be parsed and analyzed via the offered compiler's APIS, the precise reconstruction of an ICFG from the binary version of a program can be a hard task, especially for modern ISAs. Typical problems in CFG reconstructions include for example the memory indirections used to influence the control flow (jump tables, jumps depending on local variables, etc.), the ambiguous usage of machine instructions, the support of multiple ISAs and others [177]. For this reason, the binary ICFG is reconstructed with the support of the well-known and publicly available tool

`Radare2` [155]. This tool was initially developed for the purpose of reverse-engineering binary executables but nowadays it supports in different ways the analysis of different ISAs. Therefore, it can be trusted for accurate reconstructions of ICFG of even highly optimized binaries.

In order to fully respect the VIVU mapping requirements, and considering the ICFG structure of a program in a similar way to the one presented in Figure 5.1, the ICFG reconstructed by `Radare2` needs to be refined. The refinement requires the splitting of the basic blocks containing function call instructions. Therefore, in the final version of the reconstructed ICFG, every function call is treated as a basic block terminator instruction.

Traces Extractor

The timing information for the different contexts is extracted via non-intrusive tracing activities. The utilized tracer is the well-known professional tracing tool Lauterbach Trace32 [1]. This tracer supports multiple architectures implementing different ISAs, including ARM, x86, PowerPC, and others. A simple and common interface is provided for accessing the data in an architecture-independent way. The non-intrusive traces are extracted directly from the target without requiring any instrumentation of the binary program.

Relying on the target processor support, the Lauterbach Trace32 can extract precise execution time information for every basic block in the program. In fact, if properly set, starting from the first executed instruction in the program, the tracer can generate a timing packet containing an absolute time stamp every time a branch or a call instruction is executed in the program. A final time stamp is produced at the end of the program or when the tracer is interrupted. The different timing packets generated during the tracing can be consequently analyzed and processed for associating one execution time for every context in the program.

Eventually, and especially in case of the target processor is not yet available, the same tracing activities can be conducted via an external simulator (e.g., using a cycle accurate model). Even in this case, the simulation methodology is still beneficial because, with only partial tracing, it can be used for predicting the performance of the target system for non-traced input data ensuring a substantial speedup in the exploration process.

Timing Database Generation

The timing information previously extracted via tracing activities together with the structure of the reconstructed binary ICFG are used for generating a timing database (TDB). A timing database represents the concrete implementation of the timing model required by the simulation and initially introduced in Section 2.2.1. The key idea at the base of this timing model implementation is to store one execution time (expressed in cycles count) for every context generated applying a specific configuration of the VIVU function to the paths in the ICFG of the binary program representation.

Multiple measurements are taken by tracing the execution of the given software program and by varying the input data. Therefore, depending on the values of the n and k parameters of the VIVU mapping function, multiple execution times can be observed for a specific context causing a conflict. For instance, it is likely to observe multiple execution times for the same context setting both the parameters to zero. In case of a conflict, an execution time that considers all the different observed timing behaviors is selected. Instead of taking the minimum or maximum cycle count values, the methodology proposes to select the average value between them.

Another possible problem that might arise when generating a TDB is the lack of a value for a context because it has never been observed during the tracing activities. In this case, the problem can be tackled relying on the consideration that the contexts generated with a VIVU mapping enable the differentiation between loop iterations and the general stateful resources behavior (e.g. it is expected that, except for an initial phase, the execution time of an iteration of a loop is very similar to the execution time required by the previous one). Therefore, if the timing information for a context with recursion count n is not available at the end of the TDB generation, this value is computed with a sort of rollback computation. In this case, the timing information assigned to the context consists of the value previously assigned to the same context but generated with a recursion count value n', where $n' \leq n$ and n' is the closest recursion count value to n for which a timing value has been assigned to a context. If the context with recursion count n' does not exist, no timing information can be assigned to the context. A context without timing information represents a non-visited ICFG path during the tracing activities. Therefore, achieving a value of 100% code coverage [110] of the binary program representation while tracing ensures having, at least, one timing value for each context generated by the VIVU mapping function.

In addition to the tracing coverage level achieved while tracing, it is expected that the performance estimation accuracy strongly depends on the TDB generation. In particular, it is expected that the setting of the n and k parameters of the VIVU mapping function can influence the simulation results accuracy as well as its performance. For instance, by definition, a $VIVU(\infty, \infty)$ configuration should hypothetically lead to extremely accurate results. Unfortunately, this setting would require the simulation to consider full-length call strings. This implies two major consequences:

1. *Possible simulation slowdown* - Due to the continuous consideration of the substantial length of eventual call strings to query in the TDB.
2. *Unexpected inaccuracy* - Due to the infeasibility request of exact tracing of all the eventual call strings.

In a similar way, the configuration $VIVU(0,0)$ should lead to highly inaccurate results that are produced in a very fast simulation time. In fact, the limited size of the call string to manage during the simulation implies a minimal slowdown but the prediction cannot distinguish the different timing behavior of the basic blocks due to different contexts. A trade-off between the two parameters is consequently necessary for obtaining accurate results in an acceptable simulation time.

5.3 Simulation Methodology

The main purpose of the proposed simulation methodology is to reduce the actual limitations of the current context-sensitive timing simulations that are too coupled with a specific configuration (software, hardware and compiler optimizations). In particular, the methodology is intended to enable the analysis and the simulation of multiple configurations in support of the design exploration activities that are essential for the development of a system. The proposed new methodology evolves the former context-sensitive simulation techniques based on the software binary representation by moving the simulation to a higher level of abstraction that is represented by the IR code. The same IR code is shared between the different configurations and it allows making multiple timing considerations for different target platforms in only one simulation. This property ensures a substantial speedup in the simulation throughput. The consequent elevate simulation speed and the accurate mapping between IR and binary CFGs enable the production of accurate performance estimations that can also be used as an early feedback in the design space exploration of heterogeneous systems that consider complex MPSoC platforms.

In this section, two different simulation approaches are presented and both of them are based on the dynamic compilation of LLVM IR code. In particular, the first one is based on the simple but slow interpretation mode. The second one is based on its JIT-compilation instead. Both the simulation approaches are based on three interconnected main components shown in Figure 5.2 and that have been previously described:

1. *The dynamic compilation of LLVM IR code* - The interpretation or JIT-compilation of IR code on a host machine via `lli` (Section 3.2.3).
2. *The two-phase IR to binary CFGs mapping approach* - The proposed mapping approach that ensures accurate mapping between the IR and binary structure of a program (Section 4.4).
3. *The timing databse* - The selected timing model for implicitly modeling the timing behavior of a program considering its execution on a target processor (Section 5.2.2).

As shown in the figure, the three components collaborate for producing the final timing estimation by continuously updating it with relative execution time values. These values are provided by a TDB every time that a visited IR path, converted to a binary execution path

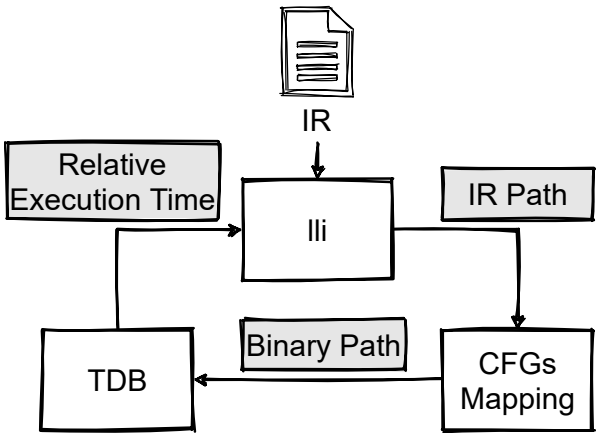


Figure 5.2 – *High-level simulation workflow: The simulation is based on three main interconnected components that determine the requested timing estimation. The simulation updates the final estimation by adding relative execution times that are provided by a TDB according to the simulated IR path that is translated to the corresponding binary path via the presented CFG mapping approach.*

utilizing the CFGs mapping, determines a context present in the TDB. This high-level workflow representation is common to both the simulation approaches. Furthermore, the elevated performance ensured by the second of the approaches allows to define a new methodology for producing early performance estimations of heterogeneous systems.

5.3.1 Interpretation-Based Context-Sensitive Timing Simulation

The first simulation approach is based on the interpretation of LLVM IR code via the `lli` tool. As described in Section 3.2.3, the interpretation is the slowest option between the available dynamic compilation strategies. Nevertheless, a simulator based on the interpretation enables easier debugging and easier interaction capabilities compared to others.

According to the proposed simulation concept described in Section 2.2, an ideal implementation of the simulation methodology should be composed of two different phases. An initial phase is dedicated to extract the information necessary to the simulation and it has to be executed only once per configuration. The consecutive phase instead allows running multiple fast simulations relying on the information previously extracted and considering different input data. Relying on this architectural design, in Figure 5.3 is presented the proposed workflow for executing interpretation-based context-sensitive timing simulations based on the LLVM IR code representation. The two phases and their components are described in details hereafter.

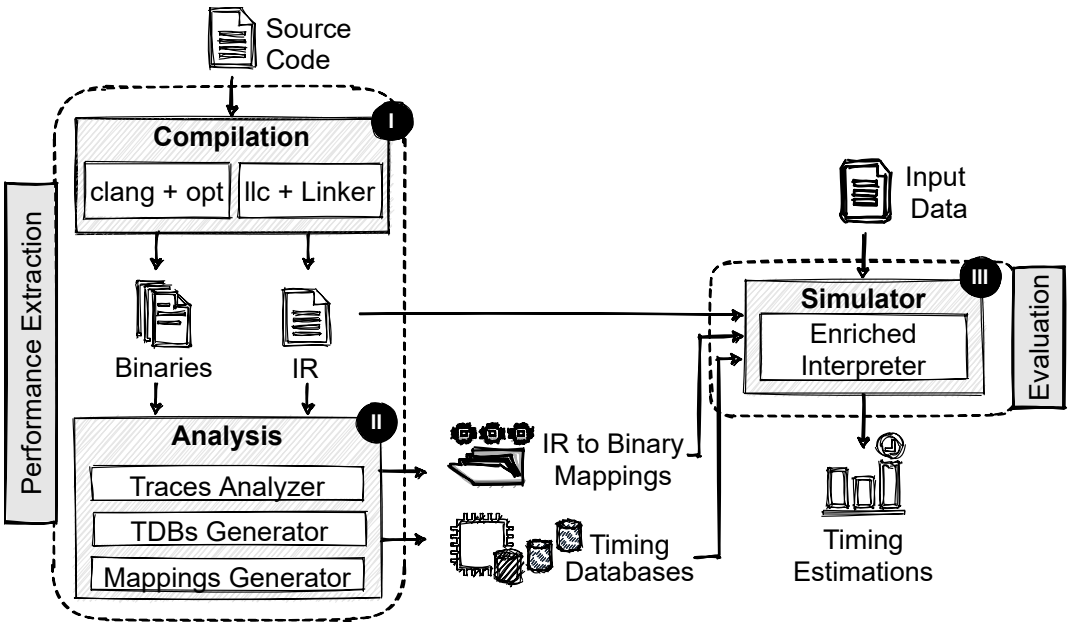


Figure 5.3 – *Interpretation-based simulation workflow: The workflow is composed of two phases. During the initial performance extraction phase, both the IR and binary representations of the source code are analyzed. The results of the analysis activities is one timing database and one IR to binary mapping for every required configuration to be analyzed. Both of them are given in input to the consequent evaluation phase in addition to the IR code and an initial input data. Based on these inputs, the simulator relying on the enriched interpreter of* `lli` *produces the timing estimations for all the requested configurations. Multiple simulations can be executed keeping the same inputs and only varying the program's input data.*

Performance Extraction Phase

The performance extraction phase starts with the compilation Ⓘ of the given source code that has to be analyzed. Multiple tools provided by the LLVM Compiler Infrastructure are used to generate both the IR and the binary representations of the input program. The considered IR version is the last architecture-independent version optimized via `opt`. It is the same version that is given in input to `llc` for applying the necessary architecture-dependent optimizations. One binary executable per configuration is generated according to the architecture-dependent compiler optimizations and target platform.

The workflow continues with the analysis phase Ⓘⓘ. The goal of this phase is the generation of all the necessary IR to binary CFG mappings and timing databases. The mappings are automatically generated by applying the two-phases algorithm previously described in Section 4.4. It is necessary to generate one mapping for every different configuration. Differently, the generation of the contexts-dependent TDBs requires initially conducting an appropriate activity of tracing. The execution traces are extracted directly from the target platform in a non-intrusive way and relying on the support provided by the Lauterbach Trace32 tracer. The tracing activities have to be repeated for all the configurations that are requested to be analyzed. Configuration per configuration, the traces are consequently analyzed for generating and populating the different TDBs. A specific setting of the VIVU mapping function drives the TDBs generation. As for the CFG mappings, it is requested to generate one timing database per configuration.

Evaluation Phase

The evaluation phase allows executing multiple rapid simulations Ⓘⓘⓘ for evaluating the performance of multiple configurations in parallel. Different timing behaviors of the target systems can be evaluated by varying the simulation input data sets. The simulator is based on the native implementation of the interpreter offered by the `lli` tool.

The proposed methodology requires to enrich the interpreter execution mode of `lli`. In this way, its execution is used for producing the required timing estimations. While compiling and executing the IR code, the extension forces the interpreter to consider the additional input information contained in the IR to binary CFG mappings and in the TDBs. The proposed modification forces the interpreter to query, at run-time, the given TDBs depending on the binary execution history of the program. Both the ISA and the addresses of the binary executed on the host-machine differ from the ones in the target binaries. A TDB is ISA- and addresses-dependent instead. Consequently, a mechanism for considering target-dependent call strings is required for accessing the TDBs. The proposed solution dynamically generates the binary call string by translating the visited IR paths depending on the previously generated CFG mappings. Therefore, every time a new IR basic block is accessed, at run-time the interpreter tries to translate the actual IR call string to the eventual corresponding binary call strings. If at least one binary call string is returned, the interpreter fetches from the appropriate TDBs the relative execution times useful for updating the simulation predictions. The steps performed by the enriched interpreter are summarized in Algorithm 4.

Algorithm 4 Enriched Interpreter Mode(bitcode, Mappings, TDBs).

```
1: terminate := false
2: timingEstimations[||TDBs||] := 0
3: contexts [||TDBs||] := ∅
4: while not terminate do
5:     instruction_IR := getNextIrInstruction(bitcode)
6:     executeInstruction( instruction_IR)
7:     if instruction_IR ∈ basicBlocksStartSet_IR then
8:         label_IR := getLabel(instruction_IR)
9:         for all TDB ∈ TDBs do
10:            basicBlocks_Bin[ ] := getMapped(label_IR, Mappings)
11:            for all basicBlock_Bin ∈ basicBlocks_Bin do
12:                contexts [TDB] := updateContext(basicBlock_Bin)
13:                relativeTime := queryTDB(TDB, contexts [TDB])
14:                increment(timingEstimations[TDB], relativeTime)
15:            end for
16:        end for
17:    end if
18:    if isLastInstruction(instruction_IR) then
19:        terminate := true
20:    end if
21: end while
22: return all timingEstimations
```

5.3.2 JIT-Based Context-Sensitive Timing Simulation

The proposed interpretation-based simulation approach suffers the slowdown due to the reduced speed of the IR code interpretation. The faster compilation option offered by `lli` is represented by its JIT compiler. Other simulation approaches substantially improved their simulation speed capabilities by taking advantage of the faster JIT-compilation possibility [135, 90]. Therefore, an evaluation has been conducted for quantifying the benefit of the JIT-compilation compared to the interpretation. The performance of the two `lli`'s execution modes have been compared by measuring the MIPS achieved by executing the complete Mälardalen benchmarks suite (this benchmark suite is later described in more details in Section 6.1). The evaluation results for the optimization level `-O0` are shown in Table 5.1. The results confirm that the JIT execution of the benchmarks ensures a substantial speedup compared with the interpretation. The average speedup ensured by the JIT-execution is close to 20 and it achieve a value of up to 52 for the `fac` benchmark. The achieved speedup is even larger when the benchmarks are compiled with higher level of optimizations. For example, the average speedup observed for the optimization level `-O3` is above 50. The results of this evaluation encourage for the definition of a different simulation strategy based on the JIT execution of the IR code.

The proposed JIT-based simulation approach follows the high-level simulation workflow described in Figure 5.2. At the same time, its workflow is similar to the one presented for the interpretation approach but it is focused in improving the simulation speed while keeping the same level of accuracy. The previously presented workflow has been consequently updated for taking advantage of the JIT-compiler capabilities. The updated workflow is shown in in Figure 5.4. Even if not explicitly marked, both the performance extraction phase and the evaluation phase are enriched but preserved. In fact, the former still requires the same compilation ❶ and analysis ❷ processes, and the simulation's concept ❹ in the latter remains the same. However, the JIT-based simulation does not require any modification to `lli`. The simulator can produce the requested timing estimations by simply relying on a well-defined

Table 5.1 – *Speedup ensured by JIT-executing IR code compared with interpretation.*

Benchmark	MIPS Interpreter	MIPS JIT	Speedup	Benchmark	MIPS Interpreter	MIPS JIT	Speedup
adpcm	9.1	28.4	**3**	janne_complex	13.5	355.1	**26**
bs	8.4	137.9	**16**	jfdctint	3.2	91.4	**29**
bsort100	8.5	135.1	**16**	lcdnum	12.4	172.3	**14**
cnt	11.2	429.6	**38**	matmult	8.3	86.7	**10**
compress	8	138.5	**17**	minver	6.5	45	**7**
cover	5.2	164.3	**32**	ndes	11	211.3	**19**
crc	7.5	158	**21**	ns	9	143.2	**16**
duff	11.3	145.3	**13**	nsichneu	4	13.4	**3**
edn	6.4	36.5	**6**	prime	10.2	108	**11**
expint	12.1	261	**22**	qsort_exam	8.3	63.7	**8**
fac	10	523.8	**52**	qurt	8.9	82.1	**9**
fdct	2	82.1	**41**	recursion	10	460.2	**46**
fft1	6.9	112.1	**16**	select	6.6	88.5	**13**
fibcall	13.5	181.2	**13**	sqrt	10.5	92.6	**9**
fir	9.9	338.8	**34**	statemate	4.8	30.7	**6**
insertsort	5.4	123.5	**23**	ud	13	102.1	**8**

annotation technique. An additional step is consequently required (III) for injecting in the simulation IR code the necessary annotation for enabling the JIT-compiler to produce the required performance estimations. The JIT annotation mechanism is presented hereafter.

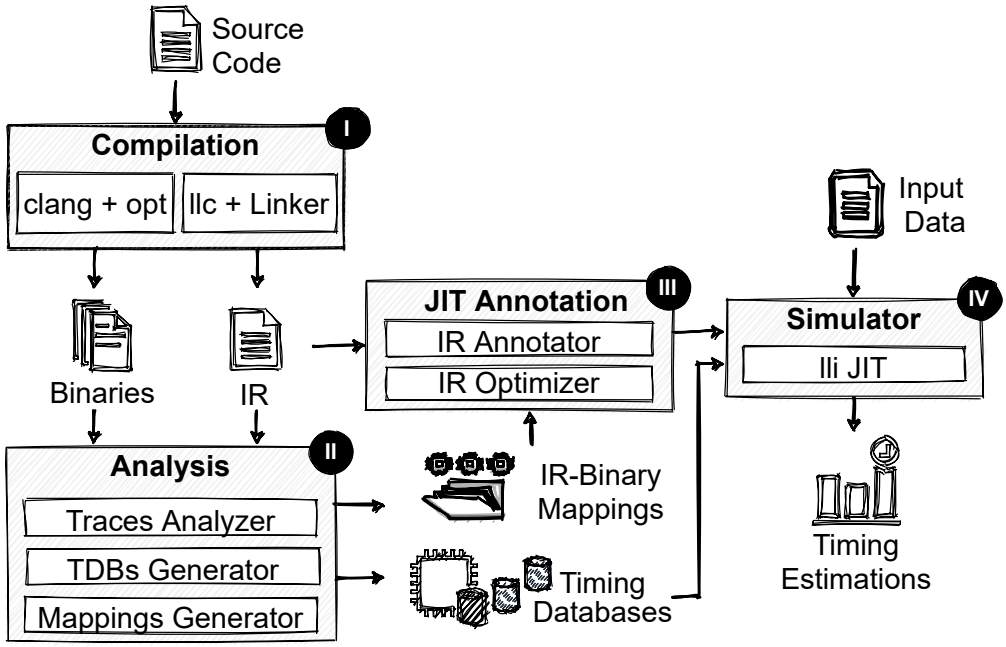


Figure 5.4 – *JIT-Based context-sensitive timing simulation methodology workflow: The JIT-based approach instead of proposing a modification for the interpreter, it simply requires the execution of an additional annotation step. In this step, the simulation bitcode is annotated with specific function call instructions that forces* `lli`*, at run-time, to update the requested performance estimations. Eventually, the annotated bitcode can be further optimized before being executed for improving the simulation speed.*

JIT Annotation

The scope of the JIT annotation component (III) in the workflow presented in Figure 5.4 is to enrich the IR simulation code with instructions that, at run-time, force the simulator to update the performance estimations depending on the executed IR code. As discussed in Section 3.2.2, the LLVM IR code is organized in compilation modules that collect the complete program information. Every function in a module contains at least one entry basic block followed by potential ones. The structure of a basic block is fixed and it is organized as shown in the left part of Figure 5.5. Each IR basic block is identified by a label and it contains order:

- *Phi-instructions* - A basic block can start with zero or multiple phi-instructions placed at the beginning for optimization reasons.
- *Instructions* - A sequence of zero or multiple instructions that define the overall basic block behavior.
- *Terminator* - A terminator instruction always conclude a basic block causing a branch or a termination (a function call instruction is not consider as a terminator).

In a similar way to the proposed algorithm presented for executing interpreter-based context-sensitive timing simulations, the JIT-annotation is intended to force the simulator to possibly update the timing estimations every time that the execution of an IR path determines a binary path translation. The methodology requires the annotation of only the IR code that is simulated. The version of the IR file that is annotated is a copy of the one produced during the compilation process. The bitcode utilized for producing the executable binaries and the CFG mappings remains unaltered.

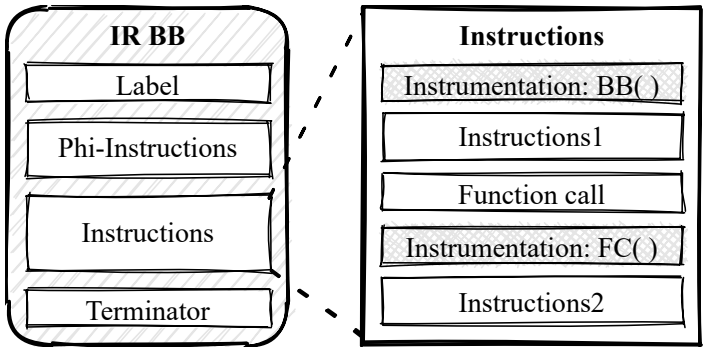


Figure 5.5 – *Proposed instrumentation strategy for enabling JIT-based simulations: Every IR basic block in the module contained in the simulation bitcode has to be instrumented. A function call instruction to an external custom IR library is inserted at the beginning of the beginning of the basic block or consecutively after any function call instruction. The instrumentation informs the simulator that a new IR basic block is accessed or that the execution of an IR basic block is resumed after returning from a function call.*

The annotation is performed by executing a custom pass via `opt` on the module contained in the bitcode. The pass iterate through all the basic blocks of all the functions in the module. A specific instrumentation function is inserted in the basic block depending on two specific conditions. The instrumentation consists of an IR function call instruction that forces `lli` to update the timing prediction values by querying the necessary timing databases. The function call executes some external code that is implemented in an IR library that has to be linked to the simulation code via the `llvm-link` tool. This code implements the mechanism for letting the simulator consulting both the CFG mappings and the TDBs. As shown in Figure 5.5,the two locations inside a basic block where the annotation has to be inserted are:

1. *At the basic block's beginning* - Immediately after eventual Phi-instruction for informing the simulator that a new basic block has been reached.
2. *After a function call instruction* - For informing the simulator that the execution of an IR basic block is resumed after that a function call returned.

The instrumentation function call instruction requires as single parameter the basic block's label. The algorithm for updating the timing estimations is completely implemented in the custom IR library. Depending on the visited IR paths during the bitcode execution, the library supports the simulator in translating the IR paths to binary paths relying on the CFGs mapping and consequently updating the performance estimations with the relative execution times extracted from the provided timing databases. The algorithm implemented in the IR custom library is very similar to the concept previously presented in Algorithm 4. The main difference consists in triggering the checks for a possible simulation performance update only when the instrumentation function is invoked. The simulation results are returned at the end of the program's execution.

5.3.3 Early Performance Estimation of Heterogeneous MPSoC

The proposed JIT-based context-sensitive simulation methodology is designed for producing highly accurate results in a very fast way. Furthermore, it is expected that the simulation speed will increase if multiple SoCs or configurations are evaluated in parallel. The JIT-compilation of the IR simulation code should determine a consistent speedup compared with the interpretation-based methodology. This property allows the definition of an extension for the presented JIT-based simulation workflow focused on early estimating the performance of heterogeneous systems. The resulting performance estimations can represent a valuable resource in the early design space exploration decisions. In fact, they can represent the feedback required by the system designers when evaluating the execution time of heterogeneous applications considering different MPSoC mappings for identifying the most suitable one.

Two different aspects have to be considered for enabling the simulation and the consequent evaluation of heterogeneous systems. The first aspect to consider is an extension required by the JIT annotation phase for forcing the simulator to behave differently. The second aspect requires the definition of a methodology for the generation of the TDBs in support of timing simulations of heterogeneous systems. These two aspects are separately analyzed in the following two sections.

JIT Annotation Considering a System Partition Scheme

In the JIT annotation phase ❸ of the workflow presented in Figure 5.4, it is assumed that a program has to be instrumented in a way that its performances are produced via simulation for the complete execution of a program. One or multiple configurations can be evaluated in parallel. Therefore, the simulation of the same IR control flow path implies the simultaneously querying of multiple given TDBs. Similarly, the heterogeneous extension requires the consideration of multiple TDBs during the simulation. However, instead of simultaneously querying all of them, the simulator dynamically selects the one designated for timing describing the specific part of the program. Therefore multiple TDBs are still considered during a complete simulation but every single part of a heterogeneous application can be simulated only considering the timing information stored in a specific TDB.

The idea of dynamically switching the source of timing information during the simulation according to a specific IR code annotation scheme is shown in Figure 5.6. An analyzable heterogeneous system is considered to be composed of multiple software functional units

(FU), which consist of collections of functions that can be mapped to the different processors included in an MPSoC. This mapping can be manually defined or, eventually, it can be exported from a Simulink model as later described in Section 5.4. The modification of the static annotation process is intended to drive the simulator's dynamic choice of the TDB to be queried while simulating. The choice has to be made depending on the given FUs mapping. This can be done by implementing a straightforward extension for the previously presented workflow. The annotation phase has to consider the additional information that describe a possible system partition scheme between the FUs and the execution processors. In fact, relying on a given system mapping partition scheme, the instrumentation statically determines which TDB has to be queried at run-time by the simulator. This simulation methodology enables the simulation of synchronous heterogeneous systems and it neglects the synchronization time between the processors.

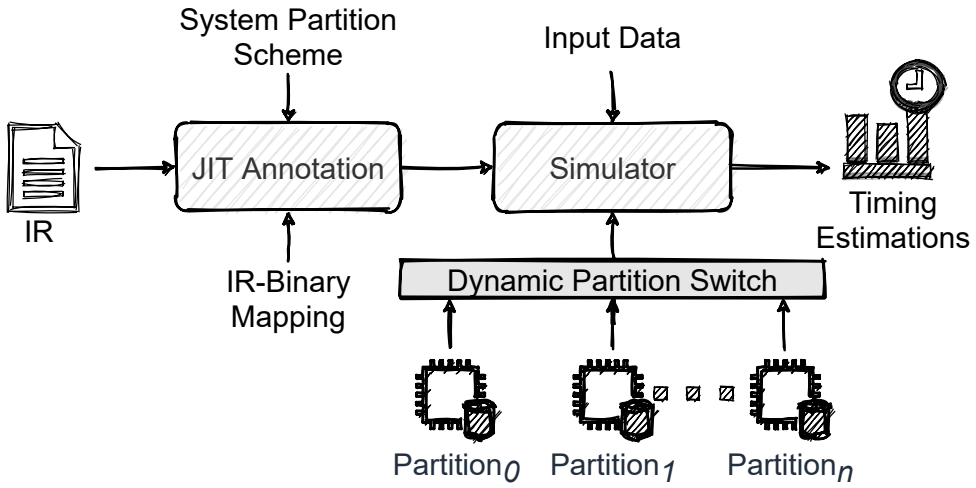


Figure 5.6 – *Simulation methodology extension for evaluating heterogeneous systems: Instead of accessing multiple TDBs in parallel, the performance of a heterogeneous system can be simulated by considering multiple TDBs but querying them only once per time depending on a dynamic partition switch mechanism. Considering the mapping scheme of the FUs of a heterogeneous application to the available processors in an MPSoC platform, the JIT annotation mechanism forces the simulator to dynamically switch between the TDBs.*

Generation of Timing Databases for MPSoC Simulations

The generation of a TDB designed for the simulation of a single core system was previously presented in Section 5.2.2. In this case, all the different FUs contained in the program are always executed on the same processor. Therefore, the timing information necessary for generating a TDB is extracted by tracing the complete program and considering different input data. The tracing process ignores eventual FUs structures and the extracted measurements are analyzed considering the complete program's ICFG. In the single core scenario, the execution of a specific FU influences the execution context of the following ones. In a heterogeneous scenario instead, where the FUs can be assigned to be executed on different processors included in an MPSoC, the execution of a FU cannot influence the execution context of any other FU that is consequently executed on a different processor. However, the single core scenario is still valid when considering only the execution of the FUs assigned to the same processor.

Multiple compilation techniques are available in the literature that tackle the problem of compiling heterogeneous applications for being efficiently executed on MPSoC platforms [23, 92, 49, 8, 140]. However, the choice of an efficient compilation technique is out of the

scope of this thesis. It is assumed that an application composed of multiple FUs is always entirely compiled for all the different ISAs implemented by the processors included in an MPSoC platform. This implies that a platform implementing n different ISAs requires the architecture-dependent compilation of the complete program for at least n times. Every compilation can be differently optimized. Therefore, as shown in Figure 5.7, it is requested to generate one TDB per partition in the system.

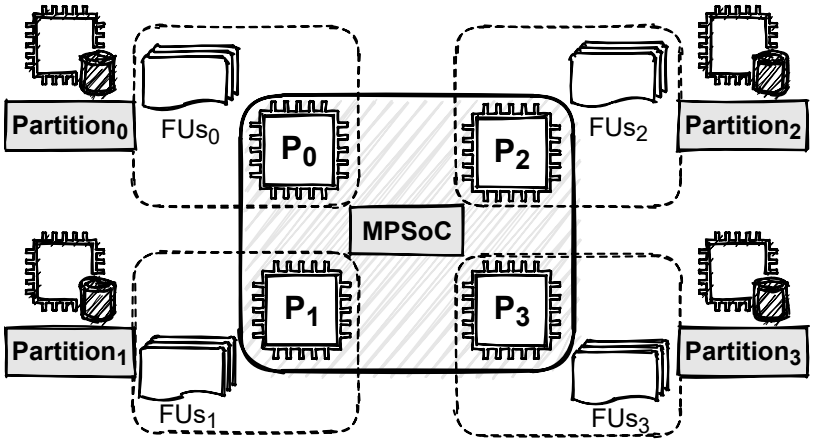


Figure 5.7 – *Multiple TDBs for describing the timing behavior of every MPSoC partition: The proposed simulation methodology for producing early performance estimations of a heterogeneous system requires the generation of one TDB per partition. Every single TDB describes the context-based timing behavior of a specific set of FUs when executed in one of the processors available in an MPSoC platform.*

The simulation results highly depend on the timing information contained in the considered TDBs. The heterogeneous simulation scenario offers the possibility of defining different ways for tracing the timing information required for the generation of the TDBs. Three different strategies has been identified. As shown in Figure 5.8, the three different identified tracing possibilities are:

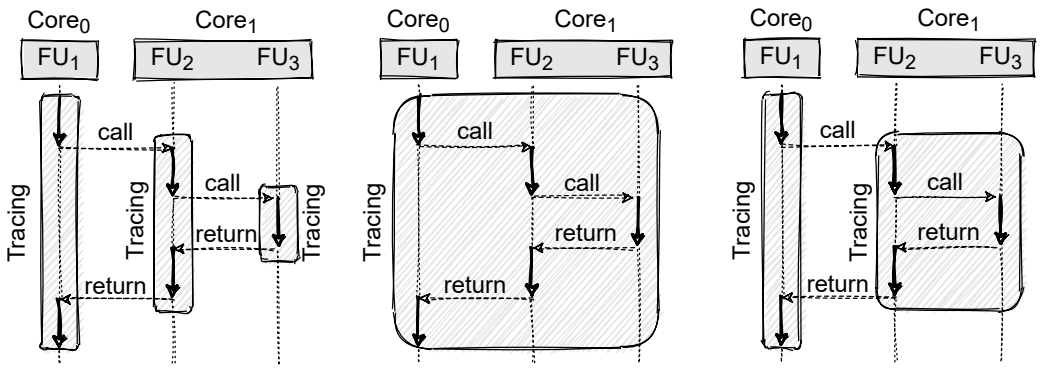
1. *Functions in isolation* - This option is the simpler possibility. It requires to systematically tracing every function in isolation by varying the input data sets. The execution of a function has to be traced on a given processor only if the function belongs to a FU that is mapped to the same processor. This option implies that the tracing activity always starts with an empty execution context. The tracing can be implemented with an appropriate Lauterbach script that forces the processor to start executing from the program directly from the first instruction of a given function. By changing the input values, it is possible to trace different execution paths. The execution program is stopped when the function returns.

2. *Complete program* - Independently from the partition scheme, this option requires the tracing of the complete program by varying the input data set. The tracing activity has to be repeated for every processor included in the MPSoC platform. This option represents a simple strategy that considers more realistic execution contexts. However, ignoring the information about the system partitioning may cause the observation of unrealistic timing behaviors for the contexts. This may be source of inaccuracies in the simulation results. The tracing activities that have to be executed for every processor included in the target platform are identical to the ones required for generation single core TDBs.

3. *Partitions in isolation* - A different way of producing the TDBs consists in considering the system partition's information. This information is used for statically identifying

the functions of the FUs in the binary. In a similar way to the first option, the consecutive functions in the FUs assigned to the same processor are traced. The tracing of consecutive functions can be ensured by forcing the program execution at the beginning of one of them and interrupting the execution at the end of the last return instruction. This choice allows the observation of the contexts due to the history of function calls inside the same FU. Compared to the other two options, the implementation of this tracing technique is more complex but, considering more effective contexts, it is expected to lead to more accurate results.

The possibility of tracing every function in isolation for every processor included in an MPSoC platform is less attractive than the other two options. The implemented tracing methodology is fully automatic and it supports all three the tracing strategies. However, tracing every function is isolation in every processor leads to loose essential context-dependent information. Therefore, in the evaluation section, only the second two possibilities will be considered.



(a) *Functions in isolation.* **(b)** *Complete program.* **(c)** *Partitions in isolation.*

Figure 5.8 – *Different tracing approaches for generating TDBs for MPSoC simulations: Three different approaches are proposed for generating the TDBs necessary for MPSoC simulations. The approaches differ in the tracing complexity and in the different ways of considering the function execution contexts. The choice of an appropriate tracing strategy is essential for ensuring accurate timing estimations via simulation.*

5.4 Timing-Aware Simulink Simulation

In the embedded systems domain, Simulink is a widely utilized model-based development tool because of the possibility of re-utilization of tested reliable components and especially because of its simulation capabilities. Fast functional simulations can be executed since the early stages of the development of a system. Regrettably, as discussed in Section 3.1.2, these simulations ignore any non-functional property of a system. This limitation makes Simulink a non-suitable tool for supporting the design-space exploration activities. In fact, at the current time, it is impossible to evaluate different hardware/software configurations. Considering this significant limitation of the Simulink simulations, the present section describes a novel approach, based on the previously presented LLVM IR context-sensitive timing simulation methodology, for considering specific non-functional information while natively simulating a Simulink model. Furthermore, relying on the MPSoC extension presented in Section 5.3.3, a Simulink model can be eventually simulated by considering its execution as a heterogeneous system. The proposed methodology is fully automatic and it mainly focused on the performance behavior aspect of a system. However, it can be straightforwardly ported to any different non-functional property.

The fully automatic high-level simulation workflow is shown in Figure 5.9. Given an input Simulink model, a co-simulation [50] is executed between Simulink and the proposed LLVM IR context-sensitive simulation methodology. This is enabled by generating a functionally equivalent model via annotation. The resulting annotation implements the mechanism for letting Simulink to consider the architecture-dependent performance effects for the model that are generated via timing simulation. Finally, the performance and the behavior of the given model can be visualized directly on Simulink.

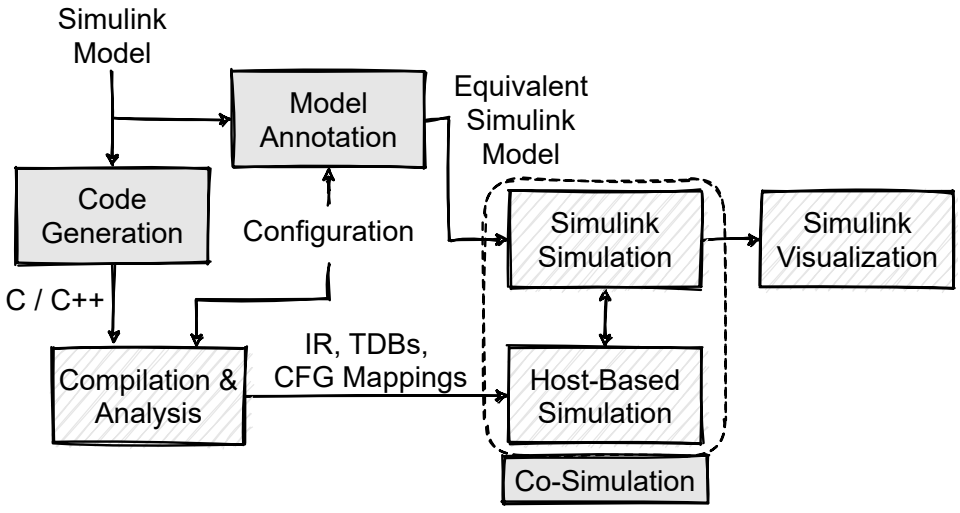


Figure 5.9 – *Timing-aware Simulink simulation methodology workflow: A given Simulink model can be annotated with custom library blocks producing a functionally-equivalent model whose native simulation enables the execution of a co-simulation together with the proposed LLVM IR context-sensitive timing simulation approach. Therefore, before running a co-simulation, the model is appropriately translated into source code and consequently compiled. The co-simulation between the functionally equivalent Simulink model and the IR context-sensitive simulation, based on the information produced during the analysis, allows visualizing the effects of the non-functional properties on the system's behavior directly on the Simulink environment.*

5.4.1 Code Generation

One of the key aspects in the proposed co-simulation methodology consists in mapping the components of a given Simulink model to the structure of the binaries that can be generated from its compilation. This mapping is essential for enabling the co-simulation between Simulink and the proposed context-sensitive timing methodology. The compilation of a Simulink model requires first its translation into a high-level programming language, such as C or C++. The proposed fully automatic methodology relies on this translation for implicitly generating the necessary mapping.

A Simulink model is commonly structured in hierarchically interconnected components that belong to one of two possible main subsystems that interact with external sensors. The plant is one of the two available subsystems and it represents the physical world interacting with the system. The controller is the other available subsystem and it represents the software part of a model that is expected to be executed on a target platform. The controller is commonly designed to process the data that is periodically produced by both the plant and sampled by the external sensors. Therefore, the final binary executable compiled for the target system includes only the code regarding the controller subsystem.

The proposed methodology enables the automatic generation of C code for the controller subsystem of a Simulink model relying on the Embedded Coder toolbox [164]. After identifying the controller subsystem, the code generation process produces automatically the source code that has to be compiled for being executed on a target platform. This code does not contain any additional or superfluous instruction concerning the Simulink simulation. This property is ensured by properly configuring the Embedded Coder settings.

Simulink Components to Source Code Implicit Mapping

A mapping between two different code representations can be usually produced by annotating one of the two representations. However, this can be a hard task. A simpler and implicit way of mapping is chosen for enabling the proposed co-simulation technique. In fact, the Embedded Coder toolbox can offer more than the support for ensuring the generation of source code suitable for being executed on a target platform. It represents a precious support for defining an implicit mapping between the different components that are part of the controller subsystem and the resulting source code structure. The implicit mapping consists of generating one C function for every component in the controller subsystem. Every function is called as the corresponding Simulink block. This property can be ensured by properly setting, in an automatic way, the Embedded Coder and the model settings.

The Embedded Coder TLC has to be programmatically configured. The configuration's goal is to produce C code whose structure simplifies the task of mapping Simulink components to the corresponding C code. All the possible code generation optimization options configurable via the TLC are disabled. The implicit mapping technique requires the C code to reflect the original structure of the components in the controller subsystem. Optimizations such as blocks merging, blocks removal and other must be consequently disabled. Furthermore, every block in the controller component has to be properly configured. The configuration requires the reduction of eventual algebraic loops and the generation of one C function per component by managing each of them as an atomic unit. Finally, the TLC has to be accurately set for ensuring that every generated C function is named in the same way as its corresponding Simulink component. In this way, every function in the source code is distinguishable by the Simulink block name.

Source Code Structure

The source code is composed of a number of functions that is at least equal to the number of components in the Simulink model, including the controller subsystem. Additional functions can be arbitrarily included by the Embedded Coder and function calls to external libraries may occur. The source code of the C function corresponding to the controller subsystem component is the one responsible for iteratively scheduling the consequent function calls according to the connection between the components defined in the Simulink model. If properly configured, the Embedded Coder produces function calls without parameters. Eventual inputs and outputs between the Simulink components are managed via shared C variables between the functions in the source code. This mechanism is helpful in case of the inputs of a component are outputs of multiple different components.

The SRP allows the Simulink simulations to simulate, if possible, multiple components in parallel. In fact, a component can be simulated as soon as all its inputs are given. This is unrealistic. The different functions of a program are sequentially executed according to the program's call graph and starting the execution from the function generated for the controller subsystem. Internally, Simulink defines a fixed execution order between the components that can be simulated in parallel. This order is reflected in the function calls order of the generated source code. Furthermore, the Embedded Coder generates the source code in a way that a function can call is inserted in the code of a function only if the caller is in the Simulink model a component that contains the sub-component corresponding to the callee. Therefore, only a subsystem component can activate the execution of its directly contained components. For example, in Figure 5.10 it is shown the sequence diagram for the given exemplary model. The execution of such a model starts from the function called `Controller` that represents the implementation of the controller subsystem. This function invokes, in order, the functions `A` and `B` because the first one has a higher priority than the second one. The function called `Subsystem` can only be executed after the completion of the previous two functions because their outputs represent the function's inputs. The function `C` can only be invoked by the function implementing its closest subsystem. The execution of the function `Controller` terminates only after the control returns from function `Subsystem`. The implementation of the source code structure is essential for the next activities.

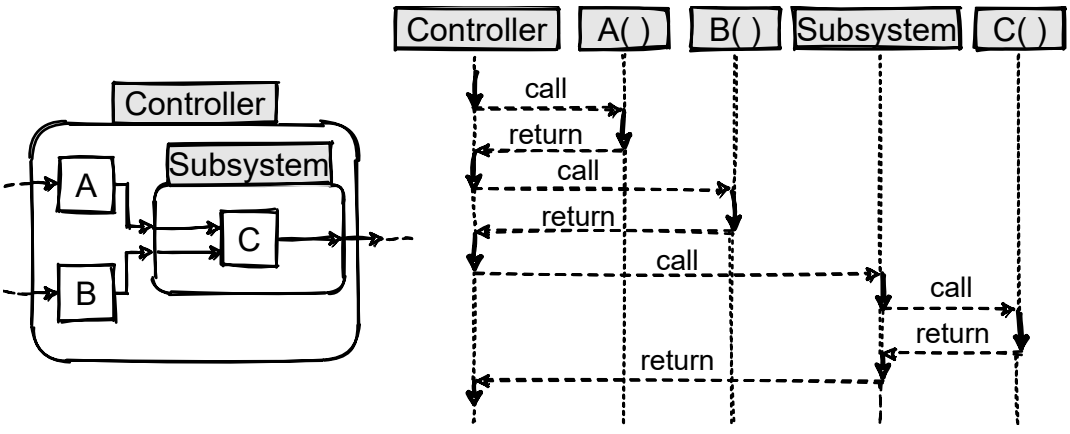


Figure 5.10 – *Exemplary sequence diagram of a simple Simulink model: The execution of the generated source code starts always from the implementation of the controller subsystem. Thereafter, the functions implementing the different components are invoked by following the program call graph based on the original Simulink model components order and hierarchy. Function calls can be performed only by components of subsystem type and they can invoke only components belonging the same subsystem hierarchy.*

5.4.2 Model Annotation

The model annotation step has two different objectives. The first one is the definition of the system configuration for the design exploration choice that has to be simulated. The second one is the generation of a functionally equivalent model to the one given in input. This model enables the later presented co-simulation technique. This section presents hereafter the methodologies proposed for satisfying these two objectives.

System Configuration Definition

A mechanism has been defined and implemented for annotating a given model. The annotation is intended for specifying the selected design configuration that has to be evaluated via simulation. Therefore, the annotation process requires the following information:

1. *Name of the controller subsystem* - The name of this component is essential. In fact, it identifies the Simulink subsystem that has to be analyzed. The code generation process considers only the components contained in this subsystem. This component has to be explicitly indicated because it does not exist an automatic way for recognizing this component. Furthermore, this information is forwarded to the later stages.
2. *Target platform* - A target platform including one or more processors has to be specified. This platform is consequently considered during the compilation of the source code and during the complete process for enabling the required timing simulation.
3. *System partition* - In case of a heterogeneous system, a partitioning map has to be provided. This mapping specifies, for every subsystem contained in the controller component of the Simulink model, which is its designated execution processor between the ones included in an MPSoC platform. Otherwise, in case of a single core system, this information can be omitted.
4. *Target configuration* - For every processor included in the target platform, it has to specify the configuration of its physical resources (e.g. caches enabled/disabled, branch predictor enabled/disabled, etc.).
5. *Compiler optimizations* - The optimization level to apply at compile time to the source code generated for the controller subsystem.

Given this information, the first model's annotation phase can be executed. Initially, a new Simulink model is created. The new model is an identical copy of the given input one. From now on, all the steps utilizes the new version of the model and the structure of the original one is preserved. Thereafter, the new model and its different components are annotated according to the given specification. The annotated information is later processed by the consequent activities required for the preparation of the proposed co-simulation methodology.

Functionally Equivalent Timing-Aware Model

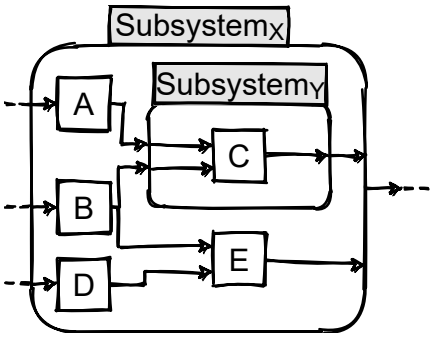
The goal of the second step of the model annotation phase is to enrich the Simulink model for enabling the co-simulation between Simulink and the proposed context-sensitive timing methodology. Similarly to other approaches [114, 84], the base idea of the annotation process consists of adding appropriate components to the model that mimic the timing behavior of the target system during the native Simulink simulation. Therefore, the model is automatically enriched with the insertion of stateful delay blocks connected at the outputs of the different components in the model. These blocks permits the Simulink simulation to consider the timing effects depending on the target hardware platform implementation and configuration. Consequently, an enriched Simulink model represents the translation of an architecture independent model into an architecture-specific one.

The delay block component that is added during the annotation is a custom block that can be implemented as a Simulink library component. If connected to the outputs of a given component, the delay block forces the input data to be available for the consequent components only after a certain amount of simulation time. After this delay, the input signal is forwarded at the output of the delay block and it is unchanged. The effects of a delay block are useful in overcoming the unrealistic zero or fixed-computation time assumed by the SRP Simulink simulation, that were previously presented in Section 3.1.2. The delay imposed by such new components represents the execution time of the associated block in a target embedded platform. The amount of time that a block has to be delayed cannot be constant. This value depends on the context of the program's execution history. It is dynamically determined during the co-simulation by considering the timing estimation produced executing an LLVM IR context-sensitive simulation. In this way, timing considerations are taken into account by influencing the simulation but preserving its semantic.

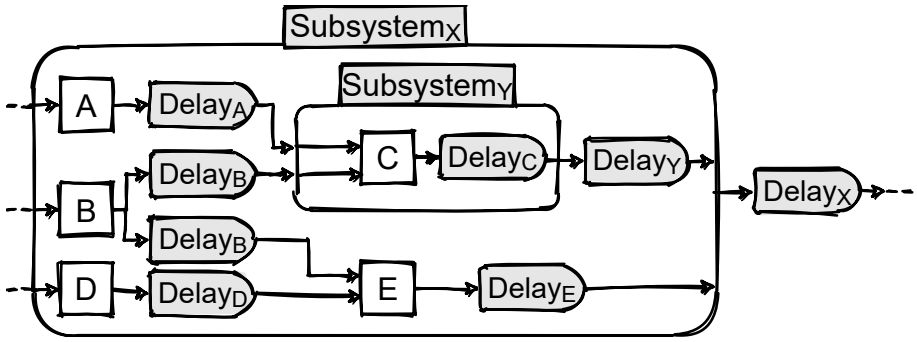
This annotation step enriches the given Simulink model by inserting one delay block at every output connection of every component that is part of the controller subsystem. In Figure 5.11 is shown an example for the annotation process. The annotation of the simple controller subsystem in Figure 5.11(a) produces the functionally equivalent subsystem represented in Figure 5.11(b). The timing delays imposed by such blocks are only simulated. They do not require to effectively stop the simulation for the given amount of time. In fact, the simulation itself is not slowed down and the delay proposed mechanism shows only a minor overhead in the time required for executing a simulation. Furthermore, the functionality of the given input model is fully preserved. In fact, the simulation of a model annotated with delay blocks considering null delays leads to identical results of the simulation of the original model.

5.4.3 Co-Simulation Methodology

The proposed co-simulation methodology workflow for enabling the possibility of evaluating a Simulink model considering the timing effects due to its effective execution on a target platform was previously shown in Figure 5.9. The goal of the methodology is to make possible a realistic and timing-aware evaluation of a model by directly simulating it in Simulink. Therefore, the proposed workflow requires the arrangements of all the necessary aspects for executing two simulators. One of the two simulators involved is the one provided by Simulink. The semantics and the dynamic behavior of the delay blocks added during the annotation allows considering the effects of the target specific timing in the Simulink simulation of an enriched model. An enriched model contains all the information needed for preparing the execution of an LLVM IR context-sensitive timing simulation. Initially, the source code of the controller subsystem is automatically generated. The source code is consequently compiled according to the given configuration annotated in the model. Thereafter the analysis



(a) *Original controller subsystem to be simulated.*



(b) *Functionally equivalent annotated controller subsystem for timing-aware simulation.*

Figure 5.11 – *Exemplary delay block annotation of a Simulink subsystem: The proposed annotation methodology is essential for enabling the possibility of considering the timing effects due to the execution of a model on a target platform during a Simulink simulation. A custom delay library block has to be inserted at the output connections of all the components contained in a controller subsystem, like the one in (a). The native simulation of the resulting functionally equivalent model, like the one in (b), allows the evaluation of the target system directly on Simulink.*

of the bitcode and the timing traces, extracted from the target, allows the generation of the necessary IR to binary CFG mapping and the TDBs. All the produced information represents the input of the co-simulation.

Multiple fast simulations can be executed for evaluating different scenarios. In fact, different configurations and settings of the plant subsystem and of the input sensors can be evaluated by simply modifying the model and running a new co-simulation. The value of the simulation time to consider by the different delay blocks is dynamically updated by executing a new context-sensitive timing simulation. The proposed workflow is fully automatic and it allows evaluating the simulation results directly on Simulink. The proposed methodology for enabling the co-simulation of the two simulators is presented hereafter.

Performance Estimations for Timing-Aware Simulink Components

The co-simulation between Simulink and the LLVM IR context-sensitive timing estimation methodology presented in this thesis is based on a communication mechanism between the two simulators. The communication between them is event-driven and it can be triggered by two kinds of events:

1. *New controller iteration* - Starting from its first execution, every time the simulation requires iterating over the controller subsystem.

2. *Execution of a delay block* - Every time a component that is part of the controller subsystem is executed and the generation of its outputs determines the execution of a delay block.

In the former case, the Simulink simulation has to notify the timing simulation of the values generated by the plant and sampled by the sensors. For every iteration, this value represents the controller inputs and consequently the input data set for the timing simulation. In the latter case, a component in the controller subsystem has been simulated and its outputs have been computed. Therefore, a performance estimation is expected from the delay block for postponing the availability of the outputs by additionally simulating the timing behavior of the component.

The proposed technique for enabling the communication between the two simulators is based on a socket mechanism. A high-level overview of this technique is shown in Figure 5.12. The technique proposes to open a socket [163] between MATLAB and the well-known debugging tool GDB [148]. Furthermore, the JIT-execution of bitcode can be debugged via GDB [157]. Consequently, the socket mechanism together with GDB represent the connection bridge between the two simulators.

The co-simulation mechanism requires that only one simulator per time can be in the running state. The co-simulation starts by executing the timing simulator first. A breakpoint is set via GDB at the initial instruction of the function generated for the controller subsystem. The value of the variables representing the first sampling of the sensors are programmatically extracted from Simulink. These values are consequently imported in the timing simulation by initializing the appropriate variables via GDB. This initialization also relies on the implicit mapping naming-based mechanism utilized for the model's components. Thereafter, the timing simulator is started and it is stopped only after the program reaches the initial breakpoint. The stop event of the timing simulator determines the beginning of the Simulink simulation. Every time that the Simulink simulation triggers a new event, its simulation is stopped. If the event is of the first type, MATLAB notifies GDB the state of the plant represented by the value of the sampling sensors. These values are used by GDB to update the corresponding variables in the program before returning the control to Simulink. Otherwise, if the event is of the second type, Simulink notifies MATLAB the name of the block that caused the event's

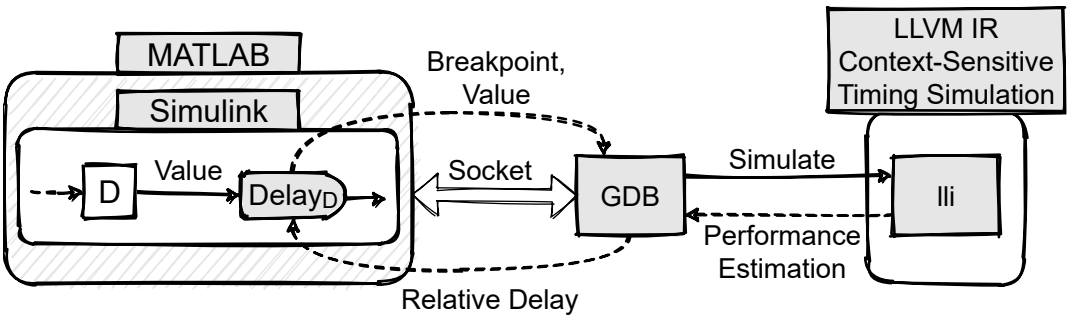


Figure 5.12 – *Co-simulation mechanism for evaluating timing-aware Simulink models: The simulation of an enriched Simulink model can postpone the availability of the outputs of the different components by relying on the custom delay block library. The delay amount can be determined relying on the performance estimations generated via simulation adopting the proposed LLVM IR context-sensitive timing simulation methodology. The two simulators can communicate by managing a socket between MATLAB and GDB. The simulation of a delay block implies GDB to continue executing the timing simulation until the return of the function identified by the delay block is reached. The consequent performance estimation is used to set the relative value of the delay block that last invoked GDB.*

triggering. This name is forwarded to GDB and it is used for timing simulating the program until the symbol identified by the function name returns. The performance estimation is forwarded to MATLAB and a relative delay time is set for the delay block library that last triggered the event. Thereafter, the Simulink simulation can continue. The process continues until the end of the Simulink simulation.

At the end of the simulation, the performance of the model considering a given input configuration can be assessed directly on Simulink. In fact, the delay blocks utilized for enforcing the model to consider the architecture-dependent timing effects preserve the model semantic and functionality. Multiple consequent co-simulation can be executed by varying the configuration of the plant or of the sensors.

5.5 Summary

This chapter presented the overall simulation methodology that is at the base of this thesis. Initially, the chosen context-sensitive technique for generating the timing models considered during the simulation has been presented. Thereafter, the simulation methodology has been described. The description started with the definition of a simpler interpretation-based simulation methodology. Thereafter, an improvement has been proposed for ensuring higher simulation speed capabilities by relying on a JIT-based compilation mechanism. Furthermore, it is discussed the possibility of producing early approximate estimations for the execution time of synchronous heterogeneous systems. For this purpose, the bitcode annotation's mechanism previously proposed for enabling the possibility of JIT-based simulation can be appropriately adapted. In this regard, the chapter proposes different ways of tracing for generating the necessary timing models. Finally, a co-simulation methodology is presented for enabling the Simulink simulations to consider the timing effects due to the execution of the software on a target platform.

CHAPTER 6

Experimental Evaluation and Results

The scope of this chapter is to present the experimental results obtained during the evaluation of the proposed simulation methodology. After an initial description of the evaluation setup, the effectiveness of the LLVM IR to binary CFGs mapping approach is assessed. Thereafter, the simulation speed and the results accuracy are investigated by considering multiple target platforms that include different ARM processors. The evaluation considers also the possibility of evaluating multiple target processors in parallel and it investigates the eventuality of generating early performance estimations for heterogeneous systems. Finally, an exemplary Simulink model is used for showing its different behavior when forced to consider the target-dependent timing effects that are generated via the proposed simulation methodology.

6.1 Evaluation Setup

The present section is intended to describe the environment utilized for evaluating the presented simulation methodology. In particular, in the first part, it is given a short description about the selected evaluation benchmarks. Thereafter, some technical details are given about the simulation host machine and about the analyzed and simulated ARM-based target platforms.

Benchmarks

It is hard to obtain real industrial applications or to get the permission for publishing their analysis results. For this reason, the number of available benchmark suites is large and they are focused in covering multiple benchmarking purposes [64, 73]. For the evaluation of the proposed simulation methodology, it has been decided to utilize the widely-used Mälardalen benchmark suite [54]. This suite is specifically designed for the timing analysis domain. In fact, these benchmarks include a wide and representative set of program constructs that can be found in industrial applications and their code is mainly focused on flow analysis.

From the list of the available benchmarks, those that introduce non-available library sources have been excluded from the evaluation. In fact, the excluded benchmarks may introduce inaccuracies that are not caused by the mapping algorithm's accuracy or by the timing simulation approach.

Both the compilation and the simulation are based on tools of the LLVM Compiler Infrastructure version 9.0. These tools have been all compiled and built enabling the `release` option that is specifically designed for achieving better performance. During the experiments, the benchmarks have been compiled with different optimization levels. If not explicitly specified, the benchmarks have to be intended to be compiled with `-O3`, the highest level of optimization. This choice is intended for observing eventual properties that are common in real industrial scenarios instead of analyzing unoptimized code.

Simulation Environment

The complete evaluation has been conducted on the same common host machine. For reproducibility reasons and for making the results comparable between them and with other approaches (especially when talking of simulation speed expressed in MIPS) some details about the host environment are necessary. The utilized host machine is a Linux machine that includes an Intel Core i7-2600K processor running at 3.4GHz supported by 32GB DDR3 RAM memory.

Evaluated and Simulated Platforms

The different stages of the evaluation have been conducted by considering four different ARM-based target platforms. These platforms include different processor versions of the Cortex family and they implement in total three different ISAs. In Table 6.1 are listed the names of the considered target platforms and. For each platform, it is given the name of the processor implementation followed by a list reporting some of its major architectural details. The table shows that, except for the Cortex-M4 processor, the target processors selected for the evaluation of the proposed methodology implement complex architectures that are hard to model and to analyze. For example, the processors implements mechanisms such as out of order speculative execution, different dynamic branch prediction mechanisms, second-level set-associative caches, lock-step execution mode (only the SoC including the Cortex-R5 processor) and others.

The platforms' setup is similar between them. For all of them, the execution mode has been arranged to be bare-metal. Both the instructions and data of the programs have been placed on the available external memories of each platform. In this way, the execution of the evaluation programs utilizes all the provided caches as well as the branch predictor mechanisms that are enabled before the start of the programs.

Table 6.1 – *List of considered ARM-based platforms and their architectural properties.*

	Evaluated SoCs			
Platform	Xilinx Zynq-7000 [181]	TI EVM-K2E [65]	TI RM57LHDK [66]	Hitex LPC4350 [62]
Processor	ARM Cortex-A9	ARM Cortex-A15	ARM Cortex-R5	ARM Cortex-M4
ISA	ARMv7-A	ARMv7-A	ARMv7-R	ARMv7E-M
Frequency	400 MHz	100 MHz	20 MHz	120 MHz
L1 D-Cache	32 KB	32 KB	32 KB	No
L1 D-Cache	32 KB	32 KB	32 KB	No
L2 Cache	512 KB	4 MB	No	No
Pipeline Stages	8	15	8	3
Floating Point	Yes	Yes	Yes	No
Out-of-Order	Yes	Yes	No	No
Superscalar	Yes	Yes	No	No
Speculative Execution	Yes	Yes	Yes	No
Lock-step	No	No	Yes	No
NEON Extension	Yes	Yes	Yes	No

6.2 Simulation Accuracy

A first objective in the evaluation of the proposed methodology consists in determining its accuracy. As previously discussed in Section 2.1.2, an ideal simulation approach should produce highly-accurate performance estimations in a very fast way. Considering the proposed methodology simulation, its accuracy mainly depends on the:

1. *CFGs mapping approach* - The quality of the proposed LLVM IR to binary CFGs mapping approach.
2. *Timing consideration while simulating* - The quality of the timing model queried while simulating for generating the requested target performance estimations.

These two quality assessment objectives are independent from the kind of simulation (interpreter-based or JIT-based) that is utilized for generating the results. In fact, the overall results show identical performance estimations when analyzing the same target system configurations. Therefore, the accuracy results presented in this section can be achieved indistinguishably utilizing any of the two kinds of proposed simulation approaches. Differently, the results that are shown in this section consider as target only the ARM processor Cortex-A15 included in the TI EVM-K2E [65]. This processor represents the most complex option in the evaluation setup. In fact, it implements the most complex architecture and hardware features between the processors listed in Table 6.1.

6.2.1 LLVM IR to Binary CFGs Mapping

This section presents the results of the experiments conducted for evaluating the validity of the proposed two-phases algorithm for mapping LLVM IR to binary CFGs and previously presented in Section 4.4. Unfortunately, a direct validation of the mapping accuracy is not feasible. Considering different input data, this would require to validate every visited control flow path in the ICFG during the execution of the program on the target platform with the ones visited via simulation. In other words, for every experiment, it would be necessary to perform a string matching between the call graph observed while tracing the execution of the program on the target platform with the one observed while simulating. Therefore, a different validation methodology has been selected. The mapping accuracy has been assessed relying on an indirect approach. The selected approach consists of measuring the number of executed instructions and the program execution time. Consequently, the indirect approach continues by comparing them to the respective results of a simulation based on the proposed mapping approach. The two measured metrics strongly depend on the executed control flow. Therefore, in an indirect way, a high level of accuracy in the simulated values indicates a precise mapping of the CFGs.

Before starting the evaluation, the binary file is statically analyzed. The goal of this analysis step is to determine a fixed number of instructions for every binary basic block in the program. As previously discussed in Section 3.3.1, it is assumed that the execution of a basic block implies the consecutive execution of all its instructions starting from its start address. As mentioned in Section 5.2.2, any function call is treated as a basic block terminator instruction. However, as explained in Section 5.2.1 the execution time of a basic block cannot be fixed. It may vary depending on the program's execution history due to the possible different timing behavior of the stateful resources included in the processor. Therefore, multiple execution times have been consequently measured via non-intrusive tracing measurements for every binary basic block. The different measurements allow accounting for the variation caused by the different execution contexts. These measurements are consequently used for producing a

TDB generated applying the $VIVU(\infty,\infty)$ configuration. In case of fully traced call strings, this choice enables the generation of exact (or highly accurate) results.

The evaluation has been conducted by requiring the simulator to generate the selected two metrics by updating their values, at run-time, and by relying on the IR execution paths and the mapping information. All the possible combinations of standard middle-end and back-end optimization levels have been considered. Between them, the highest level of optimization `-O3` allows producing more challenging CFGs to be mapped. In fact,compared with lower optimization levels, the `-O3` optimization level introduces substantial changes in the structure of the different program representations and is therefore harder to map. For most of the functions in the benchmarks, a manual direct matching of IR to binary CFGs is not possible. For this reason, only the results for the highest level of optimization are shown. More accurate results, achieved evaluating lower optimization levels, are here omitted because they are a direct consequence of a lower mapping complexity.

The results for the conducted evaluation of benchmarks compiled with `-O3` optimization level are collected in Table 6.2. For every benchmark in the table are reported two additional metrics useful for describing the complexity of the contained CFGs. In particular, the complexity of the CFGs is expressed by the benchmark's total lines of code (LOC) and the cyclomatic complexity number (CCN) [108] of its most complex CFG. The CNN indicates the complexity of a CFG by considering the number of linearly independent paths contained in it. Based on its definition, the CCN can describe the complexity of a CFG according to the following classification:

- *Easy structure* - $1 \leq \text{CCN} \leq 5$.
- *Difficult structure* - $6 \leq \text{CCN} \leq 10$.
- *Very difficult structure* - $11 \leq \text{CCN} \leq 15$.
- *Structure impossible to be tested* - $\text{CCN} \geq 16$.

The results in Table 6.2 show a high-level of accuracy for both the metrics. In fact, the difference between the measured and simulated values is minimal. In most of the cases, the accuracy is close to 100%. The CFGs of simple programs are exactly mapped and minimal inaccuracies are observed only for programs that include CFGs with difficult or harder structures. The high level of accuracy of the simulation results implies the valid effectiveness of the proposed two-phases mapping algorithm.

Consequently, after an appropriate investigation, it has been ascertained that one reason for the deviation in the results is that the ARM instruction set includes conditional execution instructions. As previously discussed in Section 3.3.1, this kind of instructions are not considered by the proposed methodology because they partially invalidate the assumption on the complete execution of a basic block. Nevertheless, the effectiveness of the mapping is proven by the accuracy of the simulation results. Therefore, the proposed mapping algorithm can be utilized for generating accurate mapping between the CFGs in support of the presented simulation methodology.

Table 6.2 – *Evaluated mapping's accuracy for benchmarks compiled with `-O3` optimization level.*

Benchmark	Metrics		Instructions Count			Execution Time (us)		
	LOC	CCN	Measured	Simulated	Accuracy	Measured	Simulated	Accuracy
bs	144	9	51	51	**100%**	7	7	**100%**
bsort100	128	8	45949	44286	**96.38%**	2380	2301	**96.68%**
cnt	267	3	1525	1525	**100%**	165	165	**100%**
crc	128	9	11661	11661	**100%**	635	635	**100%**
duff	86	10	537	537	**100%**	37	37	**100%**
edn	285	4	84992	84301	**99.19%**	5989	5945	**99.27%**
fdct	239	3	1834	1834	**100%**	159	159	**100%**
fir	276	5	133588	133588	**100%**	9270	9340	**99.24%**
insertsort	92	8	1108	1083	**97.74%**	92	90	**97.83%**
ludcmp	147	14	1512	1506	**99.61%**	235	231	**98.30%**
matmult	163	4	35229	35229	**100%**	3015	3040	**99.17%**
ndes	231	11	28625	28566	**99.79%**	1502	1489	**99.13%**
prime	47	4	4233	4223	**99.76%**	449	448	**99.78%**
qsort_exam	121	15	851	867	**98.12%**	79	76	**96.23%**
qurt	166	5	514	514	**100%**	135	135	**100%**
select	144	16	368	368	**100%**	34	34	**100%**
sqrt	77	5	447	447	**100%**	124	124	**100%**
st	177	4	70067	70067	**100%**	4548	4518	**99.34%**
ud	163	11	1205	1161	**96.35%**	109	106	**97.24%**

6.2.2 Context-Sensitive Timing Simulation

This section presents the evaluation results obtained investigating the achievable level of accuracy in the performance estimations produced via the proposed LLVM IR context-sensitive simulation methodology. In particular, the results initially show the importance of considering different timing behaviors for specific parts of a program depending on the execution contexts. Thereafter, the accuracy of the simulation results is assessed. In the complete evaluation, the accuracy is always computed comparing the value of the performance estimated via simulation with the ones measured directly from the target. Therefore, the accuracy of the results (expressed in percentage) can be computed utilizing the following formula:

$$Accuracy\ \% = \frac{estimated}{measured} \times 100$$

Relying on this definition, the percentage of error in the simulation results can be straightforwardly computed as:

$$Error\ \% = 100 - accuracy$$

The overall results of the evaluation are summarized in Table 6.3. These results are expressed in error percentage. Considering that every program can be simulated by giving in input different data sets, multiple experiments are conducted for every single program and here it is shown only the maximum observed value of error percentage. A low value of error percentage implies that the performance estimation value produced via simulation is very similar to the one measured directly on the target. Therefore, a zero value in the table implies an exact result.

Table 6.3 – *Evaluation accuracy considering three different VIVU mapping configurations.*

	Error %		
Benchmark	**VIVU(0,0)**	**VIVU(20,20)**	**VIVU(∞,∞)**
bs	5.08	3.16	0
bsort100	26.7	3.32	0.86
cnt	52.31	0.5	0
crc	23.82	1.06	0
duff	6.07	0.03	0
edn	36.54	3.80	0.72
fdct	3.56	0.08	0
fir	29.42	1.92	0.76
insertsort	19.98	4.65	2.17
ludcmp	25.45	2.98	2.70
matmult	5.22	1.44	0.83
ndes	42.10	1.23	0.87
prime	44.47	1.12	0.32
qsort_exam	39.12	6.20	4.77
qurt	31.63	0.13	0
select	14.97	0.70	0
sqrt	31.54	0.18	0
st	32.69	1.92	0.66
ud	43.37	3.54	2.76
Average	**27.01**	**1.99**	**0.92**

The Importance of Context-Sensitive Timing Information

The first column of Table 6.3 is intended for showing the necessity of considering different execution times for the same part of the program depending on its execution context. In fact, the column shows the percentage of error for the performance estimations produced by simulating the programs and considering TDBs that have been generated with the $VIVU(0,0)$ mapping configuration. This configuration does not consider the possible different timing behaviors of the software programs depending on their execution contexts. During the TDBs generation process, only one time per binary basic block (or sequence of consecutive basic blocks) is computed and stored in the appropriate TDB. This value is the average value between the observed ones.

The same values reported in the table are graphically represented in the chart shown in Figure 6.1. Except for some small benchmarks, the error percentage in the performance estimations that do not consider the execution contexts is substantial. The average error value is close to 27%, an unacceptable level of inaccuracy. In the conducted evaluation, the highest value of inaccuracy has been over 52% when simulating the `crc` benchmark. In this specific case, the poor performance is due to multiple context-dependent loops in the benchmark's ICFG that are executed during the simulation. Simpler benchmarks including less and simpler loop structures, like in the case of the `bs` and the `matmult` benchmarks, the inaccuracy drops to 5%. Therefore, the consideration of an appropriate TDB when executing a context-sensitive timing simulation is essential for the accurate analysis of complex programs like the ones that compose industrial and real applications.

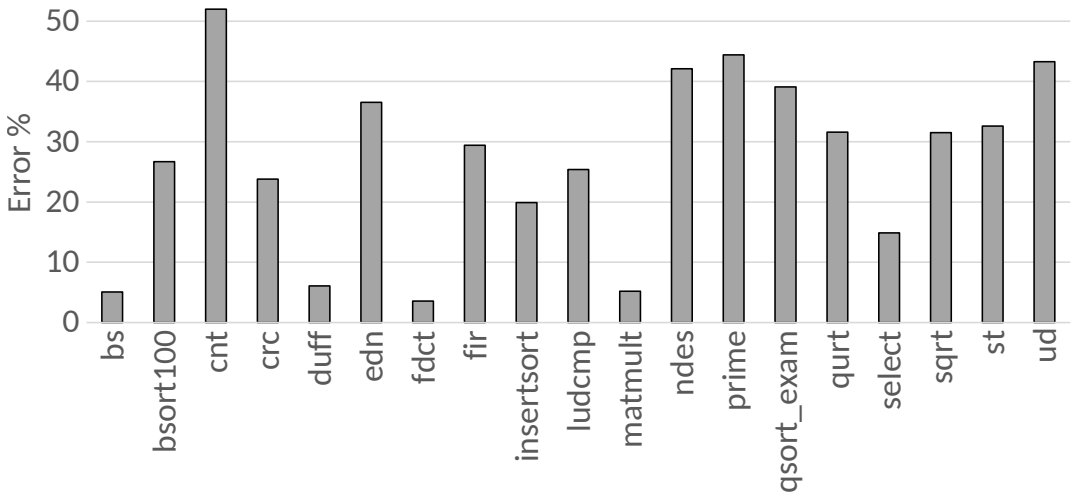


Figure 6.1 – *Inaccuracy due to missing context timing information: The chart shows the percentage of error in the performance estimations of multiple Mälardalen benchmarks produced via simulation (the values are extracted from Table 6.3). In all the different simulations, the resulting timing estimations have been produced by considering TDBs generated with the $VIVU(0,0)$ configuration. The missing consideration of the possible different timing behaviors due to the execution contexts in the programs is recognized to be the cause of such inaccuracy.*

Simulation Accuracy Assessment

Ensuring an elevated level of accuracy for the performance estimations produced via simulation is one of the two key requirements of an ideal solution, as previously discussed in Section 2.1.2. Therefore, further experiments have been conducted for assessing the accuracy capabilities of the proposed simulation methodology.

The accuracy capabilities have been initially assessed by executing context-sensitive timing simulations of the benchmarks and relying on the timing information contained in TDBs generated with the $VIVU(\infty,\infty)$ configuration. It is expected that this configuration of the VIVU mapping can lead to observing the most accurate simulation results. This is possible only if the conducted tracing activities ensure the measurement of an execution time for all the encountered contexts during a simulation. The results of this evaluation are reported in the most right column of Table 6.3. The same results are plotted with light gray bars in the chart shown in Figure 6.2. As expected, the $VIVU(\infty,\infty)$ configuration leads to more precise results than the ones observed for the $VIVU(0,0)$ configuration. For eight of the shown benchmarks the performance estimations resulted always exact. The average error value is below 1%. Furthermore, the overall error percentage is lower than 3%, except for the `qsort_exam` result that is slightly below 5%. The higher inaccuracy encountered for the `qsort_exam` benchmark is due to an insufficient quality contexts coverage during the program tracing. The benchmark includes multiple nested loops that make the process of visiting different control flow input paths hard. In fact, this is done by only changing the program's input data. This problem is present also for the other evaluated VIVU configuration. Therefore, it is evident that the tracing strategy is essential in determining the maximum achievable simulation accuracy.

The presented results for the $VIVU(\infty,\infty)$ configuration show that an elevated level of accuracy can be achieved in estimating the performance of a system via simulation. However, this configuration requires the simulator to manage very long call strings, eventually unbounded. It is expected that this property influences the simulation performance introducing an undesired slowdown. Therefore, more experiments have been conducted for identifying

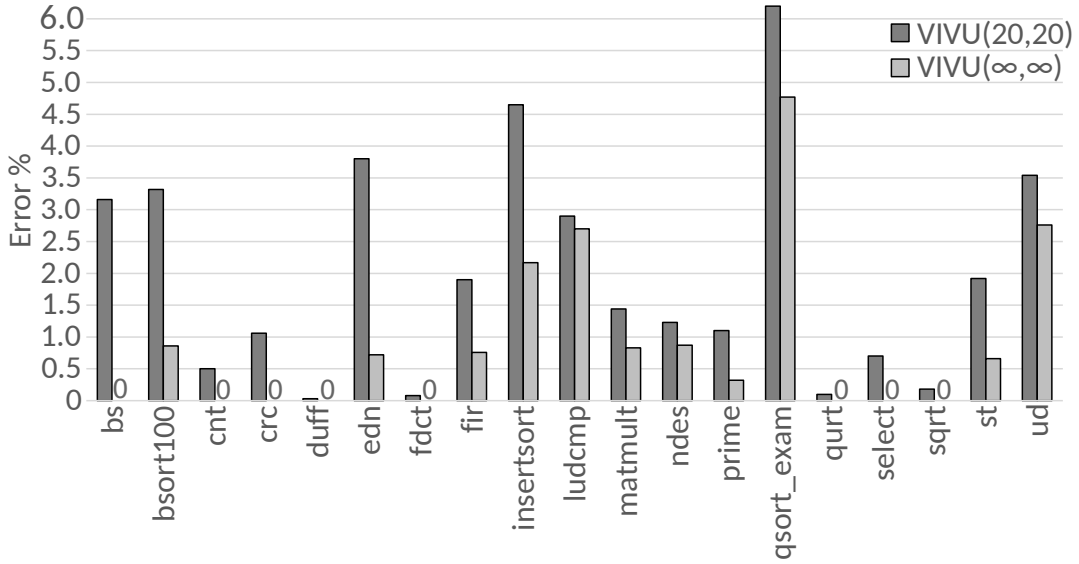


Figure 6.2 – *Elevated accuracy of LLVM IR context-sensitive timing simulations: Compared to the results shown in Figure 6.1, the consideration of context-dependent different timing behaviors implies a significant level of accuracy in the performance estimations produced via simulation. In particular, the chart shows the limited percentage of error in the performance estimations based on the* $VIVU(20,20)$ *and* $VIVU(\infty,\infty)$ *configurations (the values are extracted from Table 6.3). The results show that the performance estimations produced with TDBs generated with the* $VIVU(20,20)$ *configuration can achieve accurate results. The accuracy of these results is comparable to the ones based on the* $VIVU(\infty,\infty)$ *configuration if appropriate tracing activities are performed during the TDB generation.*

a VIVU mapping configuration that reduces the overhead by keeping an adequate level of accuracy. An iterative process has been followed that, starting from the less accurate configuration $VIVU(0,0)$, increased both the VIVU mapping function's n and k parameters until the simulation accuracy stopped substantially improving for most of the benchmarks. The conducted experiments identified the $VIVU(20,20)$ configuration as the first configuration that can ensure good accuracy results. The error percentage in the performance estimations produced via simulation and considering TDBs generated applying the $VIVU(20,20)$ configuration are reported on the central column in Table 6.3. Comparing these values with the ones produced with the configuration $VIVU(\infty,\infty)$, as shown in the chart in Figure 6.2, it is possible to observe a similar behavior for the two configurations. As expected, the results produced considering the $VIVU(20,20)$ configuration are less accurate than the ones produced considering potential unbounded call strings. In some cases, the difference between the accuracy of the two estimations differ. The observed average error percentage is below 2%. For multiple benchmarks, the accuracy is comparable with the most accurate VIVU configuration. In a similar way to what observed for the $VIVU(\infty,\infty)$ configuration, the highest level of inaccuracy has been observed for the `qsort_exam` benchmark. In this case, the error percentage is close to 6%. The difficulties encountered in the tracing of the previous experiment remain valid also for this case.

In general, the conducted experiments show that the proposed context-sensitive timing simulation methodology can achieve a good level of accuracy. Depending on the simulation requirements, an appropriate configuration of the VIVU mapping function parameters has to be defined. The presented results show that the $VIVU(20,20)$ configuration can support the simulation in achieving sufficiently accurate results and reducing the simulation overhead.

6.2.3 Early Evaluation of MPSoC

The previous sections showed the results of the evaluation conducted for estimating the level of accuracy of the performance estimations produced via simulations of single core systems. Differently, this section is intended to show the accuracy level that can be achieved when early estimating the performance of a heterogeneous system utilizing the MPSoC simulation methodology extension presented in Section 5.3.3.

The MPSoC extension evaluation has been conducted in a different way from the single core accuracy assessment. In fact, the unavailability of an MPSoC platform and the choice of neglecting any heterogeneous compilation technique, because it is out of the scope of this thesis, suggested the evaluation to be conducted via a practical example. Therefore, in the evaluation it is considered a hypothetical AMP asynchronous system composed of four interconnected processors. The selected four processors are the ARM Cortex processors described in Table 6.1.

The system's software application has been created by combining multiple different benchmarks extracted from the Mälaralen suite. These benchmarks have been grouped in four different software functional units: FU_0, FU_1, FU_2, FU_3. Following a system partition scheme, every processor in the heterogeneous system is designated to execute only one of the available FUs. Inside every single functional unit, it has been implemented a minimal scheduling mechanism that invokes, in order, one after the other all the benchmarks in the partition. The design of the system defines FU_0 as the first executed functional unit. Thereafter, after the first execution of FU_0 and relying on the support of the scheduling mechanisms, the functional units are executed in the order FU_1, FU_2, FU_3, FU_0. The scheduling mechanism in FU_0 determines the system termination after one thousand iterations. The system design and partition information is summarized in Table 6.4.

The so designed heterogeneous application has been consequently compiled. As requested by the methodology, the application has been compiled for all four the processors in the hypothetical MPSoC platform. Four different binary executables have been generated from the same IR code. Thereafter, the simulation bitcode has been appropriately annotated for enabling the possibility of relying on the JIT-base simulation technique. During the annotation, it has been followed the information provided by the partition scheme shown in Table 6.4. As a consequence, during the JIT-simulation, the simulator dynamically determines which of the TDBs has to be queried for updating the timing estimation regarding the code of a specific FU.

Table 6.4 – *Evaluated heterogeneous system design and its relevant details.*

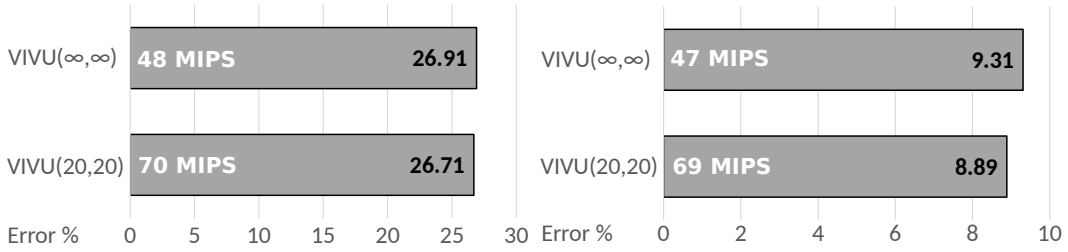
Functional Units	SoC	Processor	Benchmarks	LOC	CCN
FU_1	TI EVM-K2E	Cortex-A15	bsort100	128	8
			cnt	267	3
			crc	128	9
FU_2	TI RM57LHDK	Cortex-R5	fac	27	2
			fdct	239	3
			fft1	219	16
FU_3	Hitex LPC4350	Cortex-M4	fibcall	72	2
			insertsort	92	8
			janne_complex	64	6
FU_4	Xilinx Zynq-7000	Cortex-A9	matmult	163	4
			minver	201	21
			ns	535	9

The simulation requires in input four different TDBs, one for each processor in the hypothetical MPSoC. Three different possibilities are presented in Section 5.3.3. One of them, the possibility of generating a TDB considering the tracing of every function in the program in isolation, is not evaluated. In fact, this solution does not scale for large systems and it is consequently excluded from the evaluation. However, both the other two possibilities are considered. Therefore, for every processor have been generated two different TDBs. The first one is generated by following the procedure defined for the so-called *Complete program* technique, and the second one is generated by tracing only the appropriate partitions per core as described for the *Partitions in isolation* technique.

The evaluation is based on the comparison of the performance estimation results produced via simulation with a reference target execution. This reference time has been measured via a structured tracing activity. Relying on the Lauterbach Trace32 support, the tracing has been conducted processor per processor in a way of respecting the local execution contexts of the different FUs.

The evaluation goal is assessing the accuracy of the simulation results considering both the possibilities of generating a TDB. For every possibility, two different VIVU mapping configurations have been utilized for generating the simulation TDBs. According to the previously conducted evaluation, the TDBs have been generated by considering the $VIVU(20,20)$ and the $VIVU(\infty,\infty)$ configurations. The results of the complete evaluation are presented in Figure 6.3. In particular, the chart in Figure 6.3(a) shows the accuracy achieved by considering TDBs generated according to the *Complete program* technique. In this case, the error percentage considering both the VIVU configurations is similar and it is slightly below 27%. Differently from the previous case, and as shown in the chart in Figure 6.3(b), the accuracy substantially increases by simulating the system and considering TDBs generating according to the *Partitions in isolation* technique. For both the VIVU configurations, the error percentage drops below 10%. Furthermore, the simulation accuracy resulting from generating the TDBs according to the $VIVU(20,20)$ configuration achieved above 90%. The simulation speed observed during the simulations executed during this evaluation is aligned with the results later shown in the appropriate evaluation section.

The accuracy demonstrated by the two techniques proposed for generating the TDBs in support of the simulation of a heterogeneous system substantially differs. The first possibility



(a) *Complete program tracing technique.* **(b)** *Single partitions in isolation tracing technique.*

Figure 6.3 – *Evaluation accuracy of MPSoC extension methodology results: The two proposed tracing techniques for tracing and extracting the necessary timing information for generating the TDBs necessary for simulating a heterogeneous system have been evaluated. The experimental results show that the complete program tracing technique is not suitable for producing sufficiently accurate results, reporting a percentage of error in the performance estimation close to 27%, because of missing or misleading context-timing information. More precise results can be achieved adopting the single partitions in isolation tracing technique. In this way, useful indications can be produced for the early stages of the development of a system. The performance estimations produced relying on the timing information extracted via the second technique showed an accuracy above 90%.*

results easier to trace, it requires a simple end-to-end tracing, but it showed high inaccuracy due to inappropriate timing values assigned to some contexts. The inaccuracy level is objectively too high even for early performance estimations. The results of the second possibility showed a significant accuracy improvement due to a more time consuming tracing procedure that allows the observation of more realistic execution contexts. A percentage of error below 10% for an early estimation of a heterogeneous system can still be considered an appropriate indication for the designers of a system during the early stages.

6.3 Simulation Speed

Producing highly accurate performance estimations is only one of the two requirements of an ideal simulation solution. In fact, it is requested that the estimations have to be produced in a very fast way ensuring a substantial simulation speed. The goal of this section is to present the conducted evaluation for assessing the simulation speed capabilities of the proposed simulation methodology. Initially, it is evaluated the interpretation-based simulation technique. Consequently, the performance of the equivalent accurate JIT-based simulation technique is assessed and the performance of the two techniques are compared. Thereafter, the simulation speed of the fastest technique is compared with the well-known and widely used gem5 simulator [14, 96]. Finally, the beneficial effects on the simulation speed capabilities due to the possibility of evaluating multiple SoCs in parallel are assessed and presented.

6.3.1 Interpretation-Based

The interpretation-based simulation technique is the more flexible one between the proposed two solutions. It offers easier debugging and interaction capabilities but it is expected to be the slowest one because of the slowdown due to the code's interpretation. The conducted evaluation mainly focused on determining the simulation speed capabilities of this technique and on estimating the overhead in the `lli` execution due to the consideration of timing information while interpreting the bitcode. The evaluation has been conducted considering all the four processors and SoCs listed in Table 6.1 but here are shown only the results obtained simulating the ARM Cortex-A15 processor. As expected, the simulation speed capabilities observed by simulating the other processors showed irrelevant differences. In fact, the simulation speed, as well as the simulation accuracy, does not depend on the target architecture but on the configuration of the VIVU mapping parameters. Smaller values ensure higher simulation speed capabilities because shorter call strings, and consequently simpler context, are managed during the simulation.

The results for the simulation speed achieved by the interpretation-based simulation methodology are shown in Figure 6.4. The chart in the figure reports the evaluation results of simulations executed considering TDBs generated with the $VIVU(\infty,\infty)$ and $VIVU(20,20)$ configurations. It has been shown that the first configuration ensures the achievement of more accurate results compared to any other configuration. The second one can achieve a good level of accuracy instead. The evaluation results show that the performance estimations produced considering the $VIVU(\infty,\infty)$ configuration achieve an average simulation speed of 1.59 MIPS with a minimal standard deviation value of 0.15. The minimal deviation in the simulation speed between the benchmarks suggests that the time spent by the simulator in interpreting the programs is dominated by the time spent in dynamically managing the call string and the corresponding contexts. The trade-off between simulation accuracy and speed due to the adoption of the $VIVU(20,20)$ configuration ensures a sensible simulation speedup. In fact, the average simulation speed increases up to 2.84 MIPS and the maximum observed value is 4.57 MIPS by simulating the `fac` benchmark.

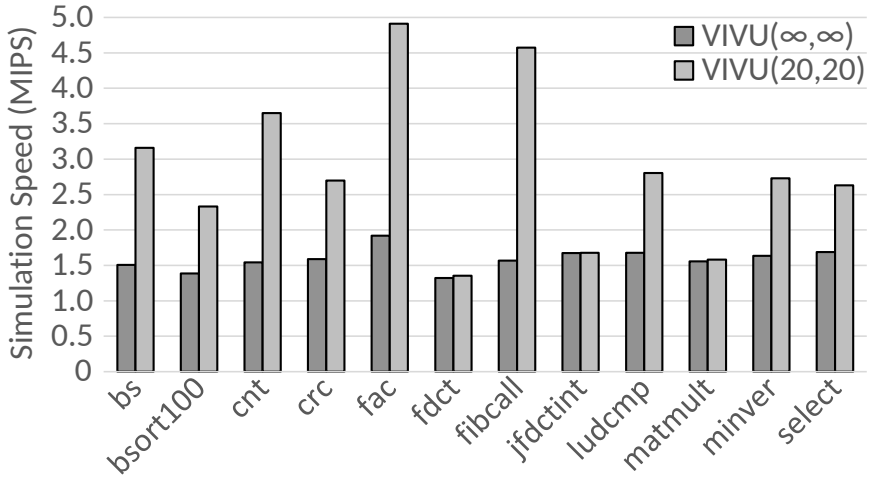


Figure 6.4 – *Interpretation-based context-sensitive methodology simulation speed: The chart shows the simulation speed, expressed in MIPS, observed when simulating different Mälardalen benchmarks and utilizing the interpretation-based methodology. Two different VIVU mapping configurations are evaluated. The simulations performed considering TDBs generated according to the $VIVU(\infty,\infty)$ achieve a maximum simulation speed close to 2 MIPS. Faster capabilities are shown by the $VIVU(20,20)$ configuration. The simulation of the* `fac` *benchmark achieves up to 5 MIPS simulation speed while the average value is 2.8 MIPS.*

The chart represented in Figure 6.5 shows the slowdown caused by simulating the benchmarks and considering both the VIVU configurations compared with the native interpretation of `lli`. The difference in simulation speed when considering the two different VIVU configurations is considerable only for some of the benchmarks. These benchmarks contain nested loop structures. Therefore, for these benchmarks, the number and the complexity of the contexts to consider during the simulation is higher.

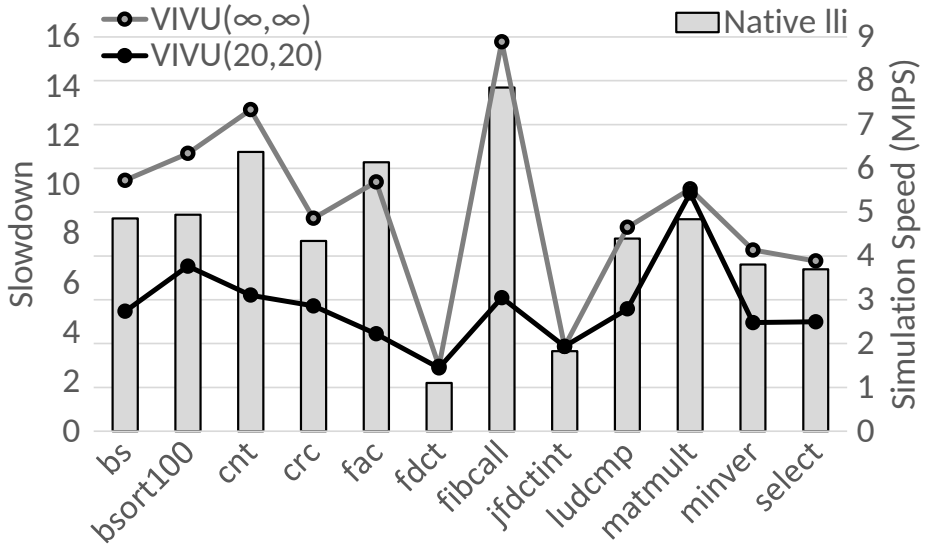


Figure 6.5 – *Interpretation slowdown due to the consideration of timing information: The native performance of the* `lli` *simulation has been compared with the enriched version implementing the interpretation-based timing simulation methodology. In general, the $VIVU(20,20)$ configuration ensures lower undesired slowdown than the one observed for the $VIVU(\infty,\infty)$ configuration while keeping an adequate level of accuracy in the resulting performance estimations.*

The observed simulation speed values for both the configuration are higher than the simulation speed capabilities achievable by standard cycle accurate simulators. However, they are substantially slower than other performance estimation techniques. For example, the context-sensitive simulation technique presented in [117] achieves approximately a simulation speed of 50 MIPS on a standard host machine. Despite the interpretation benefits, it is essential to rely on the JIT speedup for achieving better and more attractive simulation speed capabilities.

6.3.2 Just-In-Time Speedup

The same VIVU mapping configurations have been tested for evaluating the simulation speed achievable by executing JIT-based context sensitive simulations. The results of the evaluation are reported in Table 6.5. In this case, the values ensured by the JIT-compilation of the simulation code are significantly higher than the ones previously presented. The table allows directly comparing the JIT-based simulation performance capabilities with the ones observed for the interpretation-based solution.

Table 6.5 – *Comparison between the observed simulation speed capabilities of the two proposed simulation methodologies considering two different VIVU mapping configurations.*

	MIPS Interpretation		**MIPS JIT-Based**	
Benchmark	**VIVU(∞,∞)**	**VIVU(20,20)**	**VIVU(∞,∞)**	**VIVU(20,20)**
adpcm	1.48	2.4	9.05	41.82
bs	1.52	3.16	25.31	58.55
bsort100	1.36	2.35	70.92	121.56
cnt	1.55	3.58	10.63	27.74
compress	1.9	2.9	191.71	198.55
crc	1.62	2.62	18.33	54.84
duff	1.62	3.58	127.22	131.37
expint	1.74	3.65	132.04	143.18
fac	1.93	4.9	15.72	41.86
fdct	1.32	1.38	170.43	201.33
fft1	1.62	3.46	15.19	36.16
fibcall	1.6	4.62	22.71	65.55
fir	1.74	2.42	12.13	41.7
insertsort	1.46	2.36	18.64	55.24
janne_complex	1.56	3.59	9.58	28.52
jfdctint	1.64	2.03	52.57	151.43
lms	1.7	1.98	19.23	46.31
ludcmp	1.65	2.71	17.84	42.16
matmult	1.55	1.56	8.37	8.78
minver	1.62	2.77	15.46	37.55
ns	1.36	3.1	11.21	18
prime	1.28	3.32	19.06	67.84
qsort_exam	1.49	3.08	134	157
qurt	1.5	3.09	14.73	36.03
select	1.68	2.64	27.9	62
ud	1.63	3.47	16.79	37.64
Average	**1.58**	**2.95**	**45.65**	**73.57**
Std. Deviation	**0.16**	**0.82**	**55.32**	**55.92**

The average simulation speed observed for the slower but more accurate $VIVU(\infty,\infty)$ configuration is 45.65 MIPS. The average value for the faster but less accurate $VIVU(20,20)$ configuration is 73.57 MIPS. Both the configurations show the faster simulation speed values for the `compress` and `fdct` benchmarks. Differently from the interpretation case, the repeatedly executed nested loops are efficiently compiled only once by the JIT-compiler and this optimization ensures a substantial simulation speedup. In general, it has been observed that the simulation speed is directly related to the number of simulated ARM instructions. The experiment results show that the simulation of a substantial number of instructions reduces the minimal overhead due to the instrumentation in the bitcode and the time required by its JIT compilation.

Considering the simulation speed values reported in Table 6.5, in Figure 6.6 it is shown a chart representing the speedup due to the execution of JIT-based simulations compared with the interpretation option. Except for the `matmult` benchmark, the simulation speedup ensured by the JIT-based simulation methodology is substantial. In average, the simulation speedup for simulation based on the $VIVU(\infty,\infty)$ configuration is 45. The average speedup increases up to 73 for the $VIVU(20,20)$ configuration. The JIT-based simulation methodology executes the `fdct` benchmark at least 200 times faster than the simulation methodology based on the interpretation of the bitcode. The minimum speedup has been observed for the `matmult` benchmark. If highly compiled, this benchmark requires the simulation of only a limited number of instructions. Therefore, in this case, the benefit of the JIT-compilation cannot be recognized. However, the ensured substantial speedup confirms that it is possible to obtain accurate timing estimations by executing fast LLVM IR context-sensitive simulations.

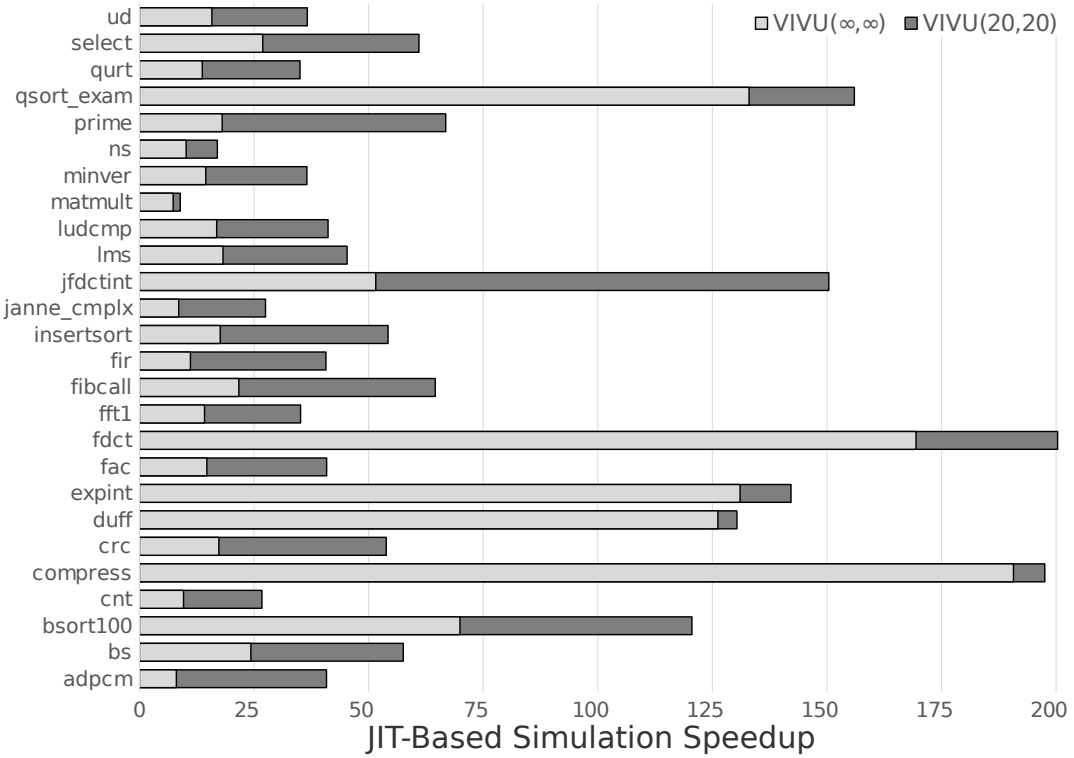


Figure 6.6 – *JIT-based simulation speedup: This chart shows the observed simulation speedup improvement ensured by the proposed JIT-based simulation technique compared with interpretation-based simulation's performance. The results show a substantial beneficial speedup for both the evaluated VIVU configurations. The JIT-based simulation technique ensures higher simulation speed capabilities that make it more attractive for producing early timing estimations during the early activities of the design space exploration.*

6.3.3 Simulation Speed Comparison

An additional evaluation of the simulation speed capabilities has been conducted. The simulation speed achievable by the JIT-based simulation methodology has been compared with the well-known and public available gem5 simulator [14, 96]. The gem5 simulator is a modular platform for performing research about hardware target architectures. The simulator is, at the time of writing, widely utilized in computer system design by academia and industry. Therefore, the gem5 simulator is a valid candidate for comparing its simulation speed capabilities with the ones achievable by the proposed JIT-based context-sensitive simulation methodology.

For the evaluation, the gem5 simulator has been compiled and configured for executing according to the fast implementation of the provided full system simulation mode [96]. Thereafter, the amount of time required for simulating the benchmarks with the gem5 simulator has been compared against the measured time required by the proposed JIT-based simulation methodology. Both the simulators have considered the processor ARM Cortex-R5. This choice is due to the availability of such a model for the gem5 simulator. The results of the conducted evaluation are shown in Figure 6.7.

The presented histogram in Figure 6.7 is intended to show the speedup resulting from simulating the benchmarks with the proposed JIT-based simulation methodology compared with the gem5 simulator. The different simulation speeds achieved by the JIT-based simulations are plotted with dark dots referring to the secondary axis. For a more fair comparison, the TDBs considered during the different simulations have been generated according to the $VIVU(\infty,\infty)$ configuration, the one that showed the slowest speed capabilities. Nevertheless, the plotting of the speedup in the chart requires a logarithmic scale. Two different kinds of gem5 simulations are considered in the chart. The light gray columns represent the comparison results when gem5 only simulates ARM instructions without performing any consideration about the timing behavior of the program that can be influenced by the hardware resources included in the target. Differently, the darker columns show the comparison

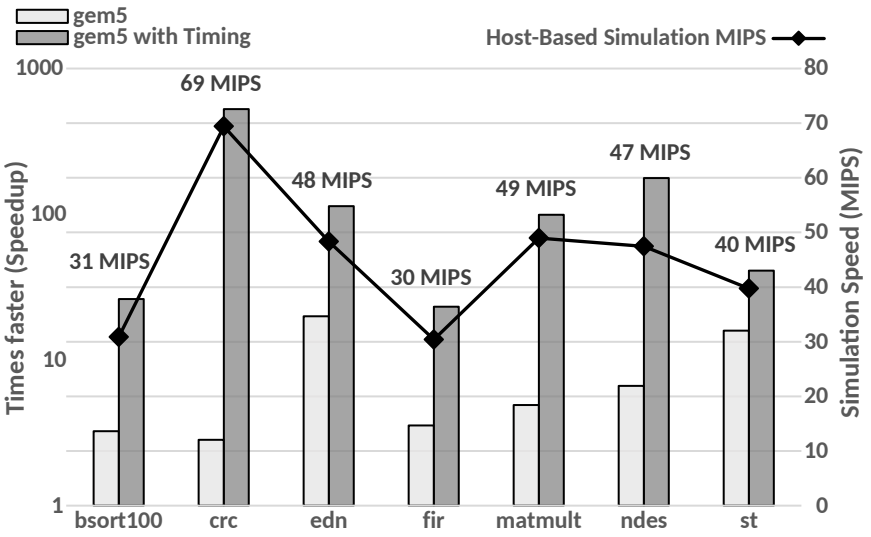


Figure 6.7 – *Simulation speed comparison against the gem5 simulator: The simulation speed capabilities of the proposed JIT-based context-sensitive timing simulation methodology have been compared against the gem5 simulator. The JIT-based simulations have been executed considering TDBs generated according to the $VIVU(\infty,\infty)$ configuration. The achieved simulation speed has been compared against the gem5 simulator with and without timing considerations. In both the cases, the simulation speedup ensured by the proposed simulation methodology is substantial.*

with a full timing gem5 simulation. In the first case, the maximum observed speedup ensured by the JIT-based simulation methodology compared with the gem5 simulator has been 20 for the `edn benchmark`. The observed average value is slightly above 8 instead. The speedup substantially increases when considering the time requested for executing full gem5 simulations. In fact, the maximum observed speedup reached a value up to 527 when simulating the `crc` benchmark and enabling gem5 to consider the timing behavior of caches, pipeline and branch predictor. For this second case, the average observed speedup is 144. These results show that the proposed JIT-based context-sensitive timing simulation approach allows executing accurate simulations requiring an amount of simulation time that is orders of magnitude shorter than simulating with the gem5 simulator.

6.3.4 Parallel Evaluation Speedup

A further experiment has been conducted in assessing the simulation speed capabilities of the proposed simulation methodology. In this final assessment, it has been evaluated and quantified the beneficial impact on the simulation speedup ensured by simulating in parallel multiple hardware/software configurations. This assessment has been conducted for both the interpretation-based and the JIT-based proposed techniques. In both the cases, it is expected that the simulation speed can increase by incrementing the number of configurations to be analyzed in parallel. In fact, it is requested by the simulator to compile and run the bitcode only once per execution. During the execution, the simulator can visit different TDBs in parallel, one for every configuration that has to be simulated. Thus, the overhead for the simulation setup, due to the interpretation or the JIT-compilation and necessary annotation code, can be significantly reduced.

The same TDBs are utilized for evaluating both the proposed simulation techniques. For every benchmark, four different TDBs have been produced, one for every processor listed in Table 6.1. All the TDBs have been generated according to the $VIVU(20,20)$ configuration. Consequently, the simulation speed of analyzing in parallel multiple configurations has been assessed. The evaluation results obtained for the interpretation-based and JIT-based methodologies are shown respectively in Figure 6.8 and in Figure 6.9.

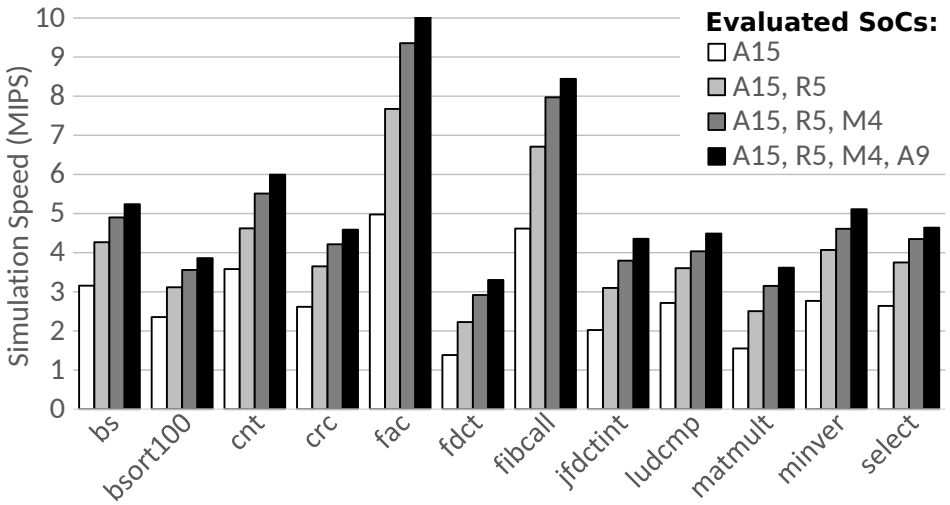


Figure 6.8 – *Resulting parallel evaluation speedup of interpretation-based simulations: The simulation speed increases by simulating in parallel multiple configurations, for every single increase. The average simulation speedup observed by analyzing four configurations in parallel compared to the analysis of a single configuration is up to 0.9.*

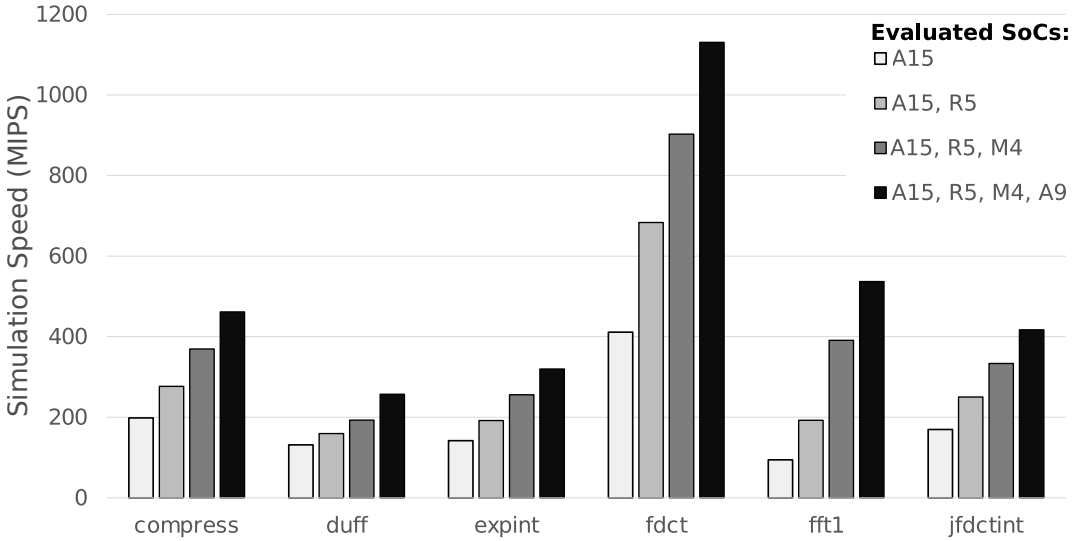


Figure 6.9 – *Resulting parallel evaluation speedup of JIT-based simulations: Even for this proposed simulation methodology, simulating in parallel multiple configurations of a system ensures a beneficial speedup compared with running a simulation that considers only one configuration. The maximum simulation speed observed when simulating up to four configurations in parallel exceeded the value of 1,000 MIPS. In general, the average simulation speedup is at least 1.5.*

The evaluation results of the interpretation-based simulation methodology show that increasing the number of configurations to simulate in parallel implies an improvement in the simulation speed capabilities, for every increment. When analyzing four different configurations in parallel, the maximum speedup value observed has been 1.4 for the `fdct` benchmark. The average observed value during the same value has been 0.9. Relying on the growth shown by the results in Figure 6.8, it is expected that even higher speedup values can be achieved by simulating in parallel more configurations.

A similar behavior has been observed when simulating in parallel multiple configurations via the JIT-based simulation methodology. In this case, the simulation speedup is considerably more substantial than the one observed for the interpretation-based measurements. The measurements show that the maximum observed speedup has been achieved by simulating the `fdct` benchmark and when evaluating four configurations in parallel. For this benchmark, the simulation speed exceeded the 1,000 MIPS showing a speedup value that is slightly below 3. In the average, the simulation speedup achieved by simulating four configurations in parallel is 1.5 and never below 1.0.

The showed beneficial speedup due to the simulation of multiple configurations in parallel is an important property of the proposed simulation methodology. In fact, during the early stages of the development of a system, the designers are interested in evaluating multiple configurations for identifying the most suitable one. Especially the JIT-based context-sensitive simulation based on the LLVM IR code representation can be a precious support in producing rapid performance estimations for driving the design space exploration activities in the development of a complex system.

6.4 Timing-Aware Simulink Simulation Effectiveness

The content of this section presents the last activity conducted during the experimental evaluation of the proposed simulation methodology. In particular, this section shows the effectiveness of the co-simulation methodology described in Section 5.4. This methodology enables the possibility of considering the architecture-dependent timing effects due to the execution of the software on a target platform by simulating the model directly on Simulink. Therefore, given a Simulink model, the evaluation is focused on showing the effects in the model's behavior functionally due to the execution of a timing-aware Simulink simulation based on the proposed co-simulation methodology.

6.4.1 Simulation Specification

The evaluation of the co-simulation methodology is conducted by analyzing a complex Simulink model from the ones provided by the MATLAB environment. The chosen model is called `sldemo_fuelsys` and its high-level design structure is shown in Figure 6.10. This model implements a closed-loop system for the automotive domain by modeling a hypothetical fault-tolerant fuel control system [165] that manages the air-fuel rate (AFR) of an engine. The AFR is the ratio between the mass of air and the mass of fuel that are internally present at a specific moment in a combustion engine. In this case, the performance of the system considering its execution on a target platform can strongly influence the performance of the vehicle's engine (such as emissions, fuel economy, and others). Therefore, the model is composed of a controller, a plant and some input sensors (such as throttle sensor, speed sensor, and others). The controller subsystem, colored in gray, is responsible for dynamically adjusting the AFR depending on the given input values. These values are determined by the sampling of the connected sensors and the plant feedback signals. The controller subsystem in the model shows an appropriate level of complexity for evaluating the co-simulation methodology. In fact, its implementation consists of more than one hundred components and Simulink blocks. Its implementation also includes complex features provided by the Stateflow toolbox [167].

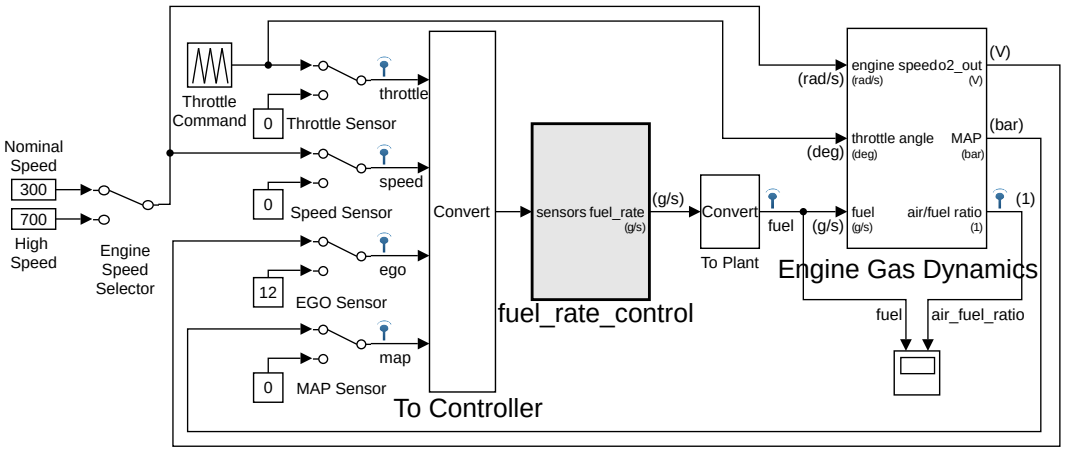


Figure 6.10 – *Simulated Simulink model for timing-aware co-simulation validation: The validation of the proposed co-simulation methodology has been conducted by analyzing the automotive* `sldemo_fuelsys` *model provided by Simulink. The model shows an appropriate level of complexity for managing the AFR of a vehicle's engine. The performance of the controller's subsystems strongly influence the performance of the engine. Therefore, a timing-aware Simulink simulation is useful for evaluating the system considering the effective execution of the controller's software on a real target platform.*

Table 6.6 – *Configurations considered during the co-simulation evaluation.*

	Target Processor	Hardware Configuration		Optimization Level
		Caches	Branch Predictor	
$Config_A$	Cortex-A15	Disabled	Disabled	-O2
$Config_B$		Disabled	Enabled	
$Config_C$		Enabled	Disabled	
$Config_D$		Enabled	Enabled	
$Config_E$	Cortex-R5	Disabled	Disabled	-O2
$Config_F$		Disabled	Enabled	
$Config_G$		Enabled	Disabled	
$Config_H$		Enabled	Enabled	

The conducted evaluation considers the simulation of eight different configurations based on two of the available ARM processors. The two considered processors are the Cortex-A15 and the Cortex-R5. For both of them are evaluated four different hardware configurations that consist in all the possible combinations resulting from enabling and disabling the all the cache memories and the branch prediction mechanisms. The source code generated for the controller component has been always compiled applying the `-O2` optimization level. The eight configurations are summarized and listed in Table 6.6.

6.4.2 Timing-Aware Simulation Effects

The performance behavior of all the eight configurations have been simulated and evaluated via the co-simulation methodology described in Section 5.4. The automatic procedure for annotating the Simulink model and executing the timing co-simulation have been repeated for all the configurations. The simulation configuration requires the controller subsystem to be periodically executed every ten milliseconds. The original system's behavior of the Simulink simulation without neglecting the possible effects due to the execution of the controller subsystem on a real target platform is shown in Figure 6.11. The AFR value managed by the system is represented by the darker solid curve.

As expected, the simulation results show different system behaviors. The discrepancies between the behaviors are due to the consideration of different configuration-dependent timing estimations. In fact, it has been observed that the dynamic behavior of the selected target platform configuration influences the resulting AFR values. The timing effects considered in the proposed timing-aware Simulink simulations have a direct impact in the ratio managed by the system. For example, in the native Simulink simulation, at every controller activation, the fuel output value is immediately available for the plant due to the SRP paradigm. This behavior is unrealistic. In a timing-aware Simulink simulation instead, the delay library components inserted in the controller subsystem introduce some jitter in the output's computation. These delays influence the complete system behavior. All these behaviors differ from the one observed by executing a native Simulink simulation.

For readability reasons, there is not a trivial way of showing the different system behaviors observed via simulation for all the considered configurations. Therefore, instead of showing all the different behaviors, in Figure 6.11 it is shown the deviation observed comparing the results produced via the native Simulink simulation with the ones produced via timing-aware simulations. In particular, the chart shows for both the two simulated processors the maximum deviation percentage observed for the respective configurations. The system behavior deviation in the initial phase of the simulation is substantial. The deviation decreases once

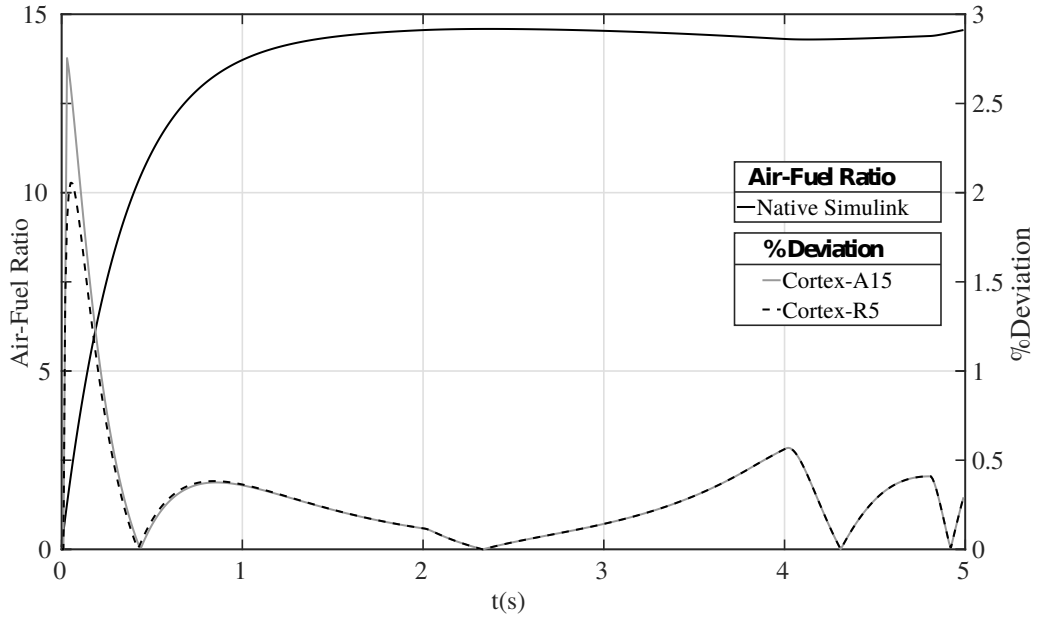


Figure 6.11 – *Deviation between native and timing-aware Simulink simulation behaviors: The chart shows the maximum percentage of deviation observed comparing the system performance behavior resulting from a native Simulink simulation with the ones resulting from timing-aware simulations of the functionally equivalent model. The resulting native Simulink simulation behavior is plotted with the darker solid curve and it represents the computed AFR value during the simulation. Instead, the other curves represent the maximum percentage of deviation observed executing timing-aware cos-simulations and considering multiple configurations of two ARM processors.*

that the AFR value is stable. However, in the simulations, it starts increasing again every time that the AFR value shows a minimal change.

Considering only four of the configurations and a limited amount of simulation time, it is possible to show four different behaviors for the same Simulink model. The different behaviors of the four configurations proposed for the processor ARM Cortex-A15 are shown in Figure 6.12. The chart shows an arbitrary simulation interval extracted from the scope component connected to the plant subsystem's output. This interval includes two controller execution cycles. In this convenient interval, it is possible to plot together the curve resulting from the timing-aware simulations and the native Simulink simulation. The native Simulink simulation behavior, still plotted with the darker solid curve, differs from the timing-aware simulation results. According to the simulation results of the original model, its behavior appears to be more optimistic than the ones of the timing-aware simulations. In fact, the lack of the delay blocks allows the controller to be more reactive. Differently, considering the appropriate timing effects in a timing-aware co-simulation, the system shows a less reactive behavior. Between them, the configuration considering the enabling of both the caches and the branch predictor mechanism results to be the most reactive one. This behavior is the consequence of better performance estimations for the simulated system. On the opposite, simulating the system for a configuration that considers the disabling of both the hardware resources leads to observing the less reactive behavior. Therefore, different configurations can be explored via the proposed timing-aware Simulink simulation methodology.

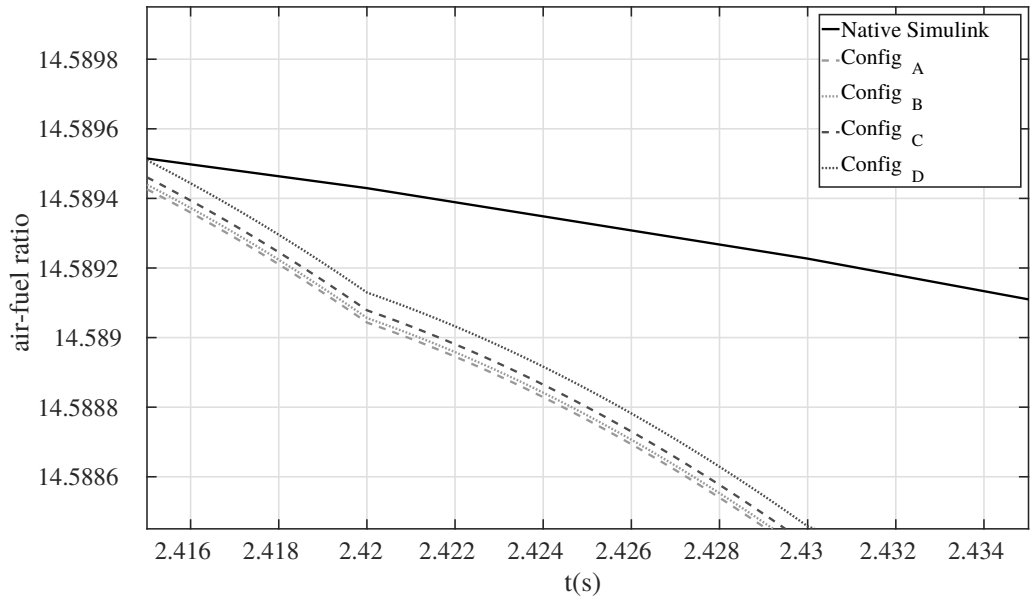


Figure 6.12 – *Different system behaviors observed via timing-aware Simulink simulations: The chart shows the distinct behaviors of the system observed by executing different Simulink simulations. Considering the arbitrary simulation time slot, it is possible to plot together the system behavior produced by simulating the original model with the ones generated via timing-aware simulations of the configurations proposed for the Cortex-A15 processor. The proposed annotation mechanism, that inserts delay blocks in the controller subsystem, allows the Simulink simulations to be more realistic by considering the performance estimations produced during the co-simulation.*

6.5 Summary

This chapter presented the overall experimental results collected during the conducted evaluation activities. Initially, the evaluation focused on assessing the level of accuracy for the CFG mappings produced via the proposed two-phases algorithm. In this case, the measured accuracy is always close to 100% even in case the analyzed programs are compiled with the highest optimization level. Consequently, the evaluation focused on assessing the accuracy and the simulation speed capabilities of the proposed simulation methodology. The JIT-based simulations showed exactly the same level of accuracy of the interpretation-based simulations. For the slowest but potentially more accurate simulation's configuration, the performance estimations showed a prediction error below 1%. The average error slightly increased up to 2% when considering a faster configuration. An acceptable level of accuracy (prediction error below 9%) has been observed also in producing a rough estimation for the execution time of a synthetic synchronous heterogeneous system. However, the JIT-based simulations showed a substantial and beneficial simulation speedup compared to the interpretation-based simulations. In fact, in the average case, a JIT-based simulation can be executed 73 times faster reaching a simulation speed up to 74 MIPS. The simulation performance has been compared with gem5, a well-known simulator in the state of the art. For the analyzed programs, the proposed simulation approach produced accurate results, in average, 144 times faster (the maximum observed speedup value has been 527). Furthermore, the evaluation quantified the benefit of evaluating multiple configurations in parallel. In fact, the simulation exceeded the speed of 1.000 MIPS by evaluating in parallel four different configurations. Finally, an exemplary Simulink model has been analyzed for evaluating the previously presented co-simulation methodology. The evaluation showed that the system's behavior changes when the model considers the different timing estimations predicted for the execution of the software components.

CHAPTER 7

Conclusions and Future Research

The first scope of this final chapter is to summarize the main contributions and the key results presented in this thesis. Based on this initial description, the second scope of the chapter is to propose and discuss possible future related research directions considering eventual current limitations and open research opportunities.

7.1 Thesis Summary and Conclusions

This thesis presented a novel context-sensitive timing simulation methodology in support of the early design space exploration of embedded systems. Fast and accurate performance estimations of a system can be assessed via host-based simulation. Multiple system's configurations can be evaluated in parallel. Every configuration can vary in software or hardware related aspects. The evaluation of multiple configurations in parallel in only one simulation, desirable possibility for the design space exploration, ensures higher simulation speed capabilities. Furthermore, it is shown that the same simulation methodology can be used for early evaluating also the performance of heterogeneous embedded systems that are designed to be executed on MPSoC platforms. In particular, different system's partitions can be rapidly evaluated for investigating and determining the most suitable one. This can be enabled by following one of the tracing techniques proposed for generating the timing models considered during the simulation. The simulation methodology supports the definition of the system configurations via Simulink. In this case, when the system is described via a Simulink model, the thesis proposes an additional new co-simulation methodology for evaluating the performance of a system directly in a Simulink simulation. This is possible by enabling the simulation of an enriched, but functionally equivalent, version of the given model. During the native Simulink simulation, these kind of models are able to consider the performance estimations produced via timing simulation.

The context-sensitive timing simulation methodology is based on the IR level of the software programs. This program representation is internal to the compiler and, differently from the source code, it already includes part of the effects of the compiler optimizations. On the one hand, this choice ensures a beneficial level of abstraction to both the analysis and simulation stages, but on the other hand, it implies the need of an appropriate mapping that is hard to

define. The mapping matches the structure of a program at the IR level to the corresponding structure at the binary level. Aggressive compiler optimizations can substantially change the structure of a program making a direct matching impossible. Considering the problem complexity, it is desirable for the matching algorithm to be fully automatic without requiring any modification of the compiler or the supervision of an expert. Therefore, this thesis presented two contributions to the state of the art. Initially, the thesis described a tracing-based extension for an existing approach. Consequently, a second different methodology has been presented that overcomes the drawbacks of the first solution. This second mapping approach is based on an innovative and fully automatic two-phases algorithm. The conducted experimental evaluation shows that the mapping accuracy is for most of the analyzed programs close to 100%. The accuracy of the mapping algorithm is essential for producing precise performance estimations via the LLVM IR context-sensitive timing simulation methodology described in this thesis.

An ideal timing simulator is expected to produce accurate performance estimations in a very fast way. Unfortunately, accuracy and simulation speed are two contrasting metrics that require the definition of a suitable trade-off. For this reason, the conducted experimental evaluation focused on assessing both the level of accuracy in the simulation results and the simulation speed capabilities achievable by the proposed simulation methodology. The JIT-based simulation approach resulted substantially faster than the interpretation one but, as expected, both showed exactly the same level of accuracy. The performance estimations produced by considering the most accurate configuration of the timing model showed a prediction error below 1%. The prediction error slightly increased to 2% simulating the programs with a less accurate but faster configuration. Approximate performance estimations, observed error percentage below 9%, can be produced for early evaluating different configurations of a heterogeneous system. The JIT-based simulation ensures a simulation speed that is on the average 73 times faster than the interpretation approach. The fastest observed simulation speed for a JIT-based simulation of only one configuration has been 74 MIPS. Compared with gem5,

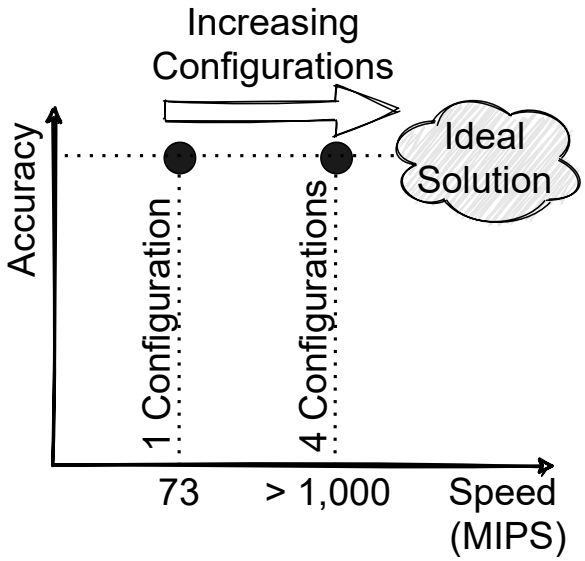


Figure 7.1 – *Benefits of simulating in parallel multiple configurations: When evaluating only one system's configuration, the average observed simulation speed for JIT-based simulations is 73 MIPS. Increasing the number of configurations simulate in parallel ensures a substantial simulation speedup while keeping the same elevated level of accuracy. In fact, evaluating in parallel four different configurations, the simulation speed exceeded the value of 1,000 MIPS. It is expected the simulation speed to further increase in case more configurations are evaluated in parallel.*

a well-known timing simulator, the proposed simulation methodology can produce accurate results in the average 144 times faster. Higher simulation speed can be achieved by simulating in parallel multiple system's configurations. In one specific case, during the evaluation, it has been observed a simulation speed that exceeded the value of 1,000 MIPS when four different configurations were simulated in parallel. As shown in Figure 7.1, it is expected the simulation to achieve faster speed capabilities by evaluating in parallel more configurations and keeping the elevated level of accuracy. The elevated speed capabilities of the JIT-based simulation enables the possibility of evaluating the design of an embedded system directly in Simulink. This is possible by simulating an enriched version of the model that enables a fast co-simulation technique between Simulink and the context-sensitive timing simulation methodology described in this thesis.

7.2 Future Work

In this section are outlined possible eventual future research directions. In particular, some considerations can be made for possible optimization opportunities. Improvements can be applied at multiple levels for covering different simulation objectives.

7.2.1 Improving Simulation Speed

The simulation methodology proposed in this thesis shows elevated simulation speed capabilities. However, a first possible optimization's goal consists in further improving the simulation speed capabilities of the proposed simulation methodology. Some research can be conducted in this direction. For instance, some considerations can be made on the annotation mechanism proposed for the JIT-based simulation technique. The annotation determines an unavoidable slowdown to the native `lli` performance. In fact, in a simulation, some external code has to be executed every time that the annotation code contained in the different IR basic blocks is encountered. The external code initially checks if an LLVM IR path can be translated into a binary one. If a translation is given, the binary path is consequently utilized for querying the necessary timing databases and the requested performance estimations are finally updated. This operation mode shows two main sources of slowdown:

1. *Mapping translation* - The simulation spends time in checking if, according to the previously defined mapping, a binary path translation is available every time that the annotation is encountered without knowing if a translation will be effectively provided.
2. *Repeated translations* - The simulator does not store any translation that is encountered during the simulation. In case of loops, multiple identical IR paths are visited. Therefore, the same translations are repeatedly requested and consequently wasting precious time that slowdowns the simulation.

Considering these two sources of undesired slowdown, some expedients can be investigated. For instance, the mapping translation slowdown can be contained by trying to reduce the amount of annotation in the simulation code. The annotation code should appear only in convenient parts of the code (e.g. only on conditional basic blocks, only on basic blocks that represent the last block in a translatable path, etc.). In a different way, the repeated translation slowdown can be mitigated by enriching the simulator with a sort of a smart mechanism (similar to a cache memory) that recalls the translations that have been already performed during the simulation.

7.2.2 Adaptive Timing Model

The timing model chosen for supporting the definition of the simulation methodology proposed in this thesis ensures the possibility of producing highly accurate performance estimations. However, its actual implementation presents some limitations. The limitations are mainly due to the scarce flexibility of a timing database. Furthermore, other issues arise at the time of generating such timing models. These issues are:

1. *Fixed configuration* - A timing database is fixed for a single system's configuration and it allows simulating only the performance of that specific configuration. In fact, it describes the timing behavior of a specific version of a software that is compiled applying a certain level of optimization and that is executed on a well-defined configuration of a target platform. Any minimal change in this setup invalidates the content of a timing database.

2. *Tracing methodology* - At the time of writing, there does not exist a metric that allows evaluating the effectiveness or quality of a timing database.Measuring the program's timing behavior for at least 100% of the binary coverage only ensures having a relative execution time for the first iteration of every context in the program. However, the simulation accuracy strongly depends on the coverage of all the contexts that are considered in a timing database. In addition to a beneficial metric, it is hard to define the input data set for appropriately covering all the necessary contexts via measurements.

3. *VIVU mapping configuration* - When generating a timing database, a VIVU mapping configuration has to be specified. Unfortunately, it is not trivial to determine the most appropriate configuration for a given program. This configuration may different according to the program structures. In fact, it affects both the simulation speed and accuracy. It would be beneficial to have some kind of support for determining a suitable configuration.

Some research activities can be planned for limiting these limitations and making a timing database more flexible and scalable. Considering that the proposed simulation methodology is designed for the early development stages of a system, it would be beneficial to support the possibility of evaluating small changes in the system. The simulation methodology proposed in this thesis overcomes some of these problems by allowing the evaluation of multiple configurations in parallel. The integration of additional models (such as analytical models or others) or simulators can enable the possibility of further polishing this limitation. However, changes in the software program are not considered. Furthermore, the issues related to the tracing activities and the identification of the ideal VIVU mapping configuration could be addressed by relying on appropriate machine learning algorithms. These algorithms should achieve two specific goals. The first one is the generation of a valid input data set that allows producing a sufficiently accurate timing database. The second one is the generation of a timing database that applies the VIVU mapping according to the program's structure.

7.2.3 Multi-Core Support

The proposed simulation methodology offers the possibility of early evaluating the performance of synchronous embedded systems. However, due to the interference effects caused by the shared resources included in multiprocessor platforms, this simulation methodology is not directly applicable for simulating and evaluating multi-core systems. In this regard, the main issue is represented by the difficulty of modeling the interference effects via the timing database approach. At the same time, it is hard to accurately reproduce these effects during

the simulation. However, some research has been conducted for ensuring a more deterministic behavior for programs that are executed concurrently on a multi-core platform [67, 38, 47]. These approaches propose to modify the code at compile time for improving their timing analyzability in a multi-core scenario. The modifications make the programs less susceptible to the timing effects caused by the unavoidable multi-core interference. In particular, the simple approach presented in [38] achieves promising results by adding *nop* instructions in convenient parts of a program. A program whose timing behavior is slightly influenced by the parallel execution of other programs can be eventually timing analyzed in isolation. The resulting timing model can be consequently considered in a multi-core simulation scenario. Prospectively, further research in this direction can open the possibility of adopting different context-sensitive timing simulation techniques for analyzing multi-core systems.

List of Abbreviations

AMP	**A**symmetric **M**ulti-**P**rocessing
BCET	**B**est-**C**ase **E**xecution **T**ime
CCN	**C**yclomatic **C**omplexity **N**umber
CFG	**C**ontrol **F**low **G**raph
DSE	**D**esign **S**pace **E**xploration
ESL	**E**lectronic **S**ystem **L**evel
FU	**F**unctional **U**nit
ICFG	**I**nterprocedural **C**ontrol **F**low **G**raph
ILP	**I**nstruction **L**evel **P**arallelism
IoT	**I**nternet **o**f **T**hings
IPC	**I**nstructions **P**er **C**ycle
IR	**I**ntermediate **R**epresentation
ISA	**I**nstruction **S**et **A**rchitecture
ISS	**I**nstruction **S**et **S**imulator
LOC	**L**ines **O**f **C**ode
MC	**M**achine **C**ode
MIPS	**M**illion **I**nstructions Simulated **P**er **S**econd
MIR	**M**achine **I**ntermediate **R**epresentation
MMU	**M**emory **M**anagement **U**nit
MOET	**M**aximal **O**bserved **E**xecution **T**ime
MPSoC	**M**ulti-**P**rocessor **S**ystem **o**n **C**hip
RTL	**R**egister **T**ransfer **L**evel
SLDL	**S**ystem **L**evel **D**esign **L**anguage
SMP	**S**ymmetric **M**ulti-**P**rocessing
SSA	**S**ingle **S**tatic **A**ssignment
SRP	**S**ynchronous **R**eactive **P**aradigm
TDB	**T**iming **D**ata**B**ase
TLB	**T**ranslation **L**ookaside **B**uffer
TLC	**T**arget **L**anguage **C**ompiler
VIVU	**V**irtual **I**nlining and **V**irtual **U**nrolling
WCET	**W**orst-**C**ase **E**xecution **T**ime

List of Figures

List of Tables

Listings

References

[1] Lauterbach GmbH. *Lauterbach Trace32 Tracer*. URL: https://www.lauterbach.com/frames.html?home.html.

[2] Infineon Technologies AG. *TriCore AURIX*. URL: https://www.infineon.com/cms/de/product/microcontroller/32-bit-tricore-microcontroller/32-bit-tricore-aurix-tc3xx.

[3] A Aho, M Lam, R Sethi, J Ullman, Keith Cooper, Linda Torczon, et al. *Compilers: Principles, Techniques and Tools*. Addison-Wesley 2nd edition, 2007.

[4] Krste Asanovic and Andrew Waterman. "The RISC-V Instruction Set Manual". In: *Privileged Architecture, Document Version 20190608-Priv-MSU- Ratified*. Vol. 2. RISC-V Foundation, 2019.

[5] Colin Atkinson and Thomas Kuhne. "Model-driven development: a metamodeling foundation". In: *IEEE software* 20.5 (2003), pp. 36–41.

[6] Mossaad Ben Ayed, Faouzi Bouchhima, and Mohamed Abid. "CODIS+: Co-simulation environment for heterogeneous systems". In: *Journal of Control Engineering and Applied Informatics* 20.1 (2018), pp. 98–107.

[7] Felice Balarin, Paolo Giusto, Attila Jurecska, Michael Chiodo, Claudio Passerone, Ellen Sentovich, et al. *Hardware-software co-design of embedded systems: the POLIS approach*. Springer Science & Business Media, 1997.

[8] Prithviraj Banerjee, Nagraj Shenoy, Alok Choudhary, Scott Hauck, Chris Bachmann, Malay Haldar, et al. "A MATLAB compiler for distributed, heterogeneous, reconfigurable computing systems". In: *Proceedings 2000 IEEE Symposium on Field Programmable Custom Computing Machines (Cat. No. PR00871)*. IEEE. 2000, pp. 39–48.

[9] Markus Becker, Daniel Baldin, Christoph Kuznik, Mabel Mary Joy, Tao Xie, and Wolfgang Mueller. "XEMU: an efficient QEMU based binary mutation testing framework for embedded software". In: *Proceedings of the tenth ACM international conference on Embedded software*. 2012, pp. 33–42.

[10] Martin Becker, Marius Pazaj, and Samarjit Chakraborty. "WCET Analysis meets Virtual Prototyping: Improving Source-Level Timing Annotations". In: *Proceedings of the 22nd International Workshop on Software and Compilers for Embedded Systems*. 2019, pp. 13–22.

[11] Fabrice Bellard. "QEMU, a fast and portable dynamic translator." In: *USENIX annual technical conference, FREENIX Track*. Vol. 41. Califor-nia, USA. 2005, p. 46.

[12] Guillem Bernat, Antoine Colin, and Stefan M Petters. “WCET analysis of probabilistic hard real-time systems”. In: *23rd IEEE Real-Time Systems Symposium, 2002. RTSS 2002.* IEEE. 2002, pp. 279–288.

[13] David Biancolin, Sagar Karandikar, Donggyu Kim, Jack Koenig, Andrew Waterman, Jonathan Bachrach, et al. “FASED: FPGA-accelerated simulation and evaluation of DRAM”. In: *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2019, pp. 330–339.

[14] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, et al. “The gem5 simulator”. In: *ACM SIGARCH computer architecture news* 39.2 (2011), pp. 1–7.

[15] Massimo Bombino and Patrizia Scandurra. “A model-driven co-simulation environment for heterogeneous systems”. In: *International Journal on Software Tools for Technology Transfer* 15.4 (2013), pp. 363–374.

[16] Aimen Bouchhima, Patrice Gerin, and Frédéric Pétrot. “Automatic instrumentation of embedded software for high level hardware/software co-simulation”. In: *2009 Asia and South Pacific Design Automation Conference*. IEEE. 2009, pp. 546–551.

[17] F Bouchhima, M Briere, G Nicolescu, M Abid, and EM Aboulhamid. “A SystemC/Simulink co-simulation framework for continuous/discrete- events simulation”. In: *2006 IEEE International Behavioral Modeling and Simulation Workshop*. IEEE. 2006, pp. 1–6.

[18] Florian Brandner, Andreas Fellnhofer, Andreas Krall, and David Riegler. “Fast and accurate simulation using the LLVM compiler framework”. In: *Proceedings of the 1st Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools, RAPIDO*. Vol. 9. 2009, pp. 1–6.

[19] Tomas Brezina, Zdenek Hadas, and Jan Vetiska. “Using of Co-simulation ADAMS-SIMULINK for development of mechatronic systems”. In: *14th International Conference Mechatronika*. IEEE. 2011, pp. 59–64.

[20] Oliver Bringmann, Wolfgang Ecker, Andreas Gerstlauer, Ajay Goyal, Daniel Mueller-Gritschneder, Prasanth Sasidharan, et al. “The next generation of virtual prototyping: Ultra-fast yet accurate simulation of HW/SW systems”. In: *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2015, pp. 1698–1707.

[21] Oliver Bringmann, Christoph Gerum, and Sebastian Ottlik. “Timing Models for Fast Embedded Software Performance Analysis”. In: *Handbook of Hardware/Software Codesign*. Ed. by Soonhoi Ha and Jürgen Teich. Dordrecht: Springer Netherlands, 2017, pp. 655–682.

[22] Juan Castillo, Hector Posadas, Eugenio Villar, and Marcos Martinez. “Fast instruction cache modeling for approximate timed HW/SW co- simulation”. In: *Proceedings of the 20th symposium on Great lakes symposium on VLSI*. 2010, pp. 191–196.

[23] Jeronimo Castrillon, Rainer Leupers, and Gerd Ascheid. “MAPS: Mapping concurrent dataflow applications to heterogeneous MPSoCs”. In: *IEEE Transactions on Industrial Informatics* 9.1 (2011), pp. 527–545.

[24] Francisco J Cazorla, Eduardo Quiñones, Tullio Vardanega, Liliana Cucu, Benoit Triquet, Guillem Bernat, et al. “Proartis: Probabilistically analyzable real-time systems”. In: *ACM Transactions on Embedded Computing Systems (TECS)* 12.2s (2013), pp. 1–26.

[25] Suhas Chakravarty, Zhuoran Zhao, and Andreas Gerstlauer. “Automated, retargetable back-annotation for host compiled performance and power modeling”. In: *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*. IEEE. 2013, pp. 1–10.

[26] Eric Cheung, Harry Hsieh, and Felice Balarin. “Fast and accurate performance simulation of embedded software for MPSoC”. In: *2009 Asia and South Pacific Design Automation Conference*. IEEE. 2009, pp. 552–557.

[27] Yuze Chi, Young-kyu Choi, Jason Cong, and Jie Wang. “Rapid cycle-accurate simulator for high-level synthesis”. In: *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2019, pp. 178–183.

[28] Bob Cmelik and David Keppel. “Shade: A fast instruction-set simulator for execution profiling”. In: *Fast simulation of computer architectures*. Springer, 1995, pp. 5–46.

[29] Antoine Colin and Stefan M Petters. “Experimental evaluation of code properties for wcet analysis”. In: *RTSS 2003. 24th IEEE Real-Time Systems Symposium, 2003*. IEEE. 2003, pp. 190–199.

[30] DWARF Standards Committee. *DWARF Debugging Standard*. URL: `http://dwarfstd.org`.

[31] Fabio Cremona, Matteo Morelli, and Marco Di Natale. “TRES: a modular representation of schedulers, tasks, and messages to control simulations in simulink”. In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. 2015, pp. 1940–1947.

[32] Christoph Cullmann, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza, Jan Reineke, et al. “Predictability considerations in the design of multi-core embedded systems”. In: *Proceedings of Embedded Real Time Software and Systems* 36 (2010), p. 42.

[33] Stephen Anthony Edwards. *The specification and execution of heterogeneous synchronous reactive systems*. Electronics Research Laboratory, College of Engineering, University of California, 1997.

[34] Andreas Fauth, Johan Van Praet, and Markus Freericks. “Describing instruction set processors using nML”. In: *Proceedings the European Design and Test Conference. ED&TC 1995*. IEEE. 1995, pp. 503–507.

[35] Christian Ferdinand and Reinhold Heckmann. “ait: Worst-case execution time prediction by static program analysis”. In: *Building the Information Society*. Springer, 2004, pp. 377–383.

[36] Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, et al. “Reliable and precise WCET determination for a real-life processor”. In: *International Workshop on Embedded Software*. Springer. 2001, pp. 469–485.

[37] Christian Ferdinand, Florian Martin, Reinhard Wilhelm, and Martin Alt. “Cache behavior prediction by abstract interpretation”. In: *Science of Computer Programming* 35.2-3 (1999), pp. 163–189.

[38] Gabriel Fernandez, Javier Jalle, Jaume Abella, Eduardo Quinones, Tullio Vardanega, and Francisco J Cazorla. “Computing safe contention bounds for multicore resources with round-robin and FIFO arbitration”. In: *IEEE Transactions on Computers* 66.4 (2016), pp. 586–600.

[39] Robert France and Bernhard Rumpe. “Model-driven development of complex software: A research roadmap”. In: *Future of Software Engineering (FOSE’07)*. IEEE. 2007, pp. 37–54.

[40] Nikolina Frid, Danko Ivošević, and Vlado Sruk. “Elementary operations: a novel concept for source-level timing estimation”. In: *Automatika: časopis za automatiku, mjerenje, elektroniku, računarstvo i komunikacije* 60.1 (2019), pp. 91–104.

[41] Nikolina Frid, Danko Ivošević, and Vlado Sruk. “Performance estimation in heterogeneous MPSoC based on elementary operation cost”. In: *2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE. 2016, pp. 1202–1205.

[42] Jon Friedman. “MATLAB/Simulink for automotive systems design”. In: *Proceedings of the Design Automation & Test in Europe Conference*. Vol. 1. IEEE. 2006, pp. 1–2.

[43] Andreas Gerstlauer, Suhas Chakravarty, Manan Kathuria, and Parisa Razaghi. “Abstract system-level models for early performance and power exploration”. In: *17th Asia and South Pacific Design Automation Conference*. IEEE. 2012, pp. 213–218.

[44] Christoph Gerum, Oliver Bringmann, and Wolfgang Rosenstiel. “Source level performance simulation of gpu cores”. In: *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2015, pp. 217–222.

[45] Hamid Reza Ghasemi and Zainalabedin Navabi. “An effective VHDL-AMS simulation algorithm with event”. In: *18th International Conference on VLSI Design held jointly with 4th International Conference on Embedded Systems Design*. IEEE. 2005, pp. 762–767.

[46] Jason Ghidella and Jon Friedman. “Model-based design streamlines development of body electronics systems”. In: *Automotive Electronics* 5.6 (2005).

[47] Jeremy Giesen, Pedro Benedicte, Enrico Mezzetti, Jaume Abella, and Francisco J Cazorla. “Modeling contention interference in crossbar-based systems via sequence-aware pairing (SeAP)”. In: *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2020, pp. 253–266.

[48] Marius Gligor, Nicolas Fournel, and Frédéric Pétrot. “Using binary translation in event driven simulation for fast and flexible MPSoC simulation”. In: *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*. 2009, pp. 71–80.

[49] Andrés Goens, Robert Khasanov, Jeronimo Castrillon, Marcus Hähnel, Till Smejkal, and Hermann Härtig. “Tetris: a multi-application run-time system for predictable execution of static mappings”. In: *Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems*. 2017, pp. 11–20.

[50] Cláudio Gomes, Casper Thule, David Broman, Peter Gorm Larsen, and Hans Vangheluwe. “Co-simulation: State of the art”. In: *arXiv preprint arXiv:1702.00686* (2017).

[51] Matthias Gries. “Methods for evaluating and covering the design space during early design development”. In: *Integration, the VLSI Journal* 38.2 (2004), pp. 131–183.

[52] Matthias Gries and Kurt Keutzer. *Building ASIPS: The mescal methodology*. Springer Science & Business Media, 2006.

[53] David Griffin, Benjamin Lesage, Iain Bate, Frank Soboczenski, and Robert I Davis. “Forecast-based interference: Modelling multicore interference from observable factors”. In: *Proceedings of the 25th International Conference on Real-Time Networks and Systems*. 2017, pp. 198–207.

[54] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. “The Mälardalen WCET benchmarks: Past, present and future”. In: *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2010.

[55] Claude Helmstetter, Vania Joloboff, Zhou Xinlei, and Gao Xiaopeng. “Fast instruction set simulation using LLVM-based dynamic translation”. In: *International MultiConference of Engineers and Computer Scientists 2011*. Vol. 2188. 2011, pp. 212–216.

[56] Jörg Henkel. “Closing the SoC design gap”. In: *Computer* 36.9 (2003), pp. 119–121.

[57] Dan Henriksson, Anton Cervin, and Karl-Erik Årzén. “TrueTime: Real-time control system simulation with MATLAB/Simulink”. In: *Proceedings of the Nordic MATLAB Conference*. Copenhagen, Denmark. 2003.

[58] G Glenn Henry and Terry Parks. *Microprocessor that fuses if-then instructions*. US Patent 10,394,562. 2019.

[59] Thomas A Henzinger and Joseph Sifakis. "The embedded systems design challenge". In: *International Symposium on Formal Methods*. Springer. 2006, pp. 1–15.

[60] Vladimir Herdt, Daniel Große, and Rolf Drechsler. "Verification of Embedded Software Binaries using Virtual Prototypes". In: *Enhanced Virtual Prototyping*. Springer, 2021, pp. 143–174.

[61] Vance Hilderman and Tony Baghi. *Avionics certification: a complete guide to DO-178 (software), DO-254 (hardware)*. Avionics Communications, 2007.

[62] Hitex. *Hitex LPC4350 Evaluation board*. URL: https://www.nxp.com/design/designs/hitex-lpc4350-evaluation-board:OM13031.

[63] Harold Hoehne and Robert Piloty. "Design verification at the register transfer language level". In: *IEEE Transactions on Computers* 100.9 (1975), pp. 861–867.

[64] Kenneth Hoste and Lieven Eeckhout. "Comparing benchmarks using key microarchitecture -independent characteristics". In: *2006 IEEE International Symposium on Workload Characterization*. IEEE. 2006, pp. 83–92.

[65] Texas Instruments Incorporated. *EVMK2EX - K2E Development Board*. URL: https://www.ti.com/tool/EVMK2EX.

[66] Texas Instruments Incorporated. *Hercules RM57Lx Development Kit*. URL: https://www.ti.com/tool/tmdxrm57lhdk.

[67] Dan Iorga, Tyler Sorensen, John Wickerson, and Alastair F Donaldson. "Slow and steady: Measuring and tuning multicore interference". In: *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2020, pp. 200–212.

[68] Canturk Isci and Margaret Martonosi. "Runtime power monitoring in high-end processors: Methodology and empirical data". In: *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* IEEE. 2003, pp. 93–104.

[69] Tsuyoshi Isshiki, Dongju Li, Hiroaki Kunieda, Toshio Isomura, and Kazuo Satou. "Trace-driven workload simulation method for Multiprocessor System-On-Chips". In: *Proceedings of the 46th Annual Design Automation Conference*. 2009, pp. 232–237.

[70] Zai Jian Jia, Antonio Núñez, Tomás Bautista, and Andy D Pimentel. "A two-phase design space exploration strategy for system-level real-time application mapping onto MPSoC". In: *Microprocessors and Microsystems* 38.1 (2014), pp. 9–21.

[71] Rik Jongerius, Andreea Anghel, Gero Dittmann, Giovanni Mariani, Erik Vermij, and Henk Corporaal. "Analytic multi-core processor model for fast design-space exploration". In: *IEEE Transactions on Computers* 67.6 (2017), pp. 755–770.

[72] Rik Jongerius, Giovanni Mariani, Andreea Anghel, Gero Dittmann, Erik Vermij, and Henk Corporaal. "Analytic processor model for fast design-space exploration". In: *2015 33rd IEEE International Conference on Computer Design (ICCD)*. IEEE. 2015, pp. 411–414.

[73] Ajay Joshi, Aashish Phansalkar, Lieven Eeckhout, and Lizy Kurian John. "Measuring benchmark similarity using inherent program characteristics". In: *IEEE Transactions on Computers* 55.6 (2006), pp. 769–782.

[74] Tejas S Karkhanis and James E Smith. "A first-order superscalar processor model". In: *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.* IEEE. 2004, pp. 338–349.

[75] Daniel Kästner and Stephan Wilhelm. "Generic control flow reconstruction from assembly code". In: *Proceedings of the joint conference on Languages, compilers and tools for embedded systems: software and compilers for embedded systems*. 2002, pp. 46–55.

[76] Yuki Kawarabatake, Mulya Agung, Kazuhiko Komatsu, Ryusuke Egawa, and Hiroyuki Takizawa. "Use of code structural features for machine learning to predict effective optimizations". In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2018, pp. 1049–1055.

[77] Torsten Kempf, Kingshuk Karuri, Stefan Wallentowitz, Gerd Ascheid, Rainer Leupers, and Heinrich 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*. Vol. 1. IEEE. 2006, 6–pp.

[78] Kurt Keutzer, A Richard Newton, Jan M Rabaey, and Alberto Sangiovanni-Vincentelli. "System-level design: orthogonalization of concerns and platform-based design". In: *IEEE transactions on computer-aided design of integrated circuits and systems* 19.12 (2000), pp. 1523–1543.

[79] Bart Kienhuis, Ed Deprettere, Kees Vissers, and Pieter Van Der Wolf. "An approach for quantitative analysis of application-specific dataflow architectures". In: *Proceedings IEEE International Conference on Application-Specific Systems, Architectures and Processors*. IEEE. 1997, pp. 338–349.

[80] Bart Kienhuis, Ed F Deprettere, Pieter Van der Wolf, and Kees Vissers. "A methodology to design programmable embedded systems". In: *International Workshop on Embedded Computer Systems*. Springer. 2001, pp. 18–37.

[81] Donggyu Kim, Adam Izraelevitz, Christopher Celio, Hokeun Kim, Brian Zimmer, Yunsup Lee, et al. "Strober: Fast and accurate sample-based energy simulation for arbitrary RTL". In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2016, pp. 128–139.

[82] Yeseong Kim, Pietro Mercati, Ankit More, Emily Shriver, and Tajana Rosing. "P 4: Phase-based power/performance prediction of heterogeneous systems via neural networks". In: *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2017, pp. 683–690.

[83] Raimund Kirner, Roland Lang, Gerald Freiberger, and Peter Puschner. "Fully automatic worst-case execution time analysis for Matlab/ Simulink models". In: *Proceedings 14th Euromicro Conference on Real-Time Systems. Euromicro RTS 2002*. IEEE. 2002, pp. 31–40.

[84] Raimund Kirner, Roland Lang, Gerald Freiberger, and Peter Puschner. "Fully automatic worst-case execution time analysis for Matlab/ Simulink models". In: *Proceedings 14th Euromicro Conference on Real-Time Systems. Euromicro RTS 2002*. IEEE. 2002, pp. 31–40.

[85] Rajat Kumar, Amit Mankodi, Amit Bhatt, Bhaskar Chaudhury, and Aditya Amrutiya. "Cross-Platform Performance Prediction with Transfer Learning using Machine Learning". In: *2020 11th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*. IEEE. 2020, pp. 1–7.

[86] Chris Lattner. "LLVM: An Infrastructure for Multi-Stage Optimization". MA thesis. Urbana, IL: Computer Science Dept., University of Illinois at Urbana-Champaign, 2002.

[87] Chris Lattner and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *International Symposium on Code Generation and Optimization*. IEEE. 2004.

[88] Mihai T Lazarescu, Jwahar R Bammi, Edwin Harcourt, Luciano Lavagno, and Marcello Lajolo. "Compilation-based software performance estimation for system level design". In: *Proceedings IEEE International High-Level Design Validation and Test Workshop (Cat. No. PR00786)*. IEEE. 2000, pp. 167–172.

[89] Edward Ashford Lee and Sanjit A Seshia. *Introduction to embedded systems: A cyber-physical systems approach*. Mit Press, 2016.

[90] Jong-Yeol Lee and In-Cheol Park. “Timed compiled-code functional simulation of embedded software for performance analysis of SOC design”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 22.1 (2003), pp. 1–14.

[91] Rainer Leupers, Miguel A. Aguilar, Jeronimo Castrillon, and Weihua Sheng. “Software Compilation Techniques for Heterogeneous Embedded Multi-Core Systems”. In: *Handbook of Signal Processing Systems (3rd Edition)*. Ed. by Shuvra S. Bhattacharyya, Ed F. Deprettere, Rainer Leupers, and Jarmo Takala. Springer New York, Sept. 2018, pp. 1021–1062.

[92] Rainer Leupers, Miguel Angel Aguilar, Jeronimo Castrillon, and Weihua Sheng. “Software compilation techniques for heterogeneous embedded multi-core systems”. In: *Handbook of Signal Processing Systems*. Springer, 2019, pp. 1021–1062.

[93] ARM Limited. *ARM big.LITTLE*. URL: `https://www.arm.com/why-arm/technologies/big-little`.

[94] Kai-Li Lin, Chen-Kang Lo, and Ren-Song Tsay. “Source-level timing annotation for fast and accurate TLM computation model generation”. In: *2010 15th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE. 2010, pp. 235–240.

[95] Shuangnan Liu, Francis Lau, and Benjamin Carrion Schafer. “Predictive Compositional Method to Design and Reoptimize Complex Behavioral Dataflows”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.10 (2020), pp. 2615–2627.

[96] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, et al. “The gem5 Simulator: Version 20.0+: A new era for the open-source computer architecture simulator”. In: *ArXivorg* (2020).

[97] Kun Lu, Daniel Müller-Gritschneder, and Ulf Schlichtmann. “Hierarchical control flow matching for source-level simulation of embedded software”. In: *2012 International Symposium on System on Chip (SoC)*. IEEE. 2012, pp. 1–5.

[98] Thomas Lundqvist and Per Stenstrom. “Timing anomalies in dynamically scheduled microprocessors”. In: *Proceedings 20th IEEE Real-Time Systems Symposium*. IEEE. 1999, pp. 12–21.

[99] Thomas Lundqvist and Per Stenstrom. “Timing anomalies in dynamically scheduled microprocessors”. In: *Proceedings 20th IEEE Real-Time Systems Symposium (Cat. No. 99CB37054)*. IEEE. 1999, pp. 12–21.

[100] Hosein Mohammadi Makrani, Farnoud Farahmand, Hossein Sayadi, Sara Bondi, Sai Manoj Pudukotai Dinakarrao, Houman Homayoun, et al. “Pyramid: Machine learning framework to estimate the optimal timing and resource usage of a high-level synthesis design”. In: *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 2019, pp. 397–403.

[101] Hosein Mohammadi Makrani, Hossein Sayadi, Tinoosh Mohsenin, Setareh Rafatirad, Avesta Sasan, and Houman Homayoun. “XPPE: cross-platform performance estimation of hardware accelerators using machine learning”. In: *Proceedings of the 24th Asia and South Pacific Design Automation Conference*. 2019, pp. 727–732.

[102] Amit Mankodi, Amit Bhatt, and Bhaskar Chaudhury. “Performance Prediction of Physical Computer Systems Using Simulation- Based Hardware Models”. In: *2020 International Conference on High Performance Big Data and Intelligent Systems (HPBD&IS)*. IEEE. 2020, pp. 1–5.

[103] Florian Martin. “PAG–an efficient program analyzer generator”. In: *International Journal on Software Tools for Technology Transfer* 2.1 (1998), pp. 46–67.

[104] Florian Martin, Martin Alt, Reinhard Wilhelm, and Christian Ferdinand. “Analysis of loops”. In: *International Conference on Compiler Construction*. Springer. 1998, pp. 80–94.

[105] Grant Martin. “ESL requirements for configurable processor-based embedded system design”. In: *IP-SoC 2005* (2005), pp. 15–20.

[106] Grant Martin. “Overview of the MPSoC design challenge”. In: *2006 43rd ACM/IEEE Design Automation Conference*. IEEE. 2006, pp. 274–279.

[107] Omayma Matoussi and Frédéric Pétrot. “A mapping approach between IR and binary CFGs dealing with aggressive compiler optimizations for performance estimation”. In: *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE. 2018, pp. 452–457.

[108] Thomas J McCabe. “A complexity measure”. In: *IEEE Transactions on software Engineering* 4 (1976), pp. 308–320.

[109] Trevor Meyerowitz, Alberto Sangiovanni-Vincentelli, Mirko Sauermann, and Dominik Langen. “Source-level timing annotation and simulation for a heterogeneous multiprocessor”. In: *Proceedings of the conference on Design, Automation and Test in Europe*. 2008, pp. 276–279.

[110] Joan C Miller and Clifford J Maloney. “Systematic mistake analysis of digital computer programs”. In: *Communications of the ACM* 6.2 (1963), pp. 58–63.

[111] Daniel Mueller-Gritschneder and Andreas Gerstlauer. “Host-Compiled Simulation”. In: *Handbook of Hardware/Software Codesign*. Ed. by Soonhoi Ha and Jürgen Teich. Dordrecht: Springer Netherlands, 2017, pp. 593–619.

[112] Daniel Mueller-Gritschneder, Kun Lu, and Ulf Schlichtmann. “Control-flow-driven source level timing annotation for embedded software models on transaction level”. In: *2011 14th Euromicro Conference on Digital System Design*. IEEE. 2011, pp. 600–607.

[113] Andreas Naderlinger. “Simulating preemptive scheduling with timing-aware blocks in Simulink”. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE. 2017, pp. 758–763.

[114] Andreas Naderlinger. “Simulating preemptive scheduling with timing-aware blocks in Simulink”. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE. 2017, pp. 758–763.

[115] V Nikolskiy and V Stegailov. “Floating-point performance of ARM cores and their efficiency in classical molecular dynamics”. In: *Journal of Physics: Conference Series*. Vol. 681. 1. 2016, p. 012049.

[116] Achim Nohl, Gunnar Braun, Oliver Schliebusch, Rainer Leupers, Heinrich Meyr, and Andreas Hoffmann. “A universal technique for fast and flexible instruction-set architecture simulation”. In: *Proceedings of the 39th annual Design Automation Conference*. 2002, pp. 22–27.

[117] Sebastian Ottlik, Jan Micha Borrmann, Sadik Asbach, Alexander Viehl, Wolfgang Rosenstiel, and Oliver Bringmann. “Trace-based context-sensitive timing simulation considering execution path variations”. In: *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE. 2016, pp. 159–165.

[118] Sebastian Ottlik, Christoph Gerum, Alexander Viehl, Wolfgang Rosenstiel, and Oliver Bringmann. “Context-sensitive timing automata for fast source level simulation”. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE. 2017, pp. 512–517.

[119] Sebastian Ottlik, Stefan Stattelmann, Alexander Viehl, Wolfgang Rosenstiel, and Oliver Bringmann. “Context-sensitive timing simulation of binary embedded software”. In: *Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. 2014, pp. 1–10.

[120] R Palin, D Ward, I Habli, and R Rivett. “ISO 26262 safety cases: compliance and assurance”. In: *IET Conference Proceedings*. The Institution of Engineering & Technology. 2011.

[121] Preeti Ranjan Panda. “SystemC: a modeling platform supporting multiple design abstractions”. In: *Proceedings of the 14th international symposium on Systems synthesis*. 2001, pp. 75–80.

[122] Reena Panda, Xinnian Zheng, Shuang Song, Jee Ho Ryoo, Michael LeBeane, Andreas Gerstlauer, et al. “Genesys: Automatically generating representative training sets for predictive benchmarking”. In: *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*. IEEE. 2016, pp. 116–123.

[123] Charalampos Papamanthou. *Depth First Search & Directed Acyclic Graphs*. University of Maryland, College Park, 2004.

[124] Vladimir-Alexandru Paun, Bruno Monsuez, and Philippe Baufreton. “On the Determinism of Multi-core Processors”. In: *1st French Singaporean Workshop on Formal Methods and Applications*. 2013, p. 32.

[125] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. “Using simpoint for accurate and efficient simulation”. In: *ACM SIGMETRICS Performance Evaluation Review* 31.1 (2003), pp. 318–319.

[126] Stefan M Petters, Patryk Zadarnowski, and Gernot Heiser. “Measurements or static analysis or both?” In: *7th International Workshop on Worst-Case Execution Time Analysis (WCET’07)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2007.

[127] Roman Plyaskin, Thomas Wild, and Andreas Herkersdorf. “System-level software performance simulation considering out-of-order processor execution”. In: *2012 International Symposium on System on Chip (SoC)*. IEEE. 2012, pp. 1–7.

[128] Michael Pressler, Alexander Viehl, Oliver Bringmann, and Wolfgang Rosenstiel. “Execution cost estimation for software deployment in component-based embedded systems”. In: *Proceedings of the 17th international ACM Sigsoft symposium on Component-based software engineering*. 2014, pp. 123–128.

[129] Masudul H Quraishi, Hessam S Sarjoughian, and Soroosh Gholami. “Co-simulation of hardware RTL and software system using FMI”. In: *2018 Winter Simulation Conference (WSC)*. IEEE. 2018, pp. 572–583.

[130] G Ramalingam and Thomas Reps. “An incremental algorithm for maintaining the dominator tree of a reducible flowgraph”. In: *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1994, pp. 287–296.

[131] Rapita Systems Ltd. *Measurement-based timing and WCET analysis with RapiTime*. URL: `https://www.rapitasystems.com/files/MC-PB-101%20RapiTime%20Product%20Brief_1.pdf`.

[132] Jan Reineke and Rathijit Sen. “Sound and efficient WCET analysis in the presence of timing anomalies”. In: *9th International Workshop on Worst-Case Execution Time Analysis (WCET’09)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2009.

[133] Jan Reineke, Björn Wachter, Stefan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, et al. “A definition and classification of timing anomalies”. In: *6th International Workshop on Worst-Case Execution Time Analysis (WCET’06)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2006.

[134] Jan Reineke, Björn Wachter, Stefan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, et al. “A definition and classification of timing anomalies”. In: *6th International Workshop on Worst-Case Execution Time Analysis (WCET’06)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2006.

[135] Mehrdad Reshadi, Prabhat Mishra, and Nikil Dutt. “Instruction set compiled simulation: A technique for fast and flexible instruction set simulation”. In: *Proceedings 2003. Design Automation Conference (IEEE Cat. No. 03CH37451)*. IEEE. 2003, pp. 758–763.

[136] Stefan Resmerita, Patricia Derler, Wolfgang Pree, and Kenneth Butts. *The Validator tool suite: filling the gap between conventional soft-ware-in-the-loop and hardware-in-the-loop simulation environments*. 2012.

[137] Stefan Resmerita, Anton Poelzleitner, and Stefan Lukesch. "Modeling and Simulation of Software Execution Time in Embedded Systems". In: *2020 10th Annual Computing and Communication Workshop and Conference (CCWC)*. IEEE. 2020, pp. 0888–0894.

[138] Adam Rose, Stuart Swan, John Pierce, Jean-Michel Fernandez, et al. "Transaction level modeling in SystemC". In: *Open SystemC Initiative* 1.1.297 (2005).

[139] Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. "Global value numbers and redundant computations". In: *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1988, pp. 12–27.

[140] Christopher J Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. "Dandelion: a compiler and runtime for heterogeneous systems". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 2013, pp. 49–68.

[141] J Rowson and A Sangiovanni-Vincentelli. "System level design". In: *EE Times* (1996).

[142] Farhan Shafiq, Tsuyoshi Isshiki, Dongju Li, and Hiroaki Kunieda. "A Fast Trace Aware Statistical Based Prediction Model with Burst Traffic Modeling for Contention Stall in A Priority Based MPSoC Bus". In: *IPSJ Transactions on System LSI Design Methodology* 9 (2016), pp. 37–48.

[143] Timothy Sherwood and Brad Calder. "Time varying behavior of programs". In: *In UC San Diego* (1999).

[144] Timothy Sherwood, Erez Perelman, and Brad Calder. "Basic block distribution analysis to find periodic behavior and simulation points in applications". In: *Proceedings 2001 International Conference on Parallel Architectures and Compilation Techniques*. IEEE. 2001, pp. 3–14.

[145] Kohta Shigenobu, Kanemitsu Ootsu, Takeshi Ohkawa, and Takashi Yokota. "A translation method of ARM machine code to LLVM-IR for binary code parallelization and optimization". In: *2017 Fifth International Symposium on Computing and Networking (CANDAR)*. IEEE. 2017, pp. 575–579.

[146] Shuang Song, Qinzhe Wu, Steven Flolid, Joseph Dean, Reena Panda, Junyong Deng, et al. "Experiments with spec cpu 2017: Similarity, balance, phase b ehavior and simpoints". In: ().

[147] Rafael Stahl, Daniel Mueller-Gritschneder, and Ulf Schlichtmann. "Automated Redirection of Hardware Accesses for Host-Compiled Software Simulation". In: *2018 Forum on Specification & Design Languages (FDL)*. IEEE. 2018, pp. 5–16.

[148] Richard Stallman, Roland Pesch, Stan Shebs, et al. "Debugging with GDB". In: *Free Software Foundation* 675 (1988).

[149] Stefan Stattelmann, Oliver Bringmann, and Wolfgang Rosenstiel. "Dominator homomorphism based code matching for source-level simulation of embedded software". In: *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. 2011, pp. 305–314.

[150] Stefan Stattelmann, Oliver Bringmann, and Wolfgang Rosenstiel. "Fast and accurate source-level simulation of software timing considering complex code optimizations". In: *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE. 2011, pp. 486–491.

[151] Stefan Stattelmann, Gernot Gebhard, Christoph Cullmann, Oliver Bringmann, and Wolfgang Rosenstiel. "Hybrid source-level simulation of data caches using abstract cache models". In: *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2012, pp. 376–381.

[152] Stefan Stattelmann, Sebastian Ottlik, Alexander Viehl, Oliver Bringmann, and Wolfgang Rosenstiel. “Combining instruction set simulation and wcet analysis for embedded software performance estimation”. In: *7th IEEE International Symposium on Industrial Embedded Systems (SIES’12)*. IEEE. 2012, pp. 295–298.

[153] Stefan Stattelmann, Alexander Viehl, Oliver Bringmann, and Wolfgang Rosenstiel. “Towards Accurate Source-Level Annotation of Low-Level Properties Obtained from Optimized Binary Code”. In: *System Specification and Design Languages*. Springer, 2012, pp. 175–190.

[154] Sam Van den Steen, Sander De Pestel, Moncef Mechri, Stijn Eyerman, Trevor Carlson, David Black-Schaffer, et al. “Micro-architecture independent analytical processor performance and power modeling”. In: *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2015, pp. 32–41.

[155] Radare Team. “Radare2 book”. In: *GitHub*. 2017. URL: `https://github.com/radareorg/radare2`.

[156] DRVLB Thambawita, Roshan G Ragel, and Dhammike Elkaduwe. “To use or not to use: CPUs’ cache optimization techniques on GPGPUs”. In: *2016 IEEE International Conference on Information and Automation for Sustainability (ICIAfS)*. IEEE. 2016, pp. 1–6.

[157] The LLVM Compiler Infrastructure. *Debugging JIT-ed Code*. URL: `https://llvm.org/docs/DebuggingJITedCode.html`.

[158] The LLVM Compiler Infrastructure. *LLVM’s Analysis and Transform Passes*. URL: `https://llvm.org/docs/Passes.html` (visited on 09/15/2020).

[159] The LLVM Compiler Infrastructure. *Machine IR (MIR) Format Reference Manual*. URL: `https://llvm.org/docs/MIRLangRef.html`.

[160] The LLVM Compiler Infrastructure. *The LLVM bitcode execution engine: lli directly executes programs from LLVM bitcode*. URL: `https://llvm.org/docs/CommandGuide/lli.html`.

[161] The LLVM Compiler Infrastructure. *The LLVM Target-Independent Code Generator*. URL: `https://llvm.org/docs/CodeGenerator.html`.

[162] The LLVM Compiler Infrastructure. *Writing an LLVM Pass*. URL: `https://llvm.org/docs/WritingAnLLVMPass.html`.

[163] The MathWorks Inc. *MATLAB - Communicate Using TCP/IP Server Sockets*. URL: `https://www.mathworks.com/help/instrument/communicate-using-tcpip-server-sockets.html`.

[164] The MathWorks Inc. *MATLAB Simulink - Embedded Coder*. URL: `https://www.mathworks.com/products/embedded-coder.html`.

[165] The MathWorks Inc. *MATLAB Simulink - Modeling a Fault-Tolerant Fuel Control System*. URL: `https://www.mathworks.com/help/simulink/slref/modeling-a-fault-tolerant-fuel-control-system.html`.

[166] The MathWorks Inc. *MATLAB Simulink - Simulation and Model-Based Design*. URL: `https://www.mathworks.com/products/simulink.html`.

[167] The MathWorks Inc. *MATLAB Simulink - Stateflow: Model and simulate decision logic using state machines and flow charts*. URL: `https://www.mathworks.com/products/stateflow.html`.

[168] Henrik Theiling. “Control flow graphs for real-time systems analysis”. In: *Universität des Saarlandes, Diss* (2002).

[169] Theo Ungerer, Francisco Cazorla, Pascal Sainrat, Guillem Bernat, Zlatko Petrov, Christine Rochange, et al. “Merasa: Multicore execution of hard real-time applications supporting analyzability”. In: *IEEE Micro* 30.5 (2010), pp. 66–75.

[170] Harry Wagstaff. “From High Level Architecture Descriptions to Fast Instruction Set Simulators”. In: (2015).

[171] Zhonglei Wang and Jörg Henkel. “Accurate source-level simulation of embedded software with respect to compiler optimizations”. In: *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2012, pp. 382–387.

[172] Zhonglei Wang and Andreas Herkersdorf. “An efficient approach for system-level timing simulation of compiler-optimized embedded software”. In: *2009 46th ACM/IEEE Design Automation Conference*. IEEE. 2009, pp. 220–225.

[173] Vincent M Weaver and Sally A McKee. “Are cycle accurate simulations a waste of time”. In: *Proc. 7th Workshop on Duplicating, Deconstructing, and Debunking*. 2008, pp. 40–53.

[174] Ingomar Wenzel, Raimund Kirner, Bernhard Rieder, and Peter Puschner. “Measurement-based timing analysis”. In: *International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*. Springer. 2008, pp. 430–444.

[175] Stephan Werner, Leonard Masing, Fabian Lesniak, and Jürgen Becker. “Software-in-the-loop simulation of embedded control applications based on virtual platforms”. In: *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 2015, pp. 1–8.

[176] Thomas Wild, Andreas Herkersdorf, and Gyoo-Yeong Lee. “TAPES—Trace-based architecture performance evaluation with SystemC”. In: *Design Automation for Embedded Systems* 10.2 (2005), pp. 157–179.

[177] Reinhard Wilhelm. “Determining bounds on execution times”. In: *Handbook on Embedded Systems* (2009).

[178] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, et al. “The worst-case execution-time problem—overview of methods and survey of tools”. In: *ACM Transactions on Embedded Computing Systems (TECS)* 7.3 (2008), pp. 1–53.

[179] Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. “Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28.7 (2009), pp. 966–978.

[180] Michael E Wolf, Dror E Maydan, and Ding-Kai Chen. “Combining loop transformations considering caches and scheduling”. In: *Proceedings of the 29th IEEE/ACM International Symposium on Microarchitecture. MICRO 29*. IEEE. 1996, pp. 274–286.

[181] Xilinx. *ZedBoard Zynq-7000 ARM/FPGA SoC Development Board*. URL: `https://www.xilinx.com/products/boards-and-kits/1-elhabt.html`.

[182] Hideaki Yanagisawa, Minoru Uehara, and Hideki Mori. “Automatic generation of a simulation compiler by a HW/SW codesign system”. In: *Proceedings. 15th IEEE International Workshop on Rapid System Prototyping, 2004.* IEEE. 2004, pp. 53–59.

[183] Youngmin Yi, Dohyung Kim, and Soonhoi Ha. “Fast and accurate cosimulation of MPSoC using trace-driven virtual synchronization”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26.12 (2007), pp. 2186–2200.

[184] Joon-Seo Yim, Yoon-Ho Hwang, Chang-Jae Park, Hoon Choi, Woo-Seung Yang, Hun-Seung Oh, et al. “A C-based RTL design verification methodology for complex microprocessor”. In: *Proceedings of the 34th annual Design Automation Conference*. 1997, pp. 83–88.

[185] Zhuoran Zhao, Andreas Gerstlauer, and Lizy K John. “Source-level performance, energy, reliability, power and thermal (PERPT) simulation”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 36.2 (2016), pp. 299–312.

[186] Xinnian Zheng, Lizy K John, and Andreas Gerstlauer. "Lacross: Learning-based analytical cross-platform performance and power prediction". In: *International Journal of Parallel Programming* 45.6 (2017), pp. 1488–1514.

[187] Jianwen Zhu and Daniel D Gajski. "A retargetable, ultra-fast instruction set simulator". In: *Proceedings of the conference on Design, automation and test in Europe*. 1999, 62–es.

[188] Vojin Zivojnovic and Heinrich Meyr. "Compiled HW/SW co-simulation". In: *33rd Design Automation Conference Proceedings, 1996*. IEEE. 1996, pp. 690–695.