TECHNISCHE UNIVERSITÄT MÜNCHEN
Lehrstuhl für Integrierte Systeme

FAST AND ACCURATE PERFORMANCE SIMULATION OF OUT-OF-ORDER PROCESSING CORES IN EMBEDDED SYSTEMS

Roman Plyaskin

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

genehmigten Dissertation.

Die Dissertation wurde am 30.10.2013 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 23.06.2014 angenommen.

ABSTRACT

Recent embedded systems employ multiple processing cores on a single chip. Such multiprocessor system-on-chip (MPSoC) can incorporate heterogeneous processing cores with diverse internal complexity in order to offer an optimized solution in terms of performance, power consumption and dependability. Considering the growing MPSoC complexity, system architects require *fast* and *accurate* performance simulation of out-of-order processing cores to enable design space exploration (DSE) of MPSoC in reasonable time. Interpretive cycle-accurate instruction set simulators (ISS) employ detailed models of the core microarchitecture. Therefore, their use in iterative system-level DSE of MPSoC is limited because of low simulation speed. At the same time, recent approaches for fast software performance simulation leveraging annotated source code are too abstract to consider the effects of out-of-order instruction execution.

This thesis addresses the gap between interpretive cycle-accurate ISS and abstract source-level simulation and presents a novel approach for software performance simulation considering out-of-order execution. The proposed approach enables fast and accurate reproduction of the processor's out-of-order behavior and accelerates DSE at the system level. Furthermore, the thesis presents a SystemC-based simulation tool for performance evaluation of multicore architectures. The tool additionally supports trace-driven simulation of target applications and incorporates a high-level scheduler for flexible evaluation of SW partitioning in MPSoC platforms.

ZUSAMMENFASSUNG

Moderne eingebettete Systeme setzen Chips mit mehreren Prozessorkernen ein. Solche Multiprozessor Systeme-on-Chip (MPSoC) können heterogene Prozessorkerne mit diverser interner Komplexität enthalten und somit eine optimierte Lösung im Bezug auf Performanz, Leistungsverbrauch und Zuverlässigkeit bieten. Wegen der steigenden Komplexität der MPSoC benötigen Systemarchitekten eine schnelle und genaue Performanzsimulation der Out-of-Order-Prozessorkerne, um eine Entwurfsraumexploration der MPSoC in annehmbaren Zeiten zu ermöglichen. Interpretierende, zyklenakkurate Instruktionssatz-Simulatoren (ISS) enthalten detaillierte Modelle der Mikroarchitektur des Prozessorkerns und können nur kleine Simulationsgeschwindigkeiten erreichen. Aus diesem Grund ist die Nutzung dieser Werkzeuge für iterative Entwurfsraumexplorationen der MPSoC auf Systemebene begrenzt. Neueste Ansätze, die eine schnelle Software-Performanzsimulation basierend auf annotierten Quellcodes ermöglichen, sind zu abstrakt, um die Effekte der Out-of-Order-Ausführung der Instruktionen zu berücksichtigen.

Diese Dissertation stellt ein neues Verfahren für eine Software-Performanzsimulation mit Berücksichtigung der Out-of-Order-Ausführung vor und schließt somit die Lücke zwischen den interpretierenden, zyklenakkuraten ISS und einer abstrakten Simulation auf Quellcode-Ebene. Der vorgeschlagene Ansatz ermöglicht eine schnelle und genaue Wiedergabe des Verhaltens der Out-of-Order-Prozessoren und beschleunigt Entwurfsraumexplorationen auf Systemebene. Außerdem stellt diese Dissertation ein SystemC-basiertes Werkzeug für Performanzabschätzungen der Mehrkern-Architekturen vor. Das Werkzeug ermöglicht zusätzlich eine Trace-getriebene Simulation der Zielanwendungen und enthält einen Scheduler für flexible Evaluierungen der Software-Partitionierungen in MPSoC-Plattformen auf hoher Ebene.

ACKNOWLEDGMENTS

I would like to sincerely thank Prof. Andreas Herkersdorf for supervising this thesis and providing me with the opportunity to work and do the research at the institute. I particularly appreciate our discussions on this project, which kept me motivated and helped me to gain a more comprehensive view on the research problems. I also would like to express my gratitude to Prof. Wolfgang Rosenstiel for co-examining this work and his comments which helped me to improve the dissertation.

In addition, I would like to thank Prof. Walter Stechele and Dr. Thomas Wild for their help and valuable comments on this work. Special thanks go to all present and past colleagues at the institute for integrated systems for creating a nice working atmosphere. Finally, this work wouldn't be possible without loving support of my family and my fiancée Michaela who always stood by my side during these years.

CONTENTS

1 INTRODUCTION 1
- 1.1 Scope of the thesis 1
- 1.2 Problem statement 2
- 1.3 Contributions 3
- 1.4 Outline 4

2 PRIOR ART 5
- 2.1 Instruction Set Simulation 5
  - 2.1.1 Interpretive ISS 5
  - 2.1.2 Compiled ISS 6
  - 2.1.3 Sampled simulation 10
  - 2.1.4 ISS-based simulation of multicore architectures 12
- 2.2 Simulation based on target code 14
  - 2.2.1 Source-level simulation 14
  - 2.2.2 Simulation based on intermediate representation 16
  - 2.2.3 Instruction-level simulation 18
- 2.3 Trace-driven simulation 20
- 2.4 High-level OS modeling 23
- 2.5 Summary 28

3 BACKGROUND OF COMPILED SW SIMULATION AT BINARY LEVEL 31
- 3.1 Functional behavior 32
  - 3.1.1 Binary-to-C translation 32
  - 3.1.2 Organization of translated code 33
  - 3.1.3 Modeling of target memory 34
- 3.2 Timing behavior 36
  - 3.2.1 Annotation of timing information 36
  - 3.2.2 Modeling of caches 37

4 COMPILED SW SIMULATION CONSIDERING OUT-OF-ORDER INSTRUCTION EXECUTION 41
- 4.1 System-level effects of out-of-order execution 41
  - 4.1.1 Classification of out-of-order effects 41
  - 4.1.2 Limitations of conventional host-compiled simulations 44
- 4.2 Derivation of basic block timing 47
  - 4.2.1 ISS enhancements 47
  - 4.2.2 Identification of basic block boundaries 50
- 4.3 Context dependency of basic block timing 51
  - 4.3.1 Concept 51
  - 4.3.2 Derivation of context-dependent timing 53

4.3.3 Context-aware host-compiled simulation 56
4.4 Reordering of memory accesses 57
4.4.1 Classification of basic blocks 57
4.4.2 Simulation of memory reordering 59
4.5 Non-blocking behavior of data cache 61
4.5.1 Modeling of non-blocking behavior 61
4.5.2 Dependency analysis of memory instructions . 68
4.6 Optimizations 69
4.6.1 Optimization of binary-to-C translation 71
4.6.2 Averaging of basic block timings 73
4.6.3 Static reordering of memory accesses 74

5 SYSTEM-LEVEL SIMULATION OF MULTICORE ARCHITECTURES 85
5.1 Simulation methods 86
5.1.1 Trace-Driven Simulation 86
5.1.2 Binary-level simulation 91
5.1.3 Summary 93
5.2 High-level scheduler model 96
5.2.1 Task model 97
5.2.2 Scheduler model 99
5.2.3 Implementation details 109
5.3 SystemC models of hardware components 115
5.3.1 Out-of-order core 115
5.3.2 Communication infrastructure and memory . . 123

6 EXPERIMENTAL RESULTS 129
6.1 Context-aware host-compiled SW simulation 129
6.1.1 Experimental setup 129
6.1.2 Optimization of binary-to-C translation 132
6.1.3 Context-aware compiled simulation 135
6.1.4 Averaging of basic block timing 143
6.1.5 Consideration of a data cache 147
6.2 System-level simulation based on SystemC 153
6.2.1 Experimental setup 153
6.2.2 Evaluation of BLS and TDS methods 155
6.3 Multicore design space exploration: A use case 163
6.3.1 Experimental setup 163
6.3.2 Results 166

7 CONCLUSIONS AND FUTURE WORK 171
7.1 Conclusions 171
7.2 Outlook for future work 175

BIBLIOGRAPHY 177

LIST OF FIGURES

Figure 3.1 Workflow of binary-level SW compiled simulation . . . 31
Figure 3.2 Binary-to-C translation . . . 32
Figure 3.3 Timing annotation . . . 37
Figure 3.4 Calls to dynamic cache models . . . 38
Figure 4.1 Context-dependent timing of basic blocks . . . 42
Figure 4.2 Reordering of memory accesses . . . 44
Figure 4.3 Error of conventional compiled simulation . . . 45
Figure 4.4 Timing behavior during a data cache miss . . . 46
Figure 4.5 Derivation of basic block timing . . . 47
Figure 4.6 Consideration of overlapping of basic blocks . . . 50
Figure 4.7 Contexts of basic blocks . . . 52
Figure 4.8 Context-aware measurement . . . 53
Figure 4.9 Differentiation of basic block instances . . . 54
Figure 4.10 Algorithm for obtaining context-dependent timing . . . 55
Figure 4.11 Context-aware basic block function . . . 56
Figure 4.12 Reordering of memory accesses . . . 58
Figure 4.13 Detection of possible memory reordering over multiple basic blocks . . . 59
Figure 4.14 Simulation of reordered memory accesses . . . 60
Figure 4.15 Example code with data dependency . . . 62
Figure 4.16 Hit under dependent miss . . . 63
Figure 4.17 Second miss under a dependent miss . . . 65
Figure 4.18 Simulation of independent miss . . . 66
Figure 4.19 Consideration of instruction window . . . 67
Figure 4.20 Dependency analysis . . . 68
Figure 4.21 Total amount of basic block timings . . . 70
Figure 4.22 Optimization of context signatures . . . 73
Figure 4.23 Averaging of block timings . . . 74
Figure 4.24 Simulation of exchanged accesses to the same cache line . . . 75
Figure 4.25 Simulation of exchanged accesses to different cache lines with no dependency . . . 76
Figure 4.26 Simulation of exchanged accesses to different cache lines with dependency . . . 78
Figure 4.27 Static reordering of memory accesses . . . 79
Figure 4.28 Types of basic block overlapping . . . 80
Figure 4.29 Block overlapping without data cache accesses 83
Figure 4.30 Measurement of basic block timings with statically reordered accesses . . . 84

Figure 5.1 SystemC simulator 85
Figure 5.2 Generation of an abstract trace 88
Figure 5.3 Trace-driven simulation of multicores 89
Figure 5.4 Different levels of trace abstraction 91
Figure 5.5 Different abstraction levels of traces. 91
Figure 5.6 Binary-level compiled simulation of multicores 92
Figure 5.7 Comparison of TDS and BLS methods 95
Figure 5.8 State diagram of tasks. 98
Figure 5.9 Structure of the OS scheduler model 99
Figure 5.10 Algorithm for generation of instances of user tasks 101
Figure 5.11 Algorithm for scheduling of user tasks 102
Figure 5.12 Preemption of user tasks 103
Figure 5.13 Termination of user tasks 104
Figure 5.14 Simulation of OS ticks without task switching 104
Figure 5.15 Synchronization of user tasks 106
Figure 5.16 Adding synchronization to TDS tasks 108
Figure 5.17 Adding synchronization to BLS tasks 108
Figure 5.18 Class hierarchy of TDS- and BLS-tasks. 110
Figure 5.19 Execution of a TDS task in SystemC. 111
Figure 5.20 Memory modeling in case of multiple target applications 113
Figure 5.21 Mapping of virtual and physical address spaces 114
Figure 5.22 Abstracted core model 116
Figure 5.23 Queue in the non-blocking cache extension . . 117
Figure 5.24 Behavior of the non-blocking cache extension on an data access 118
Figure 5.25 Behavior of the non-blocking cache extension after simulating a miss penalty 119
Figure 5.26 Stalling due to the limited size of the instruction window 120
Figure 5.27 Stalling due to data dependency 121
Figure 5.28 Database in the cache model 122
Figure 5.29 Model of arbitrated bus 123
Figure 5.30 Bus arbitration process 124
Figure 5.31 States in MESI protocol 125
Figure 5.32 Cache coherency extension 126
Figure 5.33 Operation in case of a write hit 127
Figure 6.1 Workflow of experiments 131
Figure 6.2 Optimization of binary-to-C translation 133
Figure 6.3 Amount of basic block contexts 134
Figure 6.4 Size of translated code 134
Figure 6.5 Compilation time 134
Figure 6.6 Timing error of context-aware simulation . . . 136
Figure 6.7 Average timing error 136
Figure 6.8 Speed of context-aware simulation 136

Figure 6.9 Average speed of context-aware simulation . . 136
Figure 6.10 Reorder depth 138
Figure 6.11 RMSE of reorder depth 140
Figure 6.12 Mean RMSE of reorder depths 140
Figure 6.13 Percentage of out-of-order blocks 141
Figure 6.14 Deterioration of simulation time 142
Figure 6.15 Mean deterioration of simulation time 142
Figure 6.16 Timing error of context-aware simulation with averaging 143
Figure 6.17 Mean timing error with averaging 144
Figure 6.18 Speed of context-aware compiled simulation with averaging 144
Figure 6.19 Mean speed of context-aware compiled simulation with averaging 145
Figure 6.20 RMSE of reorder depths with averaging 146
Figure 6.21 Mean RMSE of reorder depths with averaging 146
Figure 6.22 Simulation scenarios with data cache 148
Figure 6.23 Accuracy of simulation with static memory reordering 149
Figure 6.24 Estimated miss rate of data cache 149
Figure 6.25 Simulation speed with static memory reordering 150
Figure 6.26 Percentage of reorder depths 151
Figure 6.27 Average percentage of reorder depths 152
Figure 6.28 valuation of cache accesses 152
Figure 6.29 Experimental setup for BLS and TDS 154
Figure 6.30 Speed of TDS and BLS simulation 156
Figure 6.31 Simulation error of TDS and BLS 157
Figure 6.32 Rate of accesses to on-chip interconnect 157
Figure 6.33 Accuracy of data cache simulation in BLS . . . 158
Figure 6.34 Accuracy of instruction cache simulation in BLS 159
Figure 6.35 Simulation of random bus traffic 160
Figure 6.36 Utilization of data/instruction bus 160
Figure 6.37 Estimated time and accuracy of BLS 162
Figure 6.38 Simulation speed of a multicore architecture . 163
Figure 6.39 Task graph of JPEG encoding 164
Figure 6.40 Pareto front 167
Figure 6.41 Distributions of execution time 168
Figure 6.42 Scheduling diagram 169
Figure 6.43 Invalidations per bus master 169

LIST OF TABLES

Table 2 Content of the task queue 107

Table 3 Configuration of the core simulated in the SimpleScalar. 130

Table 4 Configuration of a data cache employed in simulations 147

Table 5 Evaluation results of the JPEG application . . . 167

ACRONYMS

ALU	Arithmetic and Logic Unit
API	Application Programmable Interface
BCET	Best-case Execution Time
BLS	Binary-Level Compiled Simulation
CFG	Control-Flow Graph
CPI	Cycles per Instruction
DBT	Dynamic Binary Translation
DCT	Discrete Cosine Transform
DSE	Design Space Exploration
EDF	Earliest Deadline First
EDS	Execution-Driven Simulation
EU	Execution Unit
FIFO	First In First Out
GUI	Graphical User Interface
HDL	Hardware Description Language
IO	Input/Output
IPC	Instructions per Cycle
IR	Intermediate Representation
ISA	Instruction Set Architecture
ISS	Instruction Set Simulator
IW	Instruction Window
LSQ	Load/Store Queue
MESI	Modified, Exclusive, Shared, Invalid
MIT	Mean Interarrival Time
MPSoC	Multiprocessor System-on-Chip
MSHR	Miss Status Holding Register

MSI	Modified, Shared, Invalid
OS	Operating System
RAW	Read After Write
RISC	Reduced Instruction Set Computer
RMSE	Root Mean Square Error
RTOS	Real-Time Operating System
SLS	Source-Level Simulation
SMP	Symmetric Multiprocessing
SoC	System-on-Chip
TDS	Trace-Driven Simulation
TLM	Transaction-level Modeling
WAR	Write after Read
WAW	Write after Write
WCET	Worst-case Execution Time

1 INTRODUCTION

1.1 SCOPE OF THE THESIS

Recent embedded systems leverage chips with multiple processing cores to improve the performance of target applications and reduce power consumption. Such multiprocessor systems-on-chip (MPSoCs) can have a heterogeneous architecture and incorporate processing cores with diverse complexity to offer an optimized solution in terms of performance, power consumption and dependability. For example, a heterogeneous MPSoC may incorporate processing cores with a *simple* microarchitecture optimized in terms of chip area and power consumption. In addition, the MPSoC may contain processing cores with a *complex* microarchitecture optimized for performance. Complex cores may incorporate multiple execution units, support branch prediction and dynamic scheduling of instructions for faster execution of the software.

The heterogeneous MPSoC structure allows for more efficient distribution of software execution on the processing cores. For example, parallel software parts can be executed on multiple simple cores to efficiently exploit thread-level parallelism. In turn, sequential software parts, which are inevitable due to the limited parallelism in the target software, can be executed on fewer complex but faster cores to profit from instruction-level parallelism as much as possible. Advanced processing cores capable of out-of-order instruction execution have been recently emerged in embedded systems, e.g. Freescale e5500, ARM Cortex-R7 or Cortex-A15.

Furthermore, because of the increasing performance capabilities of MPSoCs, the complexity of target applications executed in embedded devices is steadily increasing. As as a result, design of MPSoCs involves many decisions that need to be made both on the software and hardware side. For example, a hardware designer may need to find a suitable hardware architecture with respect to allocation of processing elements and interconnect in order to meet the application requirements. Moreover, target applications can be composed of multiple communicating tasks. In this case, a software designer may need to explore various mappings of the tasks on the underlying hardware architecture, e.g. while trying to reduce the overall execution time of the target application. In this evaluation process, a key requirement is the flexibility during evaluation of SW partitioning. Different task mappings and scheduling strategies need to be studied at early stages, without the need of porting the applications to a specific implementation of an operating system (OS). Consequently, the developers are faced with a large amount of possible design solutions. In order to enable exploration of design alternatives and rapidly obtain their performance characteristics at early stages, performance modeling and simulation of MPSoC architectures at the system level is necessary.

1.2 PROBLEM STATEMENT

Efficient system-level simulation of multicore architectures is a contradicting problem as it requires *fast* and at the same time *accurate* performance models of processing cores. For many years, a typical approach for conducting performance simulation of the target software has been to use cycle-accurate instruction set simulation (ISS). Cycle-accurate simulators incorporate a detailed model of the processor's microarchitecture. In this type of simulators, fetching, decoding and scheduling of instructions is performed dynamically at simulation run-time. Cycle-accurate simulators are indispensable for exploration of the processor's internal microarchitecture. However, the use of these simulators during system-level design space exploration (DSE) of MPSoC is limited because of their low simulation speed. Consequently, for fast system-level performance analysis, the abstraction level of processor models must be raised.

On the other side of the spectrum are recently introduced host-compiled source-level simulation methods. These methods are based on a native execution of the target code on the host computer. In these approaches, timing simulation of the target software is enabled by annotating pre-estimated execution delays in the target code. Source-level techniques can efficiently abstract the microarchitectural details of processing cores and achieve significantly higher simulation speed compared to conventional cycle-accurate ISS. However, current approaches assume processors with in-order instruction execution only. At the moment, there is a gap in performance simulation of out-of-order processors between very accurate but slow cycle-accurate ISS and fast but very abstract host-compiled source-level simulation. To improve the efficiency of system-level DSE of multiprocessor architectures, new methodology for performance simulation is required that would consider out-of-order execution of instructions, while achieving speed higher than cycle-accurate ISS.

Moreover, one of the major goals of system-level DSE is to assess different options of SW partitioning in a multicore architecture, i.e. to investigate various task mappings and scheduling strategies, and to derive the requirements for a future OS implementation. Many recent approaches, that address high-level OS modeling at early design stages, consider the execution time of tasks at very high granularity. Particularly, they abstract hardware timing effects during a task execution, e.g. temporal behavior of coherent caches or arbitration on the shared interconnect, and model application tasks as a set of fixed processing latencies. For more comprehensive performance analysis of SW partitioning, the low-level hardware effects of the tasks' execution must be taken into consideration, e.g. cache coherency. Cycle-accurate processor simulators are not applicable for these investigations due to their low simulation speed, as mentioned earlier.

1.3 CONTRIBUTIONS

The goal of this thesis is *fast* and yet *accurate* performance simulation of out-of-order processors at the system-level that enables design space exploration of multiprocessor system-on-chip in reasonable time.

Firstly, I introduce a novel approach for performing host-compiled SW performance simulation that considers out-of-order instruction execution. The proposed method is based on simulation of the target code at the binary (i.e. instruction) level. The simulation is performed on the host computer by translating target instructions into equivalent C code, which is then annotated with timing information. The thesis presents the methodology for obtaining and annotating the execution delays as well as considering the timing effects of out-of-order execution at simulation run-time. In particular, context-dependent execution time of instructions as well as reordering of memory accesses are tackled. To the best of my knowledge, it is the first approach that addresses out-of-order instruction execution in host-compiled SW simulation.

Secondly, I present a SystemC-based simulation framework for performance evaluation of multiprocessor architectures at the system level. In addition to the host-compiled binary-level SW simulation (BLS) method mentioned earlier, the framework supports trace-driven simulation (TDS). TDS completely abstracts the internal microarchitecture of processing cores. In contrast to BLS, TDS employs abstract application traces and does not require functional simulation of the target code. At the same time, TDS provides pin-accuracy of the communication requests and allows detailed evaluation of contentions in the chip interconnect. In the sequel of the thesis, we will see that out-of-order effects are exposed differently at the system level for various target applications. As a result, TDS offers a better trade-off between simulation speed and accuracy than BLS for certain target applications.

Finally, I present a high-level model of a scheduler. The scheduler model is a component of the simulation framework that enables evaluation of different SW partitioning options in multicore architectures. The scheduler model enables high-level management of application tasks and provides capabilities for inter-task synchronization. At the same time, the workload of the application tasks is simulated with either BLS or TDS method, while considering the low-level hardware timing effects and still retaining the flexibility for high-level task management. Concepts presented in this dissertation have been published in [42, 59, 60, 61, 62].

1.4 OUTLINE

The thesis is structured as follows. Chapter 2 provides an overview of the state-of-the art techniques for SW performance evaluation. As the simulation method presented in this dissertation employs host-compiled simulation at the binary level, in Chapter 3 I briefly review the basics and the workflow of binary-level host-compiled simulation. Chapter 4 presents details of the proposed approach.

Chapter 5 describes in detail the proposed SystemC-based simulation framework. Particularly, I discuss the employment of BLS and TDS methods for system-level performance simulation of multicore architectures. Furthermore, I present implementation details of SystemC models of a high-level scheduler model, out-of-order processing cores, arbitrated on-chip bus and memory. Chapter 6 presents evaluation results of the BLS and TDS methods. In addition, it describes a use case of system-level DSE of a parallelized JPEG application on different multicore platforms. Finally, Chapter 7 concludes the dissertation with a discussion on the obtained results and presents an outlook on future work.

2 PRIOR ART

This chapter presents an overview of the related work in SW performance evaluation at different levels of abstraction, starting from cycle-accurate instruction set simulation (ISS) up to highly abstracted simulation of application models during system-level design. In the first part, I provide an overview of different ISS approaches and discuss recent advancements in this methodology. Afterwards, I review techniques for performance simulation at a higher abstraction level based on a native host-compiled execution of the annotated target code. Both ISS and host-compiled simulation are *execution-driven* techniques. In addition, we will retrospect estimation methods based on *trace-driven* performance simulation. Finally, I review recent approaches for the timing analysis of complex multi-tasking applications by means of a high-level model of a real-time operating system.

2.1 INSTRUCTION SET SIMULATION

Instruction set simulation allows for accurate evaluation of the SW execution on a target processor and can be employed for various purposes. For example, instruction set simulators are indispensable tools for evaluating the design of processors and compilers [50, 54]. Particularly, the processor designer can assess the efficiency of specific instructions in the instruction set and make decisions on the parameters of microarchitectural components in a processor, e.g. dimensions of internal memory structures or type of branch prediction [50]. SW developers can employ ISS to validate the functionality of the target software before the actual HW implementation becomes available [5]. The review starts with *interpretive* instruction set simulation.

2.1.1 *Interpretive ISS*

In interpretive ISS, the simulation of target instructions is organized in a loop. In this loop, the processor model sequentially performs instruction fetching, decoding as well as performing the actual operation according to the semantics of the instructions [41]. A. Nohl *et al.* claimed that the simulation of the instructions in the loop "enables the highest degree of simulation accuracy and flexibility" [54].

SimpleScalar [5] is an example of interpretive ISS. The suite consists of multiple tools, which incorporate processor models at different levels of abstraction. At the highest abstraction level (*sim-fast* tool), only functional processor simulation is performed without any notion of timing. At the lowest abstraction level (*sim-outorder*), the tool incorporates a detailed cycle-accurate model of an out-of-order processor capable of speculative execution. For accurate timing evaluation, *sim-outorder* employs performance models of internal microarchitectural components, e.g. instruction pipeline, instruction queues, branch predictor and caches. The simulator supports several instruction sets including ARM, PPC, Alpha and PISA (a slightly modified version of the MIPS-IV ISA [13]). In [14], D. Burger *et al.* presented further extensions to SimpleScalar that enable more detailed modeling of memory hierarchies and bus interconnect. The simulator has been widely adopted by computer architecture researchers as stated in [5].

Gem5 [9] (formerly M5 [10]) is a simulation framework that provides a range of processor models at different abstraction levels, including a model of a complex pipelined processing core with multiple hardware threads and capable of out-of-order execution. The processor models support multiple instruction set architectures including MIPS, ARM, Alpha and x86. In addition, the simulator allows modeling and evaluation of complex hierarchies of coherent caches and memories. The tool can perform a full-system simulation of target applications, considering IO devices and an operating system.

The major drawback of interpretive ISS is a low simulation speed, which, among other factors, results from time-intensive decoding of target instructions at simulation run-time [54].

2.1.2 *Compiled ISS*

Compiled instruction set simulators try to optimize simulation performance by moving the frequent operation of instruction decoding from run-time to compile-time. Two approaches of performing compiled simulation can be differentiated: static compilation of the target instructions and dynamic binary translation (DBT).

2.1.2.1 *Statically compiled ISS*

The key idea behind statically compiled ISS is to translate the *complete* code of the target application into an intermediate, functionally equivalent representation. The intermediate code is then compiled and executed on the host computer.

In [50], C. Mills *et al.* suggested to employ static compilation to improve the simulation performance. The proposed method uses macro-statements and case statements of the C programming language to translate each target instruction into functionally equivalent C-operations. Afterwards, the resulting C-code is compiled and executed on the host machine at higher simulation speed than conventional interpretive ISS. The authors showed that the instruction compilation prior to simulation allows achieving a simulation speedup of 3 times compared to interpretive ISS. The suggested approach requires additional efforts for code compilation on the host. However, the associated timing overhead can be amortized if the target program must be simulated multiple times.

J. Zhu and D. Gajski in [90] proposed a statically compiled ISS based on intermediate instructions of a RISC-like *virtual machine*. In this approach, the target program is translated to virtual instructions which perform operations on a set of virtual registers. Afterwards, the virtual instructions are used to generate either host instructions or a C-code which has to be compiled on the host computer. By introducing the virtual machine, the authors address both adaptability of the simulator to different target/host platforms and the ability to directly manipulate the hardware resources of the host machine for higher simulation efficiency (e.g. by mapping virtual registers to the host registers). The authors reported that the speed of compiled ISS was only 1.1–2.5 times lower than the native execution of the target code on the host computer.

In [53], T. Nakada *et al.* proposed an approach in which target instructions are statically translated into a functionally equivalent C-code. In this method, multiple target instructions belonging to one basic block are grouped into a basic block function. The basic block functions are invoked based on the value of the program counter register, which is used to select an appropriate function according to the program flow. The use of basic block functions has many benefits with respect to the final code size and simulation performance. However, such representation of the translated code does not allow simulation of *indirect* branches, i.e. the branches whose target address is calculated at run-time and cannot be determined at compile-time. In this case, the address of the target basic block function cannot be determined during the simulation. The authors solve this problem by using a second simulation mode, in which each instruction is simulated individually until the address of the known basic block function is found. At simulation run-time, the translated C-code is co-executed with the models of the processor microarchitecture to dynamically obtain the instruction execution time. In addition to the ode translation, the authors suggested optimized, workload-specific simulation of caches to further improve the simulation performance. On average, the use of static translation and compilation of the target instructions improved simulation speed by 19 times compared to the fast untimed simulator and 3.8x compared to the cycle-accurate out-of-order simulator from the SimpleScalar tool set.

Simulation methods based on static compilation require the target program code to be completely known prior to simulation. Therefore, the use of this approach may be limited if the target code changes at run-time (e.g. as in case of self-modifying codes). If the target code is not completely available prior to simulation, a more flexible approach is to translate the target instructions dynamically as discussed in the next section.

2.1.2.2 *Dynamic binary translation*

The limitations of static compiled simulation mentioned above are addressed by *just-in-time* cache compiled simulation introduced in [54]. The idea behind this approach is to pre-compile the behavioral description of the target instructions into C-functions. At simulation run-time, these functions are dynamically selected and executed on the host computer. The references to the functions are temporarily stored in a cache and subsequently reused multiple times during the simulation. If the program code has been changed, the simulator updates the cache with references to new C-functions. The authors reported that just-in-time compilation achieves 95% of the performance of conventional compiled simulation (if the size of the simulation cache is selected appropriately), while providing the full flexibility of the dynamic approach.

The Edinburgh High Speed simulator [31] supports both interpretive and DBT mode of instruction simulation. The simulation starts in the interpretive mode first. At simulation run-time, the tool profiles the program execution and dynamically translates the code of the most frequently used basic blocks. During the translation, the corresponding code sections are converted into a C-code, which is then dynamically compiled using a gcc-compiler and linked to the simulator. In the sequel of simulation, the translated basic blocks are simulated using the compiled code approximately 10 times faster than in the interpretive mode. In [31], the authors suggested that the granularity of translation units must be set either at the level of basic blocks or groups of multiple basic blocks. By using the variable size of translated code sections, an average speedup from 9.4 to 15.5 could be achieved compared to pure interpretive ISS.

Hybrid compiled instruction set simulation introduced in [65] follows a similar approach. The method employs *instruction templates* that are specific to various instruction classes of the target ISA. Based on these templates, the tool generates a custom template for each instruction of the target code. The generated code of custom templates is optimized and pre-compiled prior to simulation. The use of customized templates allows for a compact representation of the target code. At simulation run-time, a specialized decoder processes the transformed target code and then executes pre-compiled code of the respective custom templates. Thus, the authors tried to combine the benefits of statically compiled ISS and DBT. However, the approach still requires the availability of the complete target code prior to simulation. With the proposed technique, the authors achieved a speedup of 3 times compared to functional simulation in SimpleScalar.

QEMU [7] is a tool aiming at emulation of multiple target processors on different host machines. The tool enables full-system emulation of the target machine and is capable of executing unmodified target operating systems. The emulator dynamically translates target instructions into host instructions. Particularly, each target instruction is translated into a set of simpler *micro operations*, which are written in C and pre-compiled on the host platform. At simulation run-time, the micro operations are dynamically concatenated by the code generator and executed on the host machine. The translated target code is partially cached during the simulation in order to achieve better performance. The author of the tool reported a slowdown of the simulated code by 4 times (integer operations) and 10 times (floating point operations) compared to the native code execution.

The methods above focus on functional simulation only and do not consider the timing properties of software execution. Performance estimation of program execution is addressed in [11]. The paper aims at providing cycle-accurate performance modeling of an in-order processor during functional DBT-based ISS introduced in [31]. Particularly, the method employs a simplified performance model of a pipeline, which is capable of considering data dependencies among the target instructions. In this approach, the performance model is decoupled from the functional model and operates on the instruction-by-instruction and not cycle-by-cycle basis for the purpose of higher simulation efficiency. Similarly to [31], the tool translates most frequently used sections of the target code into C-code at simulation run-time. Additionally, the translated code includes functional calls to the pipeline model in order to update the microarchitecture state. Afterwards, the code is compiled as a shared library and dynamically linked to the simulator. With the proposed technique, the authors achieved a speedup of 3 times and an average simulation error of 1.4% compared to a cycle-accurate ISS.

D. Thach *et al.* [77] introduced a method for cycle count estimation by combining functional simulation in QEMU [7] with a timing analysis of the processor pipeline. In contrast to [31], the approach statically evaluates the pipeline timing prior to simulation, while making some assumptions on the status of cache accesses and branch predictions. At simulation run-time, the timing estimates are additionally adjusted to consider dynamic behavior of caches and branch prediction. The authors reported a simulation error up to 26% for an ARM processor with the average value of 10% compared to a real HW implementation. The extended version of QEMU was on average 3.37 times slower compared to functional simulation.

S. Stattelmann *et al.* [76] similarly added the capability of timing estimation to a DBT-based functional simulation in QEMU [7]. However, in this approach the execution time of basic blocks is derived using worst-case execution time (WCET) analysis of the target code prior to simulation. The timing estimates are used during run-time of simulation to obtain an overall execution time of the target program. The proposed QEMU extension resulted in a relatively small slowdown of 1.6 times compared to the normal functional execution. However, due to the usage of the WCET values for the execution time estimation, the proposed approach still resulted in large simulation errors for some benchmarks up to 33%.

There are several commercial products currently available and aiming at fast simulation of processing cores. Synopsys CoMET [97] (formerly VaST CoMET) allows fast simulation of software on a virtual platform of the target system-on-chip. The tool includes a number of cycle-accurate models of processors from various vendors. Although there is no detailed information on the simulator available neither in the literature nor on the company's website, several recent publications [70, 66, 75] report that the tool is based on DBT technique.

OVPsim [95] from Imperas Software is a simulation tool that includes a large number of functional models of diverse processors. The simulator is based on dynamic binary translation of target instructions into x86 instructions on the host machine. The processor models are instruction-accurate and can be employed for early development and validation of the target software.

2.1.3 *Sampled simulation*

The key idea behind *sampled* simulation is to perform detailed simulation only for representative parts (or samples) of the target application[1]. Thus, the overall simulation time can be effectively reduced while still providing accurate simulation results.

HySim [34] is a hybrid approach that combines the native execution of annotated target code and functional instruction-accurate ISS. During the native execution, which is also denoted as *fast-forwarding* mode, the performance is only approximated. For this, the authors propose to statically analyze the target source code and assign execution costs to each operation in the code. Switching of the simulation mode between ISS and native execution occurs at the boundaries of target functions. The designer can manually define which functions should be executed in a particular mode. Alternatively, the switching between the modes can be performed automatically, e.g. in order to promptly reach a breakpoint set by the designer. The authors also mentioned that certain functions, e.g. calls to standard C libraries or third party libraries, for which the source code is not available, cannot be executed in the fast-forwarding mode. Simulation of these functions must be carried out in the ISS mode only. Nevertheless, the proposed method still allows achieving an average speedup of 36.6 times compared to pure ISS-based simulation. In addition, the au-

1 This approach is sometimes referred to as sampling simulation.

thors reported an estimation error up to 17.6% with the average error value of 9.5%.

SMARTS framework [86] employs systematic sampling of program execution at fixed intervals. The simulation runs in two alternating modes: detailed simulation of the representative parts and fast functional simulation. During the functional simulation, the simulator updates the programmer view of the architecture only. At the same time, in detailed simulation the complete microarchitecture is updated. Furthermore, the authors showed that the state of the microarchitecture at the beginning of a detailed simulation has a significant impact on simulation accuracy. In order to reconstruct the microarchitecture state, the processor model undergoes *functional warming* of critical microarchitecture components, e.g. caches and branch predictor, before starting the detailed simulation. The approach allows achieving a very low average simulation error of 0.64% with the average speedup of 35 and 60 times depending on the considered processor microarchitecture.

Similarly to SMARTS, D. C. Powell and B. Franke [63] propose a method for accelerating performance estimation by running the processor simulation tool in two modes: slow cycle-accurate interpretive simulation and fast instruction-accurate simulation with performance prediction. However, in contrast to SMARTS, this approach employs adaptive sampling intervals. The decision on the appropriate simulation mode is made by the tool at simulation run-time. The simulation starts in the cycle-accurate mode, in which the performance prediction model is trained. If the prediction model produces accurate results relative to the cycle-accurate execution, the simulator switches to the faster instruction-accurate mode, while relying on the predicted values for performance estimation. In this mode, the tool continues evaluating the accuracy of the prediction model. If the accuracy drops below a certain threshold, the simulation is switched back to the cycle-accurate mode until the quality of prediction modeling is again improved. The authors reported an estimation error up to 19.97% with the average value of 2.36% and a speedup of 50%.

SimPoint [24] exploits the fact that target programs exhibit a repetitive behavior during an execution. This approach is based on automatic recognition of regular patterns in the program execution introduced in [72]. In SimPoint, a program's execution is divided into a set of non-overlapping intervals and the program behavior in each interval is analyzed. Intervals with a similar behavior are grouped in so called *phases*. In the next step, a representative interval is selected in each phase. The authors showed that by simulating only one representative interval per phase and extrapolating the obtained results, the overall performance of the program can be accurately estimated without the need of simulating the program completely.

2.1.4 *ISS-based simulation of multicore architectures*

Many research works employ ISS for performance evaluation of multicore architectures. For example, Gem5 [9] mentioned earlier in this chapter simulates multicore architectures by instantiating multiple ISS objects running in parallel. Simics [46, 45] can instantiate multiple instruction-level processor models to simulate a wide range of target systems, from a small embedded system to multiple complex interconnected servers. The models are simulated by translating the target instructions into an intermediate representation. The instructions in the intermediate format are then interpreted on the host machine. The processor models are instruction-accurate and do not provide accurate estimation of the cycle count. Nevertheless, the models are claimed to provide sufficient timing accuracy for simulating hardware peripherals required for full-system simulation of modern operating systems. The simulator's speed is 26 to 75 times slower compared to a native execution of the code on the host machine.

Many approaches propose to integrate multiple ISS instances into a SystemC model of a complete system-on-chip. MPARM [8] encapsulates several cycle-accurate simulators of an ARM processor into a wrapper. The wrapper is used to synchronize timing and enable data exchange between the ISS and SystemC environment, where an on-chip AMBA bus and shared memory components are simulated. The authors demonstrated an example of design space exploration of a MPSoC architecture by evaluating different cache sizes and bus arbitration policies. In a single-core architecture, the proposed approach achieved a speed of 60 Kcycles/s on a Pentium 4 2.6 GHz host computer. In a six-core architecture, the simulation speed of individual cores was reduced to 10 Kcycles/s.

Y. Yi *et al.* [88] improved co-simulation of multiple ISS instances by reducing the synchronization overhead and distributing the simulation over multiple processors in the host computer. In this method, each instruction set simulator generates a sequence of events, which are then transferred to a simulation backplane in the form of a trace. The trace is transferred at time points of inter-core communication. The backplane reconstructs the global timing of the supplied events under consideration of possible contentions on the interconnect. Thus, multiple instances of ISS can be co-simulated with a very small synchronization overhead. The method makes a simplifying assumption that the processors do not exhibit out-of-order behavior, i.e. the accesses latencies determined in the backplane do not impact on the time stamps of upcoming events. The authors showed that the proposed synchronization method achieves a simulation speed 8 times larger than the speed of a commercial SystemC simulator with conventional lock-step synchronization between ISS instances. Furthermore, by using the multicore architecture of the host computer, simulation performance could be further improved by 28%, resulting in a speed of 400 Kcycles/s. In addition, the simulation could be further distributed over multiple hosts. In this case, simulation performance could be approximately doubled on 4 host computers.

Sesame [58] is a simulation framework for design space exploration of embedded system-on-chip architectures. The framework allows for flexible mapping of application models to the underlying model of a HW architecture. The timing properties of the architecture components, among other methods, can be dynamically calibrated by an ISS co-simulated in the simulator. The cycle count estimated in the ISS is then used for high-level timing simulation of the HW architecture.

Hybrid simulation introduced in [19] aims at fast performance evaluation of multiprocessor architectures. This approach is based on the HySim simulator [34] that was also discussed in the previous section. Similarly to [34], the method relies on hybrid co-simulation of a cycle-accurate instruction set simulator and a native execution of the annotated target code. During the native execution, the timing of the target code is only approximated. For this, the tool statically analyzes the intermediate representation of the target code. Afterwards, the tool evaluates timing of the intermediate operations using fixed cycle costs and annotates the calculated time values back to the target source code. In addition, the approach employs dynamic simulation of data caches in order to improve the accuracy of timing estimation. Simulation of multiprocessor architectures is performed by instantiating multiple processor models in the simulator. Each of the processor model can run either in the ISS- or native execution mode and has its own local time which is then synchronized with the global time. In the experiments, the authors employed a very abstract bus model that does not consider contentions. The proposed method allowed simulation of the target application with an error of 3% at a speedup of 3–5 times compared to full cycle-accurate simulation.

O. Almer *et al.* [3] employ just-in-time DBT and parallelized simulation on the host computer to enable fast functional simulation of multicore architectures. This technique combines interpretive and compiled ISS and follows similar simulation principles introduced in [31]. The parts of the target code, which are most frequently executed during the interpretive simulation, are dynamically compiled and simulated on the host machine at a very high speed. The high efficiency of multicore simulation is achieved by simulating each individual core in a separate thread, which is then scheduled and executed in the host OS. Moreover, the authors show that the translated and compiled target code can be effectively shared among multiple simulations of individual cores. The synchronization of the cores is carried out with the support of the hardware synchronization instructions in the x86 host computer. The proposed technique was evaluated on a host machine with a 32-core Intel Xeon processor, achieving a speed of 11982 MIPS when simulating a 2048-core target architecture. The presented approach assumes functional simulation only and does not perform evaluation of execution time.

2.2 SIMULATION BASED ON TARGET CODE

A different approach for performance estimation is to employ the code of target applications. In this case, the target code is annotated with pre-estimated execution delays and then compiled and executed on the host computer. Host-compiled simulation achieves very high speed compared to cycle-accurate ISS. Consequently, host-compiled techniques have been widely used during simulation of MPSoC architectures [20], for which simulation performance is very critical. The target code can be employed at different levels of abstraction: source-level, intermediate-representation level or binary-level. We discuss each of these levels in more detail in the following sections.

2.2.1 *Source-level simulation*

Many recent research works employ simulation of the target code at source level, e.g. written in C language. The idea behind this approach is to annotate pre-estimated timing directly to the target source code. The methods reviewed in this section differentiate in the way of obtaining the timing information prior to simulation.

J. Bammi *et al.* [6] perform partial compilation of the source code in order to derive initial information on the target instructions. The instructions are then mapped to processor-independent virtual instructions. Performance estimation is enabled by assigning a cost to each virtual instruction. The costs are target-specific and represent approximated execution time of the instructions on the target processor. The authors suggested to determine the costs using either information provided in the processor's documentation or by employing ISS-based calibration. The timing estimates are then back-annotated in the original source code of the target application in order to enable its timing evaluation. This approach approximates the pipeline effects, e.g. stalls in the pipeline are not assumed, and neglects the timing effects of caches. The authors also suggested timing estimation based on the object code. This method will be discussed in the next section.

In [49], T. Meyerowitz *et al.* annotate the target source code with timing estimations obtained on a cycle-accurate virtual prototype of the target system. In this approach, the target software is executed in the cycle-accurate simulator first in order to obtain an execution trace. Afterwards, the tool analyzes the trace and calculates the timing delays of the instructions. The average execution time is then back-annotated to the source code using the debugging information. In the presented approach, caches, memory and communication components are not simulated and the communication delays are pre-characterized and annotated in the code. The proposed method achieves a speedup of 10–1000 times with the average estimation error of 4.9% and a maximum estimation error of 17.5% .

J. Schnerr *et al.* [70] similarly employ back-annotation of timing to the original source code of the target application. The authors suggested to employ static worst-case (WCET) and best-case execution time (BCET) analysis to obtain the execution time of basic blocks prior to compiled simulation. The proposed method uses a dynamic model of an instruction cache in order to adjust the estimated cycle count at simulation run-time. The resulting annotated code is simulated using SystemC. The method increases the simulation speed up to 91% compared to conventional ISS.

SciSim [83] is a source-level simulation technique in which the execution time of target instructions is pre-estimated using static pipeline analysis prior to simulation. The approach uses debugging information to find mapping between instructions of the object code and corresponding lines in the source code. In addition to static timing annotations, dynamic models of caches and branch predictors are co-simulated to consider dynamic timing effects. To support the simulation of data caches, additional code is added for calculating the target addresses at simulation run-time. For target code, which was compiled without optimizations, the proposed approach achieves a speedup of 16 times (assuming that the branch predictor and caches are co-simulated) at the average error of 0.1% compared to a cycle-accurate ISS of a PowerPC processor.

Y. Hwang *et al.* [28] presented a technique for performance estimation based on automatic generation of annotated transaction level models of processing elements. In this method, timing estimation is carried out using generic models of pipelined processing elements under consideration of data dependencies between operations. In addition, the authors employ statistical models of caches and branch predictor to consider dynamic timing effects. The timing estimation is performed for all basic blocks of the target application. Afterwards, the approach uses the LLVM compiler infrastructure in order to generate a C code of the target application annotated with the estimated timing values. The generated code is then simulated in the scope of SystemC processes for performance evaluation of the complete architecture.

K.-L. Lin *et al.* [43] propose a similar approach that relies on static back-annotation of the timing estimates to the source code of the target application. In this method, timing estimation is performed for target basic blocks using static pipeline analysis as well. In addition, the authors consider pipeline effects between two adjacent basic blocks and add a correcting factor to the obtained timing values. During the timing annotation, the proposed method tries to find the basic block boundaries in the source code in order to locate the annotation points as accurately as possible. The timing effects of branch prediction are considered by using a prediction model at simulation runtime. The method makes use of a dynamic instruction cache model in order to dynamically consider the impact of miss penalties on the SW execution time. In turn, the data cache is considered by using statistic modeling because target memory addresses are difficult to obtain at the source level as mentioned by the authors. The method achieves a speedup of three orders of magnitude compared to a reference cycle-accurate ISS at the average error of 2%.

All research works presented above rely on back-annotation of pre-estimated instruction timing into the source code. However, there are two problems associated with this approach. The first problem occurs if the target software is compiled with compiler optimizations. The target compiler may modify the structure of the binary program without changing its functional behavior [74]. Thus, the control flow graph (CFG) of the source code does not match the CFG of the binary code. In this case, it is particularly difficult to find mapping between instructions in the object code and the corresponding statements in the source code. Secondly, for accurate simulation of data caches, SW compiled simulation requires actual target memory addresses that are used by load/store instructions. However, this information is not visible at the source level, at which computational operations are performed on the source-code variables.

Some of the recent works address the problems above. In [74], S. Stattelmann *et al.* suggested to find matching between binary- and source-level code using dominator homomorphism. This method relates the execution order of basic blocks in the source- and binary-level code. Thus, the control flow of binary basic blocks can be reconstructed at the source level. In [75], the authors aim at accurate simulation of data caches during source-level simulations. Since the target addresses cannot be always determined correctly using the debugging information, the authors propose to use interval analysis on the target processor's registers in order to evaluate possible address ranges. Further, the authors introduce an abstract cache model that operates on the annotated address ranges and approximates the behavior of real caches. K. Lu *et al.* [44] address the matching problem by decomposing the CFG of the binary- and source-level code into multiple nested sub-graphs. The basic blocks in the sub-graphs are then matched using the domination principle.

2.2.2 *Simulation based on intermediate representation*

As mentioned in the previous section, the problems in matching binary- and source-level code of the target application hinder accurate timing annotations in the source code. One of possible solutions to this problem is to employ the *intermediate representation* (IR) of the target code for simulation. The IR code already considers many compiler optimizations and can be obtained during the compilation of the target software.

J.-Y. Lee and I.-C. Park in [39] employ a target-independent IR for performance simulation of the application code. Using the description of the target processor, cost of each intermediate operation is determined and annotated in the intermediate code. Afterwards, the intermediate code is compiled and executed on the host computer. In this approach, timing effects of instruction scheduling in the target processor are considered by scaling the annotated timing information by the pre-estimated value. Moreover, this method assumes co-simulation of cache models in order to determine cache access latencies at simulation run-time. To enable this, additional IR operations are added to obtain the target memory addresses.

T. Kempf *et al.* [33] use a similar approach to perform instrumentation of the target code represented at the intermediate, target-independent level. In this method, each operation in the IR is additionally instrumented with the code that accumulates the associated timing costs. The memory accesses are simulated by adding additional function calls to the communication model that adjust the simulation time accordingly. The instrumented code is then compiled and executed on the host computer for performance estimation of the target software. The proposed approach relies on approximated values of the execution time of the intermediate operations. Dynamic target-dependent timing effects, e.g. pipeline interlocks and effects of the caches and branch prediction, are not considered at this abstraction level.

A. Bouchhima *et al.* [12] employ IR-level of the target code for instrumentation and additionally introduce the concept of cross intermediate representation. The cross IR is the extended version of the original IR that considers target-specific transformations of the program control flow. The authors correspondingly extended the back-end of the LLVM compiler infrastructure achieving one-to-one mapping between the cross IR and the latest stage of the target-dependent IR. The authors reported an accuracy of 100% in reconstructing the sequence of basic block execution in the native execution with respect to the execution of the target binary in the instruction set simulator.

E. Cheung *et al.* [15] use a modified version of the target compiler to generate annotated *structural models* of the target software. The structural models are derived in the GCC compiler based on the IR of the target code. In addition, these models are annotated with the execution time of the instructions obtained using the documentation of the target processor. The annotated timing considers data dependencies among the instructions in the processor's pipeline. In the next step, the structural models are wrapped into SystemC components and, along with other components' models, are employed for MPSoC performance simulation. The proposed approach allowed a simulation speedup of three orders of magnitude compared to cycle-accurate ISS at an average error of 1%. However, such a speedup was achieved assuming a number of simplifications. Particularly, dynamic branch prediction and cache effects were not addressed in this approach. Moreover, the authors assume only coarse-grained communication between the processing cores in MPSoC.

In [82], Z. Wang and A. Herkersdorf proposed to generate a new *intermediate* C-code based on the IR of the target code. The resulting C code is very close to the target binary code and considers optimizations of the target compiler. The timing estimation is performed on a binary code generated from the intermediate C-code. The authors showed that in this case the source-code statements and the binary-level instructions can be matched very accurately using the debugging information of the intermediate C-code. The instructions' timing is determined prior to simulation using static pipeline analysis. In addition, cache models are employed to consider dynamic cache effects. As the target addresses are not available at the intermediate level, the authors suggested to use the host addresses of the variables in the intermediate C-code. The authors showed that simulation of the intermediate C-code produces the execution time which is very close to the execution time of the original source code. The proposed approach allowed for a simulation speed close to the native host-execution, showing the average timing error of 0.53% compared to a cycle-accurate ISS of an in-order processor.

2.2.3 *Instruction-level simulation*

At the instruction- (or binary-) level, timing simulation of the target code is performed at the granularity of instructions. The code at the binary level already contains all optimizations made by the target compiler. Therefore, the matching problem due to compiler optimizations is not relevant at this abstraction level.

J. Bammi *et al.* [6] suggested to use the modified version of the target compiler in order to produce an assembler-level C code which considers compiler optimizations. The code has a similar functional behavior as the original program and contains additional timing annotations. The timing estimates are derived using generic virtual instructions with the associated costs. The resulting assembler-level C code is then compiled and executed on the host computer in order to perform accurate performance estimation of the target software. The proposed approach allowed improvement of simulation speed by a factor of 18 times compared to a cycle-accurate ISS.

M. Lazarescu *et al.* [36] employ the assembler representation of the compiled target code to produce equivalent assembler-level C code.

The translated assembler-level C code can be co-simulated with untranslated functions of the target C code. This is enabled by using shared variables in the translated and untranslated parts and by implementing target-specific conventions of function calls. Afterwards, the resulting code is annotated with timing estimates produced by static pipeline analysis. In this approach, dynamic timing effects of instruction and data caches are abstracted. The proposed method allowed improvement of simulation speed by one order of magnitude compared to a cycle-accurate ISS.

J. Schnerr *et al.* [69] proposed an approach in which a target processor is emulated on a prototyping platform consisting of a VLIW processor and FPGAs. The execution of the target code is enabled by translating target instructions into instructions of the VLIW processor. Time estimation is then enabled by adding a pre-estimated cycle count in each basic block of the translated code. In addition, special code is inserted to simulate the timing behavior of the instruction cache and branch predictor. The annotated translated code is executed on the prototyping platform to obtain the overall execution time of the target software. The proposed approach achieved speed comparable to an FPGA emulation. At the highest level of accuracy, the authors reported a speed in the range of 3–10 MIPS, with an estimation error of 3–15% compared to the reference evaluation board.

M.-H. Wu *et al.* [85] translate target instructions into equivalent C-code annotated with timing, which is then simulated using SystemC. For simplicity reasons, the authors assume that each target instruction is executed in one cycle. Thus, the target-dependent timing, e.g. pipeline or cache effects, are not addressed. Moreover, similarly to [15], the authors simulate only coarse-grained communication requests, represented by inter-task communication using shared variables. However, the approach does not consider contentions on the shared interconnect between inter-task communication points. The simulation methodology achieves a speedup of 22–101 times compared to a cycle-accurate ISS.

In [81], Z. Wang and J. Henkel introduced a hybrid approach that combines simulation of the target code at the source and binary levels. Thus, the authors address one of the major restrictions of the source-level methods, which cannot simulate parts of the target application, for which the source code is not available, e.g. third-party libraries. In particular, this work focuses on data synchronization between the source- and binary-level code to enable functionally correct simulation. The proposed approach could double the speed of simulation compared to the pure binary-level simulation, achieving an overall speedup of 128 times compared to cycle-accurate instruction set simulator.

2.3 TRACE-DRIVEN SIMULATION

Trace-driven simulation (TDS) has been widely used for performance evaluation of computing systems [71, 79]. The key idea behind this approach is to collect a trace of events using a reference computing system first. The obtained trace is then simulated on a model of the system to predict its performance in a new environment. Alternatively, the trace can be used to assess new configurations of the system.

For multiple decades TDS has been applied for analysis of different aspects of computing systems. S. Sherman and J. C. Browne [71] presented an overview of trace-driven modeling techniques that were employed for evaluation of computer systems at the level of OS services. In these approaches, event-sensitive probes in the operating system were used to collect a trace of events. These events describe points when the system's resources were requested and allocated. Afterwards, performance estimation was carried out by simulating the resulting trace in a system's model. Trace-driven modeling techniques reviewed in [71] were used, among others, to evaluate algorithms for CPU scheduling, resource allocation or dynamic storage management.

TDS techniques have been also employed to improve performance simulation of the processor's microarchitecture. ReSim [18] is a parameterizable, trace-driven performance simulator of out-of-order processors. In this approach, the authors employ traces which capture a complete sequence of the executed target instructions. The traces are generated using an ISS and then used to stimulate the components of the processor's microarchitecture implemented in FPGA. L. Eeckhout *et al.* [17] proposed to employ reduced *synthetic* traces in order to accelerate performance evaluation of diverse processor microarchitectures. In this approach, a target program is executed first to derive a set of execution characteristics. Afterwards, the obtained values are used to generate a representative synthetic trace reflecting the collected characteristics. The trace is then used during trace-driven statistical simulation of the target processor. The synthetic traces are significantly smaller than full execution traces. Therefore, they require significantly less time to simulate.

Due to the increasing gap between the speed of processor units and memory components, TDS has gained high popularity in the performance assessment of memory systems [79]. In this type of TDS, traces contain a sequence of memory addresses generated by a processor during a program execution. In the next step, the resulting trace is used to stimulate a model of a memory system, e.g. which incorporates hierarchical caches and diverse memory structures, and to explore possible design solutions. In [79], the authors discuss advances in the three main aspects of trace-driven memory simulation:

collection of representative traces, reducing the size of large trace files and efficient simulation of the traces.

In [27, 64], TDS was applied in simulation of multiprocessor systems with shared memory. The work in [64] used TDS to evaluate caches containing shared data, trying to improve the efficiency of the existing coherency protocols. In addition to conventional traces, the authors advocate the use of synthetic traces generated by a stochastic model for the purpose of better controlling the properties of the simulated workload. In [27], the authors address the correctness of TDS in a situation when the memory architecture and memory management policy change in a new multiprocessor environment. In this situation, the addresses captured in a trace may change. The presented approach correspondingly modifies the addresses during a simulation. For this, the authors propose to identify points in the traces at which the values of addresses might change and then attempt to reconstruct the address values. The presented approach is based on the analysis of the target code represented in the intermediate three-address format.

Many recent approaches employ TDS to accelerate design space exploration of MPSoC architectures. In contrast to conventional TDS methods, these approaches employ traces that additionally contain processing delays in order to enable *timed* simulation of MPSoC models. TAPES [84] is a tool for trace-based architecture performance evaluation in SystemC. In this approach, the functionality of each MPSoC component is specified at a very high abstraction level and represented in the form of abstract execution traces. Each execution trace defines a sequence of communication requests and processing latencies between them, thereby determining the system-level behavior of the respective component. Performance estimation of the MPSoC is performed by superposed co-simulation of the traces on the model of the shared interconnect. Processing latencies are fixed and derived prior to simulation. In turn, communication latencies are determined dynamically at simulation run-time, depending on the contentions on the shared HW resources.

Sesame [57] employs TDS to enable fast design space exploration of multimedia applications in embedded systems-on-chip. In this framework, a target application modeled using Kahn Processing Networks (KPN) generates a trace of events, which is then used to simulate the hardware models. Thus, in contrast to conventional TDS methods, traces in Sesame are generated at simulation run-time. The traces abstract the behavior of the applications on processing components. In fact, the traces specify coarse-grained interaction between KPN processes using very basic commands, e.g. read, write and execute. Furthermore, the simulation framework includes a mapping layer which is used to assign and schedule the execution of traces on the underlying hardware components. The aim of the mapping layer is to enable flexible evaluation of SW partitioning. Moreover, this layer is employed to refine the traces events into fine-granular events during the architecture refinement process.

S. Mahadevan *et al.* [47] employ TDS to model interconnect traffic generated by processing elements in a MPSoC. For this purpose, the authors present an abstracted generic IP model which is capable of executing a small set of simple instructions, e.g. reading/writing data or waiting for a specific amount of cycles. The generic IP model reconstructs the timing behavior of a programmable processing core. The authors demonstrated how to derive a program for the IP model using an execution trace captured on a reference cycle-accurate ISS. The set of IP models instructions includes control-flow instructions for performing conditional and unconditional branches to a certain label. Thus, using the available instructions, the designer can write custom programs for the IP model in order to create traffic of not yet existing peripherals.

T. Isshiki *et al.* [29] aim at overcoming the performance limitations of cycle-accurate ISS by making use of execution traces. In this method, the traces are represented in the from of branch streams. A branch stream is a sequence of the branch operation results which was captured during the execution of a target program. To generate such a trace, the target source code is instrumented and natively executed on the host computer. The approach makes an assumption that the control flow graphs of the source and binary code are identical. The produced trace is then used to reconstruct the execution flow in an optimized control flow graph of the target program. To enable performance evaluation, this graph is annotated with cycle counts of the respective parts of the target code. In turn, the cycle counts are obtained by means of the target compiler and by employing static timing analysis of instruction execution. In this approach, only static execution time of instructions is considered. Dynamic timing effects, e.g. originating from instruction and data caches and memory access latencies, were abstracted.

H. Lee *et al.* [38] employ TDS for fast performance simulation of multicore architectures. The authors propose a simulation approach consisting of two separate phases. In the first phase, an abstract execution trace of the target application is generated by means of a cycle-accurate instruction set simulator. The generated trace is then filtered leaving only L1-cache miss events and timing intervals between them. In the second phase, the trace is simulated in a new multicore environment, in which a mesh network on-chip and shared L2-caches are modeled. The authors mentioned that filtering out the L1-cache introduces a problem in case of coherent caches. For example, an access to the L1 cache data, which resulted in a hit during the trace generation, may result in a un-expected miss in a new multicore environment, because the data may have been invalidated by another core. In this case, the trace will not reflect the correct behavior of the processing core anymore. The authors mentioned that a possible solution to this problem could be not to filter accesses to the L1 cache for the data that may be potentially shared between multiple cores. However, this approach would require reconstruction of the complete L1-cache state at a simulation run-time which is not a trivial task. This issue was left by the authors as a future work.

In [40], K. Lee *et al.* employ trace-driven simulation to accurately reconstruct the timing behavior of out-of-order processors during explorations of diverse memory systems. Similarly to the previous approach, the authors employ filtered traces specifying the events of L1 misses. The obtained traces are simulated to evaluate the execution time of SW in an out-of-order processor in a different environment. In this paper, the authors introduced a model that considers *pairwise dependent* cache misses to predict the timing behavior of the processor during L1 cache misses. Particularly, in this approach, the simulation of an L1-cache miss is postponed if the miss depends on a previous L1-cache miss and, at the same time, the data transfer caused by the previous miss has not yet been completed. Furthermore, the authors introduced a method for reorder buffer (ROB) occupancy analysis to prevent the simulation of a L1-cache miss if the corresponding memory instruction can not be placed into the ROB. With the proposed methodology, the authors could significantly improve the accuracy of TDS in case of out-of-order processors.

2.4 HIGH-LEVEL OS MODELING

In complex embedded systems integrating the functionality of multiple applications, the use of a real-time operating system (RTOS) is vital to manage the execution of application tasks on the underlying processing elements. Many recent research work address high-level modeling of RTOS in order to consider the behavioral and temporal effects of a RTOS at early design stages and, thus, to improve the accuracy of system-level performance evaluations. The largest part of this section reviews simulative approaches. However, I also present a few examples employing formal timing analysis in the end of this section.

The simulative methods presented here differentiate, among other criteria, in the abstraction level and simulation method of task models. The overview starts with OS modeling approaches, in which the timing behavior of tasks is represented at a very high-level of abstraction using coarse-grained execution delays. This abstract representation is typical for early stages of system-level design.

ABSTRACT TASK MODELING S. Yoo *et al.* in [89] aim at fast building of OS simulation models and exploration of different OS candidates at system-level. The process starts from a behavioral description of the target SoC, which is then refined to the implementation level by means of so called HW and SW wrappers. The method presumes that the wrappers are available from the libraries in the form of synthesizable code and simulation models. The wrappers are employed to generate an application-specific timed OS model using the same principles as used for the generation of the final OS code. Thus, the authors address the equivalence between the generated OS model and the final OS code used in the SoC. In this approach, the execution time of tasks is considered by annotating pre-calculated execution delays into the OS model. The resulting OS model is then simulated using SystemC.

A. Gerstlauer *et al.* in [21] aimed at the lacking capability of system-level modeling languages in capturing the behavior of real-time operating systems (RTOS) during system-level design. In the first design stage, the behavior of the target system is represented as a set of communicating processes and abstract processing elements. The system model is further refined to consider the serialized execution of application processes on precessing elements according to the assigned priorities and scheduling algorithm. To enable this, the authors present a high-level RTOS model for dynamic scheduling of the application processes independently of a particular RTOS implementation. The timing simulation of application tasks is enabled by inserting coarse-grained pre-estimated execution delays.

Similarly to the previous paper, R. Le Moigne *et al.* [51] address serialization of task execution on processing elements and introduce a high-level RTOS model implemented in SystemC. This approach additionally considers the RTOS timing properties. Particularly, the designer can specify the duration of the scheduling process and task context switches. The authors presented two implementations of the RTOS model. In the first implementation, the model is simulated as a separate SystemC thread. The second implementation improves the simulation efficiency by moving the RTOS methods directly to the tasks objects, thus, reducing the overall number of cost-intensive SystemC context switches. The proposed approach has been employed in CoFluent Studio, a commercial product which is now the part of the Intel's portfolio [94].

In [1], the authors address co-simulation of embedded software and an application-specific RTOS model in SystemC. The proposed RTOS model offers a set of APIs that can be used in the application's code to access the RTOS services. The application's code is simulated in the form of tasks using separate SystemC threads. The execution time of the tasks and RTOS services are considered by annotating execution latencies in the model. The latencies are obtained by calculating the execution time of instructions in the application/RTOS code using the datasheet of the target processor. In addition to RTOS calls, the tasks can perform memory and IO operations by invoking the bus functional models of the processor. The timing of the memory and IO operations is assumed to be fixed and can be configured by the designer.

Z. He *et al.* [25] presented a generic RTOS model that can be refined to simulate the behavior of existing RTOS implementations. The model is represented by a configurable state machine and implemented using SystemC. The timed simulation is enabled by annotating pre-estimated execution delays in the model. To obtain the timing information for the application models, the authors suggested to perform instruction-level timing analysis of the target code. In turn, the execution timing of the OS services is obtained from the benchmark data provided by the OS vendor.

In [32], the authors aim at flexible performance evaluation of task mappings with a particular focus on application-specific SoC incorporating multiple processing elements with hardware multi-threading. The tasks are mapped and scheduled on the available processing resources by means of an intermediate mapping layer. In this approach, tasks are modeled by means of finite state machines which are annotated with coarse-grained timing information. In addition, the designer can specify the time required to swap the tasks in a processing element. In the provided example, the authors used the timing values published in the vendor's documentation.

ARTS [48] is a SystemC based simulation framework for evaluation of various task mappings on a MPSoC platform taking the RTOS effects into consideration. The approach has been applied for investigation of streaming multimedia applications. In this framework, target applications are modeled as a set of task graphs. In these graphs, each task is characterized by a set of timing properties which has to be set by the designer prior to simulation, e.g. start time, period, worst- and best-case execution time as well as pre-estimated execution costs for various types of processing elements. In addition, the designer needs to assign communication costs between the tasks, which are required to determine data transfer time in the communication model. In the simulation phase, the tasks are mapped onto architectural components and simulated in order to produce the overall program completion time on a given architecture.

ISS-BASED SIMULATION In contrast to previous approaches that use abstracted task models, in this part we discuss research works that employ cycle-accurate ISS in order to simulate the task workload. In [87], Y. Yi *et al.* address efficient co-simulation of SW and HW simulators, while considering the effects of a preemptive RTOS. In this approach, multiple application tasks are executed on an ISS, while the abstract RTOS model is implemented in the simulation backplane at an abstraction level higher than ISS. In this approach, the ISS executes application's code without any notion of an RTOS and provides the execution time stamps to the backplane. In the backplane, the RTOS model reconstructs the actual simulation time considering the effects of the RTOS-supervised execution. In addition, the model allows consideration of the RTOS timing overhead by accounting delays of context switches and interrupt handling. The presented approach was also applied during simulation of multiprocessor architectures in [88].

M. Krause *et al.* [35] suggested to employ a cycle-accurate simulator to execute the application's code. The simulator is wrapped into a SystemC environment and co-simulated with an abstracted RTOS model. In this approach, scheduling decisions are made in the RTOS model which does not require cycle-accurate simulation. To consider the timing overhead of the RTOS, the model is annotated with a measured execution latencies of the corresponding RTOS services, e.g. latencies for creating tasks and starting their execution. The authors showed that implementing the RTOS functionality at the higher abstraction level allows achieving higher simulation speed compared to pure ISS, particularly in a use case when task switching occurs frequently.

J. Chevalier *et al.* [16] address architectural exploration of a SoC with a focus on evaluation of HW/SW implementation alternatives for the target application. Similarly to [21], the method follows a top-down approach of the system-level design, starting at a high-level representation of the application in the form of untimed, communicating SystemC modules. In the next step, SystemC modules are classified as either HW or SW modules to represent the HW and SW parts of the system respectively. The presented method enables accurate simulation of the SW modules by using an ISS of the target processor. Furthermore, the authors employ the code of an existing commercial RTOS to schedule the execution of SW modules in the ISS. To enables this, an interface adapter was created which maps SystemC function calls made by a SW module to the equivalent RTOS functions. Afterwards, the SW module is cross-compiled for the target ISA and linked with the RTOS code. The resulting binary code is executed in the ISS which, in turn, is co-simulated with other HW modules in the common SystemC environment. The advantage of this approach is a possibility to represent the application's either as a HW or a SW module very easily.

EMPLOYMENT OF ANNOTATED TARGET CODE This part reviews approaches that employ annotated target code for simulation of applications tasks. In [82], the workload of user tasks is simulated using intermediate-level source code annotated with pre-estimated timing information. To manage the simulation of tasks on a multicore architecture, the authors present a SystemC-based simulation framework incorporating a very abstract RTOS model.

M. Müller *et al.* [52] aimed at automated generation of system-level models of the target platform for early design space explorations. The presented approach starts with an abstract functional model of the application represented in the form of Kahn Process Networks (KPN). The models are then refined and employed in system-level modeling of the target platform which considers the timing effects of an RTOS. The method proposes two types of RTOS models. Firstly, the authors employ a generic RTOS model for coarse evaluation of thread scheduling. Secondly, the authors demonstrated refinement of the generic RTOS model to a specific RTOS implementation in order to provide higher accuracy of performance estimation. For this, an interface was created that couples the refined generic RTOS model with the specific RTOS implementation. In order to consider the timing properties of software execution in the system-level model, the modified front-end of the LLVM compiler is used to statically determine execution time of the applications and RTOS at the level of basic blocks. Dynamic timing effects are considered by simulating instruction and data caches. The resulting annotated code is compiled and executed on the host computer to estimate performance of the complete system.

FORMAL ANALYSIS A different approach for evaluating performance of task mappings and scheduling strategies is to perform formal performance analysis of the target platform. In [80], the authors propose a method for evaluating non-preemptive scheduling and detecting possible deadlocks in an MPSoC with shared resources. The approach is based on abstraction of a functional implementation of SW processes and their formal representation in the form of communication dependency graphs with annotated best- and worst-case execution times. Afterwards, analysis is performed to evaluate different task mappings considering the impact of cooperative scheduling. In [80], the approach was applied to a JPEG application executed on a platform with different configurations in order to obtain the utilization of computational resources.

The formal analysis can be also employed for performance validation of complex distributed embedded systems. In such systems, the challenge is to consider a multicore architecture of individual nodes in the scope of holistic performance analysis at the system level [68]. In [68], the authors employ a concept of event model propagation to determine the maximum response time of tasks on a multicore platform under assumption of dynamic task scheduling and the usage of shared resources. The analysis starts with an estimation of the maximum number of accesses performed by tasks to a shared resource. In the next step, the access latency to the shared resource is determined considering possible interference among the tasks. Finally, scheduling analysis is performed considering the estimated access latencies to the shared resources.

Formal analysis is a suitable method for investigation of corner cases, which are difficult to discover with simulative approaches. Particularly, the formal analysis allows the designer to identify worst-/best-case execution times of the application on a MPSoC platform as well as best-/worst-case response times of tasks in a distributed real-time embedded system.

2.5 SUMMARY

As can be concluded from the previous sections, different techniques have been proposed allowing performance estimation of SW execution at early design stages. Cycle-accurate interpretive instruction set simulators provide very high estimation accuracy because they contain performance models of the microarchitectural components, e.g. instruction pipeline, instruction queues, caches or branch predictor. In these tools, the simulation of a target program is organized in a loop, in which the instructions are fetched, decoded and executed at simulation run-time. For out-of-order processors, these simulators can accurately capture the effects of out-of-order execution since they model dynamic scheduling of the target instructions. However, due to the high level of details, these simulators suffer from low speed and their use in system-level design space exploration of multiprocessor architectures is limited. At the level of SoC components, the designers are primarily interested in *inter*-core events, e.g. communication over the shared interconnect or the usage of common hardware resources, rather than in *intra*-core events.

Recent research works aim at improving the speed of interpretive ISS by moving cost-intensive instruction decoding to compile-time. For example, the workload specific simulator introduced by T. Nakada *et al.* [53] decodes target instructions at compile-time and improves the performance of an interpretive cycle-accurate ISS by an average factor of 3.8 as reported in [53]. However, this method still achieves relatively low simulation speed because it only abstracts the decoding part of the processor's microarchitecture. In fact, H. Lee *et al.* in their publication [38] conducted performance analysis of an interpretive cycle-accurate ISS and reported that the most simulation efforts are spent rather in modeling the pipeline's behavior and instruction scheduling. Some of the recent techniques employ dynamic binary translation (DBT) of the target instructions. However, these methods primarily focus on *functional* simulation of the target code. One of the latest approaches presented in [11] does address performance simulation by combining interpretive cycle-accurate ISS with cycle-approximate just-in-time DBT. However, this method still requires co-simulation of a pipeline's performance model and assumes in-order target processors only.

Host-compiled techniques, which have been introduced recently, are based on a native execution of the target code and employ pre-estimated timing annotations instead of detailed modeling of the microarchitecture. Consequently, these methods achieve higher simulation speed than ISS and hence they are very attractive for system-level simulation of multiprocessor architectures. Most of them attempt to completely substitute cycle-accurate ISS and *predict* the SW execution time of target instructions, e.g. by employing static pipeline analysis prior to simulation. However, to the best of my knowledge, none of the existing techniques considers the capability of modern processors to perform out-of-order execution of instructions.

Host-compiled simulation introduces many challenges in reconstructing the timing behavior of an out-of-order processor. Particularly, the target code used for host-compiled simulation has a static structure and cannot reflect context-dependent deviations of the instructions' execution time as well as the order of memory operations (see Section 4.1.2 for further details). Moreover, non-blocking behavior of data caches is not considered in conventional host-compiled approaches. Currently, there is a demand for host-compiled simulation methodology that could capture the timing behavior of out-of-order processors, while having a reasonable level of accuracy and enabling simulation speed higher than interpretive cycle-accurate ISS.

This thesis addresses this gap in simulation techniques by introducing a novel approach for host-compiled SW performance simulation, which considers complex timing behavior of out-of-order processors. The proposed technique is based on simulation of the target code at the binary level and employs the principles of the target code translation originally proposed by T. Nakada *et al.* in [53]. However, in contrast to [53], in the presented approach the processor's microarchitecture is abstracted at a much greater extent, allowing for a larger speedup with respect to an interpretive cycle-accurate ISS. Essential background information on the code translation will be provided in Chapter 3. To capture dynamic effects of out-of-order execution, the proposed approach follows the idea which was initially proposed by K. Lee *et al.* for trace-based simulations in [40]. I employ a similar method, which is adapted for binary-level host-compiled simulation and considers data dependencies between memory access instructions. The corresponding details will be given in Section 4.5.

Trace-driven simulation (TDS) is an alternative method for system-level performance evaluation. Compared to execution-driven simulation methods, TDS is performed at a very high abstraction level. It *completely* abstracts the internal processor's microarchitecture and does not require functional execution of the target code or availability of the target code in general. Thus, in case of TDS, more computational efforts in the host computer can be spent in simulating the system-level aspects, e.g. interaction of MPSoC components at the system-level. However, this advantage comes at the expense of simulation flexibility. Particularly, the control flow of the target application captured in a trace is fixed. Thus, a new trace must be generated if the control flow has been changed, e.g. if different input data has been applied to the target application. Moreover, because of the abstracted functionality, no data-dependent relations between tasks in a complex application can be reconstructed at simulation run-time using TDS. Nevertheless, if the control-flow of the application is kept fixed, e.g. if the designer is interested in evaluation of *exactly* the same workload on different hardware architecture, the TDS method can alleviate the simulation complexity and accelerate design space exploration.

Abstract RTOS modeling is essential to capture the realistic behavior of modern multi-tasking applications at the system level. Many approaches abstract the *complete* execution of tasks to processing delays and do not consider low-level timing effects in HW, e.g. dynamic behavior of caches or contentions on the shared on-chip interconnect. In general, this level of abstraction is suitable at very early design stages, when the details of the target HW architecture are not known yet. If the target processor architecture is known, many approaches suggest to co-simulate an ISS instance with a high-level RTOS model. As mentioned earlier, the use of ISS during design space exploration is limited, particularly if multiple simulations have to be performed. Some of the recent approaches simulate tasks at a higher abstraction level by employing annotated target code of the application tasks. However, they make a number of simplifying assumptions on the processor's microarchitecture and assume in-order processors only.

3 BACKGROUND OF COMPILED SW SIMULATION AT BINARY LEVEL

Host-compiled simulation allows the designer to efficiently reproduce the functional and temporal behavior of the target[1] code at the system level by abstracting unnecessary microarchitectural details of the target processing core. For the simulation purposes, the target code can be employed at different representation levels, e.g. source level, intermediate representation level or binary level as discussed in Chapter 2. This section provides details on host-compiled simulation at the binary level.

The workflow of binary-level simulation is presented in Fig. 3.1. In the first step, the binary code of the target application is translated into a functionally equivalent C code. The translation is necessary because of the following reasons. Firstly, the target and host processors may have different instruction set architectures (ISAs). In this case, the target code cannot be natively executed on the host computer. Secondly, afterwards the resulting C code can be easily extended for other simulation purposes as will be discussed later in this chapter.

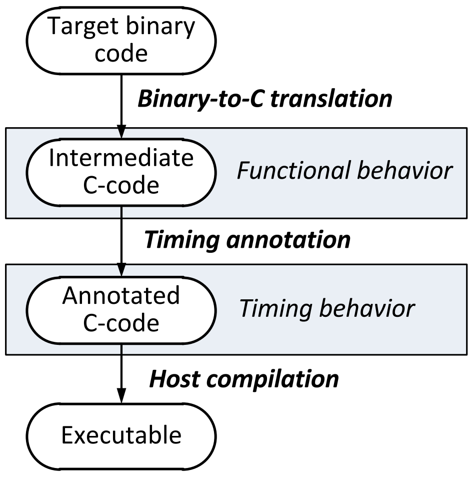


Figure 3.1: Workflow of binary-level SW compiled simulation

The resulting C code preserves the semantics of the original binary code. Therefore, during the execution, the C code accurately reproduces the *functional* behavior of the target application. In order to reconstruct the *timing* behavior of the application under considera-

1 For the following explanations, the term *target* refers either to a simulated processing core or to the software code running on that core. In turn, the computer which performs the simulation is referred to as *host*.

tion of the performance characteristics of the target processor, the C code must be additionally annotated with timing information. In the last step, the annotated code is compiled on the host computer, producing in an application-specific *executable* performance model of the target processor. The resulting model can be further employed for simulations of multiprocessor architectures at the system level.

The following sections describe the reconstruction of the functional and temporal behavior of the target code in greater details.

3.1 FUNCTIONAL BEHAVIOR

3.1.1 *Binary-to-C translation*

The aim of binary-to-C translation is to represent instructions of the target binary code in the form of functionally equivalent C expressions. In contrast to instructions, in which arithmetic operations are performed on registers of the target processor, the C expressions perform operations on a set of register variables. Fig. 3.2 illustrates an example of binary-to-C translation of the binary code (shown in the left part) into an equivalent C-code (shown in the right part).

```
lui   r4, 16
add   r5, r4, r3
lw    r10, 0(r16)
sub   r4, r10, r5
sw    r4, 4(r16)
```

```
r[4] = 16;
r[5] = r[4] + r[3];
r[10] = read(r[16]);
r[4] = r[10] - r[5];
write(r[4], r[16] + 4);
```

Figure 3.2: Translation of the target binary code into a functionally equivalent C code.

During the translation, arithmetic and logic instructions, e.g. `lui`, `add` or `sub`, are transformed to the respective arithmetic C operations on array of register variables `r`. At the same time, translation and execution of load/store operations, e.g. `lw` and `sw`, requires additional modeling of the target memory, which will be discussed in detail in Section 3.1.3. The memory model is accessed by means of `read` and `write` functions. The purpose of these functions is to copy the values of register variables to/from the target memory. The address of memory locations is specified in another register variable and may require an additional offset. In this case, the address is computed directly in the translated code as shown in `sw` instruction in Fig. 3.2.

A target application may perform system calls to request services of the operating system, e.g. to read data stored on the hard disk. Therefore, the translated code must be capable of handling system call requests. For these purposes, system calls can be *emulated*, i.e. each system call request made in the target code is converted into an equivalent system call in the host computer. Afterwards, the result of the system call is written back to the target memory if necessary. The emulation of system calls is often supported in instruction set simulators [5, 9].

3.1.2 *Organization of translated code*

The execution flow of target instructions can be changed on jump instructions. These instructions can modify the value of the program counter register and, thus, break the sequential order of instruction execution. Consequently, in addition to arithmetic operations on the register variables and accessing the target memory, the translated code must be capable of dynamically changing the flow of execution. It can be achieved by organizing the translated code in *basic block functions* as proposed in [53].

A basic block function contains the translated code of instructions belonging to one basic block[2]. In the example shown in Listing 1, the translated code is structured into two basic block functions. These functions incorporate the code of basic blocks starting at addresses 0x100 and 0x200. The functions are invoked using an array of function pointers (see line 2 in Listing 1), which is indexed by the addresses of the first instructions of basic blocks. Please note that the start addresses cannot be employed directly for indexing, as in this case the array would have a very large size and contain 2^n elements, where n is the instruction width in the target processor. Instead, the array's index can be represented in a more compact form using macro function `INDEX()` shown in line 1. This function calculates the offset between the start addresses of the current basic block and the start address of the first basic block[3] in the target program. Afterwards, the offset is divided by the instruction width expressed in bytes. The result of this manipulation is the *sequence number* of a given instruction in the target code (assuming that the instruction width is constant in the target processor). In this case, the size of the array will equal the total number of instructions in the target code. The array of function pointers needs to be initialized prior to the code's execution as shown in lines 3–5.

2 A section of the binary code that has only one input point and one exit point. Basic blocks may consist of one or several instructions.

3 The first basic block in the program order having the smallest start address.

Listing 1: Organization of the translated code in basic block functions.

```
 1 #define INDEX(x) ((x - FIRST_INST_ADDR) / INST_WIDTH)
 2 void (*table[])() = {    /* array with function pointers */
 3     [INDEX(0x100)] = b_0x100,
 4     [INDEX(0x200)] = b_0x200,
 5     [INDEX(0x300)] = b_0x300,
 6 };
 7 unsigned int r[32];      /* processor registers */
 8 unsigned int pc = 0x100; /* initialize program counter */
 9 unsigned int *mem_data;  /* virtual data memory */
10 unsigned int *mem_stack; /* virtual stack memory */
11 
12 void b_0x100() {         /* basic block function 0x100 */
13     r[2] = read(r[28] + 32);
14     r[2] = r[2] + 1;
15     r[3] = read(r[28] + 48);
16     if (r[2] != r[3])    /* conditional jump */
17         pc = 0x200;
18     else
19         pc = 0x300;
20 }
21 void b_0x200() {         /* basic block function 0x200 */
22     r[2] = r[0] + 1;
23     write(r[2], r[28] + 16);
24     pc = 0x300;          /* unconditional jump */
25 }
26 int main() {
27     while (1) {            /* main execution loop */
28        table[INDEX(pc)](); /* call to basic block function */
29     }
30 }
```

The execution of basic block functions is organized in a loop (see line 27). The invocation of basic block functions is carried out using the value of program counter variable pc, which specifies the next basic block function to be executed. The value of pc is modified inside the basic block functions (line 24) and is used to select an appropriate function pointer in the array (28). In case of a conditional jump, e.g. as shown in line 16, the value assigned to pc depends on other register variables.

3.1.3 *Modeling of target memory*

For the correct operation of memory instructions, the translated code must provide a model of the target memory. Ideally, the memory instructions should be able to access the full range of target addresses. However, it results in large consumption of resources in the host computer. In fact, the target program consume only a limited amount of memory in predefined memory ranges. Therefore, the target memory can be efficiently modeled using two arrays which represent the data segment (also including the target heap memory) and stack segment as shown in Listing 2. The size of the arrays (`DATA_SIZE` and `STACK_SIZE`) depends on the range of memory addresses which are accessed by the target application. This information can be obtained by analyzing the target code prior to simulation. Alternatively, the memory consumption can be evaluated during a preliminary execution of the target code in an instruction set simulator.

Listing 2: Initialization of data and stack memory

```
unsigned int *data_mem;
unsigned int *stack_mem;

void init_data() {
   data_mem = new unsigned int[DATA_SIZE]();
   // initialize array with data words
   data_mem[0]=0x47415355; data_mem[1]=0x62203a45;
   data_mem[2]=0x6a32706d; data_mem[3]=0x73206770;
   ...
}
void init_stack(const char *fname) {
    stack_mem = new unsigned int[STACK_SIZE]();
    FILE *f = fopen(fname, "rb");
    // initialize array using file data
    fread((void *)((char *) stack_mem + INIT_OFFSET), 1,
           INIT_SIZE, f));
    fclose(f);
}
```

For the correct operation of the translated code, some entries in the data and stack memory have to be pre-initialized. For example, the data segment must be initialized with the values of global and static variables. These values can be retrieved from the target binary code using a disassemble tool, e.g. *objdump* provided in the GNU binary utilities. In the depicted example, the initialization of the data segment is preformed in function `init_data()`. In turn, the stack segment must be pre-initialized with run-time information such as command line arguments and environment variables. This information can be obtained from the simulation environment on the host computer. Alternatively, the stack memory can be initialized with the predefined values stored in a file. The latter approach is implemented in Listing 2 in function `init_stack()`.

Listing 3: Implementation of functions for accessing the target memory model

```
#define IN_RANGE(addr,min,max) (addr >= min && addr <= max)

int read(unsigned int addr)
{
   if (IN_RANGE(addr, MIN_DATA, MAX_DATA))
       return *(int*)((char*) &data_mem[0] + (addr - MIN_DATA);
   else if (IN_RANGE(addr, MIN_STACK, MAX_STACK))
       return *(int*)((char*) &stack_mem[0] + (addr - MIN_STACK);
}
void write(int src, unsigned int addr)
{
   if (IN_RANGE(addr, MIN_DATA, MAX_DATA))
      *(int*)((char*) data_mem + (addr - MIN_DATA)) = src;
   else if (IN_RANGE(addr, MIN_STACK, MAX_STACK))
      *(int*)((char*) stack_mem + (addr - MIN_STACK)) = src;
}
```

As mentioned previously, the modeled target memory is accessed using `read()` and `write()`. The implementation of these functions is shown in Listing 3. Firstly, these functions check the range of the given memory address by comparing its value with predefined boundaries `MIN_DATA` and `MAX_DATA`. Afterwards, depending on whether the address corresponds to the data or stack segment, the functions determine the offset in the respective array and copy the content of the memory.

3.2 TIMING BEHAVIOR

In host-compiled simulation, the reconstruction of timing behavior of the binary code consists of static and dynamic parts. In the static part, execution delays of the target instructions are pre-estimated at compile time and annotated in the translated code prior to simulation. In the dynamic part, the delays are additionally adjusted at simulation run-time in order to consider those timing effects that cannot be determined at compile time, e.g. behavior of instruction and data caches.

3.2.1 *Annotation of timing information*

One of the central idea of host-compiled simulation is to move the estimation of instruction timing from simulation time to compile time and, thus, to improve simulation efficiency. An example of translated code which has been annotated with timing information is presented in Fig. 3.3.

```
0x4014a0: lw r2,60(r16)
0x4014a8: lw r3,64(r16)
0x4014b0: addu r19,r19,r4
0x4014b8: addu r2,r4,r2
0x4014c0: addu r3,r4,r3
0x4014c8: sw r2,60(r16)
```

```
r[2] = mem_access(READ, r[16] + 60);
r[3] = mem_access(READ, r[16] + 64);

cycle += 4;

r[19] = r[19] + r[4];
r[2]  = r[4] + r[2];
r[3]  = r[4] + r[3];

cycle += 5;

mem_access(WRITE, r[2], r[16] + 60);
```

Figure 3.3: Target binary code and translated code annotated with execution delays.

The annotated values represent execution latencies of the respective instructions in the target processor. Please note that in contrast to arithmetic instructions, the execution time of memory instructions is highly dynamic and cannot be precisely determined prior to simulation. The consideration of dynamic memory access latencies will be discussed later in this chapter. During the execution of the translated code, the values are added to variable `cycle`, thereby performing evaluation of the program's execution time. Thus, at the end of the execution, this variable contains the total amount of clock cycles elapsed from the beginning of the simulation.

The execution time of instructions can be determined in different ways, e.g. by using data sheets provided by the manufacturer and statically analyzing the timing effects in the pipeline [83]. Alternatively, the execution time can be obtained using WCET analysis as in [76]. In this work, the timing of instructions is derived by executing the target code on a reference cycle-accurate simulator prior to host-compiled simulation. During the measurement, the completeness of collected timing information depends on the choice of input stimuli applied to the cycle-accurate simulator. Therefore, we assume that the target code is measured using a typical input set that covers most relevant code parts. Nevertheless, if host-compiled simulation reveals unmeasured sections of the target code, the uncovered path has to be measured again and the translated code must be updated with the missing timing values.

3.2.2 *Modeling of caches*

In the dynamic part, the execution time is adjusted at simulation runtime in order to reflect the dynamic timing behavior of processor components. For example, if the target processor employs data and instruction caches, the latency of a memory operations depends on the current state of the cache.

Although the state of caches can be captured during the measurement in the reference cycle-accurate simulator, this information cannot be annotated in the translated code because of the following reasons. First, a memory instruction may access different data and cause both cache hits and cache misses during the program execution. As a result, neither a permanently annotated hit nor a miss would reflect the access status correctly. Second, if a memory instruction always causes a cache hit during the measurement, it may not be the case in a new environment. Particularly, this situation may occur in a multiprocessor environment, in which other processors may potentially invalidate the respective cache line. In this case, the annotated hit will not be valid anymore. Therefore, it is essential to measure the instruction timing in the reference cycle-accurate simulator assuming *perfect*, i.e. always-hit, instruction and data caches. In turn, the information about misses must be determined dynamically as it is critical for deriving correct values of memory access latencies.

```
0x4014a0: lw r2,60(r16)
0x4014a8: lw r3,64(r16)
0x4014b0: addu r19,r19,r4
0x4014b8: addu r2,r4,r2
0x4014c0: addu r3,r4,r3
0x4014c8: sw r2,60(r16)
```

```
cycle += icache(0x4014a0);

r[2] = mem_access(READ, r[16] + 60);
cycle += dcache(READ, r[16] + (60));

r[3] = mem_access(READ, r[16] + 64);
cycle += dcache(READ, r[16] + 64);

cycle += 4;

r[19] = r[19] + r[4];
r[2]  = r[4] + r[2];

cycle += icache(0x4014c0);

r[3]  = r[4] + r[3];

cycle += 5;

mem_access(WRITE, r[2], r[16] + 60);
cycle += dcache(WRITE, r[16] + 60);
```

Figure 3.4: A piece of target binary code (left side) and the respective translated code (right side) accessing dynamic models of instruction and data cache using `icache()` and `dcache()` functions.

In order to determine whether a memory access results in a hit or a miss, host-compiled simulation requires performance models of caches. These models allows to determine whether the requested data is present in the cache during the simulation. In the translated code, the cache models are accessed using functions `icache()` and `dcache()` as shown in Fig. 3.4. If an access to the instruction or data cache results in a miss, the value of cache miss penalty is additionally added to `cycle` variable. In turn, if an access results in a hit, no additional value is added as the annotated timing has been derived assuming a cache hit.

In reality, the target processor access the instruction cache for every instruction. However, in host-compiled simulation, the effective number of instruction cache accesses can be reduced, thus, improving the overall simulation performance. The optimization is based on the fact that the instructions of a basic block are always stored sequentially in the memory. If an instruction accesses results in a miss, fetching of consecutive instructions fitting into the same cache line will always result in a hit. Therefore, if the size of the cache lines is known in advance, unnecessary accesses to the instruction cache can be removed. In the example shown in Fig. 3.4, the instruction cache is accessed only for `lw` instruction at address `0x4014a0` and `addu` instruction at address `0x4014c0`. Other instructions fit in these cache lines[4] and always generate a cache hit. Consequently, cache accesses for these instructions can be omitted. In contrast, the model of the data cache has to be invoked for each access to data memory. The target address, which is necessary for the cache model, is determined using the values of register variables as shown in the example in Fig. 3.4.

Please note that the cache models are used for evaluation of memory access latencies but not for actual caching of data. The real data are not handled by the models for the sake of better simulation performance. In fact, the cache models only contain address tags which are sufficient to determine a hit or a miss for the current memory access. Given the memory address, the cache model performs a lookup in its internal database with address tags. Further details on the implementation of cache models will be given in Section 5.3.

4 In this example, the cache line size is assumed to be 32 bytes, i.e. one cache line can hold four 64-bit instructions.

4 COMPILED SW SIMULATION CONSIDERING OUT-OF-ORDER INSTRUCTION EXECUTION

This chapter presents details of the proposed method of performing host-compiled binary-level SW simulation considering out-of-order execution of instructions in the target processor. Firstly, we will discuss the effects of out-of-order execution at the system level that need to be addressed in host-compiled simulation. Afterwards, I will present a technique for deriving the execution time of basic blocks using a reference simulator of an out-of-order processor. Furthermore, I will demonstrate how to annotate this information in the translated target code and how to consider the out-of-order effects at simulation run-time in order to reproduce the processor timing behavior more accurately. Finally, I will present a number of optimization techniques that improve the speed of the proposed simulation method without a significant loss in simulation accuracy.

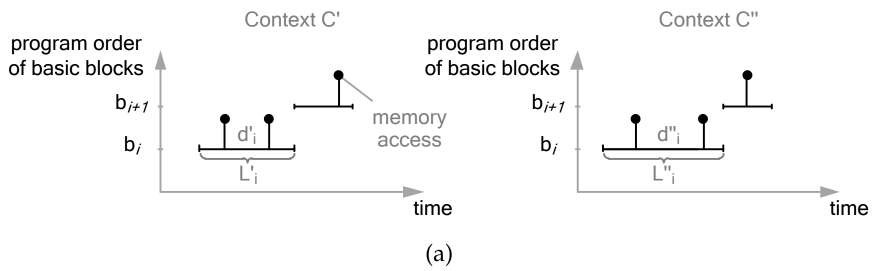
4.1 SYSTEM-LEVEL EFFECTS OF OUT-OF-ORDER EXECUTION

In this section, we will characterize the system-level effects of out-of-order execution. Particularly, we will focus on context-dependent deviations of instruction timing as well as reordering of memory accesses. In addition, we will discuss the limitations of conventional host-compiled simulation techniques in case of out-of-order target processors.

4.1.1 *Classification of out-of-order effects*

Data dependencies among instructions is one of the major issues limiting the instruction-level parallelism. In simple processors that have a statically scheduled pipeline, the program execution flow is stalled if the input operands of an instruction are not yet available [26]. To overcome this problem and reduce the total number of stalls caused by data dependencies, advanced processors perform *dynamic scheduling* of the instructions [26]. This technique allows detecting independent instructions within a certain instruction window and enabling their execution *out-of-order*, i.e. irrespectively of their program order. The execution of any instruction in the window can start as soon as the input operands and a corresponding execution unit become available. In addition, advanced processors may support *speculative execution* [26]. Speculative execution allows instructions to be scheduled and executed beyond the boundaries of basic blocks by predicting the outcome of branch instructions. Thus, in out-of-order processors the computational resources are utilized more efficiently, though, at the cost of additional hardware.

Out-of-order execution of instructions leads to a more complex timing behavior of the processor at the system level. The availability of the execution units is highly dynamic and may rapidly change during the program execution. Hence, in out-of-order processors the execution time of an instruction may not remain constant but determined by the current state of the internal processor components. In fact, each instruction of the target code starts executing within a particular *context* which defines the timing of this instruction. From the system level perspective, the execution time of an entire basic block as well as the time intervals between load/store operations may also deviate depending on the current execution context for that basic block.



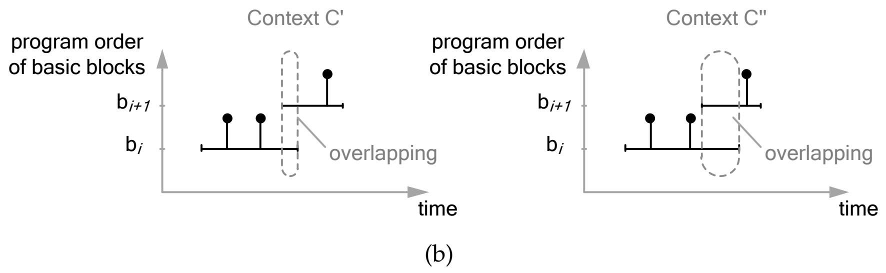


Figure 4.1: Context-dependent execution of two adjacent basic blocks in a speculative out-of-order processor: (a) processing latency L_i of block b_i and the time intervals between the two memory accesses d_i change depending on the execution context. (b) Overlapped execution of two adjacent basic blocks at two different contexts.

In the example shown in Fig. 4.1a, basic block b_i is executed at two different contexts C' and C'', representing different states of the processor's microarchitecture. Therefore, processing latency of the block L_i as well as the time intervals between the two memory accesses d_i are specific for each context.

As mentioned earlier, speculative out-of-order processors can execute instructions of the succeeding basic blocks in advance. As a result, the execution of adjacent basic blocks may be partially overlapped. In the example shown in Fig. 4.1b, several instructions of basic block b_{i+1} are executed out-of-order with respect to the instructions of basic block b_i. The execution of the both blocks is overlapped, thereby resulting in a reduced execution time. Please note that the size of the overlapping interval can be context-dependent too, as shown in the figure.

In addition to context-dependent latencies of basic blocks, out-of-order execution may result in reordering of memory instructions. As a result, the order of memory accesses performed by the processor will also change at the system level. J. L. Hennessy and D. A. Patterson in [26] described the following situations that may occur in an out-of-order processor:

- A load instruction can be executed out-of-order with respect to any another load instruction, as long as the address calculation of the reordered load does not dependent on other loads.
- A load instruction can be executed out-of-order with respect to a store instruction if the both instructions accesses memory at different addresses. Otherwise, out-of-order execution may lead to a *read-after-write* (RAW) hazard. This hazard occurs when the reordered load instruction retrieves a wrong value which has not yet been updated by the store.
- A store instruction can be executed out-of-order with respect to other memory instructions as long as this execution does not result in a *write-after-read* (WAR) and *write-after-write* (WAW) hazards. In a WAR hazard, the reordered store instruction updates the memory before it has been read by the load instruction. As a result, the load instruction will read wrong data. In a WAW hazard, two store instructions are exchanged. As a result, the correct value in the main memory will be overwritten by outdated data. Please note that during speculative execution, store operations are performed differently. In case of a branch misprediction, the processor must be able to recover its state before the mispredicted branch. Thus, store instructions can update the main memory only when the respective store instructions can commit from the pipeline, i.e. on the execution path predicted correctly. As a consequence, in speculative processors, write operations are performed in the program order at the commit stage of store instructions (for further details see [26]).

In this thesis, I refer to *intra-block* reordering if memory instructions are reordered within one basic block. Similarly to arithmetic instructions, memory instructions are dynamically scheduled during the program execution, depending on the current availability of the memory unit. Moreover, memory instructions may have data dependencies on other dynamically scheduled instructions. Thus, the actual order of memory operations may depend on the context of the basic block as well (see block b_i in Fig. 4.2). If the target processor is capable of speculative execution, memory instructions can be reordered

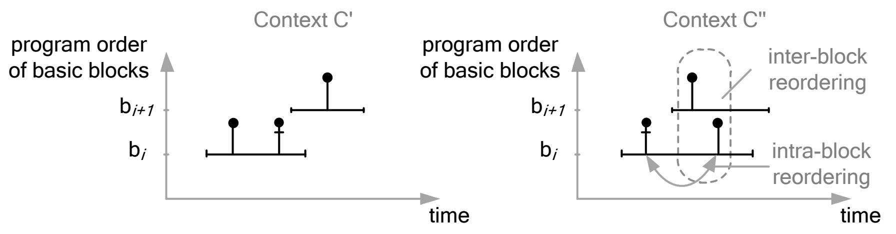


Figure 4.2: Context-dependent reordering of memory accesses. The accesses can be reordered within a single basic block (intra-block reordering) or among multiple basic blocks (inter-block reordering).

among multiple adjacent basic blocks as shown for blocks b_i and b_{i+1}. This type of memory reordering is referred to as *inter-block*. The inter-block reordering results from the overlapped execution of basic blocks as described above. Inter-block memory reordering can be context-dependent as well.

4.1.2 *Limitations of conventional host-compiled simulations*

In host-compiled simulations, the static organization of translated code (see Section 3.1.2) has a number of limitations if the target processor supports out-of-order execution of instructions.

First, the annotated timing information in basic block functions is fixed. If a basic block is executed only once, the annotated delays precisely describe the execution time of this basic block. However, if a basic block is executed multiple times, the annotated delays might not be able to reflect alternating timing behavior at different contexts correctly. Moreover, the annotated timing cannot consider context-dependent overlapping of basic blocks' execution.

Fig. 4.3 shows the timing error of conventional binary-level host-compiled simulation, which does not consider context-dependent timing of instructions and overlapped execution of basic blocks. In this experiment, the code of multiple benchmarks from MiBench [23] and MediaBench [37] were evaluated. Delays annotated in the translated code have been preliminary obtained by means of a cycle-accurate simulator of out-of-order processors *sim-outorder*[1] from the SimpleScalar suite [5]. The depicted timing error was calculated as a relative error of the execution time produced by host-compiled simulation compared to the reference value obtained by the SimpleScalar simulator. As can be seen from the experimental results, neglecting the out-of-order effects leads to a large overestimation of the execution time (from 58% for *rijndael_decode* and up to 308% for *dijkstra_small*).

1 The following processor configuration was assumed: 64-entry re-order buffer and load-store queue, 4-entry instruction fetch queue, 4 integer ALUs, 1 integer multiplier, 4 floating point ALUs, 1 floating point multiplier, 2 memory ports, decode/is-

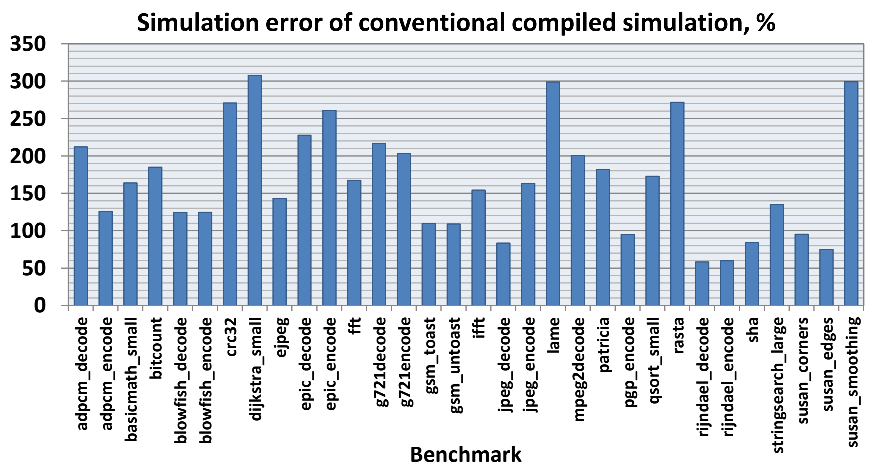


Figure 4.3: Error of timing estimation with conventional (i.e. not considering out-of-order effects) compiled simulation compared to the reference, cycle-accurate processor simulator.

The second limitation of conventional host-compiled simulation originates from the fact that the order of memory accesses within basic block functions is fixed. In case of *intra*-block reordering, this problem could be partially solved by changing the order of cache accesses in the translated code. However, if the order of memory instructions is context-dependent, the correct order of cache accesses cannot be reconstructed since the code of basic block functions is static. Moreover, *inter*-block memory reordering cannot be considered as well. In reality, an out-of-order processor can start executing a memory instruction from the next basic block out-of-order, while the current block is still being executed. However, in the translated code, basic block functions are executed sequentially and the following basic block function can start executing only after the previous one has been completed. In this case, a memory instruction cannot be simulated if its target address is determined in the basic block function which is about to be executed.

Summarizing the discussions above, we can outline the following requirements for system-level, host-compiled simulation with respect to ouf-of-order execution of instructions:

- the simulation must be capable of considering context-dependent execution delays of a basic block and overlapped execution of multiple basic blocks;
- the simulation must support reordered execution of memory instructions within one or multiple basic blocks.

sue/commit width is 4 instructions per cycle. In this experiment, instruction and data caches were assumed to be perfect.

These requirements will be addressed in Sections 4.2, 4.3 and 4.4 of this chapter.

4.1.2.1 *Non-blocking cache behavior*

In the previous section, we discussed reordering of memory instructions within one or multiple basic blocks due to dynamic scheduling of instructions. If a target processor accesses the main memory via caches, additional modeling efforts are required to reconstruct its timing behavior, as out-of-order processors are capable of *hiding* long memory access latencies.

In conventional host-compiled simulation (see Fig. 3.4), miss penalties are simply added to the overall cycle count in a situation of a cache miss. In other words, it is assumed that the execution is stalled during a miss, i.e. cache accesses are assumed to be blocking. In an out-of-order processor, this assumption is valid for instruction caches only. In fact, if an instruction cache miss occurs the processor cannot continue executing the program because the next instructions are not yet read from the instruction memory. However, in case of a data cache miss, the behavior of an out-of-order processor is different.

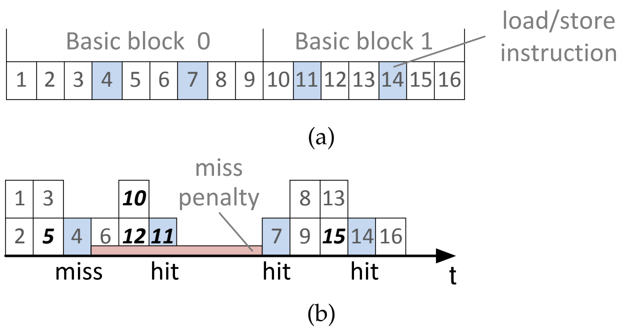


Figure 4.4: (a) Instructions of the target code in the program order; (b) timing behavior of an out-of-order processor in case of a data cache miss. Instructions, which do not depend on the active miss, continue executing out-of-order.

Assume a sequence of target instructions shown in Fig. 4.4a. In this figure, the numbers represent the sequential number of the instructions according to their program order. The highlighted rectangles denote memory instructions and the remaining rectangles denote arithmetic instructions.

The execution of this instruction sequence in an out-of-order processor is shown in Fig. 4.4b. Assume that the first memory instruction (sequential number 4) results in a miss in the data cache. Despite the active miss, the processor can execute some subsequent instructions in advance and thus partially hide the cache miss latency.

Simulation of this behavior with an assumption of blocking caches results in overestimation of the execution time. Since some instructions can continue their execution in the presence of a cache miss, the execution delays annotated after this memory instruction will not be correct anymore. Thus, host-compiled simulation requires additional means for considering the effects of non-blocking data caches. This issue will be addressed in Section 4.5.

4.2 DERIVATION OF BASIC BLOCK TIMING

4.2.1 *ISS enhancements*

In the proposed simulation approach, timing of basic blocks, which is required for code annotations, is derived using a cycle-accurate simulator of the target processor (Fig. 4.5). For this purpose, the simulator must be additionally enhanced with a monitoring unit, which captures and stores the execution information of target instructions. To enable capturing, the simulator requires the binary code of the target application as well as the boundaries of the basic blocks. These boundaries must be obtained prior to measurements. A possible algorithm for obtaining the boundaries will be presented in Section 4.2.2.

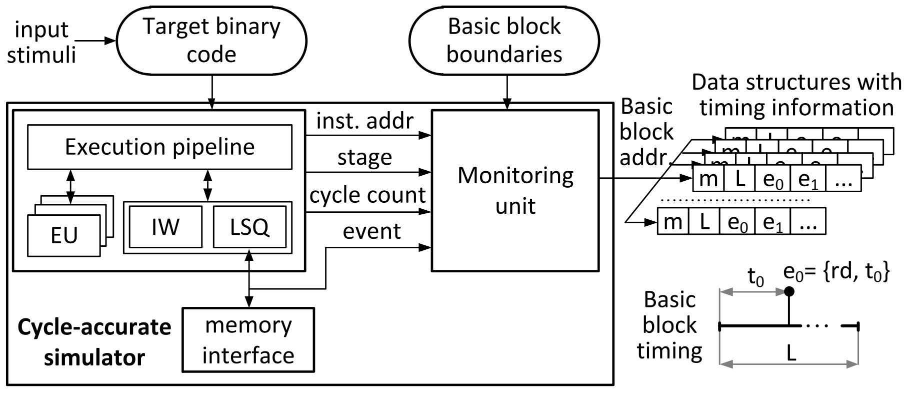


Figure 4.5: Derivation of basic block timing using a cycle-accurate simulator of the target out-of-order processor.

Obtaining of basic block timing requires a convention on when basic blocks start and stop executing. In a pipelined processor, the instructions of the target code are processed in multiple stages. For example, the execution of instructions in an out-of-order processor typically involves the following processing steps, as described by M. Johnson in [30]:

- *Fetch/Decode*: Target instructions are fetched from the program memory and placed after decoding into the *instruction window* (IW). The instruction window can be implemented either as a set of reservation stations holding the instructions for specific execution units [78] or as a centralized buffer that holds the instruction for all execution units [2, 73].

- *Issue*: In this step, instructions are checked for data and control dependencies. If there are no dependencies detected and all input operands are available, instructions are issued to the respective *execution unit* (EU).

- *Execution*: The corresponding execution unit performs computation and the result is stored in a temporary buffer. Note that in the absence of inter-instruction dependencies, instructions may be executed out-of-order if there are sufficient available hardware resources. In case of load and store operations, the processor calculates the effective memory address and places the instruction in a *load/store queue* (LSQ). The corresponding memory operation is carried out by the memory unit in the processor.

- *Commit*: At this stage, the processor registers are updated with the result from the temporary buffer. The *commit* stage decouples completing of the instruction execution and actual updating of the register file or main memory with computed values. Thus, instructions can be executed speculatively, enabling more efficient utilization of the hardware resources. Although instructions may be executed out-of-order, they are committed in the program order in order to maintain the correct data flow.

In this work, I assume that the execution of a basic block starts as soon as one of its instructions is issued to the respective functional unit. It is important to mention that the issued instruction might not be necessarily the first instruction in the program order, since out-of-order processor may change the execution order of instructions. It is further assumed that a basic block stops executing when its last instruction has been committed and removed from the pipeline. In contrast to the first issued instruction, the last committed instruction will be always the last instruction in the program order, as mentioned above. Otherwise, the correct program execution cannot be preserved.

During capturing of basic block timing (Fig. 4.5), the cycle-accurate simulator executes the target binary code and provides the following information on executed instruction to the monitoring unit:

- the instruction address, which is needed to identify the basic block the instruction belongs to,

- current pipeline stage,

- the type of memory instructions (*read* or *write*) and the time stamp of respective memory access events,

- current cycle count, which is required to obtain timing values.

For each basic block, the monitoring unit allocates a data structure holding the timing information as well as some additional control information. One of the most important timing properties of a basic block is its *execution latency*, which is denoted as L in Fig. 4.5. If a basic block contains one or multiple memory instructions, the data structure contains the type (read or write) as well as the time offset of the memory access event measured relative to the start time of the basic block (fields e_i). The memory events are captured at the output interface of the load/store queue before the local caches. Moreover, the measurement is performed assuming perfect instruction and data caches, which always result in a hit. This condition is necessary in order to obtain execution latencies of basic blocks independently from the timing characteristics of the interconnect and main memory modeled in the reference simulator. These latencies will be obtained dynamically at run-time of host-compiled simulation in a new environment, which may be different from the environment assumed in the reference simulator. Dynamic modeling of cache effects will be discussed in detail in Section 4.5.

CONSIDERATION OF BASIC BLOCK OVERLAPPING A straightforward method to obtain execution latency L of a basic block is to calculate the difference of time stamps when the block started and finished its execution. However, if the execution of basic blocks in an out-of-order processor is overlapped (see Section 4.1.2), the simulation of block latencies obtained in this way will result in an overestimated execution time, as the overlapping interval will be considered twice for adjacent basic blocks.

A simple and yet efficient solution to this problem is to assume that a basic block starts executing when the last instruction of the previous block has been committed. In the example shown in Fig. 4.6a case, the execution of two adjacent basic blocks is overlapped (at this moment it is assumed that no memory accesses are performed during the overlapped execution). In order to consider overlapping, the execution latency of succeeding basic block b_{i+1} can be reduced by the size of the overlapping interval. Although the latency does not correspond to the real timing observed in the cycle-accurate simulator, the sequential simulation of the two execution latencies of the blocks will not result in a visible timing error.

If the processor executes some memory instructions of the succeeding basic block out-of-order within the overlapped interval (event e_2 in Fig. 4.6b), the latency of the second block is determined similarly as in the previous scenario in Fig. 4.6a. However, in this situation, offset t_2 of event e_2 will have a negative value. Simulation of memory accesses that have a negative offset will be discussed in Section 4.4.2.

CONSIDERATION OF SPECULATIVE EXECUTION In case of speculative execution, certain target instructions may be flushed from the pipeline without being committed, if they have been executed on a mispredicted execution path, i.e. the outcome of preceding branch

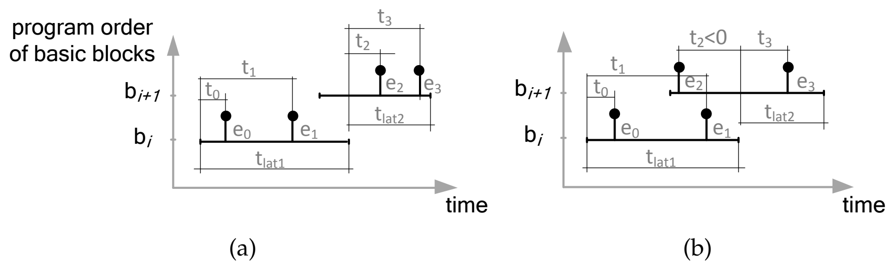


Figure 4.6: Consideration of overlapped execution of two adjacent basic block during the derivation of block timing in the cycle-accurate simulator. Two possible scenarios are considered: (a) no memory instructions are executed in the overlapping interval, (b) one or multiple memory instructions of the succeeding basic block are executed in the overlapping interval.

has been predicted incorrectly. To address this issue, each data structure holding the basic block timing (see Fig. 4.5) additionally contains misprediction flag m. When at least one instruction from the basic block gets flushed from the pipeline, this flag is set to one and the complete data structure gets invalidated. In this case, the penalty due to the branch misprediction is considered in the execution latency of the first basic block executed after the misprediction. At the same time, the invalidated timing is discarded and it must be captured once again when the basic block is executed on the path predicted correctly.

4.2.2 *Identification of basic block boundaries*

In order to obtain the execution latencies of basic blocks, it is essential to identify the boundaries of each basic block in the target binary code. The boundaries of a basic block are defined by the addresses of its first and last instructions. A possible algorithm for finding the block boundaries is shown in Algorithm 1.

As basic blocks cannot overlap, it is sufficient to determine the beginning of basic blocks (i.e. the addresses of their first instructions). The search starts from the first instruction of the target code $addr_0$, which also denotes the beginning of the first basic block, and is performed within the outer *while*-loop.

In most cases, basic blocks terminate with a *branch* instruction. This instruction can modify the value of the program counter register and, thus, change the execution flow of the program. In the nested loop shown in lines 4 and 5, the algorithm sequentially searches for the next branch instruction. In line 7, the algorithm continues if a such instruction is found[2].

2 Please note that the condition checking whether the nested loop has reached the last instruction of the code ($addr_N$) is not shown for simplicity reasons.

Algorithm 1 Identification of basic block boundaries in the target binary code ($addr_0$ and $addr_N$ are the first and last instructions in the code respectively).

```
1:  addr ⇐ addr_0
2:  while addr ≠ addr_N do
3:      known list ⇐ addr
4:      while INSTRUCTION(addr) = branch do
5:          INCREMENT(addr)
6:      end while
7:      if INSTRUCTION(addr) = direct branch then
8:          known list ⇐ branch target
9:      end if
10:     INCREMENT(addr)
11: end while
```

The *target* of the branch as well as the instruction immediately following the branch denote the beginning of new basic blocks. A branch, whose target address is known at compile time, is referred to as a *direct* branch. Direct branches are typically used to perform conditional execution (e.g. *if* statements in the C code) or to perform a function call. If the branch is direct, its target address is added to the list of known basic blocks as shown in line 8. The address of the instruction following the branch denotes a new basic block as well. It is added to the list in the next iteration of the outer *while*-loop in line 3.

In a certain type of branch instructions, the target address is stored in an architecture register and, hence, cannot be defined at compile time. Such branches are referred to as *indirect*. The target address of an indirect branch may point to an instruction of another basic block, thus, splitting the block into two new basic blocks which cannot be determined by the algorithm. These basic blocks can be identified dynamically during the reference cycle-accurate simulation.

4.3 CONTEXT DEPENDENCY OF BASIC BLOCK TIMING

4.3.1 *Concept*

To address the complex behavior of out-of-order processors in host-compiled simulation, I propose to consider multiple *context-dependent* timings of a basic block of the target code. Each timing determines the execution latency of the block at a unique execution context. In host-compiled simulation, the details of the processor's microarchitecture are abstracted. Therefore, a suitable specification of the execution context must be found first, allowing context-dependent timings to be differentiated at simulation run-time.

To find an appropriate definition of a context, it is essential to understand the cause of context diversity. During the execution of instructions, the state of internal processor's components changes. Thus, the context at which an instruction starts executing, is determined by the sequence of instructions executed previously. In general, the context is defined by all instructions executed from the start of the program till the current instruction. However, it is reasonable to assume that only *recent* instructions have the largest impact. Thus, to differentiate possible contexts of an instruction, we can take into a consideration a sequence of previous instructions within a certain observation window of a fixed size.

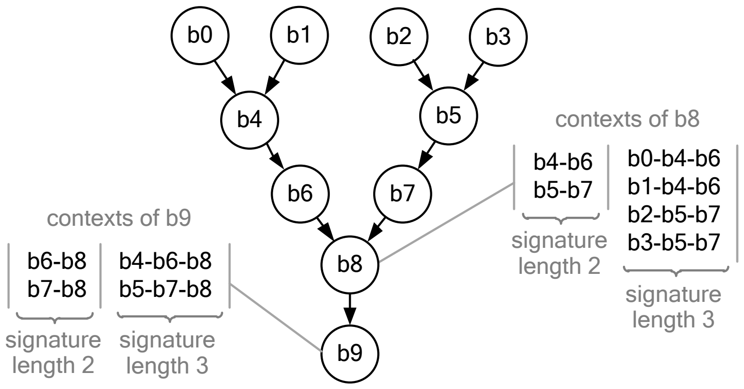


Figure 4.7: Consideration of multiple contexts for basic blocks. The size of context signatures determines the amount of contexts that can be differentiated for each basic block.

The execution flow of instructions changes at the boundaries of basic blocks only. Therefore, we can similarly define a context of a basic block by the sequence of previously executed basic blocks. I refer to a possible sequence of the preceding blocks as a *context signature* for the given basic block. The length of a context signature determines the amount of contexts that can be captured for a basic block. In the example shown in Fig. 4.7, basic block b_8 can be reached in 4 different paths in the depicted control flow graph. If the signature length equals two, 2 contexts for block b_8 with signatures b_4–b_6 and b_5–b_7 can be differentiated. Similarly, at the same signature length, basic block b_9 will have 2 contexts with signatures b_6–b_8 and b_7–b_8. Please note that a context signature must unambiguously identify the sequence of previous blocks. For this purpose, we can employ the start addresses of basic blocks, as the start addresses are always unique in the target program.

If the length of signatures is increased, more contexts can be differentiated for a given block. For example, with the signature length of 3, block b_1 (Fig. 4.7) will have 4 contexts with signatures b_0–b_4–b_6, b_1–b_4–b_6, b_2–b_5–b_7, b_3–b_5–b_7. In turn, for block b_9 the number of distinguishable contexts will not change, i.e. there will be 2 contexts with signatures b4–b6–b8, b5–b7–b8. The length of context signatures can be arbitrarily chosen. In most cases, with a larger signature more contexts of a basic block can be specified and, hence, more context-dependent timings can be associated with the block. Consequently, by employing larger signatures, the timing behavior of the target processor can be reproduced more accurately. However, as we will see in the following sections, larger signatures result in a larger translated code, thereby deteriorating the simulation speed.

4.3.2 *Derivation of context-dependent timing*

4.3.2.1 *Concept*

In order to support the context dependency, the monitoring unit of the reference cycle-accurate simulator has to be enhanced as shown in Fig. 4.8. During the program execution, the unit temporarily stores the start addresses of the previously executed blocks in a buffer. The buffer operates as a sliding window, storing the addresses of last N blocks. When the execution of a basic block is completed, the content of the buffer is used to construct a context signature of size N for the captured basic block timing. Afterwards, if the current context has not been previously recorded, the timing and its signature is added to the central database. Thus, the database stores multiple timings for each basic block.

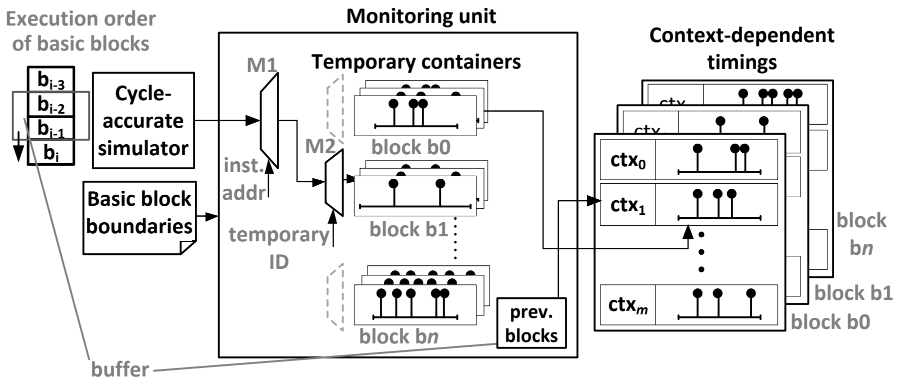


Figure 4.8: Measurement of multiple context-dependent timings per one basic block. Each timing is tagged with context signature ctx_i, which is a sequence of previously executed basic blocks.

During speculative execution, the pipeline of the processor may contain instructions of multiple basic blocks simultaneously and the execution of multiple basic blocks is overlapped. A special case occurs when the pipeline of the processor contains multiple instances of the same basic block executing at a time, e.g. as part of a loop. During such overlapped execution, it is particularly difficult to distinguish the events and timing properties of different block instances.

To address this issue, for each basic block the measurement unit allocates a set of temporary containers (see Fig. 4.8). These containers temporarily store timings of all currently active instances of the basic blocks. The instances of basic blocks can be recognized by differentiating the instances of their instructions. In turn, the instructions can be differentiated by their sequence numbers obtained at the fetch stage. To enable this, the sequence number of each instruction instance is associated with a temporary ID number. The temporary ID number is always unique among the currently active instances.

For example, assume a simple program shown in Fig. 4.9. The program consists of three basic blocks where the second block constitutes the body of a loop. If there are no data dependencies between the loop iterations, the processor can fetch and execute multiple instances of the loop body within a certain instruction window. The instructions of the first and second iteration are assigned a temporary identifier of 0 and 1 respectively. This identifier is used as to select an appropriate temporary container in the monitoring unit (Fig. 4.8). Thus, knowing the sequence number of an instruction, we can always determine its temporary ID and associate this instruction with the respective temporary container. When an instruction instance commits, its temporary ID is released and reused by the following instances.

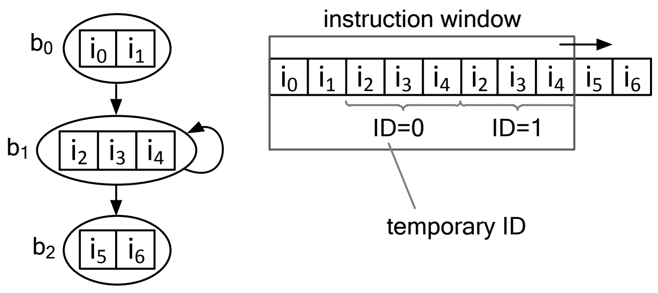


Figure 4.9: Differentiation of basic block instances in the processor's pipeline. Instances of the same instruction are assigned a temporary ID. The ID is related with the sequence number of instruction's instances.

4.3.2.2 *Implementation details*

The algorithm for obtaining context-dependent timing of basic blocks is shown in Fig. 4.10. The algorithm is executed every time when the stage of a target instruction changes. If the instruction is at the *fetch* stage, the monitoring unit allocates a temporary ID for the current instruction instance. The ID is used to select an appropriate temporary container for the block timing.

The timing measurement of a basic block is triggered as soon as one of its instruction moves to the *execute* stage. The monitoring unit allocates a new temporary container for the current basic block in-

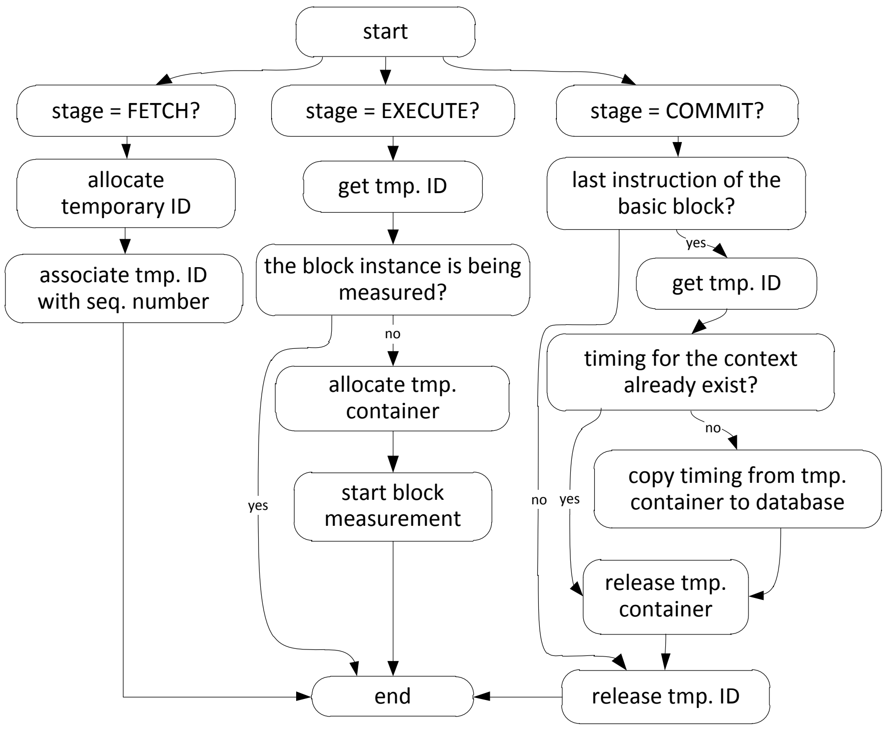


Figure 4.10: Algorithm for obtaining context-dependent timing of basic blocks.

stance, which will hold all timing information of this block instance. The measurement stops when the last instruction of the basic block has been committed and left the pipeline. At this moment of time, the measurement unit retrieves the sequence of previously executed blocks from the buffer to obtain the context signature of the timing. If the timing for this context has been already measured before, the timing is discarded. Otherwise, the timing is tagged with the context signature and copied from the temporary container into the database. Afterwards, the temporary container is released and can be reused for other block instances. In the last step, the temporary ID assigned for the last instruction is released as well.

Listing 4: Allocation of a temporary ID for instruction instances

```
void allocate_tmp_id (addr_t iaddr, seq_t seq)
{
    int id = global_id[iaddr];
    CREATE_HASH(seq, id);

    global_id[iaddr]++;
    global_id[iaddr] %= MAX_OVERLAP;
}
```

Allocation of a temporary ID is presented in Listing 4. The temporary IDs are treated separately for every instruction. The IDs are derived using array `global_id`, which is indexed using the instruction addresses. Each element of the array contains the next free temporary ID that can be used for the instruction instance. After the assignment, the global index is incremented in a circular way by applying a modulo operation. Value `MAX_OVERLAP` must be pre-defined by the user and denotes the maximum amount of instances contained in the pipeline at a time. The ID lookup operation from the sequential number is a time-consuming task, which is often performed during the measurement. To speed up this operation a hash table is employed.

4.3.3 *Context-aware host-compiled simulation*

In order to enable context-dependent timing in host-compiled simulation, basic block functions must be capable of reflecting multiple timings. An example of a block function incorporating multiple context-dependent timings is presented in Fig. 4.11. Consideration of context dependency during the execution of translated code requires a new data structure, which holds the sequence of recently executed basic block functions. For this purpose, a *history* FIFO is employed, which temporarily stores the sequence of start addresses of the previously executed blocks. The size of the FIFO must correspond to the length of context signatures which was used during capturing of basic block timings.

```
// context signatures of length 2
addr_t history_FIFO[2];
// context-aware implementation
void b_0x700() {
   if (in_FIFO(0x100, 0x500)) {
       // use timing 1
       reg[2] = r[5] + 16;
       cycle += 2;
       ...
   }
   else if (in_FIFO(0x200, 0x500)) {
       // use timing 2
       reg[2] = r[5] + 16;
       cycle += 5;
       ...
   }
   ...
   add_to_FIFO(0x700);
}
```

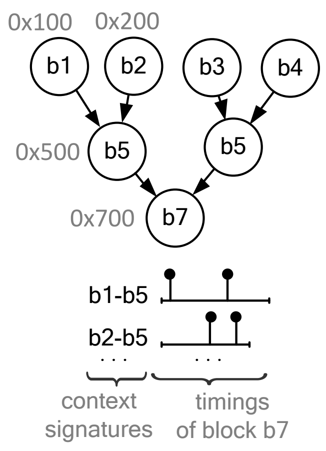


Figure 4.11: Basic block function incorporating multiple context-dependent timings of the basic block.

During simulation, the current context of a block function is determined by evaluating the content of the history FIFO in the beginning of the function. Depending on the block sequence stored in the FIFO, the appropriate execution delays of the block are selected using multiple *if*-statements. In the presented example, the signature length equals two. Thus, the history FIFO holds the last two blocks executed prior to the current function. When the execution of the block function is completed, the start address of the current basic block is added to the FIFO, replacing the least recently executed block. This address forms a new context signature for the succeeding block function.

4.4 REORDERING OF MEMORY ACCESSES

In this section, I address intra- and inter-block reordering of memory accesses. First, I introduce a classification of basic blocks with respect to reordering of data memory accesses. Afterwards, I provide a solution for handling the reordering at run-time of host-compiled simulation.

4.4.1 *Classification of basic blocks*

Overlapping of basic block depends on several factors including the size of the instruction window, amount and availability of the execution units and data dependencies among the instructions. Depending on the degree of block overlapping, several types of memory reordering may exist as shown in Fig. 4.12. In the first type (shown in basic block b_0), the memory accesses are reordered locally within the same basic block. In the second type, memory accesses are reordered between two *adjacent* basic blocks as shown for basic blocks b_2 and b_3. Finally, reordering may occur over multiple basic blocks, whose execution is overlapped as shown for block b_4. In the presented example, the memory access performed in block b_4 is reordered with the memory operations of multiple blocks b_1, b_2 and b_3. To handle memory reordering at run-time of compiled simulation correctly, it is essential to identify basic blocks involved in memory access reordering. Such basic blocks are referred to as *out-of-order*. In contrast to *normal* basic blocks, in which reordering of memory accesses never occurred, out-of-order blocks require a different simulation mode, as will be discussed later.

Classification of basic blocks is performed after the derivation of block timings in the reference cycle-accurate simulation. If the order of memory accesses within a basic block did not match the actual program order of memory instructions, the first type of memory reordering is detected. In this case, the respective basic block is classified as out-of-order. The second type of reordering can be detected by analyzing the timing offsets of memory accesses. As described in Section 4.3.2.1, if two adjacent blocks overlap and the succeeding block performs a memory operation in the overlapped region (see Fig. 4.6b),

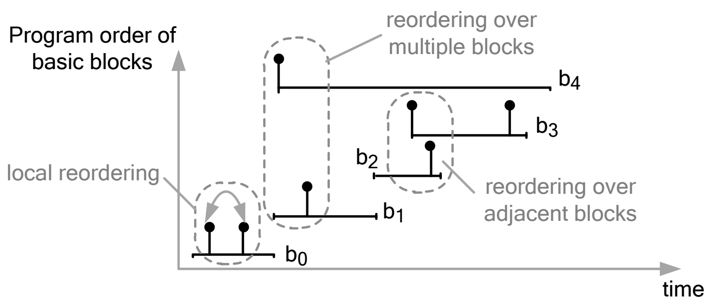

Program order of basic blocks
reordering over multiple blocks
b4
b3
b2
local reordering
reordering over adjacent blocks
b1
b0
time

Figure 4.12: Three possible types of memory access reordering: local reordering within one basic block, reordering among two adjacent basic blocks and reordering over multiple basic blocks.

this memory operation has a negative offset. If a basic block has an event with a negative offset, this block as well as the preceding block are classified as out-of-order.

The third type requires more careful consideration. In this case, more than one preceding basic blocks must be classified as out-of-order. We can employ a conservative algorithm that determines all possible preceding blocks that *might* be involved in reordering of memory accesses. The algorithm performs backward analysis of basic block timings as presented in Fig. 4.13. Let us assume that the execution of basic block b_0 is overlapped with the previous basic blocks, and the block's execution latency equals L_0. Moreover, let us assume that memory access event $e0$ of that block occurs in the overlapped region and, hence, it has negative offset t_0. The algorithm finds all possible preceding blocks, whose execution may overlap with block b_0, thereby not exceeding interval t_0.

For the following explanation, we define *complete execution latency* of an i^{th} basic block L'_i as

$$L'_i = \begin{cases} L_i + |t_{i,0}|, & \text{if } t_{i,0} < 0, \\ L_i, & \text{if } t_{i,0} \geqslant 0, \end{cases} \tag{1}$$

where L_i is the execution latency of the basic block as defined in Section 4.3.2.1, and $t_{i,0}$ is the offset of the first memory access event. If the basic block does not contain memory instructions, $L'_i = L_i$ as well.

The algorithm estimates the sum of complete execution latencies of previous basic blocks denoted as L'. The algorithm starts with the first level of preceding basic blocks (b_1, b_2 and b_3 in Fig. 4.13). Firstly, the complete execution latencies of these blocks are evaluated. If a preceding block has multiple context-dependent timings, the algorithm conservatively takes the timing with the smallest complete execution latency. If the complete latency of the preceding block is not larger than the absolute value of t_0, the preceding block is considered as out-of-order. In the shown example, all three preceding blocks b_1, b_2

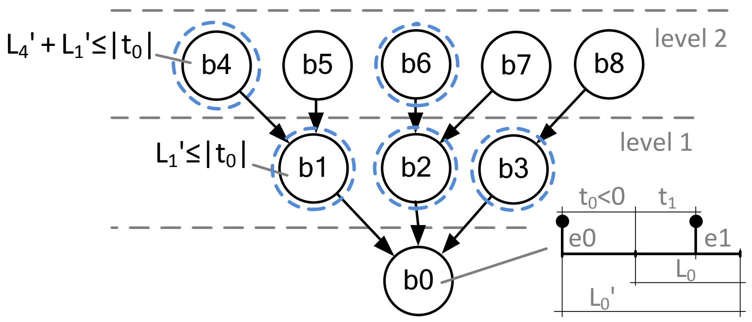


Figure 4.13: Detection of possible memory reordering over multiple basic blocks for basic block b_0. If the sum of complete latencies along a backward path in the CFG is not greater than $|t_0|$, the respective basic blocks constituting the path are considered as out-of-order (blue dashed circles).

and b_3 are classified as out-of-order and respectively denoted by the dashed blue circles.

In the next step, the algorithm identifies the next level of preceding basic blocks by evaluating the context signatures of timings of blocks b_1, b_2 and b_3. For each block in the new level, the algorithm again determines the smallest complete execution latency and adds it to the previous latency on the same path, e.g. $L'_4 + L'_1$ for path b_4–b_1. If the sum is not greater than $|t_0|$, block b_4 is considered as out-of-order. However, if the sum is larger than $|t_0|$, e.g. as for block b_5, the algorithm stops further analysis on the respective path. The similar analysis is performed for all possible backward paths of block b_0.

Please note that a preceding basic block is considered out-of-order independently of whether it contains any memory instructions, as long as its complete execution latency fits to interval $|t_0|$. We will discuss the need for this assumption in the following section, in which simulation of memory reordering is discussed. The presented algorithm is performed for every basic block of the target code. Basic blocks, which have never been classified as out-of-order during the analysis, are classified as normal.

4.4.2 *Simulation of memory reordering*

In Section 3.1.2, we discussed a conventional way of performing host-compiled simulation, in which basic block functions are executed sequentially according to the program order of the target code. For normal basic blocks, the sequential execution of basic block functions is capable of reconstructing the correct order of memory events. In the example shown in Fig. 4.14, basic blocks b_0 and b_1 have been classified as normal. They are simulated in a *normal* mode of host-compiled simulation.

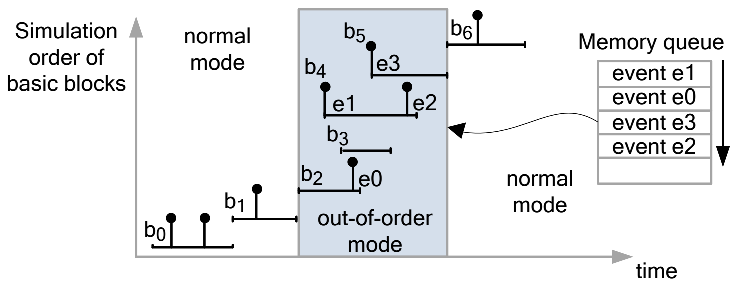


Figure 4.14: Simulation of reordered memory accesses. In the normal mode (blocks b_0, b_1, b_6), the memory accesses are simulated immediately. In the out-of-order mode (blocks b_2–b_5), memory events are placed in a sorted queue, which reconstructs the correct event order at simulation run-time. The events in the queue are simulated before the mode switches back to normal.

In contrast, basic blocks b_2, b_3, b_4 and b_5 have been classified as out-of-order (Fig. 4.14). The sequential simulation of the respective basic block functions will produce a wrong order of memory access events $e0$–$e1$–$e2$–$e3$. To solve this issue, the respective basic block functions must be executed in the *out-of-order* mode. In this mode, the simulation time is not yet advanced as the basic block functions execute and the memory access events are temporarily placed to a *memory queue*. Each event in the queue is tagged with an absolute time stamp, which is calculated using the event's offset. To reconstruct the correct event order, the queue must be sorted by the time stamps. Simulation in the out-of-order mode continues till the next normal basic block is reached. At this point, the memory events are simulated according to the reconstructed order and the simulation time is advanced. Afterwards, the simulator flushes the contents of the queue, which remains empty till the next first out-of-order block.

Please note that due to consistency reasons, block b_3 (Fig. 4.14) must be executed in the out-of-order mode as well, although it does not contain any memory instructions. It is necessary since the following block b_4 performs reordered memory operations, and the out-of-order mode must be preserved for obtaining the correct order. Because of this reason, block b_3 must be classified as out-of-order during the classification phase.

The size of the memory queue may not be sufficient for storing all memory events in the out-of-order mode. In this case, the simulation in the out-of-order mode must be stopped and the queue must be emptied to avoid possible drops of memory accesses. This situation may potentially lead to an error in the order of memory accesses. However, the resulting error will not be significant if the selected size of the memory queue is large enough compared to the size of the processor's instruction window.

The memory queue improves the accuracy of host-compiled simulation with respect to the order of memory accesses. However, the run-time manipulation on the memory queue, e.g. insertion of new elements and their sorting, requires additional computational efforts. In Section 4.6, we will discuss a possible optimization technique that addresses this issue.

4.5 NON-BLOCKING BEHAVIOR OF DATA CACHE

4.5.1 *Modeling of non-blocking behavior*

During the program execution, an out-of-order processor fetches multiple target instructions at a time. A certain part of these instructions may have data dependencies among each other[3]. Dependent instructions cannot start executing until all input data dependencies are resolved. In turn, independent instructions can be executed immediately. In case of a long-latency miss in the data cache, a processor can continue executing subsequent instructions out-of-order if two conditions are fulfilled. First, the subsequent instructions are independent of the data being currently retrieved from the main memory. Second, the respective execution units are available.

Please note that not only arithmetic operations can be performed out-of-order. If the data cache supports non-blocking behavior, the processor can perform subsequent independent memory operations in the presence of an active miss. Generally, non-blocking data caches may support the following operation modes, as described in [26]:

- *hit under miss*: In this mode, another memory access request can be served by the cache during an active miss if the requested data is present in the cache.
- *miss under miss*: In this mode, multiple outstanding misses can overlap in time. This policy imposes large bandwidth requirements on the main memory and is often employed in recent high-performance processors (e.g. in Intel Core i7) [26] or in large server systems [56].

In this thesis, I focus on data caches implementing the hit-under-miss policy. Such caches are employed in many recent embedded processors, e.g. ARM1136 [91] or Freescale e5500 [93]. In this type of non-blocking behavior, only one cache line can be retrieved from the memory at a time, i.e. there can be only one outstanding miss. In the following, I refer to a cache miss as *dependent* if one or multiple subsequent instructions in the instruction window of the processor depend on the missing data. This type of misses is caused by load instructions retrieving the data which is then processed by other instructions. An

3 We assume that instruction a depends on instruction b if a's input operand depends on the b's output (directly or indirectly via other instructions).

independent cache miss is typically caused by store instructions. However, an independent miss can be caused by a load instruction as well. This situation occurs when the instruction depending on the retrieved data has not yet been fetched and placed in the instruction window.

The timing behavior of out-of-order processors is different for dependent and independent misses. Both scenarios are discussed separately in the following sections.

4.5.1.1 *Out-of-order execution of instructions under a dependent miss*

To understand the timing behavior of an out-of-order processor in case of a dependent miss, let us assume the following target code:

```
 1  addiu  r16, r16, 20
 2  lw     r2, 4(r16) // access a0
 3  sub    r5, r7, r9
 4  lw     r3, 4(r17) // access a1
 5  addiu  r4, r3, 1
 6  sub    r6, r4, r5
 7  slt    r9, r2, r6
 8  andi   r16, r9, 240
 9  lw     r5, 8(r16) // access a2
10  addu   r1, r3, r5
```

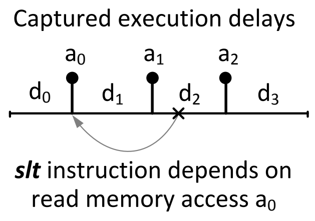

Captured execution delays
a_0
a_1
a_2
d_0
d_1
d_2
d_3
slt instruction depends on read memory access a_0

Figure 4.15: Example target code (left) and the respective timing captured during the reference cycle-accurate simulation with perfect caches (right); `slt` is the first instruction which depends on read access a_0.

The code contains three load instructions `lw`, which perform read accesses a_0, a_1 and a_2 to the data cache. The first instruction depending on the data retrieved by a_0 is `slt` in line 7 (the dependent data is stored in register r2). All instructions between a_0 and `slt`, including access a_1 caused by the second load instruction, are independent of a_0. In turn, access a_2 depends on access a_1 (indirectly over multiple instructions). The execution delays of the code obtained in the reference cycle-accurate simulation are shown in the timing diagram in the right part of Fig. 4.15. As discussed in Section 4.2, the execution delays are measured assuming perfect caches. The diagram shows the data cache access as well as the time intervals d_i between them.

Now assume that if the presented code is executed in the cycle-accurate simulator with a *realistic* data cache, access a_0 results in a miss. In the following sections, we will discuss different cache scenarios. In all of these scenarios, the goal is to reconstruct the processor's timing behavior at miss a_0 as accurate as possible during host-compiled simulation.

SCENARIO 1: HIT AFTER MISS In the first scenario, let us assume that the following accesses a_1 as well as a_2 result in a cache hit. In this case, the execution of the code in the reference simulator with a realistic data cache is shown in detail in Fig. 4.16a.

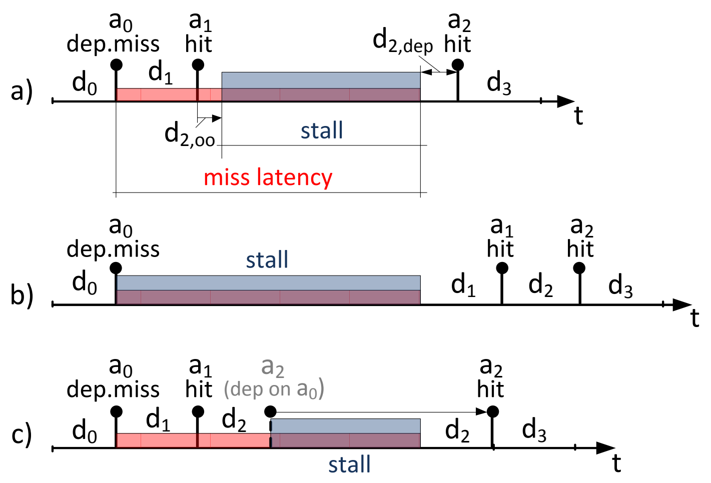


Figure 4.16: Timing behavior of an out-of-order processor during a hit under a dependent miss: (a) behavior in the cycle-accurate simulator, the execution stalls at the dependent instruction after the hit, (b) simulated behavior assuming blocking cache accesses, the execution time is overestimated by interval $d_1 + d_{2,oo}$, (c) approximated behavior, in which the overlapped execution continues until the first dependent memory access and the execution time is overestimated only by interval $d_{2,oo}$.

During the miss on a_0 (the miss penalty is denoted by the red rectangle), the processor continues executing further instructions, which are independent of a_0, in an out-of-order fashion without stalling. Access a_1 does not dependent on a_0. Therefore, the cache can supply data to the processor during the active miss according to the hit-under-miss policy. The out-of-order execution stalls on the first dependent `slt` instruction (residing between accesses a_1 and a_2), until the missing cache line is arrived from the memory. The stall interval is marked by the blue rectangle in the figure. Thus, the processor can hide the miss latency by executing the independent instructions out-of-order in parallel to the cache miss. As a result, execution delay d_1 and a part of delay d_2 (denoted as $d_{2,oo}$) are *masked* by the cache miss latency. When the missing data arrives, the processor continues executing further instructions (execution delay $d_{2,dep}$).

Conventional host-compiled simulation of the code with blocking cache behavior is shown in Fig. 4.16b. Here, the execution in the processor stalls directly after the miss, i.e. the miss penalty is simply added to the execution delays. As a result, the total execution time of the code is overestimated by the value of $d_1 + d_{2,oo}$.

The exact time interval after which the execution stalls, i.e. the value of $d_{2,oo}$, is difficult to predict. This interval could be measured in the reference simulator with a realistic cache. However, this solution would not be feasible for large programs. Particularly, this solution would require a very large number of independent measurements, where a miss had to be forced for every load instruction. A simple and yet efficient solution is presented in Fig. 4.16c. In this approximation, the out-of-order execution during the miss is simulated as follows:

1. The execution delays d_i are simulated in parallel to the miss until the first dependent access is discovered (access a_2 in the example).
2. When the dependent access is discovered, the processor execution is stalled till the missing data is arrived.
3. Afterwards, the execution delay preceding the dependent access (d_2) is simulated again. It is a conservative assumption since the exact stalling point within interval d_2 is not known.
4. The following access a_2 resulting in a hit and execution delay d_3 are simulated as usual.

The conservative approximation results in overestimation of the total execution time. The resulting error will be determined by the exact position of the stalling point (i.e. by the relation of $d_{2,dep}$ and d_2 intervals in the presented example). The error is minimal if the dependent instruction is executed right after the last independent hit (right after access a_1 in the example). In turn, this error is maximal if the dependent instruction is executed just before the first dependent access (just before a_2 in the example). Thus, the error is generally bounded by the size of the preceding interval (the size of interval d_2 in the example).

SCENARIO 2: MISS AFTER MISS In the second scenario, a second miss occurs during the active first miss. Fig. 4.17a shows the execution of the code in the cycle-accurate simulator with a realistic data cache when both a_0 and a_1 result in a miss.

A cache implementing the hit-under-miss policy can retrieve only one cache line at a time. Hence, the simulation of a_1 miss penalty can start only when the miss on a_0 is completed. Afterwards, the data retrieved from a_1 is processed by instruction `addiu` (line 5 in Fig. 4.15), which directly follows a_1. Since no independent instructions can be executed out-of-order during the second miss, i.e. latencies d_2 and d_3 are not masked by the second miss penalty. Fig. 4.17b shows conventional host-compiled simulation with a blocking data cache. Under this assumption, the total execution time is overestimated by interval d_1.

With the approximation which was introduced in the previous section (Fig. 4.17c), execution delay d_1 will be masked. Afterwards, the

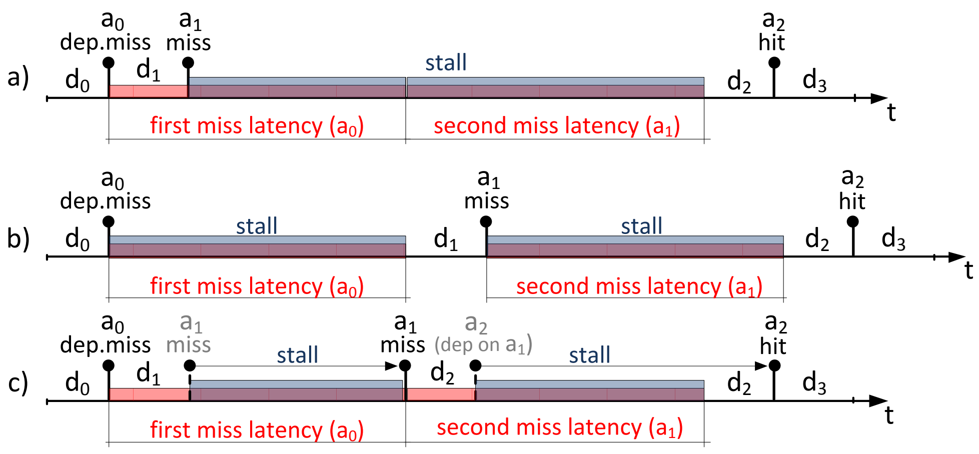


Figure 4.17: Timing behavior of an out-of-order processor during a second miss (access a_1) under an active miss (access a_0): (a) behavior in the cycle-accurate simulator, (b) simulated behavior assuming blocking cache accesses, the execution time is overestimated by interval d_1, (c) approximated behavior, in which overlapped execution continues till the first dependent memory access without a visible timing error.

execution is stalled till the first miss is completed. When the simulation of the second miss latency is started, the following latency d_2 is masked till the first dependent access a_2 is discovered (a_2 indirectly depends on a_1). At this point, the processor model stalls again till the data from the second miss is arrived. According to the conservative approximation made in the previous section, delay d_2 is simulated again, followed by the simulation of access a_2. In this case, the conservative approximation does not result in a visible timing error.

4.5.1.2 *Out-of-order execution under an independent miss*

In case of an independent data cache miss (e.g. on a store instruction), two possible scenarios should be considered. In the first scenario (Fig. 4.18a), all subsequent accesses to the cache result in a hit. According to the hit-under-miss policy, the cache is capable of supplying the requested data and the execution is not stalled. As a result, the miss latency is completely hidden. The reconstruction of this behavior does not require much efforts. Particularly, after initiating the simulation of the miss penalty, the processor model has to continue simulating of the execution delays in parallel, as if there was no miss.

In the second scenario (Fig. 4.18b), one of the subsequent accesses results in a miss. Since there can be only one outstanding miss in a hit-under-miss cache, the memory access starts only when the first miss is ready. However, if the second miss is independent as well, the

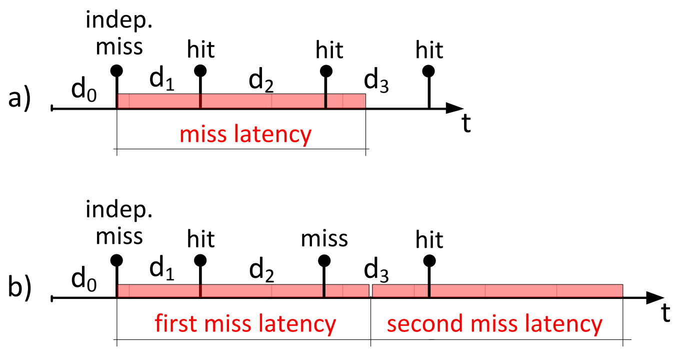


Figure 4.18: Timing behavior of an out-of-order processor under an independent miss: (a) the following memory accesses result in a cache hit; the miss latency is completely hidden, (b) one of the following accesses results in a second miss (miss under an active miss); both cache miss latencies are still hidden.

execution can continue[4]. As a result, both cache miss latencies will be hidden. The reconstruction of this behavior is performed in the same way as in the scenario shown in Fig. 4.18a.

4.5.1.3 *Consideration of instruction window*

In the approximated timing behavior presented above, we assumed that under an active miss the out-of-order execution continues till the first cache access dependent on the active miss is discovered. In reality, the duration of out-of-order execution is limited by the size of the instruction window of the processor. In fact, there may be more instructions independent of the active miss than the instruction window can contain. However, they cannot be fetched and executed because the instruction window has a limited size. The new instructions can be fetched and placed into the window only when the memory instruction which caused the miss can be committed. If a such situation occurs, the active miss causes head-of-line blocking for the subsequent independent instructions in the program flow.

An example of head-of-line blocking observed in the cycle-accurate simulator is shown in Fig. 4.19a. During the active miss, the processor continues executing subsequent independent instructions out-of-order. The execution is stalled after interval $d_1 + d_{2,W}$ since the instruction window is full and no further independent instructions can be fetched. When the missing data arrives from the memory, the processor commits the instructions which have been executed out-of-order, including the one caused the miss. Afterwards, new instructions can be fetched and executed. Due to the limited size of the window, two intervals $d_{2,W}$ and $d_{2,new}$ will be observed in place of

4 If the second miss was dependent, this scenario would resemble the scenarios discussed in the previous section

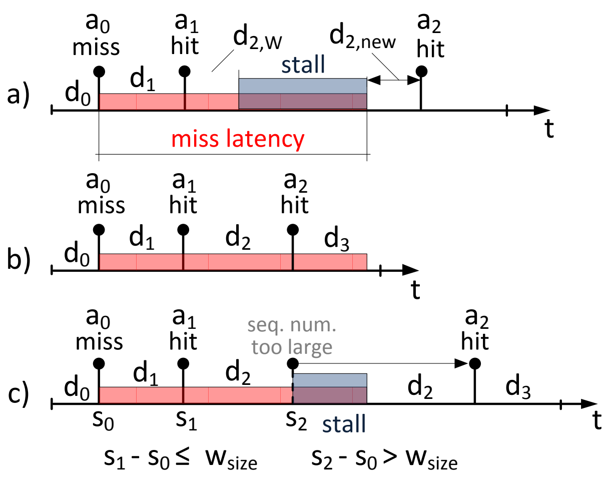


Figure 4.19: Timing behavior of an out-of-order processor during a an active data cache miss a_0 followed by two independent hits a_1 and a_2: (a) behavior of the cycle-accurate simulator, the execution stalls because no further independent instructions can be fetched due to the limited size of the instruction window, (b) simulated behavior that does not consider the limited window's size, the total execution time is underestimated, (c) approximated behavior considering the limited size of the instruction window, the execution stalls at the first memory instruction not fitting into the window.

interval d_2. Interval $d_{2,W}$ represents the execution latency of the independent instructions which were present in the window during the miss. Interval $d_{2,new}$ represents the execution latency of those independent instructions that had to be fetched after the miss.

If the limited window's size is not considered in the processor model (Fig. 4.19b), the execution will continue without stalling, assuming that the processor contains an infinitely large instruction window. As a result, the complete miss latency will be erroneously hidden by execution delays d_1, d_2 and d_3, resulting in underestimation of the total execution time.

Similar to stalls caused by data dependencies, the exact point in time at which the processor stalls due to head-of-line blocking is difficult to predict. Fig. 4.19c shows approximated simulation of the processor's timing behavior based on the sequential numbers of the instructions. The sequential numbers can be easily determined in host-compiled simulation. In this approach, the execution stalls on the first memory instruction which does not fit to the instruction window. In the presented example, the sequential number of the memory instruction causing the miss is s_0. The memory instruction with sequential number s_1 (performing access a_1) still fits into the window of size w_{size}, since $s_1 - s_0 \leqslant w_{size}$. In turn, sequential number s_2 of the memory instruction performing access a_2 is too large to fit into the window. The exact stalling point of the processor between access a_1 and

a_2 is not precisely known. Therefore, the model stalls when access a_2 is discovered. When the missing data is arrived, preceding latency d_2 is simulated once again to obtain a conservative estimation of execution time. For this kind of approximation, the total execution time will be only slightly overestimated with an error not exceeding the value of preceding interval d_2.

4.5.2 *Dependency analysis of memory instructions*

Modeling of out-of-order cache effects discussed in the previous sections requires information on dependency among memory instructions. A possible way to determine the dependencies is to statically analyze the target binary code. The analysis requires knowledge of the ISA-specific format which defines source and target registers of the instructions. However, not all dependencies can be identified statically since some of them may not be known at compile-time, e.g. due to data-dependent branches [26]. In this work, the data dependencies among instructions are determined during the timing measurements of the target code in the cycle-accurate processor simulator. Similarly to K. Lee *et al.* in [40], the pairs of dependent instructions are identified by analyzing the dependency chains constructed by *sim-outorder* simulator from the SimpleScalar suite. After the measurement, the pairs are processed to obtain dependent memory instructions as described below.

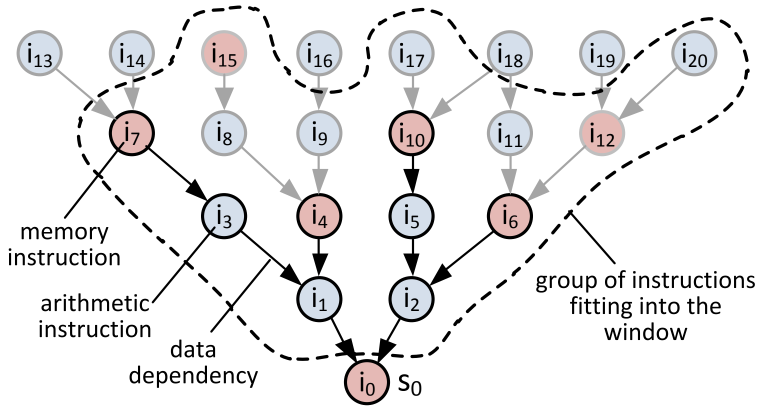


Figure 4.20: Backward analysis allowing to identify memory instructions, which given memory instruction i_0 depends on. The nodes and edges of the graph represent preceding instructions and data dependencies among them.

In order to determine other memory instruction, which given memory instruction i_0 depends on, we can construct a graph consisting of the instruction's predecessors as shown in Fig. 4.20. The branches of the graph represent data dependencies among the instructions. The search is performed among the instructions preceding i_0, which fit with the given instruction in the instruction window. For such instructions, sequential number s_i satisfies condition

$$s_0 - s_i \leqslant w_{\text{size}}, \tag{2}$$

where s_0 is the sequential number of instruction i_0 and w_{size} is the size of the instruction window.

The algorithm starts at the first level of preceding instructions and gradually expands along each of the dependency paths. In each path, the algorithm searches for the first memory instruction still fitting into the instruction window. For example, it is instruction i_4 in dependency path i_1–i_4–i_8–i_{15}. K. Lee *et al.* in [40] claimed that consideration of only one most recent instruction is sufficient during the dependency analysis. Our experiments also showed that simulation accuracy is not significantly changed if more than one memory instructions are considered. Thus, although instruction i_0 depends on other memory instructions on the path (e.g. i_{15}), consideration of only i_4 instruction is sufficient for accurate out-of-order modeling.

When the first memory instruction satisfying the condition in Eq. (2) is found, the analysis on the current path is stopped and the search continues in the next dependency path. As a result, the algorithm produces a group of memory instructions, which the given memory instruction depends on. In the presented example, instruction i_0 depends on memory instructions i_4, i_6, i_7 and i_{10}.

The algorithm is repeated for every memory instruction of the target code. If the analysis reveals a load instruction, which has no memory instructions depending on it, the cache misses caused by this instruction is always considered as *independent* and handled as described in Section 4.5.1.2. Otherwise, this instruction may eventually cause a dependent miss which is then handled as discussed in Section 4.5.1.1.

4.6 OPTIMIZATIONS

The use of context-dependent timing of basic blocks increases the diversity of the simulated timing behavior of the processor and, hence, increases the accuracy of the produced timing estimates. As mentioned earlier, with a larger signature length, more timings can be differentiated and associated with a basic block. The total number of context-dependent timings depends on several factors. Firstly, it depends on the structure of the control flow graph of the target binary code. The graph determines how many preceding basic blocks a current basic block might have. The more ramified the control flow graph is, the more possible execution paths can lead to each basic block, and hence more contexts can exists at a given length of context signatures. The second factor is the input data which is applied to the target code. The input data determines the coverage of executed blocks in the control flow graph.

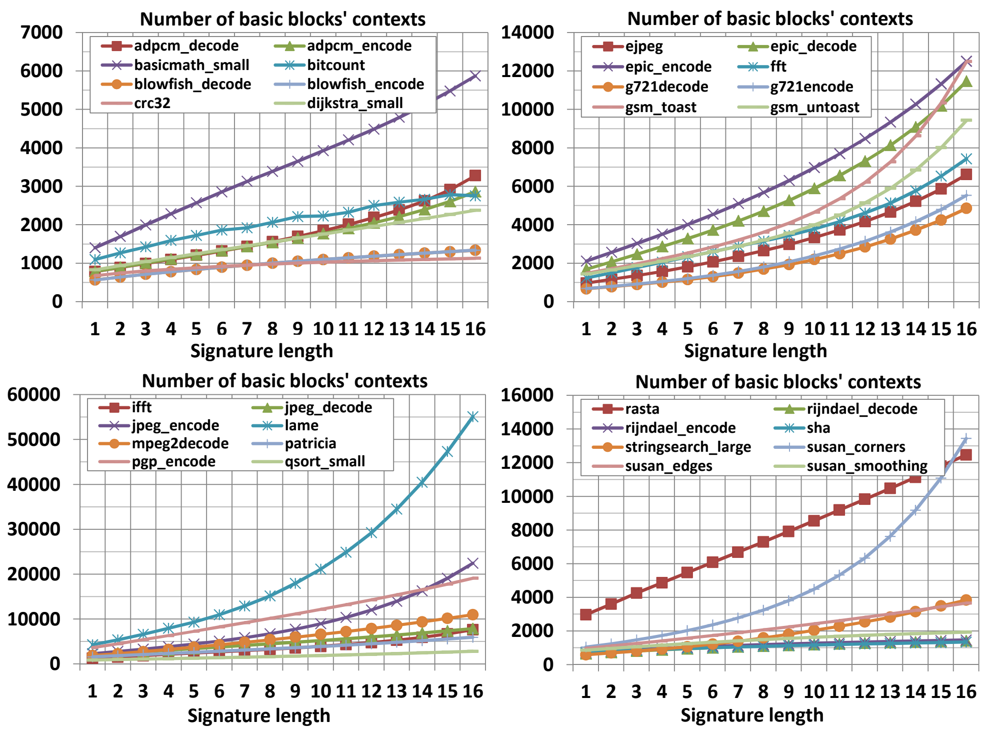


Figure 4.21: The total amount of contexts (and hence the total number of captured context-dependent basic block timings) for different benchmarks at various signature lengths (see Section 4.1.2 for the processor configuration used). The number of timings linearly increases with the signature length. For some benchmarks, e.g. *lame*, *jpeg_encode*, *susan_corners*, the increase has an exponential character.

The plot in Fig. 4.21 shows the total amount of contexts that can be differentiated at various signature lengths for different benchmarks[5] As can be seen from the results, the amount of contexts is highly benchmark dependent. For most benchmarks, the number increased linearly with the signature length. For these benchmarks, a rate at which the number of captured contexts increases with each signature length can be approximately estimated as:

$$r = \sqrt[15]{\frac{n_{16}}{n_1}}, \tag{3}$$

where n_1 and n_{16} are the number of contexts at signature length of 1 and 16 correspondingly. In the presented benchmarks, the value of r ranged from 1.034 (for *crc32*) to 1.136 (for *epic_decode*) with the mean value of 1.087.

5 The experimental results were obtained for standard input data provided with the benchmarks.

In turn, in benchmarks *lame*, *jpeg_encode*, *gsm_toast*, *gsm_untoast*, *fft*, *susan_corners*, *ejpeg*, *g721encode*,*g721decode* the increase in the amount of timings was close to exponential. Consequently, the size of the produced translated code significantly increased at larger signature lengths. A larger size of the translated code results in a larger compilation time and in lower overall performance of host-compiled simulation.

The problem with the code size can be solved with two different approaches. Firstly, the effective number of block timings can be reduced without sacrificing the simulation accuracy by optimizing the binary-to-C translation. This approach is discussed in Section 4.6.1. Secondly, timing of basic blocks can be obtained differently to achieve better accuracy at smaller signature lengths. This approach is discussed in Section 4.6.2.

4.6.1 *Optimization of binary-to-C translation*

The amount of block contexts that can be differentiated increases with the length of context signatures. Nevertheless, our preliminary experiments showed that the diversity of basic block timings does not increase at the same rate as the amount of contexts. In other words, many contexts of a basic block have an equal timing[6].

The similarity of timings can be exploited to reduce the size of the translated code. Particularly, the contexts with equal timings can be combined, leaving only *unique* timings and thereby reducing the size of annotated code. A unique timing can span one or multiple contexts of the basic block, and hence it can be tagged with multiple context signatures. For correct context-aware host-compiled simulation, the translated code must support multiple signatures per timing in basic block functions. An example of such code is shown in Listing 5. Here, each of the timings spans two basic block contexts. In the beginning of the block function, two comparisons with the history FIFO must be performed (function calls to `in_FIFO()`). If one of the signatures matches, the respective timing is simulated.

6 Timings of a basic block are equal if the execution latency of the block and time intervals between memory access events are identical.

Listing 5: Annotation of unique timings that span multiple contexts of the basic block

```
void b_0x700() {
   if (in_FIFO(0x100, 0x500) && in_FIFO(0x300, 0x500)) {
       // use timing 1
       reg[2] = r[5] + 16;
       cycle += 2;
       ...
   }
   else if (in_FIFO(0x200, 0x500) && in_FIFO(0x400, 0x500)) {
       // use timing 2
       reg[2] = r[5] + 16;
       cycle += 5;
       ...
   }
   ...
}
```

REDUCTION OF CONTEXT SIGNATURES With the unique timings, the size of the translated code can be reduced without sacrificing the accuracy of timing estimation. However, it became possible at the expense of additional comparisons with the history FIFO. If a unique timing spans N contexts and the signature length is L, $(N-1) \times L$ comparisons must be additionally performed. A problem occurs in a situation when the amount of signatures is large and many comparisons have to be made. If the size of the basic block function is relatively small, the comparison efforts may even dominate the effort required for executing basic block functions itself.

This problem can be solved by optimizing the structure of context signatures. In fact, not all signature elements must be evaluated to select an appropriate timing. Identification of a minimum set of sufficient elements is a challenging task. A simple and yet efficient algorithm is shown in Fig. 4.22a. The example shows three unique timings of a basic block with a respective signature. Each element in the signatures represents the start addresses of previously executed basic blocks. The goal is to identify the position of those elements, the evaluation of which is sufficient to differentiate these timings. In the first step, only *single* elements of the signatures are compared at a certain fixed position, e.g. only first elements. In the example in Fig. 4.22a, the second element is unique among all signatures. Hence, the evaluation of the second element is sufficient to identify the right context. As a result, the comparison efforts can be reduced by two thirds, compared to a case when all elements are compared.

Fig. 4.22b shows an example, in which the signatures cannot be differentiated by one element. In this case, the algorithm starts evaluating larger groups of elements. Grouping starts from the first element

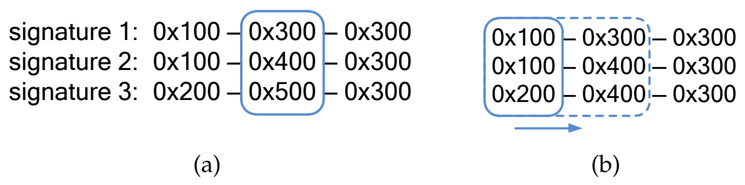

signature 1: 0x100 – 0x300 – 0x300
signature 2: 0x100 – 0x400 – 0x300
signature 3: 0x200 – 0x500 – 0x300
0x100 – 0x300 – 0x300
0x100 – 0x400 – 0x300
0x200 – 0x400 – 0x300
(a)
(b)

Figure 4.22: Optimization of context signatures: (a) evaluation of the middle element is sufficient to distinguish the signatures, (b) searching for a suitable size of the elements' group that is sufficient to distinguish the signatures.

(representing the oldest block in the execution history), since the last element (representing the most recent block in the execution history) is likely to be same in all signatures. The algorithm stops when the group is large enough to contain a sequence of elements, which is unique for each signature. In the presented example, a group of the first two elements is sufficient in order to differentiate the signatures. As a results, the resulting comparison effort is reduced by one third.

4.6.2 *Averaging of basic block timings*

As discussed previously, the length of context signatures determines the amount of contexts per basic block that can be differentiated in the measurement phase. However, the selected length may be not sufficient for capturing all possible timings that a basic block has. In this case, *multiple* timings can be captured per basic block context in the cycle-accurate simulator. Generally, any of these timings could be associated with the context. In Section 4.3.2.1, *first* observed timing was captured and annotated in the target code, while other timings were discarded. Inevitably, neglecting of other timings leads to an error. The value of this error can get significant if the captured timing is not typical for that context, i.e. it is observed not as frequently as compared to other timings. In order to reduce this error, the length of context signatures must be increased, which leads to an increased size of the translated code.

This section presents a different approach that allows achieving better accuracy of context-aware host-compiled simulation at smaller signature lengths. Particularly, instead of dropping other timings, multiple timings of a basic block observed in a certain context can be *averaged*. An example of such averaging is shown in Fig. 4.23. Execution latency t_{lat} of the block as well as the offsets of memory events t_i are averaged among all timings. To implement this, the timing values observed at the context are summed up and divided by the total amount of block executions at this context. Note that in host-compiled simulation, the simulated time is defined at the granularity of clock cycles. Therefore, the resulting mean values must be rounded to the next in-

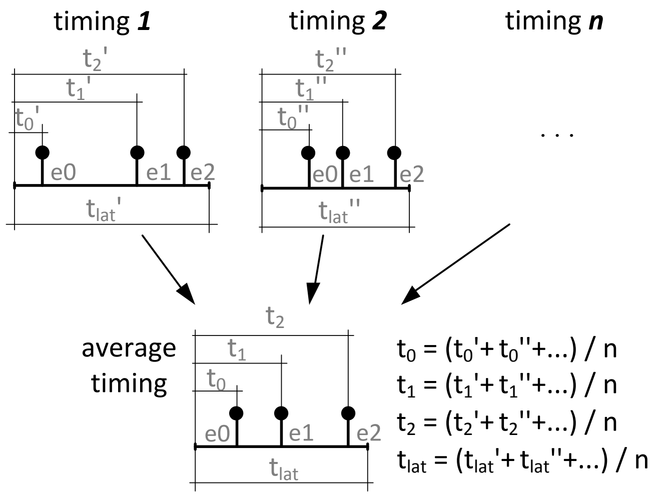


Figure 4.23: Averaging of basic block timings observed for the same context signature.

teger number. The resulting timing error for the complete target code due to rounding will be defined as:

$$e = \frac{\Delta T_e}{T} = \frac{\sum_{k=0}^{K} \sum_{j=0}^{M_k} \sum_{i=0}^{N_{jk}} \Delta_{kj}}{T}, \quad -0.5 < \Delta_{kj} \leqslant 0.5, \tag{4}$$

where T is the total execution time obtained without rounding, K is the number of basic blocks in the target code, M_k is the amount of contexts that can be captured for k^{th} block at the current signature length, N_{kj} is the number of executions of j^{th} context of k^{th} block, and Δ_{kj} is the rounding error of the block's execution latency at j^{th} context.

The averaging approximates the temporal properties of basic blocks at the given context signature length, thereby improving the overall accuracy of context-aware host-compiled simulation. The averaged timing is artificial as it may have never been observed during the timing measurements in the cycle-accurate simulator. The quantitative evaluation of this approach will be presented in Chapter 6.

4.6.3 *Static reordering of memory accesses*

Reordering of memory accesses is one of the key effects of out-of-order execution that has to be taken into account in host-compiled simulation. The order of memory accesses is particularly critical for the behavior of caches. A cache is a complex, state-based structure. Whether an access to the cache results in a hit or miss is determined by the current state of the cache, which, in turn, is defined by previous cache accesses. Therefore, the order of memory accesses produced by the processor model and later applied to the cache model needs to be accurately reconstructed.

Section 4.4.2 presented a general solution for reconstructing the order of memory accesses at simulation run-time based on a sorted queue. In this approach, the memory accesses were not simulated directly but temporarily stored in the queue first. The content of the queue was dynamically sorted to obtain the correct order of accesses. However, the manipulations on the data in the queue requires additional computational efforts. The quantitative estimation of this additional efforts will be provided in Chapter 6.

In the following, we discuss in detail when a wrong memory order may result in a timing error. Afterwards, an optimization method will be proposed.

4.6.3.1 *Timing error analysis*

As mentioned earlier, the timing behavior of data caches is sensitive to the order of memory accesses. However, not always will an error in the order result in a *visible* timing error in host-compiled simulation. The reason for this is the temporal locality of accessed data as well as the organization of data in lines. Let us assume a basic block with the execution timing shown in Fig. 4.24a. The basic block contains two *independent* memory instructions, which perform accesses A and B to the data cache. During the execution of this block in the cycle-accurate simulator (left part of Fig. 4.24a), access B is performed out-of-order with respect to A. Furthermore, let us assume that the both accesses request data which are located in the same cache line. If the line is present in the cache, both accesses result in a cache hit. If we now simulate the memory accesses in host-compiled simulation in their program order, i.e. A followed by B (the right part of Fig. 4.24a), the both accesses will similarly result in a cache hit. From the timing perspective, there will be no difference between these two simulation runs. Thus, given the assumptions above, the exchange of A and B does not produce a timing error.

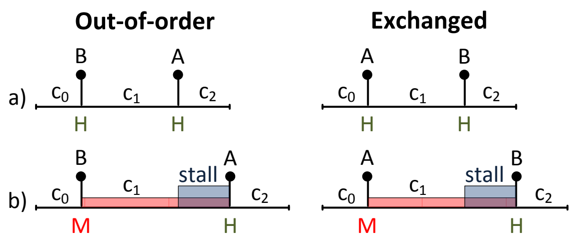


Figure 4.24: Simulation of a basic block with two independent accesses to the same line in the data cache: (a) the line is present in the cache, (b) the line is missing. In the both scenarios, the exchange of memory accesses does not result in a visible timing error.

If the requested line is not present in the data cache (Fig. 4.24b), access B causes a cache miss in the cycle-accurate simulation (left part of Fig. 4.24b). Access A requires the same cache line. Therefore, the execution is stalled on A till the missing data is arrived. Note that access A does not depend on B and hence, processing latency c_1 is masked by the cache miss latency. If we now exchange accesses A and B during host-compiled simulation (as shown in the right part of Fig. 4.24b), the miss will occur on A instead of B. However, the timing behavior of the model will not change because the execution will be similarly stalled on B and latency c_1 will be masked again (since B does not depend on A too). As a result, the exchange of two independent accesses to the same cache line in case of a miss will not produce a timing error as well.

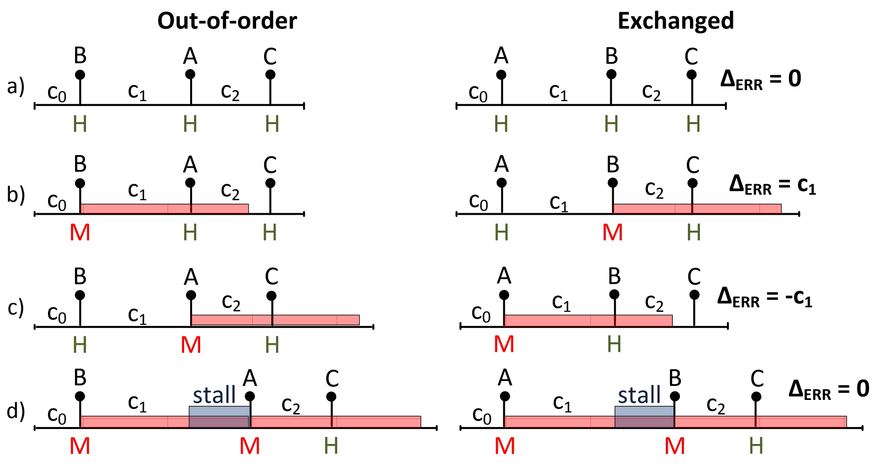


Figure 4.25: Simulation of three independent accesses A, B and C. The requested data for accesses A and B reside in different cache lines. Four scenarios are investigated: no cache misses on A and B, at least one miss either for A or B, and when both A and B result in a cache miss.

The simulation error may occur if the two consecutive cache accesses request data from different cache lines. In this case, the error will depend on third access C following A and B. In the following, we discuss two cases. In the first case, C does not depend on A and B. In the second case, C depends either on A or B. Furthermore, in all following figures, we similarly assume reference cycle-accurate simulation in the left part and host-compiled simulation with exchanged memory accesses on the right part.

C IS INDEPENDENT OF A AND B This case is shown in Fig. 4.25. Four possible scenarios must be considered with different combinations of hits and misses for A and B. In the first scenario (Fig. 4.25a), both A and B result in a cache hit. The host-compiled simulation of the same block with exchanged A and B (shown on the right side of the figure) will produce exactly the same result, i.e. there will be no timing error.

If either A or B generates a miss, a different timing behavior will be observed. Fig. 4.25b demonstrates a situation when B is a miss and A is a hit. If we exchange A and B, processing latency c_1 will not be masked by the cache miss latency. As a result, the execution time will be overestimated by the value of c_1, i.e. $\Delta_{err} = c_1$. A similar situation occurs when access B is a hit and A is a miss (Fig. 4.25c). In this case, the exchange will result in erroneous masking of latency c_1. As a result, the execution time will be underestimated by c_1. Finally, if both A and B result in a cache miss (Fig. 4.25d), the exchange will not result in an error because processing latencies c_1 and c_2 will be equally masked during both simulations.

C DEPENDS ON EITHER A OR B If access C depends on either A or B, we have to consider five possible scenarios shown in Fig. 4.26. In the first scenario in Fig. 4.26a, accesses A and B are hits. Here, the exchange of A and B will not produce a simulation error (similarly when C is independent). However, if either A or B is a miss, we will either overestimate or underestimate the execution time by the value of c_1. In the example shown in Fig. 4.26b, access B is a miss, access A is a hit and access C depends on B. During cycle-accurate simulation shown on the left side of the figure, latency c_1 is masked and access A is an out-of-order hit under the active miss. When the missing data is arrived, the model conservatively simulates latency c_2 once again (see Section 4.5.1.1 for further details) followed by access C. If we exchange A and B, latency c_1 won't be masked and the simulation will produce error $\Delta_{err} = c_1$. Similarly, if access C depends on the miss caused by A while B is a hit (Fig. 4.26c), exchanging of A and B will lead to error $\Delta_{err} = -c_1$, since latency c_1 will be erroneously masked.

Finally, let us consider two situations when both A and B result in a miss (Fig. 4.26d and Fig. 4.26e). If access C depends on B, latency c_2 is masked by the miss on A in cycle-accurate simulation (left part of Fig. 4.26d). However, if we exchange A and B, the model will erroneously simulate latency c_2 according to the conservative approximation made in Section 4.5.1.1. As a result, the total execution time will be overestimated by c_2. If access C depends on A (Fig. 4.26e), delay c_2 will be erroneously masked if we exchange A and B. Thus, the exchange will result in underestimation of execution time, thereby producing error $\Delta_{err} = -c_2$.

Summarizing the observations above, we can make the following conclusion. The exchange of simulation order for two subsequent memory accesses in compiled simulations does not always lead to a visible timing error. In fact, the error will occur only if the following conditions are met:

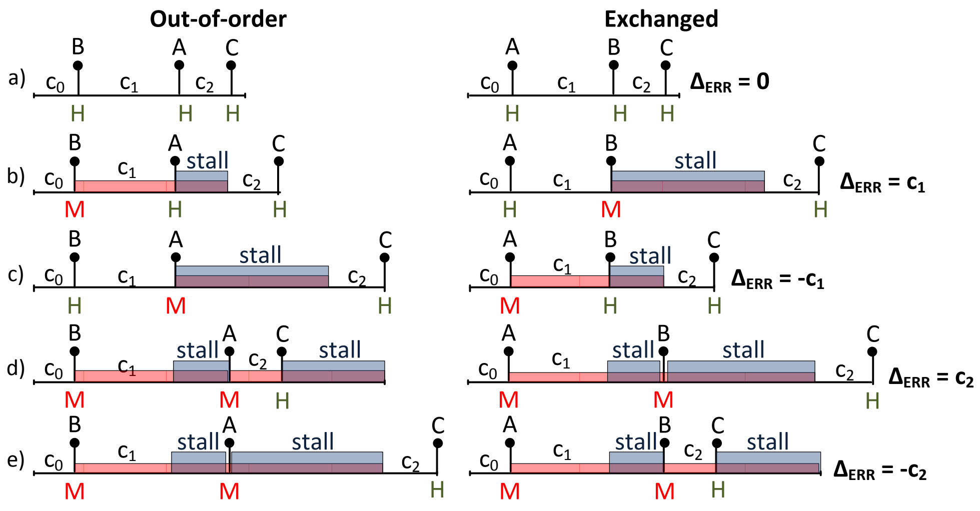


Figure 4.26: Simulation of a basic block with independent accesses A and B, followed by C which depends on either A or B. The target data of accesses A and B reside in different cache lines. Five scenarios are investigated: (a) A and B are hits, (b) B results in a miss and C depends on B (c) A results in a miss and C depends on A, (d) both A and B result in a miss, C depends on B, (e) both A and B result in a miss, C depends on A.

- The accessed data reside in different cache lines, and
- either one access results in a miss, while the other one results in a hit, or both cache accesses result in a miss, and
- the third following access depends on one of the misses.

4.6.3.2 *Static access reordering*

In the previous section, we saw that the exchange of cache accesses results in a visible timing error in particular cases only. If the error conditions occur infrequently, the efficiency of host-compiled simulation can be improved. Particularly, we can avoid computationally expensive, queue-based reconstruction of out-of-order memory accesses at a marginal loss of simulation accuracy.

The optimization method introduced in this section is based on a hypothesis that the *simulated* sequence of out-of-order cache accesses can be changed without introducing a large timing error. The idea behind this optimization technique is to simulate out-of-order data cache accesses in the program order, while leaving the processing latencies between them unchanged. For example, assume instructions of the target code shown in Fig. 4.27. When executed in the cycle-accurate simulator of the target out-of-order processor, these instructions produce a sequence of data cache accesses performed in intervals d_i. However, instead of queue-based recovery, the order of accesses can be changed back to their program order *prior to* host-

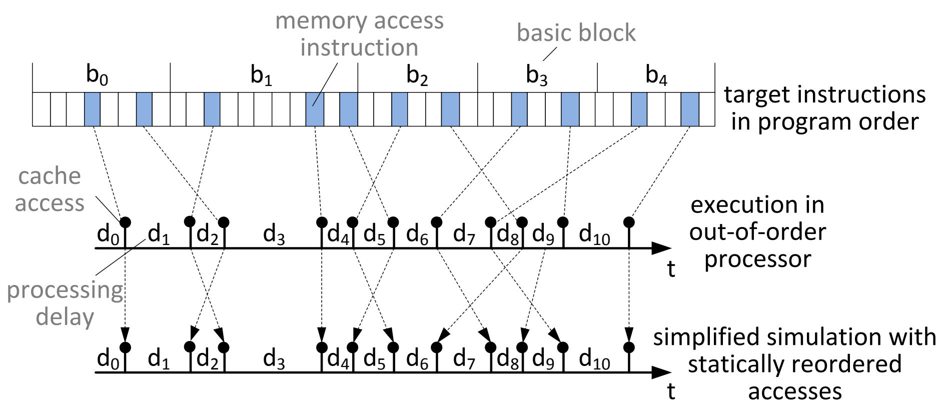

memory access instruction
basic block
b_0
b_1
b_2
b_3
b_4
target instructions in program order
cache access
d_0 d_1 d_2 d_3 d_4 d_5 d_6 d_7 d_8 d_9 d_{10}
t
execution in out-of-order processor
processing delay
d_0 d_1 d_2 d_3 d_4 d_5 d_6 d_7 d_8 d_9 d_{10}
t
simplified simulation with statically reordered accesses

Figure 4.27: Static reordering of out-of-order cache accesses prior to host-compiled simulation.

compiled simulation. At the same time, execution delays d_i are simulated as captured in the reference-cycle accurate simulation. By *static reordering* the accesses, we intentionally introduce an error in host-compiled simulation. However, as will be tested by the experiments later, the exchanged order results only in a small deviation of the produced timing estimate. Meanwhile, due to elimination of the queue-based reordering at simulation run-time, the overall simulation performance can be substantially improved. The quantitative estimation of the proposed improvement will be presented in Chapter 6.

4.6.3.3 *Derivation of block timing considering static reordering*

The assumption of static memory reordering requires new principles of deriving execution delays of basic blocks in the reference cycle-accurate simulator. In contrast to the derivation method introduced in Section 4.2, a new scheme is proposed for associating the execution delays under consideration of static reordering. The main difference of this scheme and the method presented in Section 4.2 is the interpretation of basic block overlapping.

As mentioned previously, when statically reordered, accesses to the data cache are simulated in their program order. Thus, there is no need for determining *offsets* of data cache accesses[7] in basic block timings as suggested in Section 4.3.2.1. Instead, in case of static reordering, it is sufficient to determine *execution delays* d_i between data cache accesses.

Furthermore, we can characterize the basic block timing to be annotated by a set of time constants. If i^{th} basic block of the target code

7 Recall that the offsets can be either positive or negative, since they are measured relative to the beginning of the basic block, which is also the end of the previous block.

contains n memory instructions, its timing can be specified by $(n+1)$ time constants as follows:

$$T_i = \{c_0^i,\ c_1^i,\ c_2^i,\ \dots,\ c_n^i\}, \qquad (5)$$

where c_k^i is a kth time constant of ith block and n is the number of memory instructions in the respective basic block.

In the following explanations, I differentiate between these time constants c_i to be annotated in the translated code and actual execution delays d_i between data cache accesses observed in the cycle-accurate simulator. In fact, execution delays may involve execution of multiple overlapping basic blocks. Therefore, it is essential to decide which basic blocks these execution delays d_i must be assigned to.

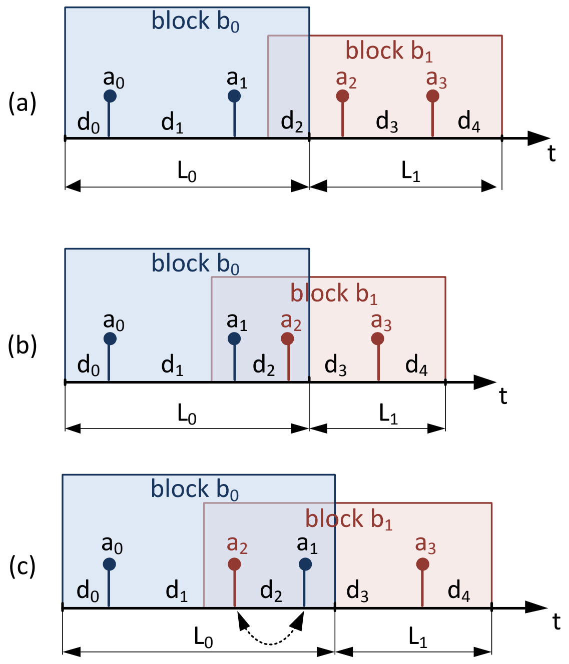


Figure 4.28: Different types of basic block overlapping under consideration of static reordering of data cache accesses: (a) partial overlapping within execution delay d_2; (b) complete overlapping of execution delay d_2; (c) deep overlapping of basic blocks resulting in reordering of data cache accesses.

OVERLAPPING OF BASIC BLOCKS Consider the execution of two basic blocks b_0 and b_1 in the cycle-accurate simulator (Fig. 4.28). Each block performs two accesses to the data cache: accesses a_0 and a_1 (by block b_0) and accesses a_2 and a_3 (by block b_1). Execution delays d_0–d_4 represent the latencies observed between accesses a_i. The execution latencies of the basic blocks are denoted as L_i. They are determined according to the rule defined in Section 4.3.2.1. In the following, we discuss three types of possible overlapping of the basic blocks.

In the first type (Fig. 4.28a), the execution of basic blocks in the cycle-accurate simulator is overlapped. Nevertheless, data cache accesses are still performed in the program order. Here, execution delay d_2 is *partially* shared between the two blocks and, hence, it can be assigned to any of these two blocks. As a matter of convention, I assume that the shared delay is always assigned to the preceding block. As a result, the timing of basic blocks will be defined as:

$$T_0 = \{c_0^0,\ c_1^0,\ c_2^0\} = \{d_0,\ d_1,\ L_0 - (d_0 + d_1)\}, \tag{6}$$

$$T_1 = \{c_0^1,\ c_1^1,\ c_2^1\} = \{d_2 - c_2^0,\ d_3,\ d_4\}, \tag{7}$$

where L_0 is the execution latency of basic block b_0.

In the second type (Fig. 4.28b), the execution of blocks b_0 and b_1 is overlapped and access a_2 is performed while block b_0 is executing. Accesses a_1 and a_2 are still performed in the program order. In this scenario, entire delay d_2 is shared by the blocks. According to the convention made above, d_2 is assigned to block b_0. However, since this delay is considered in block b_0, the respective time constant in the timing of block b_1 must be set to zero in order to avoid double inclusion. Thus, in this case the observed execution delays will be distributed over the blocks as follows:

$$T_0 = \{c_0^0,\ c_1^0,\ c_2^0\} = \{d_0,\ d_1,\ d_2\}, \tag{8}$$

$$T_1 = \{c_0^1,\ c_1^1,\ c_2^1\} = \{0,\ d_3,\ d_4\}. \tag{9}$$

In the last type (Fig. 4.28c), the execution of basic blocks is overlapped such that one or multiple accesses to the data cache are performed out-of-order. In this case, we statically exchange out-of-order accesses and simulate them in their program order, i.e. a_1 is simulated in place of a_2 and vice versa. After exchanging, this scenario converges to the second type of basic block overlapping and the timing of basic blocks is defined by Eq. (8) and (9).

BLOCKS WITHOUT MEMORY INSTRUCTIONS A special case of basic block overlapping occurs when one of the blocks does not contain memory instructions and, hence, no data cache accesses are performed during its execution. For example, assume three basic blocks b_0, b_1 and b_3 shown in Fig. 4.29. Each of blocks b_0 and b_2 contains two memory instructions, resulting in data cache accesses a_0, a_1, a_2 and a_3. Basic block b_1 does not contain memory instructions. In the following, we discuss four possible scenarios of overlapped execution of block b_1.

In the first scenario (Fig. 4.29a), the execution of block b_1 does not overlap with any data cache accesses from the neighboring blocks. Execution delay d_2 is shared by the three blocks. I propose a "greedy" assignment of d_2 among the three blocks starting from b_0. Particularly, block b_0 is assigned the part of d_2 which fits into execution latency L_0. In this case, the timing of block b_0 will be defined as:

$$T_0 = \{c_0^0,\ c_1^0,\ c_2^0\} = \{d_0,\ d_1,\ L_0 - d_0 - d_1\}. \tag{10}$$

In turn, block b_1 is assigned the next part of d_2 which fits into latency L_1. Thus, the timing of block b_1 will contain only one time constant equal to the block's latency:

$$T_1 = \{c_0^1\} = \{L_1\}. \tag{11}$$

Finally, block b_2 is assigned the remaining part of d_2 left after the previous assignments. The timing of block b_2 will be defined as:

$$T_2 = \{c_0^2,\ c_1^2,\ c_2^2\} = \{d_2 - L_1 - c_2^0,\ d_3,\ d_4\}. \tag{12}$$

In the second scenario (Fig. 4.29b), the execution of block b_1 is overlapped with a data cache access from the preceding block (access a_1). According to the greedy policy made above, the assignment execution delays starts from the preceding block. Therefore, the timing of the three blocks in this scenario will be same as in the first scenario, i.e. defined by Eq. (10)–(12).

In the third scenario (Fig. 4.29c), the execution of block b_1 overlaps with a cache access a_2 belonging to succeeding block b_2. Similarly to the previous scenarios, block b_0 will be assigned the first part of d_2 which fits to latency L_0. However, in this scenario, the greedy assignment cannot be applied to block b_2. Otherwise, the simulation would produce a positive timing error $e = L_0 + L_1 - (d_0 + d_1 + d_2)$ (denoted in Fig. 4.29c). This error can be omitted by *masking* latency L_1 by block b_2. Thus, timings for block b_0, b_1 and b_2 will be defined as:

$$T_0 = \{c_0^0,\ c_1^0,\ c_2^0\} = \{d_0,\ d_1,\ L_0 - d_0 - d_1\}, \tag{13}$$

$$T_1 = \{c_0^1\} = \{0\}, \tag{14}$$

$$T_2 = \{c_0^2,\ c_1^2,\ c_2^2\} = \{d_2 - c_2^0,\ d_3,\ d_4\}. \tag{15}$$

Finally, in the last scenario (Fig. 4.29d), the execution of block b_1 is overlapped with the cache accesses of the preceding and succeeding blocks (accesses a_1 and a_2). With respect to block b_0, this scenario is similar to all previous scenarios. By employing the greedy assignment, the timing of block b_0 will be defined by Eq. (10) or Eq. (13). In turn, with respect to block b_1 and b_2. this scenario is similar to the third scenario shown in Fig. 4.29c, i.e. the timings of blocks b_1 and b_2 will be defined by Eq. (14) and Eq. (15) respectively.

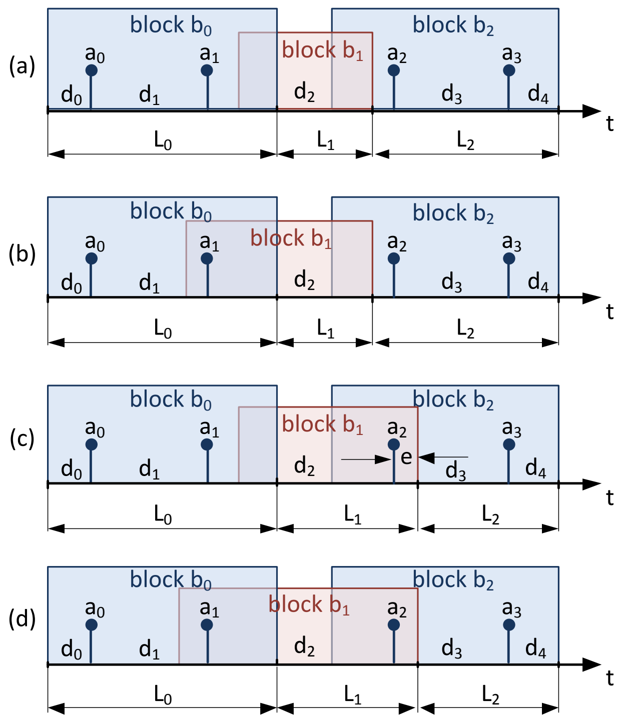


Figure 4.29: Overlapped execution of a basic block which does not contain memory instructions (block b_1): (a) execution of b_1 is not overlapped with any data cache accesses; (b) execution of b_1 is overlapped with a cache access belonging preceding block b_0; (c) execution of b_1 is overlapped with a cache access belonging to succeeding block b_2; (d) execution of b_1 is overlapped with cache accesses belonging to both preceding and succeeding blocks.

SETUP FOR TIMING DERIVATION During the derivation of block timing under consideration of statically reordered cache accesses, the same setup of cycle-accurate simulator can be used as described in Section 4.3.2.1. The only difference is the fact that the resulting timing of basic blocks contain *values* determined using Eq. (6)–(15) and not *offsets* as was assumed in Section 4.3.2.1.

The setup of the reference cycle-accurate simulator for deriving basic block timings under consideration of statically reordered access is shown in Fig. 4.30. The monitoring unit captures the timing values according to Eq. (6)–(15). For each basic block, multiple timings may be captured per context. If averaging of block timing is enabled, time constants c_i to be annotated in the code are averaged using the following formula:

$$c_k^j = \left[\frac{\sum_{i=0}^{n_j} c_{ki}^j}{n_j} \right], \qquad (16)$$

where c_k^j is a kth element in the set of time constants for block b_j as defined in Eq. (5), c_{ki}^j is the time constant derived at the ith execution of block b_j, n_j is the amount of block executions and [ ] is the

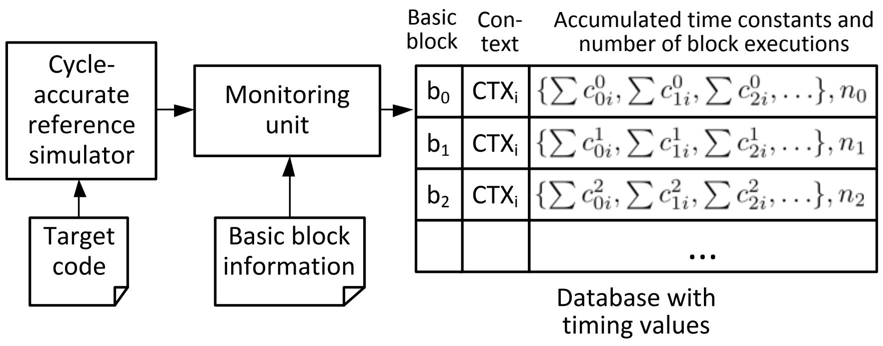

Cycle-accurate reference simulator
Target code
Monitoring unit
Basic block information
Basic block
Con-text
Accumulated time constants and number of block executions
b_0 CTX_i $\{\sum c^0_{0i}, \sum c^0_{1i}, \sum c^0_{2i}, \ldots\}, n_0$
b_1 CTX_i $\{\sum c^1_{0i}, \sum c^1_{1i}, \sum c^1_{2i}, \ldots\}, n_1$
b_2 CTX_i $\{\sum c^2_{0i}, \sum c^2_{1i}, \sum c^2_{2i}, \ldots\}, n_2$
...
Database with timing values

Figure 4.30: Setup for deriving basic block timing with statically reordered accesses.

rounding operation to the nearest integer value. In the final step, the calculated time constants are annotated in the respective basic block functions in the translated target code.

5 SYSTEM-LEVEL SIMULATION OF MULTICORE ARCHITECTURES

So far I have discussed efficient timing simulation of a single out-of-order core by means of host-compiled simulation. In this chapter, I will address a broader scope and address simulation of a complete system-on-chip incorporating multiple processing cores. The goals of system-level performance simulation are manifold. For example, it allows for studying the implication of shared hardware resources on the system performance or identification of bottlenecks in the system architecture. In addition, system-level simulation can be employed for rapid evaluation of different options for mapping and scheduling of the target software on the underlying processing elements.

In the following, I will focus on system-level simulation methods and present implementation of high-level performance models of major SoC components, including out-of-order processing cores, arbitrated shared bus and main memory. The models are incorporated into a simulation tool that has been developed in the scope of this dissertation. The simulation tool is written using SystemC [22], a system-level modeling language based on C++ that allows creating an executable model of a complete system-on-chip. In addition to the models of HW components, the tool includes an abstract model of a scheduler (see Fig. 5.1) that manages the simulation of target software on the core models.

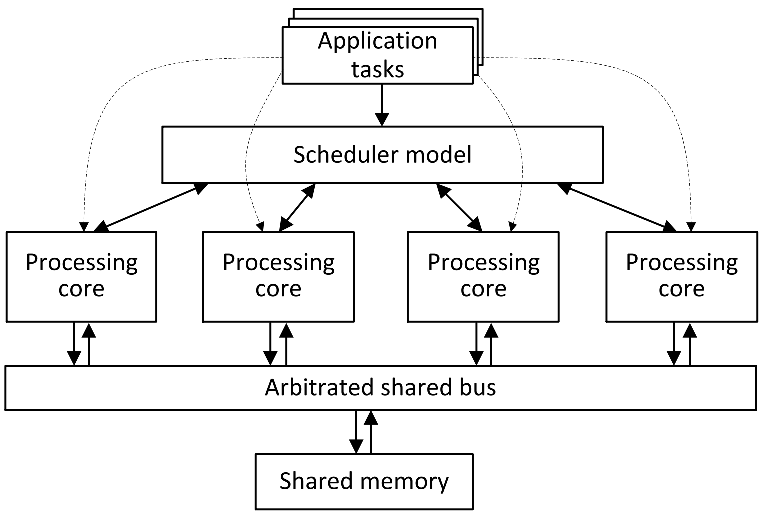


Figure 5.1: An example of a system-on-chip architecture which can be modeled and simulated in the SystemC-based tool.

5.1 SIMULATION METHODS

In this section, I discuss and compare two different simulation techniques employed in the SystemC tool: *trace-driven simulation* (TDS) and host-compiled *binary-level simulation* (BLS). Both of them can be applied for fast and accurate performance simulation of a complete MPSoC, as they abstract the details of the core's microarchitecture. However, they are fundamentally different in the way of representing the execution of target software.

5.1.1 *Trace-Driven Simulation*

In trace-driven simulation, the execution of target software on processing cores is represented in the form of abstract traces. In a general form, a trace contains a sequence of events obtained prior to system-level simulation, e.g. by means of a cycle-accurate simulator. Traces can reflect the software execution at different abstraction levels. At the highest level, a trace captures accesses which a processing core performs on the interconnect during the program execution, i.e. refills/evictions of cache lines, and execution delays between them.

Listing 6: Abstract trace captured at the output interface of a processing core

```
DELAY 3   # processing latency of 3 cycles
READ  32  # read request for 32 bytes
DELAY 17
READ  32
WRITE 32  # write request for 32 bytes
```

An example of a trace is shown in Listing 6. It is a list of commands or so called *trace primitives* stored in a file. Each primitive specifies the type of a system-level event as well as its parameters. For example, `READ` and `WRITE` primitives represent communication requests made by the core on the on-chip interconnect. The argument of these primitives defines the amount of transferred data. In our example, the transferred data are 32-Byte cache lines being read or written to the main memory. `DELAY` primitives represent internal processing latencies in the core. The argument of `DELAY` primitives is the number of clock cycles between the adjacent communication requests. Note that represents only performance characteristics of the target software. The functionality of the target code is abstracted.

The trace shown in Listing 6 completely abstracts the core's microarchitecture including local caches. By simulating this trace, the timing behavior of the core can be accurately reconstructed at its bus interface without modeling any of the core's internal components. As a result, the overall simulation performance can be significantly improved compared to a detailed cycle-accurate simulator of the core.

5.1.1.1 *Workflow of TDS*

The workflow of trace-driven simulation consists of two stages. In the first stage, the target code is executed on a reference implementation of the processing core to generate an abstract execution trace. The reference implementation defines the accuracy of generated traces. Generally, for this purpose the designer may employ either an existing hardware implementation, or an HDL model or a cycle-accurate simulator of the target core. One of the most critical factors when choosing a reference implementation (in addition to the availability of the respective prototypes/models) is the time required to generate a trace. For example, simulation of a very detailed gate-level HDL model of the core may require a significant amount of time in order to generate a complete execution trace. In this work, I employ cycle-accurate instruction set simulators to generate abstract traces. These simulators have adequate accuracy for early system-level design space exploration and offer sufficient simulation speed to derive complete execution traces in reasonable time.

In the second stage, generated traces are simulated in a new multicore environment using abstracted models of system-on-chip components. By means of the simulation, the designer can investigate interaction of multiple cores and their contention for the shared resources. The key advantage of TDS is the possibility to *reuse* traces during multiple simulation runs, enabling faster investigation of possible design solutions. Note that the generation of traces takes at least as much time as the complete execution of the code on the cycle-accurate simulator. However, the generation is performed only once, assuming that the target software and input data do not change[1]. In this case, the overhead of trace generation is spread over multiple simulation runs. In the following sections, I provide further details on each of the two stages of trace-driven simulation.

GENERATION OF TRACES Generation of traces by means of a cycle-accurate instruction set simulator is shown in Fig. 5.2. The code of the simulator must be correspondingly extended to enable trace generation. The efforts required for the modification are marginal. In fact, during the program execution, the trace generation unit has to capture a system-level memory access (e.g. cache line eviction) and its time stamp. The time stamps are then used to calculate processing latencies between two consecutive memory accesses.

The result of the trace generation is a file which contains trace primitives captured during the execution of the target software. The format of the primitives can be chosen freely as long as it matches the core model which is going to interpret this trace. For example, the trace primitives can be stored in a text file using the format shown

1 If the target software changes or input data change, abstract traces must be generated once again.

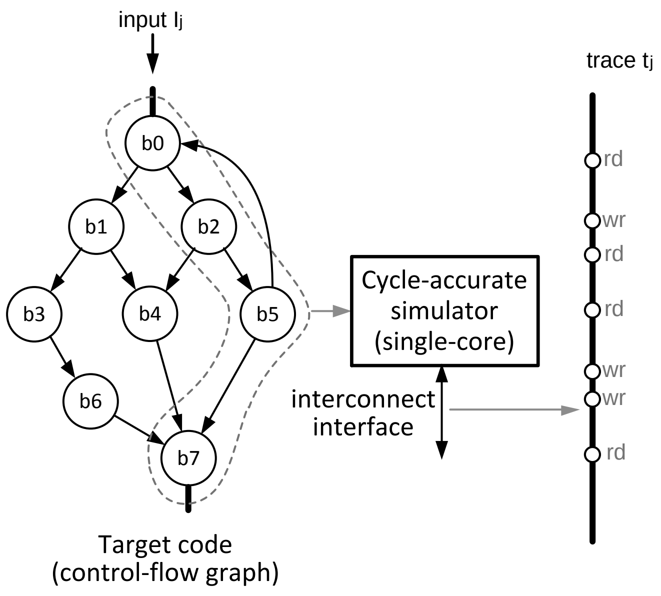


Figure 5.2: Generation of an abstract trace

in Listing 6. However, large traces can be processed more efficiently if they are stored in a binary form. In this case, the size of trace files as well as the time required to parse the primitives during a trace simulation can be significantly reduced.

The generated trace represents a complete execution of the target program. Therefore, the size of the trace is proportional to the execution time of the code. Because of this fact, the use of TDS may be restricted if the target application produces a very large workload and, therefore, requires a large space on the hard disk for storing the trace. Furthermore, there may be multiple execution paths in the control flow graph of the target code (see Fig. 5.2). Each execution path is determined by the results of branch instructions, which in turn depend on the input data. Thus, the trace represents only *one* of possible execution paths, corresponding to a specific input applied during the trace generation.

SIMULATION OF MULTICORE Traces are generated on a simulator in the context of a single core and then simulated to predict the performance of target software in a new multicore environment. The simulation of traces in a system-on-chip model consisting of abstracted processing cores, on-chip interconnect and shared memory is shown in Fig. 5.3.

The processing cores are modeled as black boxes, which reconstruct the core's timing behavior by reading and interpreting the assigned traces. In case of `DELAY` primitives, the core models just wait for the specified amount of cycles, thereby simulating internal processing of

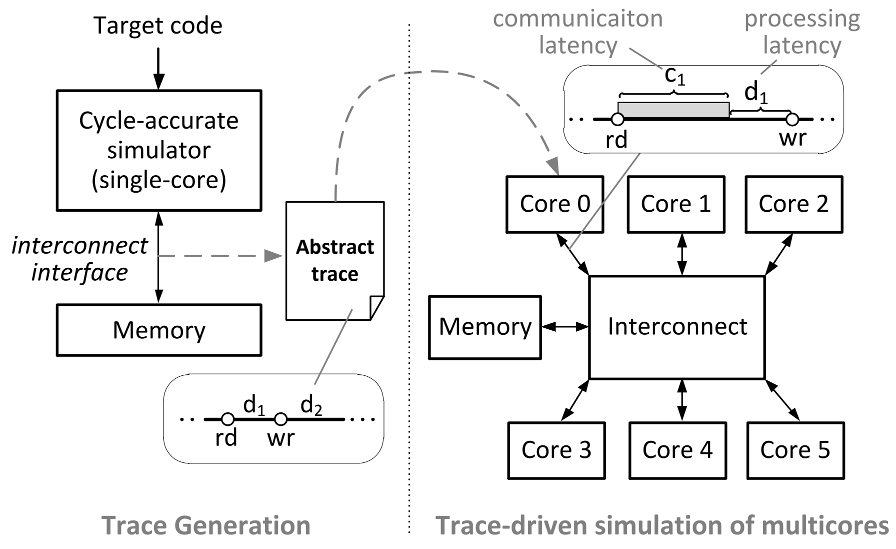


Figure 5.3: Employment of abstract traces for performance simulation of multicore architectures.

data. In case of `READ` or `WRITE` primitives, the models make communication requests to the shared on-chip interconnect model.

Performance evaluation of the complete system-on-chip is performed by superposed simulation of multiple traces on the common interconnect. Contrary to the processing latencies, which are explicitly defined in the traces, communication latencies c_i are determined dynamically at simulation run-time (Fig. 5.3). Communication latencies consist of the memory access latency as well as dynamic arbitration delays which appear if multiple cores want to simultaneously access the shared interconnect. The overall performance of the system-on-chip is evaluated by combining processing latencies and communication latencies determined during the simulation. Please note that the communication latencies are determined solely in the new multicore environment. Therefore, it is important to assure that the processing latencies contained in traces do not depend on the communication latencies in the reference cycle-accurate simulator. This can be achieved by setting communication latencies to zero during the trace generation.

5.1.1.2 *Trace modifications*

Representation of software execution in the form of abstract traces allows for flexible modification of the workload imposed by the target software. This can be particularly useful for investigation of functional repartitioning between processing cores and hardware peripherals. To enable this, a part of the trace representing a certain function may be replaced by new primitives. These primitives can be used to simulate accesses to a peripheral model that is supposed to implement this function in hardware. Since the trace simulation does not involve processing of real data, the peripheral model can be abstracted to pre-configured processing latencies. In this case, the designer can still evaluate the impact of hardware acceleration on the software execution time. Moreover, if the peripheral is shared by multiple cores, the designer can additionally investigate the additional latencies caused by the contentions for the shared resource.

Abstract traces can be also augmented with additional primitives for code profiling. The purpose of these primitives is to associate sections of the trace with respective functions of the target code. Particularly, profiling primitives denote points in the trace, at which functions start or stop their execution. The boundaries of the target functions can be obtained from the debugging information of the target binary code. Given the function boundaries, the trace generation unit can determine which function is currently being executed and insert the profiling primitives in the generated trace.

5.1.1.3 *Various abstraction levels*

In Section 5.1.1.1, we introduced traces in the most abstract form. At this level of abstraction, the behavior of local caches is completely hidden in the processing latencies, and the traces capture only memory accesses caused by cache misses. If such traces are simulated in a new environment, it is assumed the local caches of the core have the same behavior as during the trace generation.

In some scenarios, the designer may want to involve local caches into the design space exploration process, e.g. to study the impact of cache parameters in the scope of multicore system-on-chip. In this case, local cache have to be modeled and simulated in the multicore environment. Moreover, simulation of caches is inevitable for considering the effects of cache coherency in the multicore environment. In coherent caches, certain lines may be invalidated by other cores. Consequently, a cache access resulted in a hit during the trace generation may result a miss in a multicore environment, producing an additional communication request on the shared interconnect.

In order to enable simulation of caches, traces have to be defined at finer granularity than the one assumed in Section 5.1.1.1. Particularly, traces have to capture memory accesses before the local caches as shown in Fig. 5.4. Afterwards, in the simulation phase, the captured accesses can stimulate the cache models in order to determine hit or miss events at run-time. The cache models do not need to hold real data since the functionality of the target software is abstracted in TDS. In fact, it is sufficient to store only the address tags of cache lines in order to determine whether an access results in a hit or a miss. The arguments of trace primitives have to be refined as well as shown in Fig. 5.4. In contrast to a highly abstracted trace (left part), in which `READ` and `WRITE` primitives specify amount of data to be transferred,

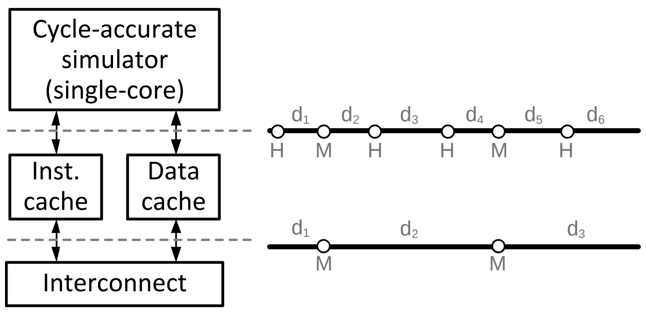

Cycle-accurate simulator (single-core)
Inst. cache
Data cache
Interconnect
d_1 d_2 d_3 d_4 d_5 d_6
H M H H M H
d_1 d_2 d_3
M M

Figure 5.4: Different abstraction levels of captured traces. The lower trace captures cache misses only, while the upper trace has higher granularity and captures all cache accesses.

the refined trace (right part) must contain target memory addresses that must be supplied to the cache models.

```
DELAY   3
READ   32
DELAY  17
READ   32
DELAY  11
```

```
WRITE  0xfffff200
DELAY  3
READ   0xfffff280
DELAY  5
READ   0xfffff284
DELAY  7
READ   0xfffff288
DELAY  5
READ   0x40000100
DELAY  6
WRITE  0x40000108
DELAY  5
```

Figure 5.5: Different abstraction levels of traces.

Fine-grained traces capture a larger amount of events. As a result, they are significantly larger in size compared to abstract traces. Processing of larger files and simulation of local caches inevitably slows down the TDS performance. Therefore, the level of trace abstraction has to be considered carefully depending on the goals and the context of system-level design space exploration.

5.1.2 *Binary-level simulation*

In contrast to trace-driven simulation, the simulation of software execution in BLS is based directly on the target code. Therefore, BLS is *execution-driven* simulation. During the simulation, the target code (or more precisely its equivalent C-representation) is co-executed with the performance models of system-on-chip components. The simulation of the target code in BLS is fully functional at the instruction level. At the same time, the internal core's components—which are required for processing the instructions in the real processor, e.g. the instruction pipeline or execution units—can be completely abstracted. In BLS, the execution time of instructions is considered by pre-annotating timing information in the code. The key aspects of the BLS approach were discussed in detail in Chapters 3 and 4 (in the scope of a single core). This section is primarily focused on the employment of BLS for simulation of multicore architectures.

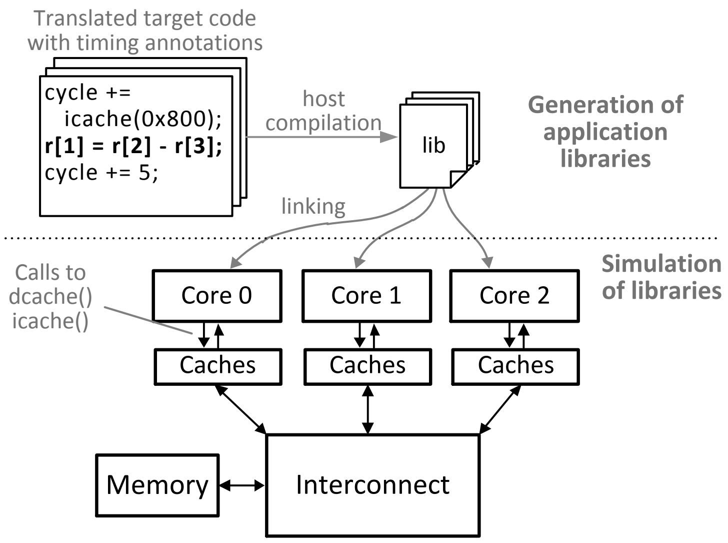


Figure 5.6: Employment of BLS for simulation of multicore architectures.

SIMULATION OF MULTICORE Simulation of multicore architectures based on BLS is shown in Fig. 5.6. To enable multicore BLS, the annotated translated code of each target application has to be compiled as a library. In the simulation phase, the libraries are dynamically linked to the core models, where the basic block functions of the translated code are executed. In addition, each abstracted core is attached to the models of local instruction/data caches. The cache models are accessed by the compiled target code via `icache()` and `dcache()` functions. If the core components have to simulate the out-of-order, additional modeling is required. The structure and implementation of out-of-order core models will be presented in Section 5.3.

Given the target memory address, the cache models dynamically determine whether a current memory operation results in a hit or a miss. In case of a miss, the cache models initiate a communication request to the model of the shared on-chip interconnect. The interconnect model arbitrates requests of multiple cores. The resulting arbitration latencies are added to the access latencies of the main memory resulting in the total communication latency for each request. The models of the interconnect and memory are not functional. Therefore, real data do not have to be transferred and stored in the memory model.

The actual data needed for functional execution are stored internally in the local arrays of the translated code as described in Section 3.1.1.

Note that the abstraction of functional data in the cache models does not allow investigation of race conditions in MPSoC. A race condition is an undesired situation when two or more execution threads access shared data in an uncontrolled way due to improper synchronization. As a result, a thread may retrieve wrong data from the main memory and the application behavior may change in this case. During simulation of multicore architectures, parallel threads have to be properly synchronized, e.g. using synchronization mechanisms of a run-time system (see Section 5.2.2.3 as an example). In the proposed approach, the actual data is acquired directly from the functional model of the target memory. If the identification of race conditions on the interconnect is desired, the models of caches and on-chip communication have to be correspondingly enhanced for considering functional data. The associated modification efforts are marginal.

5.1.3 *Summary*

In the sections above, I discussed the employment of BLS and TDS for simulation of multicore system-on-chip architectures. Both simulation methods are similar to a certain extent:

1. They abstract the details of instruction execution in the core's microarchitecture to latencies. TDS offers the highest abstraction level, at which the complete microarchitecture can be hidden including caches. BLS similarly abstracts the execution of instruction. However, it still requires co-simulation of caches.

2. The workflow of the both techniques consists of two separate phases: derivation of processing latencies in the reference cycle-accurate simulator and further simulation of the latencies in a new multicore environment. In the simulation phase, the target software (represented as traces in TDS or translated code in BLS) accesses the interfaces of the abstract core model to stimulate the models of on-chip interconnect and memory in order to obtain communication latencies at simulation run-time.

3. Both TDS and BLS are processor- and application-specific as they reconstruct the execution of a particular target code on a particular processing core. If the target code or the configuration of the processing core changes, the abstract trace has to be generated once again. Similarly, the translated target code has to be re-annotated and re-compiled.

TDS and BLS have a number of differences as well. Monolithic traces required by TDS capture the complete execution of programs. In traces, the control flow of the target program is fixed and cannot be changed at simulation run-time. Thus, in addition, traces are specific to input data. In order to simulate the target code, which is applied a different input, a new trace must be generated. In turn, the control flow in BLS is determined at run-time and can be changed during simulation. In this case, the derived timing of basic blocks can be reused among multiple simulations with different input data.

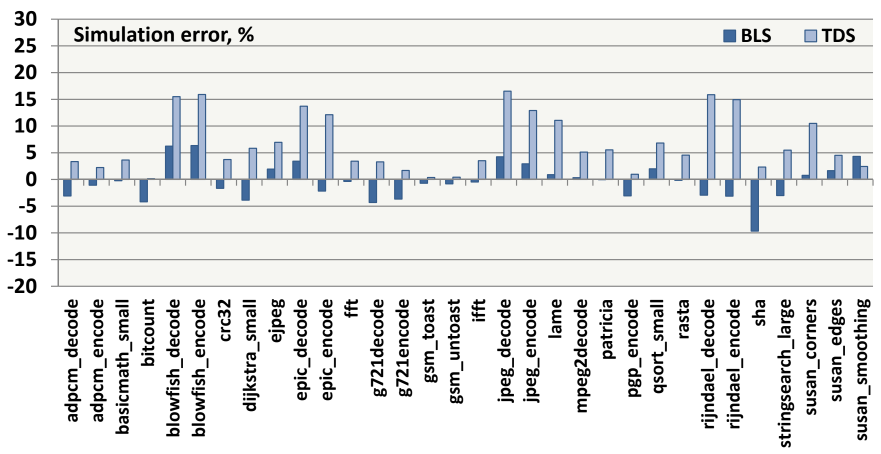
Monolithic traces completely abstract the functionality of the target code. This fact leads to certain restrictions in simulation of complex applications, which may consist of multiple communicating tasks. For this type of applications, the designer may want to generate separate traces for each task and simulate them independently in a multicore architecture. However, due to the abstraction of functionality, only very simple synchronization between traces is possible. We will discuss synchronization mechanisms for traces in detail in Section 5.2. In contrast, BLS is a functional simulation and, therefore, it allows for more complex, data-dependent synchronization of multiple tasks.

Finally, TDS enables faster performance estimation compared to BLS, since monolithic traces do not require co-simulation of caches and co-execution of the target code. However, due to the complete abstraction of the microarchitecture, memory accesses in TDS are always considered to be blocking. This simplification results in overestimation of software execution time in case of out-of-order cores. In contrast, BLS can capture dynamic out-of-order effects (see Chapter 4) and, therefore, it can achieve better accuracy than TDS.

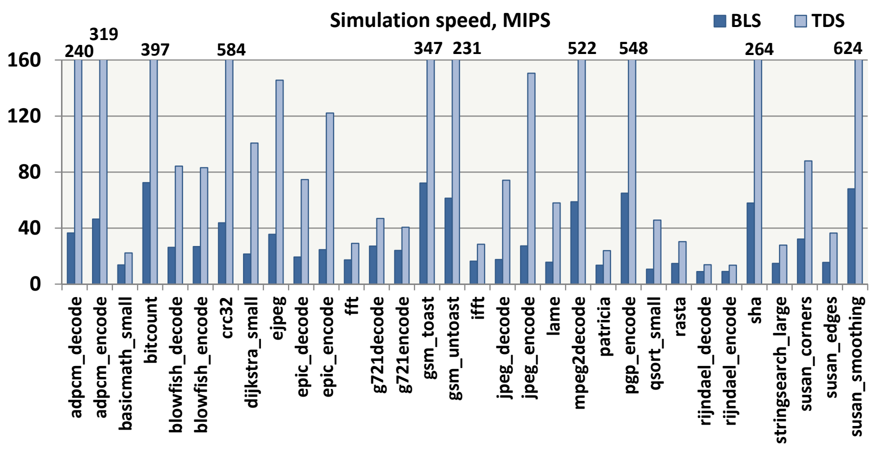
I will thoroughly evaluate TDS and BLS techniques in terms of performance and accuracy in Chapter 6. Nevertheless, some of the key numbers for BLS and TDS are highlighted in Fig. 5.7. The figure shows the results of abstracted system-level simulations performed for different benchmarks using TDS and BLS. For both methods I used *sim-outorder*[2] simulator from SimpleScalar to derive timing information of basic blocks for BLS and to generate abstract traces for TDS. The goal of TDS and BLS was to reproduce the timing behavior of *sim-outorder* as fast and accurate as possible.

Fig. 5.7a shows the error of TDS and BLS in estimating the execution time compared to the setup simulated in *sim-outorder* (single-core). The performance numbers for TDS and BLS are presented in Fig. 5.7a and expressed in millions of simulated instructions per second (MIPS). As can be seen from the results, the simulation accuracy and speed are not permanent but rather benchmark-dependent. Moreover, the experiments showed that none of BLS and TDS suits best for system-level simulation. For some benchmarks, e.g. *bitcount*, *crc32*, *gsm_toast*, *gsm_untoast*, *mpeg2decode*, *pgp_encode*, *susan_smoothing*, TDS showed almost the same accuracy as BLS (or even slightly better

2 During the experiments, the same tool configuration was used as described in Section 4.1.2. Both TDS and BLS were performed in SystemC environment. Further details on the SystemC models employed for the simulation will be presented in Section 5.3.



(a)



(b)

Figure 5.7: Comparison of TDS and BLS methods for different benchmarks executed in an out-of-order core: (a) timing error compared to the reference cycle-accurate simulator SimpleScalar; (b) simulation speed expressed in millions of simulated instructions per second.

for *sha*) but achieves significantly better performance than BLS. At the same time, for other benchmarks, e.g. *lame*, *blowfish*, *epic*, *jpeg*, *rijndael* (both encoding and decoding parts), BLS achieves significantly better accuracy compared to TDS at the same performance level. We will discuss in detail why TDS and BLS exhibit this behavior for some benchmarks in Chapter 6.

None of these simulation technique achieves the best trade-off between accuracy and performance in all benchmarks. Efficient design space exploration requires consideration of the both methods. Therefore, the SystemC simulation tool developed in the scope of this dissertation supports both methods.

5.2 HIGH-LEVEL SCHEDULER MODEL

The workload of embedded systems-on-chip typically consists of multiple concurrent applications. For example, recent smartphones simultaneously process the communication protocol stack, provide the graphical user interface and run several user applications, e.g. an e-mail client, web browser or MP3-player. A common approach of managing the execution of multiple applications is to employ an operating system. The purpose of operating systems is to provide necessary services to the applications and schedule their execution on the platform in order to efficiently utilize the available hardware resources. Target applications, in turn, may consist of one or multiple communicating tasks. I use the term *task* to describe a thread of execution, which is initiated by the target application and which can be managed by the operating system's scheduler. Thus, I refer to a target application as single-task if it consists of one execution thread. In a more general case, a target application can consist of several execution threads, i.e. be comprised of several tasks.

There may be multiple strategies of employing an operating system in a multicore system-on-chip. For example, a possible solution is to execute an individual OS instance on each core. In this setup, the OS instances operate in own private memory and work independently of each other. An alternative configuration is to declare one processing core as a master. The master core executes the OS code and schedules the execution of tasks on the slave cores. In symmetric multiprocessing (SMP) systems, all cores may cooperatively execute a single OS instance which controls the execution of tasks in a centralized way.

Multicore operation and algorithms for task management differentiate among various OS implementations. For comprehensive design space exploration involving task management, the system designer requires an abstracted scheduler model that would allow flexible evaluation of different task mappings and scheduling policies without the need of porting the application code to a specific operating system. Moreover, having the abstracted scheduler model, it is still desirable to simulate the execution of tasks, while capturing the hardware timing effects. Such timing effects may be caused by the behavior of local caches or additional communication latencies caused by contentions of the cores on the shared interconnect. For example, a task, which is co-executed with other tasks on a processing core, may evict some of the cache lines and, thus, cause higher miss rates for the other tasks after the task switch. Consideration of these effects allows for more accurate evaluation of tasks' execution time.

In this section, I address high-level OS modeling for system-level DSE of multicore architectures while considering the hardware timing effects of tasks' execution. In particular, I present high-level models of a scheduler and user tasks. The simulation of tasks' workload is based either on trace-driven (Section 5.1.1) or host-compiled binary-level simulation (Section 5.1.2). The scheduler manages the task execution on the underlying core models according to a certain pre-configured policy. The proposed scheduler model is not intended to identify the optimal task mapping on the given multicore architecture, which is a NP-hard problem. The purpose of the scheduler model is to validate the execution time and evaluate different scheduling policies and task migration strategies for a set of task mappings which has been already provided by the designer. Thus, the designer is capable of assessing various load distributions at early stages of MPSoC design when no concrete operating system is yet considered. The results of these estimations can guide the designer towards the final implementation with a specific OS.

One of the key properties of the proposed model is the ability to consider the workload that would be imposed by the scheduler in the cores. The *OS-related* workload is represented in the form of abstract traces which are simulated using the TDS method. The scheduler model supports various types of OS-related traces. For example, *context-save* or *context-load* traces represent writing and reading the core's registers from the main memory during task switching. A *scheduler* trace represents the execution of the scheduling code itself. The simulation of OS-related traces is interleaved with the simulation of tasks, thus, allowing for more realistic reproduction of the OS-based execution. In summary, the scheduler model has the following features:

- it can handle both TDS- and BLS-based user tasks;
- it is highly configurable in terms of task mapping and scheduling options in a multicore system-on-chip model;
- it allows preemptive scheduling of tasks. Moreover, it supports both local (when tasks are pinned to specific cores) as well as global scheduling (when tasks can be executed on any of the available cores);
- it supports task priorities and can be easily extended with various scheduling policies, e.g. earliest deadline first (EDF);
- it provides means for inter-task synchronization.

In the following sections, I discuss the models of the scheduler and tasks in detail.

5.2.1 *Task model*

The scheduler model manages the simulation of tasks according to the state diagram shown in Fig. 5.8. The tasks can be in one of four possible states. The initial state of a task is `Ready`. At this state, a task can immediately start executing. If there is an available core, the scheduler starts the task's execution and changes its state to `Executing`. If during the execution another higher priority task needs to be executed on that core, the scheduler preempts the low-priority task and changes its state back to `Ready`. `Hold` is an intermediate state between `Ready` and `Executing`. A user task is put into `Hold` state when OS-related workload associated with this task is simulated, e.g. context save or context load.

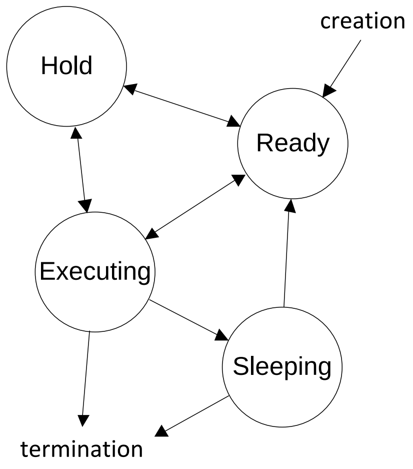


Figure 5.8: State diagram of tasks.

If an executing task has to be synchronized with another task and has to wait for a synchronization event, its execution is blocked and its state is set to `Sleeping`. When the event occurs, the state of the sleeping task is set back to `Ready` and the task can continue executing. Finally, the scheduler terminates a task when the task has finished executing or when a termination signal has been fired while the task was in the `Sleeping` state.

The parameters of a task as well as its invocation scenario are defined in a *task description*. Task descriptions are stored in a configuration file and contain the following data:

`name`	Symbolic name of the task;
`type`	TDS- or BLS-based task;
`core`	ID of the processing core on which the task has to be executed. If the value is not defined, the task can be executed on any core;
`priority`	Priority of the task (used for fixed priority scheduling);
`n`	Total number of times the task must be executed;
`period`	Period of task invocation;
`arg`	Path to an application trace (in case of a TDS-task) or compiled library (in case of a BLS-task);

`csave`	Path to an OS-related trace representing a context save;
`cload`	Path to an OS-related trace representing a context load.

Although the workload of TDS- and BLS-tasks are simulated differently, these tasks are handled equally by the scheduler. In Section 5.2.3, we will discuss the details of the initialization, execution and preemption of BLS- and TDS-tasks.

5.2.2 *Scheduler model*

5.2.2.1 *Structure*

The structure of the scheduler model is shown in Fig. 5.9. It is a dedicated SystemC module that manages the execution of tasks on core models. The scheduler requires a list of task descriptors in order to decide when tasks have to be created and on which cores they should be mapped. In addition to the task descriptions, the scheduler model requires execution traces for TDS-tasks and compiled libraries for BLS-tasks. They are used to simulate the actual workload of the tasks in core models. To enable simulation of the OS-related workload, the scheduler has to be additionally provided with OS-related traces, representing context save/load activity or the workload of the scheduler itself.

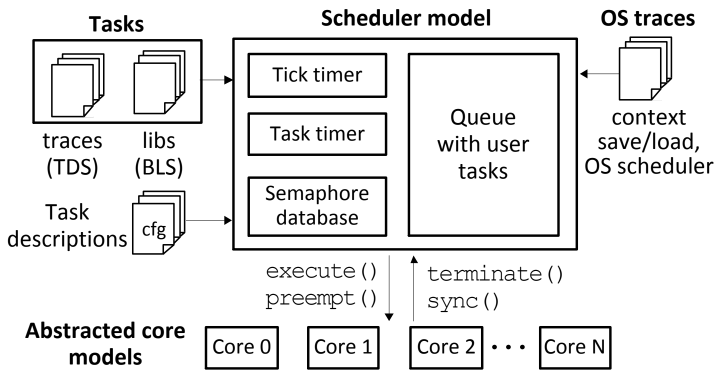


Figure 5.9: Structure of the OS scheduler model

In the beginning of simulation, the scheduler analyzes the task descriptions and configures the internal task timer. This timer stores invocation times of all tasks. At the time point of task's invocation, the scheduler model creates a temporary *task object* and adds it to the task queue (see Fig. 5.9). The task queue contains all current task objects independently of their state. During task scheduling, the scheduler traverses through this queue to determine `Ready` tasks that are eligible for execution. Depending on the tasks' priorities, the scheduler makes a decision whether to preempt currently executing tasks (if they have lower priorities) or to allow their further execution till the next scheduling phase. In addition to the task timer, the scheduler model contains a *tick timer*. This timer triggers periodical rescheduling of user tasks at pre-configured time intervals. Furthermore, in order to support task synchronization, the scheduler contains a database of semaphores. I will discuss the synchronization mechanisms of the scheduler in detail in Section 5.2.2.3.

The scheduler has a defined interface to the underlying core models. The simulation of the tasks' workload is started by calling function `execute()` implemented in the core. The workload simulation is preempted by the scheduler by calling function `preempt()`. In turn, the core models access the scheduler if there is a need for task synchronization (function `sync()`) or the simulation of the task's workload has been completed (function `terminate()`). When the workload simulation is completed, the corresponding task is terminated and the scheduler removes the task object from the task queue. The following sections describe the scheduler operations in more detail.

5.2.2.2 *Scheduler operations*

GENERATION OF TASK INSTANCES Generation of task objects in the scheduler is triggered by the task timer. When the signal from the timer is received, the scheduler iterates over the list with task descriptions, thereby operating according to the algorithm shown in Fig. 5.10. In the first step, the scheduler compares the invocation time in each task description with the current simulation time. In case of a match, the scheduler checks whether the previous object of that task is still active. This situation may occur if the task is periodic and the workload simulation from the previous task invocation is not yet completed. In this case, the scheduler postpones the creation of a new task object, notifying the designer that the deadline of the task has been violated.

If no previous task object is found in the queue, the scheduler creates a new task object and inserts it into the task queue with state `Ready`. Before switching to the next task description, the scheduler calculates the next invocation time of the task and configures the task timer correspondingly.

SCHEDULING OF USER TASKS The main steps performed during scheduling of tasks are shown in Fig. 5.11. The process of task scheduling (or re-scheduling) can be initiated in one of the following situations:

- A new task object has been created and inserted into the task queue;
- Periodical rescheduling occurs (OS tick);

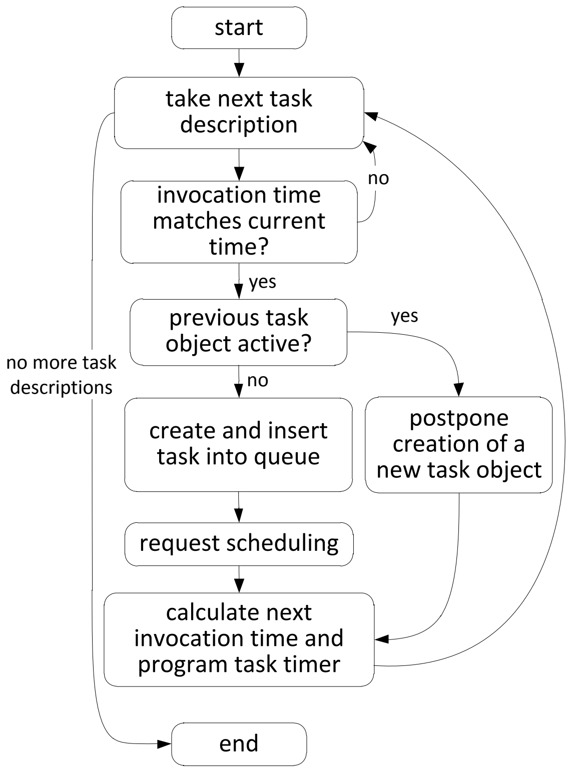

start
take next task description
invocation time matches current time?
no
yes
previous task object active?
yes
no
no more task descriptions
create and insert task into queue
postpone creation of a new task object
request scheduling
calculate next invocation time and program task timer
end

Figure 5.10: Algorithm for generation of instances of user tasks

- One of the currently executing tasks has made a request for synchronization,
- One of the tasks has finished its execution.

In the scheduling phase, the scheduler model iterates over the task queue, searching for entries in `Ready` state. Ready tasks have to be sorted according to their priorities first. In case of fixed priorities, the entries must be sorted according to the static priority values configured by the designer in the respective task descriptions.

For each ready task, the scheduling process starts with searching for an *idle* core first. I refer to a core as idle if no workload is currently simulated on this core. If an idle core is found, the scheduler then checks whether this core is *suitable* for the current task. The core is considered as suitable if:

- the current task has been pinned to this core by the designer (partitioned scheduling) or
- the current task can be executed on any core (global scheduling).

If an idle *and* suitable core is found, the scheduler assigns the OS-related traces to this core first. These traces are simulated before the

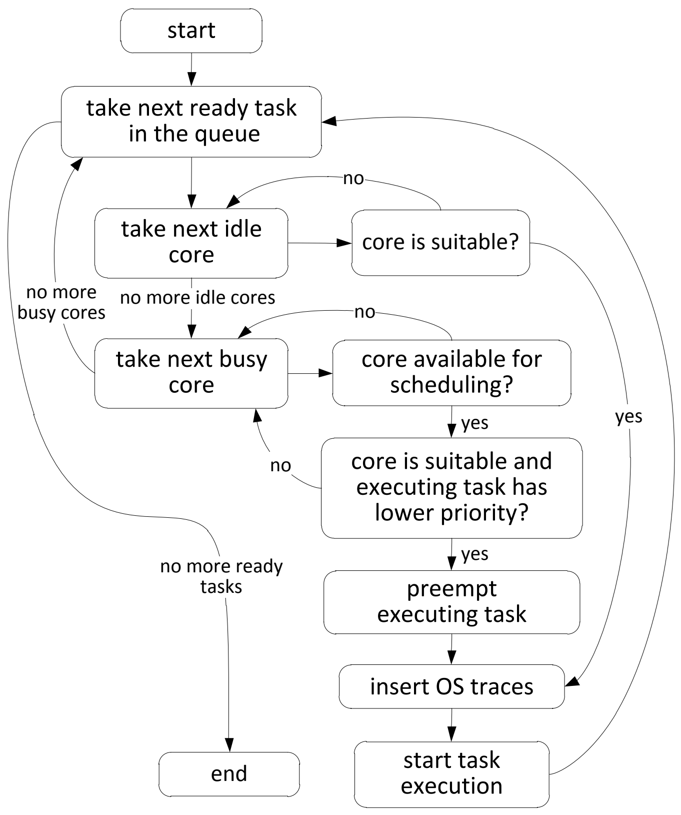


Figure 5.11: Algorithm for scheduling of user tasks

actual task's workload. They include a trace representing the workload of the scheduler itself and—if the current task has been preempted before—the context-load trace for the task. Afterwards the execution of the task can be started.

If there are no idle/suitable cores available, the scheduler model analyzes *busy* cores. First, the scheduler checks whether the current core is *available* for scheduling. A core is not available for scheduling if it is executing an OS-related trace. This situation can be compared to the execution of a critical section. If the core is available for scheduling, i.e. it is executing a task, the scheduler compares the priority of the executing task with the priority of the current ready task. If the executing task has a lower priority, it is preempted by the scheduler. Afterwards, the scheduler assigns a set of OS-related traces to the current core and the higher priority task can start executing. The preemption mechanism will be discussed in detail in the next section.

If the ready task cannot start executing on any of the available cores (e.g. because it has the lowest priority), it remains in `Ready` state and the scheduler moves to the next ready task in the queue. The scheduling process completes when all ready tasks have been processed.

PREEMPTION OF TASKS As mentioned earlier, a lower priority task can be preempted by a higher priority task in the scheduling phase.

The time diagram of the preemption process is shown in Fig. 5.12. In the beginning a core model executes low priority task T_1. At time t_1, the task timer notifies the scheduler that a new object of high priority task T_2 has to be created. After creating the task object, the OS scheduler calls `preempt()` function in the core. At the same time t_1, the state of task T_2 is changed from `Ready` to `Hold`. The `Hold` state remains until the beginning of T_2 execution. This intermediate state is needed to avoid multiple rescheduling of T_2 by the scheduler model. The state of T_1 is switched from `Executing` to `Hold` as well. In this case, `Hold` state is required to avoid too early execution of the task, since its context-save trace must be simulated first. The preemption of T_1 may take additional time to complete (denoted in the figure as *preemption lag*). This issue will be discussed in detail in Section 5.2.3.

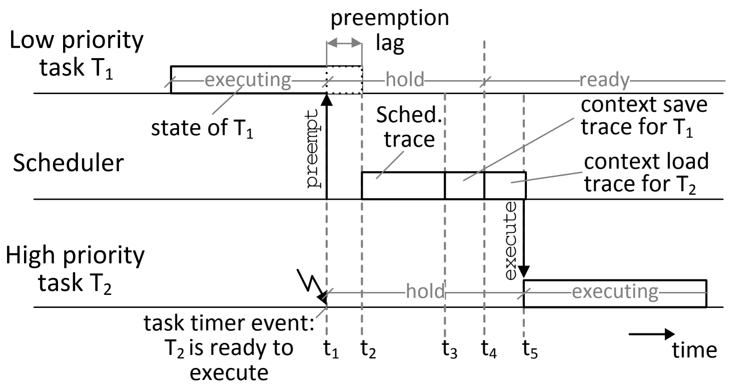


Figure 5.12: Preemption of user tasks

At time t_2, the preemption of T_1 is completed. In the next step, the scheduler assigns the scheduler trace representing the workload imposed by the scheduler itself. Afterwards, the scheduler initiates simulation of the context-save trace of task T_1 and context-load trace of task T_2. When the simulation of all OS-related traces is completed (time t_5), the preemption phase is finished and the execution of task T_2 can start.

TERMINATION OF TASKS Tasks are terminated when the simulation of its workload is completed. In case of TDS tasks, this situation occurs when the end of the application trace is reached. In BLS tasks, the workload simulation stops when the translated target code makes an *exit* system call. Upon the termination, the core calls `terminate()` function in the scheduler. Note that more than one tasks can terminate at a time. Therefore, in a general case, the scheduler processes a list of completed tasks as shown in Fig. 5.13.

In the first step, the object of the completed task is deleted from the task queue. Afterwards, the scheduler checks whether the creation of a new instance has been previously postponed for this task. If the result of the check is positive, the scheduler creates the post-

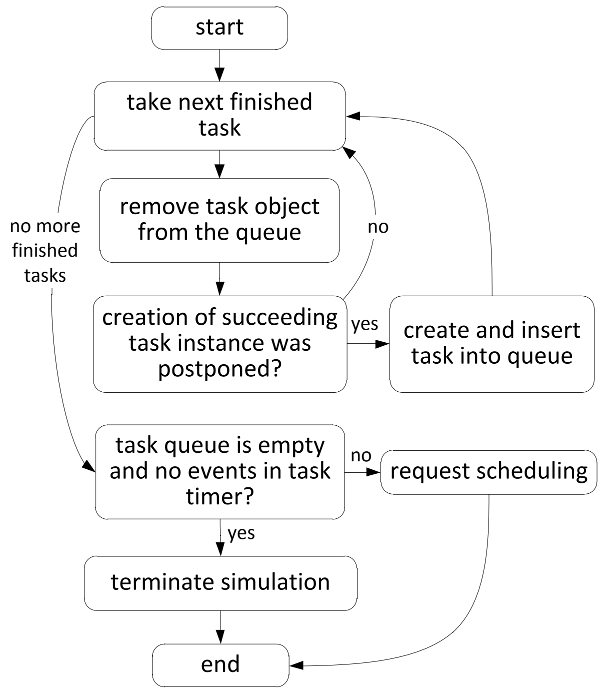

start
take next finished task
remove task object from the queue
creation of succeeding task instance was postponed?
yes
create and insert task into queue
no
no more finished tasks
task queue is empty and no events in task timer?
no
request scheduling
yes
terminate simulation
end

Figure 5.13: Termination of user tasks

poned instance and inserts it into the queue (similarly to the normal generation of task objects). When the task list is processed, the scheduler makes a decision whether to terminate the system simulation. The system simulation terminates when the task queue is empty and there are no entries task timer, i.e. no task invocations are planned in the future. If these conditions are not satisfied, the simulation continues and tasks get rescheduled.

OS TICKS In some OS implementations, tasks are rescheduled periodically during *OS ticks*, i.e. periodic, interrupt-triggered invocations of the OS scheduler. In the proposed scheduler model, ticks are simulated using the internal tick timer. A special situation has to be considered when no rescheduling of user tasks occurs. In this case, the scheduler trace still needs to be simulated as shown in Fig. 5.14.

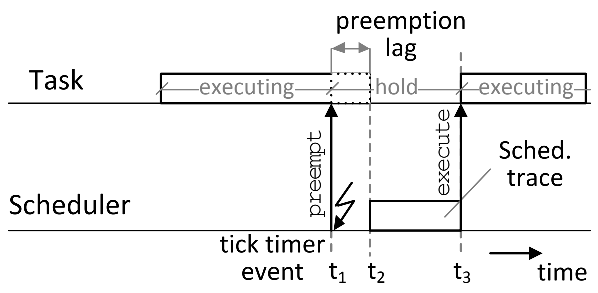


Figure 5.14: Simulation of OS ticks without task switching

The figure shows an execution of a task in the core model. At time t_1, the scheduler model is notified by the tick timer about the OS tick event. The execution of the task is temporarily stopped, however, no task switch occurs at this moment. When the simulation of the task is stopped at t_2, the scheduler model triggers simulation of the scheduler trace on the core model. At time t_3, the trace simulation is completed and the task can continue executing. Please note that no context save/load traces are simulated on the core in this situation, as no task switching has been occurred.

5.2.2.3 *Task synchronization*

The proposed scheduler model supports two synchronization mechanisms between tasks: *signals* and *semaphores*. The purpose of the synchronization mechanisms is to block the simulation of a task until another task explicitly notifies a synchronization event, allowing the first task to continue its execution.

Signal-based synchronization consists of two operations. In the first operation, a task calls `wait` function implemented in the scheduler model and specifies the ID of a signal it is waiting for. Please note that multiple tasks can wait for a specific signal ID. During the second operation, another task notifies the awaited signal by calling `fire` function. At this time point, the task waiting for the signal can continue executing. An important detail of signal-based synchronization is the fact that a waiting task reacts on the *event* of signal firing. In other words, firing of the signal has to follow the `wait` call. A signal fired before the `wait` call has no effect on the task waiting for this signal.

Semaphores are another synchronization mechanism supported by the scheduler. It is a very convenient method of synchronizing multiple tasks that process shared data. The semaphore-based synchronization consists of three steps. In the first step, a task registers a semaphore in the scheduler model by providing the semaphore's ID as well as its initial value. The value of the semaphore is stored in the semaphore database (see Fig. 5.9). A task can start processing the shared data only when it manages to decrement the semaphore value. When the processing of shared data is completed, the semaphore's value has to be incremented.

Multiple user tasks may try to decrement a semaphore at a time. However, these tasks can continue executing only if they can successfully decrement the semaphore, i.e. the semaphore's value is non-negative afterwards. If a task tries to decrement a zero value, its execution is blocked until the semaphore is incremented by another task. Semaphores initialized to 1 implement the functionality of *mutexes*. Mutexes allows for mutually exclusive execution of user tasks, i.e. they assure that only one task can process shared data at a time.

SYNCHRONIZATION PROCESS The synchronization process of tasks is shown in Fig. 5.15. In this example, high priority task T_1 waits for a signal which is fired by low priority task T_2.

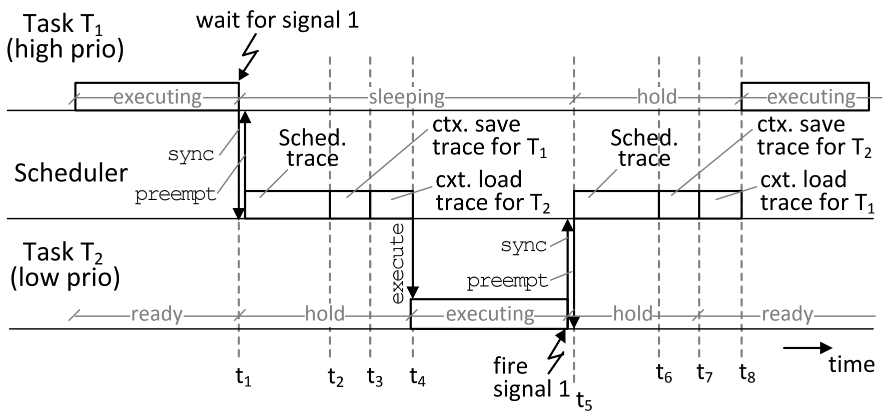


Figure 5.15: Synchronization of user tasks

At time t_1, task T_1 makes a synchronization request to the scheduler model, specifying that it is waiting for a signal with ID 1. The scheduler preempts the execution of this task and puts it into `Sleeping` state. In addition, the scheduler internally stores the information on the signal, which the task is waiting for. At the same time t_1, the core becomes available and the scheduler assigns ready task T_2 to the core. The execution of T_2 does not start immediately. The scheduler puts the task into `Hold` state, since the OS-related workload is not yet simulated. The scheduler initiates simulation of the scheduler trace followed by the context-save trace of T_1 and context-load trace of T_2. At time t_4, the simulation of the OS-related traces is completed, and task T_2 can start executing.

At time t_5, task T_2 performs a synchronization request to the scheduler model notifying fire of signal 1. The scheduler processes all tasks in `Sleeping` state and compares the fired event with the event the sleeping tasks are waiting for. Sleeping task T_1 matches the fired signal. Therefore, the scheduler changes its state to `Ready` and performs rescheduling of the tasks. T_1 has a higher priority than T_2. Therefore, the scheduler preempts T_2 and assigns T_1 to the core. At the same moment t_5, the scheduler sets the state of T_1 to `Hold`. After the simulation of the OS-related traces, which is completed at time t_8, T_1 can start executing.

The presented time diagram looks identical if the tasks would be synchronized using a semaphore. If task T_1 failed to decrement the semaphore, its execution would get preempted and its state would be set to `Sleeping`. T_1 could continue executing only when the semaphore had been incremented by T_2.

TASK QUEUE The information on the event's type (signal or semaphore), which a sleeping task is waiting for, as well as its ID are stored in the task queue. In addition, the queue contains all relevant information on the current task objects needed for the scheduler operations. An example of the queue's content is shown in Table 2.

Table 2: Content of the task queue

Name	State	Event	Evt ID	Prio	Core
Task 0	SLEEP	SIG	0	2	—
Task 1	SLEEP	SEM	2	2	—
Task 2	EXECUTING	—	—	1	0
Task 3	SLEEP	SEM	5	1	—
Task 4	EXECUTING	—	—	1	1
Task 5	EXECUTING	—	—	2	2
Task 6	SLEEP	SEM	10	2	—
Task 7	READY	—	—	3	—

Each entry in the queue has several fields. In the first field the task name is stored. The name is required for debugging purposes and for interpretation of the simulation results. The next field stores the current state of the task. The third and forth fields contain the type of the synchronization event and its ID. These two fields are relevant only if the task is currently in `Sleeping` state. The priority of the task is contained in the fifth field. Finally, the ID of the core, which the task is being executed on, is stored in the sixth field. This field is relevant only when the task is in `Executing` state.

ADDING SYNCHRONIZATION MECHANISMS TO TASKS In order to enable synchronization between tasks, the designer has to decide *which* mechanism (signals or semaphores) is required in the target scenario and *where* the synchronization points have to be inserted in the tasks' workload. These decisions depend on the functionality of the multi-tasking application as well as the target code.

The way *how* synchronization functions are inserted is different for BLS- and TDS-tasks. In TDS-tasks, the workload is simulated using an abstract trace. At simulation run-time, a core model reads the trace file and interprets the trace commands, e.g. `Delay`, `Read` or `Write` as shown in Listing 6. Synchronization in traces can be enabled by inserting additional trace commands to the trace file prior to trace-driven simulation. An example of adding signal-based synchronization to a trace is shown in Fig. 5.16.

As shown in the figure, signal waiting is initiated by adding a `SIGWAIT` command with the signal's ID provided in the argument. In turn, signal firing is initiated by a `SIGFIRE` command. The format of the synchronization trace commands must be understandable by the

```
SIGWAIT  1
DELAY    3
READ    32
DELAY    6
```

```
DELAY    4
WRITE   32
DELAY    3
SIGFIRE  1
```

Figure 5.16: Adding signal-based synchronization to traces by employing `SIGWAIT` and `SIGFIRE` trace commands.

trace decoder of core models. Semaphore-based synchronization can be added in a similar way by inserting `SEMINIT`, `SEMINC` and `SEMDEC` commands to the trace for initializing, incrementing or decrementing a semaphore correspondingly.

Adding synchronization to BLS-tasks is more difficult compared to TDS-tasks. BLS-tasks are based on functional simulation and, therefore, the use of synchronization mechanism may be data-dependent. For example, a BLS-task may need to perform a synchronization request only if a certain local variable in the target source code has a specific value. In this case, insertion of synchronization function calls in the functionally equivalent C- code is particularly difficult because of two reasons. First, it is difficult to precisely locate the synchronization point in the binary code. If the binary code is compiled with compiler optimizations, the lines in the source code and the instructions in the binary code cannot always be matched. Second, it is difficult to determine the source code variables in the binary code, since the binary code operates on the core's registers.

```
void func()
{
 if (var == 1)
    sigwait_1();
 ...
}

void sigwait_1()
{
 \\ empty function
}
```

```
// translated code
// of function
// sig_wait_1()

void b_0x420a() {
   PC = r[31];
}

...
```

```
// modified translated
// code of function
// sig_wait_1()

void b_0x420a() {
 // inserted sync.
 sched->sync(SIGWAIT, 1);
 PC = r[31];
}
...
```

Figure 5.17: Adding signal-based synchronization to BLS tasks using a call to an empty function `sigwait_x()`.

To address the problems above, I propose a simple solution based on calls to empty functions. This solution is shown in Fig. 5.17. Consider a target application containing function `func()` (left part of Fig. 5.17). Furthermore, assume that the execution of `func()` must to be synchronized by waiting for a signal and the synchronization has to take place only when local variable `var` equals 1. To enable synchronization, the designer has to insert an if-condition and a call to an empty function in the source code. The empty function has a specific naming convention. For example, in case of signal waiting the function name should be specified as `sigwait_x()`, where `x` is the signal ID. The translated code of the empty function is shown in the middle part of Fig. 5.17. It does not contain any processing instructions and consists only of a return (jump to the address stored in register `r31`). The key idea is to configure the binary-to-C code translator such that it automatically inserts a synchronization request to the scheduler model if the function name matches the proposed naming scheme. In the shown example, the generator inserts waiting for signal 1 (right part of Fig. 5.17) after analyzing the name of the empty function.

The reason of inserting a function call is the fact that position of function calls cannot be changed by the compiler (otherwise it would result in incorrect functional behavior). Moreover, the body of empty functions can always be exactly located in the binary code. As a result, synchronization requests to the scheduler model (i.e. calls to function `sync()`) can be inserted very precisely in the translated code. In addition, there is no need in matching local variable `var` with the processor's registers in order to perform a synchronization request conditionally. The if-condition added to the source code will be executed as part of the target application. If the condition is not true at simulation runtime, block function `b_0x420a()` (i.e. function `sigwait_x()` in the source code) will not be invoked. Consequently, the synchronization request to the scheduler model using `sync()` call will not take place too (as intended). The semaphore-based synchronization can be added to the translated target code in a similar way, i.e. by inserting calls to empty functions `semdec_x()`, `siminc_x()` or `seminit_x_y()` in order to decrement, increment a semaphore with ID=`x` and initialize its to value `y` respectively.

5.2.3 *Implementation details*

The scheduler model in the simulator handles TDS- and BLS-tasks equally. The exact simulation method of the workload (TDS or BLS) is hidden from the scheduler, and the scheduling process is completely decoupled from the actual workload simulation. To enable this decoupling, the tasks are implemented using a class hierarchy as shown in Fig. 5.18.

BLS-tasks are implemented in class `App`. This class is specific for *each* BLS-task and contains the actual implementation of the basic block functions. In turn, class `App` is derived from generic class `BLSlib`, which is common for all BLS-tasks. It defines common data structures (e.g. register variables) and functions (e.g. initialization of the stack memory). TDS-tasks are implemented in class `Trace`. From the simulation perspective, TDS-tasks differentiate only in the trace file which

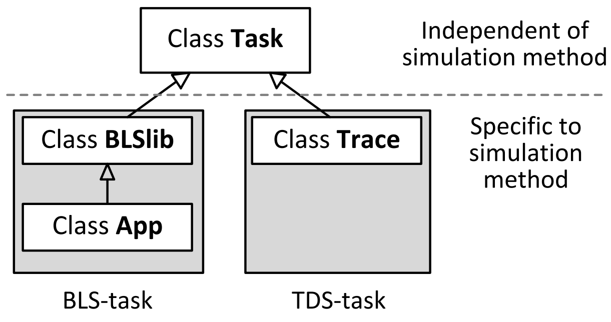

Class Task
Independent of simulation method
Class BLSlib
Class App
Class Trace
Specific to simulation method
BLS-task
TDS-task

Figure 5.18: Class hierarchy of TDS- and BLS-tasks.

is processed at simulation run-time. Therefore, class `Trace` is common for all TDS-tasks since the data structures and functions required for trace processing are similar among all TDS-tasks.

Although classes `BLSlib` and `Trace` are specific for the respective simulation method, they are derived from base class `Task` that does not depend on the simulation method. The scheduler handles TDS- and BLS-tasks (objects of `App` and `Trace` classes) by casting them to class `Task`. Class `Task` defines member variables and functions required for the scheduling purposes only. Additionally, the class declares virtual functions, e.g. `execute()` or `preempt()`, required by the scheduler. The implementation of these functions is specific for each simulation method.

The simulation of task workload is initiated in the context of a SystemC process in an instance of the core model. The core instance is selected by the scheduler as part of scheduling process. Because of different simulation methods, TDS- and BLS-tasks have different mechanisms for initialization, execution and preemption. In the following, these mechanisms are discussed separately for TDS- and BLS-tasks.

5.2.3.1 *TDS tasks*

INITIALIZATION In the initialization phase, a TDS task opens a trace file specified in the task description. The file is closed when the simulation of the trace file is completed and the task object is terminated in the scheduler.

EXECUTION To start the simulation of the trace file, the core model calls `execute()` function which is implemented in class `Trace`. During the simulation, the trace file is processed as shown in Fig. 5.19.

Processing of traces is performed using blocks of trace commands. These blocks are read from the trace file at simulation run-time. Each time the core model starts processing a trace command, it checks whether the preemption[3] of the current TDS-task has been initiated

3 The preemption process will be discussed in detail in the next section

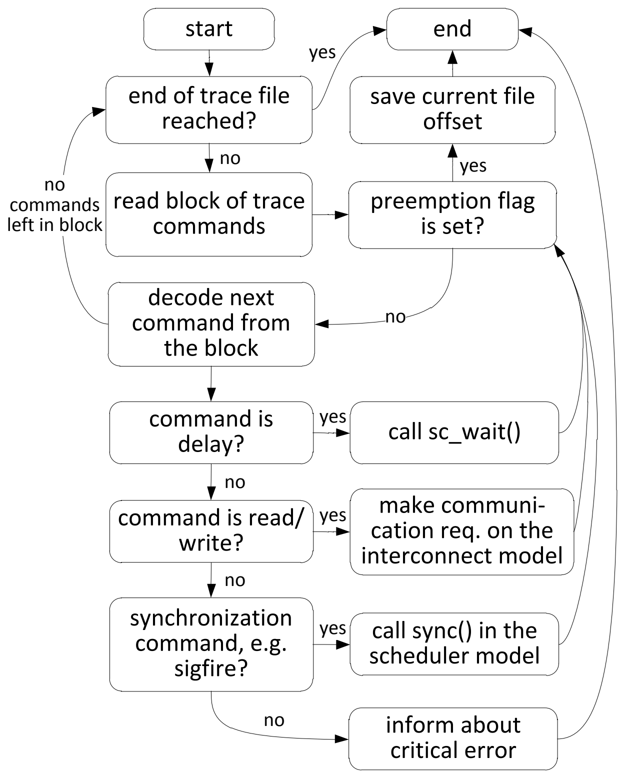

start
end
yes
end of trace file reached?
save current file offset
no
yes
no commands left in block
read block of trace commands
preemption flag is set?
decode next command from the block
no
command is delay?
yes
call sc_wait()
no
command is read/ write?
yes
make communi-cation req. on the interconnect model
no
synchronization command, e.g. sigfire?
yes
call sync() in the scheduler model
no
inform about critical error

Figure 5.19: Execution of a TDS task in SystemC.

by the scheduler. If the task has not been preempted, the core continues interpreting the current trace command. In case of a `Delay` command, the core model calls the `sc_wait()` function which is used to simulate a processing latency. On `Read` or `Write` commands, the core model makes a communication request to the model of on-chip interconnect. If the decoded command denotes a synchronization request, i.e. firing of a signal, the core calls the respective synchronization function in the scheduler model. A critical error occurs if the trace decoder reveals an unknown command. In this case, the core model informs the designer and the simulation is terminated. The trace simulation is completed when the end of the trace file is reached and there are no more trace commands to be processed.

PREEMPTION In order to preempt a TDS-task, the scheduler model has to set a special flag, which is polled in the beginning of the execution loop (Fig. 5.19). In certain situations, e.g. when the preemption is initiated while a trace command is being simulated, an additional time lag may appear as shown in Fig. 5.12. In this case, the task can be preempted only after the simulation of the current trace command is completed, thus, resulting in an additional delay or lag. The duration of the lag is not constant and depends on the time required for the current trace command to complete.

Despite the lag, the current preemption model is very efficient in terms of simulation speed, since the preemption condition is checked between trace commands only. The resulting lag can be neglected if task switching occurs significantly less frequently compared to the largest possible simulation time of a trace command. Moreover, the lag can be neglected if tasks do not have hard real-time requirements and the resulting timing error can be tolerated.

Nevertheless, the accuracy of preemption modeling can be still improved if it is required in the target scenario. To accomplish this, an additional SystemC event has to be added to all relevant SystemC wait functions. This event must be additionally notified when the scheduler wants to preempt the task's execution. In this case, the `sc_wait()` functions will return immediately after the notification of the preemption event. However, this solution is less efficient in terms of simulation speed because waiting for multiple events in SystemC requires more computational efforts.

When the preemption flag has been set by the scheduler, the execution of the loop is stopped (Fig. 5.19). In the next step, the current position in the trace file is stored in the task's object. When the preempted task will be executing again, the core will start to process the trace file from the stored position.

5.2.3.2 *BLS tasks*

INITIALIZATION To a great extend, the initialization of BLS tasks is related to the initialization of memory structures. First, the register variables have to pre-initialized to the values as specified by the target instruction set architecture. In addition, the program counter variable has to be set to the entry point[4] of the target code. For the correct program operation the arrays representing virtual heap and stack memory needs to be initialized with data as discussed in Section 3.1.1.

Modeling the target memory in case of multiple BLS tasks requires careful consideration. As mentioned in Section 3.2.2, functional and performance modeling of memory accesses is decoupled in BLS. The organization of the memory models is shown in Fig. 5.20. Each target application (consisting of one or multiple tasks) has its own virtual memory for storing functional data. The memory is allocated locally for each application using arrays `data_mem` and `stack_mem` (see Listing 2). This memory is private and cannot be directly accessed by other applications. However, the virtual memory of tasks of one application (e.g. tasks T_1–T_3 of application App_1) is shared. To enable this, arrays `data_mem` and `stack_mem` have to be common for these tasks. In turn, performance modeling does not require a real transfer of data and used solely to determine the timing of memory accesses.

4 The address of the first instruction to be executed.

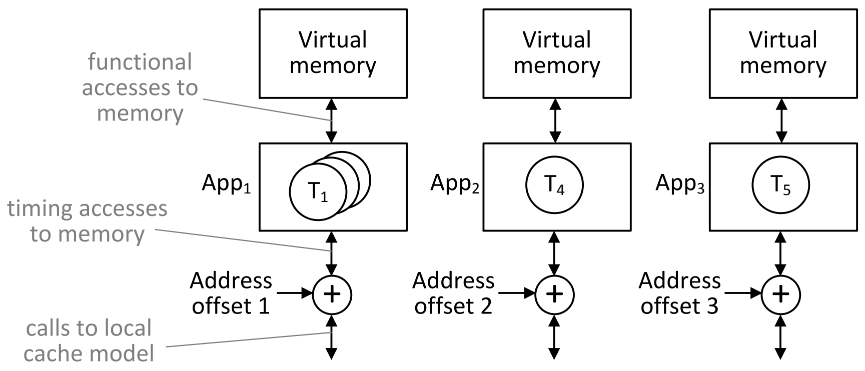


Figure 5.20: Functional and performance modeling of target memory in case of multiple applications.

The timing simulation of memory accesses is initiated by invoking the local cache models of the core, which the BLS task is assigned to. Using the target memory address provided by the application, the cache model determines whether the current accesses is a hit or a miss and, if necessary, initiates a communication request on the interconnect.

Many operating systems provide mechanisms for isolating user applications in the memory address space. The isolation is enabled by mapping application's virtual addresses to a different physical address space by means of a processor's memory management unit. Although the isolation of address spaces is not required for timing simulation, consideration of the memory management is still necessary because of the following reason. Assume two BLS tasks are mapped to the same core and erroneously operate in the same address space. In this case, the tasks will have an impact on each other by modifying the state of the local caches, i.e. the cache models can generate wrong hits or misses. To address this issue, I perform shifting of the address space for each application (using the address adders shown in Fig. 5.20) to prevent aliasing of target addresses in the cache model. The resulting space of physical memory addresses is shown in Fig. 5.21. Each application has its own address offset and, therefore, the addresses spaces of multiple applications do not overlap. The address offsets can be determined prior to simulation by analyzing the memory usage of the target applications. Please note that more complex memory mapping schemes can be modeled, e.g. using page tables, if it is required in the target scenario.

In addition to memory structures, BLS tasks require initialization of the array with function pointers (see Section 3.1.2 for details). These pointers are used to call basic block functions using the value of the program counter variable. Finally, each BLS-task requires initialization of an application-specific database with data dependencies among the memory instructions. This information is used for mod-

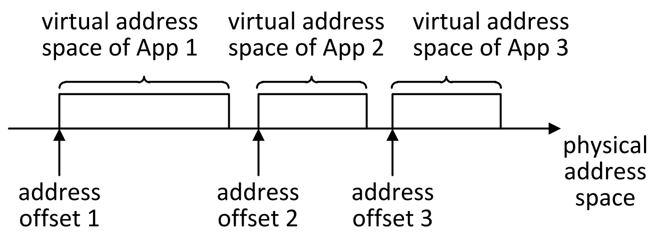

virtual address space of App 1
virtual address space of App 2
virtual address space of App 3
physical address space
address offset 1
address offset 2
address offset 3

Figure 5.21: Simplified mapping of virtual address spaces into a physical address space.

eling out-of-order cache effects as described in Section 4.5.1. I will discuss modeling of out-of-order effects later in this chapter.

EXECUTION An example of BLS-task's code adapted for the use in the SystemC simulator is shown in Listing 7. The structure of basic block functions and main execution loop are organized similarly to the translated code shown in Listing 1. Each basic block function is executed in the scope of the SystemC process of the core model. The annotated execution delays are simulated using the SystemC wait function, e.g. as shown line 9 in Listing 7. As mentioned previously, performance simulation of memory accesses is decoupled from the functional simulation and is started by calling the models of local cache. Please note that the target addresses supplied to the cache models are shifted by a constant value `offset`, which is automatically set by the scheduler to support the addressing scheme shown in Fig. 5.21.

Listing 7: The code of a BLS task adapted for the SystemC simulator.

```
1  void App::b_0x800() {      /* function for basic block 0x800 */
2    core->icache(READ, 0x800 + offset); /* performance access */
3
4    r[3]  = r[4] + r[3];
5    mem_access(WRITE, r[2], r[16] + 32); /* functional access */
6    core->dcache(WRITE, r[16] + 32 + offset); /* perf. access */
7
8    r[2] = r[2] + 1;
9    sc_wait(2, SC_NS);      /* wait for processing latency    */
10   ...
11 }
12 void App::execute() {
13   while (!preempted) {     /* main execution loop             */
14      table[INDEX(pc)]();  /* call to basic block functions   */
15   }
16   ...
17 }
```

PREEMPTION In order to preempt a BLS task, the scheduler model has to set a preemption flag. The flag is the object of the task's class. This flag is polled every time a new basic block function is going to be executed (line 13 in Listing 7). If the flag is set, the execution of the `while` loop is terminated and the program counter with the address of the next basic block is temporarily stored. Please note that the register variables do not have to be stored at the preemption. These variables are the members of the task's class and not of the core model. Thus, each BLS task in the simulator has its own set of the register variables. When the preempted task is set back to `Executing` after the preemption, the execution of the basic block functions will resume from the previously stored location.

Note that the preemption of BLS tasks is more coarse-grained compared to TDS tasks, because BLS tasks can be preempted between basic block functions only. This fact may result in a relatively large preemption lag (see Fig. 5.12) comparable with the execution latency of a complete basic block. Nevertheless, I use this preemption model because of its simplicity. It is efficient in terms of simulation speed and can be used as long as the resulting inaccuracy can be tolerated. However, preemption at a finer granularity level is possible as well. For example, basic block functions can be split into smaller parts. Using additional conditional statements, the execution of a block function can be preempted at each of these parts. However, this solution results in a larger size of the translated code and additional computational efforts. This solution must be used only if a higher timing accuracy of task preemptions is required.

5.3 SYSTEMC MODELS OF HARDWARE COMPONENTS

5.3.1 *Out-of-order core*

In the SystemC simulator, out-of-order processing cores are modeled as standalone SystemC modules. Each of these modules specifies a SystemC process, in the scope of which tasks are simulated. When the scheduler sends a command to execute a task, the core model calls `execute()` function implemented in the task and the simulation of the task's workload is started[5].

As discussed in Section 5.1, BLS and TDS methods presume different abstraction levels of the core's microarchitecture. Consequently, the internal organization of the core model depends on the type of a user task which is going to be simulated (see Fig. 5.22). In case of TDS tasks, traces completely abstract the core's microarchitecture and describe the core's timing behavior at the communication interface. Therefore, during simulation of TDS tasks, communication requests to instruction and data memory are directly forwarded to the external

5 See Section 5.2.3 for further details on simulation of BLS- and TDS-tasks.

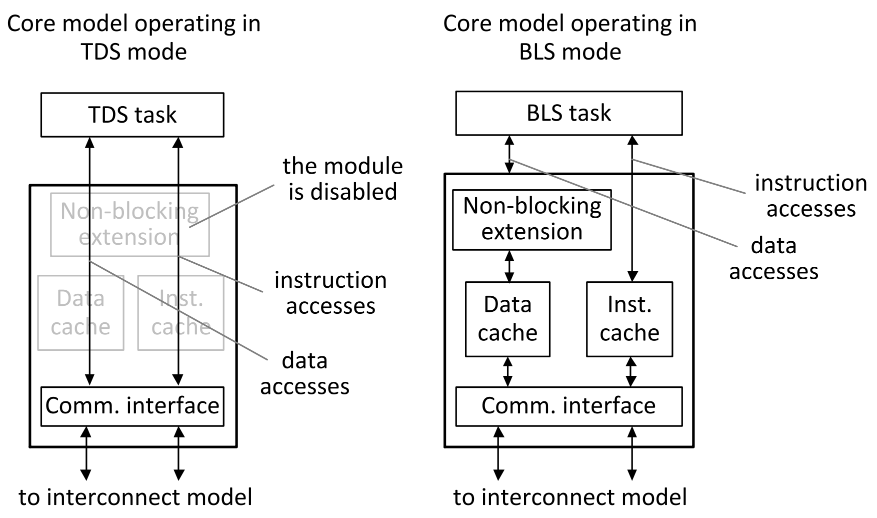


Figure 5.22: The internal structure of a core model. The model can operate in two modes, depending on the type of a task.

communication port of the core's model. In turn, BLS tasks presume a lower abstraction level of the core's microarchitecture and require co-simulation of local caches. Therefore, to support the simulation of BLS tasks (right part of Fig. 5.22), the core model incorporates models of local caches. In addition, it includes an *non-blocking extension* for considering the non-blocking behavior of a data cache. In the following sections, I discuss these internal components in detail.

5.3.1.1 *Non-blocking extension*

The purpose of the non-blocking extension of a data cache model is to reconstruct the core's timing behavior according to the principles described in Section 4.5.1. Particularly, this extension allows simulation of the core's out-of-order behavior in the presence of active data cache misses. If a data cache miss occurs, the extension unblock the execution of a BLS task and the task's execution is simulated further in the out-of-order mode. During the out-of-order execution, a BLS task can continue accessing the data cache. In case of a hit, the simulation of the task's execution continues according to the hit-under-miss policy. The out-of-order execution of a BLS task is stalled in one of the following situations:

1. The non-blocking cache extension discovers a memory instruction which depends on the active miss. In this case, the simulation of the task's execution is stalled till the simulation of the miss penalty in the interconnect model is completed.

2. The non-blocking cache extension discovers a memory instruction whose sequential number is too large to fit into the instruction window. In this case, the task's execution is stalled till the

active miss is completed and the instruction caused the miss can leave the instruction window.

The central element of the non-blocking extension is a queue shown in Fig. 5.23. The queue contains the status of recent accesses to the data cache. For each memory instruction, an entry in the queue is created. This entry specifies the address of the memory instruction as well as its sequential number. In addition, a queue entry may contain a sequential number of another memory instruction, which the current instruction depends on[6]. The information on instruction dependencies has to be pre-initialized in the BLS task object. Finally, each entry contains a *ready* flag indicating whether the memory operation is completed or not. This flag is set when the corresponding line in the data cache becomes available.

Inst. address	*Seq. number*	*Depends on*	*Ready*
0x1a0	134	-	no
0x2b4	141	-	yes
0x2bc	143	-	yes
0x110	151	134	no
0x124	155	-	yes

Figure 5.23: A queue employed in the non-blocking cache extension. The queue contains the status of memory instructions.

The queue can be considered as a model of the instruction window because of the following properties. Firstly, a new entry in the queue is created each time when a BLS task accesses the data cache. Secondly, the queue is always sorted by the sequential numbers of the entries, i.e. according to the program order of the memory instructions. Finally, the entries of the queue can be removed only in the program order starting from the top of the queue and only when the respective memory operations are completed. Thus, the operation of the queue is similar to the operation of an instruction window, in which instructions are committed in the program order as well.

FUNCTIONAL DESCRIPTION The execution of the non-blocking extension is invoked each time when a BLS task accesses the model of the data cache. At each invocation, the extension operates according to the algorithm shown in Fig. 5.24. In the first step, the sequential number of the memory instruction is analyzed. If the difference between the sequential numbers of the current instruction and the top instruction in the queue is smaller than the pre-configured parameter (representing the size of the instruction window), the current memory instruction fits into the instruction window. In this case, a new entry in the queue is created for this memory instruction.

6 Please refer to Section 4.5.2 for further information on the dependency analysis between memory instructions.

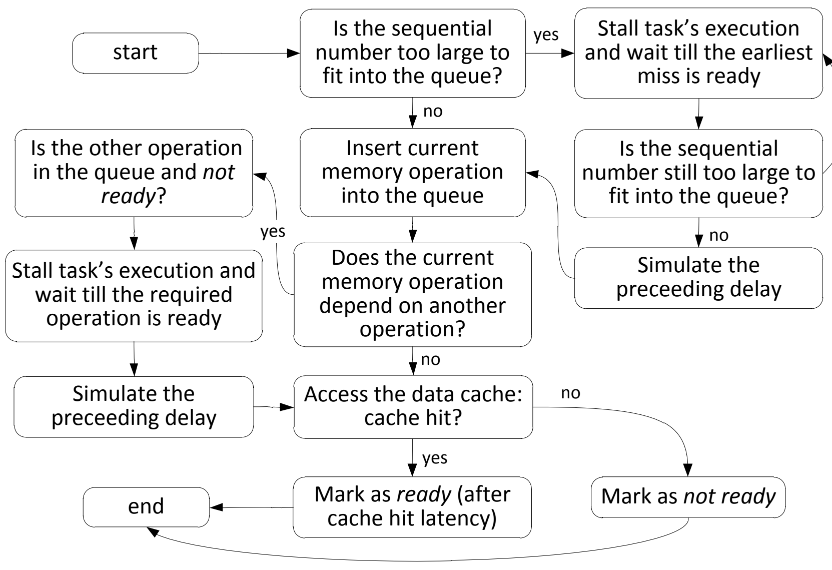

start
Is the sequential number too large to fit into the queue?
yes
Stall task's execution and wait till the earliest miss is ready
no
yes
Is the other operation in the queue and *not ready*?
Insert current memory operation into the queue
Is the sequential number still too large to fit into the queue?
yes
no
Stall task's execution and wait till the required operation is ready
Does the current memory operation depend on another operation?
Simulate the preceeding delay
no
Simulate the preceeding delay
Access the data cache: cache hit?
no
yes
end
Mark as *ready* (after cache hit latency)
Mark as *not ready*

Figure 5.24: Behavior of the non-blocking cache extension in a situation when a BLS task accesses the local data cache.

If the difference is larger than the size of the instruction window, the task's execution will be stalled till the active miss becomes ready. In a real core, this situation occurs when the instruction window is full and no further instructions can be fetched. When the active miss is ready, the ready entries in the queue can be removed starting from the top of the queue. When the sequential number of the memory instruction *suits* into the window, the non-blocking extension conservatively simulates the delay preceding the current access[7].

When a new entry was created, the extension checks whether the current memory instruction depends on any other memory instructions in the queue. If such instruction exists and it is not yet ready (see the left column in Fig. 5.24), the task's execution is stalled till the required memory operation is ready. Afterwards, the delay preceding the current memory operation is conservatively simulated once again[8].

If the current memory instruction does not depend on any other memory instructions in the queue (or the dependency has been already resolved as described above), the non-blocking extension calls the data cache model, providing the target address of the memory instruction. If the cache model generates a hit, the respective entry is marked as ready after the interval of the cache hit latency. In case of a miss, the current memory instruction remains in *not ready* state till the simulation of the miss latency is completed. Please note that the BLS task can further access the cache model in the presence of an active miss. In case of a hit (i.e. hit-under-miss), the memory instruction is

7 See Section 4.5.1.3 and Fig. 4.19(c) for further details.
8 Refer to Section 4.5.1.1 as well as Fig. 4.16(c) for further details.

marked as ready (after a hit latency). In case of a miss under miss, the memory operation is marked as *not ready* and the simulation of the second cache miss penalty is postponed till the simulation of the first miss is completed.

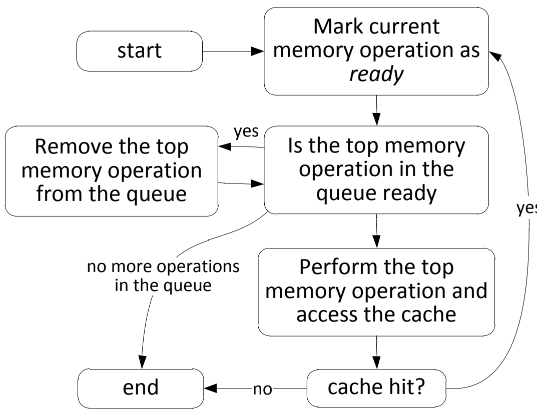

start
Mark current memory operation as *ready*
Remove the top memory operation from the queue
yes
Is the top memory operation in the queue ready
yes
no more operations in the queue
Perform the top memory operation and access the cache
end
no
cache hit?

Figure 5.25: Behavior of the non-blocking cache extension in a situation when the simulation of a cache miss penalty in the interconnect model is completed.

Fig. 5.25 shows the behavior of the non-blocking extension in a situation when the simulation of the miss penalty in the interconnect model is completed. In the first step, the corresponding entry in the queue is marked as *ready*. Afterwards, the queue is updated by removing the ready entries starting from the queue's top. If the top queue's entry is not ready after the update, it is a postponed miss under miss. In this case, the non-blocking extension calls the data cache once again, thereby starting the simulation of the postponed miss in the interconnect model. The algorithm ends if there are no more entries in the queue to be processed.

EXAMPLE TIMING DIAGRAM 1 In this section, I present an example of the module's operation and interaction of the module with other components. The timing diagram in Fig. 5.26 shows a situation when the execution of a BLS task is stalled due to the finite size of the instruction window. The core model executes a BLS task by simulating the annotated processing latencies[9] and performing accesses to the data memory as shown in the figure. Assume that the queue in the non-blocking extension is empty at the beginning. At time t_1, the task performs the first access to the data memory and calls the non-blocking extension. In the next step, the extension forwards the memory access to the data cache model which generates a miss event. The cache model makes the communication request to the interconnect module, thereby starting the simulation of the miss penalty. At

9 The simulation of annotated processing latencies is performed using SystemC `sc_wait()` function. For further details, please refer to Section 5.2.3.2.

the same time, the execution of the task is not stalled by the data cache miss. The non-blocking extension immediately responses and the task continues executing in parallel to the miss.

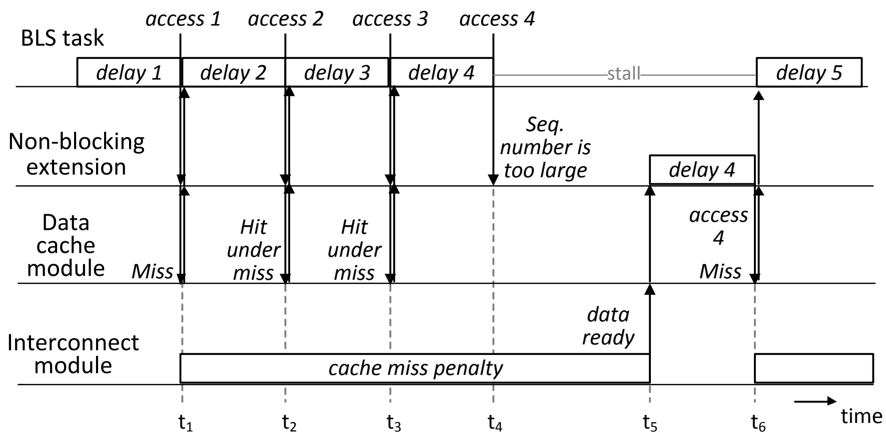


Figure 5.26: Stalling of the task's execution due to the finite size of the instruction window.

At time t_2 the simulation of the second execution delay in the out-of-order mode is completed and the BLS tasks makes the second request to the data memory. The extension makes sure that this memory instruction does not depend on the active miss and accesses the cache model in the presence of a miss. The access results in a hit and the task can continue executing (hit-under-miss policy). At time t_3, the BLS task makes the third independent access to the cache resulting in a hit under miss as well. At time t_4, the task makes the fourth request to the data memory. The extension checks the sequential number of the current memory instruction, which is too large to fit into the queue. Therefore, the extension does not respond to the task immediately and stalls the task's execution. Moreover, the extension postpones the data cache call.

At time t_5, the simulation of the first miss penalty is completed and the data cache model correspondingly notifies the non-blocking extension. As in the previous example, *delay 4* is conservatively simulated once again and the ready memory instructions is removed from the queue. At time t_6, the extension creates a new queue entry for the fourth data accesses (which is now fitting into the queue) and performs the postponed call to the data cache. The access results in a miss and the simulation of the miss penalty in the interconnect is started. At the same time, the core model continues simulating the fifth delay of the BLS task in the out-of-order mode without blocking the task's execution.

As can be seen from this experiment, the non-blocking extension allows reproducing the timing behavior of out-of-order cores, which are capable of hiding cache miss latencies. The extension stalls the execution of BLS tasks if the execution has to be blocked due to the finite size of the instruction queue. The task's execution continues only when the sequential number of the upcoming memory instructions fit in the suggested instruction window.

EXAMPLE TIMING DIAGRAM 2 The timing diagram in Fig. 5.27 shows the behavior of the non-blocking extension in a situation when a BLS task performs a memory access depending on the currently active miss. Similarly to the previous example, the queue in the extension is initially empty. At time t_1, the task accesses the data cache and the operation results in a miss. The cache makes a request to the interconnect model and the simulation of the cache miss penalty is started. At the same time, the task continues executing without being stalled. At time t_2, the task performs an independent access to the data cache resulting in a hit. It is a hit under miss and, therefore, the task's execution continues.

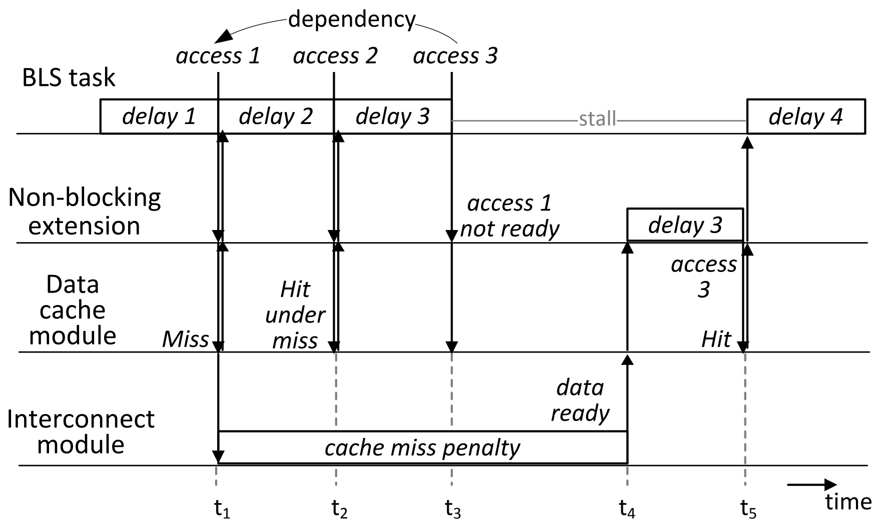


Figure 5.27: Interaction of the non-blocking cache extension of a data cache with other core's components in a situation when the task's execution is stalled due to dependency on the active data cache miss.

At time t_3 the task performs the third access to the data cache which depends on the first access. Since the first miss is not ready yet, the non-blocking extension stalls the task's execution. At time t_4, the simulation of the miss penalty is completed. In the next step, *delay 3* is conservatively simulated once again and the third access to the data cache is repeated. At the same time, the task continues executing and the core model starts simulating the next delay. In this way, the non-blocking extension allows reconstructing the behavior presented in Section 4.5.1.1.

5.3.1.2 *Cache*

The purpose of a cache module is to generate a hit or miss event for the given target memory address. The real data is not stored in the cache model since the caches are employed for timing simulation only. The central part of the cache model is a database shown in Fig. 5.28. Each entry corresponds to a cache line and specifies the following control information: the address tag of the line, *valid* bit as well as the *dirty* bit. The amount of entries in the database is determined by the total number of cache lines. This value can be configured by the designer.

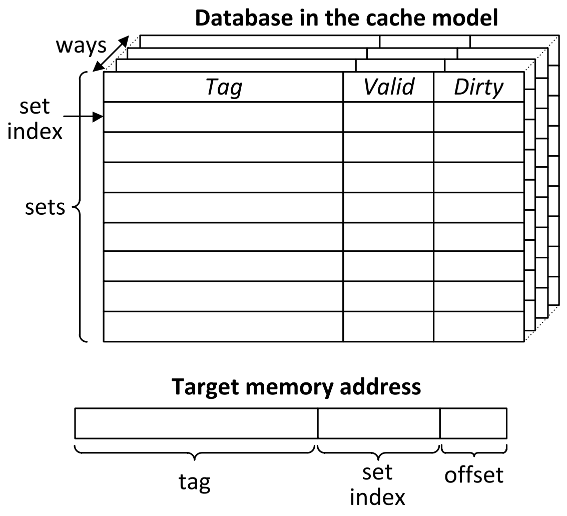


Figure 5.28: The structure of the database employed in the cache model.

The cache model can be employed for various configurations of set-associative caches. For this, the entries of the database are organized in a two-dimensional array. The first dimension of the array represents *sets* of a cache line. The second dimension of the array represents *ways* within each set. On each access, the cache model analyzes the given target memory address and extracts the *tag* and *set index* values from the address as shown in Fig. 5.28. The set index determines the position of the set in the database. In the next step, the model iterates through the ways in the selected set and compares the provided address tag with the tags stored in the set. If the tag of the given target address matches one of the tags in the set and the associated line is marked as valid, the cache model generates a hit event. If the current access is a write, the cache model additionally sets the dirty flag for the cache line. In this case, the line will be written back to the main memory if it is going to be replaced by another line.

If none of the valid tags in the set matches the provided tag, the memory access results in a miss. In this situation, the cache model makes a communication request to the on-chip interconnect, which, in turn, starts simulating the cache miss penalty. When the simulation is completed, the address tag of the line is placed into the set. During this operation, the current line may replace another valid line in the set[10]. Different algorithms for line replacement can be considered. For example, the lines be replaced according the FIFO principle. Alternatively, the least recently used (LRU) line in the set can be replaced. Upon the replacement, the cache model overwrites the old address tag and sets the dirty flag if the current access is a write.

5.3.2 *Communication infrastructure and memory*

In this section, I introduce a generic model of an arbitrated bus for obtaining communication latencies in the system-on-chip environment. In addition, I present an extension to the bus model for considering the effects of the snooping-based coherence mechanism employed in local caches.

5.3.2.1 *Arbitrated bus*

The SystemC simulator incorporates a model of a generic arbitrated bus shown in Fig 5.29. The model enables communication between multiple masters and slaves components. To enable arbitration, the interface of each bus master is assigned a fixed priority. Thus, when multiple masters want to communicate simultaneously, the arbiter grants access to the master having the highest priority. In this model, I assume that the bus is blocked during the complete interval of data transfer including the slave's access latency.

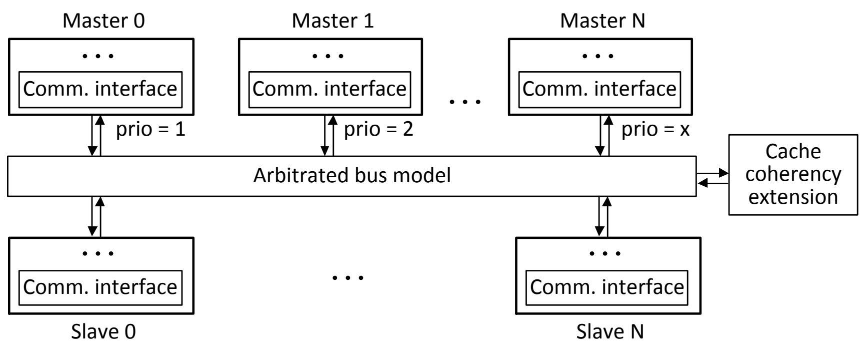


Figure 5.29: A model of an arbitrated shared bus employed in the simulator.

The algorithm of bus arbitration is shown in Fig. 5.30. The arbitration process starts when a master makes a communication request to the bus model. If the bus is currently occupied by another master, the current request is added to an internal request queue. In this case, the requesting master has to wait till the next arbitration, which will start when the current communication is completed. If the bus is idle or the communicating master has just completed the data transfer, the

10 This situation occurs if all lines in the set are valid.

bus arbiter checks the queue for all masters waiting to be granted. If there are multiple pending requests available in the queue, the arbiter selects a master with the highest priority and the other masters have to wait for the next arbitration. Each time the simulation of communication is completed, the master which initiated the communication is notified by the bus.

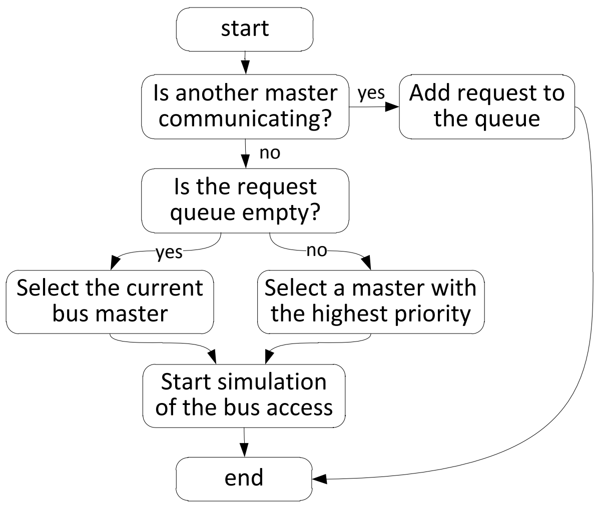

start
Is another master communicating?
yes
Add request to the queue
no
Is the request queue empty?
yes
no
Select the current bus master
Select a master with the highest priority
Start simulation of the bus access
end

Figure 5.30: Bus arbitration process

5.3.2.2 *Cache coherence support*

The use of shared data by multiple processing cores with local caches requires additional mechanisms for keeping the caches coherent among each other. In most cases, the use of a specialized cache coherency protocol results in additional traffic on the interconnect. As a result, cache coherency may increase the execution time of target applications. In *write invalidate* coherency protocols, which according to [26] are currently most common, the lines in other caches are invalidated when a core writes to shared data. The other cores will experience additional cache misses when they access this data at a later time. Therefore, consideration of cache coherency is important for accurate timing simulation, even if the cache models do not contain real target data and, in fact, there is no need for keeping the cache models consistent.

In this section, I introduce an enhancement to the arbitrated bus model which adds functionality of a snooping-based protocol. Particularly, the MESI coherency protocol [55] for write-back caches is considered in the model. As described in [26], the protocol defines four states of cache lines, which also determine the protocol's name: Modified, Exclusive, Shared and Invalid. J. Hennessy and D. Patterson in [26] provided a detailed description of the operation of the MESI protocol. The states and their possible transitions are shown in

Fig. 5.31. A cache line in *shared* state is present in multiple caches. A cache line in *exclusive* state is present in one cache only. In both cases, the data in caches and the main memory is same. However, when a core writes data to an exclusive or shared line, the state of the line is changed to *modified*. A modified cache line can be present only in one cache and is dirty, i.e. the data in the line is different from the data in the main memory. Although *exclusive* state is not necessarily required in the protocol, it avoids unnecessary invalidations when a core writes data to the exclusive line. Since the line is exclusive in this core, writing to this line does not require invalidations in other caches. Thus, the additional traffic caused by the invalidation can be avoided. More information on the MESI protocol can be found in [26].

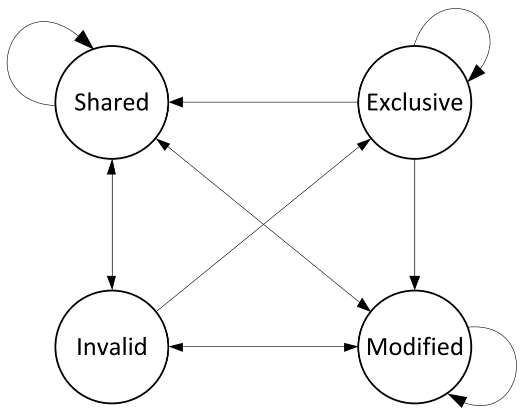


Figure 5.31: States of cache lines and possible transitions between them in the MESI cache coherency protocol [26].

In snooping-based protocols, the states of cache lines are locally stored and managed in each cache controller[11]. During the program execution, the cache controllers monitor (or snoop) the activity on the shared interconnect and correspondingly update the status of the cache lines. For simplicity reasons, I implemented the cache coherency mechanism in the bus model. In this way, modeling complexity can be reduced without changing the behavior of the coherency protocol.

Implementation of the cache coherency protocol in the bus model requires a separate database. It stores the coherency status of all cache lines currently present in the system (thereby resembling the directory-based coherency protocols). The database stores the owner of cache lines in *modified* or *exclusive* state. The operation of the coherence extension is shown in Fig. 5.32. The behavior depends on the type of communication performed by a core. In fact, two communication types in a write-back cache are possible:

11 This is in contrast to directory-based coherence protocols, in which the status of the cache lines is stored in a central database.

- the core reads a line from the main memory due to a miss on a read or write,
- the core writes a line back to the main memory due to a line replacement.

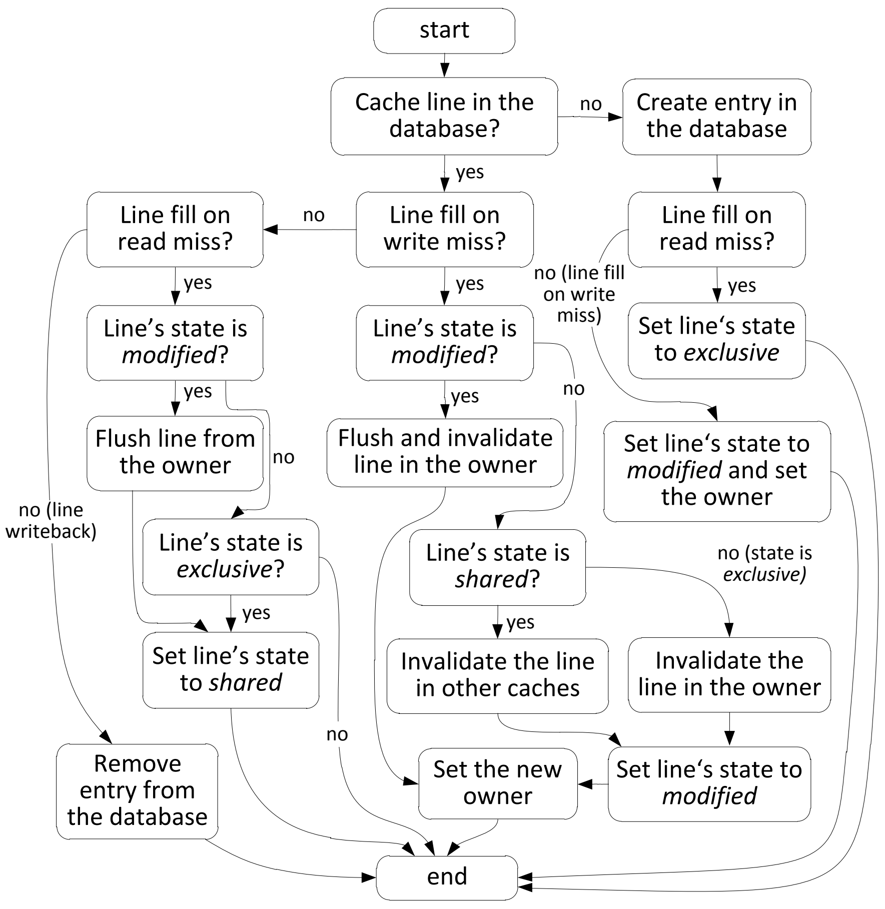

start
Cache line in the database?
no
Create entry in the database
yes
Line fill on read miss?
no
Line fill on write miss?
Line fill on read miss?
yes
yes
no (line fill on write miss)
yes
Line's state is *modified*?
Line's state is *modified*?
Set line's state to *exclusive*
yes
yes
no
Flush line from the owner
no
Flush and invalidate line in the owner
Set line's state to *modified* and set the owner
no (line writeback)
Line's state is *exclusive*?
Line's state is *shared*?
no (state is *exclusive)*
yes
yes
Set line's state to *shared*
Invalidate the line in other caches
Invalidate the line in the owner
no
Remove entry from the database
Set the new owner
Set line's state to *modified*
end

Figure 5.32: Operation of the coherency mechanism in the bus model when a core reads or writes a line on the bus. The operation is performed according to the MESI coherency protocol [55].

When a core reads or writes a cache line to the main memory, the bus model checks whether the description of this line is available in the database. If not, it is a first miss on this line and none of the caches contains this line yet. In this case, the bus model creates a new entry in the database. The following step depends on whether the miss was caused by a read or write operation. In case of a miss on a read, the state of the line is set to *exclusive*, since other caches do not contain this line. In case of a miss on a write, the line's state is set to *modified*. In addition, the bus model stores the ID of the line's owner, which is by a convention the ID of the core performing the access.

In the following, I discuss three cases when the line's description has been found in the database. In the first case, the cache line must be read from the memory due to a write miss. If the line has been already set to *modified* state, this line is in another cache and the line is dirty[12]. The current owner has to invalidate this line and flush it back to the memory. However, if the cache line is in *shared* state, the main memory contains correct data and no flushing is needed. Nevertheless, the line in other caches must be still invalidated, since the modified line can be only in one cache.

In the second case, the cache line is read from the main memory due to a read miss. If the line is currently in *modified* state, the owner has to flush the line and the state is then changed to *shared* without line invalidation. If the line is currently in *exclusive* state, the main memory contains updated data and no additional actions are needed apart from just setting the state to *shared*.

In the last scenario (left part of Fig. 5.32), a core writes back a dirty cache line. A dirty cache line is always in *modified* state, i.e. it is present in this cache only. Therefore, when a dirty line is replaced in the owner and is being written-back to the memory, none of the caches contains this line any more. Therefore, upon a write-back the line can be removed in order to keep the database as small as possible.

The implementation of the cache coherence mechanism in the bus model requires an additional communication channel with cache models for transferring control information (not shown in Fig. 5.29). Particularly, the additional channel is used by caches to report write hits to the bus model. In case of a write hit, a cache does not communicate with the bus. Nevertheless, the bus model has to be still notified about these events in order to update its internal database correspondingly.

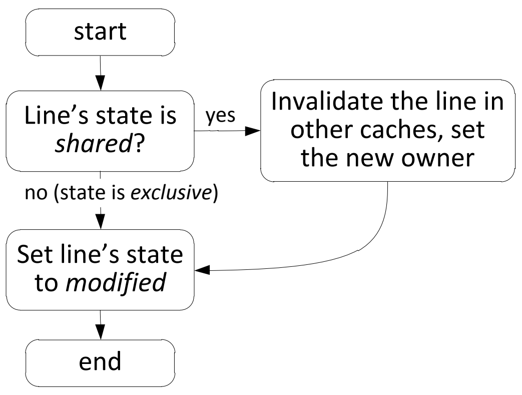

start
Line's state is *shared*?
yes
Invalidate the line in other caches, set the new owner
no (state is *exclusive*)
Set line's state to *modified*
end

Figure 5.33: Behavior of the cache coherency extension in case of a write hit.

The operation of the coherence extension in case of a write hit is shown in Fig. 5.33. If a cache reports a write hit on a line that is currently in *shared* state, the bus model sends invalidation command to other caches and stores the owner's ID in the database. Afterwards, the line's state is changed to *modified*. However, if the line was in *ex-*

12 Note that the core requesting a line due to a write miss cannot be the owner of the line. The owner of a *modified* line always experiences write hits.

clusive state, the line's state is simply set to *modified* and invalidation of other caches is not required (because the current cache is the only owner of this line). It is the benefit of using *exclusive* state in the coherency protocol. The invalidation events can be propagated using either the data bus itself or a separate medium. I assume that a separate communication channel exist in the bus for sending invalidation commands. However, the present bus model can be easily adjusted to consider propagation of invalidation signals over the data bus.

5.3.2.3 *Memory*

In the SystemC simulator, the actual functional data in caches is abstracted. Therefore, for simulation simplicity, the memory component is abstracted as well. The memory component is modeled by adding an additional memory access latency to the communication latency on the arbitrated bus. The value of the latency can be configured by the designer prior to simulation.

6 EXPERIMENTAL RESULTS

This chapter presents experimental results of the proposed host-compiled binary-level simulation method that considers complex timing behavior of out-of-order cores. Particularly, in Section 6.1 I evaluate context-aware simulation introduced in Chapter 4 and investigate the optimization techniques proposed in Section 4.6. The evaluations in this section are performed in C++ environment. In Section 6.2, host-compiled simulation and trace-driven simulation techniques are compared. The experiments in this sections are performed using models from Section 5.3 and considering the overhead of SystemC-based simulation. Finally, I demonstrate system-level design space exploration of a multi-tasking JPEG application on a multicore platform in the SystemC simulator. The exploration is carried out by means of the scheduler model introduced in Section 5.2.

6.1 CONTEXT-AWARE HOST-COMPILED SW SIMULATION

6.1.1 *Experimental setup*

6.1.1.1 *Reference core simulator and target applications*

As described in Section 4.3.2.1, context-aware host-compiled simulation requires a reference cycle-accurate simulation of the target processing core to obtain execution time of basic blocks. As a reference simulator, I used *sim-outorder* tool from the SimpleScalar suite [5] with memory extensions [14]. It is a cycle-accurate simulator implementing PISA instruction set[1] [13] and supporting out-of-order execution, branch prediction and non-blocking cache behavior. The SimpleScalar tool allows simulation of different processing cores and is highly configurable. The configuration of the processing core simulated during the experiments is shown in Table 3. The simulator was additionally extended with a monitoring unit for capturing timing information of basic blocks.

As target software, I used a set of 32 different embedded applications from MiBench [23] and MediaBench [37] benchmark suites as well Embedded JPEG Codec Library [92]. The benchmarks represent different application domains and include the following programs: *basicmath* (small workload), *bitcount*, *qsort* (small workload), *susan.corners*, *susan.edges*, *susan.smoothing*, *jpeg* (encoding and decoding), *ejpeg*, *lame*, *mpeg2* (decoding), *epic* (encoding and decoding), *rasta*,

1 A slightly modified version of the MIPS instruction set architecture

Table 3: Configuration of the core simulated in the SimpleScalar.

Amount of integer ALU's	4
Amount of integer multiplier/dividers	1
Amount of floating point ALU's	4
Amount of floating point multiplier/-dividers	1
Amount of L1 cache ports	2
Register update unit size	64
Load/store queue size	64
Instruction fetch queue size	4 inst
Ratio of front-end speed relative to execution core	1
Maximum instruction issue width	4 inst./cycle
Issue after a misspeculation	yes
Instruction decode width	4 inst./cycle
Instruction commit width	4 inst./cycle
Branch predictor type	Bimodal predictor using a branch target buffer with 2-bit counters
Bimodal predictor table size	2048
Sets in branch target buffer	512
Associativity of branch target buffer	4
Return address stack size	8
Extra branch mis-prediction latency	3 cycles

stringsearch (large), *dijkstra* (small), *patricia*, *blowfish* (encode and decode), *pgp* (encoding), *rijndael* (encoding and decoding), *sha*, *crc32*, *fft*, *ifft*, *adpcm* (encoding and decoding), *g721* (encoding and decoding), *gsm* (toast and untoast).

The benchmarks were executed using standard input data provided with the application's code. The code of the applications was compiled with *gcc* cross-compiler v.2.7.2.3 provided in the SimpleScalar suite with enabled compiler optimizations.

6.1.1.2 *Workflow of experiments*

The evaluation of the proposed host-compiled simulation was performed separately for each target application according to the workflow shown in Fig. 6.1. The experiments were carried out in two phases. Firstly, in the generation phase the binary code of the target applications was executed in SimpleScalar in order to obtain context-dependent timing of basic blocks. As discussed in Section 4.2, the block's timing was derived assuming perfect instruction and data caches in the reference simulator. Afterwards, the target binary code was translated into the equivalent C code and annotated with the timing information obtained in the previous step. The annotation was performed under consideration of multiple execution contexts of basic blocks to enable context-aware compiled simulation as described in Section 4.3.3. Afterwards, the annotated code was compiled and executed on the host computer. For the host compilation of the translated code, I employed *gcc* compiler v4.4.3 with enabled compiler optimizations. The result of the compilation was an executable program. The execution of this program produced an estimation of the execution time of the target application assuming the performance characteristics of the target processing core.

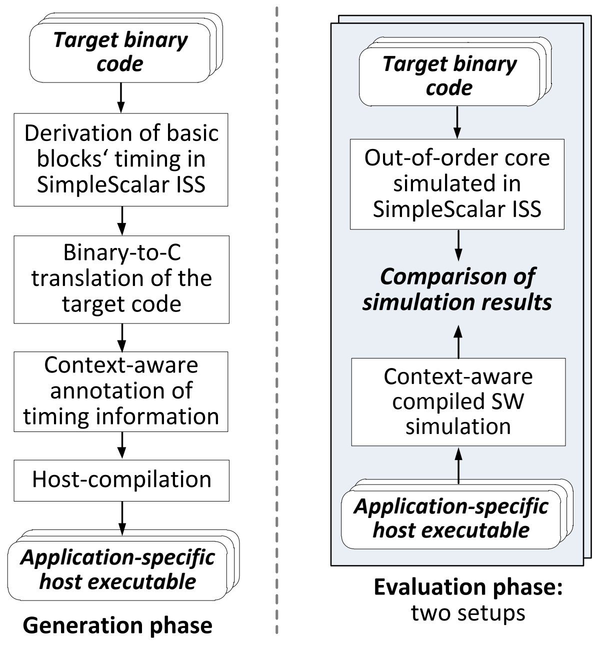

Target binary code
Derivation of basic blocks' timing in SimpleScalar ISS
Binary-to-C translation of the target code
Context-aware annotation of timing information
Host-compilation
Application-specific host executable
Generation phase
Target binary code
Out-of-order core simulated in SimpleScalar ISS
Comparison of simulation results
Context-aware compiled SW simulation
Application-specific host executable
Evaluation phase: two setups

Figure 6.1: The workflow of experiments consisting of two phases. In generation phase, the target binary code is translated to C-code, annotated with context-dependent timing information of basic blocks and compiled on the host computer. In the evaluation phase, the resulting executable was employed to perform host-compiled binary-level compiled simulation. The results of this simulation were then compared to the reference cycle-accurate simulation in SimpleScalar.

In the evaluation stage, the produced executable file was employed to perform context-aware host-compiled simulation. In the next step, the results of compiled simulation are compared with the reference simulation of the same target code in SimpleScalar. Two different setups are used in the evaluation phase. In the first setup, both SimpleScalar reference simulation and host-compiled simulation are performed assuming perfect instruction and data caches. In this setup, the following investigations are conducted:

- First, I evaluate the efficiency of the optimization of translated code introduced in Section 4.6.1. The results of this evaluation are presented in Section 6.1.2 of this chapter.
- Afterwards, I investigate the impact of employing multiple execution contexts per basic block in host-compiled simulation (see Section 4.3 for further details), while considering the above optimization of the translated code. The results of this investigation are presented in Section 6.1.3.
- Finally, I evaluate averaging of basic block timings described in Section 4.6.2. This technique is evaluated in Section 6.1.4.

In the second setup, compiled SW simulation and SimpleScalar simulation are performed assuming a realistic data cache and perfect instruction cache. In this setup, I focus on the behavior of non-blocking data caches and perform the following investigations:

- Firstly, I evaluate modeling of the non-blocking cache behavior discussed in Section 4.5. The results of these investigations are presented in Section 6.1.5.
- Secondly, I assess the technique of static reordering of cache accesses described in Section 4.6.3. The corresponding results are presented in Section 6.1.5.1.

All experiments presented in the next sections were performed on a host computer with an Intel Core i7 870 2.93 GHz processor and 16 GB of RAM running a 64-bit Linux Ubuntu 10.04 operating system.

6.1.2 *Optimization of binary-to-C translation*

As discussed in Section 4.6.1, with a larger length of context signatures (i.e. with larger considered sequences of previously executed basic blocks), more contexts of basic blocks can be differentiated in context-aware host-compiled simulation. In Section 4.6.1, I already presented some experimental results showing that the total amount of contexts differentiated by signatures significantly increases at larger signature lengths. Fig. 6.2 shows the total percentage of identical block timings at various signature lengths. At a signature length of 16, almost 60% of context-dependent timings are identical and, therefore, redundant. This observation provides an opportunity for an optimization of the translated code.

In Section 4.6.1 I presented a solution to this problem. This solution relies on the fact that even if the *total* amount of block timings increases with the signature length, the amount of *unique* basic block timings does not increase at the same rate. In other words, a large portion of context-dependent timings of a basic block are identical, i.e. at many contexts the basic block has the same execution delays

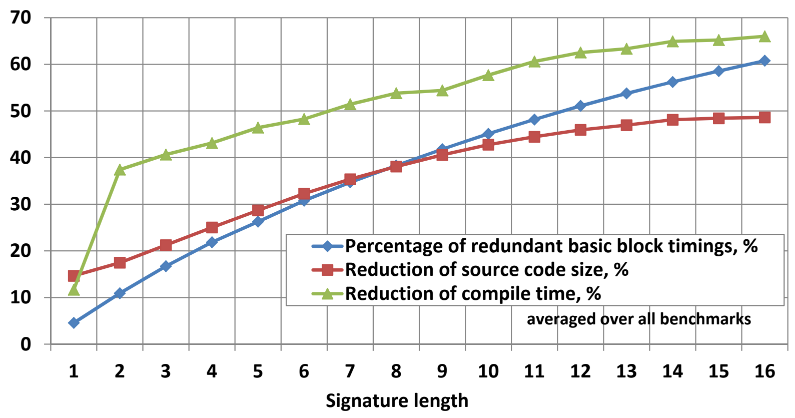


Figure 6.2: Results of the proposed optimization of binary-to-C translation of the target code. Elimination of redundant basic block timings allows for significant reduction of the size and compilation time of the translated code. The plots represent average values for all benchmarks.

and the same order of memory accesses. In fact, the unique timings can be reused among multiple contexts as proposed in Section 4.6.1, thereby reducing the size of the translated code as well as its compilation time. The results of this optimization are shown in Fig. 6.2. On average, for all benchmarks the size of the translated code could be reduced by 11% at a signature length of 1 and 66% at a signature length of 16.

The absolute number of block timings annotated in the translated code after optimization is shown in Fig. 6.3. The higher the signature length is, the more timings are redundant and hence can be eliminated. Compared to a case with non-optimized translated code shown in Fig. 4.21, the amount of annotated basic block timings is significantly reduced. At a signature length of 16, the optimization is most efficient. For example, for *lame* benchmark the amount of annotated timings was reduced from 55085 to 8723 (see Fig. 4.21 for comparison). However, for some benchmarks the optimization is less efficient. For example, in *crc_32* he amount of annotated block timings was reduced only from 1130 to 880.

Fig. 6.4 shows the size of translated code assuming the optimization above as well as the reduction of context signatures for identical contexts introduced in Section 4.6.1. Despite the optimization, the code size still significantly increases with a signature length. Notably, the size changes differently for benchmarks. For *crc32* benchmark, which is the smallest one in terms of the context count, the size of the translated code increased from 1.02 MByte (at a signature length of 1) to only 1.15 MByte (at a signature length of 16). At the same time, for benchmark *lame* the code size increased significantly from 4.93 MByte to 46.44 MByte.

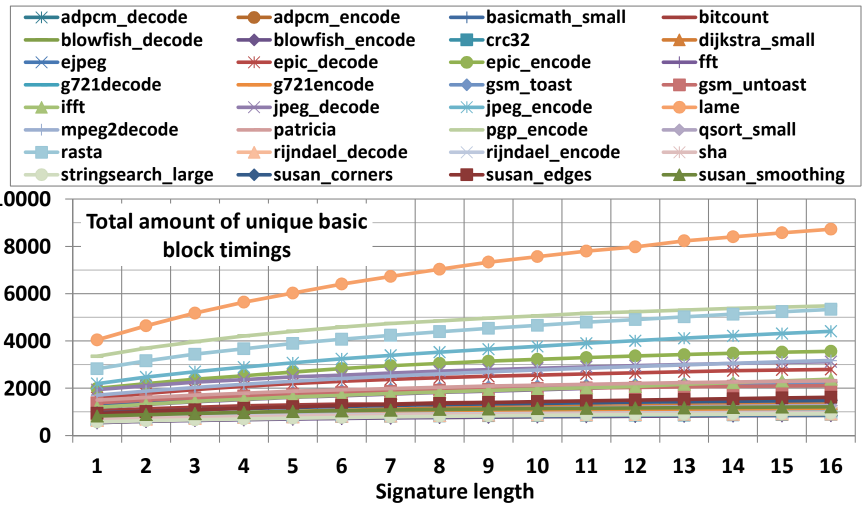


Figure 6.3: Total amount of unique basic block timings after optimization (see Fig. 4.21 for comparison).

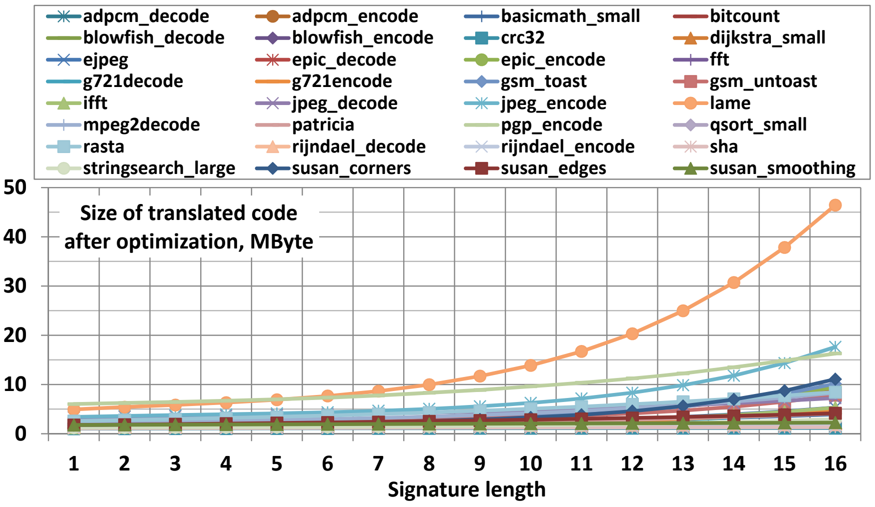


Figure 6.4: Size of translated code after optimization.

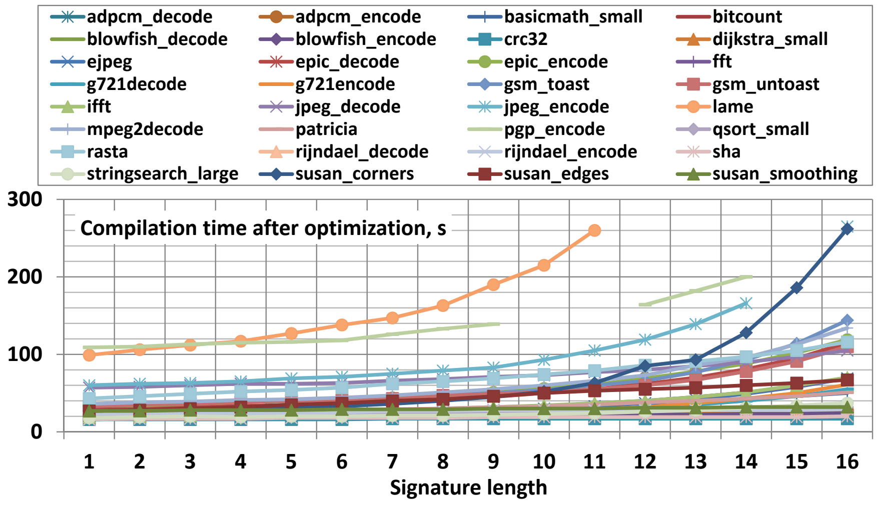


Figure 6.5: Compilation time of translated code after optimization.

The compilation time of the translated code on the host computer is shown in Fig. 6.5. Despite the optimization, the host-compilation failed for some benchmarks at large signatures. The failures were caused due to an internal error in the compiler. For example, the compilation failed for *pgp_encode* at signature lengths of 10, 11 and *jpeg_encode* at the length of 15. Nevertheless, the compilation of these benchmarks at larger signatures was again successful even if the amount of contexts was higher. I presume that this issue is related to limitations of the compiler and further clarification is required. The compilation time was smallest for benchmarks *adpcm_decode*, *adpcm_encode*, *crc32* and *sha* at signature length of 1, taking in total 16 s. As can be seen in Fig. 6.5, despite the optimization the compilation time still increases with a signature length as well. The highest compilation time of 265 s was measured for *jpeg_encode* benchmark at a signature length of 16.

6.1.3 *Context-aware compiled simulation*

In this section, I evaluate the efficiency of employing context-aware timing of basic blocks (see Section 4.3) and run-time reordering of memory accesses (see Section 4.4.2). The goal of these techniques is to improve the accuracy of host-compiled simulation in case of out-of-order processors, i.e. to reconstruct the execution time and the order of memory accesses as close to the reference simulation as possible. As mentioned earlier, in this section instruction and data caches are assumed to be perfect.

6.1.3.1 *Estimation of execution time*

First, I evaluate the accuracy of execution time estimation using context-aware host-compiled simulation. For this, I compare the execution time of benchmarks estimated using the reference cycle-accurate simulation in SimpleScalar and the execution time produced by context-aware host-compiled simulation at various lengths of context signatures. In the following, the timing error of context-aware host-compiled simulation for i^{th} benchmark is defined as

$$e_i = \frac{t_{c,i} - t_{s,i}}{t_{s,i}}, \tag{17}$$

where $t_{c,i}$ is the execution time of the benchmark estimated using context-aware host-compiled simulation and $t_{s,i}$ is the benchmark's execution time estimated using in SimpleScalar.

The timing error is shown in Fig. 6.6. As anticipated, the error decreases with larger signature lengths. Moreover, the experimental results show that for some benchmarks the error may even slightly increase at higher signature lengths, e.g. for *dijkstra_small*, *epic_decode* or *susan_smoothing* benchmarks. The reason for this is the fact that only

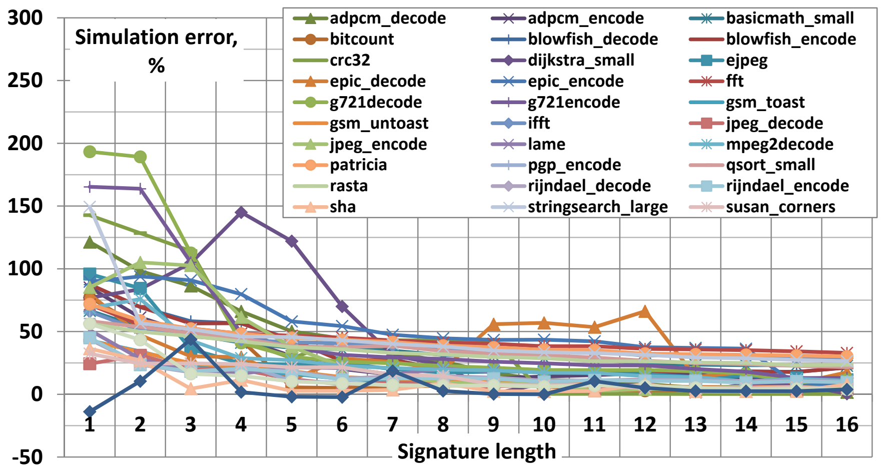


Figure 6.6: Timing error of context-aware host-compiled simulation relative to the reference simulation in SimpleScalar.

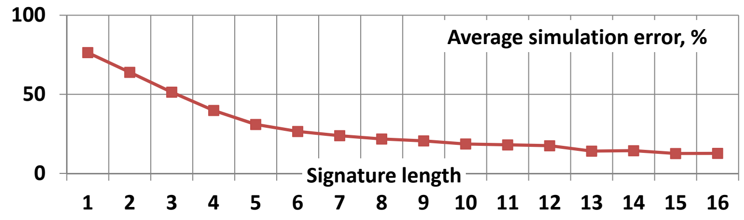


Figure 6.7: Average timing error of context-aware host-compiled simulation over multiple benchmarks compared to the reference simulation in SimpleScalar.

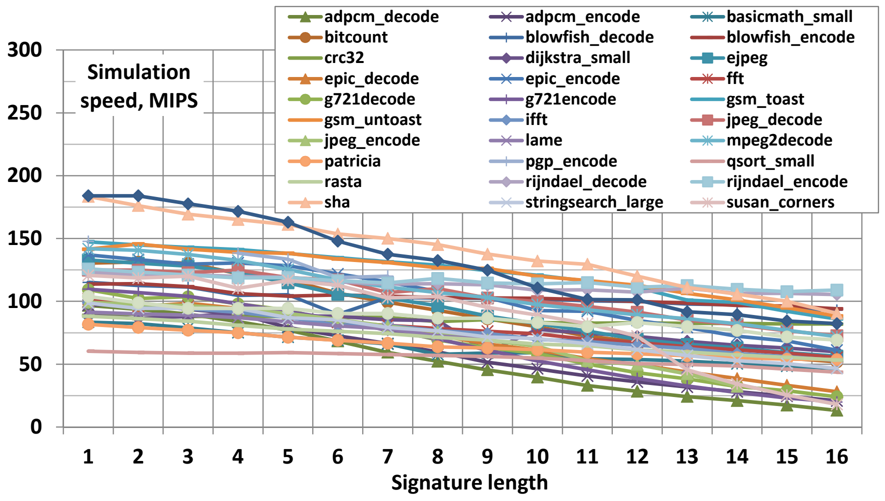


Figure 6.8: Speed of context-aware host-compiled simulation expressed in millions of simulated target instructions per second of real time.

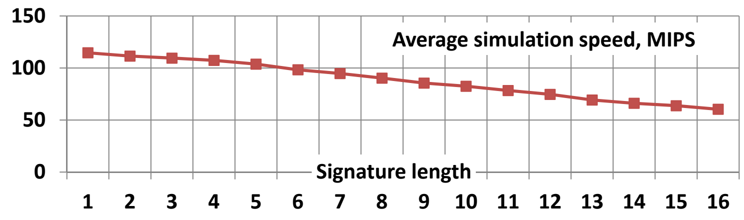


Figure 6.9: Average speed of context-aware host-compiled simulation over multiple benchmarks.

the first observed timing is annotated in the translated code. In reality, the signature length can be too small for considering all possible timings of basic blocks. If the first observed timing occurs infrequently and differs significantly from the other timings. As a result, a timing error is produced. This error can be positive as well as negative, because the first observed timing of a context can be smaller or larger than the average timing.

In addition, I evaluate the mean error of context-aware host-compiled simulation over all benchmarks shown in Fig. 6.7. The mean error E_j at signature length j is defined using the following formula:

$$E_j = \frac{1}{n} \sum_{i=0}^{n} |e_{i,j}|, \tag{18}$$

where $e_{i,j}$ is the simulation error for i^{th} benchmark at signature length j, and n is the total amount of benchmarks.

As can be seen in the figure, the average error of context-aware host-compiled simulation was gradually reduced from 76% at signature length of 1 to 12.7% at signature length of 16. The reduction was at largest for signature lengths between 1 and 6. However, starting from a signature length of 7 the decrease of simulation error was only marginal. This is due to the fact that the diversity of basic block timing is affected by the previous blocks executed most recently. In turn, consideration of more than 6 blocks as a context signature does not have further significant impact on simulation accuracy.

The speed of context-aware host-compiled simulation is presented in Fig. 6.8. The simulation speed is expressed in millions of simulated instructions of the target code per second of real time (MIPS). As can be seen from the experimental results, the speed decreases with a signature length. This is due to the fact that with a larger amount of contexts per basic block, more conditions have to be checked at simulation run-time. Moreover, the speed of context-aware host-compiled simulation differentiates among benchmarks. The reason for this is the fact that each benchmark is different in terms of the code structure and the total amount of contexts that have to be simulated. At a signature length of 1, the largest speed of 184 MIPS was measured for *susan_smoothing* and the lowest speed of 60 MIPS was measured for *qsort_small*. In turn, at a signature length of 16, the largest speed of 109 MIPS was measured for *rijndael_encode* and the lowest speed of 13 MIPS was observed for *adpcm_decode* benchmark. The average speed of context-aware host-compiled simulation over all benchmarks is shown in Fig. 6.9. The average speed gradually decreased from 115 MIPS at a signature length of 1 to 60 MIPS at a signature length of 16.

6.1.3.2 *Order of memory accesses*

In this section, I evaluate the correctness in reconstructing the order of memory accesses in context-aware host-compiled simulation. Particularly, I investigate the impact of memory reordering at simulation run-time introduced in Section 4.4.2.

REORDER DEPTH Estimation of an error of a memory access order is a challenging task. There is no straightforward approach for a quantitative assessment of the error. For this purpose, I introduce a new metric which is called *reorder depth*. The reorder depth characterizes the degree at which memory accesses are reordered in an out-of-order processor. The reorder depth is determined for every memory access. For explanation purposes, let us assume a sequence of memory accesses shown in Fig. 6.10. The numbers labeled on the axis denote the sequential numbers of the respective memory instructions. In the presented example, memory instructions 4, 5 and 3 are simulated out-of-order. In the following explanation, I use terms memory access and memory instruction interchangeably.

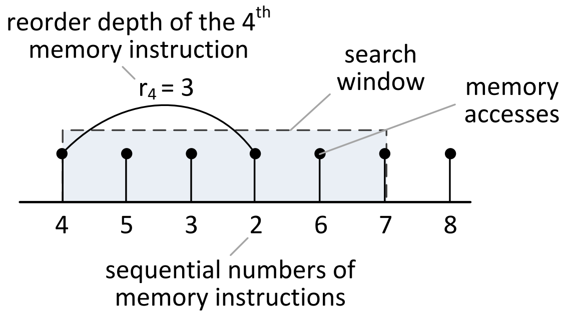

reorder depth of the 4th memory instruction
$r_4 = 3$
search window
memory accesses
4
5
3
2
6
7
8
sequential numbers of memory instructions

Figure 6.10: Determination of reorder depth for memory accesses observed in a simulation.

The reorder depth of a memory access is determined by analyzing subsequent memory accesses within a certain search window. The size of this window corresponds to the size of the instruction window of the target processing core. The reorder depth of a given instruction is defined by the most distant memory instruction in the window with a sequential number lower than the one of the given instruction. As shown in the example, for instruction 4 there are two memory instructions in the search window, whose sequential number is lower than 4 (instructions 2 and 3). The reorder depth of instruction 4 is determined by instruction 2 since it is most distant. The actual value of the reorder depth is calculated as the amount of memory accesses simulated between instructions 4 and 2 plus one. There are two accesses between instruction 2 and 4, hence the reorder depth of instruction 4 (r_4) equals 3. Similarly in the shown example, the reorder depth of instruction 5 equals 2 and the reorder depth of instruction 3 equals 1. The following instructions 2, 6, 7 and 8 are simulated in the program order and, therefore, their reorder depth equals 0.

Taking these assumptions into consideration, for each benchmark, we can determine a vector of reorder depths for a complete sequence of memory accesses simulated in a host-compiled simulation as

$$c = [r_1, r_2, ..., r_n], \tag{19}$$

where n is the total amount of simulated memory accesses.

Similarly, for the same benchmark, we can define a vector of reorder depths for memory accesses if the benchmark is simulated in the reference simulator SimpleScalar as

$$s = [r_1, r_2, ..., r_n]. \tag{20}$$

The difference between these two vectors determines the error of reconstructing the memory access order in host-compiled simulation relative to the reference SimpleScalar simulation. The error can be quantitatively estimated by determining the root mean square error (RMSE) of the vectors' elements as follows:

$$e = \sqrt{\frac{\sum_{k=0}^{n} (c_k - s_k)^2}{n}}, \tag{21}$$

where c_k and s_k are the reorder depth of k^{th} memory access in compiled simulation and SimpleScalar correspondingly, n is the total number of simulated memory accesses. In addition, we can calculate the *mean* RMSE of reorder depths over all benchmarks using the following formula:

$$e = \sqrt{\frac{\sum_{i=0}^{m} \sum_{k=0}^{n} (c_{i,k} - s_{i,k})^2}{\sum_{i=0}^{m} n_i}}, \tag{22}$$

where $c_{i,k}$ and $s_{i,k}$ are the reorder depth of k^{th} memory access for i^{th} benchmark in compiled simulation and SimpleScalar correspondingly, n_i is the number of simulated memory accesses of i^{th} benchmark and m is the total amount of benchmarks evaluated during the experiments.

The estimation results of the RMSE of reorder depths for each benchmark at different signature lengths are shown in Fig. 6.11. For most benchmarks, the RMSE does not change significantly with the signature length. This behavior has been also observed for the mean RMSE over all benchmarks shown in Fig. 6.12. However, for some benchmarks the signature length had a notable impact on the RMSE values. For *crc32* benchmark, the RMSE value could be significantly decreased with the signature length. For *epic_decode* benchmark, the RMSE decreased at smaller signature lengths and then increased at larger lengths. For *dijkstra_small* benchmark, a highly distinguishable peak could be observed at a signature length of 9. I anticipate that the reason for these non-typical RMSE curves is the fact that only the *first* timing and memory access order is captured for each context during

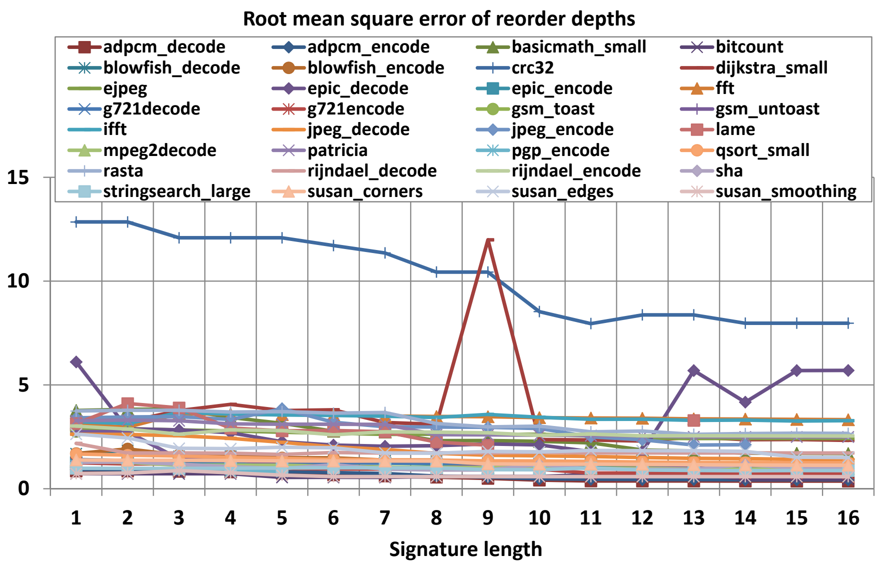


Figure 6.11: Root mean square error of reorder depths of memory accesses simulated in context-aware host-compiled simulation compared to the reference cycle-accurate simulation in SimpleScalar.

measurements, while the others are ignored. If the captured memory access order occurs infrequently during the benchmark's execution and significantly differentiates from the ignored ones, the resulting error will be accumulated during the simulation.

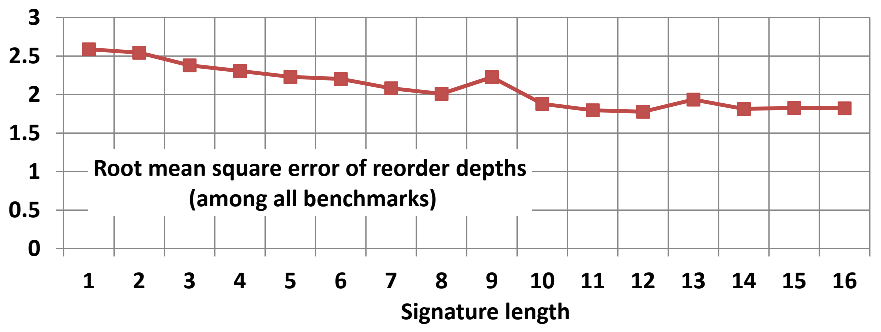


Figure 6.12: Root mean square of reorder depths averaged over all benchmarks.

In Section 6.1.4, the above problem will be solved by averaging timings of a basic block at a certain context.

OVERHEAD OF DYNAMIC MEMORY REORDERING As discussed in Section 4.4.2, the order of memory accesses is reconstructed at simulation run-time by means of a queue. The queue temporarily holds the memory accesses performed within basic blocks which are classified as *out-of-order* prior to simulation using the algorithm presented Section 4.4.1 .

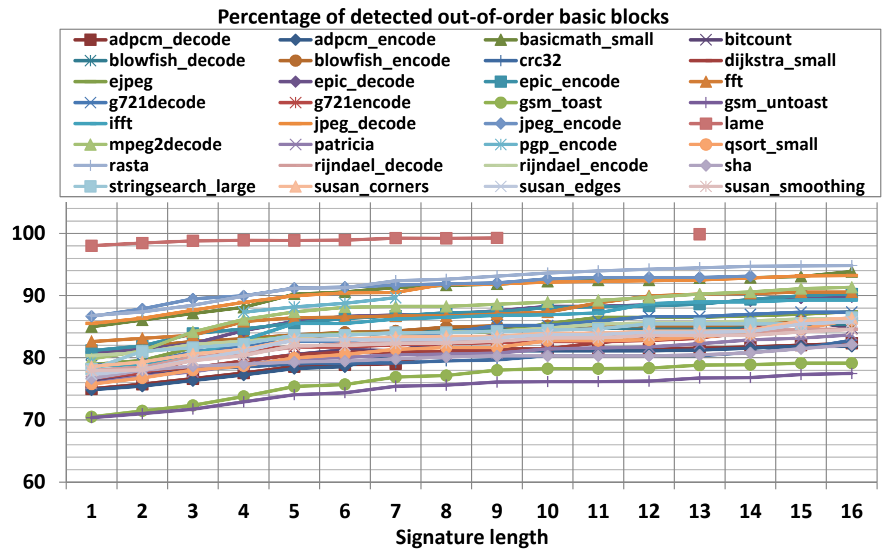


Figure 6.13: Percentage of out-of-order basic blocks[2] discovered using the classification algorithm from Section 4.4.1.

The share of out-of-order blocks in different benchmarks is shown in Fig. 6.13. As can be seen from the results, the percentage of out-of-order blocks slightly increases with the signature length. The reason for this is the fact that with larger signature lengths a larger amount of diverse basic block timings can be captured. Consequently, the chance that a basic block will be classified as out-of-order increases at larger signature lengths because more potentially overlapping basic blocks can be discovered. The employment of the memory access queue decreases the speed of context-aware compiled simulation, since additional efforts are required for reading and writing values into the queue as well as keeping the queue sorted at simulation run-time. I define the deterioration of simulation time (i.e. wall-clock time) as

$$w = \frac{t_{dr} - t_0}{t_0}, \tag{23}$$

where t_{dr} and t_0 is correspondingly a simulation time with and without queue-based reordering of memory accesses at simulation run-time. The deterioration of simulation time for each benchmark at different signature lengths is shown in Fig. 6.14. In addition, we can determine the mean deterioration of the simulation time over all benchmarks at signature length i as follows:

$$\overline{w_i} = \frac{1}{n} \sum_{j=0}^{n} |w_{i,j}|, \tag{24}$$

where $w_{i,j}$ is deterioration of the simulation time of j^{th} benchmark at signature length i, and n is the total amount of benchmarks. The mean deterioration is shown in Fig. 6.15.

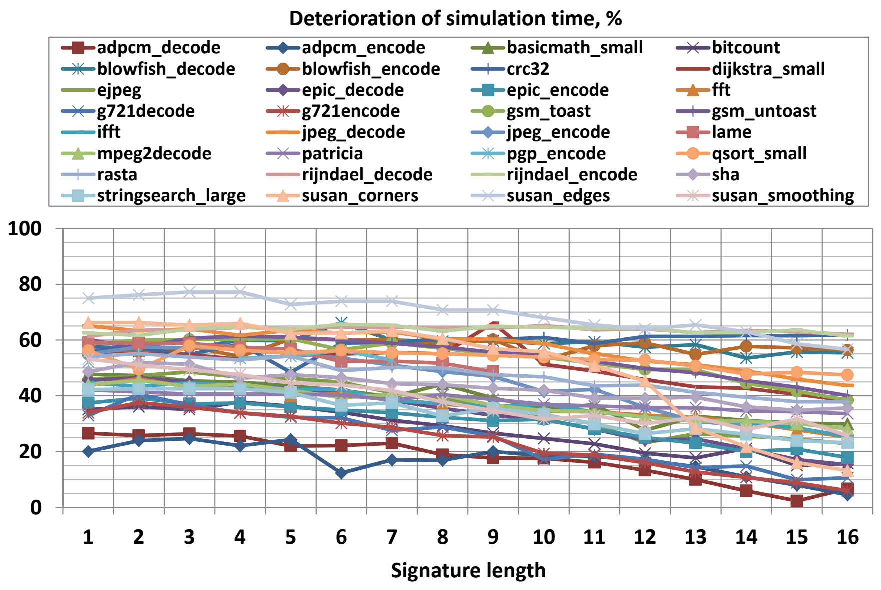


Figure 6.14: Deterioration of simulation time of context-aware host-compiled simulation due to dynamic reordering of memory accesses.

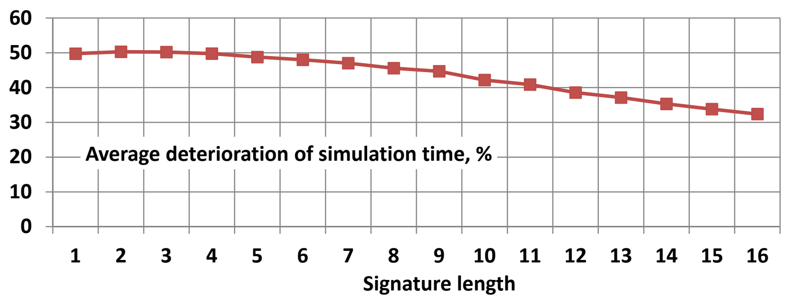


Figure 6.15: Mean deterioration of simulation time for all benchmarks at different context signature lengths.

As can be seen in the experimental results, at shorter signature lengths the deterioration does not significantly change and remains at the level of 50% for all benchmarks on average. However, at larger signatures, the deterioration slightly decreases to 30%. This is due to the fact that at larger signatures the simulation efforts for memory access reordering are getting relatively smaller compared to the efforts required for context-aware host-compiled simulation. Nevertheless, the simulation overhead of access reordering is still significant at all signature lengths. Addressing this issue, in Section 6.1.5.1 we will apply and evaluate an optimization technique based on static-memory reordering introduced in Section 4.6.3.2.

6.1.4 *Averaging of basic block timing*

The length of context signatures selected by the designer may not be sufficient for considering all possible deviations of a basic block's execution. In the previous section, I assumed that only first timing observed for a context in the measurement phase is captured and annotated in the translated code. In this section, I evaluate an optimization method, in which timings of a basic block observed within at a certain context are averaged and not dropped as assumed before. Details of this technique can be found in Section 4.6.2. For the following experiments, I use the same simulation setup and benchmarks as employed in previous Section 6.1.3.

6.1.4.1 *Estimation of SW execution time*

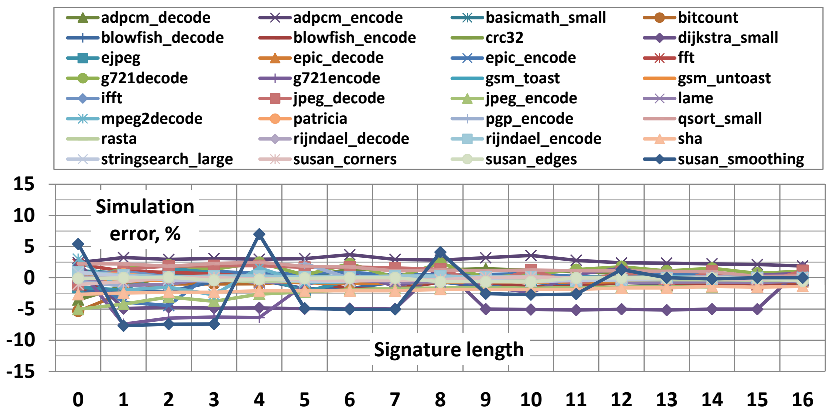


Figure 6.16: Timing error of context-aware host-compiled simulation with averaged basic block timing relative to the reference simulation in SimpleScalar.

The error of context-aware simulation with enabled averaging is presented in Fig. 6.16. The error is determined according to Eq. (17). Compared to the results without averaging shown in Fig. 6.6, the error of context-aware simulation can be significantly reduced with the proposed optimization technique. Notably, the simulation error is negative for many benchmarks at certain signature lengths. The underestimation of execution time is caused by rounding of annotated timing values to the next integer number according to Eq. (16). The timing values annotated in the basic block functions can be rounded in both directions. Consequently, the resulting rounding error accumulated over the complete simulation can be both positive and negative.

Fig. 6.16 shows the simulation result starting from a signature length of 0. At a zero signature length, all timings of a basic block observed in the measurement phase are averaged, and host-compiled simulation is performed without signatures. Surprisingly, averaging of basic

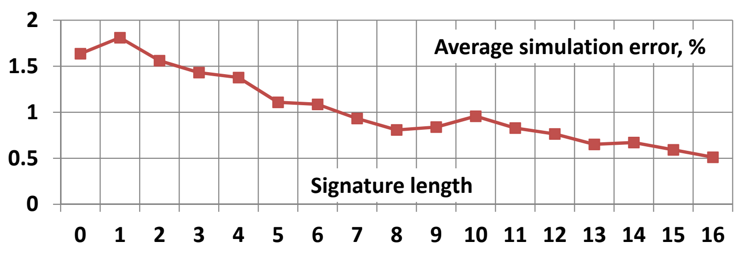


Figure 6.17: Mean timing error of context-aware host-compiled simulation with averaged basic block timing relative to the reference simulation in SimpleScalar.

block timing at signature length of 0 resulted in accurate estimations of the execution time as well. In general, the experiments showed that with averaging the simulation error does not significantly change at larger signature lengths. The largest error of -7.7% was observed for benchmark *susan_smoothing* at signature length of 1.

In addition, I determined the mean simulation error among all benchmarks using Eq. (18). The mean error of host-compiled simulation at different signature lengths is shown in Fig. 6.17. As can be seen in the figure, the mean error decreased from 1.6% at a length of 0 to 0.5% at a length of 16. Although the decreasing rate of the error is low, the use of signatures still improves the accuracy of timing estimation.

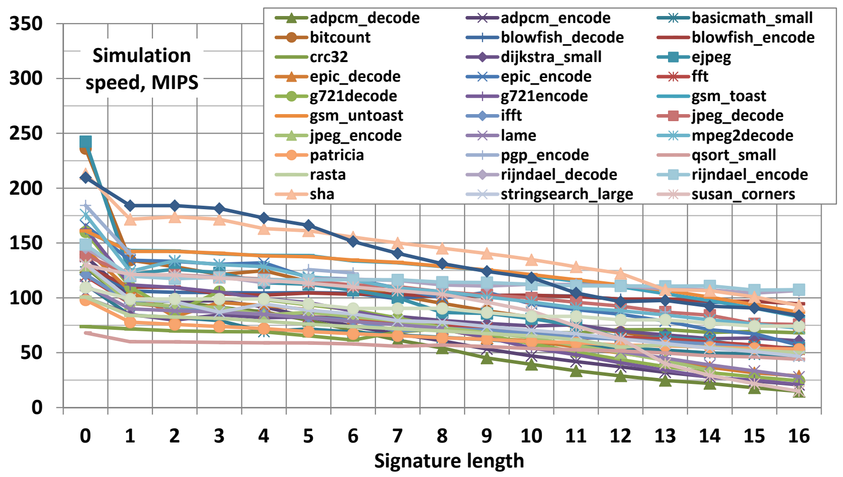


Figure 6.18: Speed of context-aware host-compiled simulation with averaged basic block timing expressed in millions of simulated target instructions per second.

The speed of context-aware host-compiled simulation with averaged basic block timing is shown in Fig. 6.18. For all benchmarks, the simulation speed gradually decreased with larger lengths of context signatures. Notably, the reduction of simulation speed from a signature length of 0 to a length of 1 is higher compared to the remaining

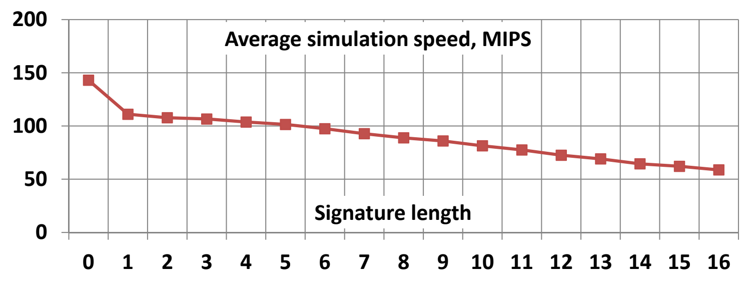


Figure 6.19: Mean speed of context-aware compiled simulation with averaged basic block timing.

part of the curves. This is due to the fact that there is no overhead in host-compiled simulation caused by signatures at a signature length of 0 . Particularly at a signature length of 0, the highest simulation speed of 242 MIPS was in *ejpeg* benchmark and the lowest speed of 68 MIPS was in *qsort_small* benchmark. On average, the simulation speed decreased from 143 MIPS at a length of 0 to 59 MIPS at a length of 16 as shown in Fig. 6.19.

6.1.4.2 *Order of memory accesses*

In addition to the estimation of execution time, the accuracy of compiled SW simulation is determined by the order of simulated memory accesses. In this section, I evaluate the impact of averaging on the error in the order of memory accesses. For this purpose, I employ the reorder depth metric[3] and calculate the RMSE relative to the reference simulation in SimpleScalar using Eq. (21). The resulting RMSE for different benchmarks at different signature lengths is shown in Fig. 6.20. As can be seen in the figure, averaging of basic block timing allows for significant reduction of the RMSE compared to the scenario without averaging (see Fig. 6.11). If no signatures are employed, i.e. the signature length is 0, the RMSE of reorder depths is still high for several benchmarks, e.g. for *crc32*, *gsm_toast*, *gsm_untoast* or *lame*. However, the error for these benchmarks could be already significantly reduced at a signature length of 1.

Furthermore, the experiments showed that the RMSE of reorder depths gradually decreases with the signature length. Fig. 6.21 shows the average RMSE values over multiple benchmarks at different signature lengths. Signature-based simulation with a signature length of 1 could reduce the average RMSE from 2.8 to 1.3. However, further increasing of the signature length did not have a significant impact on the RMSE. The value of RMSE reached 0.8 at a signature length of 16.

3 See Section 6.1.3.2 for further details

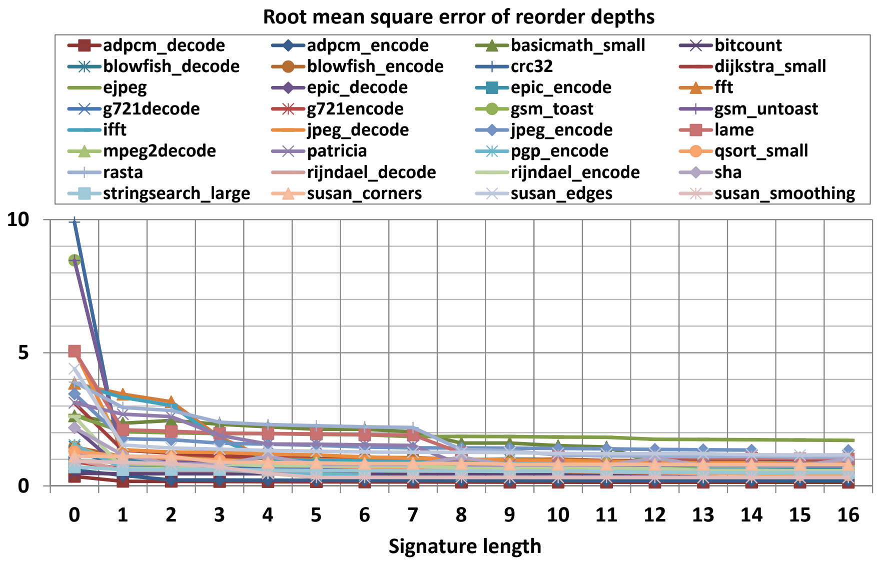


Figure 6.20: Root mean square error of reorder depths in context-aware compiled simulation with averaging compared to the reference SimpleScalar simulation.

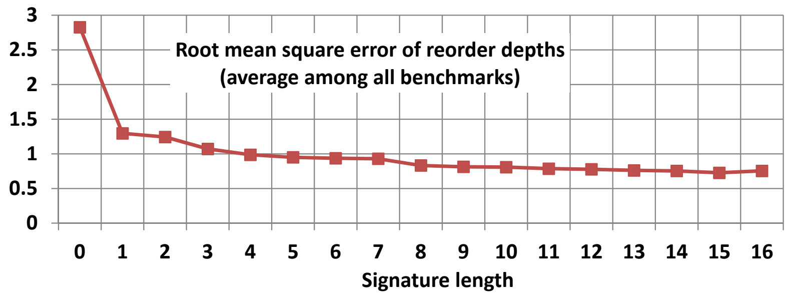


Figure 6.21: Root mean square error of reorder depths in context-aware compiled simulation averaged over all benchmarks

6.1.4.3 *Summary*

So far we have seen the impact of employing multiple context timings per basic block in host-compiled simulation under an assumption of perfect instruction and data caches. Consideration of multiple contexts per basic block allows for more accurate simulation of the target software. Moreover, the experiments showed that averaging of basic block timing can further reduce the simulation error. The impact of the averaging technique is large. Employment of context signatures for averaged timings still allows for a small accuracy improvement. However, already at a zero signature length, the timing error could be reduced to 1.6% (on average for all benchmarks), which is a reasonable accuracy for system-level simulations. Furthermore, simulations at a zero signature length are preferable in terms of simulation speed and offer an acceptable performance/accuracy ratio. Therefore, in the following experiments, I employ context-aware host-compiled simulation with averaged basic block timings assuming a zero signature length.

6.1.5 *Consideration of a data cache*

In this section, I evaluate context-aware host-compiled simulation while considering non-blocking behavior of a data cache. For this, I assume simulations with a realistic data cache and a perfect instruction cache as mentioned in Section 6.1.1.2. The configuration of the data cache used during simulations is shown in Table 4.

Table 4: Configuration of a data cache employed in simulations

Size	4 kByte
Line size	32 Byte
Associativity (ways)	2
Hit latency	1
Miss penalty	16
Amount of sub-blocks	0
Replacement policy	FIFO
Address translation	Virtually Indexed, Virtually Tagged (VIVT)
Pre-fetching	no
Number of miss status holding registers (MSHRs)	1
Number of pre-fetch MSHRs	1
Number of targets per MSHR	1

In addition, the data cache module requires a *non-blocking extension* to consider the non-blocking behavior. The purpose of this extension is to adjust the simulated time according to the rules specified in Section 4.5.1 and, thus, to reconstruct the timing behavior observed in the reference cycle-accurate simulation as accurate as possible. The simulation accuracy is assessed by comparing the results of host-compiled simulation and the results of the reference SimpleScalar simulation.

Before accessing the data cache module, it essential to obtain the desired order of memory accesses. In the following experiments, I investigate two scenarios shown in Fig. 6.22.

In the first scenario, simulation is performed with dynamic reordering of memory accesses based on the memory access queue (see Section 4.4.2 for the details on this method). The queue is employed to reconstruct the order of memory accesses captured in the measurement phase. In the second scenario, simulation is performed with statically reordered memory accesses as proposed in Section 4.6.3. By exchanging the order of memory accesses at compile time, I intention-

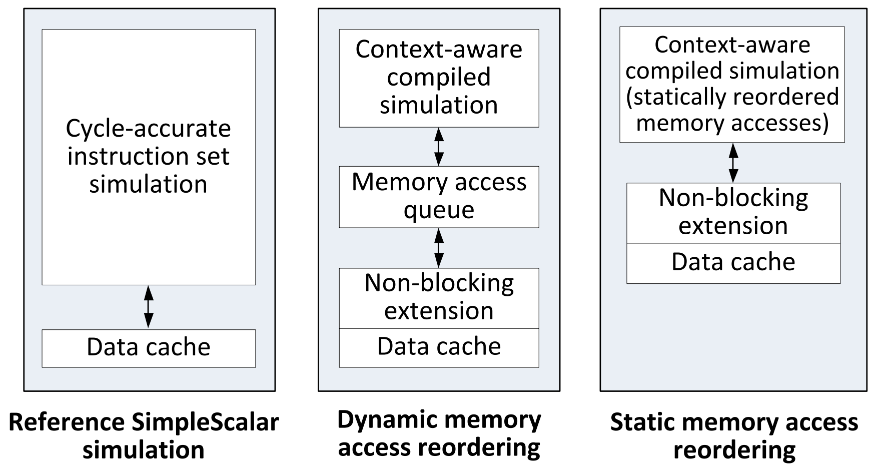

Cycle-accurate instruction set simulation
Data cache
Reference SimpleScalar simulation
Context-aware compiled simulation
Memory access queue
Non-blocking extension
Data cache
Dynamic memory access reordering
Context-aware compiled simulation (statically reordered memory accesses)
Non-blocking extension
Data cache
Static memory access reordering

Figure 6.22: Simulation considering non-blocking behavior of a data cache. Two method of performing compiled simulation are evaluated: with dynamic and static memory access reordering. The results of the both methods are then compared with the reference SimpleScalar simulation.

ally introduce an error in host-compiled simulation. However, in this way host-compiled simulation can be performed more efficiently as will be shown in the next section.

6.1.5.1 *Comparison of static and dynamic reordering*

The purpose of static reordering of memory accesses is to simplify context-aware host-compiled simulation and eliminate costly dynamic reordering based on the memory access queue. This technique aims at improving the simulation performance, while having only a marginal difference in the produced simulation results. As discussed in Section 4.6.3.1, not always the exchange of memory accesses performed out-of-order will result in a visible timing error. In Section 6.1.5.2, I will analyze in detail the pattern of memory accesses of different benchmarks in order to reveal the situations when such an error can occur. In this section, I compare dynamic and static reordering of memory accesses under consideration of a realistic data cache.

First, I compare estimated execution time produced by the both types of compiled simulation. The experimental results are shown in Fig. 6.23. The execution time is expressed as the amount of clock cycles required to execute the target code on the processing core. As can be seen in the figure, with static reordering, the estimated number of clock cycles was almost the same as in case of dynamic reordering. Both techniques produced results that are very close to the reference simulation in SimpleScalar.

In some cases, simulation with static reordering resulted in a lower cycle count compared to dynamic reordering, e.g. in *crc32*, *rijndael_de-*

code or *rijndael_encode* benchmarks. For some benchmarks, e.g. in *g721-decode*, *g721encode* or *rijndael_encode*, the estimated cycle count was higher. The reason for this inaccuracy was explained in Section 4.6.3.1. In fact, the static exchange of memory accesses may result in a positive as well as in a negative timing error. Thus, when accumulated over the complete simulation, these errors can result either in a small underestimation or overestimation of the total execution time.

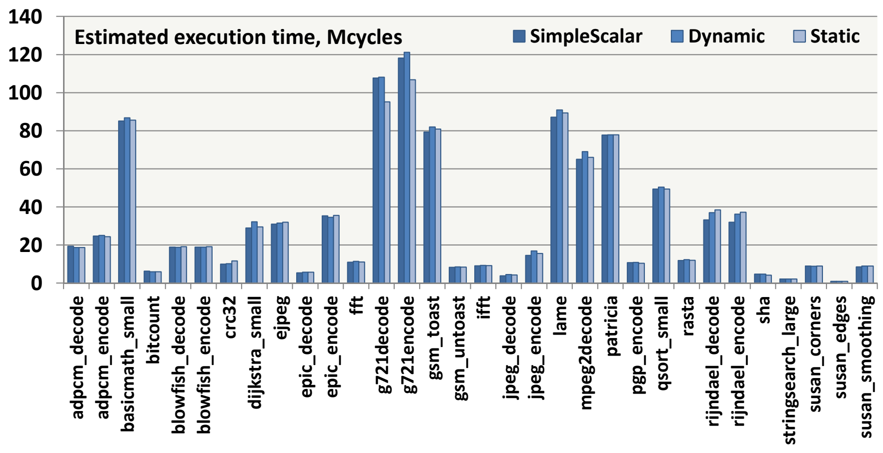


Figure 6.23: Comparison of dynamic or static reordering of memory accesses in context-aware host-compiled simulation compared to the reference SimpleScalar simulation, while assuming a realistic data cache.

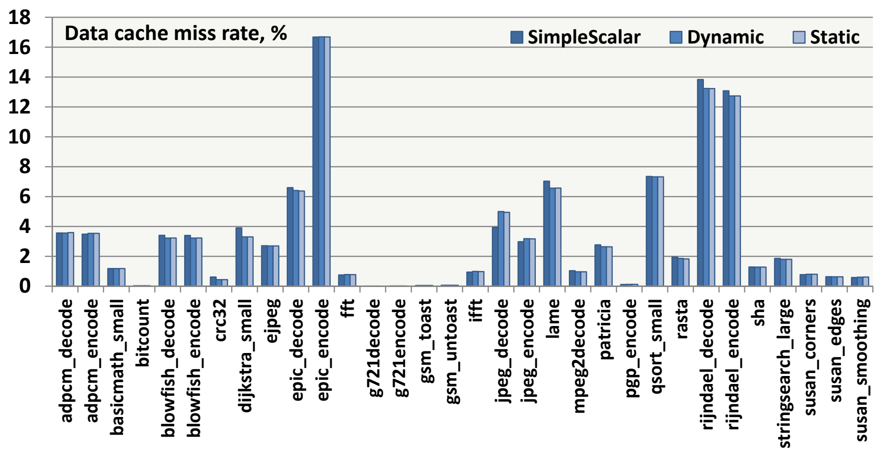


Figure 6.24: Comparison of data cache miss rates produced by the reference SimpleScalar simulation and context-aware host-compiled simulation with either dynamic or static reordering of memory accesses.

In addition to the execution time of the benchmarks, the accuracy of cache simulation has to be evaluated for the both methods. For this, we can compare the data cache miss rates produced by context-aware host-compiled simulation and the reference simulation in SimpleScalar. The experimental results are shown in Fig. 6.24. As can be seen in the figure, compiled simulation with static reordering produced identical miss rates of the data cache as simulation with dynamic reordering. Moreover, the results of the both simulation methods are very close to the reference SimpleScalar simulation.

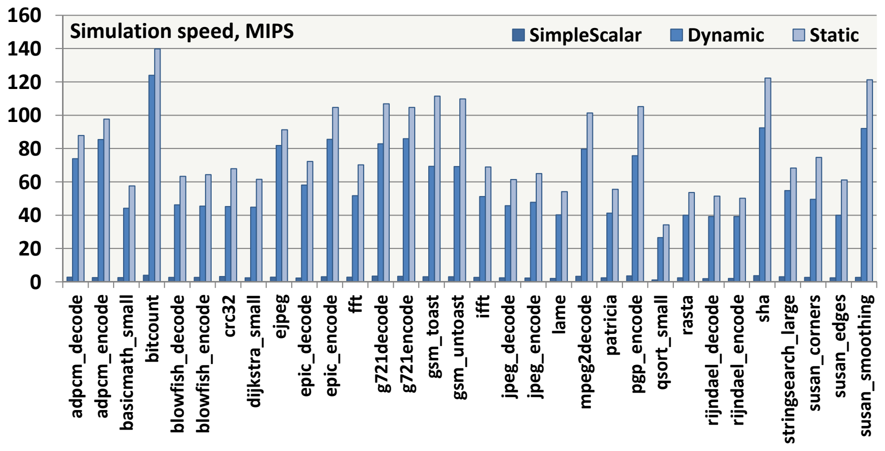


Figure 6.25: Comparison of speed of the reference SimpleScalar simulation and context-aware compiled simulation with dynamic and static reordering of memory accesses.

Finally, the speed of the both simulation methods is shown in Fig. 6.25. As can be seen in the figure, context-aware host-compiled simulation with static reordering has indisputable advantage over the simulation with dynamic reordering in terms of simulation speed. The average speed of compiled simulation with static reordering was 80 MIPS versus 60.9 MIPS with dynamic reordering. Thus, due to the elimination of the memory access queue, the speed of context-aware compiled simulation could be improved by 31% with the proposed optimization at almost no deterioration of simulation accuracy. The average speed of cycle-accurate simulation in SimpleScalar was only 2.7 MIPS, which is almost 30 times slower than the average speed of compiled simulation with static reordering.

6.1.5.2 *Analysis of memory accesses*

In this section, we will see why static reordering of memory accesses provides almost the same level of accuracy as dynamic reordering based on the queue. Reordering of memory accesses is strongly correlated with the structure of the binary code and, hence, is different for each of the target applications. In addition, the way in which memory access are reordered is also specific for the target processing core.

First, let us quantitatively analyze the degree of memory accesses reordering in an out-of-order processing core for different benchmarks. Fig. 6.26 shows the percentage of memory accesses at a certain reorder depth. As can be seen in the figure, in all benchmarks the majority of memory accesses have a reorder depth of 0, i.e. most memory instructions were performed in the program order. The remaining memory accesses were performed out-of-order at a reorder depth ranging from 1 up to 46. Notably, the share of memory accesses having large reorder depths is relatively small.

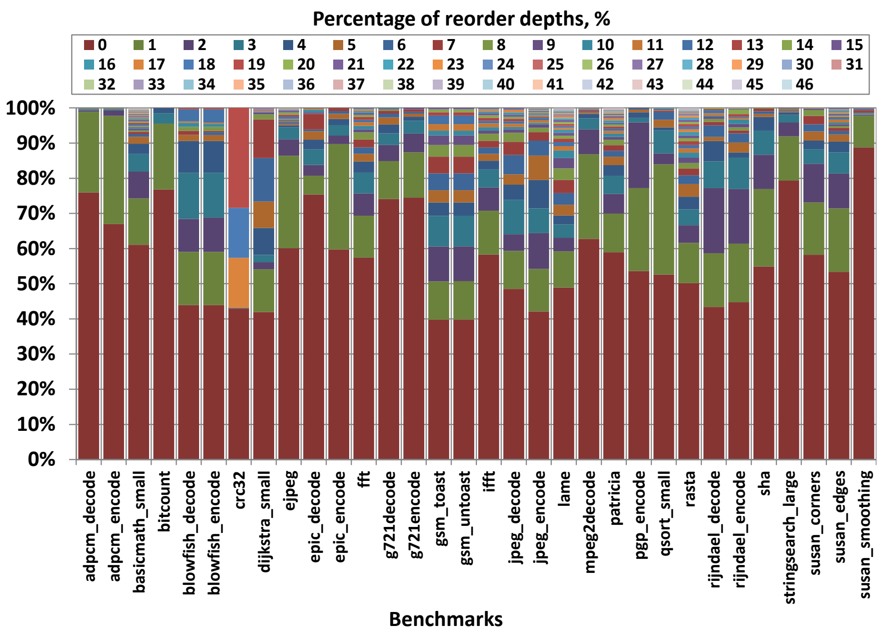


Figure 6.26: Percentage of reorder depths of memory accesses observed in the reference SimpleScalar simulation for multiple benchmarks.

For a better representation, let us additionally evaluate the percentage of reorder depths averaged over all benchmarks. For this, we can add all memory accesses having a certain depth among all benchmarks and determined their share in the total sum of memory accesses performed by all benchmarks. The results of this evaluation are presented in Fig. 6.27. On average, 56.5% of all memory accesses were performed in the program order. The majority of our-of-order memory accesses (14% of all accesses) had a reorder depth of 1. The share of accesses with reorder depth of 2 was only 6%.

In Section 4.6.3.1, I performed analysis of possible timing errors due to static reordering for memory accesses with a reorder depth of 1. It was identified that the resulting timing error is hidden due to spatial locality of data in the cache. Furthermore, it was identified that exchanging the position of an out-of-order cache access with a reorder depth of 1 will not result in a timing error in the following situations:

1. If the target addresses of the exchanged accesses are located in the same cache line. In this situation, it is irrelevant whether these accesses result in a cache hit or a miss. In both cases, there will be no timing error.

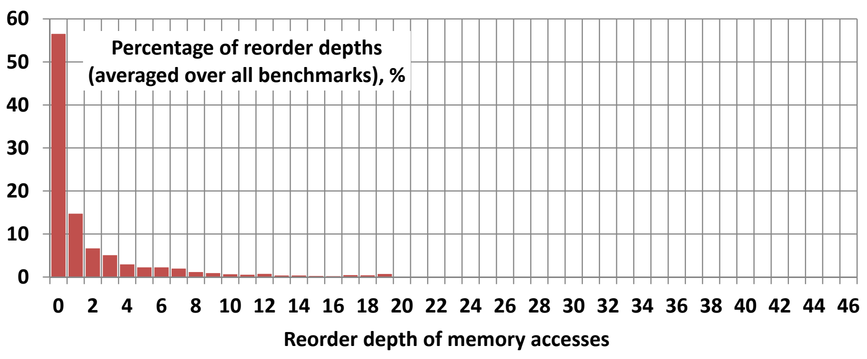


Figure 6.27: Percentage of reorder depths of memory accesses observed in the reference SimpleScalar simulation averaged over multiple benchmarks.

2. When the target addresses are located in different cache lines, however, both of the accesses result in a cache hit.

3. If the target addresses are located in different cache lines, both of the accesses are independent and result in a miss, and the following third memory access is also independent.

In other situations, changing the order of cache accesses may *potentially* but not necessarily result in an error. In fact, the timing error due to static reordering depends on the rate at which the above conditions occur. I performed analysis of all data cache accesses with a reorder depth of 1 in the reference simulation to identify how often such conditions occur during the benchmarks' execution. The respective evaluation results for all benchmarks are shown in Fig. 6.28.

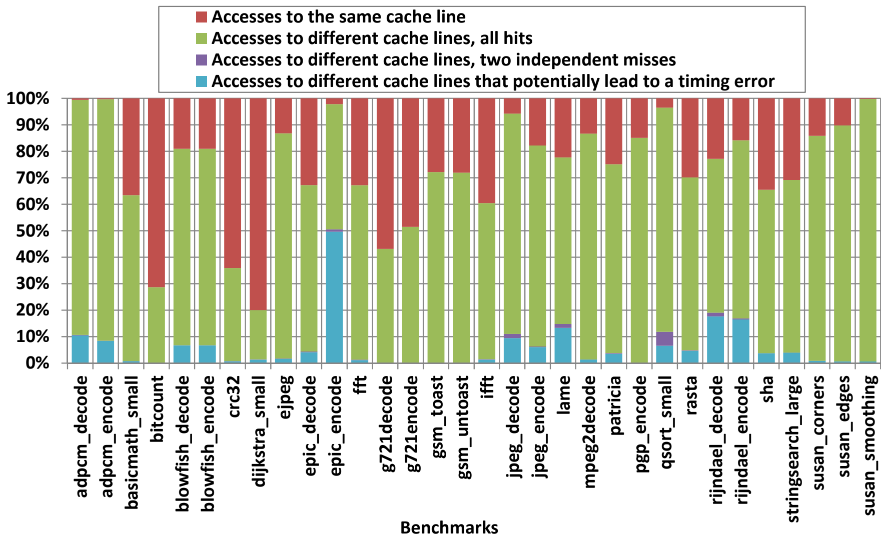


Figure 6.28: Evaluation of cache accesses with a reorder depth of 1 for different benchmarks.

As can be seen from the experimental results, the majority of accesses to the data cache were performed either to the same cache line or to different lines with two consecutive hits. The percentage of two independent misses was very low almost for all benchmarks. The share of accesses potentially leading to an error was relatively low for all benchmarks, with an exception of *epic_encode* benchmark. This benchmark had the largest data cache miss rate of 16.7% among all the benchmarks. Consequently, the occurrence of two consecutive dependent misses is more probable for this benchmark. Taking these experimental results into account, it can be anticipated that the probability of a timing error caused static reordering is also small for memory accesses with a reorder depth larger than 1. This hypothesis was tested by the experiments in the previous section.

6.1.5.3 *Summary*

In this section, accuracy of context-aware compiled simulation under consideration of a realistic data cache was investigated. In particular, we evaluated the efficiency of static reordering of memory accesses. The experimental results showed that simulation with static reordering can estimate the execution time and data cache miss rates almost as accurate as simulation in which memory access are reordered dynamically based on a memory queue. At the same time, with static memory reordering, the simulation speed could be increased by 31%. Thus, simulation with static reordering is more efficient in terms of the speed/accuracy ratio.

6.2 SYSTEM-LEVEL SIMULATION BASED ON SYSTEMC

In this part of the chapter, I evaluate and compare binary-level host-compiled simulation (BLS) and trace-driven simulation (TDS) methods at the system level. Both of these methods are employed to reconstruct the timing behavior of out-of-order cores in the scope of a complete system-on-chip.

6.2.1 *Experimental setup*

In contrast to Section 6.1, in this part the simulation methods are evaluated in a SystemC environment. To enable this, the target software is represented as either BLS or TDS tasks. Moreover, simulation employs the SystemC models of out-of-order cores, caches and arbitrated bus and main memory introduced in Section 5.3. In this section, it is assumed that both instruction and data caches are realistic. The use of SystemC environment allows concurrent simulation of multiple cores. However, at the same time, SystemC introduces additional overhead due to its simulation kernel.

For the following experiments, I assume the same configuration of an out-of-order processing core as specified in Table 3 and the same benchmarks as described in Section 6.1.1.1. Furthermore, I assume that instruction and data caches have a similar configuration as described in Table 4. The communication latencies, including both the memory access time and data transfer time, are equal to 16 clock cycles without contentions on the bus. In case of a bus contention, the communication latencies are correspondingly increased depending on the current utilization of the bus.

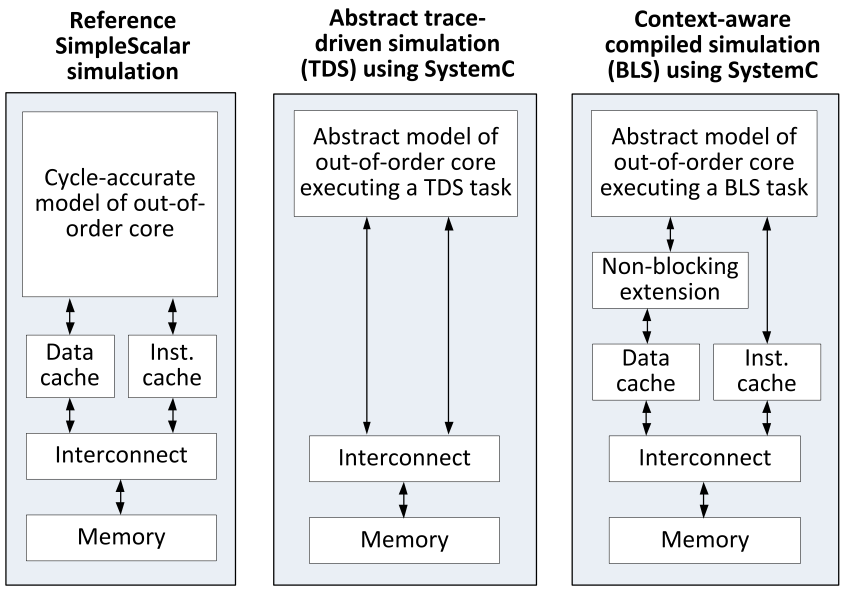

Reference SimpleScalar simulation
Cycle-accurate model of out-of-order core
Data cache
Inst. cache
Interconnect
Memory
Abstract trace-driven simulation (TDS) using SystemC
Abstract model of out-of-order core executing a TDS task
Interconnect
Memory
Context-aware compiled simulation (BLS) using SystemC
Abstract model of out-of-order core executing a BLS task
Non-blocking extension
Data cache
Inst. cache
Interconnect
Memory

Figure 6.29: Experimental setup for evaluating BLS and TDS methods in a SystemC environment. The evaluation is performed using multiple benchmarks, which are simulated as standalone applications in the scope of a complete system-on-chip as either BLS or TDS task. The simulation results are then compared to the reference simulation of the same benchmarks in SimpleScalar.

In the next section, I evaluate and compare TDS and BLS methods (see Fig. 6.29). TDS method employs highly abstracted execution traces, which specify the sequence of bus accesses as well as the time intervals between them. The traces are derived using the reference cycle-accurate simulation in SimpleScalar and then simulated in SystemC in the form of a TDS task. The traces completely abstract the internal microarchitecture of the processing core (including local caches). Therefore, no dynamic models of caches are needed in TDS, since the bus requests captured in the trace stimulate the model of the interconnect directly. Further details on the TDS workflow can be found in Section 5.1.1.

In case of BLS (right part of Fig. 6.29), I employ context-aware host-compiled simulation with all enabled optimizations investigated in the previous section. Particularly, it is context-aware simulation with a zero signature length, with enabled averaging of basic block timings and with static reordering of memory accesses. Compiled simulation is performed in the form of a BLS task. This method requires dynamic models of instruction/data caches as well as the non-blocking data cache extension for considering the non-blocking cache behavior at simulation run-time.

In the next step, the TDS and BLS results are compared with the reference cycle-accurate simulation in SimpleScalar. The intention of the TDS and BLS methods is to reproduce the timing behavior of benchmarks on the reference SimpleScalar simulator as accurately and as fast as possible. For comparability reasons, the configuration of instruction/data caches (for BLS) and the parameters of the interconnect and main memory are same. In the following experiments, independent buses for instructions and data are considered.

6.2.2 *Evaluation of BLS and TDS methods*

The speed of BLS and TDS methods in SystemC environment for different benchmarks is shown in Fig. 6.30. The largest speed of BLS was measured during the simulation of *bitcount* benchmark (72.5 MIPS), while the lowest speed of 9.0 MIPS was measured for *rijndael_decode*. The average BLS speed over all benchmarks was 31.8 MIPS. In all benchmarks, the BLS method was slower compared to TDS because of the several reasons. First, BLS requires co-simulation of caches models as well as non-blocking data cache extension. These components slow down the simulation performance. Moreover, BLS is fully functional simulation based on the execution of the translated target code. The speed of the reference simulation in SimpleScalar was also different for the benchmarks and ranged from 1.1 MIPS to 3.8 MIPS with the average value of 2.5 MIPS. Please note that the reported speed of BLS includes the overhead of the SystemC simulation kernel, which is required for simulation of multicore architectures. Thus, in SystemC environment BLS achieves an average speedup of 12.7 times[4]. At the same time, this speedup does not consider the overhead of one-time derivation of the execution delays in the reference simulation. In fact, this overhead spreads over multiple binary-level compiled simulations performed during system-level design space exploration. In [60], it was shown that at 10000 iterations the corresponding overhead can be neglected.

In contrast to BLS, TDS completely abstracts the functionality of the target software and the core's internal microarchitecture including data and instruction caches. The largest speed of TDS was measured

4 In contrast, SimpleScalar simulator was written in C and designed for evaluation of single-core processors only. If BLS is also implemented in C (i.e. without calling SystemC functions), its average speedup compared to SimpleScalar is approximately 25 times.

for *susan_smoothing* benchmark (623.9 MIPS), while the lowest speed of 13.6 MIPS was measured for *rijndael_encode* benchmark. Over all benchmarks, the average simulation speed of TDS in SystemC environment was approximately 169.3 MIPS, resulting in a speedup of 67 compared to SimpleScalar. Similarly to BLS, this speedup does not consider the efforts of generating traces in the reference cycle-accurate simulator. The associated overhead is spread over multiple trace-driven simulations performed during design space exploration. As can be seen in the figure, the speed is not permanent but depends on a benchmark being simulated. The reason for the diverse values of simulation speed is the fact that the benchmarks impose different loads on the interconnect. Higher access rates on the instruction and data buses result in more frequent calls to SystemC functions, e.g. `sc_wait()`, and more frequent stimulation of the interconnect model. Consequently, the overhead of SystemC simulation increases and the overall simulation performance deteriorates.

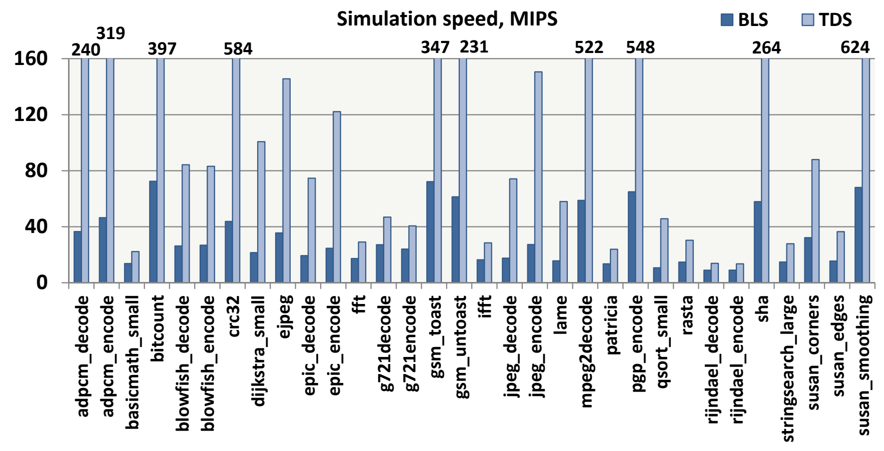


Figure 6.30: Simulation speed of BLS and TDS expressed in millions of target instructions simulated per one second of real time (MIPS).

The timing error of TDS and BLS methods compared to the reference simulation in SimpleScalar is shown in Fig. 6.31. TDS is a very abstract simulation approach, in which data read operations are always considered blocking. In TDS, communication latencies are simply added to the computational latencies and, as a result, TDS always overestimates the execution time. In some benchmarks, the timing error of TDS is larger than 15%, e.g. in *blowfish_decode*, *blowfish_encode*, *jpeg_decode* or *rijndael_decode* benchmarks. In turn, BLS method considers the effects of out-of-order execution and, therefore, achieves better simulation accuracy on average. For example, for *jpeg_decode* benchmark, BLS showed a considerably smaller error of 4.2% (BLS) compared to 16.5% in TDS. With the exception of *sha* benchmark, in which the execution time was notably underestimated, BLS achieves an average error of 2.6% in contrast to 6.7% in TDS.

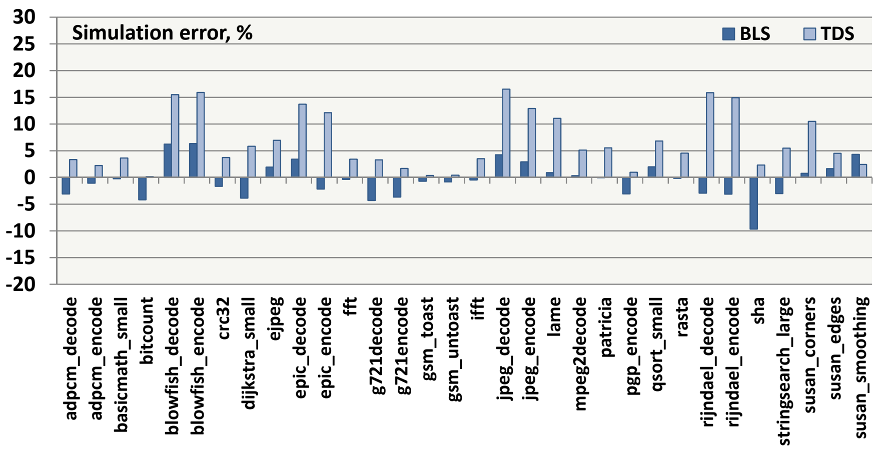


Figure 6.31: Timing error of BLS and TDS compared to the reference SimpleScalar simulation.

Although BLS achieves better accuracy, it is not always the best simulation method in terms of the speed-accuracy ratio. For many benchmarks, e.g. *adpcm* (decode and decode), *bitcount*, *crc32*, *gsm* (toast and untoast), *gsm_untoast*, *pgp_encode*, *sha* and *susan_smoothing*, TDS could produce accurate results as well (the timing error for these benchmarks is below 5%). At the same time, compared to BLS, the TDS method could achieve significantly higher simulation speed for these benchmarks. The reason why the TDS method can be both accurate and fast can be explained by analyzing the amount of bus accesses performed during the simulation of a benchmark. Fig. 6.32 shows the rate of interconnect accesses per target instruction for all simulated benchmarks. This rate is defined as the total amount of accesses to the data and instruction buses divided by the total number of simulated instructions in the respective benchmarks.

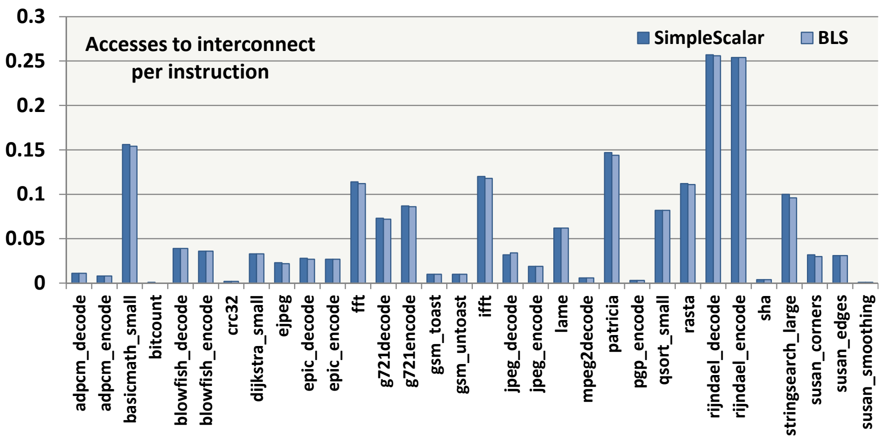


Figure 6.32: Rate of accesses to on-chip interconnect, which is the number of line fills and line write-backs in the instruction and data caches, per simulated target instruction.

As can be seen in the figure, the benchmarks mentioned above have a very low access rate. The reason for this is the low miss rates in instruction and data caches. Because of the low cache miss rates, the execution latencies captured by traces abstract larger portions of the executed code. Thus, the efficiency of the trace abstraction increases, resulting in a higher speed in terms of simulated target instructions per second. Moreover, since the amount of bus accesses is low, the sum of the timing errors produced during these bus accesses is also low relative to the overall execution time of the benchmark. Therefore, for these benchmarks, TDS achieves both high accuracy and high simulation speed and, therefore, is more efficient than BLS. However, it is important to mention the use of TDS may be limited because of its abstraction level. For example, TDS abstracts the behavior of caches and, therefore, this method cannot be employed in a scenario when the cache lines in the simulated core can be invalidated by other cores.

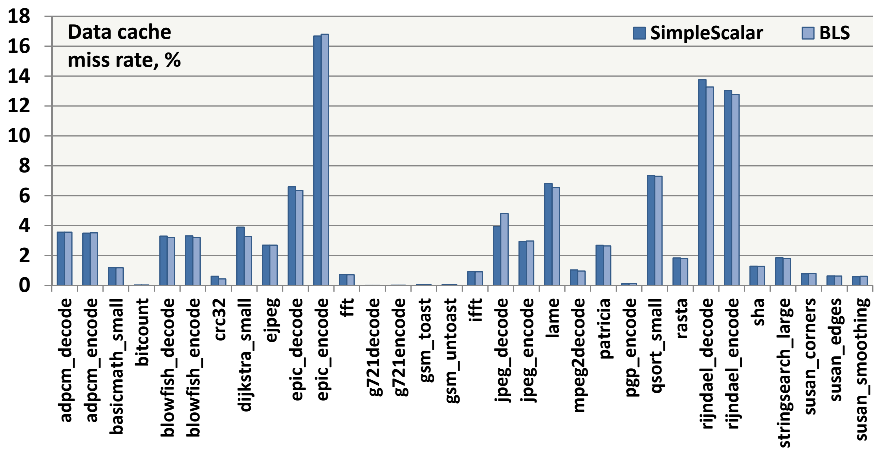


Figure 6.33: Estimation of the data cache miss rate in BLS. The results are compared with the reference simulation in SimpleScalar.

The BLS method involves the simulation of local caches. Therefore, it is essential to evaluate the accuracy of estimated cache miss rates. The experimental results are shown in Fig. 6.33 and Fig. 6.34. As can be seen in the figures, the BLS method achieves very close estimation results compared to the reference simulation in SimpleScalar for all benchmarks.

6.2.2.1 *Validation of BLS with random communication latencies*

Out-of-order processing cores exhibit very complex timing behavior because of the capability of hiding long memory access latencies. BLS considers the out-of-order capabilities of cores and allows for more accurate reconstruction of their timing behavior in the presence of data cache misses. So far, we have tested the BLS approach assuming single-core architectures and fixed memory access latencies. However, in case of a multicore architecture, the communication latencies do

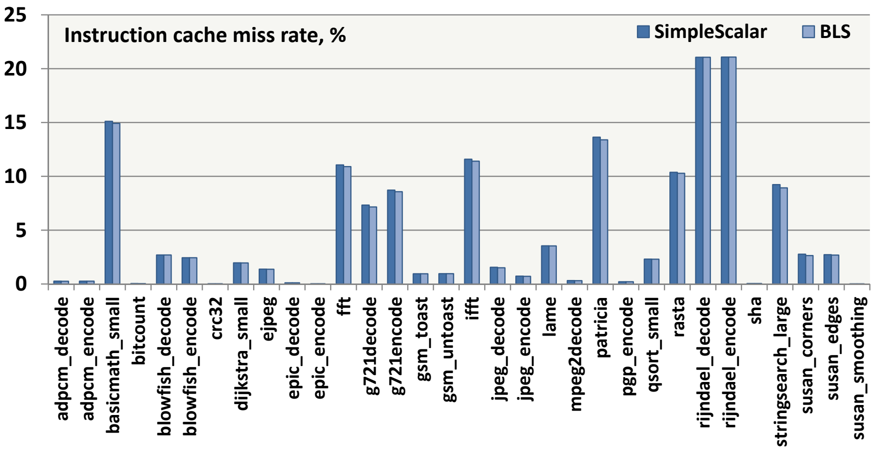


Figure 6.34: Estimation of the instruction cache miss rate in BLS. The results are compared with the reference simulation in SimpleScalar.

not remain constant but depend on the activity of other cores. The aim of this section is to validate the BLS approach assuming a multi-core environment, in which communication latencies can alternate.

SimpleScalar tool was designed for simulation of single-processor systems only and cannot be extended for multiprocessor architectures easily. Nevertheless, this section presents a simple and efficient solution based on artificial random traffic that causes alternating communication latencies. From the timing perspective, for each processing core the presence of other cores results in increased communication latencies due to contentions on the shared bus. Thus, the artificial traffic can be additionally simulated on the communication bus in order to mimic the activity on the bus with multiple cores. The artificial traffic is highly abstracted and does not represent any particular data. Its purpose is to increase the percentage of time when the bus is considered as occupied by other masters. If the artificial traffic can be equally reproduced both in SimpleScalar and BLS simulation, we can compare the results and assess the accuracy of the BLS method. The advantage of this approach is the capability of validating the BLS accuracy in different load scenarios, since the artificial traffic can be changed easily.

For this purpose, the bus models in SimpleScalar and BLS were extended to enable artificial traffic. An example of simulation with the extended bus model is shown in Fig. 6.35. At time t_1, the core model simulating the execution of real target software makes a communication request on the bus. Starting from this moment, the bus is considered as occupied. After a certain time, which includes a data transfer latency and memory access latency (labeled as *service* time in the figure), the transaction is completed and the bus changes back to the idle state. At time t_2, the core model makes the second request. During the service time of the second request, a dummy bus master

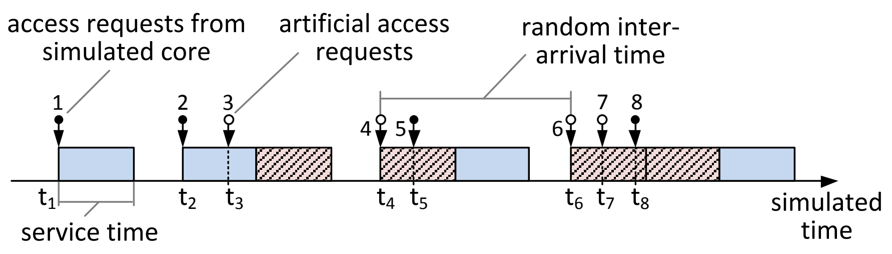

access requests from simulated core
artificial access requests
random inter-arrival time
1
2
3
4
5
6
7
8
t1
t2
t3
t4
t5
t6
t7
t8
service time
simulated time

Figure 6.35: Simulation of random bus traffic in SimpleScalar and BLS. The bus requests of the simulated core are interleaved with synthetic access requests made by dummy bus masters. The inter-arrival time of the synthetic requests is a random Poisson process.

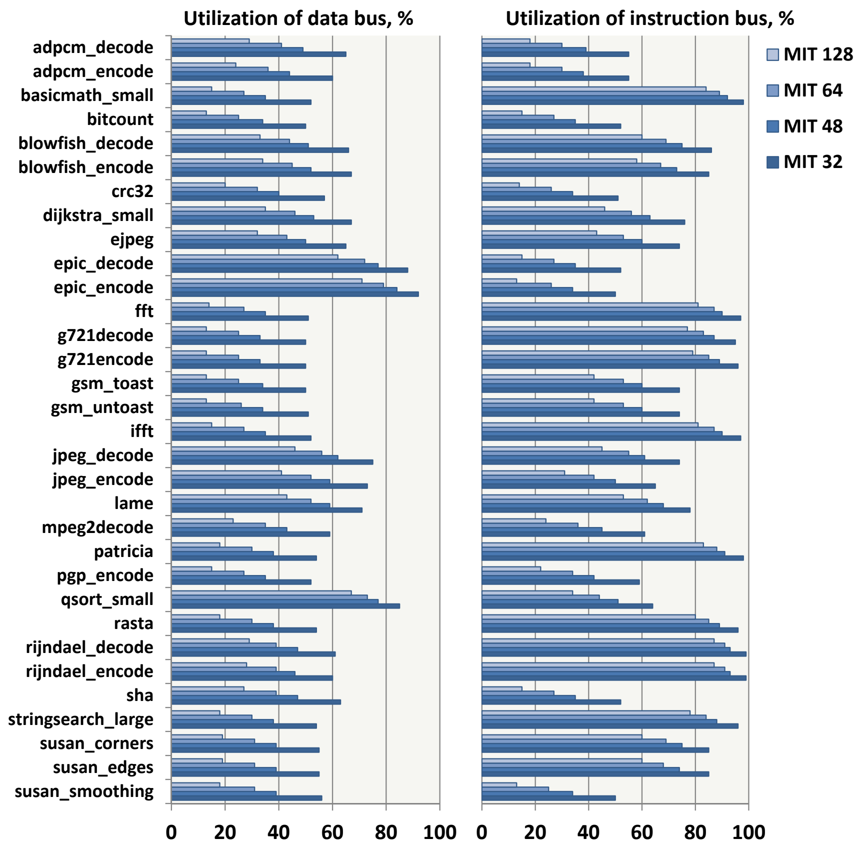


Figure 6.36: Utilization of data and instruction bus in SimpleScalar under consideration of synthetic access requests with a mean inter-arrival time (MIT) of 128, 64, 48 and 32 bus cycles.

makes the third request on the bus. However, since the bus is occupied, this request can be processed only after the second request is completed. The service time of the synthetic request is denoted by a striped rectangle. At time t_4, the dummy master makes another request followed by the request of the simulated core. In this situation, the simulated core has to wait till the synthetic request is completed. Finally, at times t_6 and t_7, two synthetic requests precede the request of the simulated core at time t_8. In this case, the simulated core has to wait till the service time of the first two requests is completed. Thus, it is assumed that the simulated core has a low priority on the bus.

The time interval between any two consecutive synthetic requests is a Poisson random variable. The expected value of the variable representing the mean inter-arrival time (MIT) between two synthetic requests can be modified by the user. By decreasing the expected value, the amount of synthetic requests within a certain time interval and, thus, the load on the bus is increased. The use of artificial traffic allows achieving various utilization of the data and instruction bus as shown in Fig. 6.36. Here, four values of mean inter-arrival time are selected: 128, 64, 48 and 32. The presented results were obtained in SimpleScalar assuming a service time of 16 bus cycles for the simulated core and for synthetic requests.

The execution time of all benchmarks in SimpleScalar with the artificial bus traffic is shown in the left part of Fig. 6.37. For each benchmark and for each MIT value, 8 simulations were performed with different artificial traffic by using different seeds in the random generator. The estimated execution time shown in the figure represents the average time over these 8 simulations. As expected, with smaller values of MIT, the overall bus utilization is increased, resulting in larger execution time of the benchmarks. Notably, for some benchmarks, e.g. *adpcm_decode*, *adpcm_encode* or *crc32*, the estimated time did not significantly change. This is due to the fact that these benchmarks impose small miss rates in the data and instruction caches. Consequently, memory access latencies do not have a significant impact on the execution time. In contrast, the execution time of *basicmath_small*, *g721decode* or *g721encode* benchmarks increased significantly at smaller values of MIT.

In the next step, the benchmarks were simulated using BLS. For this simulations, identical configuration was employed as in SimpleScalar, including the parameters of the instruction and data buses, MIT values and seeds of the random generator. Thus, exactly the same artificial bus traffic could be reproduced in BLS as in SimpleScalar. Similarly to SimpleScalar, 8 simulations per benchmark and MIT value were performed. The resulting average execution time was then compared with SimpleScalar. The timing error of BLS at different MIT values is shown in the right part of Fig. 6.37. Remarkably, for many benchmarks, e.g. *dijkstra_small*, *epic_decode* or *epic_encode*, the timing error decreased with higher bus utilization. This is because the absolute timing error in BLS remains constant. If the communication latencies increase, the percentage of this error in the total execution time of the benchmark decreases. As a result, the overall timing error of BLS decreases as well. In general, the BLS timing error did not significantly changed at different artificial bus traffics. Thus, BLS achieves accurate results with alternating communication latencies as well.

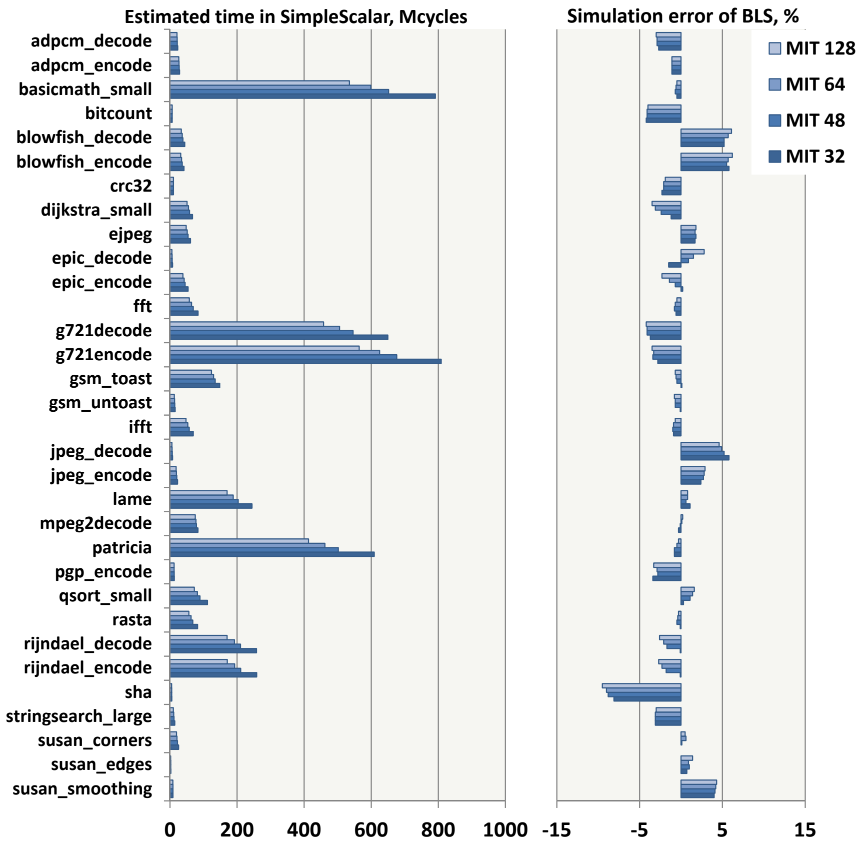


Figure 6.37: *Left:* Estimated execution time of benchmarks in SimpleScalar under consideration of synthetic access requests with different mean inter-arrival time. *Right:* Timing error of BLS relative to the reference SimpleScalar simulation under the same conditions.

6.2.2.2 *Scalability*

This section presents evaluation of the scalability of the BLS approach[5] during simulation of multicore architectures. In the following experiment, an architecture with multiple identical cores attached to a shared bus is simulated. All cores execute the same instance of *epic_encode* benchmark that has the highest data cache miss rate among all benchmarks (see Fig. 6.33). Two scenarios are considered: with coherent and non-coherent data caches.

The speed of the BLS approach changes with a number of cores as shown in Fig. 6.38. The speed is determined as the total number of target instructions simulated in the multicore architecture divided by the wall-clock time of the simulation. As can be seen in the figure, the simulation speed decreases with the number of simulated cores, from 23,1 MIPS in a single-core architecture to 17,4 MIPS in an architecture with 128 cores. With coherent caches, the speed of

5 The scalability of BLS is evaluated only since this simulation technique requires more computational resources on the host computer compared to TDS.

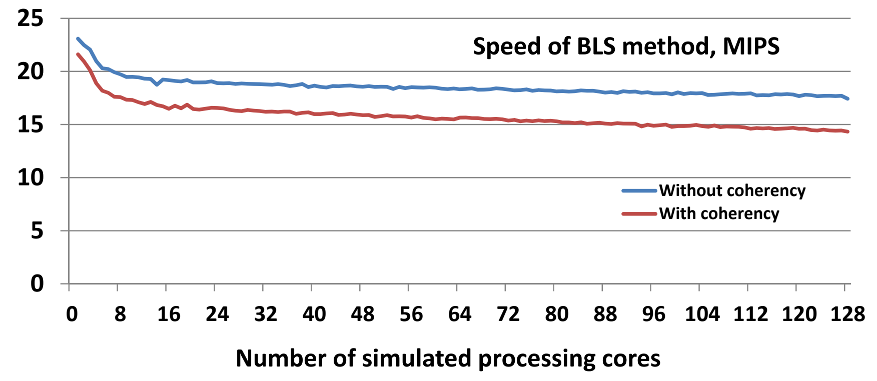


Figure 6.38: Speed of the BLS method during the simulation of a multicore architecture (*epic_encode* benchmark).

BLS decreases from 21,6 MIPS to 14.3 MIPS. The simulation speed does not remain constant because of the increasing efforts for simulating the shared interconnect, as the amount of simulation events due to higher contention on the bus increases. Nevertheless, the experimental results show that the BLS approach can be still applied for evaluation of multicore architectures with a large number of cores.

6.3 multicore design space exploration: a use case

In this section, I demonstrate an example of design space exploration of a JPEG encoding application. The application consists of multiple tasks and some of the tasks can be executed in parallel. The aim of the exploration is to evaluate the execution time of different task mappings on the underlying multicore platform by means of system-level simulations based on BLS.

6.3.1 *Experimental setup*

For the design space exploration, a target application that performs encoding of bitmap images into the JPEG format was selected. The application is based on the embedded JPEG Codec library [92]. The JPEG encoding algorithm consists of the following operations. Firstly, a macroblock of 16×16 pixels is read from the input bitmap file. Afterwards, the color space of the macroblock is converted from RGB to YCbCr and represented as four Y blocks (luminance components), one Cr and one Cb block (color-difference components) with a size of 4×4 pixels. In the next step, discrete cosine transform (DCT) is performed for each of the 6 blocks. The values of the block elements are then quantized and reordered. Finally, the blocks are encoded using Huffman coding and the result is written to the output JPEG file.

In the original form, the embedded JPEG Codec application consisted of a monolithic code which performs encoding of macroblocks sequentially. For the exploration purposes, the application was decomposed into a larger number of dependent tasks. The tasks were defined using the function boundaries in the target code. Moreover, to represent the application in the multi-tasking form, a number of small modifications in the original source code had to be performed. Firstly, the function interfaces were adapted to ensure that data is transferred to the functions via global variables and not by using function arguments. Thus, after being encapsulated in tasks, the functions can communicate with each other while executing on different cores.

Secondly, the main execution loop of the application was changed such that the functions constituting tasks are called sequentially in the target code. In this way, the execution of task functions can be clearly separated in the measurement phase and the execution time of basic blocks belonging to different tasks can be obtained independently of each other. During the generation of the translated code, the code of each task function was encapsulated into a separate BLS task.

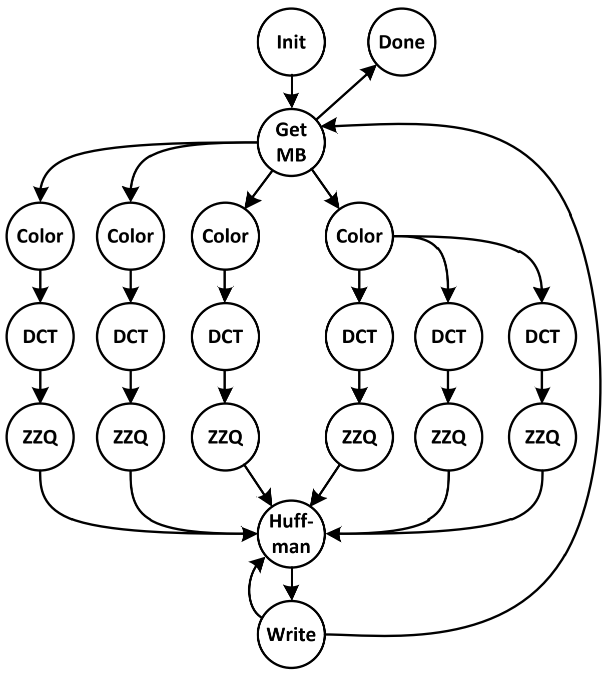


Figure 6.39: Task graph of the modified JPEG encoding application used for design space exploration.

The resulting task graph of the modified application is shown in Fig. 6.39. *Init* task opens the input bitmap and output JPEG file and analyzes the header of the bitmap image. The remaining execution of the application is organized in a loop. *GetMB* task reads a macroblock from the input bitmap file. The four *Color* tasks perform conversion of the RGB space of a respective part of the macroblock into YCbCr space. The computational results of these task are stored in 6 blocks

(4 luminance and 2 color-difference components). *DCT* task applies discrete cosine transform to each of the 6 blocks. *ZZQ* task performs quantization and reordering of the block elements. When the first block is ready, the *Huffman* task encodes of the resulting data and *Write* task stores the data in the output JPEG file. When encoding and storing of all 6 blocks is completed, *GetMB* task reads the next macroblock from the input bitmap file. When the input bitmap file is completely processed, final task *Done* writes the concluding data into the output JPEG file and closes the files.

In total, 8 different tasks are required to encode a bitmap image. However, since each of *Color* and *DCT* tasks operate on different blocks, a separate task instance for each of the block is considered, and the task instances are treated separately in the simulator. Thus, the JPEG encoding application is decomposed in 21 BLS tasks. The tasks operate in the same address space. The virtual memory for storing functional data is allocated in *Init* task and shared among the tasks as described in Section 5.2.3.2.

6.3.1.1 *Inter-task synchronization*

Decomposition of the application into multiple dependent tasks requires additional inter-task synchronization mechanisms. For this purpose, I employ a set of mutexes which are implemented as semaphores initialized to value 1. The mutexes are required in order to guarantee the correctness of the producer-consumer relations between the tasks. In the initial state, the mutexes are locked. A consumer has to successfully lock the mutex before reading and processing the input data. In turn, the mutex is unlocked by the producer right after it writes the output data to the shared memory. At this moment of time, the consumer can successfully lock the semaphore and start processing the input data. Thus, it can be assured that the consumer does not read wrong data in the shared memory.

The synchronization operations are inserted in the BLS tasks by means of empty synchronization functions (see Section 5.2.2.3). First, the empty functions were manually inserted into the target source code using the naming conventions described in Section 5.2.2.3. These functions do not change the functionality of the application in the measurement phase. In fact, in the measurement phase the tasks are executed sequentially without the need for actual synchronization. However, the inserted functions allow the translation unit to precise determine the synchronization points in the target binary code. During the generation of the translated code, the tool automatically inserts calls to the scheduler model with the parameters specified in the synchronization functions' names.

6.3.1.2 *Hardware platform*

As a target hardware platform, a generic symmetric multiprocessing (SMP) architecture is considered, consisting of multiple homogeneous processing cores with local instruction/data caches, arbitrated shared bus and main memory. In the exploration phase, 7 architecture types are evaluated. They differentiate in the number of processing cores from 2 to 8. The cores has identical configuration as specified in Section 6.2. Furthermore, for these experiments a fixed memory access latency of 16 cycles is assumed (however, not including dynamic arbitration delays). The local caches of the processing cores are coherent. During the simulation, the BLS tasks of the application are scheduled on the cores using the high-level OS scheduler introduced in Section 5.2.2.

6.3.2 *Results*

The purpose of these experiments is to estimate performance of different task mappings of the JPEG application on each of the 7 multicore architectures. For each architecture, 1000 randomly generated task mappings are evaluated. Thus, during the exploration 7000 task mappings were evaluated. The experimental results for each architecture are shown in Fig. 6.40. As can be seen in the figure, mapping of tasks has a large impact on the overall execution time of the application. Notably, a larger amount of processing core does not always improve the application performance. In fact, an unfavorable mapping can even increase the total execution time.

The figure shows a Pareto front for task mappings with the lowest execution time. Notably, the largest steps in the front can be observed at a smaller amount of cores (from 1 to 5). However, the execution time does not further decrease with a larger amount of cores and saturates at the value of 20 Mcycles. Thus, the experimental results resemble the effect of Amdahl's law [4], which postulates that the largest speedup achievable on a multiprocessor system is generally limited by the sequential part of the application.

The evaluation of 7000 mappings took 9 hours and 8 minutes on the host computer. Some of the key data from the experiments are presented in Table 5. As can be seen in the table, the simulation speed was approximately constant for all architectures with a value of 16.5–17.0 MIPS[6]. The simulation time did not significantly change with the number of cores because efforts required for simulating the processing cores dominated the simulation efforts for the communication infrastructure and cache coherence. As the total amount of simulated

6 The simulation speed is expressed in millions of target simulated instructions per second of real time.

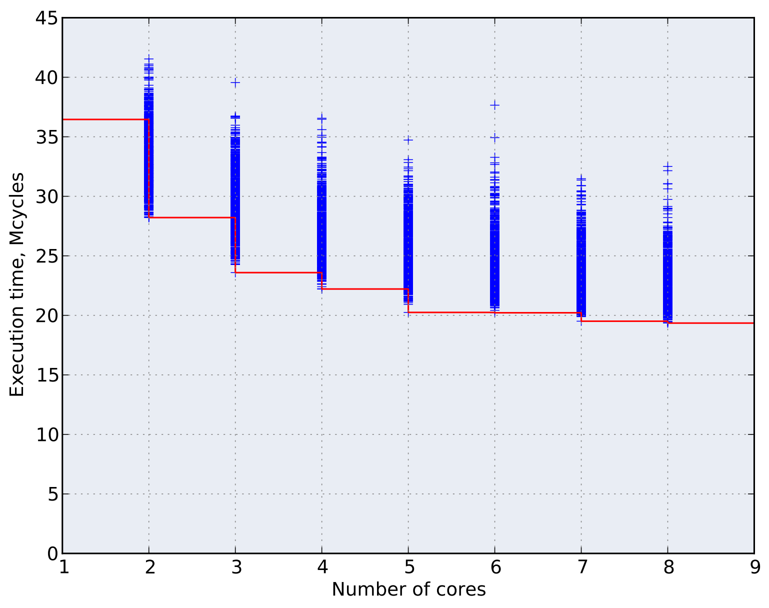


Figure 6.40: Estimated time of different task mappings on a SMP platform with a variable number of processing cores.

Table 5: Evaluation results of the JPEG application

No. of cores	Min. execution time, Mcycles	Max. execution time, Mcycles	Average exec. time, Mcycles	Average sim. speed, MIPS
2	28.2	41.5	33.4	16.9
3	23.6	39.6	29.6	16.7
4	22.2	36.6	27.1	16.5
5	20.2	34.7	25.5	16.5
6	20.2	37.7	24.4	16.5
7	19.5	31.5	23.5	16.6
8	19.3	32.5	22.8	16.7

target instructions did not change among the architectures, the overall wall-clock time required for simulation did not change as well.

The distribution of execution time for all evaluated task mappings in each simulated architecture is shown Fig. 6.41. The figures show the frequency of execution time values observed during the exploration. Notably, in architectures with a smaller amount of cores, the values of execution time are spread over a larger interval of execution times compared to architectures with a larger amount of cores. The average execution time over all task mappings decreased from 33.4 Mcycles in the 2-core architecture to 22.8 Mcycles in the 8-core architecture.

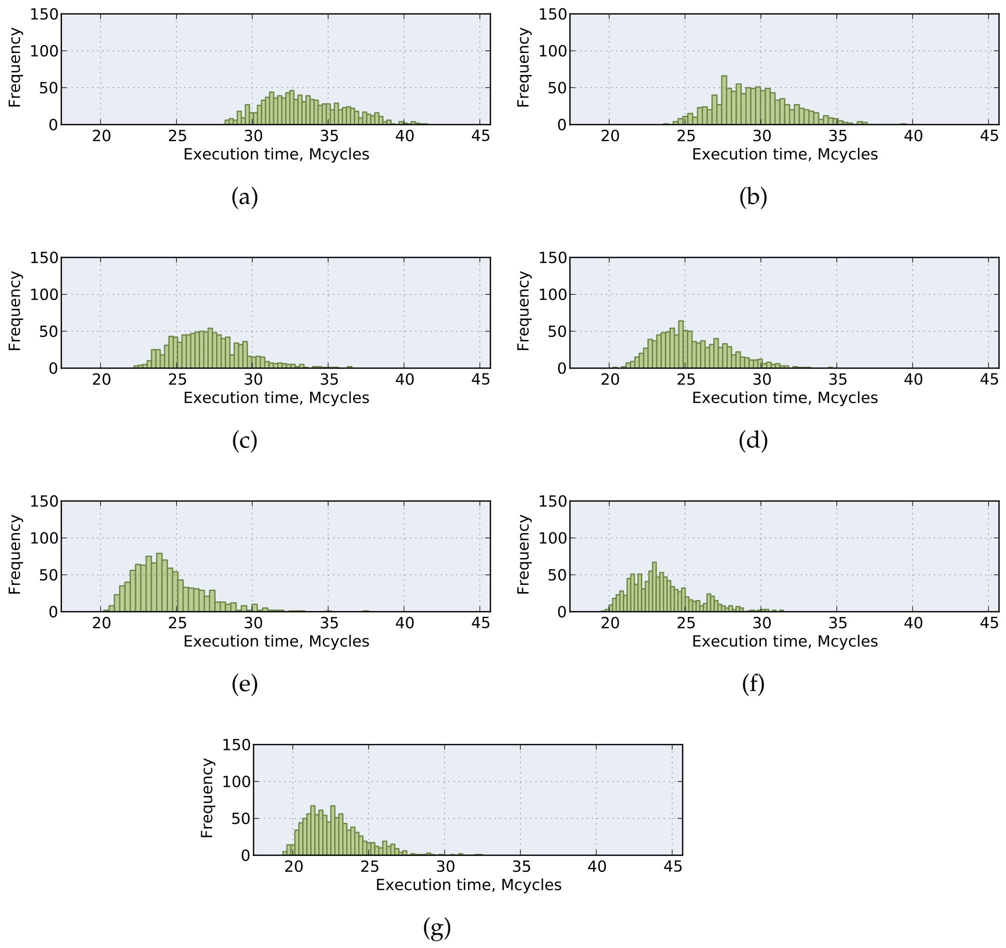


Figure 6.41: Frequency of the execution time values of the JPEG application on the HW platform with: (a) 2 cores, (b) 3 cores, (c) 4 cores, (d) 5 cores, (e) 6 cores, (f) 7 cores, (e) 8 cores.

The simulation tool implemented in the scope of this dissertation additionally has a graphical user interface (GUI). The interface allows for more detailed evaluation of simulation results. The tool generates a large number of graphical plots providing information on the utilization of cores, miss rates of instruction/data caches, execution time of target SW functions (profiling information), utilization of the data/instruction buses and main memory, amount of memory access to particular addresses, amount of invalidations in data caches as well as the scheduling-related information.

An example of the scheduling diagram is shown in Fig. 6.42. The diagram shows the execution flow of BLS tasks of the JPEG application on a 4-core architecture. By means of this diagram, the user can visually assess the distribution of the application's workload on the cores. Moreover, the use can prove the correctness of the inter-task synchronization mechanisms. To support this, the diagram can be additionally extended with labels showing the time points at which

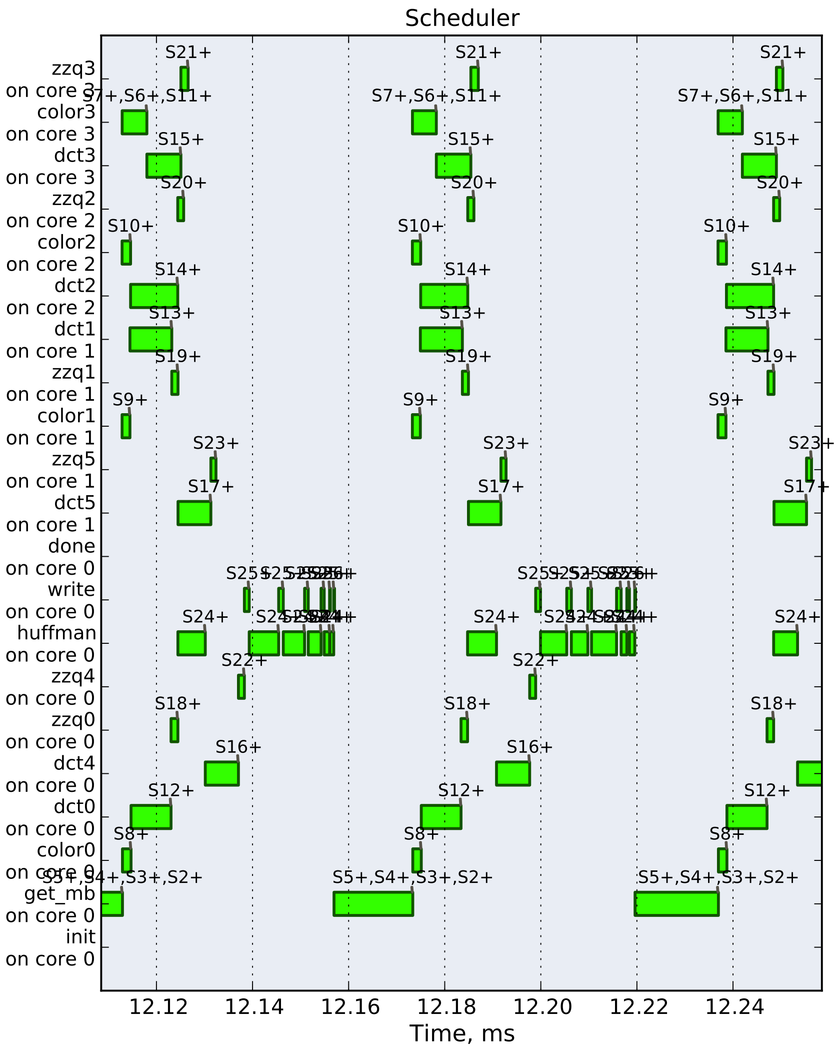


Figure 6.42: Scheduling diagram of the tasks running on a 4-core platform. The diagram is produced automatically by the GUI of the simulation tool.

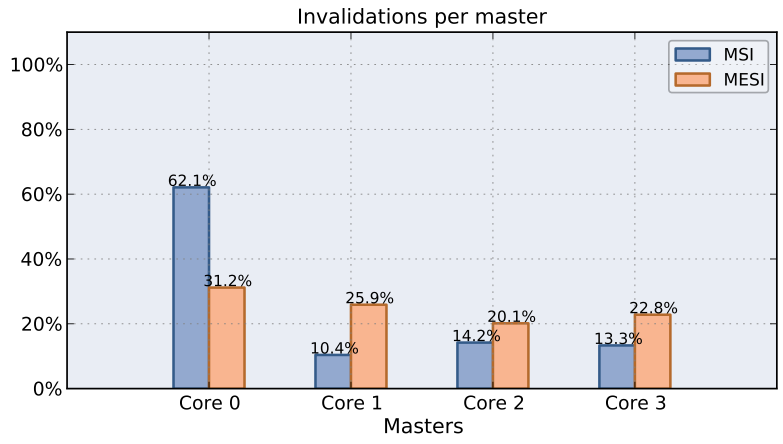


Figure 6.43: Percentage of data cache invalidations per processing core on a 4-core platform. The diagram is produced automatically by the GUI of the simulation tool.

the semaphores have been incremented. If necessary, the plot may include the states of the tasks as well as their deadlines.

Fig. 6.43 shows an example of a plot with coherency-related information generated by the tool. The figure shows the percentage of data cache invalidations per processing core. Note that two coherency protocols can be evaluated at a time by the tool: MSI and MESI[7]. Thus, by means of the generated plots, the designer can not only evaluate the execution time of the target software but also identify possible bottlenecks in the hardware architecture. Due to space limitation, this section presents only the two examples of the plots generated by the tool.

7 The MESI protocol reduces the amount of invalidation signals because of the additional *exclusive* state.

7 CONCLUSIONS AND FUTURE WORK

7.1 CONCLUSIONS

In this thesis, a novel method of compiled software simulation at the binary-level was introduced. The method is intended to support hardware and software developers during performance evaluation of MP-SoC architectures incorporating out-of-order processing cores. The proposed simulation technique considers the effects of out-of-order instruction execution in a target processor, such as context-dependent timing of instructions and reordering of memory accesses. Furthermore, the thesis presented a SystemC-based simulation framework for performance design space exploration of multiprocessor system-on-chip. In order to offer a better trade-off between accuracy and speed of performance estimations, the framework supports two simulation methods: compiled binary-level simulation (BLS) mentioned above and trace-driven simulation (TDS) based on abstract application traces. The framework allows the developers to evaluate diverse SW partitioning options and scheduling strategies of the target software on multicore hardware architectures. This chapter concludes the thesis with key findings and presents advantages and limitations of the presented approaches.

COMPILED BINARY-LEVEL SIMULATION The proposed method of *context-aware* compiled simulation can accurately reproduce the out-of-order processor behavior at the system-level at a speed much higher than cycle-accurate ISS. As a reference cycle-accurate simulator, the SimpleScalar tool was used, which is a highly configurable and widely adopted processor simulator[1]. The method was evaluated using 32 different embedded benchmarks. The experimental results showed that in SystemC environment the proposed method achieves an average speedup of 12.7 with an average timing error of 2.6% compared to the reference cycle-accurate ISS[2]. Thus, the suggested method improves the simulation efficiency and, thus, accelerates system-level design space exploration. The acceleration is achieved by creating application-specific cycle-approximate models of the target processing core, which are then employed during performance simulation of multicore architectures. The accuracy of the approach

1 According to the developer's website [96], "in 2000 more than one third of all papers published in top computer architecture conferences used the SimpleScalar tools to evaluate their designs".

2 This speedup is obtained assuming a large number of DSE iterations and already considers the overhead of SystemC simulation kernel.

in a multicore environment was validated against the cycle-accurate simulation by means of varying communication latencies.

The proposed simulation method is not intended to completely substitute cycle-accurate ISS in the design flow. Its purpose is to avoid *repetitive* cycle-accurate ISS during multiple iterations of system-level design space exploration. Thus, having explored a large set of possible design solutions, the designer can still validate the performance of a smaller set of the most promising candidates using cycle-accurate ISS.

Simulation of the target code at the binary level has many advantages compared to higher abstraction levels, e.g. source-level simulation. Particularly, binary-level simulation is always instruction-accurate and considers all optimizations of the target compiler by construction of the translated code. Moreover, it allows simulation of target software that makes use of the C standard library or third-party libraries, for which the source code has not been provided. However, the proposed method of binary-level simulation has a number of limitations as well.

Firstly, translated binary-level code contains a large amount of C-operations. Consequently, compilation of the code in the host computer—even after optimizing the structure of the code—takes considerably more time than compilation of the target code at the source level. If a target application has a large size of the code, compilation can become a bottleneck in this workflow. Secondly, during the derivation of execution time, the proposed method assumes typical input data to be applied to the target application. However, if compiled simulation reveals a previously undiscovered part of the code, the execution time of the undiscovered part must be derived once again, followed by the recompilation of the entire translated code. Finally, the method is currently not suited for rapid development of target software code. When even only a small part of the code has been changed, the complete translated code must be recompiled. These issues are subjects of the future work, which will be discussed later in this chapter.

TRACE-DRIVEN SIMULATION Compared to compiled BLS, trace-driven simulation allows for significantly higher average speedup of 68 times. However, the faster simulation speed is achieved at the expense of lower average accuracy. The average timing error of TDS in all tested benchmarks was 6.7% versus 2.6% in BLS. Nevertheless, for a certain class of applications, TDS is still preferable solution in terms of the speed/accuracy ratio. If a target application imposes low traffic on the communication interconnect, i.e. the share of communication requests for all executed instructions is low, TDS achieves faster simulation speed while providing the same level of accuracy as the BLS method. It can be explained as follows. If the target application generates a large amount of cache hits (and hence lower traffic on the interconnect), processing latencies captured by traces cover a larger portion of target instructions. Consequently, the share of timing error produced during data cache misses[3] decreases relative to the overall execution time of the application. As a result, the timing error of TDS tends to be low for applications generating small miss rates in instruction and data caches.

Traces completely abstract the functionality of applications and internal details of the core's microarchitecture and, hence, they efficiently reconstruct the traffic produced by the core at the output interfaces. Thus, if the designer is interested in the exploration of the on-chip interconnect, traces can be employed to exercise the application workload in the interconnect model by reproducing the same traffic pattern over multiple iterations. The advantage of traces is a possibility to reconstruct the workload in situations when the target code is not available or cannot be exposed to the designer due to protection of intellectual property (IP). Traces can be co-simulated with functional processor models, thereby recreating a typical background traffic in the MPSoC. Another advantage of traces is deterministic simulation. TDS can be easily reproduced from any point without the need of reconstructing the microarchitectural state of the processing core. Finally, traces can be easily modified to adjust the traffic according to the designer's needs, e.g. for studying corner cases which cannot be easily revealed using execution-driven simulation.

However, due to the high level of trace abstraction, the use of TDS can be restricted in certain scenarios. For example, TDS cannot be employed if the functionality of the target software has to be reproduced. Furthermore, traces completely abstract the behavior of local instruction and data caches. Therefore, cache coherency cannot be evaluated by TDS, as the cache behavior captured in traces at this abstraction level is fixed and invalidation of cache lines with shared data at simulation time is not possible. Another principal limitation of TDS is a fixed execution flow of the application captured in a trace. Abstract traces are always generated at a certain input data applied to the application. If the input data has been changed, a trace must be generated once again introducing additional overhead into the simulation. Thus, trace-driven simulation is restricted to repetitive design space exploration of target applications, for which the behavior does not change among iterations.

HIGH-LEVEL SCHEDULING The purpose of the presented high-level scheduler model is to enable code partitioning on a multicore architecture at early development stages. The scheduler model leverages BLS and TDS techniques for fast and yet accurate simulation of application tasks and thereby considers the timing effects in hard-

3 Trace-driven simulation assumes data accesses to be blocking and, hence, it overestimates the execution time for out-of-order processors.

ware. The model can be employed in a situation when the designer needs to investigate various task mapping and scheduling strategies on a target multicore architecture, but the target application has not yet been ported to any existing embedded OS implementations. Particularly, the designer can investigate the implications of task mapping such as performance gain due to parallelization of the application execution or additional execution delays due to the usage of shared resources. At the end, the designer can identify task mappings which are most beneficial in terms of the application performance.

The proposed scheduler model is generic and not tailored to any specific RTOS implementation. It provides a minimum set of basic services to support a synchronized execution of concurrent tasks and, thus, to enable simulation of distributed application execution on multiple cores. At this level of abstraction, the focus lies on the dependency relations between tasks. Thus, the scheduler allows the designer to validate the correctness of task synchronization and identify possible deadlocks in synchronization. Meanwhile, the OS-specific details of the synchronization mechanisms are abstracted.

Furthermore, the scheduler provides means for considering the coarse-grained execution time of RTOS services by employing abstract traces of OS-related workload. The traces are intended for a what-if analysis of a possible RTOS impact on the timing of the target application. By modifying the traces, the designer can investigate multiple scenarios of the RTOS workload timing behavior and determine a maximum time budget available for an RTOS. In this way, the scheduler can be employed to obtain timing requirements for future RTOS candidates. This approach follows the refinement process in platform-based design, in which the parameters of virtual components, as formulated in [67], "are set by the requirements rather than by the implementation" and "have to be considered as constraints for the next level of refinement".

In the next step of the design flow, which is not covered in this thesis, the scheduler model can be substituted with a configurable RTOS model which can accurately reproduce the behavior of real RTOS implementations. Creation of configurable generic RTOS models is an on-going research topic [89, 25, 52]. A generic RTOS model needs to have a common set of properties from multiple RTOS implementations. The model has to accurately reflect the behavior of many services, e.g. rescheduling policies, inter-process communication methods (message passing or shared memory), IO services, methods for memory protection. Therefore, validation of generic RTOS models against the real counterparts is of a great importance. Refinement of the proposed high-level scheduler to one or several specific RTOS implementations is an open issue left for the future work.

7.2 OUTLOOK FOR FUTURE WORK

There are several open issues which identify directions for possible future research:

- The employment of the BLS method during testing and debugging of the target SW is currently hindered because the complete translated code has to be recompiled even if only a small part of it has been changed. New methodology is required for enabling partial recompilation of the code that would reduce the compilation overhead during the debugging process. Moreover, the timing of the modified part of the code has to be obtained using cycle-accurate simulation in an efficient way. New techniques are required that would allow the measurement of only selected parts of the target code. A particular challenge is to accurately reconstruct the microarchitectural state of the core at the beginning of basic blocks that have to be measured.

- An interesting approach for reducing the compilation overhead would be to adopt dynamic binary translation techniques, which rely on dynamic compilation of the code at simulation run-time. This approach will introduce many challenges. For example, current techniques do not differentiate instances of an instruction in the target code, as they primarily focus on functional simulation of the code. However, the same instruction executed in different parts of the code has different timing, as it is executed at different contexts of the core's microarchitecture.

- Another possible research topic would be to apply the proposed compiled simulation method at higher abstraction levels of the target code, e.g. during source-level simulation. Direct employment of the source code instead of translated binary code can significantly improve the simulation speed. However, it is also a challenging task as the instruction details are not available at the source level.

- Refinement of the current high-level model of the scheduler to a configurable generic RTOS model is also a subject of future work. Possibility for reconstructing the behavior of existing RTOS implementations would improve the accuracy of performance evaluation and allow for more comprehensive design space exploration.

- Finally, the complexity of embedded processing cores will likely increase in the future. They may implement further techniques for optimizing the application performance, e.g. hardware multithreading, or incorporate advanced caches that allow multiple outstanding misses. Consideration of these techniques may become necessary for performance evaluation of target software in the next generations of embedded processors.

BIBLIOGRAPHY

[1] H. AbdElSalam, S. Kobayashi, K. Sakanushi, Y. Takeuchi, and M. Imai. Towards a higher level of abstraction in Hardware/Software co-simulation. In *Proceedings of the 24th International Conference on Distributed Computing Systems Workshops*, pages 824–830, 2004.

[2] R. D. Acosta, J. Kjelstrup, and H. C. Torng. An instruction issuing approach to enhancing performance in multiple functional unit processors. In *IEEE Transactions on Computers*, C-35(9): 815–828, Sept. 1986.

[3] O. Almer, I. Böhm, T. von Koch, B. Franke, S. Kyle, V. Seeker, C. Thompson, and N. Topham. Scalable multi-core simulation using parallel dynamic binary translation. In *Proceedings of International Conference on Embedded Computer Systems (SAMOS)*, pages 190–199, 2011.

[4] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the AFIPS conference*, pages 483–485, ACM, 1967 (Spring).

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

[6] J. Bammi, E. Harcourt, W. Kruitzer, L. Lavagno, and M. Lazarescu. Software performance estimation strategies in a system-level design tool. In *Proceedings of the 8th International Workshop on Hardware/Software Codesign (CODES)*, pages 82–86, 2000.

[7] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference*, ATEC 2005, page 41, USENIX Association, 2005.

[8] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri. MPARM: Exploring the multi-processor SoC design space with SystemC. In *Journal of VLSI Signal Processing*, 41(2): 169–182, 2005.

[9] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. In *SIGARCH Computer Architecture News*, 39(2): 1–7, Aug. 2011.

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

[11] I. Böhm, B. Franke, and N. Topham. Cycle-accurate performance modelling in an ultra-fast just-in-time dynamic binary translation instruction set simulator. In *Proceedings of the International Conference on Embedded Computer Systems (SAMOS)*, pages 1–10, 2010.

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

[13] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. In *SIGARCH Computer Architecture News*, 25(3): 13–25, June 1997.

[14] D. Burger, A. Kägi, and M. S. Hrishikesh. Memory hierarchy extensions to the SimpleScalar tool set. Technical report, 2000.

[15] E. Cheung, H. Hsieh, and F. Balarin. Fast and accurate performance simulation of embedded software for MPSoC. In *Proceedings of Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 552–557, 2009.

[16] J. Chevalier, O. Benny, M. Rondonneau, G. Bois, E. M. Aboulhamid, and F.-R. Boyer. Space: A Hardware/Software SystemC modeling platform including an RTOS. In *Languages for system specification: Selected contributions on UML, SystemC, System Verilog, mixed-signal systems, and property specification from FDL'03*, pages 91–104. Kluwer Academic Publishers, 2004.

[17] L. Eeckhout, R. H. Bell Jr., B. Stougie, K. De Bosschere, and L. K. John. Control flow modeling in statistical simulation for accurate and efficient processor design studies. In *SIGARCH Computer Architecture News*, 32(2): 350, Mar. 2004.

[18] S. Fytraki and D. Pnevmatikatos. ReSim, a trace-driven, reconfigurable ILP processor simulator. In *Proceedings of Design, Automation and Test in Europe Conference (DATE)*, pages 536–541, 2009.

[19] L. Gao, K. Karuri, S. Kraemer, R. Leupers, G. Ascheid, and H. Meyr. Multiprocessor performance estimation using hybrid simulation. In *Proceedings of 45th ACM/IEEE Design Automation Conference (DAC)*, pages 325–330, 2008.

[20] A. Gerstlauer. Host-compiled simulation of multi-core platforms. In *Proceedings of the 21st IEEE International Symposium on Rapid System Prototyping (RSP)*, pages 1–6, 2010.

[21] A. Gerstlauer, H. Yu, and D. Gajski. RTOS modeling for system level design. In *Proceedings of Design, Automation and Test in Europe (DATE)*, pages 130–135, 2003.

[22] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[23] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of IEEE International Workshop on Workload Characterization*, pages 3–14, 2001.

[24] G. Hamerly, E. Perelman, J. Lau, and B. Calder. Simpoint 3.0: Faster and more flexible program analysis. In *Journal of Instruction Level Parallelism*, 2005.

[25] Z. He, A. Mok, and C. Peng. Timed RTOS modeling for embedded system design. In *Proceedings of the 11th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS)*, pages 448–457, 2005.

[26] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[27] M. A. Holliday and C. S. Ellis. Accuracy of memory reference traces of parallel computations in trace-drive simulation. In *IEEE Transactions on Parallel and Distributed Systems*, 3(1): 97–109, 1992.

[28] Y. Hwang, S. Abdi, and D. Gajski. Cycle-approximate retargetable performance estimation at the transaction level. In Proceedings of *Design, Automation and Test in Europe Conference (DATE)*, pages 3–8, 2008.

[29] T. Isshiki, D. Li, H. Kunieda, T. Isomura, and K. Satou. Trace-driven workload simulation method for multiprocessor system-on-chips. In *Proceedings of the 46th Annual Design Automation Conference (DAC)*, pages 232–237, ACM, 2009.

[30] M. Johnson. *Superscalar microprocessor design.* Prentice Hall series in innovative technology. Prentice Hall, 1991.

[31] D. Jones and N. Topham. High speed CPU simulation using LTU dynamic binary translation. In *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers* (HiPEAC), pages 50–64, Springer-Verlag, 2009.

[32] T. Kempf, M. Doerper, R. Leupers, G. Ascheid, H. Meyr, T. Kogel, and B. Vanthournout. A modular simulation framework for spatial and temporal task mapping onto multi-processor SoC platforms. In *Proceedings of Design, Automation and Test in Europe*

Conference (DATE), Vol. 2, pages 876–881, IEEE Computer Society, 2005.

[33] T. Kempf, K. Karuri, S. Wallentowitz, G. Ascheid, R. Leupers, and H. Meyr. A SW performance estimation framework for early system-level-design using fine-grained instrumentation. In *Proceedings of Design, Automation and Test in Europe Conference (DATE)*, pages 468–473, European Design and Automation Association, 2006.

[34] S. Kraemer, L. Gao, J. Weinstock, R. Leupers, G. Ascheid, and H. Meyr. HySim: A fast simulation framework for embedded software development. In *Proceedings of the 5th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 75–80, 2007.

[35] M. Krause, D. Englert, O. Bringmann, and W. Rosenstiel. Combination of instruction set simulation and abstract RTOS model execution for fast and accurate target software evaluation. In *Proceedings of the 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 143–148, ACM, 2008.

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

[37] C. Lee, M. Potkonjak, and W. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of 30th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 330–335, 1997.

[38] H. Lee, L. Jin, K. Lee, S. Demetriades, M. Moeng, and S. Cho. Two-phase trace-driven simulation (TPTS): A fast multicore processor architecture simulation approach. In *Software: Practice and Experience*, 40(3): 239–258, 2010.

[39] J.-Y. Lee and I.-C. Park. Timed compiled-code simulation of embedded software for performance analysis of SOC design. In *Proceedings of the 39th Design Automation Conference (DAC)*, pages 293–298, 2002.

[40] K. Lee, S. Evans, and S. Cho. Accurately approximating superscalar processor performance from traces. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 238–248, 2009.

[41] R. Leupers, J. Elste, and B. Landwehr. Generation of interpretive and compiled instruction set simulators. In *Proceedings of the Asia*

and South Pacific Design Automation Conference (ASP-DAC), Vol.1, pages 339–342, 1999.

[42] R. Leupers, F. Schirrmeister, G. Martin, T. Kogel, R. Plyaskin, A. Herkersdorf, and M. Vaupel. Virtual platforms: Breaking new grounds. In *Proceedings of Design, Automation Test in Europe Conference (DATE)*, pages 685–690, 2012.

[43] K.-L. Lin, C.-K. Lo, and R.-S. Tsay. Source-level timing annotation for fast and accurate TLM computation model generation. In *Proceedings of the 15th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 235–240, 2010.

[44] K. Lu, D. Müller-Gritschneder, and U. Schlichtmann. Hierarchical control flow matching for source-level simulation of embedded software. In *Proceedings of International Symposium on System on Chip (SoC)*, pages 1–5, 2012.

[45] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. In *Computer*, 35(2): 50–58, 2002.

[46] P. S. Magnusson, F. Dahlgren, H. Grahn, M. Karlsson, F. Larsson, F. Lundholm, A. Moestedt, J. Nilsson, P. Stenström, and B. Werner. SimICS/sun4m: A virtual workstation. In *Proceedings of the annual conference on USENIX Annual Technical Conference ATEC*, USENIX Association, 1998.

[47] S. Mahadevan, F. Angiolini, J. Sparsø, L. Benini, and J. Madsen. A reactive and cycle-true IP emulator for MPSoC exploration. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(1): 109–122, 2008.

[48] S. Mahadevan, K. Virk, and J. Madsen. ARTS: a SystemC-Based framework for multiprocessor systems-on-chip modelling. In *Design Automation for Embedded Systems*, 11(4): 285–311, Dec. 2007.

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

[50] C. Mills, S. C. Ahalt, and J. Fowler. Compiled instruction set simulation. 1991.

[51] R. L. Moigne, O. Pasquier, and J.-P. Calvez. A generic RTOS model for real-time systems simulation with SystemC. In *Proceedings of Design, Automation and Test in Europe Conference (DATE)*, Vol. 3, pages 82–87, 2004.

[52] M. Müller, J. Gerlach, and W. Rosenstiel. RTOS-aware modeling of embedded hardware/software systems. In *Proceedings of IEEE International Conference on Computer Design (ICCD)*, pages 179–186, 2010.

[53] T. Nakada, T. Tsumura, and H. Nakashima. Design and implementation of a workload specific simulator. In *Proceedings of the 39th Annual Symposium on Simulation*, pages 230–243. IEEE Computer Society, 2006.

[54] A. Nohl, G. Braun, O. Schliebusch, R. Leupers, H. Meyr, and A. Hoffmann. A universal technique for fast and flexible instruction-set architecture simulation. In *Proceedings of the 39th Annual Design Automation Conference (DAC)*, pages 22–27, 2002.

[55] M. S. Papamarcos and J. H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *SIGARCH Computure Architecture News*, 12(3): 348–354, Jan. 1984.

[56] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface (Revised Fourth Edition)*. Elsevier, Nov. 2011.

[57] A. Pimentel, C. Erbas, and S. Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. In *IEEE Transactions on Computers*, 55(2): 99–112, 2006.

[58] A. D. Pimentel, M. Thompson, S. Polstra, and C. Erbas. Calibration of abstract performance models for system-level design space exploration. In *Journal of Signal Processing Systems*, 50(2): 99–114, 2008.

[59] R. Plyaskin and A. Herkersdorf. A method for accurate high-level performance evaluation of MPSoC architectures using fine-grained generated traces. In *Proceedings of Architecture of Computing Systems Conference (ARCS)*, pages 199–210, 2010.

[60] R. Plyaskin and A. Herkersdorf. Context-aware compiled simulation of out-of-order processor behavior based on atomic traces. In *Proceedings of IEEE/IFIP International Conference on VLSI and System-on-Chip (VLSI-SoC)*, pages 386–391, 2011.

[61] R. Plyaskin, A. Masrur, M. Geier, S. Chakraborty, and A. Herkersdorf. High-level timing analysis of concurrent applications on MPSoC platforms using memory-aware trace-driven simulations. In *Proceedings of IEEE/IFIP International Conference on VLSI and System-on-Chip (VLSI-SoC)*, pages 229–234, 2010.

[62] R. Plyaskin, T. Wild, and A. Herkersdorf. System-level software performance simulation considering out-of-order processor ex-

ecution. In *Proceedings of International Symposium on System on Chip (SoC)*, pages 1–7, 2012.

[63] D. C. Powell and B. Franke. Using continuous statistical machine learning to enable high-speed performance prediction in hybrid instruction-/cycle-accurate instruction set simulators. In *Proceedings of the 7th IEEE/ACM International Conference on Hardware/-Software Codesign and System Synthesis (CODES+ISSS)*, pages 315–324, 2009.

[64] C. A. Prete, G. Prina, and L. Ricciardi. A trace-driven simulator for performance evaluation of cache-based multiprocessor systems. In *IEEE Transactions on Parallel and Distributed Systems*, 6(9): 915–929, 1995.

[65] M. Reshadi, P. Mishra, and N. Dutt. Hybrid-compiled simulation: An efficient technique for instruction-set architecture simulation. In *ACM Transactions in Embedded Computing Systems*, 8(3): 20:1–20:27, Apr. 2009.

[66] B. Sander, J. Schnerr, and O. Bringmann. ESL power analysis of embedded processors for temperature and reliability estimations. In *Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 239–248, ACM, 2009.

[67] A. Sangiovanni-Vincentelli. Quo vadis, SLD? Reasoning about the trends and challenges of system level design. In *Proceedings of the IEEE*, 95(3): 467–506, Mar. 2007.

[68] S. Schliecker, J. Rox, M. Negrean, K. Richter, M. Jersak, and R. Ernst. System level performance analysis for real-time automotive multicore and network architectures. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7): 979–992, 2009.

[69] J. Schnerr, O. Bringmann, and W. Rosenstiel. Cycle accurate binary translation for simulation acceleration in rapid prototyping of SoCs. In Proceedings of *Design, Automation and Test in Europe Conference (DATE)*, Vol. 2, pages 792–797, 2005.

[70] J. Schnerr, O. Bringmann, A. Viehl, and W. Rosenstiel. High-performance timing simulation of embedded software. In *Proceedings of the 45th Annual Design Automation Conference (DAC)*, pages 290–295, ACM, 2008.

[71] S. W. Sherman and J. C. Browne. Trace driven modeling: Review and overview. In *Proceedings of the 1st Symposium on Simulation of Computer Systems*, pages 200–207, IEEE Press, 1973.

[72] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *SIGARCH Computer Architecture News*, 30(5): 45–57, Oct. 2002.

[73] G. S. Sohi. Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers. In *IEEE Transactions on Computers*, 39(3): 349–359, Mar. 1990.

[74] S. Stattelmann, O. Bringmann, and W. Rosenstiel. Dominator homomorphism based code matching for source-level simulation of embedded software. In *Proceedings of the 7th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 305–314, ACM, 2011.

[75] S. Stattelmann, G. Gebhard, C. Cullmann, O. Bringmann, and W. Rosenstiel. Hybrid source-level simulation of data caches using abstract cache models. In Proceedings of *Design, Automation Test in Europe Conference (DATE)*, pages 376–381, 2012.

[76] S. Stattelmann, S. Ottlik, A. Viehl, O. Bringmann, and W. Rosenstiel. Combining instruction set simulation and WCET analysis for embedded software performance estimation. In *Proceedings of the 7th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 295–298, 2012.

[77] D. Thach, Y. Tamiya, S. Kuwamura, and A. Ike. Fast cycle estimation methodology for instruction-level emulator. In *Proceedings of Design, Automation Test in Europe Conference (DATE)*, pages 248–251, 2012.

[78] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. In *IBM Journal of Research and Development*, 11(1): 25–33, Jan. 1967.

[79] R. A. Uhlig and T. N. Mudge. Trace-driven memory simulation: A survey. In *ACM Computing Surveys*, 29(2): 128–170, June 1997.

[80] A. Viehl, M. Pressler, and O. Bringmann. Bottom-up performance analysis considering time slice based software scheduling at system level. In *Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 423–432, ACM, 2009.

[81] Z. Wang and J. Henkel. HyCoS: Hybrid compiled simulation of embedded software with target dependent code. In *Proceedings of the 8th IEEE/ACM/IFIP International Conference on Hardware/-Software Codesign and System Synthesis (CODES+ISSS)*, pages 133–142, ACM, 2012.

[82] Z. Wang and A. Herkersdorf. An efficient approach for system-level timing simulation of compiler-optimized embedded software. In *Proceedings of the 46th Annual Design Automation Conference (DAC)*, pages 220–225, ACM, 2009.

[83] Z. Wang, A. Sanchez, and A. Herkersdorf. SciSim: A software performance estimation framework using source code instrumentation. In *Proceedings of the 7th international workshop on software and performance*, pages 33–42, ACM, 2008.

[84] T. Wild, A. Herkersdorf, and G.-Y. Lee. TAPES—Trace-based architecture performance evaluation with SystemC. In *Design Automation for Embedded Systems*, 10(2):1 57–179, 2006.

[85] M.-H. Wu, P.-C. Wang, C.-Y. Fu, and R.-S. Tsay. An extended SystemC framework for efficient HW/SW co-simulation. *ACM Transactions on Design Automation of Electronic Systems*, 17(2): 11:1–11:16, Apr. 2012.

[86] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling. In *SIGARCH Computer Architecture News*, 31(2): 84–97, May 2003.

[87] Y. Yi, D. Kim, and S. Ha. Fast and time-accurate cosimulation with OS scheduler modeling. In *Design Automation for Embedded Systems*, 8(2-3): 211–228, June 2003.

[88] Y. Yi, D. Kim, and S. Ha. Fast and accurate cosimulation of MP-SoC using trace-driven virtual synchronization. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(12): 2186–2200, 2007.

[89] S. Yoo, G. Nicolescu, L. Gauthier, and A. Jerraya. Automatic generation of fast timed simulation models for operating systems in SoC design. In *Proceedings of Design, Automation and Test in Europe Conference (DATE)*, page 620. IEEE Computer Society, 2002.

[90] J. Zhu and D. Gajski. A retargetable, ultra-fast instruction set simulator. In *Proceedings of Design, Automation and Test in Europe Conference (DATE)*, pages 298–302, 1999.

[91] ARM1136JF-S and ARM1136J-S, technical reference manual (revision r1p5), 2009.

[92] Embedded JPEG codec library, available on http://www.sourceforge.net, 2009.

[93] e5500 core reference manual (rev. 3), Nov. 2012.

[94] Intel CoFluent technology overview, available on http://www.intel.com, 2013.

[95] OVPsim, open virtual platforms, available on http://www.ovpworld.org, 2013.

[96] The website of SimpleScalar simulation tool, http://simplescalar.com, 2013.

[97] VaST systems, available on http://www.synopsys.com, 2013.

COLOPHON

This document is based on `classicthesis` template developed by André Miede. The template is available on http://code.google.com/p/classicthesis/.