TECHNISCHE UNIVERSITÄT MÜNCHEN
Lehrstuhl für Integrierte Systeme

Software Performance Estimation Methods for System-Level Design of Embedded Systems

Zhonglei Wang

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

Doktor-Ingenieurs

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr.-Ing. habil. Gerhard Rigoll

Prüfer der Dissertation:

1. Univ.-Prof. Dr. sc. techn. Andreas Herkersdorf
2. Univ.-Prof. Dr. sc. Samarjit Chakraborty

Die Dissertation wurde am 15.04.2010 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 18.10.2010 angenommen.

Abstract

Software Performance Estimation Methods for System-Level Design of Embedded Systems

by

Zhonglei Wang

Doctor of Engineering in Electrical Engineering

Technical University of Munich

Driven by market needs, the complexity of modern embedded systems is ever increasing, which poses great challenges to the design process. The design productivity based on the traditional design methods cannot keep pace with the technological advance, resulting in an ever-widening design gap. To close this gap, System-Level Design (SLD) is seen as a promising solution. The main concept of SLD is to reduce design complexity by modeling systems at a high abstraction level, i.e., at system level. A systematic Design Space Exploration (DSE) method is one of the most critical parts of a system-level design methodology. DSE at system level is aimed at making important design decisions in early design phases under several design constraints. Performance is one of the most important design constraints.

Another obvious trend in embedded systems is the increasing importance of software. It is estimated that more than 98% microprocessor chips manufactured every year are used in embedded systems. Most parallel computing platforms use software processors as the main processing elements and are essentially software-centric. Several sources confirm that software is dominating overall design effort of embedded systems. This makes software performance estimation a critical issue in DSE of embedded systems. Hence, this work is focused on software performance estimation methods for system-level design of embedded systems.

Much effort has been made in both academia and design automation industry to increase the performance of cycle-accurate microprocessor simulators, but they are still too slow for efficient design space exploration of large multiprocessor systems. Moreover, a cycle-accurate simulator takes too much effort to build, and therefore, it is impossible to create one for each candidate architecture. Motivated by this fact, we focus on modeling processors at a higher level to achieve a much higher simulation speed but without compromising accuracy. A popular technique for fast software simulation is Source Level Simulation (SLS). SLS models are obtained by annotating application source code with timing information. We developed a SLS approach called *SciSim (Source code instrumentation based Simulation)*. Compared to other existing SLS approaches, SciSim allows for more accurate performance simulation by taking important timing effects such as the pipeline effect, branch prediction effect and cache effect into account. The back-annotation of timing information into source code is based on the mapping between source code and binary code, described by debugging information.

However, SLS has a major limitation that it cannot simulate some compiler-optimized programs with complex control flows accurately, because after optimizing compilation it is hard to find an accurate mapping between source code and binary code, and even when the mapping is found, due to the difference between source level control flows and binary level control flows the back-annotated timing information cannot be aggregated correctly during the simulation. Since, in reality, software programs are usually compiled with optimizations, this drawback strongly limits the usability of SLS. To solve this problem, we developed another approach based on SciSim, called *iSciSim (intermediate Source code instrumentation based Simulation)*. The idea behind iSciSim is to get an intermediate representation of source code, called *intermediate source code (ISC)*, which has a structure close to the structure of its binary code and thus allows for accurate back-annotation of the timing information obtained from the binary level. The back-annotated timing information can also be aggregated correctly along the ISC-level control flows.

For multiprocessor systems design, iSciSim can be used to generate Transaction Level Models (TLMs) in SystemC. In many embedded systems, especially in real time systems, multiple software tasks may run on a single processor, scheduled by a Real-Time Operating System (RTOS). To take this dynamic scheduling behavior into account in system-level simulation, we created an abstract RTOS model in SystemC to schedule the execution of software TLMs generated by iSciSim.

In addition to simulation methods, we also contributed to Worst-Case Execution Time (WCET) estimation. We apply the concept of intermediate source code to facilitate flow analysis for WCET estimation of compiler-optimized software. Finally, a system-level design framework for automotive control systems is presented. The introduced software performance estimation methods are employed in this design framework. In addition to fast software performance simulation and WCET estimation, we also propose model-level simulation for approximate performance estimation in an early design phase before application source code can be generated.

Acknowledgments

First and foremost, I would like to thank my advisor Prof. Andreas Herkersdorf. He gave me the chance to get this great research topic and provided me a lot of advices, inspiration, and encouragement throughout my Ph.D. study. I am truly grateful for his help, not only in my research, but also in my life. I also want to thank Prof. Walter Stechele for his guidance and support.

I would like to thank Prof. Samarjit Chakraborty for being the co-examiner of my thesis and for his valuable comments. I also want to thank Prof. Gerhard Rigoll for chairing the examination committee.

I am grateful to the BMW Forschung und Technik GmbH for their support of my work in the course of the Car@TUM cooperation program. In particular, I would like to thank Dr. Martin Wechs, BMW Forschung und Technik, for the constructive discussions and valuable inputs throughout our collaboration.

During my three-year work in the BASE.XT project, I was fortunate to work with a group of talented and creative colleagues from different institutes. They include: Andreas Bauer, Wolfgang Haberl, Markus Herrmannsdörfer, Stefan Kugele, Christian Kühnel, Stefano Merenda, Florian Müller, Sabine Rittmann, Christian Schallhart, Michael Tautschnig, and Doris Wild. They expertise in different aspects in embedded systems design. The discussions with them have always been a great learning experience for me. In addition, I would like to thank the professors and BMW colleagues involved in the project.

A large amount of thanks goes to my colleagues and friends at LIS. They include: Abdelmajid Bouajila, Christopher Claus, Michael Feilen, Robert Hartl, Mattias Ihmig, Kimon Karras, Andreas Laika, Andreas Lankes, Daniel Llorente, Michael Meitinger, Felix Miller, Rainer Ohlendorf, Johny Paul, Roman Plyaskin, Holm Rauchfuss, Gregor Walla, Stefan Wallentowitz, Thomas Wild, Johannes Zeppenfeld, and Paul Zuber. In particular, I want to thank Roman Plyaskin and Holm Rauchfuss for the constructive discussions in LIS-BMW meetings. Their comments from various angels helped in improving my work. I also want to thank the members of the institute administration for maintaining an excellent workplace. They are: Verena Draga, Wolfgang Kohtz, Gabi Spörle, and Doris Zeller.

I also want to take the chance to thank Bernhard Lippmann, Stefan Rüping, Andreas Wenzel and other colleagues at the department of Chipcard and Security ICs, Infineon AG, for their support during my student job and master's thesis work.

Last but certainly not least, I would like to thank my parents and my wife, to whom this thesis is dedicated. My parents raised me up and tried their best to provide me the best possible education and a healthy atmosphere at home. They are always supportive no matter what choice I make. Without them I would not have been able to finish this work. I want to thank my lovely wife Yanlan for her love, constant support, encouragement, and understanding during my work. It was the biggest surprise in my life to meet her in Germany. She made my life more colorful.

To my parents, wife and daughter

献给我的父母、妻子和女儿

Contents

Part I

Introduction and Background

Chapter 1

Introduction

Embedded systems are ubiquitous in our everyday lives, spanning all aspects of modern life. They appear in small portable devices, such as MP3 players and cell phones, and also in large machines, such as cars, aircrafts and medical equipments.

Driven by market needs, the demand for new features in embedded system products has been ever increasing. For example, twenty years ago, a mobile phone has usually only the functionality of making voice calls and sending text messages. Today, a modern mobile phone has hundreds of features, including camera, music and video playback, and GPS navigation etc. The rapid growth in application complexity places even greater demands on the performance of the underlying platform. A single high-performance microprocessor cannot fulfill the performance requirement any longer. This has leaded to the advent of parallel architectures. The typical parallel architectures used in the embedded systems domain include distributed embedded systems and Multiprocessor System-on-Chip (MPSoC). A distributed system consists of a number of processing nodes distributed over the system and connected by some interconnect network. An example of distributed systems is automotive control systems. In a car like BMW 7-series over 60 processors are networked to control a large variety of functions. An MPSoC is a processing platform that integrates the entire system into a single chip. Chips with hundreds of processor cores have been fabricated. With the technological advance, it is even possible to integrate thousands of cores in a single chip.

The complexity of today's embedded systems opens up a large design space. Designers have to select a suitable design from a vast variety of solution alternatives to satisfy constraints on characteristics such as performance, cost, and power consumption. This design complexity makes the design productivity cannot keep pace with the technological advance. This results in a gap between the advance rate of technology and the growth rate of design productivity, called *design gap*. New design methodologies are highly required to close the design gap and shorten the time-to-market.

Most agree that System-Level Design (SLD) methodologies are the most efficient solution to address the design complexity and thus to improve the design productivity. SLD is aimed to simplify the specification, verification and implementation of systems including hardware and software, and to enable more efficient design space exploration (DSE), by raising the abstraction level at which systems are modeled. System-level DSE is aimed at making design decisions as many as possible in early design phases. Good design decisions made in early phases make up a strong basis for the following steps. For example, to design an MPSoC, the first important decision to be made at system level is the choice of an appropriate system architecture that is suited for the target application domain. Then, we should decide the number of processors and hardware accelerators, add some application specific modules, make the decision of hardware/software partitioning and map the tasks to the available resources. These decisions are easier to make at a high abstraction level, where a system is modeled without too many implementation details.

During DSE, performance is one of the most important design constraints to be considered, especially for real-time systems. If performance requirements cannot be met, this could even lead to system failure. Hence, many design decisions are made in order to meet the performance requirements. If it is found in the implementation step that some performance requirements are not satisfied, this could lead to very costly redesign. Therefore, accurate performance estimation is very important in order to reduce the possibility of such design errors.

1.1 The Scope and Objects of This Work

Because software-based implementations are much cheaper and more flexible than hardware-based implementations, more and more functionality of embedded systems has been moving from hardware to software. This makes the importance of software and its impact on the overall system performance steadily increasing. Hence, this work is focused on software performance estimation methods for system-level design of embedded systems.

Software performance estimation methods fall mainly into two categories: static timing analysis and dynamic simulation. Static timing analysis is carried out in an analytical way based on mathematical models. It is often applied to worst-case execution time (WCET) estimation for hard real-time applications. In contrast, dynamic performance simulation really executes software code and is targeted at modeling run-time behaviors of software and estimating the timing features of target processors with respect to typical input data sets. Software performance simulation is our main focus, but we also researched on flow analysis for WCET estimation of compiler-optimized software.

In addition, we also worked on a system-level design framework SysCOLA for automotive systems [98]. This framework combines a modeling environment based on a formal modeling language COLA [65] and a SystemC-based virtual prototyping environment. Both COLA and the framework were developed in the scope of a cooperation project between Technical University of Munich and BMW Forschung und Technik GmbH, called BASE.XT, where 7 Ph.D. students were involved. In SysCOLA, my own contributions include (1) a VHDL code generator, generating VHDL code from COLA models for hardware implementation [99], (2) a WCET analysis tool, that estimates the WCET of each task to be used in an automatic mapping algorithm, (3) a SystemC code generator, that generates SystemC code from COLA models for performance simulation early at the model level before C code can be generated [94], and (4) the SystemC-based virtual prototyping environment for architectural design and performance validation [93]. In this thesis, I will present the SysCOLA framework and show the role software performance estimation plays in the framework. Here, software performance estimation includes software performance simulation, WCET estimation, and model-level simulation of COLA. Software performance simulation and WCET estimation are introduced separately, while, in the scope of SysCOLA, we introduce only model-level simulation.

To summarize, the scope and objects of our work include the following three parts, which are discussed in more detail in the following sub-sections.

- Software performance simulation methods for system-level design space exploration.
- Flow analysis for WCET estimation of compiler-optimized embedded software.
- Introduction of the SysCOLA framework and model-level simulation of COLA.

1.1.1 Software Performance Simulation

Today, simulative methods are still the dominant methods for DSE of embedded systems. They have the ability to get dynamic performance statistics of a system. Earlier, software is usually simulated using instruction set simulators (ISSs) to get the influence of software execution on system performance and study the run-time interactions between software and other system components. An ISS realizes the instruction-level operations of a target processor, including typically instruction fetching, decoding, and executing, and allows for cycle-accurate simulation. To simulate a multiprocessor system, a popular method is to integrate multiple ISSs into a SystemC based simulation backbone. Many simulators are built following this solution, including the commercial simulators from CoWare [5] and Synopsys [12] and academic simulation platforms like MPARM [29]. Such simulators are able to execute target binary, boot real RTOSs, and provide cycle-accurate performance data. However, ISSs have the major disadvantage of extremely low simulation speed and very high complexity. Consequently, software simulation is often a bottleneck of the overall simulation performance. This long-running simulation can be afforded only in the final stage of the development cycle, when the design space is significantly narrowed down.

For system-level DSE, an ISS covers too many unnecessary details. Rather, techniques that allow for fast simulation with enough accuracy for making high level design decisions are more desirable. Trace-based simulation methods are in this category. Pimentel et al. [78] use a set of coarse grained traces to represent the workload of each task. Each trace corresponds to a function or a high-level operation, and is associated with a latency, measured using an ISS. The coarse grained traces filter out many intra-trace events and may lead to inaccurate simulation of the whole system. Simulations using fine-grained traces, as presented in [102], are more accurate. However, the trace-driven simulations still have the common drawback that traces are not able to capture a system's functionality. Furthermore, the execution of most tasks is data-dependent, but traces can represent only the workload of one execution path of a program.

To get a high-level simulation model that captures both the function and data-dependent workload of a software task, software simulation techniques based on *native execution* have been proposed. The common idea behind them is to generate, for each program, a simulation model that runs directly on the host machine but can produce performance statistics of a target execution. To generate such a simulation model, three issues are to be tackled: functional representation, timing analysis, and coupling of the functional representation and the performance model. The existing native execution based techniques are differentiated with each other by the choice of the functional representation level, which could be the source level, the intermediate representation (IR) level, or the binary level. To get accurate timing information, the timing analysis should be performed on binary code, which will be finally executed on the target processor, using a performance model that takes the important timing effects of the target processor into account. Coupling of the functional representation and the performance model is realized by annotating the functional representation with the obtained timing information. The back-annotated timing information can generate accurate delays at simulation run-time.

Binary level simulation (BLS) that gets functional representation by translating target binary into a high level programming language or host binary, is proposed in the last decade. A binary level simulator is often called *compiled ISS* in some papers. It offers much faster simulation than interpretive ISSs by performing time-consuming instruction fetching and decoding prior to simulation. The simulation speed is in the hundreds of MIPS range for a single processor.

Recently, *source level simulation (SLS)* is widely used for system-level design, because of its high simulation speed and low complexity. A source level simulator uses source code as functional representation. Timing information that represents execution delays of the software on the target processor is inserted into the source code according to the mapping between the source code and the generated machine code. Many early SLS approaches have the disadvantage of low simulation accuracy. Some use coarse-grained timing information. Others use fine-grained timing information but do not take into account the timing effects of the processor microarchitecture. Therefore, these approaches achieve a high simulation speed at the expense of simulation accuracy.

In our work, we developed a SLS approach, called SciSim (Source code instrumentation based Simulation) [100, 101]. Compared to previously reported SLS approaches, SciSim allows for more accurate performance simulation by employing a hybrid method for timing analysis. That is, timing analysis is performed on target binary code, both statically for pipeline effects and dynamically for other timing effects that cannot be resolved statically. In SciSim, we also use a simple way to get information on the mapping between source code and binary code. We extract the mapping information from debugging information, which can be dumped automatically during cross-compilation. Thus, the whole approach can be fully automated without the need of any human interaction.

However, the SLS technique, also including our SciSim approach, still has a major disadvantage in that it cannot estimate some compiler-optimized code accurately, because after optimizing compilation it is hard to find an accurate mapping between source code and binary code, and even when the mapping can be found, due to the difference between source level control flows and binary level control flows the timing information back-annotated according to the mapping cannot be aggregated correctly during the simulation. This is especially true for control-dominated software, the control flows of which will be significantly changed during optimizing compilation. Since, in reality, software programs are usually compiled with optimizations, this drawback strongly limits the usability of SLS.

Therefore, we developed another approach called iSciSim (intermediate Source code instrumentation based Simulation) [95, 97]. iSciSim is based on SciSim but overcomes the main limitation of SciSim. It allows for accurate simulation of compiler-optimized software at a simulation speed as fast as native execution. The idea behind the iSciSim approach is to transform the source code of a program to code at another representation level that is low enough, so that the new code has a structure close to that of the binary code and thus allows for accurate back-annotation of the timing information obtained from the binary level. This kind of middle-level code is called *intermediate source code* (ISC), to be differentiated from the original source code. Compared with the existing software simulation techniques, including ISS, BLS, and SLS, iSciSim achieves the best trade-off for system-level design, concerning accuracy, speed and complexity.

SciSim is still useful, in case compiler-optimizations are not required or the software to be simulated is data-intensive and has relatively simple control flows. The simple control flows of such data-intensive software programs will not be changed much during optimizing compilation, and therefore, timing information can still be accurately back-annotated into the source code. It is our future work to address the mapping problems in SciSim to make it able to simulate more programs accurately. In comparison to iSciSim, an advantage of SciSim is that annotated source code is more readable than annotated ISC and thus allows for easier debugging during simulation.

1.1.2 Software Performance Simulation in System-Level Design Space Exploration

In Figure 1.1 we show a system-level DSE flow, where DSE is based on simulation using SystemC transaction level models (TLMs). SystemC is currently the most important System Level Design Language (SLDL). It supports system modeling at different levels of abstraction and allows hardware/software co-simulation within a single framework. TLM is a widely used modeling style for system-level design and is often associated with SystemC. In this modeling style, communication architectures are modeled as channels, which provide interfaces to functional units. Modeling effort can be reduced, if different communication architectures support a common set of abstract interfaces. Thus, communication and computation can be modeled separately. SystemC and TLM are introduced more in Section 2.5.4.

Many SLD approaches or toolsets follow a similar flow to the one shown in Figure 1.1 for DSE, for example, Embedded System Environment (ESE) from UC Irvine [6], the commercial tool CoFluent Studio [4], SystemClick from Infineon [88], and the approaches introduced in [53, 79].

The flow consists of four steps: specification, computation design, communication design and design space exploration. In the specification step, a system's functionality and the platform architecture are captured in an application model and a platform model, respectively, usually in a graphical modeling environment. The application model is usually a hierarchical composition of communicating processes, which expresses the parallelism of the application. The platform model specifies capability and services of the hardware platform. It is expressed as a netlist of processors, memories, and communication architecture. Then, the processes can be mapped to the processors, either manually or automatically, and the channels connecting the processes are mapped to the communication architecture.

The design environments or toolsets mentioned above are different from each other in terms of specification environments. ESE and CoFluent Studio are based on their own graphical modeling environments. SystemClick uses a language Click as its modeling foundation and design approaches introduced in [53, 79] use Simulink.

Our work on software performance simulation covers the computation design step. Computation design is actually a process of generating software performance simulation models. The input is application tasks generated from the application model, and the output is scheduled/timed application TLMs. This is achieved by two steps. The first step is timing annotation. Here, the application code is annotated with timing information given the information on which task is mapped to which processor. In this sense, a timed application TLM is actually a native

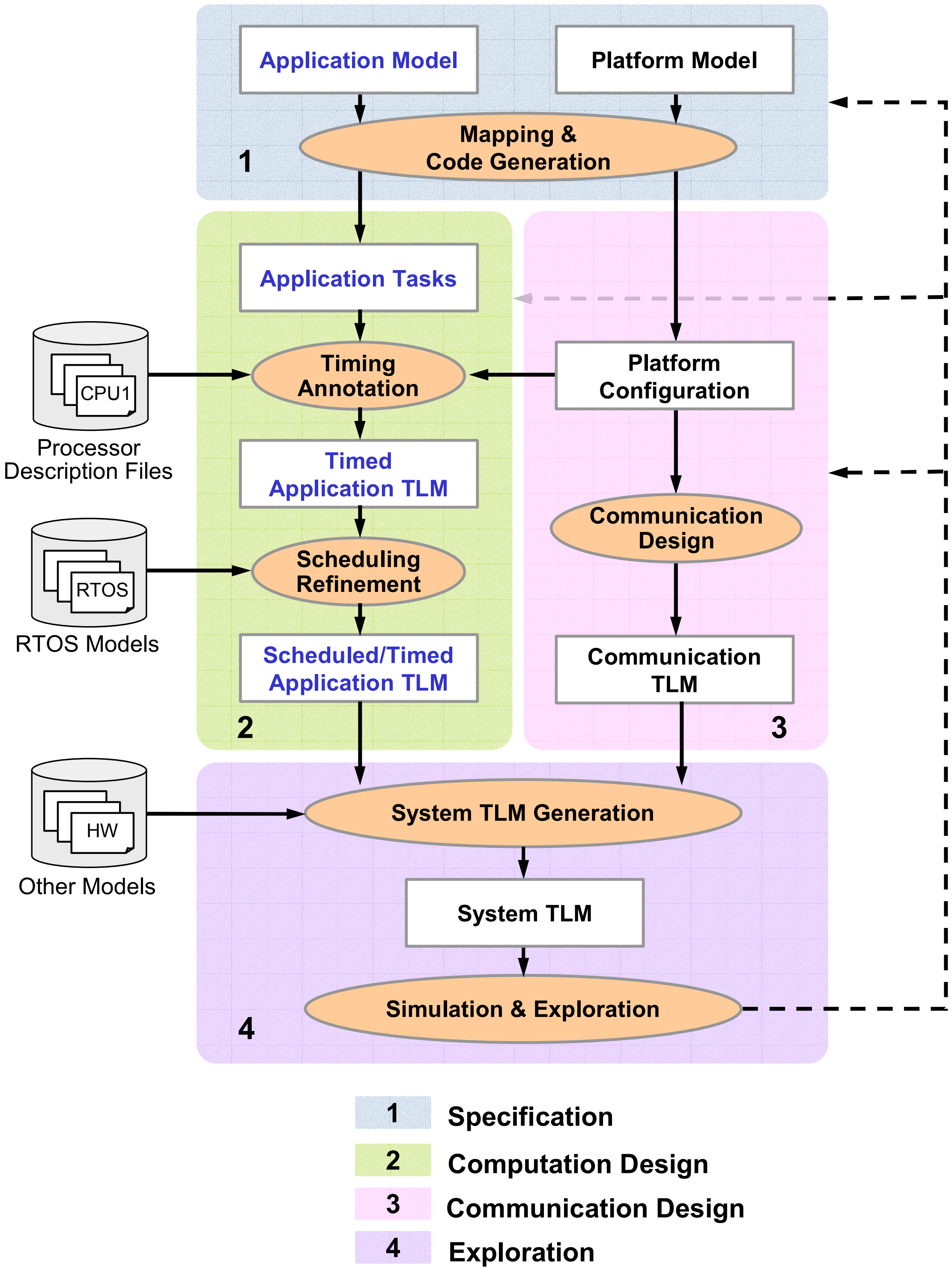


Figure 1.1: System Level Design Flow

execution based simulation model. In the second step, if dynamic scheduling is used, the application TLMs are scheduled using an abstract RTOS model.

The reason why native execution based simulation is used instead of ISSs has been discussed in Section 1.1.1. Among native execution based simulation techniques, source level simulation (SLS) is most suitable for system-level software simulation. The design frameworks or toolsets mentioned above all use source code instrumentation to get timed application TLMs. However, as mentioned already, SLS has a major problem raised by compiler optimizations. Here, our iSciSim can provide a better support for computation design. The application TLMs generated by iSciSim contain fine-grained, accurate timing annotations, taking the compiler-optimizations into account.

If multiple tasks are mapped to a single processor, an RTOS is needed to schedule the task execution dynamically. It is important to capture this scheduling behavior during system-level DSE. This can be achieved by combining application TLMs and an abstract RTOS model in SystemC. This is called scheduling refinement. In our work, we studied how to get an efficient combination of fine-grained timing annotated application TLMs and an abstract RTOS model. In the previous works mentioned above, only ESE has a description of its RTOS model in [104]. The other works either address only static scheduling like SystemClick or do not provide details about their RTOS models. Compared to the RTOS model of ESE, our work achieved a more modular organization of timed application TLMs and the RTOS model.

The main output of communication design is a communication TLM. As the communication TLMs provide a common set of abstract interfaces, the application TLMs can easily be connected to them to generate a TLM of the whole system. The design space exploration is based on simulation using the TLM, and the obtained design metrics are used to guide design modification or refinement.

1.1.3 Worst-Case Execution Time Estimation

Many embedded systems are hard real-time systems, which are often safety-critical and must work even in the worst-case scenarios. Although simulative DSE methods are able to capture real workload scenarios and provide accurate performance data, they have limited ability to cover corner cases. Hence, some analytical methods are also needed for hard real-time systems design. An example of analytical methods is the *Network Calculus* based approach described in [92]. It uses performance networks for modeling the interplay of processes on the system architecture. Another example is SymTA/S [52], which uses formal scheduling analysis techniques and symbolic simulation for performance analysis. Such analytical DSE methods are usually based on knowing worst-case execution times (WCETs) of the software tasks. Hence, bounding the WCET of each task is essential in hard real-time systems design.

Today, static analysis still dominates the research on WCET estimation. Static analysis does not execute the programs, but rather estimates the WCET bounds in an analytical way. It yields safe WCET bounds, if the system is correctly modeled. Typically, the static WCET analysis of a program consists of three phases: (1) *flow analysis* for loop bounding and infeasible path detection, (2) *low-level timing analysis* to determine instruction timing, and (3) finally, *WCET calculation* to find an upper bound on the execution time given the results of flow analysis and low-level timing analysis.

There exists a large amount of previous work addressing different aspects of WCET estimation. We will mention some in Chapter 7. For an extensive overview of previous work, the paper [103] is a very good reference.

In [103], the authors also point out some significant problems or novel directions that WCET analysis is currently facing. They are listed as follows:

- Increased support for flow analysis
- Verification of abstract processor models
- Integration of timing analysis with compilation
- Integration with scheduling
- Integration with energy awareness
- Design of systems with time-predictable behavior
- Extension to component-based design

Our work mainly studied flow analysis, which, as shown above, faces some problems. The flow analysis problems are mentioned several times in [103]. In the state-of-the-art WCET estimation methods, flow analysis is performed either on source code or binary code. It is more convenient to extract flow facts (i.e., control flow information) from the source code level, where the program is developed. However, as timing analysis and WCET calculation are usually performed on binary code that will be executed on the target processor, the source level flow facts must be transformed down to the binary code level. Due to the presence of compiler optimizations, the problem of this transformation is nontrivial. If flow analysis is performed on binary code, the obtained flow facts can be directly used for WCET calculation without the need of any transformation. However, flow analysis often needs the input of some information that cannot be calculated automatically. Such information is usually given in the form of manual annotations. Manual annotation at the binary code level is a very error-prone task. In addition, there are some previous works that propose to use a special low-level IR for flow analysis. This forces software developers to use a special compiler and therefore has limited usability in practice.

We propose to perform flow analysis on ISC [96]. For flow analysis, ISC has the following advantageous features: (1) It contains enough high-level information for necessary manual annotations; (2) It has a structure close to that of binary code for easy flow facts transformation; (3) ISC is generated from standard high-level IRs of standard compilers, so the approach is not limited to a special compiler. These features make flow analysis on ISC achieve a better trade-off between visibility of flow facts and simplicity of their transformation to the binary code level. Furthermore, the approach is easy to realize and no modification of compiler is needed.

1.1.4 A Design Framework for Automotive Systems

In the BASE.XT project we first developed a new formal modeling language COLA (the COmponent LAnguage) [65] and a modeling environment based on it for automotive software development. The modeling process consists of three levels as shown in Figure 1.2:

- Feature architecture: In this first step, the functional requirements are captured in a feature model. The hierarchical nature of COLA allows for decomposition of features into sub-features. The work on feature modeling is done by Sabine Rittmann and is introduced in detail in her Ph.D. thesis [86].
- Logical architecture: The feature model is converted to a functional model in logical architecture throughout several steps of model transformation and rearrangement, semi-automatically. The target of this modeling step is to describe the complete functionality of a system by means of stepwise decompositions. Here, several tools are integrated to help to remove modeling errors, e.g., a type inference tool [66] that detects and diagnoses errors at interconnected component interfaces and a model checker that verifies the conformance of a design to requirements expressed in SALT (Structured Assertion Language for Temporal Logic) [28]. The formal semantics of COLA enables these tools to perform automatic analysis.

  At this modeling level, I contributed a SystemC code generator that generates SystemC code from COLA models. The generated SystemC code allows for functional validation and approximate performance estimation early at the model level before C code can be generated [94].
- Technical architecture: The technical architecture bridges the functional model and implementation. In the technical architecture, units of the functional model are grouped into clusters. On the other hand, the hardware platform is modeled as an abstract platform (also called a hardware model), which captures the platform capability. Stefan Kugele and Wolfgang Haberl have developed an automatic mapping algorithm, which maps the application clusters onto the abstract platform [63, 64]. Here, my work was to integrate

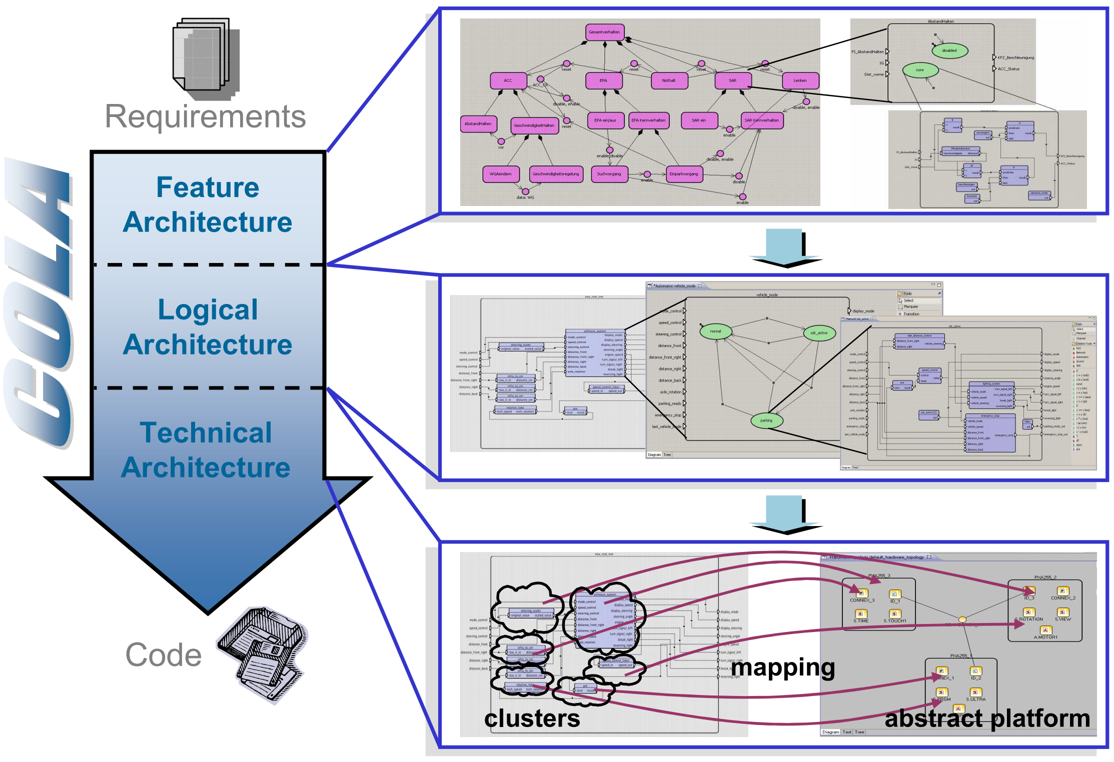


Figure 1.2: The COLA-based Modeling Environment

my WCET analysis tool to provide a WCET bound of each cluster on each possible processing element. The automatic mapping algorithm is based on the WCETs.

After finding a proper mapping that fulfills the requirements, C code can be automatically generated [49]. Although the project is focused on software development, I also researched on hardware implementation of COLA models [99]. The generation of both C and VHDL code relies on a set of template-like translation rules.

As shown, the modeling environment covers the whole process starting from functional requirements to distributed software code, with successive model refinements and supported by a well-integrated toolchain. However, it lacks an environment for systematic design space exploration. The automatic mapping can be regarded as an analytical way of DSE, but it works well only when the target platform is known, the WCET of each cluster is accurately estimated, and the mapping problem is correctly formalized. This is often not the case. Our experiences tell us that we cannot totally trust analytical methods, because, first, not all the design problems can be accurately expressed by mathematical models, and second, today's static WCET estimation techniques often produce overly pessimistic estimates and lead to over-design.

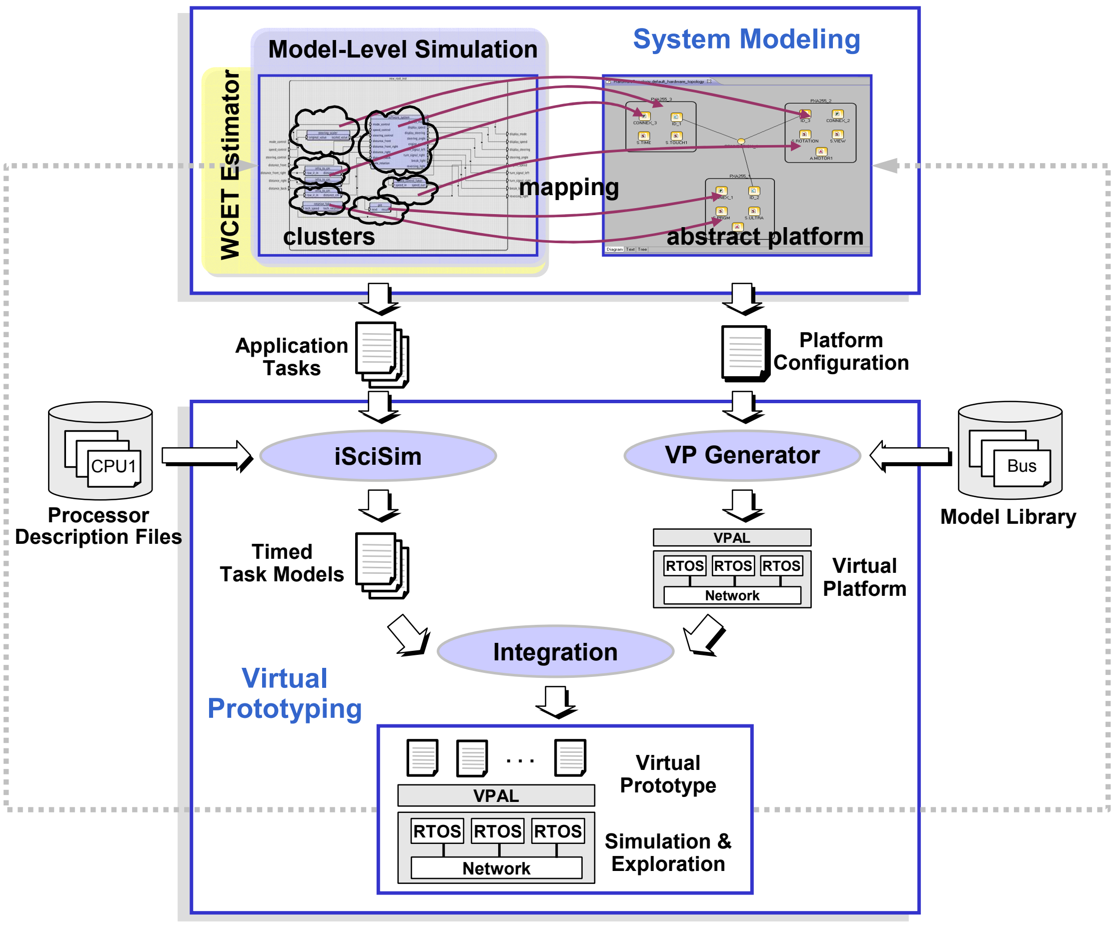


Figure 1.3: Extensions Made to the COLA Environment

Therefore, we constructed a virtual prototyping environment based on SystemC. Here, *virtual prototyping* is defined as a process of generating a simulation model that emulates the whole system under design. We focus on system-level design and use SystemC TLMs. In this sense, the virtual prototyping approach actually covers computation and communication design shown in Figure 1.1. One difference is that a virtual prototype is divided into a virtual platform that captures the dynamic behavior of the system platform under design and software tasks that run on the virtual platform, instead of being divided into application TLMs and a communication TLM. A virtual platform contains RTOS models, communication models, and a *virtual platform abstraction layer* (VPAL) on the top of them, as shown in Figure 1.3. The VPAL is aimed at wrapping the whole virtual platform and reducing the effort of virtual prototyping. Using this layer, the virtual platform can be regarded as a functional entity that provides a set of services, from the application's perspective. The tasks can be simulated on different virtual platforms without the need of changing any code. Only adaptation of the VPAL to the new platform is needed. VPAL also parses configuration information generated from the abstract platform to configure the virtual platform automatically.

The extensions I made to the COLA-based modeling environment are shown in Figure 1.3, including a WCET estimator, a model-level simulation environment and a virtual prototyping environment. Note that model-level simulation and virtual prototyping are two different approaches, used in different design phases, although both are based on SystemC. SystemC models for model-level simulation are generated from COLA application models by means of one-to-one translation. The syntactic structure and semantics of COLA models are preserved by the generated SystemC models. Model-level simulation is used for functional validation and approximate performance estimation at an early design phase when the application model is still under development. Whereas, software simulation models for virtual prototyping are generated by iSciSim after application code is generated from COLA models. Virtual prototyping is used for functional validation of the generated code, accurate performance evaluation, and design space exploration of the whole system.

1.2 Summary of Contributions

The contributions of this dissertation are summarized as follows:

- We introduce a hybrid timing analysis method to take into account the important timing effects of processor microarchitecture in high level software simulation models. Hybrid timing analysis means that timing analysis is performed both statically and dynamically. Some timing effects like pipeline effects are analyzed at compile-time using an offline performance model and are represented as delay values. Other timing effects like the branch prediction effect and the cache effect that cannot be resolved statically are analyzed at simulation run-time using an online performance model.

- We present the SciSim approach, which is a source level simulation approach. SciSim employs the hybrid timing analysis method mentioned above. In SciSim, timing information is back-annotated into application source code according to the mapping between source code and binary code described by debugging information. Here, timing information includes delay values obtained from static analysis and code that is used to trigger dynamic timing analysis at simulation run-time.

- We present the iSciSim approach, which extends SciSim to solve problems raised by compiler-optimizations during source code timing annotation. It extends SciSim by adding a step of transforming original source code to ISC. ISC has accounted for all the machine-independent optimizations and has a structure close to that of binary code, and thus, allows for accurate back-annotation of timing information. The same as SciSim, iSciSim also uses the hybrid timing analysis method.

- In both SciSim and iSciSim, data cache simulation is a problem, because target data addresses are visible in neither source code nor ISC. We propose a solution to use data addresses in the host memory space for data cache simulation. It has been validated that the data of a program in the host memory and in the target memory has similar spatial and temporal locality, if the host compiler and the cross-compiler are similar.
- We introduce an abstract RTOS model that is modular and supports better interactions with fine-grained timing annotated task models. To achieve better modularity for the purpose of module reuse, we implement the RTOS model as a SystemC channel. Task models are wrapped in a separate SystemC module. Implemented in this way, the synchronization between tasks and the RTOS cannot be realized easily using events. We use a novel method to solve this synchronization problem.
- We propose to perform WCET analysis on ISC to get a better support for flow analysis. Flow analysis on ISC achieves a better trade-off between visibility of flow facts and simplicity of their transformation to the binary code level. The whole WCET analysis approach is also easy to realize without the need of compiler modification.
- We introduce a method for model-level simulation of COLA. The model-level simulation enables functional validation and approximate performance evaluation at a very early design phase to help in making early decisions regarding software architecture optimization and partitioning. COLA models are essentially abstract and cannot be simulated directly. Due to similar syntactic structures of SystemC and COLA models, we make use of SystemC as the simulation framework. We developed a SystemC code generator that translates COLA models to SystemC automatically, with the syntactic structure and semantics of COLA models preserved.
- We present the virtual prototyping environment in the SysCOLA design framework. Virtual prototyping is aimed at design space exploration and functional validation of application code generated from COLA, using SystemC-based simulation. The virtual prototyping environment has two advantageous features: (1) it integrates iSciSim, which, together with the C code generator, can generate fast and accurate software simulation models automatically from COLA models; (2) it employs the concept of virtual platform abstraction layer, which abstracts the underlying virtual platform with an API and configures the virtual platform automatically according to the configuration information generated from the abstract platform.

1.3 Outline of the Dissertation

The organization of this dissertation is given in the following:

- Chapter 2, *Background*, gives a detailed introduction to the background of this work. In this chapter, we introduce the definition and trends of embedded systems, present traditional embedded systems design and design challenges, describe the basic concepts of system-level design, give a short survey of system-level design frameworks, and introduce SystemC and transaction level modeling.

- Chapter 3, *Execution-Based Software Performance Simulation Strategies*, provides an introduction to four software simulation techniques and their related works. The four simulation techniques are instruction set simulation (ISS), binary level simulation (BLS), source level simulation (SLS) and IR level simulation (IRLS). We give a discussion about the pros and cons of each technique, which serves as a motivation of our performance simulation methods introduced in Chapter 4 and 5.

- Chapter 4, *SciSim: A Source Level Approach for Software Performance Simulation*, introduces some basic information about software compilation, optimization, and control flow graphs, gives an introduction to the SciSim approach, presents some experimental results to show the advantages and limitations of SciSim, explain the causes of the limitations, and finally summarizes these advantages and limitations.

- Chapter 5, *iSciSim for Performance Simulation of Compiler-Optimized Software*, presents the details of the work flow of iSciSim. The work flow consists of ISC generation, ISC instrumentation, and simulation. Two levels of simulation are introduced: microarchitecture-level simulation of processor's timing effects and macroarchitecture-level simulation of multiprocessor systems using software TLMs generated by iSciSim. We show experimental results to compare iSciSim with three other simulation techniques including ISS, BLS and SciSim, and demonstrate a case study of designing MPSoC for a Motion JPEG decoder to show how iSciSim facilitates MPSoC design space exploration.

- Chapter 6, *Multi-Task Simulation in SystemC with an Abstract RTOS Model*, introduces the basic functionality of RTOSs and presents our abstract RTOS model.

- Chapter 7, *Flow Analysis on Intermediate Source Code for WCET Estimation*, shows how the usage of ISC facilitates flow analysis for WCET estimation of compiler-optimized software and how to simply transform the ISC level flow facts to the binary level using debugging information. In addition, in this chapter, we also propose an experiment method to demonstrate only the effectiveness of flow analysis. This allows us to evaluate a flow analysis method and a timing analysis method separately.

- Chapter 8, *The SysCOLA Framework*, presents the COLA-based modeling environment and SystemC-based virtual prototyping environment and show the roles the software performance estimation tools play in the framework. A case study of designing an automatic parking system is demonstrated.

- Chapter 9, *Model-Level Simulation of COLA*, proposes model-level simulation for functional validation and performance estimation of designs captured in COLA in an early design phase before application source code is generated and gives a detailed description of SystemC code generation from COLA models.
- Chapter 10, *Conclusions and Future Work*, summarizes this dissertation and outlines the directions of the future work.

All these chapters are grouped into four parts. Part I, *Introduction and Background*, includes Chapter 1 and 2. Chapter 3, 4, 5, 6, and 7 are all about software performance estimation methods and are grouped to Part II, *Software Performance Estimation Methods.* Part III, *Software Performance Estimation in a System-Level Design Flow*, includes Chapter 8 and 9, both on the work done in the scope of SysCOLA. Part IV, *Summary*, contains Chapter 10.

Chapter 2

Background

In this chapter, we first give the definition of embedded systems and show their market size in Section 2.1. Next, we show the trends of increasing complexity in embedded applications and computing platforms in Section 2.2. Following that, we present traditional embedded systems design and design challenges in Section 2.3 and Section 2.4, respectively. These motivate system-level design methodologies. Then, in Section 2.5, we describe the basic concepts of system-level design (SLD), present SLD flows, give a survey of SLD frameworks, and introduce SystemC and transaction level modeling.

2.1 Embedded Systems: Definition and Market Size

Embedded systems are computer systems that are embedded as a part of larger machines or devices, usually performing controlling or monitoring functions. They are a combination of computer hardware and software, and perhaps some additional mechanical parts. Embedded systems are usually microprocessor-based and contain at least one microprocessor, performing the logic operations. Microprocessors are far more used in embedded systems than in general purpose computers. It is estimated that more than 98% microprocessor chips manufactured every year are used in embedded systems [73].

Embedded systems are ubiquitous in our everyday lives. Their market size is huge. According to a report "*Scientific and Technical Aerospace Reports*" from National Aeronautics and Space Administration (NASA) published in 2006 [19], the worldwide embedded systems market was estimated at $31.0 billion, while the general-purpose computing market was around $46.5 billion. However, the embedded systems market grows faster and would soon be larger than the general-purpose computing market. According to another more recent report, "*Embedded Systems: Technologies and Markets*" available at Electronics.ca Publications [7], the embedded systems market was estimated at $92.0 billion in 2008 and was expected to grow at an average annual growth rate of 4.1%. By 2013, this market will reach $112.5 billion.

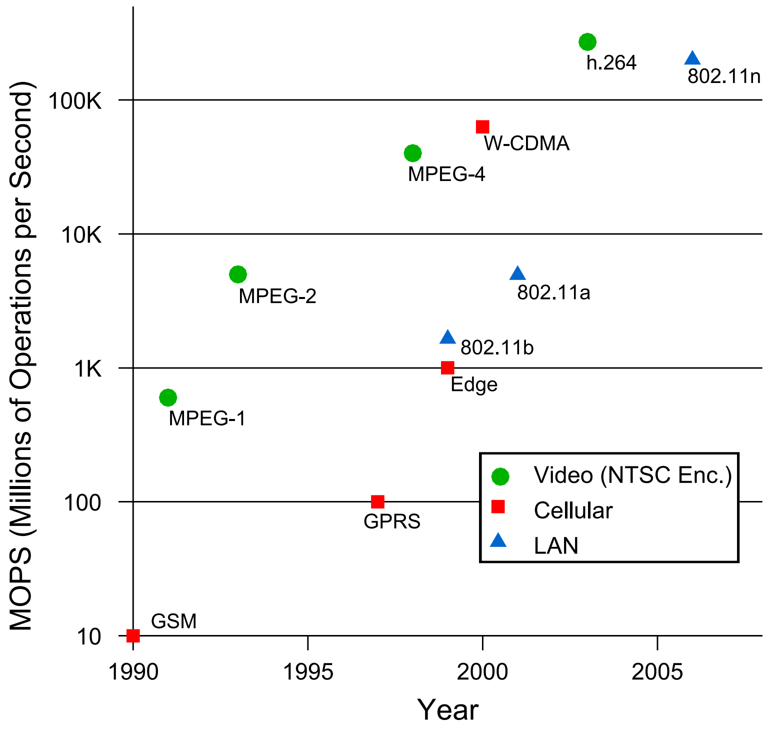


Figure 2.1: Increasing Application Complexity due to Upgrade of Standards and Protocols (source: [37])

2.2 Embedded System Trends

2.2.1 Application Trends

Over the past several decades, the demand for new features in embedded system products has been ever increasing, driven by market needs. Let's take mobile phones as an example. Twenty years ago, a mobile phone has usually only the functionality of making voice calls and sending text messages. Today, a modern mobile phone offers the user much more capabilities. It has hundreds of features, including camera, video recording, music (MP3) and video (MP4) playback, alarms, calendar, GPS navigation, email and Internet, e-book reader, Bluetooth and WiFi connectivity etc. Many mobile phones run complete operating system software providing a standardized interface and platform for application developers. Another example is modern cars. The features in high class cars are also increasing exponentially. Many new cars are featured night vision systems, autonomous cruise control systems, and automatic parking systems etc. It is estimated that more than 80 percent of all automotive innovations now stem from electronics.

Besides, upgrade of standards and protocols also increases application complexity and computational requirements in some application domains like the multimedia domain and the communication domain. Figure 2.1 from [37] shows such trends.

For example, the change of video coding standard from MPEG-1 to MPEG-2 has resulted in around 10 times increase in computational requirement.

2.2.2 Technology and Architectural Trends

The rapid growth in application complexity places even greater demands on the performance of the underlying platform. In the last decades, parallel architectures and parallel applications have been accepted as the most appropriate solution to deal with the acute demands for greater performance. On a parallel architecture, the whole work is partitioned into several tasks and allocated to multiple processing elements, which are interconnected and cooperate to realize the system's functionality.

The architectural advance is primarily driven by the improvement of semiconductor technology. The steady reduction in the basic VLSI feature size makes much more transistors can fit in the same chip area. This improvement has been following Moore's Law for more than half a century. Moore's Law states that the number of transistors on a chip doubles roughly every two years. Figure 2.2 illustrates Moore's Law through the transistor count of Intel processors over time. The same trend applies to embedded processors and other chips.

In the following, we introduce two typical parallel architectures that are widely used in the embedded systems domain.

Distributed Embedded Systems

In a distributed embedded system, tasks are executed on a number of processing nodes distributed over the system and connected by some interconnect network such as fieldbuses. The number of processing nodes ranges typically from a few up to several hundred. An example of distributed system is the network of control systems in a car. With reducing cost and increasing performance of microprocessors, many functions that were originally implemented as mechanical systems in a car are now realized in electronic systems. As the very first embedded system in the automotive industry, the Volkswagen 1600 used a microprocessor in its fuel injection system [18], in 1968. Today, in a car like BMW 7-series over 60 processors are contained, in charge of a large variety of functions.

Compared with the traditional centralized solution, the distributed solution suits better for systems with distributed sensors/actuators. Distributed systems have also the advantage of good scalability by using off-the-shelf hardware and software building blocks. In addition, as the processing nodes are loosely coupled, a system can be clearly separated into clusters, being developed by different teams.

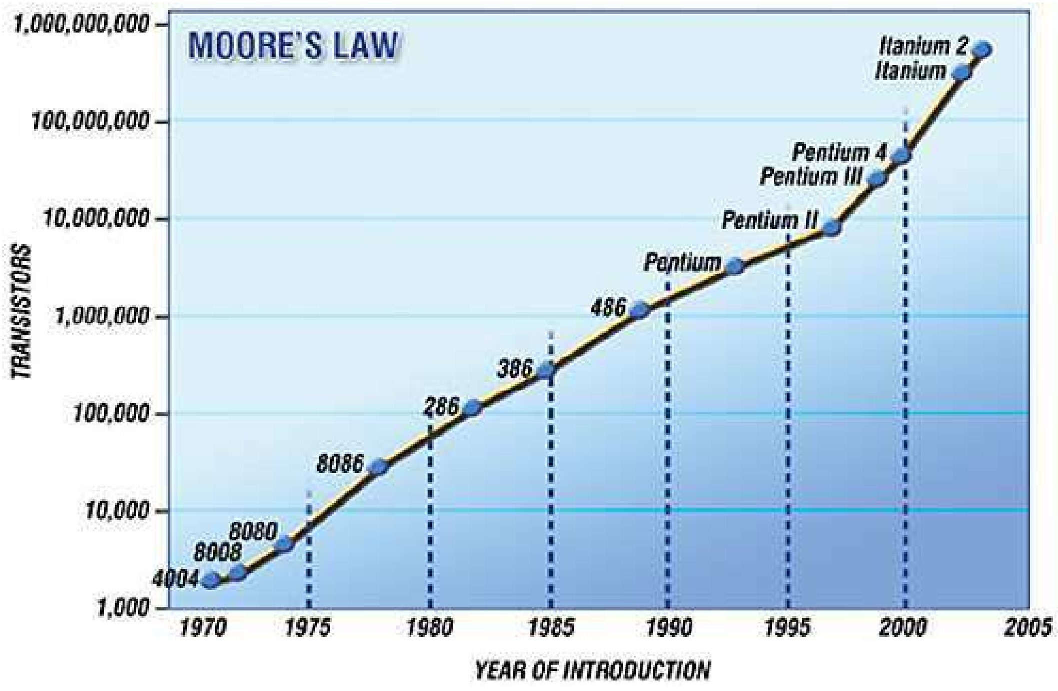


Figure 2.2: Moore's Law

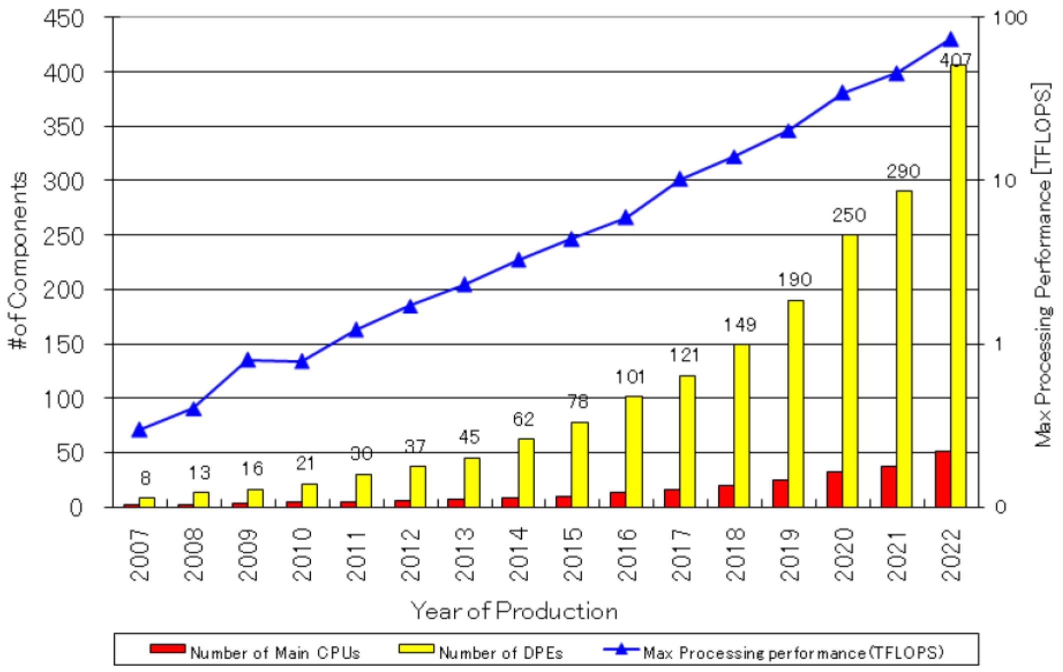


Figure 2.3: SoC Consumer Portable Design Complexity Trends (source: ITRS 2008)

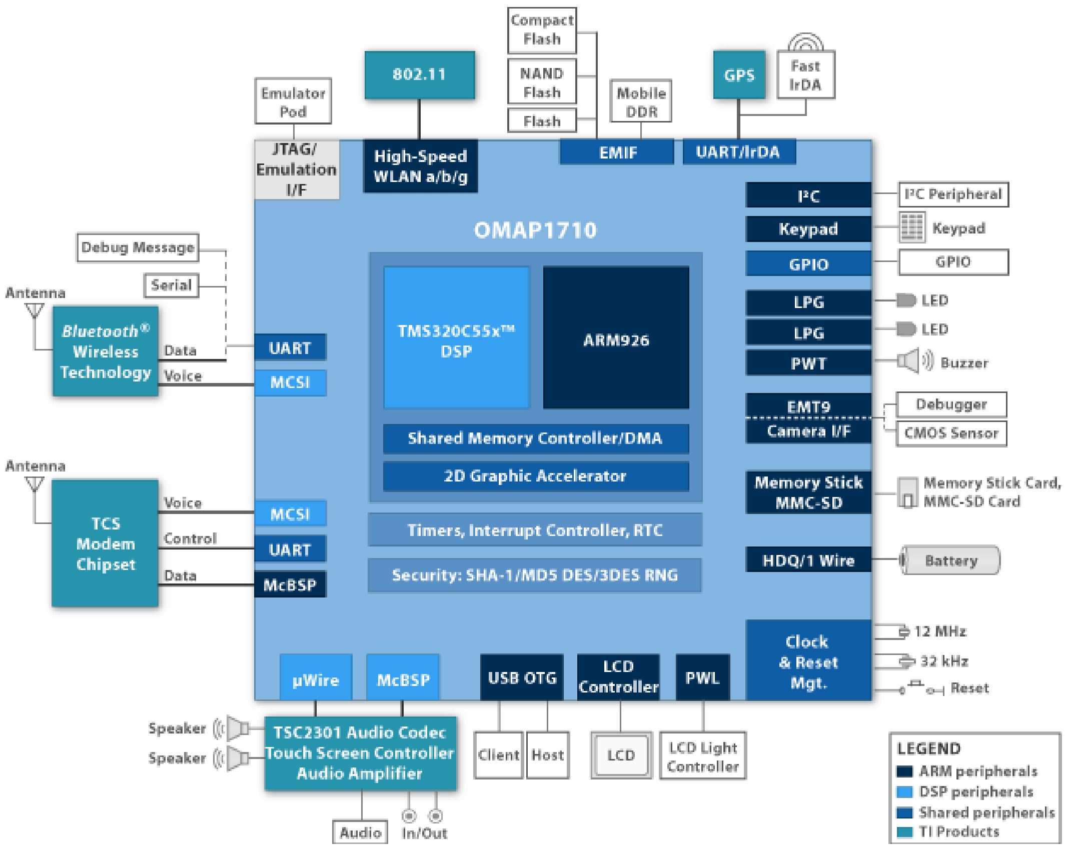


Figure 2.4: TI OMAP 1710 Architecture (source: *http://www.ti.com/*)

Multiprocessor System-on-Chip (MPSoC)

An MPSoC is a processing platform that integrates the entire system including multiple processing elements, a memory hierarchy and I/O components, linked to each other by an on-chip interconnect, into a single chip. With the size of transistors continuously scaling down, it is possible to integrate more and more processor cores into a single chip. So, an MPSoC can be regarded as a single-chip implementation of a distributed system. Chips with hundreds of cores have been fabricated (e.g., Ambric Am2045 with 336 cores and Rapport Kilocore KC256 with 257 cores), and chips with thousands of cores are on the horizon.

Compared to multi-chip systems, MPSoC designs have the advantages of smaller size, higher performance and lower power consumption. They are very widely used in portable devices like cell phones and digital cameras, especially for multimedia and network applications. For example, the OMAP1710 architecture, which is used in the cell phones of Nokia's N- and E-series, is an MPSoC. As shown in Figure 2.4, it contains a microprocessor ARM9, a DSP, hardware accelerators for video and graphics processing, buses, a memory hierarchy and many I/O components. Figure 2.3 shows the number of processing elements (PEs) predicted over the next 15 years in consumer portable devices by the International Technology

Roadmap for Semiconductors (ITRS) 2008 [21]. We can see a great increase in the number of both processors and data processing elements (DPEs) over the next 15 years. By 2022, a portable device like a cell phone or a digital camera may contain up to 50 processors and 407 DPEs. The processing performance of portable devices increases almost in proportion to the number of processing elements.

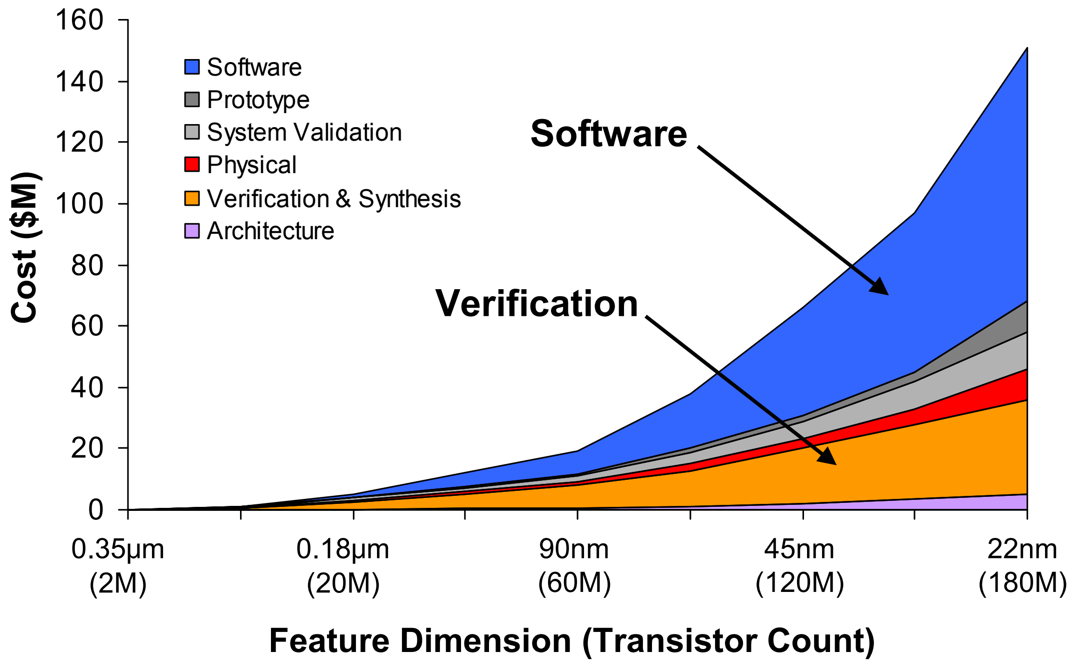


Figure 2.5: Increase of IC Design Cost (source: IBS 2009)

2.2.3 Increasing Software Development Cost

Another remarkable trend is the dramatic increase of software development cost, when software-based implementations are becoming more popular. The reason why more and more functionality has been moving to software is because software-based implementations are cheap and flexible, and in contrast, hardware-based implementations are expensive and time-consuming. Software-based implementations have low cost, because microprocessors have increasing computational power and shrank in size and cost. They are flexible, because microprocessors as well as other programmable devices can be used for different applications by simply changing the programs. In contrast, hardware-based implementations, usually as Application Specific Integrated Circuits (ASICs), have very expensive and time-consuming manufacturing cycles and always require very high volumes to justify the initial expenditure. This limitation makes hardware-based implementations uneconomical and inflexible for many embedded products. In the recent years the number of hardware-based designs has slowly started to decrease. The important parallel architectures such as distributed systems and MPSoCs use processors or processor cores as the main processing elements and are software-centric in nature.

Although software is relatively cheap to develop compared to hardware, it is dominating overall design effort while the portion of software dramatically increasing in complex systems. Several sources confirm that software development costs are rapidly outpacing hardware development costs. One source is from International Business Strategies (IBS) [22], which has studied the distribution of IC design costs at various technologies. In Figure 2.5 from IBS 2009, we can see that the IC design cost has been increasing dramatically with the advance of technologies. Since the 90 nm technology, software development cost accounts for around half of the total cost.

Another source comes from the annual “Design” report of ITRS 2007 [20], which has looked at the software versus hardware development costs for a typical high-end SoC. Its cost chart shows that in the year 2000 $21 million were spent on hardware engineering and tools and $2 million on software development. In 2007, thanks to the improvement of very large hardware block reuse, hardware costs decreased to $15M, but software costs increased to $24M. It is predicted that for 2009 hardware costs are $16M and software costs $30M, and for 2012, hardware costs are $26M and software costs reach $79M.

2.3 Traditional Embedded Systems Design

Traditional embedded systems design views hardware design and software design as two separate tasks. Because hardware and software are designed separately and there lacks efficient communication between hardware designers and software designers due to their different education backgrounds and experiences, this results in a gap between hardware design and software design. This gap is called *system gap* in [45]. The development of hardware synthesis tools in 1980s and 1990s has significantly improved the efficiency of hardware design, but has not narrowed the system gap.

In a traditional design flow, hardware design starts earlier than software design. Register Transfer Level (RTL) is the typical entry point for design. Given an initial and usually incomplete specification, hardware designers translate the specification into an RTL description, which specifies the logical operations of an integrated circuit. VHDL and Verilog are common hardware description languages for RTL designs.

Software design starts typically after hardware design is finished or a prototype of the hardware under design is available. The traditional way of embedded software development is basically the same as that of PC software development, using tools like compilers, assemblers, and debuggers etc. The entry point of software design is manual programming using low-level languages such as C or even assembly languages. This simple flow may suit for software design for uniprocessor. However, currently, it is also used for software design for parallel platforms with an ad-hoc adaptation. This ad-hoc adaptation adds an additional partitioning step, where the system's functionality is subdivided into several pieces and they are assigned to individual processing elements (PEs) before manual coding for each PE. This solution for programming parallel architectures has been proven to be not efficient enough.

To conclude, the traditional design flow has several problems for designing mixed hardware/software embedded systems:

- It lacks a methodology to guide design space exploration. In the traditional design, it is highly based on the designers' experiences to decide what to implement in software components running on processors and what to implement in hardware components. As the system's complexity scales up, this kind of manual partitioning is not able to get an efficient design.
- A design can be tested only when it is implemented or prototyped. Design mistakes or errors found in this very late design phase may cause redesign and considerable changes in the whole system. This kind of redesign is very costly and will significantly delay the time-to-market.
- Manual implementation at low abstraction levels is very error-prone and needs large effort.

A survey conducted by Embedded Market Forecasters [60], a market research company, shows the ineffectiveness of traditional design methodologies: Over 70% of designs missed pre-design performance expectations by at least 30%; Over 30% of designs missed pre-design functionality expectations by at least 50%; About 54% of designs missed schedule, with an average delay of nearly 4 months; Nearly 13% of designs were canceled. These statistics involve a large portion of single-processor systems design. The failure rate could be even worse, if only multiprocessor systems were considered. According to another survey by Collett International [85] on SoC redesign statistics, almost 40% of designs require a respin.

2.4 Design Challenges

As discussed previously, advances in hardware capability enable new application functionality, and meanwhile, the growing application complexity places even greater demands on the architecture. This cycle forms a driving force of the tremendous ongoing design and manufacturing effort in embedded systems. In addition to technical issues, the time-to-market is also a key factor to be considered. To some extent, the time-to-market decides the success of a product. For example, the digital camera PV-DC252 of Panasonic was 7 months later to market than Micro-MV DCR IP7 of Sony, its retail price was 41% less [50]. As a result, although the designs increase dramatically in complexity and require more effort,

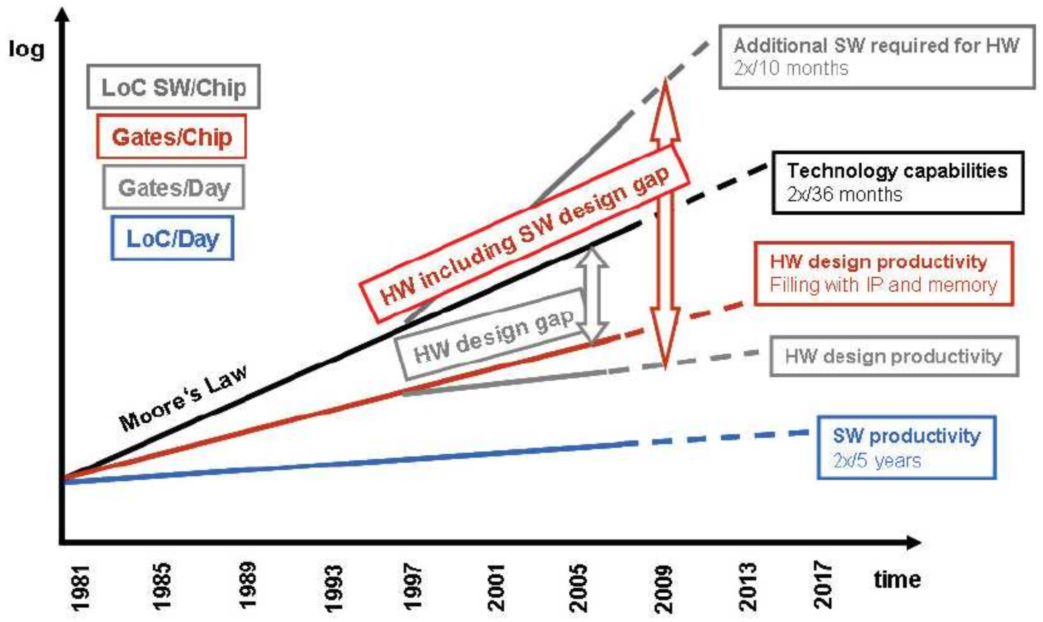


Figure 2.6: Productivity Gaps of Hardware and Software (source: ITRS 2007)

there is nevertheless an increasing demand on the time-to-market to yield competitive products. It is hard to achieve both increase in application complexity and reduction in design time using the traditional design methods.

Therefore, it is the key design challenge to achieve high design productivity under a given time-to-market requirement. Since the IC design productivity is highly limited by the technology, the design challenge is actually to make design productivity keep pace with technological advance (i.e., Moore's Law). However, the inefficiency of the traditional design methodology results in a gap between the technology advance rate and the productivity growth rate, known as *design productivity gap* or *design gap*. This gap is still widening over time. Figure 2.6 from ITRS 2007 [20] shows the design gaps of both hardware and software. It gives the following information:

- The hardware design gap is widening: the capability of technology is currently doubling every 36 months, following Moore's Law, whereas the increase of hardware design productivity is below Moore's Law.
- The *hardware including software* design gap is widening even faster: demand of software required for hardware is doubling every 10 months, whereas the increase of hardware (including software) design productivity is far behind.
- Design productivity for hardware-dependent software is only doubling every 5 years.

To summarize, the design gap is caused by the conflicting demand of increased design complexity and shortened time-to-market. New design methodologies are needed to dramatically close the design gap and to increase the design productivity. One promising design methodology is system level design, which is introduced in the following section.

2.5 System-Level Design

Most agree that rising the abstraction level, at which systems are expressed, is the most efficient solution to address the design complexity and thus to improve the design productivity. The system-level design (SLD) methodology follows this concept. The ITRS claimed that SLD would increase design productivity by 200,000 gates per designer-year and improve productivity by 60% over an "Intelligent Testbench" approach [39]. In this section, we will first discuss about the definition and primary concept of SLD in Section 2.5.1. Then, an introduction to SLD flows and a survey of SLD frameworks are given in Section 2.5.2 and Section 2.5.3, respectively. As the most important System-Level Design Language (SLDL), SystemC is introduced in Section 2.5.4.

2.5.1 Definitions

In general, "system-level design" is not an exactly defined term. A few definitions from different sources are listed in the following:

- **According to International Technology Roadmap for Semiconductors (ITRS):**

  system level is defined as "an abstraction level above the register transfer level (RTL)". At the system-level, silicon resources are defined in terms of abstract functions and blocks; design targets include software (embedded code in high level and assembly language, configuration data, etc.) and hardware (cores, hardwired circuits, buses, reconfigurable cells). "Hardware" (HW) corresponds to implemented circuit elements, and "software" (SW) corresponds to logical abstractions (instructions) of functions performed by hardware. Behavior and architecture are independent degrees of design freedom, with software and hardware being two components of architecture. The aggregate of behaviors defines the system function, while the aggregate of architecture blocks defines a system platform. Platform mapping from system functionality onto system architecture is at the heart of system-level design, and becomes more difficult with increased system complexity and heterogeneity (whether architectural or functional).

- **From Wikipedia (*http://www.wikipedia.org/*) ESL is defined as:**

  an emerging electronic design methodology that focuses on the higher abstraction level concerns first and foremost.

The basic premise is to model the behavior of the entire system using a high-level language such as C, C++, or MATLAB. Newer languages are emerging that enable the creation of a model at a higher level of abstraction including general purpose system design languages like SysML as well as those that are specific to embedded system design like SMDL and SSDL supported by emerging system design automation products like Teraptor. Rapid and correct-by-construction implementation of the system can be automated using EDA tools such as High Level Synthesis and embedded software tools, although much of it is performed manually today. ESL can also be accomplished through the use of SystemC as an abstract modeling language.

- **The book "ESL Design and Verification" [23] defines ESL as:**

  the utilization of appropriate abstractions in order to increase comprehension about a system, and to enhance the probability of a successful implementation of functionality in a cost-effective manner, while meeting necessary constraints.

In the above definitions two similar terms exist: System-Level Design (SLD) and Electronic System-Level (ESL) Design. These two terms mean the same thing and can be used interchangeably. In the book "ESL Design and Verification" [23], it is stated that ESL is a successor to the term SLD and will replace it in the near future. However, to the best of our knowledge, at the time of writing this dissertation the term SLD is still widely being used in both academia and industry. Hence, we also use the term SLD through this dissertation.

The three definitions reach a common ground that SLD addresses many design issues at higher abstraction levels and abstracts away implementation details. Whereas, the definition from ITRS lays emphasis on a design flow including hardware/software co-design and platform mapping from system functionality onto system architecture, while the definition from Wikipedia states more the utilization of design languages and EDA tools.

To summarize, SLD is aimed to simplify the specification, verification and implementation of systems including hardware and software, and to enable more efficient design space exploration, by raising the abstraction level at which systems are modeled and with sufficient support from EDA tools across design flows and abstraction levels.

2.5.2 System-Level Design Flows

According to Daniel D. Gajski [45], a design flow is defined as "a sequence of design steps that are necessary to take the product specification to manufacturing". He suggests a four-step SLD flow: specification, exploration, refinement, and implementation. Whereas, according to the "ESL Design and Verification" book, a complete SLD flow should contain six steps: (1) specification and modeling, (2) pre-partitioning analysis, (3) partitioning, (4) post-partitioning analysis and debug, (5) post-partitioning verification, and (6) HW/SW implementation. These two flows are principally the same, only from different views. The exploration and refinement steps in the former flow may contain the work of pre-partitioning analysis, partitioning, and post-partitioning analysis. Platform-based design [87] advocated by Alberto Sangiovanni-Vincentelli is a similar approach, but lays emphasis more on system components reuse.

In Figure 2.7, we generalize typical system-level design flows using a diagram that extends the famous Y-Chart diagram. Starting from a specification of the target system, an initial design is modeled, with the application tasks and the system platform developed in separate. Then, the application task models are mapped to the available platform resources to get a system level model, which makes up the input for the design space exploration step. Depending on the exploration results, the architecture is iteratively modified until the design requirements are met. Then, the design is iteratively refined to do more accurate exploration and make lower-level decisions. The result of this exploration loop is an implementation model including executable software tasks and a specification of the platform architecture that is used in the subsequent implementation phase.

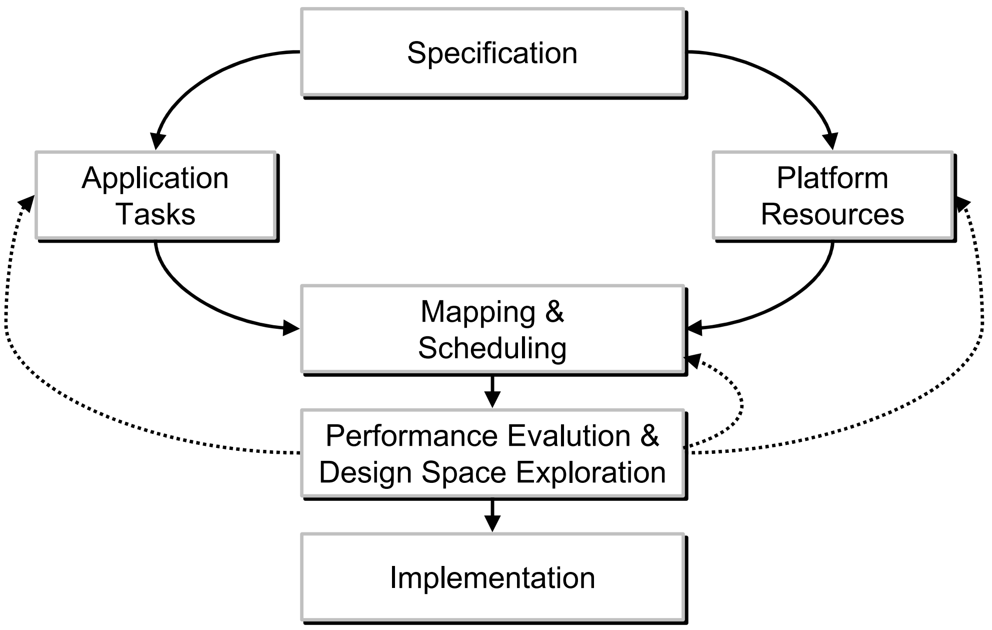


Figure 2.7: System-Level Design Flow

2.5.3 Survey of SLD Frameworks

This section provides an overview of some important work in developing system-level design frameworks. There are too many design frameworks that cover one or more aspects of system-level design, developed either by research activities or commercial companies. Here, we can only pick some to introduce. For an extensive overview, some survey papers like [39] can be referred to.

CoFluent Studio

CoFluent Studio [4] is a commercial SLD toolset for architecture exploration and performance analysis, developed by the company CoFluent Design. Its essential concept is function-architecture separation. It covers the specification and exploration steps in a typical SLD flow but does not provide a link from exploration models to implementation models. In CoFluent Studio, a system is described by graphical models. The graphical blocks of an application model specify only the causal order of tasks but do not provide a mechanism for implementation. The tasks must be programmed manually in C and associated to the graphical blocks. From the graphical models, SystemC models can be generated automatically. The SystemC models are at a high abstraction level, where tasks are wrapped in threads and communication is realized by simple message passing between threads.

CoWare Virtual Platform

CoWare virtual platform [5] is another SystemC based commercial tool for design space exploration and embedded software development, from the company CoWare. In contrast to CoFluent Studio, CoWare virtual platform is aimed at simulation at a low abstraction level, providing an environment for software development. The simulation models describe the microarchitecture of hardware platforms and can provide accurate performance data for design space exploration. However, it does not support early specification and modeling issues.

Embedded System Environment (ESE)

Embedded System Environment (ESE) [6] is a toolset for modeling, synthesis and validation of multiprocessor embedded systems. It is developed by University of California, Irvine. ESE consists of a front-end and a back-end for performance evaluation and automatic synthesis, respectively. The front-end is similar to CoFluent Studio, where the system platform is captured by graphical blocks and the application is written in C/C++ code. For performance estimation, SystemC transaction level models (TLMs) can be automatically generated from the platform model and the application code. The back-end provides automatic synthesis from TLM to RTL. Then, the RTL code can be synthesized using standard synthesis tools. Therefore, ESE covers a complete SLD flow.

Ptolemy

Ptolemy [15] is a framework from University of California, Berkeley. It is aimed at modeling, simulation, and design of concurrent, real-time embedded systems. It focuses on interactions between current components and assembly of them, using heterogeneous mixtures of models of computation (MOC). Hierarchical composition is used to handle heterogeneity. The current version of Ptolemy supports nine MOCs including continuous-time modeling (CT), finite state machines (FSM), and synchronous dataflow (SDF) etc. Other nine MOCs are still experimental. C code can be automatically generated from models constructed within Ptolemy. Ptolemy does not focus on function-architecture separation and mapping, which is usually the heart of a system-level design flow.

POLIS

POLIS [24] is a framework for hardware-software codesign of embedded systems, from UC Berkeley. In POLIS, an embedded system is specified in Esterel, a synchronous programming language. The specification is then translated into CFSMs (Codesign Finite State Machines), which are similar to classical FSMs but are globally asynchronous and locally synchronous. Each element of a network of CFSMs describes a component of the system to be modeled, unbiased towards a hardware or a software implementation. The toolchain of POLIS enables formal verification, co-simulation, hardware-software partitioning, hardware-software synthesis and interface implementation. Because of the model of computation, POLIS is well suited for designing control-dominated systems.

Metropolis

Metropolis [25] is a system-level design framework based on the principles of platform-based design [87] and orthogonalization of concerns [57], also from UC Berkeley. It is aimed at representing heterogeneous systems at different abstraction levels, expressing design problems formally and solving them using automated tools. The current version of Metropolis is based on the Metropolis Metamodel language that can be used to describe function, architecture, and a mapping between the two. It supports a wide range of models of computation.

Compaan/Laura

The Compaan/Laura [91] approach is developed by Leiden University. The input of the design flow is MATLAB specifications, which are then converted to Kahn Process Networks (KPNs) using the tool Compaan. The generated KPNs are subsequently synthesized as hardware and software using the tool Laura and then implemented on a specific architecture platform. The studied platform consisting of an FPGA and a general purpose processor. The software implementation makes use of the YAPI [38] library.

We will introduce our own SLD framework SysCOLA in Chapter 8. SysCOLA covers the whole SLD flow as discussed in Section 2.5.2. In Chapter 8, we will also give a short comparison between SysCOLA and other design frameworks.

2.5.4 SystemC and Transaction Level Modeling

SystemC is defined and promoted by OSCI (Open SystemC Initiative) [13]. It is considered as the most important system-level design language (SLDL). A lot of SLD frameworks are based on SystemC, including many commercial tools such as CoWare Virtual Platform [5], CoFluent Studio [4], AutoESL [2], and tools from Synopsys [12] and Cadence [3]. SystemC has been approved by the IEEE Standards Association as IEEE 1666-2005 [55].

Essentially, SystemC is a C++ class library featuring methods for building and composing SystemC elements. In order to model concurrent system behavior, SystemC extends C++ with concepts used by hardware modeling languages, like VHDL and Verilog.

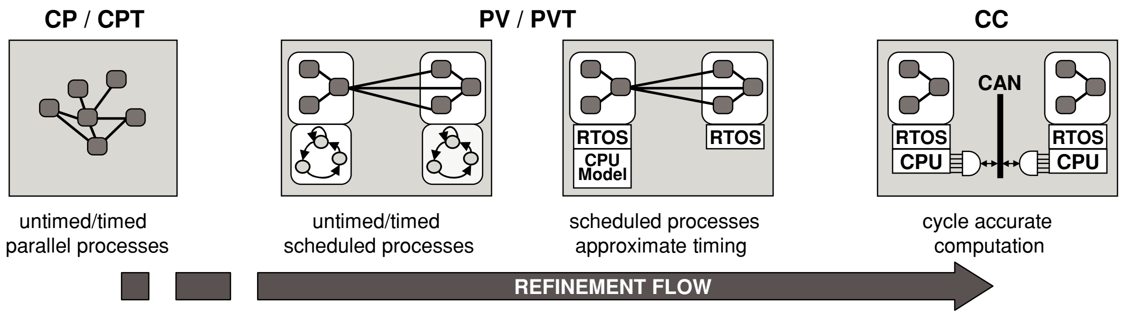

CP / CPT
PV / PVT
CC
CAN
RTOS
CPU Model
RTOS
RTOS
CPU
RTOS
CPU
untimed/timed parallel processes
untimed/timed scheduled processes
scheduled processes approximate timing
cycle accurate computation
REFINEMENT FLOW

Figure 2.8: Computation Refinement (from [62])

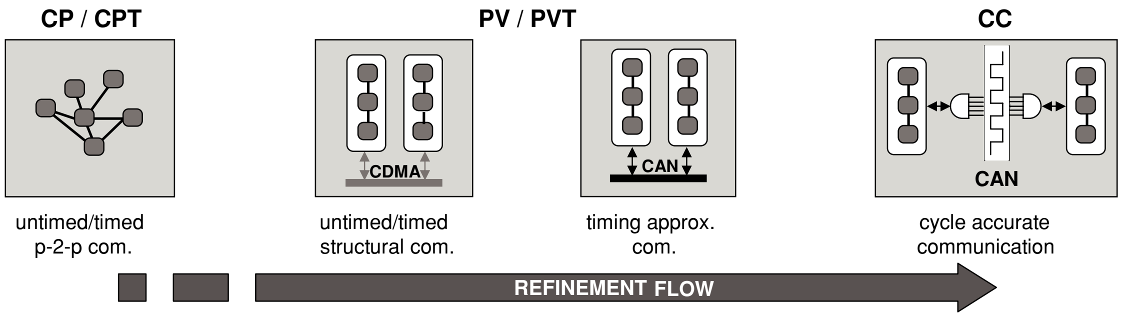


Figure 2.9: Communication Refinement (from [62])

SystemC is often associated with Transaction-Level Modeling (TLM) [33, 40], which is a widely used modeling style for system-level design. In this modeling style, communication architectures are modeled as channels, which provide interfaces to functional units. Modeling effort can be reduced, if different communication architectures support a common set of abstract interfaces. Thus, communication and computation can be modeled separately.

SystemC supports modeling systems at different levels of abstraction, from system level to register-transfer level, and allows to co-simulate software and hardware components within a single framework. The OSCI Transaction Level Working Group has defined seven levels of abstraction supported by SystemC [40]: algorithmic (ALG), communicating processes (CP), communicating processes with time (CP+T), programmer's view (PV), programmer's view with time (PV+T), cycle accurate (CA) and register transfer level (RTL). A system can be modeled at a high level of abstraction, for example ALG, and refined stepwise to a lower level of abstraction, which might be RTL, where simulation is more accurate but simulation time increases. Figure 2.8 and Figure 2.9 from [62] illustrate the respective refinements of computation and communication modeled in SystemC.

Part II

Software Performance Estimation Methods

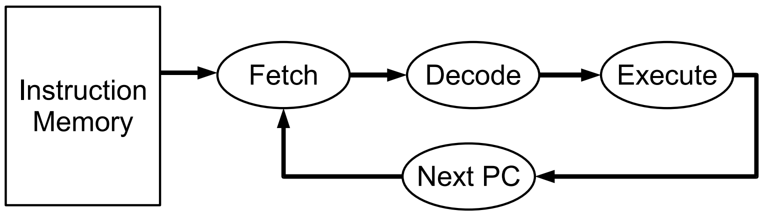
Chapter 3

Execution-Based Software Performance Simulation Strategies

A software simulation model should be able to capture both functional and temporal behaviors of embedded software programs. The functional behavior of a program is usually described by a high level programming language, mostly in the C language in embedded applications, and realized by running the cross-compiled binary code on the target processor. The temporal behavior is expressed by the processor's computation delays caused by executing the program. It is dependent on the cross-compiler, the instruction set architecture (ISA), and the timing effects of the processor microarchitecture. Correspondingly, a software simulation model also consists of two primary components: a functional model and a performance model. The functional model emulates the functionality of programs. The performance model represents the microarchitecture of a specific processor, models the run-time interactions between instructions and the processor components, and provides the corresponding performance statistics. The performance model and the functional model can be decoupled, for example in a trace-driven simulator. However, the performance model is more often built on the functional model. Such simulators are called *execution-based simulators.* Their simulation accuracy depends on both the correctness of functional modeling and the accuracy of performance modeling. The simulation techniques introduced in this chapter, including instruction set simulation (ISS) and native execution based techniques, are all execution based. Native execution based techniques are further categorized into *binary (assembly) level simulation (BLS)*, *source level simulation (SLS)* and *IR level simulation (IRLS)*, in terms of functional representation levels.

3.1 Instruction Set Simulators

Software is usually simulated by instruction set simulators (ISSs) to get the influence of software execution on system performance and study the runtime interactions between software and other system components. An ISS is a piece of software that realizes the ISA of a target processor on the host machine. Most available ISSs are interpretive ISSs. Lying between the program being simulated and the host machine, an interpretive ISS imitates the target processor to fetch, decode and execute target instructions one by one, at run-time, similar to a Java interpreter. Figure 3.1 shows the typical processing loop of interpretive ISSs and the corresponding pseudo code. An interpretive ISS has the ability to behave close to real hardware and provides cycle-accurate estimates of software execution, but it has the main disadvantage of low simulation speed due to the online interpreting of target instructions. The simulation speed of an interpretive ISS is typically in the range from a few hundred kiloinstructions per second (KIPS) to a few million instructions per second (MIPS) on today's PCs. Further, the complexity of such an ISS often leads to long development time. PPC750Sim [76] is an example of interpretive ISSs and is available online.



(a) ISS Workflow

```
while(run){
  instruction = fetch(PC);
  opcode = decode(instruction);
  switch(opcode){
    case ADDI: execute_addi(); break;
    case: ...
  }
}
```

(b) Pseudo Code

Figure 3.1: ISS Workflow and Pseudo Code

The simulation of a multiprocessor system needs multiple such interpretive ISSs running simultaneously. As the bus system and hardware accelerators are often modeled in a system level design language like SystemC [55], designers have to handle complex timing synchronization between the processor simulators and the SystemC simulation kernel, as discussed in [44]. Such a complex and long-running simulation is unaffordable in the high-level design space exploration phase.

Today, there are also some fast cycle-accurate ISSs commercially available, for example, the simulators from the company VaST [16]. The VaST simulators use sophisticated binary translation techniques to convert target code directly into host code at run-time and can reach speeds from tens of MIPS up to a few hundreds of MIPS for a single processor. However, they are too time-consuming to develop.

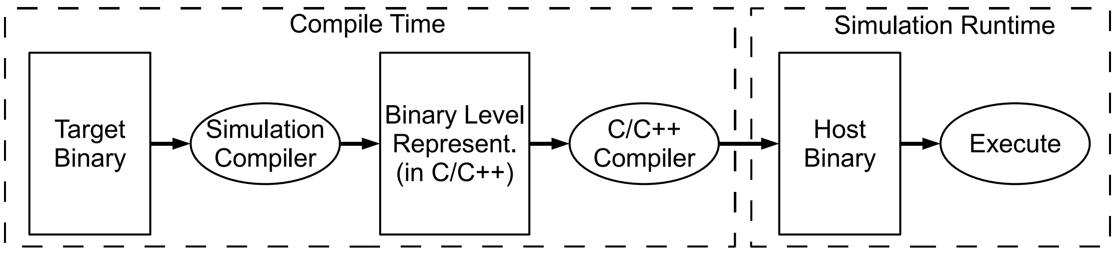
According to a presentation given by VaST, it takes 2-6 months for them to develop a new processor model. It is also inconvenient for system designers that the processor models cannot be modified or customized.

3.2 Binary (Assembly) Level Simulation

In a binary level simulation approach, the target binary of a program is translated to either host instructions or a high level language such as C or C++, with the same functionality as the original program. A typical BLS approach is shown in Figure 3.2(a). Compared with the ISS workflow in Figure 3.1(a), the BLS approach performs the time-consuming instruction fetching and decoding prior to simulation, i.e., at compile time, by means of code translation. The advantage of the compile-time fetching and decoding is that they are performed only once for each instruction. In contrast, at the run-time of an ISS, the instruction fetching and decoding are often repeated many times for most instructions, either when they are in a loop or when the task containing the instructions is activated many times in an application. In the BLS workflow, the translation is performed by a so-called *simulation compiler*. Therefore, a binary level simulator is often called *compiled ISS* in many other papers. Nevertheless, the term ISS only indicates interpretive ISSs in this dissertation. Mills et al. [75] were the first to propose this technique. Figure 3.2 also presents an example of translating a basic block of PowerPC instructions (Figure 3.2(c)), generated from the source code shown in Figure 3.2(b), to the binary level representation in C (Figure 3.2(d)).

For simulation, the C/C++ code generated by the simulation compiler is compiled by a host C/C++ compiler. The generated host executable is then executed to simulate the target execution. To estimate the temporal behavior of the target processor, timing information needs to be inserted into the functional representation. Since both the functional representation and the timing information are obtained from the binary level, their coupling is straightforward. Thus, the simulation accuracy depends solely on timing analysis. In [26], Bammi et al. used inaccurate statistical instruction timings to instrument binary level code, and thus, they got inaccurate estimation results. Nevertheless, if timing information that captures the important timing effects is used, BLS is able to achieve a simulation accuracy close to that of ISSs. The experiments done by Zivojnovic et al. [105] have confirmed this.

BLS offers simulation at a speed up to a few hundred MIPS on today's PCs. There is a slowdown of more than one order of magnitude compared with native execution, because the C/C++ code used to describe target instructions is much longer than the original source code. In the example in Figure 3.2, 13 lines of C code are used to describe the instructions generated from 4 lines of C code. Note that MEM_READ_BYTE and MEM_WRITE_BYTE in Figure 3.2(d) are both function calls that access a memory model that maps the target address space to



(a) BLS Workflow

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

(b) Source Code

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

(c) Target Binary

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

(d) Binary Level Representation in C

Figure 3.2: Binary Level Simulation Workflow and An Example

the memory of the host machine. Such accesses to the memory model are very time-consuming and are the main cause of the slowdown.

Compared with the ISS technique, BLS also has some disadvantages. The gain of speedup is achieved at the expense of flexibility. BLS assumes that the code does not change during run-time. Therefore, a hybrid interpretive/compiled scheme is needed to simulate self-modifying programs, such as the *just-in-time cache-compiled simulation* (JIT-CCS) technique reported in [32]. Further, unlike direct branches, the branch targets of which are statically known, the branch targets of indirect branches can be resolved only at run-time, so control flows of indirect branches are hard to be constructed statically in a BLS approach. Zivojnovic et al. [105] and Lazarescu et al. [68] treat every instruction as a possible target of an indirect branch and thus set a label before the translated code of each instruction. This solution makes the translated code less compiler-friendly and reduces the simulation performance. Nakada et al. [77] propose to give control to an ISS when an indirect jump is met. Neither solution is simple and efficient enough.

3.3 Source Level Simulation

Concerning only the functional modeling correctness, a software program can be simulated very simply by executing it directly on the host machine. However, such a native execution cannot provide any information of software performance on the target processor. The idea behind source level simulation is to annotate timing information into the source code for a native execution that can generate performance information of the target processor. The timing information is usually obtained by means of analysis at the binary level. The back-annotation of the timing information into the source code is regarded as the process of coupling the functional representation with the performance model. It relies on the mapping information that describes the correspondence between the source code and the binary code. More details about important issues in a SLS workflow are presented in the introduction to our SciSim approach in Chapter 4 and also in our paper [100].

The major difference among previous works on SLS is the way of timing analysis. Bammi et al. [26] calculate statistical instruction timings for each target processor using a set of benchmarks and store them in a so-called *processor basis file* for performance estimation of other programs. Because the delay caused by a single instruction is very context-related, the way to decide it statistically without taking concrete execution contexts into account is very inaccurate. A maximal error of 80.5% is reported in their paper [26]. Giusto et al. [46] improve the accuracy by generating one processor basis file for each processor in each application domain, based on the idea that the timing of each instruction in programs from the same application domain should be more constant. Still, large estimation errors are reported in their paper, although improvements are seen compared with Bammi's solution.

More accurate timing information can be obtained by means of static analysis [35] or measurement using cycle-accurate simulators [74], because the concrete context of each instruction is taken into account during the analysis or simulation. Usually, this kind of analysis and measurement is done for each basic block of a program. In the scope of a basic block, it lacks the execution context of the starting instructions, and global timing effects, such as the cache effect and the branch prediction effect, cannot be resolved. To solve this problem, in [100] we propose a way to perform dynamic analysis of the cache effect and the branch prediction effect at simulation run-time. The same solution is also proposed in [90]. In [100], we also introduce a method to do static analysis of superscalar pipelines to improve simulation accuracy when the target processor is a high performance processor with superscalar architecture.

Although these recent works on SLS have made improvements in performance modeling, there is still a big problem unsolved: the back-annotation of timing information into source code relies on the mapping between binary code and source code, and optimizing compilation makes it hard to find the mapping. Even when the mapping is found, the timing information back-annotated according to the mapping cannot be correctly aggregated along the source level control flows, due to the difference between source level control flows and binary level control flows. This problem will lead to a low accuracy in simulating compiler-optimized software. The previous works have not addressed this problem and estimated only unoptimized code. However, in reality, programs are usually compiled with a high optimization level. Therefore, this problem strongly limits the usability of the SLS technique.

3.4 IR Level Simulation

An IR level representation has accounted for processor-independent optimizations and allows for simulation as fast as SLS. Yet, the previous works on IR level simulation (IRLS) have not found an efficient way to describe the mapping between IR and binary code. Without the mapping information, timing information must be estimated also at the IR level for accurate back-annotation.

Kempf et al. propose an IRLS approach in [56]. In Figure 3.3 we take an example from [56] to explain this approach. Figure 3.3(a) shows an example of *three address code intermediate representation (3AC IR)*, where all C operators and the majority of memory accesses are visible. In order to get timing information, each IR operation is associated with a timing value according to the processor's description. For example, in Figure 3.3(b) the operations "+" and "∗" are associated with "cost of ADD" and "cost of MUL", respectively, which are read from the processor's description file by calling GetOPCost(). Coupling of the IR code and the timing information is straightforward by inserting after each IR operation its associated timing value. However, these timing values have not taken into account the timing effects of the compiler back-end and the target processor, and thus, are very inaccurate.

```
a = 1;
a = a * 4;
tmp = (Start Address of x) + a;
b = LOAD(tmp)
```

(a) 3AC IR

```
a = 1;
cycle += GetOPCost(ADD);
a = a * 4;
cycle += GetOPCost(MUL);
tmp = (Start Address of x) + a;
cycle += GetOPCost(ADD);
b = LOAD(tmp);//memory simulation
```

(b) Timed IR

Figure 3.3: An Example of IR Level Simulation

Cheung et al. [34] also associate each IR operation with a latency. Then, to take the processor-dependent optimizations into account, they use a SystemC back-end that emulates the back-end of the cross-compiler and optimizes both IR operations and their associated latencies. The output of the SystemC back-end is a simulation model in SystemC, annotated with timing information that has accounted for all the compiler optimizations. A similar approach is proposed by Lee et al. [69]. They modify the back-end of the host compiler to add a so-called *counter field* for each IR operation to record its cycle count. This field is combined with the IR operation and will be deleted, moved or duplicated when the IR operation is deleted, moved or duplicated during compiler optimizations. Thus, after compilation, the annotated timing information is also "optimized". The output simulation model is a host binary program. However, the both approaches still cannot take the timing effects of the target processor into account. Furthermore, they need either modify a compiler or write a new compiler back-end. This work requires a good knowledge of the compiler architecture and needs a large implementation effort. Furthermore, neither the modified host compiler back-end nor the SystemC back-end behave exactly the same as the back-end of the target cross-compiler. This is also a cause of estimation error.

The iSciSim approach [95] proposed in this dissertation also generates simulation models at the IR level. Unlike the previous IRLS approaches discussed above, which use a "top-down" approach to trace the effect of compiler optimizations on instruction timing, we use a "bottom-up" approach to annotate timing information, which is obtained from the binary level and already takes the effects of compiler optimizations and processor microarchitecture into account, back into the IR level code. To get the mapping between IR and binary code, we make IR compilable by formalizing it in the C language. We call this IR-level C code *intermediate source code* (ISC). Then, the debugging information describing the mapping can be generated during the compilation of the ISC. With the mapping information, the ISC can be annotated with the accurate timing information. Hence, our approach allows for very accurate simulation. The whole approach is also very easy to realize and does not need compiler modification.

Chapter 4

SciSim: A Source Level Approach for Software Performance Simulation

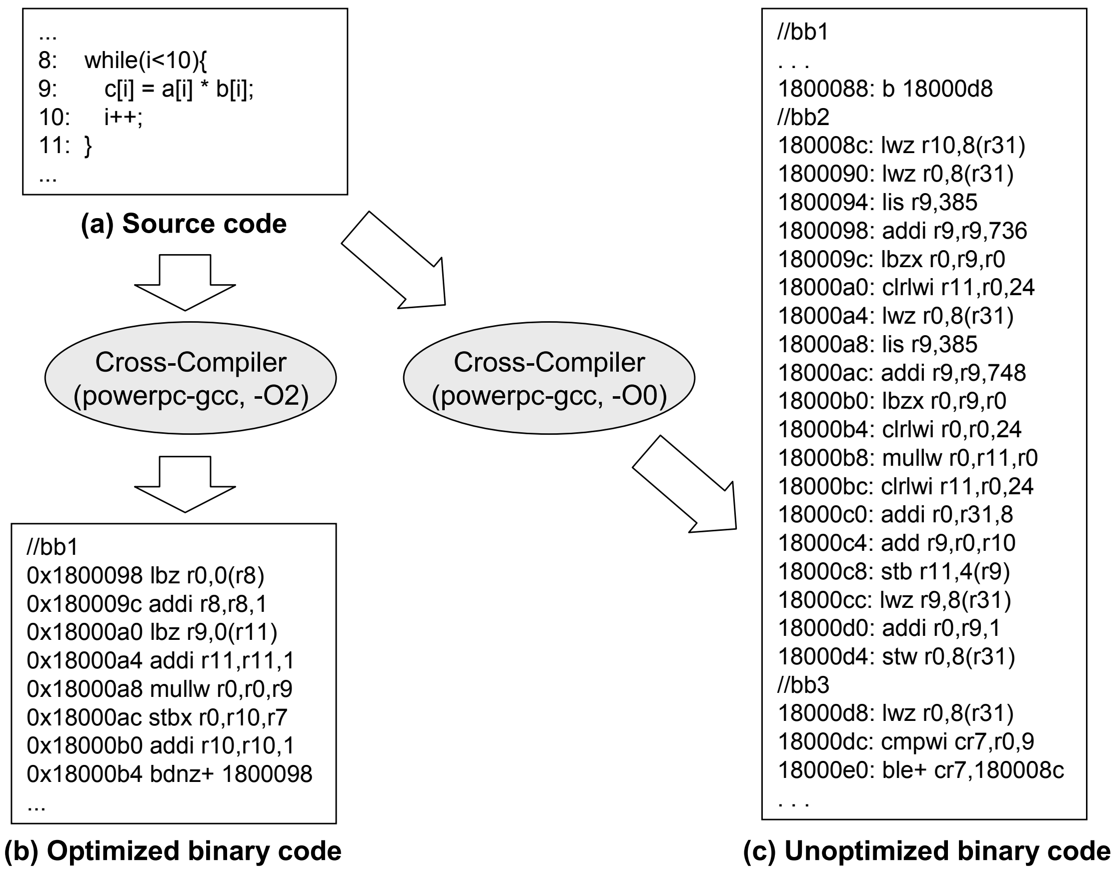
As discussed previously, for software performance simulation, cycle-accurate instruction set simulators (ISSs) are too slow and also too time-consuming to develop. Furthermore, trace-driven simulators are not able to capture the control flows of software execution. Only native-execution based simulation techniques are suitable for generating high level simulation models that capture both the function and timing of software tasks. Among several native-execution based simulation techniques, source level simulation (SLS) has gained great popularity, because of its high simulation performance and low complexity. A source level simulation model uses source code as functional representation. Timing information that represents execution delays of the software on the target processor is inserted into the source code according to the mapping between the source code and the generated machine code. That is, a source level simulation model combines the low-level timing information and a high-level functional model.

This chapter provides some basic information about software compilation, optimization, and control flow graphs, gives an introduction to source code instrumentation, and presents a new SLS approach, SciSim. Then, we present some experimental results to show the advantages and limitations of the SciSim approach. The major limitation of SciSim is that it cannot simulate some compiler-optimized software accurately. So, we discuss about the reasons for this using some examples. Finally, the advantages and limitations are summarized.

4.1 Basic Information

4.1.1 Compilation and Optimization

The programmability of a microprocessor is thanks to the presence of an abstraction layer, called *instruction set architecture (ISA)*. The ISA abstracts the underlying processor microarchitecture and provides a common instruction set to


(a) Source code
Cross-Compiler (powerpc-gcc, -O2)
Cross-Compiler (powerpc-gcc, -O0)
(b) Optimized binary code
(c) Unoptimized binary code

```
...
8:   while(i<10){
9:      c[i] = a[i] * b[i];
10:     i++;
11:  }
...
```

```
//bb1
0x1800098 lbz r0,0(r8)
0x180009c addi r8,r8,1
0x18000a0 lbz r9,0(r11)
0x18000a4 addi r11,r11,1
0x18000a8 mullw r0,r0,r9
0x18000ac stbx r0,r10,r7
0x18000b0 addi r10,r10,1
0x18000b4 bdnz+ 1800098
...
```

```
//bb1
. . .
1800088: b 18000d8
//bb2
180008c: lwz r10,8(r31)
1800090: lwz r0,8(r31)
1800094: lis r9,385
1800098: addi r9,r9,736
180009c: lbzx r0,r9,r0
18000a0: clrlwi r11,r0,24
18000a4: lwz r0,8(r31)
18000a8: lis r9,385
18000ac: addi r9,r9,748
18000b0: lbzx r0,r9,r0
18000b4: clrlwi r0,r0,24
18000b8: mullw r0,r11,r0
18000bc: clrlwi r11,r0,24
18000c0: addi r0,r31,8
18000c4: add r9,r0,r10
18000c8: stb r11,4(r9)
18000cc: lwz r9,8(r31)
18000d0: addi r0,r9,1
18000d4: stw r0,8(r31)
//bb3
18000d8: lwz r0,8(r31)
18000dc: cmpwi cr7,r0,9
18000e0: ble+ cr7,180008c
. . .
```

Figure 4.1: Embedded Software Cross-Compilation

different programs. An instruction set is a list of all the instructions implemented by the microarchitecture of a processor. A typical instruction set includes four categories of instructions: arithmetic instructions, logic instructions, data instructions, control flow instructions. Processors with different microarchitectures can share an instruction set. In the embedded systems domain, ARM, PowerPC, MIPS and SPARC etc. are widespread ISAs.

Today, embedded software is developed basically the same as PC software development, using tools like compilers, assemblers, and debuggers etc. It is usually programmed in a high-level programming language, mostly in the C language. One difference is that, for PC software development we both develop and execute software programs on PC, while for embedded software development the programs developed on PC have to be ported onto a target embedded system. Thus, the programs have to be compiled to executables that contain only instructions of the target processor. This process of generating executables of other processors by running compilers on a PC is called *cross-compilation*. The compilers used are called *cross-compilers*. The cross-compilers used in our work are obtained by porting GCC (GNU Compiler Collection) compilers to target processors. GCC compilers can be found online and downloaded for free and support the ISAs in common use, such as PowerPC, ARM, SPARC and MIPS etc. Figure 4.1 shows an example of cross-compiling a program for a PowerPC processor. Each PowerPC instruction is shown as a single line of assembly at a given program address: *[address: assembly]*. The cross-compiler, *powerpc-gcc*, is a GCC compiler ported to the PowerPC processor.

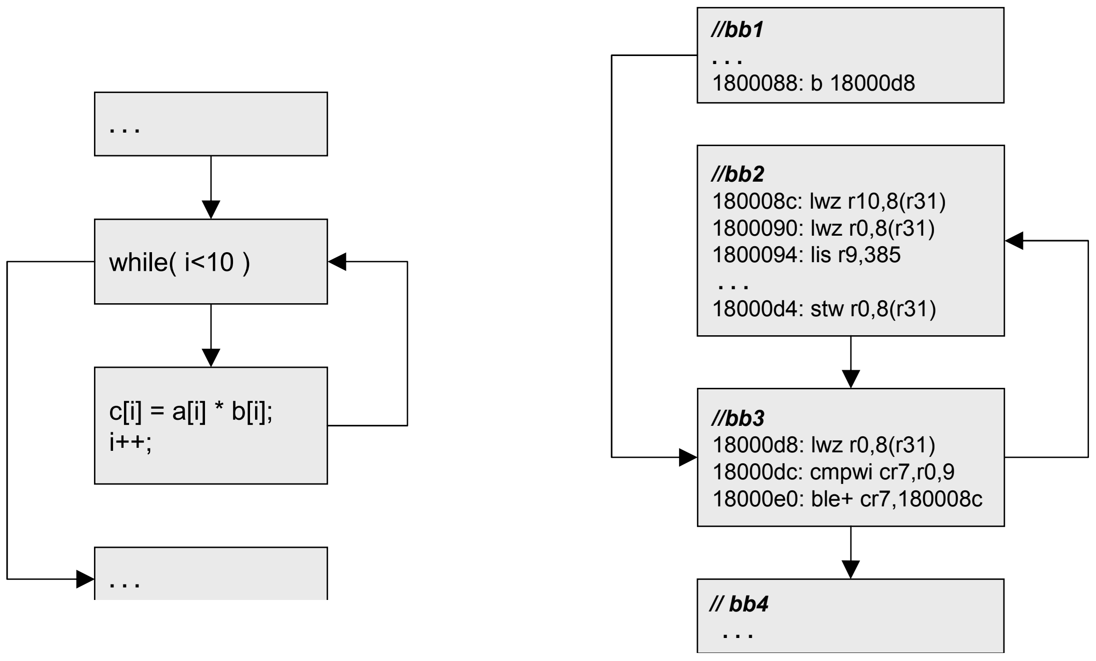
During the compilation, optimizations can be performed to minimize or maximize some attributes of the executable of a program, according to a given optimization level. For example, we can set the compiler to minimize the time taken to execute a program. We can also minimize the program size to save memory. During compiler optimizations, complex statements in a program are first transformed to single statements, which are then optimized by operations, such as dead code elimination, code motion and instruction scheduling etc. to minimize execution time or code size. The optimizations can be categorized into target machine independent and dependent optimizations. Machine independent optimizations operate on abstract programming concepts such as loops and structures, while machine dependent optimizations exploit features of the target processor. Most widely used compilers have two decoupled parts, namely the front-end and the back-end, for performing target machine independent and dependent manipulations, respectively.

GCC compilers provide a range of general optimization levels. An optimization level is chosen with the command line option -O*LEVEL*, where *LEVEL* is a number from 0 to 3. Compilation with a higher optimization level can achieve more reduction in execution time or size of an executable, but will increase in cost including compilation time and memory requirement. For example, with the option -O2, the compiler will take longer to compile programs and require more memory than with -O1. For most cases it is satisfactory to use -O0 for debugging and -O2 for development and deployment. With -O0, the compiler does not perform any optimization and converts the source code directly to the corresponding instructions without any rearrangement, while with -O2, most common optimizations are turned on and the compiler provides the maximum optimization without increasing the executable size. For a detailed description of GCC compilers' optimization levels, please refer to the book "An Introduction to GCC" [47].

Figure 4.1(b) and Figure 4.1(c) show (disassembled) binary codes generated from the C code in Figure 4.1(a) with optimization levels -O2 and -O0, respectively. Without any optimizations, the four lines of source code are converted to 23 PowerPC instructions. The number of instructions is reduced to 8, when the optimization level -O2 is used.

4.1.2 Control Flow Graph

A *control flow graph (CFG)* is basically a graphical representation of a program. It is essential to many compiler optimizations and static analysis tools. We define some basic terms to facilitate the later discussion regarding CFG.



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

```
//bb1
. . .
1800088: b 18000d8

//bb2
180008c: lwz r10,8(r31)
1800090: lwz r0,8(r31)
1800094: lis r9,385
 . . .
18000d4: stw r0,8(r31)

//bb3
18000d8: lwz r0,8(r31)
18000dc: cmpwi cr7,r0,9
18000e0: ble+ cr7,180008c

// bb4
 . . .
```

(a) Source Level Control Flow Graph **(b) Binary Level Control Flow Graph**

Figure 4.2: Control Flow Graphs

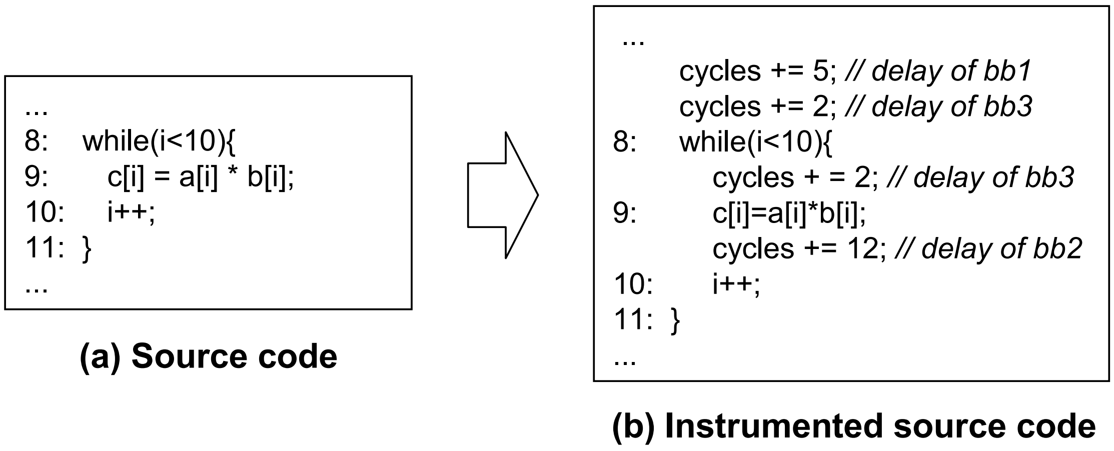
- Basic block: a *basic block* corresponds to a node in the CFG. It consists of a sequence of instructions that has only one entry point and one exit point. In the rest of the thesis, we use sometimes *block* instead of basic block.
- Entry block: an *entry block* is a special basic block, through which all control flow enters the graph.
- Exit block: an *exit block* is a special basic block, through which all control flow leaves the graph.
- Edge: an *edge* connects two basic blocks and represents a jump in a control flow. It is directed.

In the different phases of compilation, a program is represented differently as source code, intermediate representation (IR), or binary code. Each level of representation can have a CFG. Hence, a program has a source level CFG, an IR-level CFG, and a binary level CFG.

Both the source level CFG and the binary level CFG of the example in Figure 4.1 are shown in Figure 4.2. In the source level CFG, the *while* loop is splitted into two *source level basic blocks*. The loop test "*while*$(i < 10)$" is a basic block. The loop body is another basic block. The control flows in a source level CFG are expressed by semantics of high level programming constructs. In the example, the control flow begins with a jump from the code before the *while* loop to the basic block containing the loop test. According to the loop test result, the control flow jumps either to the block containing the loop body or to the block after the *while* loop. In contrast, at the binary level, control flows are expressed explicitly by control flow instructions. In Figure 4.2(b), the instructions *b* and *ble+* realize an unconditional jump and a conditional jump, respectively.

4.1.3 Introduction to Source Code Instrumentation

In the context of computer programming, instrumentation means to add some code in application code to monitor specific components in a system, in order to diagnose errors, measure performance, or write trace information. So, *source code instrumentation* means to add some code to application source code. Recently, this technique is becoming popular for system-level software performance simulation. In this context, source code instrumentation is a process of generating software simulation models by annotating application source code with some code that expresses timing information. Instrumentation is done before simulation, i.e., at compile-time. The code that expresses timing information is called *annotation code* in the following discussion.



```
...
8:   while(i<10){
9:      c[i] = a[i] * b[i];
10:     i++;
11:  }
...
```

(a) Source code

```
...
        cycles += 5; // delay of bb1
        cycles += 2; // delay of bb3
8:      while(i<10){
            cycles + = 2; // delay of bb3
9:          c[i]=a[i]*b[i];
            cycles += 12; // delay of bb2
10:         i++;
11:  }
...
```

(b) Instrumented source code

Figure 4.3: An Example of Source Code Instrumentation

Figure 4.3 presents an example of source code instrumentation. Figure 4.3(b) shows the same functional code as in Figure 4.3(a) with delay annotations. The annotated delay values are added to a counter variable *cycles* and will be aggregated along the control flows during the simulation. For example, when the instrumented source code is executed once, the delay values annotated outside the loop will be aggregated once and the delay values annotated inside the loop will be aggregated as often as the number of the loop iterations, 10 times in the example. The instrumentation follows specific rules. For example, the delay value of *bb3* must be annotated both before and after the *while* statement. We will discuss about such instrumentation rules in the introduction to our own source code instrumentation approach in the next section.

4.2 The SciSim Approach

To implement a SLS approach, there are two important issues: (1) the way of getting the delay values, and (2) the way of inserting the delay values into source code. To get accurate delay values, we need an accurate low-level performance model, with which the timing effects of the target processor can be taken into account. Inserting the obtained delay values back into source code relies on the mapping between binary code and source code. We developed a new SLS approach, called SciSim. Compared to other existing SLS approaches, SciSim allows for more accurate performance simulation by employing a hybrid method for timing analysis. That is, timing analysis is done on the target binary code, both statically for pipeline effects and dynamically for other timing effects that cannot be resolved statically. We also use a simple way to get information on the mapping between source code and binary code, namely, extract the mapping information from debugging information, which can be dumped automatically during cross-compilation. Thus, the whole approach can be fully automated without the need of any human interaction.

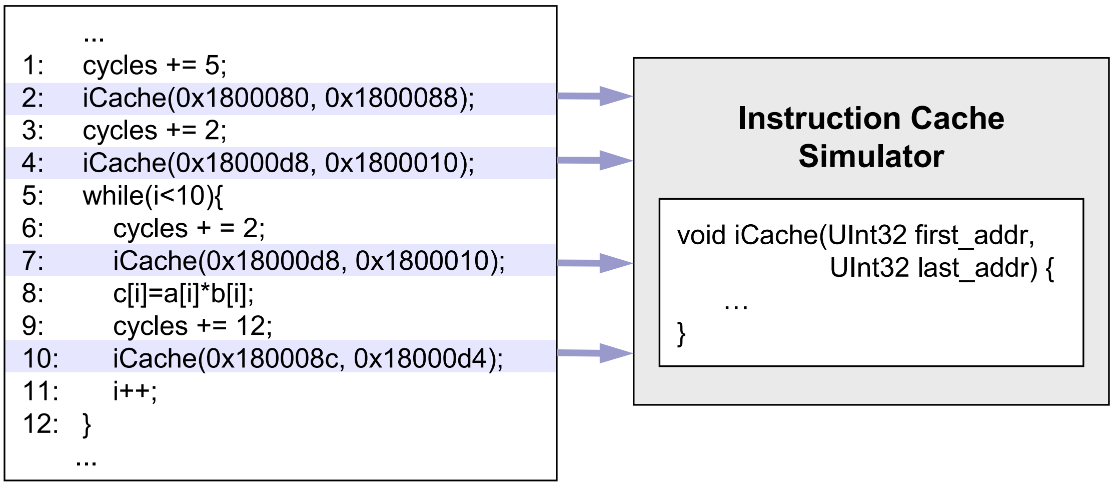
However, the existing SLS approaches, also including the proposed SciSim approach, have a major disadvantage in that they cannot estimate some compiler-optimized programs accurately, because the back-annotation of timing information into source code relies on the mapping between binary code and source code, and in many cases, optimizing compilation makes it hard to find the mapping. Sometimes, even when the mapping is found, the timing information back-annotated according to the mapping cannot be correctly aggregated along the source level control flows, due to the difference between source level control flows and binary level control flows. This problem is especially serious for control-dominated programs. We will describe this problem in more detail with some examples in Section 4.4.

In this section, we show SciSim only for performance simulation of unoptimized software. The iSciSim approach introduced in Chapter 5 extends SciSim and solves the problem raised by compiler optimizations. It reuses the whole instrumentation approach of SciSim. In this chapter, we only introduce the basic idea and concept of SciSim and do not go into the details of the instrumentation approach, which are however presented in Chapter 5 in the introduction of the iSciSim approach.

The work flow of SciSim consists of two steps, namely source code instrumentation and simulation, which are presented in the following two subsections.

4.2.1 Source Code Instrumentation

The source code instrumentation tool is responsible for specifying what code is to be annotated into the source code and where the code is to be annotated. The



```
...
cycles += 5;
iCache(0x1800080, 0x1800088);
cycles += 2;
iCache(0x18000d8, 0x1800010);
while(i<10){
    cycles + = 2;
    iCache(0x18000d8, 0x1800010);
    c[i]=a[i]*b[i];
    cycles += 12;
    iCache(0x180008c, 0x18000d4);
    i++;
}
...
```

```
void iCache(UInt32 first_addr,
                    UInt32 last_addr) {
      …
}
```

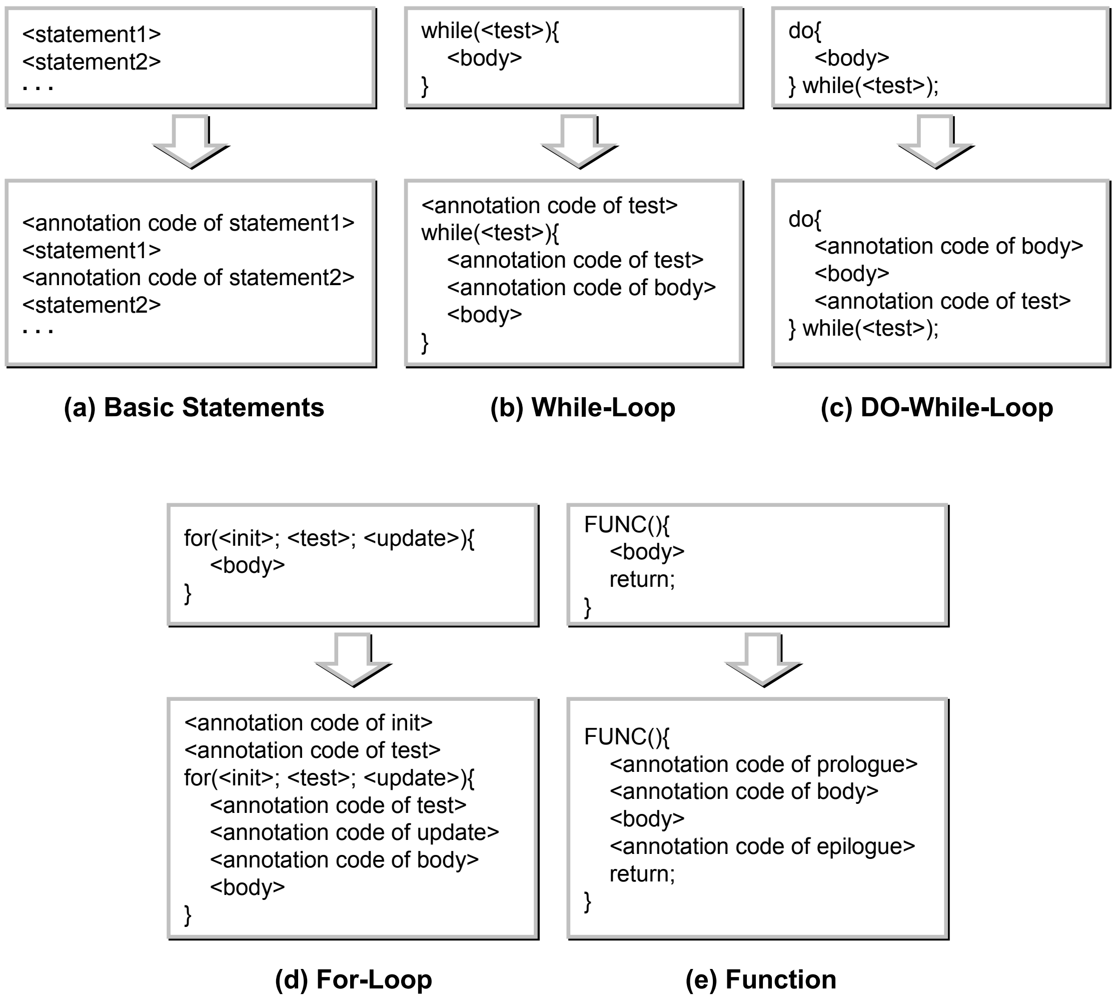
Figure 4.4: Instrumentation for Instruction Cache Simulation

two issues are discussed in the following. The details about the working steps of the instrumentation tool are introduced in Section 5.3.

What code is to be annotated?

There are two categories of annotation code according to the concept of our hybrid timing analysis method: statically obtained delay values and code for triggering dynamic timing analysis. In the instrumentation step, some timing effects are represented by delay values obtained by static analysis. These delay values can advance the simulation time directly while the simulator runs. Thus, time-consuming, iterative analysis of these timing effects can be avoided at simulation run-time. To take into account other timing effects that cannot be resolved statically but have a large impact on software performance, annotation code also contains code that can send run-time data to the corresponding performance model to trigger the dynamic analysis of these timing effects. Correspondingly, the performance model is divided into two parts: an offline performance model and an online performance model, for static timing analysis and dynamic timing analysis, respectively.

Figure 4.4 shows an example of instrumented source code, which, when runs, can take the instruction cache effect into account. The annotation code for each basic block contains a delay value, which is obtained by pipeline analysis, and a function call *icache(UInt32 first_addr, UInt32 last_addr)*, which will trigger dynamic instruction cache simulation at simulation run-time. During the simulation, the function call passes the addresses of the first instruction and the last instruction of the basic block to an instruction cache simulator, which then decides how many cache misses are caused by this basic block according to the present state of the cache. To simulate other timing effects, other code has to be annotated into the



```
<statement1>
<statement2>
. . .
```

```
<annotation code of statement1>
<statement1>
<annotation code of statement2>
<statement2>
. . .
```

(a) Basic Statements

```
while(<test>){
    <body>
}
```

```
<annotation code of test>
while(<test>){
    <annotation code of test>
    <annotation code of body>
    <body>
}
```

(b) While-Loop

```
do{
    <body>
} while(<test>);
```

```
do{
    <annotation code of body>
    <body>
    <annotation code of test>
} while(<test>);
```

(c) DO-While-Loop

```
for(<init>; <test>; <update>){
    <body>
}
```

```
<annotation code of init>
<annotation code of test>
for(<init>; <test>; <update>){
    <annotation code of test>
    <annotation code of update>
    <annotation code of body>
    <body>
}
```

(d) For-Loop

```
FUNC(){
    <body>
    return;
}
```

```
FUNC(){
    <annotation code of prologue>
    <annotation code of body>
    <body>
    <annotation code of epilogue>
    return;
}
```

(e) Function

Figure 4.5: Part of Instrumentation Rules

source code. More details about dynamic timing simulation are presented in Section 5.4, while static timing analysis for accurate delay values is introduced in Section 5.3.3.

Where is the code to be annotated?

To automate the instrumentation approach, an easy way of describing the mapping between source code and binary code is important. We generate the mapping information from debugging information, which can be dumped automatically from the object file of a program.

Using the mapping information and according to a set of instrumentation rules, the annotation code is placed into the correct position in the source code. Part of instrumentation rules used in SciSim is shown in Figure 4.5. Basically, the annotation code of each basic block is inserted before the corresponding source line, as illustrated in Figure 4.5(a). For some complex C statements, such as loop constructs, annotation code must be placed specially. Hence, a special instrumentation rule must be defined for each of such complex constructs. In all the loop constructs, *for* loop is most complex. A *for* statement typically contains three parts: "$< init >$" that initializes the loop count, "$< test >$" that tests the loop condition, and "$< update >$" that updates the loop count. The instrumentation rule of the *for* loop is shown in Figure 4.5(d). The annotation code generated from "$< init >$" is placed before the *for* statement, while the annotation code generated from "$< update >$" is placed after the *for* statement, i.e., in the loop body. The annotation code generated from "$< test >$" is annotated both before and after the *for* statement. With this placement the annotation code of "$< test >$" will be executed as often as "$< test >$". The annotation code generated from the loop body is placed according to the code constructs of the loop body.

Another example of special instrumentation rules is the one defined for placing annotation code of a function's prologue and epilogue. As shown in Figure 4.5(e), the annotation code of the function prologue is inserted after the first left curly bracket "{", i.e., in the first line of the function body. The annotation code of the function epilogue must be inserted before each *return* statement. If there is no *return* statement in a function, the annotation code is inserted before the function's last right curly bracket "}".

4.2.2 Simulation

For performance profiling of a single software task, the instrumented source code is compiled together with the online performance model. We just need to click to run the generated host executable. All the annotated timing information will then be aggregated to generate performance statistics of the target execution of the task. To simulate a multiprocessor system, a system level design language that supports modeling of the system's concurrent behavior is required. Among system level design languages (SLDL), SystemC is the most frequently used one in both academia and industry and has become a standard for system level design space exploration [55]. Therefore, for system level simulation, instrumented software tasks are generated as Transaction Level Models (TLMs) in SystemC. The software TLMs are then combined with TLMs of the other system components to get a TLM of the whole system.

Simulation is discussed in more detail in Chapter 5 and 6.

4.3 Experimental Results

The experiments have been carried out to show the advantages and limitations of SciSim, using a set of benchmarks. The benchmarks consisted of 6 programs with different workloads. They were *fibcall*, *insertsort*, *bsearch*, *crc*, *blowfish*, and *AES*. PowerPC 603e, a superscalar design of the PowerPC architecture, was chosen as the target processor.

In the first experiment, we show the high speed and accuracy of SciSim for performance simulation of the programs that were compiled without optimizations. Since the cross-compiler was obtained by porting a GCC compiler, the optimization level -O0 was set to compile the programs in the first experiment. In the second experiment, we show the limitation of SciSim that it cannot estimate some compiler-optimized software accurately. Hence, all the programs were compiled with the optimization level -O2.

We evaluated the SciSim approach quantitatively, in terms of simulation accuracy and speed. Simulation accuracy was measured by both instruction count accuracy and cycle count accuracy. Instruction count accuracy depends solely on the mapping between functional representation and binary code, while cycle count accuracy is dependent on both the mapping and the timing analysis.

To get a reference, we developed a regular interpretive ISS to simulate this processor. The estimates obtained by the ISS have taken the most important timing effects into account using a simplified performance model, which contains a pipeline model, a data cache simulator, and an instruction cache simulator. As there are still some approximations made in the performance model, the estimates from the ISS are not exactly the same as but close to the real execution times. We assume that the estimates from the ISS were accurate enough and could be used as a reference to evaluate the simulation accuracies of the other tools. The estimation error (EE) of SciSim and the mean absolute error (MAE) of simulating N programs are given by:

$$EE = \frac{(Estimate_{SciSim} - Estimate_{ISS})}{Estimate_{ISS}} \times 100\%$$

and

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|EE_i|$$

The simulation speeds of ISS and SciSim are calculated by

$$simulation\ speed = \frac{C_{instruction}}{(T_{simulation} \times 1000000)}$$

where $C_{instruction}$ is the total number of instructions executed when running the program on the target processor, estimated by simulation, and $T_{simulation}$ is the simulation duration. In the experiment, the simulation duration was measured on a 3GHz Intel CPU based PC with 4 GB memory, running Linux.

The results of the two experiments are shown in the following two subsections:

4.3.1 Performance Simulation of Unoptimized Software

The results of this experiment are shown in Table 4.1. SciSim is able to count instructions as accurate as the ISS for the unoptimized programs, as shown in the table. Except *AES*, the instruction counts of the other five programs estimated by SciSim are exactly the same as those estimated by the ISS. A trivial difference is found in the numbers of instructions of *AES* counted by SciSim and the ISS. This is because that SciSim had inaccurate instrumentation for the *switch* statements contained in *AES*.

The cycle counts estimated by SciSim has an MAE of 2.02%. The reason for this slight error is because more approximations were made in the performance model of SciSim. In the estimates from SciSim, timing effects such as the cache effect and the branch prediction effect were not taken into account.

Table 4.1: Performance Simulation of Unoptimized Software

		ISS	**SciSim**	**Error**
fibcall	instructions	16311	16311	0.00%
	cycles	16295	16130	-1.01%
insertsort	instructions	2330	2330	0.00%
	cycles	2338	2310	-1.20%
bsearch	instructions	152028	152028	0.00%
	cycles	154023	152021	-1.30%
crc	instructions	54068	54068	0.00%
	cycles	49142	47882	-2.56%
blowfish	instructions	913167	913167	0.00%
	cycles	973074	945121	-2.87%
AES	instructions	7345439828	7346851268	0.02%
	cycles	6645440614	6568823424	-1.15%

Table 4.2 shows both the native execution speed and the simulation speeds, measured in MIPS (million instructions per second). The simulation speed of the ISS is very low, in the range from 0.93 MIPS to 27.78 MIPS. It is found that the ISS has very different simulation performance in simulating programs of different sizes. It is because that the ISS has to do some initialization every time before simulation can start. The time for initialization accounts for a large portion of the simulation time of a small program and thus reduces the simulation performance. In contrast, it is ignorable in simulating a large program.

SciSim achieved very high performance of around 13500 MIPS on average, even 2.5 times as fast as the native execution (around 5400 MIPS on average). The reason for this is because a simulation model generated by SciSim contains timing information of unoptimized target binary and itself is then compiled by the host compiler with compiler optimizations. To explain this more clearly, we show the formulas used to calculate the native execution speed and the simulation speed of SciSim. The native execution speed is calculated by:

$$native\ execution\ speed \ = \ \frac{C_{Host\ Instruction}}{(T_{Native\ \ Execution} \ \times \ 1000000)},$$

while the simulation speed of SciSim is given by

$$simulation\ speed \ = \ \frac{C_{Target\ Instruction}}{(T_{Simulation} \ \times \ 1000000)}.$$

Because the target binary is unoptimized and the host binary of the simulation model is highly optimized, $C_{Target\ Instruction}$ is much larger than $C_{Host\ Instruction}$. On the other hand, because the back-annotated annotation code does not bring too much overhead, $T_{Simulation}$ usually has a comparable value to $T_{Native\ Execution}$. Therefore, the simulation speed of SciSim is much higher than the native execution speed for the simulation of unoptimized software. Compared with ISS, the simulation performance of SciSim is 1386 times higher.

Table 4.2: Simulation Speeds of Simulating Unoptimized Software (in MIPS)

	Native Execution	**ISS**	**SciSim**
fibcall	4634.04	2.1	14218.78
insertsort	4357.73	0.93	12057.40
bsearch	7881.79	7.34	10569.28
crc	4762.62	5.22	14008.82
blowfish	5291.19	13.04	13638.29
AES	5474.97	27.78	16506.75

4.3.2 Performance Simulation of Optimized Software

Because embedded programs are usually compiled with compiler optimizations in practice, the efficiency of simulating compiler-optimized programs is more important for a simulation technique. We measured the execution times of the 6 programs compiled with and without optimizations, using the ISS. The results are shown in Table 4.3. We can see, with the optimization level -O2, the software performance increases around four times on average.

In this sub-section we show the simulation accuracy and performance of SciSim for simulating compiler-optimized software. The results are shown in Table 4.4. It is found that SciSim is still able to count instructions accurately for data-intensive

Table 4.3: Software Execution Time Estimated by the ISS: With Optimizations vs. Without Optimizations

		-O0	**-O2**	**Difference**
fibcall	instructions	16311	4384	272.1%
	cycles	16295	3307	392.7%
insertsort	instructions	2330	630	269.8%
	cycles	2338	516	353.1%
bsearch	instructions	152028	59010	157.6%
	cycles	154023	46057	234.4%
crc	instructions	54068	17201	214.3%
	cycles	49142	10288	377.7%
blowfish	instructions	913167	262434	248.0%
	cycles	973074	178610	444.8%
AES	instructions	7345439828	3624426960	102.7%
	cycles	6645440614	2292275341	189.9%

Table 4.4: Performance Simulation of Optimized Software

		ISS	**SciSim**	**Error**
fibcall	instructions	4384	12973	195.92%
	cycles	3307	12996	292.98%
insertsort	instructions	630	975	54.76%
	cycles	516	672	30.23%
bsearch	instructions	59010	75013	27.12%
	cycles	46057	51008	10.75%
crc	instructions	17201	19385	12.70%
	cycles	10288	15430	49.98%
blowfish	instructions	262434	265735	1.26%
	cycles	178610	187184	4.80%
AES	instructions	3624426960	3624685359	0.01%
	cycles	2292275341	2336484895	1.93%

applications, like *blowfish* and *AES*, where the control flows are relatively simple. The average instruction count error in simulating the two programs is 0.63%. However, for the other programs, large errors are seen, because the control flows of these programs were changed by compiler optimizations and SciSim could not find a correct mapping between the binary code and the source code. The MAEs are 48.6% and 65.1% in instruction counting and cycle counting, respectively. Therefore, SciSim has a very limited ability to simulate compiler-optimized software.

The average simulation speed of SciSim was 4669.0 MIPS, close to the native execution speed, which was 5198.2 MIPS on average.

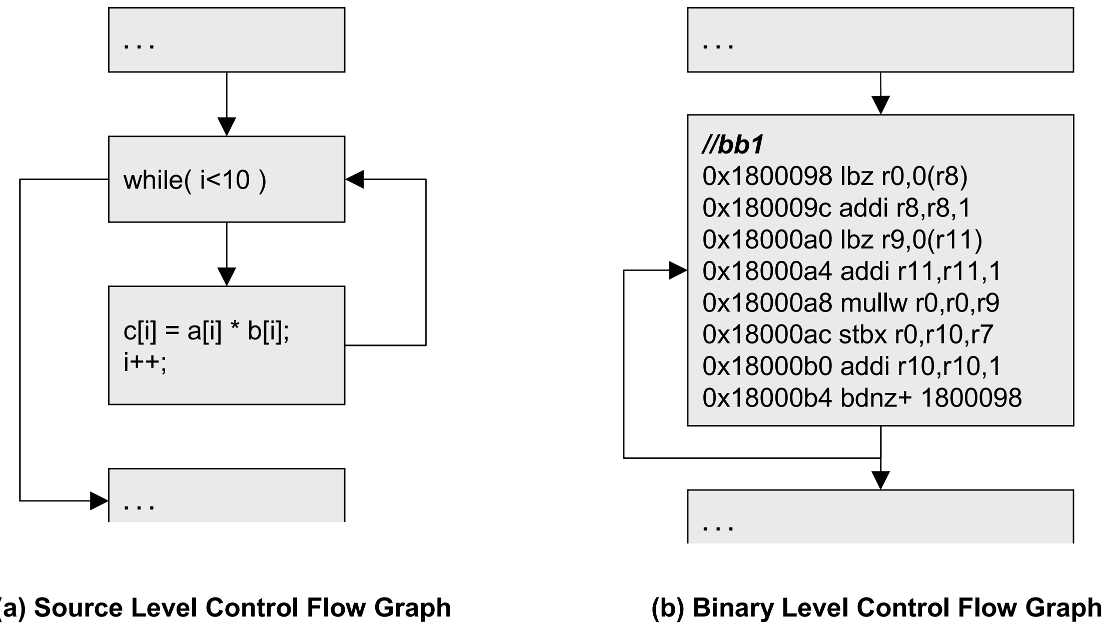
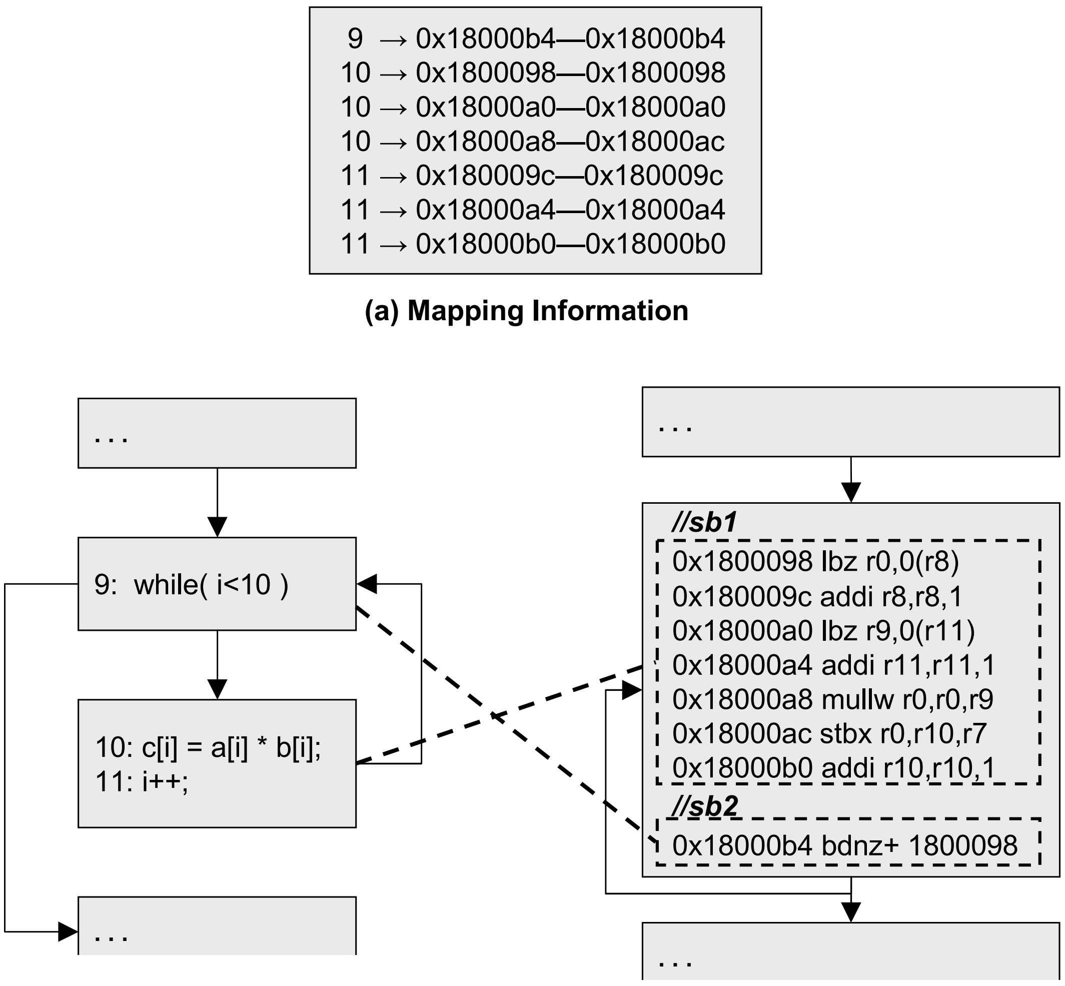


Figure 4.6: Source Level CFG and Binary Level CFG Constructed from Compiler-Optimized Binary Code

4.4 Why Does SciSim Not Work for Some Compiler-Optimized Software?

As already presented, when we compare a source level CFG and the corresponding binary level CFG constructed from unoptimized binary code, source level basic blocks can always find their counterparts at the binary level and an instrumentation rule can be made for each complex statement for accurate back-annotation of timing information. An example has been illustrated in Figure 4.2. However, when compiler optimizations are enabled during the compilation, complex statements in a program are transformed to simple statements, which are then optimized by the compiler to minimize execution time or code size. The same statement is manipulated differently in different contexts. Such manipulations invalidate the instrumentation rules defined for those complex statements.

As shown in Figure 4.6, compiled with optimizations, the *while* loop generates only one basic block of binary code. No mapping can be established between the source level basic blocks in Figure 4.6(a) and the binary level basic blocks in Figure 4.6(b).



```
9  → 0x18000b4—0x18000b4
10 → 0x1800098—0x1800098
10 → 0x18000a0—0x18000a0
10 → 0x18000a8—0x18000ac
11 → 0x180009c—0x180009c
11 → 0x18000a4—0x18000a4
11 → 0x18000b0—0x18000b0
```

(a) Mapping Information

```
//sb1
0x1800098 lbz r0,0(r8)
0x180009c addi r8,r8,1
0x18000a0 lbz r9,0(r11)
0x18000a4 addi r11,r11,1
0x18000a8 mullw r0,r0,r9
0x18000ac stbx r0,r10,r7
0x18000b0 addi r10,r10,1
//sb2
0x18000b4 bdnz+ 1800098
```

(b) Mapping Established between Source Code and Binary Code

Figure 4.7: Mapping between Source Code and Binary Code in the Sub-block Granularity

For this shown example, the mapping problem can be solved by establishing the mapping between source lines and sub-blocks of binary code instead of basic blocks. Here, a sub-block is either a basic block or a part of a basic block. Such a mapping can be established with the help of debugging information. Figure 4.7(a) shows the mapping information extracted from *DWARF's line table*, a kind of debugging information, which describes the correspondence between source lines and the generated machine code. Each entry of mapping information is expressed as "source line number → the first address of the sub-block – the last address of the sub-block". For example, the entry "10 → 0x18000a8–0x18000ac" denotes that the instructions stored in the memory block "0x18000a8–0x18000ac" are generated from source line 10. It is possible that the last address of the sub-block is the same as its first address. This means that the sub-block contains only one instruction.

We can find that, after optimizing compilation, the instructions generated by line 10 and line 11 are arranged in an alternate order. As line 10 and 11 are in the same source level basic block and the instructions generated from them are also in the same binary level basic block, these instructions are merged into a single sub-block *sb1*. In the same way, the sub-block *sb2* is mapped to the *while* statement at line 9. The mapping in the granularity of sub-blocks is depicted in Figure 4.7(b).

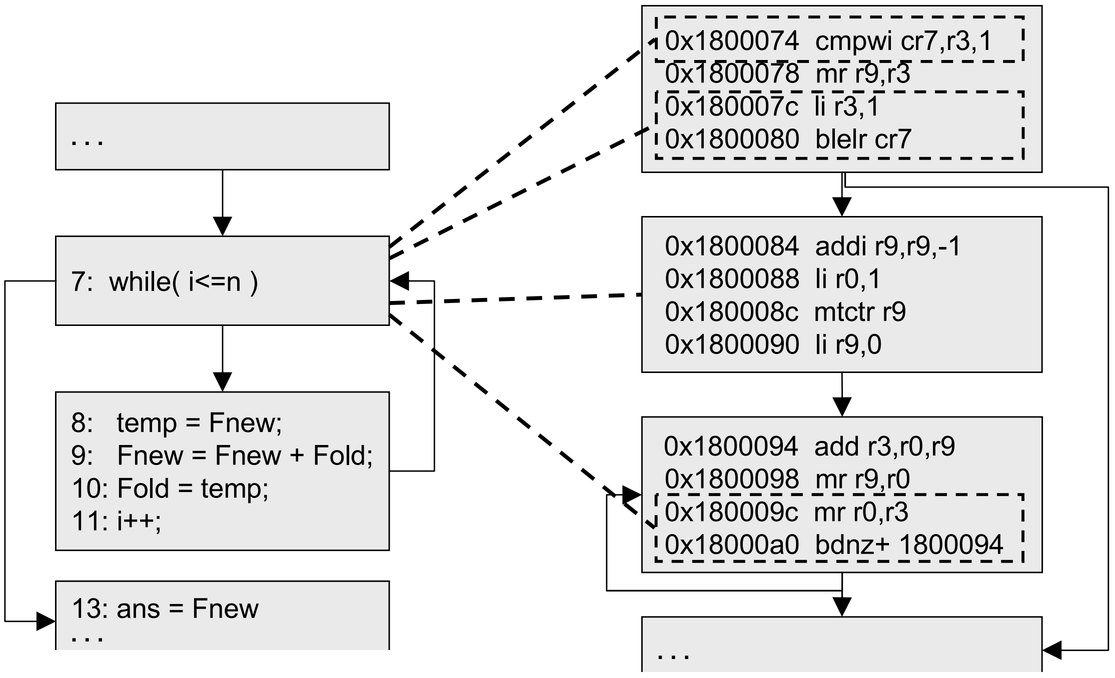
Nevertheless, in some cases, due to optimizing compilation the DWARF's line table might provide incorrect mapping information. The program *fibcall* is an example. According to the mapping information in Figure 4.8(a), four sub-blocks of binary code are generated from the *while* statement at source line 7. Figure 4.8(b) shows the relevant part of code and depicts the discussed mapping. There are two errors in the mapping: (1) the basic block from 0x1800084 to 0x1800090 is actually not generated from line 7 but the code before line 7 and (2) the instruction at 0x180009c should be generated from line 8 or line 10. Because SciSim relies on the debugging information, during the instrumentation of *fibcall* it establishes a wrong mapping, and according to the mapping, it annotates the execution time of the instructions in 0x1800084–0x1800090 into the *while* loop in the source program. During the simulation, the annotated execution time will then be aggregated as the loop iterates. However, this basic block will be executed only once in the real execution. That's one reason why the simulation of *fibcall* using SciSim had an instruction count error of 195.9% as shown in the experimental results in Section 4.3.2. Another reason will be discussed later.

The *fibcall* example shows that due to optimizing compilation the DWARF's line tables might fail to provide accurate mapping between source code and binary code and the inaccurate mapping information will lead to a large error in source level simulation. To solve this problem, a more sophisticated approach is needed to trace the code generation during the optimizing compilation. For example, the methods introduced in [42, 58] for transforming source level flow information down to binary level for worst case execution time estimation can be adapted for code tracing. Nevertheless, such methods require to extend or modify the compiler and will increase the complexity of tool development.

Nevertheless, even if we can solve the mapping problem, there still exist other problems. One serious problem is that, due to the difference between source level control flows and binary level control flows caused by compiler optimizations, the

3	→ 0x1800078—0x1800078
7	→ 0x1800074—0x1800074
7	→ 0x180007c—0x1800090
7	→ 0x180009c—0x18000a0
9	→ 0x1800094—0x1800094
11	→ 0x1800098—0x1800098

(a) Mapping Information



```
. . .
7:  while( i<=n )
8:   temp = Fnew;
9:   Fnew = Fnew + Fold;
10: Fold = temp;
11: i++;
13: ans = Fnew
. . .
```

```
0x1800074  cmpwi cr7,r3,1
0x1800078  mr r9,r3
0x180007c  li r3,1
0x1800080  blelr cr7

0x1800084  addi r9,r9,-1
0x1800088  li r0,1
0x180008c  mtctr r9
0x1800090  li r9,0

0x1800094  add r3,r0,r9
0x1800098  mr r9,r0
0x180009c  mr r0,r3
0x18000a0  bdnz+ 1800094

. . .
```

(b) Mapping Established between Source Code and Binary Code

Figure 4.8: Mapping between Source Code and Binary Code of *fibcall*

timing information cannot be back-annotated straightforwardly according to mapping. For instance, in the *fibcall* example shown in Figure 4.8, it is found that the instructions at 0x1800074, 0x180007c–0x1800080, and 0x18000a0 are really generated from source line 7. Along the binary level control flows, the instructions at 0x1800074 and 0x180007c–0x1800080 are executed only once, while the instruction at 0x18000a0 is executed the same number of times as the loop iterations. However, if we simply annotate the timing information of these instructions after source line 7. All the timing information will be aggregated as the loop iterates. Therefore, this straightforward back-annotation will cause a large error in source level simulation. As SciSim supports only straightforward back-annotation, that's another reason why the simulation of *fibcall* using SciSim had such a large error.

Another example is *insertsort*. Part of mapping of *insertsort* is depicted in Figure 4.9, which has been proven to be correct. *insertsort* contains two nested *while* loops. The instruction at 0x1800108 is generated from the C statement at line

15. Line 15 is in the inner loop, so the timing information of the instruction at 0x1800108 is back-annotated into the inner loop. However, in fact, the instruction is put outside the inner loop in the binary level CFG. The *while* statement at line 11 generates four sub-blocks of binary code. One is put inside the inner loop, while the other three are put even outside the outer loop. If we simply annotate the timing information of all the four sub-blocks after the *while* statement at line 11, it will lead to a large error. As shown in the experiment results in Section 4.3.2, the simulation of *insertsort* using SciSim had an instruction count error of 54.8%.

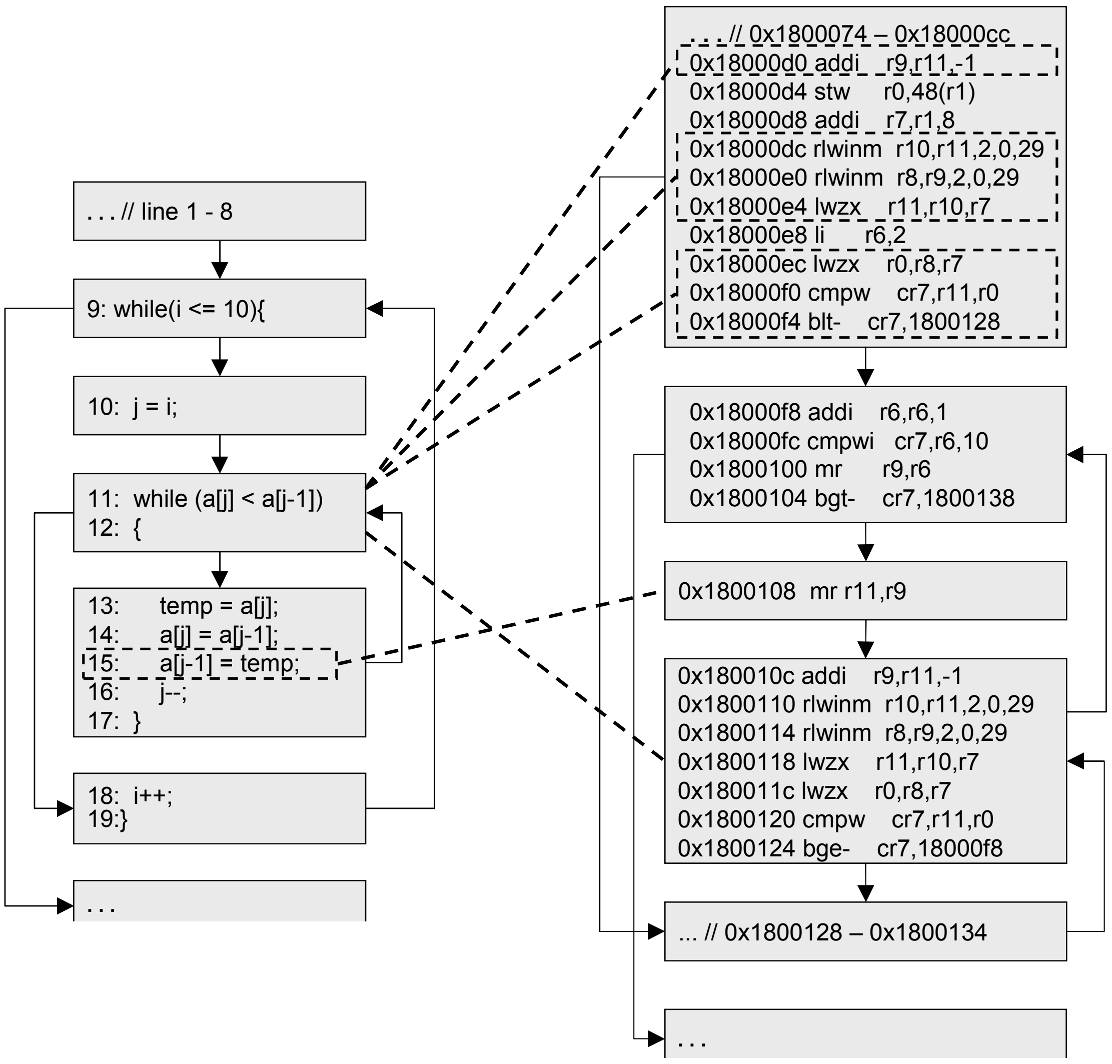
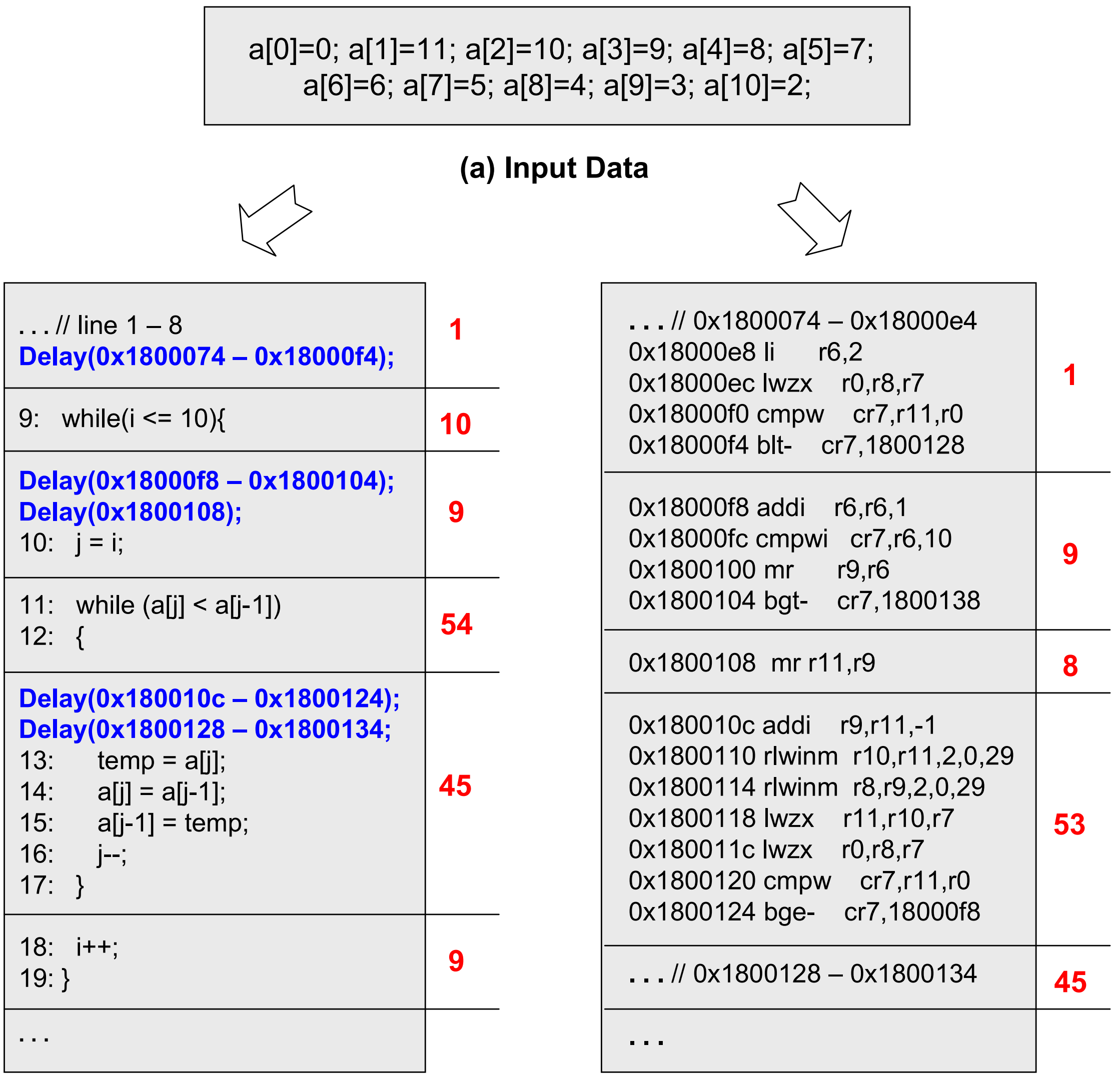

. . . // line 1 - 8
9: while(i <= 10){
10: j = i;
11: while (a[j] < a[j-1])
12: {
13: temp = a[j];
14: a[j] = a[j-1];
15: a[j-1] = temp;
16: j--;
17: }
18: i++;
19:}
. . .
. . . // 0x1800074 – 0x18000cc
0x18000d0 addi r9,r11,-1
0x18000d4 stw r0,48(r1)
0x18000d8 addi r7,r1,8
0x18000dc rlwinm r10,r11,2,0,29
0x18000e0 rlwinm r8,r9,2,0,29
0x18000e4 lwzx r11,r10,r7
0x18000e8 li r6,2
0x18000ec lwzx r0,r8,r7
0x18000f0 cmpw cr7,r11,r0
0x18000f4 blt- cr7,1800128
0x18000f8 addi r6,r6,1
0x18000fc cmpwi cr7,r6,10
0x1800100 mr r9,r6
0x1800104 bgt- cr7,1800138
0x1800108 mr r11,r9
0x180010c addi r9,r11,-1
0x1800110 rlwinm r10,r11,2,0,29
0x1800114 rlwinm r8,r9,2,0,29
0x1800118 lwzx r11,r10,r7
0x180011c lwzx r0,r8,r7
0x1800120 cmpw cr7,r11,r0
0x1800124 bge- cr7,18000f8
... // 0x1800128 – 0x1800134
. . .

Figure 4.9: Mapping between Source Code and Binary Code of *insertsort*

The two examples show that a same statement might be manipulated differently in different contexts during the compilation. For example, the *while* statement generates only one instruction in the example in Figure 4.7, whereas the *while* statement in *fibcall* generates many sub-blocks of instructions. We cannot simply make instrumentation rules for compiler-optimized code as we do for unoptimized code (Figure 4.5). Therefore, we need a method to decide the correct position in



```
a[0]=0; a[1]=11; a[2]=10; a[3]=9; a[4]=8; a[5]=7;
a[6]=6; a[7]=5; a[8]=4; a[9]=3; a[10]=2;
```

(a) Input Data

```
. . . // line 1 – 8
Delay(0x1800074 – 0x18000f4);                      1
---------------------------------------------------
9:   while(i <= 10){                              10
---------------------------------------------------
Delay(0x18000f8 – 0x1800104);
Delay(0x1800108);                                  9
10:   j = i;
---------------------------------------------------
11:   while (a[j] < a[j-1])
12:   {                                           54
---------------------------------------------------
Delay(0x180010c – 0x1800124);
Delay(0x1800128 – 0x1800134);
13:      temp = a[j];
14:      a[j] = a[j-1];                           45
15:      a[j-1] = temp;
16:      j--;
17:   }
---------------------------------------------------
18:   i++;
19: }                                              9
---------------------------------------------------
. . .
```

(b) Instrumented Source Code and Execution Count

```
. . . // 0x1800074 – 0x18000e4
0x18000e8 li      r6,2
0x18000ec lwzx    r0,r8,r7                          1
0x18000f0 cmpw    cr7,r11,r0
0x18000f4 blt-    cr7,1800128
---------------------------------------------------
0x18000f8 addi    r6,r6,1
0x18000fc cmpwi   cr7,r6,10                         9
0x1800100 mr      r9,r6
0x1800104 bgt-    cr7,1800138
---------------------------------------------------
0x1800108  mr r11,r9                                8
---------------------------------------------------
0x180010c addi    r9,r11,-1
0x1800110 rlwinm  r10,r11,2,0,29
0x1800114 rlwinm  r8,r9,2,0,29
0x1800118 lwzx    r11,r10,r7                       53
0x180011c lwzx    r0,r8,r7
0x1800120 cmpw    cr7,r11,r0
0x1800124 bge-    cr7,18000f8
---------------------------------------------------
. . . // 0x1800128 – 0x1800134                     45
---------------------------------------------------
. . .
```

(c) Binary Code and Execution Count

Figure 4.10: Instrumented Source Code and Binary Code of *insertsort*

the source code, where the timing information of each sub-block should be back-annotated. For example, for *insertsort* the tool should annotate the timing information of the instructions at 0x18000d0, 0x18000dc–0x18000e4, and 0x18000ec–0x18000f4 outside the nested loops, insert the timing information of the instructions at 0x180010c–0x1800124 into the inner loop, and put the timing information of the instruction at 0x1800108 inside the outer loop and outside the inner loop. According to our experiences, it is very hard to get an efficient solution to realize that automatically.

Now, we assume that all the mentioned problems have been solved. Then, for *insertsort*, timing annotations shown in Figure 4.10(b) are supposed to be the best annotations that can be achieved. Nevertheless, some annotated timing information still cannot be aggregated accurately. In Figure 4.10, we also show the execution count of each block of source code and binary code at its right hand side, given a set of input data shown in Figure 4.10(a). As shown, when the instrumented source code is executed, the timing information of the instruction block 0x180010c–0x1800124 is aggregated 45 times, but the instruction block is executed 53 times in the real execution. The timing information of instruction at 0x1800108 is aggregated 9 times, whereas the instruction is actually executed 8 times.

Besides the mapping problems, there are also some timing problems. To facilitate instrumentation, a basic block might be divided into sub-blocks, so static timing analysis is performed in the scope of a sub-block. Many sub-blocks consist of only less than three instructions. In the scope of such a small sequence of code the pipeline effect cannot be accurately addressed, resulting in an inaccurate estimate. Another problem is that, during the simulation, the timing information of sub-blocks might be aggregated in a different order from the execution order of binary code. Assume there are two sub-blocks of binary code. Block A is executed before block B. However, in the instrumented source code, the annotation code of block B might be executed before that of block A. If both blocks have an access to a shared resource, the reversed access order might result in simulation errors.

If all the timing effects are accurately modeled, the cycle count error should be close to the instruction count error. However, we find that in the simulation of *fibcall*, the cycle count error was much larger than instruction count error, while in the simulation of *insertsort* the cycle count error was much smaller than instruction count error. That's mainly because the static timing analysis in the scope of sub-blocks resulted in a large positive error in *fibcall* and a large negative error in *insertsort*.

Now, we conclude the above discussion. In order to get a sophisticated source level simulation approach there are three problems to be solved:

- The problem of finding accurate mapping between source code and optimized binary code. It is possible to solve this problem, for example, by realizing the code tracing function in the compiler. However, this will take large effort.

- The problem of correct back-annotation of timing information. The tool should find the correct position in the source code, where the timing information should be inserted, so that the annotated timing information can be correctly aggregated along the source level control flows.
- The problem of getting accurate timing information. We should find a way to perform accurate timing analysis, even when the binary code is splitted into very small pieces of code.

These problems exist not only in SciSim but also in other SLS approaches published in previous work.

4.5 Summary of SciSim's Advantages and Limitations

The advantages of SciSim are summarized as follows:

- Like other source level simulation approaches, SciSim allows for ultrafast software performance simulation.
- The generated simulation models are as readable as their original source code. It is easy to check the contribution of each segment of source code to the execution time of the whole program.
- SciSim uses a hybrid method for accurate performance modeling. According to the experiments, SciSim allows for an average accuracy of 98% in estimating 6 benchmark programs compiled without optimizations.
- In SciSim, the whole instrumentation approach is fully automated without the need of any human interaction. Currently, the instrumentation tool supports several processors of widely used instruction set architectures (ISAs) including PowerPC, ARM and SPARC. The tool was designed modularly to minimize users' effort to extend it. To retarget the tool to a new processor of a new ISA, we just need to add a performance model of this processor and a decoder of the ISA and do a slight adaptation in other parts of the tool. This work takes only 2–3 days according to our experiences.

However, SciSim has also many limitations summarized in the following. Some of these limitations also commonly exist in other SLS approaches.

- The defined instrumentation rules highly rely on the syntax of the programming language. Currently, only C syntax is supported. To instrument programs in another programming language, a new set of instrumentation rules must be defined.

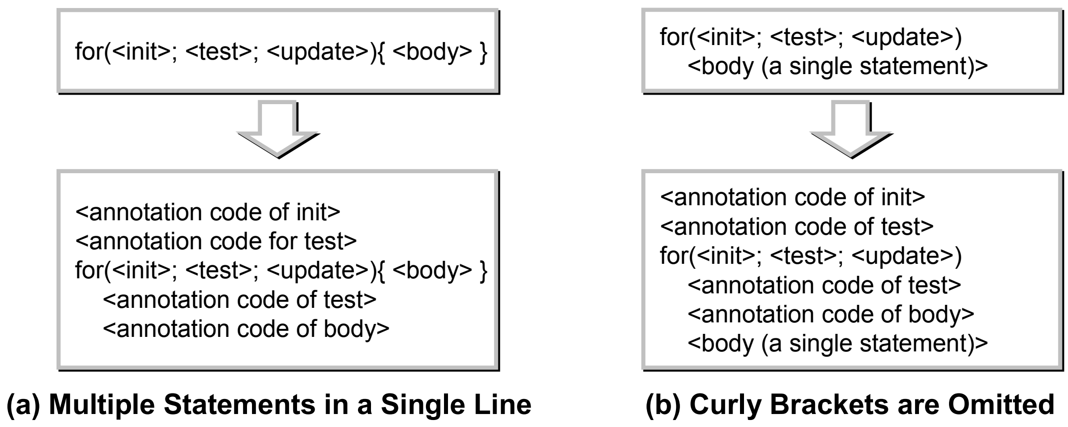

for(<init>; <test>; <update>){ <body> }
<annotation code of init>
<annotation code for test>
for(<init>; <test>; <update>){ <body> }
<annotation code of test>
<annotation code of body>
(a) Multiple Statements in a Single Line
for(<init>; <test>; <update>)
<body (a single statement)>
<annotation code of init>
<annotation code of test>
for(<init>; <test>; <update>)
<annotation code of test>
<annotation code of body>
<body (a single statement)>
(b) Curly Brackets are Omitted

Figure 4.11: Examples of Unsupported Coding Styles

- SciSim also has high requirements on the coding style of C programs to be simulated. Two examples of unsupported coding styles are shown in Figure 4.11. As shown, if multiple commands of a complex construct is written on the same source line or if the curly brackets of a construct are omitted in the case of a single-statement body, the source code cannot be instrumented correctly. The wrong instrumentation in the first example impacts the simulation accuracy, while the wrongly placed annotation code in the second example even changes the functional behavior of the original code: after instrumentation the original loop body is actually out of the loop construct. To overcome this limitation, a tool that converts original programs to programs in a supported coding style is needed.
- The largest limitation of SciSim is the problem raised by compiler optimizations. As already discussed in the last section, there are mainly three problems to be solved. These problems are common to the current SLS technique. In practice, programs are usually compiled with optimizations. For example, using GCC compilers, programs are usually compiled with the optimization level -O2. Therefore, this drawback strongly limits the usability of SciSim as well as other source level simulation approaches.

Chapter 5

iSciSim for Performance Simulation of Compiler-Optimized Software

As discussed in the last chapter, the problem raised by compiler optimizations might lead to very inaccurate source code instrumentation and makes the SLS technique hard to simulate some compiler-optimized software accurately. Motivated by this fact, we developed a new approach iSciSim that converts source code to a lower level representation, called *intermediate source code (ISC)*, and annotates timing information from the binary level back to ISC. ISC has been subject to all the machine independent optimizations and thus has a structure close to the structure of the binary code. Hence, timing information can be accurately back-annotated to ISC.

This chapter is organized as follows: First, we give an overview of the iSciSim approach in Section 5.1. The whole approach contains three working steps, intermediate source code generation, intermediate source code instrumentation, and simulation. Intermediate source code generation and instrumentation are introduced in Section 5.2 and 5.3, respectively. There are two levels of simulation: (1) simulation of the interactions between software code and a processor's components to analyze some timing effects of the processor, and (2) simulation of the interactions among processors, hardware components and shared resources to get performance statistics of a multiprocessor system. They are regarded as microarchitecture-level and macroarchitecture-level simulations, respectively. Dynamic simulation of a processor's timing effects is presented in Section 5.4. Then, Section 5.5 describes software TLM generation using iSciSim for multiprocessor simulation in SystemC. After that, in Section 5.6, experimental results are presented to compare all the discussed native execution based simulation techniques and show the benefits of iSciSim, and a case study of designing an MPSoC for a Motion JPEG decoder is demonstrated to show how iSciSim is used to facilitate design space exploration of multiprocessor systems.

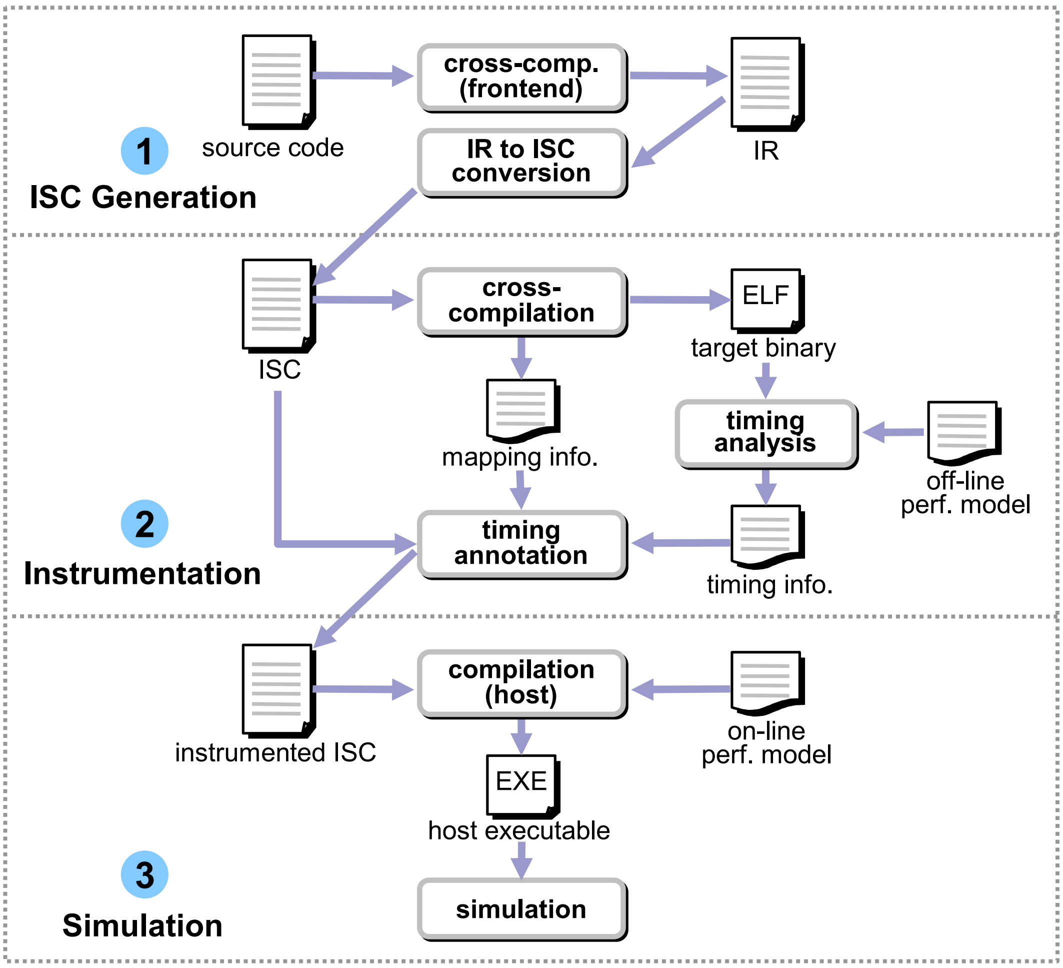


Figure 5.1: The iSciSim Approach

5.1 Overview of the iSciSim Approach

The workflow of iSciSim is illustrated in Figure 5.1. The whole approach consists of three steps: ISC generation, ISC instrumentation, and simulation. According to our current implementation, ISC is written in the C programming language and generated by slight modification of standard IRs of standard compilers. So, ISC can be regarded as IR-level C code. Because the IR is the one after all the machine-independent optimizations, the generated ISC has a structure already very close to the structure of binary code. Therefore, there is an accurate mapping between the ISC and the binary code, which makes up the basis for accurate instrumentation. More details about the ISC generation are presented in Section 5.2. As shown in Figure 5.1, in the instrumentation step, we use the same hybrid timing analysis method as in SciSim: some timing effects are analyzed at compile-time using an offline performance model and represented as timing values; other timing effects that cannot be resolved statically are simulated dynamically using an online performance model. The back-annotation of timing information is based on the mapping information extracted from debugging information. The details about the instrumentation step are introduced in Section 5.3. For simulation of a single software task, the instrumented ISC is compiled together with the online performance model to generate a host executable. We just need to click the executable to get a simulation of the task. To simulate a multiprocessor system, where multiple processors and hardware components run in parallel, the instrumented ISC of each software task is generated as a Transaction Level Model (TLM) in SystemC. The software TLMs are then connected with the TLMs of other system components to get a simulator of the whole system. The ways of dynamic simulation of a processor's timing effects and software TLM generation for multiprocessor simulation are described in Section 5.4 and 5.5, respectively.

5.2 Intermediate Source Code Generation

Basically, ISC can be in any form but must conform to the following definitions:

1. ISC is written in a high-level programming language and is compilable. This feature enables the generation of debugging information, from which a mapping between ISC and binary code can be extracted.

2. ISC retains exactly the same semantics as its original source code. When compiled with a C compiler, it shows exactly the same functionality as the original source program.

3. ISC has a structure close to that of the binary code generated from it, so that timing information can be back-annotated into it accurately.

There is more than one way to get code in accord with the definitions of ISC. In a model-driven development process, ISC can be generated from functional models using a code generator. It is even possible to let the programmer to write code with simple expressions and unstructured control statements, although it is impractical. A practical solution is to get a tool that converts complex constructs in a source program, written in a normal programming style, into simple statements. We make use of the compiler front-end to serve as such a tool. This sub-section describes our ISC generation process and the overall structure of the generated ISC.

Most widely used compilers have two decoupled parts, namely the front-end and the back-end, for performing target machine independent and dependent manipulations, respectively. In a normal compilation process, a source program is first translated to an IR by the compiler front-end, which then operates on the IR for machine-independent optimizations. The optimized IR is then forwarded to the compiler back-end for machine-dependent optimizations and object code generation. To get ISC, we let the compiler dump the optimized IR and then use a program to translate it back to C code. This C code retains both the IR structure and many high-level information of the original source code, such as variable names and high-level operators, and thus, fully conforms to the definitions of ISC.

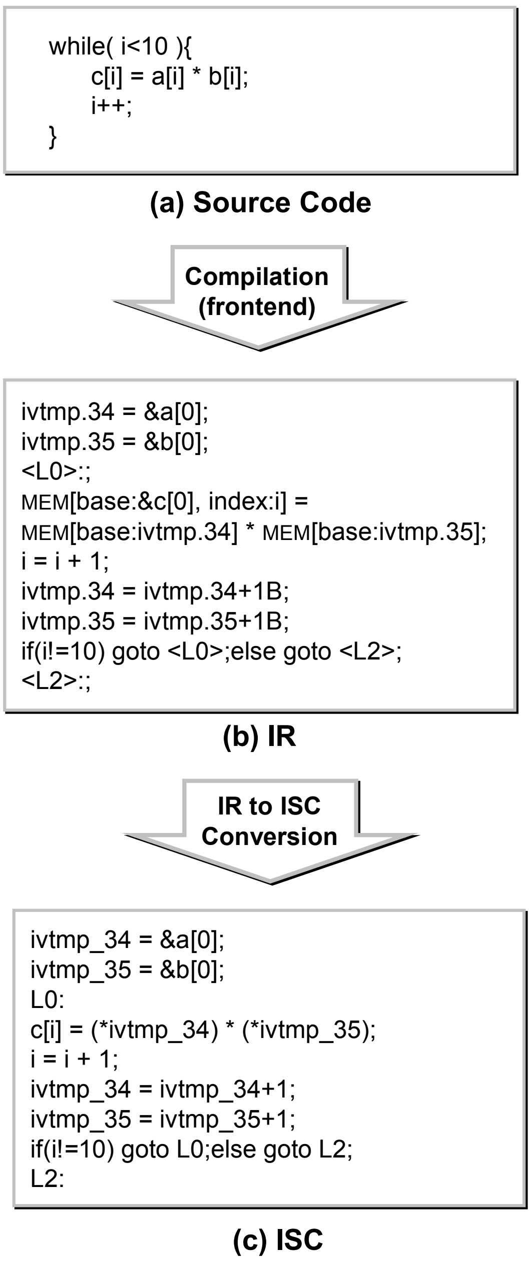
We get cross-compilers by porting GCC (GNU Compiler Collection) compilers to the target processors. GCC compilers can be found online and downloaded for free and support lots of processors in common use, such as processors of the ARM architecture and processors of the PowerPC architecture. Although we use GCC compilers to present our work, this ISC generation approach is generally applicable to other compilers that operate on IR in most widely used formats such as 3AC (3-Address Code) and SSA (Static Single Assignment). In the optimizer framework of the latest GCC compilers, the processor independent optimizations are performed on IR in GIMPLE form and then GIMPLE code is lowered to RTL code for processor-dependent optimizations. The GIMPLE representation is very similar to 3AC and SSA.

Figure 5.2 shows an example of translating a C program to an ISC program. This example is used to illustrate the whole proposed approach through this chapter. The IR shown in Figure 5.2(b) is already the one after all the passes of processor-independent optimizations. It is found that during the IR generation high-level C statements in the original source code have been broken down into 3-address form, using temporary variables to hold intermediate values. The *while* loop in the original source code is unstructured with an *if-else* statement realizing a conditional branch. Each jump is realized with a *goto* statement with the target label. Other high-level control constructs such as *for* loops and nested *if-else* statements are unstructured in the same manner. Although the original C code is lowered to a structure close to that of the binary code, most high-level information from the source code level has been retained. All the variable names are not changed.

The IR in GIMPLE form is very similar to C code. Only some naming rules do not conform to those of C. For example, the name of the temporal variable *ivtmp.34* contains a point, which is not allowed according to the variable naming rule of C. Neither is the expression of labels. As the difference between IR syntax and C syntax is very small, only a slight modification is needed to generate ISC from IR. In the illustrated example, the IR is translated to the ISC (Figure 5.2(c)) after modification of labels, variable names, and the representation of arrays.

Generally, the statements in ISC generated by the proposed process can be categorized into five types:

- Assignments: each assignment is in three-address form. This means that each assignment has no more than one operator and no more than two variables representing operands.
- Unconditional branches: an unconditional branch is a *goto* statement with a target label.



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

```
ivtmp.34 = &a[0];
ivtmp.35 = &b[0];
<L0>:;
MEM[base:&c[0], index:i] =
MEM[base:ivtmp.34] * MEM[base:ivtmp.35];
i = i + 1;
ivtmp.34 = ivtmp.34+1B;
ivtmp.35 = ivtmp.35+1B;
if(i!=10) goto <L0>;else goto <L2>;
<L2>:;
```

```
ivtmp_34 = &a[0];
ivtmp_35 = &b[0];
L0:
c[i] = (*ivtmp_34) * (*ivtmp_35);
i = i + 1;
ivtmp_34 = ivtmp_34+1;
ivtmp_35 = ivtmp_35+1;
if(i!=10) goto L0;else goto L2;
L2:
```

Figure 5.2: An Example of ISC Generation

- Conditional branches: a conditional branch is normally realized by an *if-else* statement, which has the general form of:
  `if (<Condition Test>) goto <Target Label 1> ;`
  `else goto <Target Label 2>;`
  One exception is the conditional branch with multiple targets. This kind of conditional branch is realized by a *switch* construct. Each *case* statement of the *switch* construct contains a *goto* statement and a target label.
- Labels: labels represent the branch targets. They indicate the beginning of a sequence of data flow and can be used to identify control flow nodes at the ISC level.
- Returns: a *return* statement is added implicitly for each function, even when there is no *return* statement in its original source code.

As ISC is also compilable, we can easily verify the generation process by executing both the original source code and the generated ISC with a set of input data and comparing the output data. In our experiment presented in Section 5.6, we compared the temporal behaviors between source code and ISC using a cycle-accurate instruction set simulator. For the selected benchmarks, the difference between the execution times of source code and ISC was within 1.92%. This proves that the execution time of a program can be estimated accurately by the timing analysis on the binary code generated from its ISC.

5.3 Intermediate Source Code Instrumentation

The ISC generated in the last step is forwarded to the tool for instrumentation. Figure 5.3 shows the architecture of the instrumentation tool and its workflow. The important working steps of the tool are introduced in the following sub-sections, along with the running example shown in Figure 5.4.

5.3.1 Machine Code Extraction and Mapping List Construction

For the purpose of accurate instrumentation, an efficient way to describe the mapping between ISC and binary code is very important. Most compilers do not provide any information to describe the mapping between IR and binary code. Our solution is very simple. We compile an ISC program again to generate debugging information to express this mapping.

The object files generated by our cross-compiler are in ELF (executable and linkable format) format, which is a standard binary format for Unix and Unix-like systems. An object file in ELF format is organized in several sections. We use usually only the **.text** section and the **.line** section.

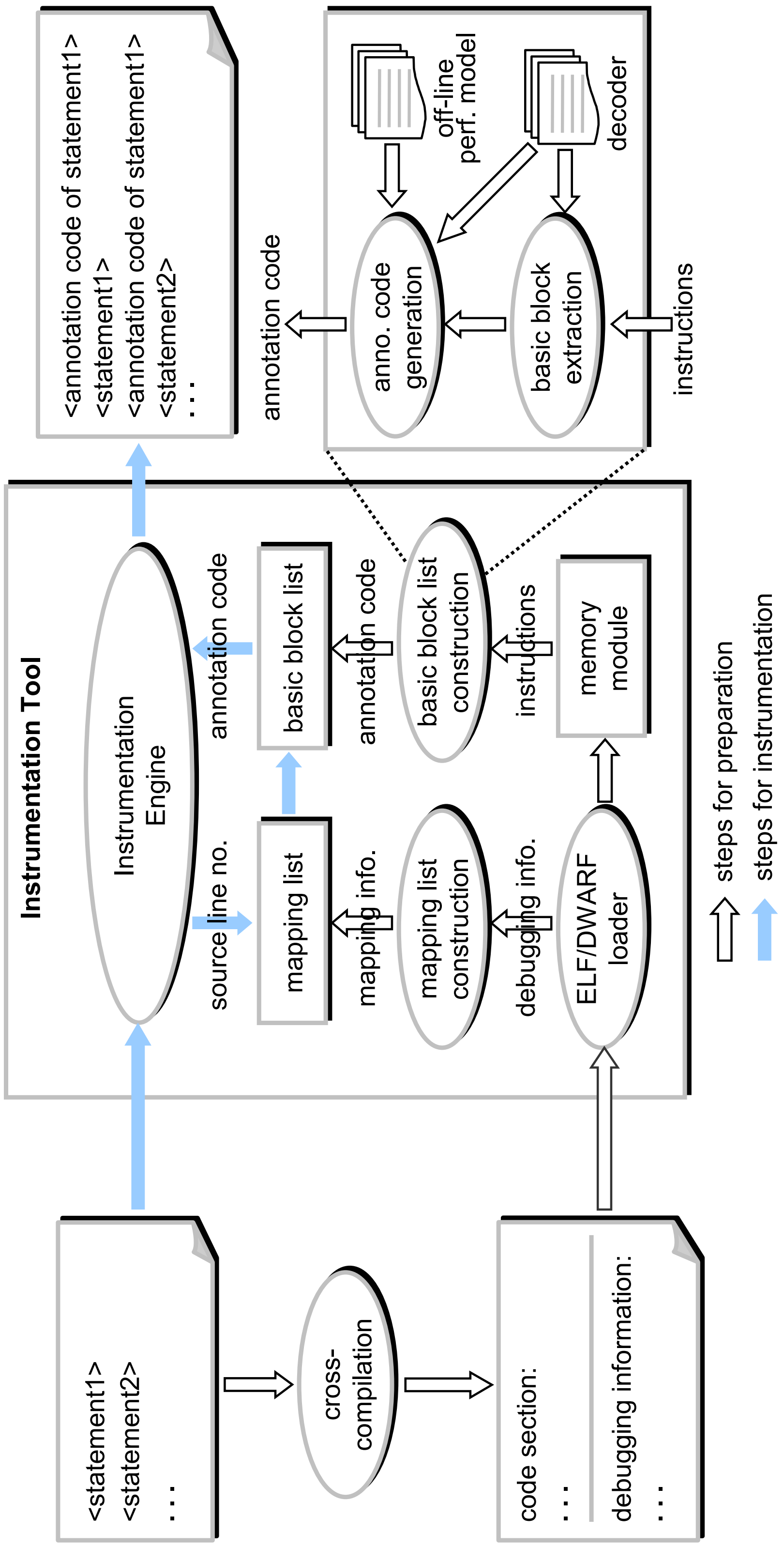
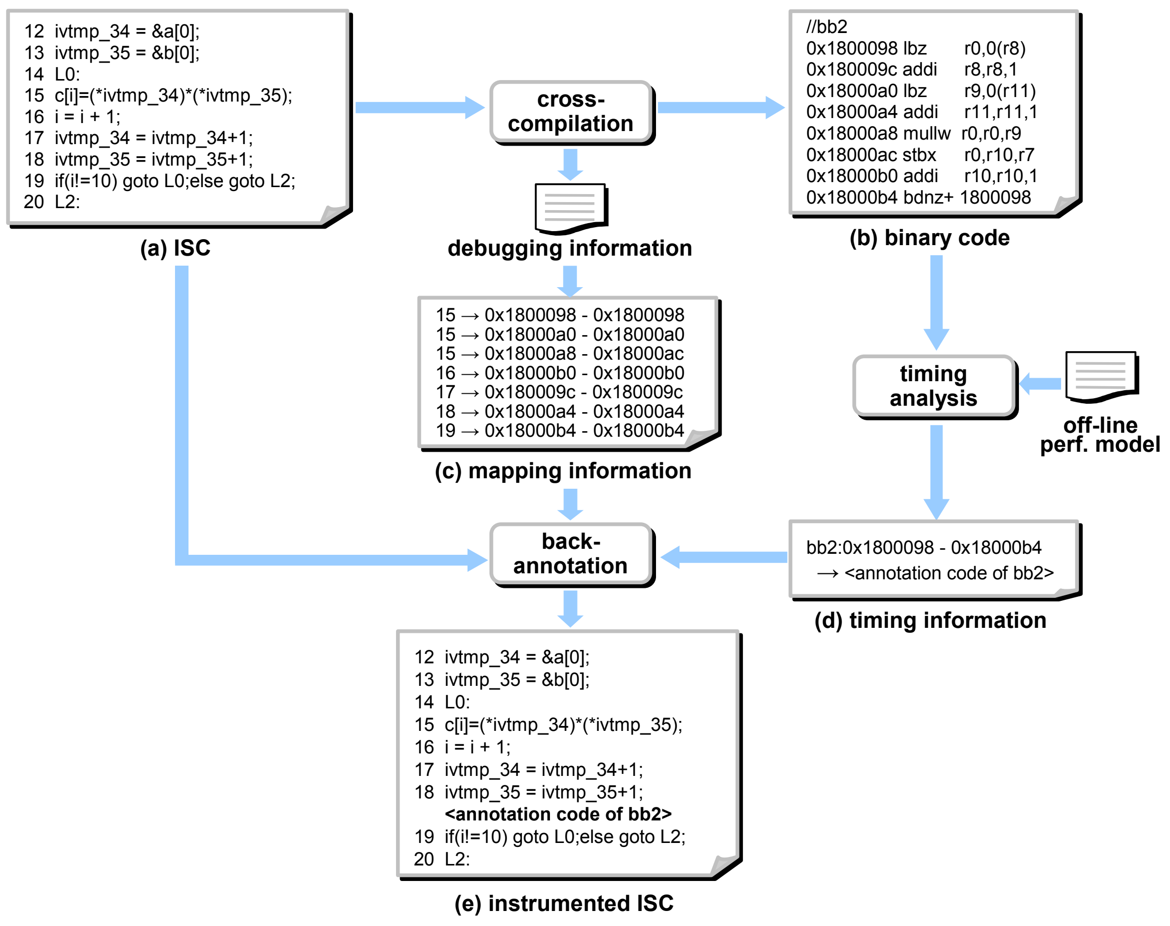


Figure 5.3: The Architecture and Workflow of the Instrumentation Tool



```
12  ivtmp_34 = &a[0];
13  ivtmp_35 = &b[0];
14  L0:
15  c[i]=(*ivtmp_34)*(*ivtmp_35);
16  i = i + 1;
17  ivtmp_34 = ivtmp_34+1;
18  ivtmp_35 = ivtmp_35+1;
19  if(i!=10) goto L0;else goto L2;
20  L2:
```

```
//bb2
0x1800098 lbz      r0,0(r8)
0x180009c addi     r8,r8,1
0x18000a0 lbz      r9,0(r11)
0x18000a4 addi     r11,r11,1
0x18000a8 mullw    r0,r0,r9
0x18000ac stbx     r0,r10,r7
0x18000b0 addi     r10,r10,1
0x18000b4 bdnz+ 1800098
```

```
15 → 0x1800098 - 0x1800098
15 → 0x18000a0 - 0x18000a0
15 → 0x18000a8 - 0x18000ac
16 → 0x18000b0 - 0x18000b0
17 → 0x180009c - 0x180009c
18 → 0x18000a4 - 0x18000a4
19 → 0x18000b4 - 0x18000b4
```

```
bb2:0x1800098 - 0x18000b4
  → <annotation code of bb2>
```

```
12  ivtmp_34 = &a[0];
13  ivtmp_35 = &b[0];
14  L0:
15  c[i]=(*ivtmp_34)*(*ivtmp_35);
16  i = i + 1;
17  ivtmp_34 = ivtmp_34+1;
18  ivtmp_35 = ivtmp_35+1;
    <annotation code of bb2>
19  if(i!=10) goto L0;else goto L2;
20  L2:
```

Figure 5.4: Instrumentation Example

The **.text** section contains the generated target machine code of a program. Figure 5.4(b) shows the code section generated from the ISC shown in Figure 5.4(a) for a PowerPC processor. The ELF/DWARF loader, which is a part of the instrumentation tool, extracts the executable instructions from the code section and stores them in a memory module for later processing. The memory module constructs the memory space of the target processor and maps the target memory addresses to the host memory. The **.line** section holds the DWARF's line table, which describes the correspondence between the source lines and the machine code. DWARF [41] is one of common debugging data formats and is associated with ELF. There is an existing GNU tool called *readelf* that can extract the DWARF's line table from the object file. The instrumentation tool then extracts the mapping information from the DWARF's line table.

Each entry of mapping information contains a source line number, the first and the last instruction addresses of the instruction block generated by this source line. Figure 5.4(c) shows the mapping information of the running example. For example, the entry "15→0x18000a8 - 0x18000ac" denotes that the instructions stored in the memory block "0x18000a8 - 0x18000ac" are generated from source line 15. If there is only one instruction generated, the first instruction address and

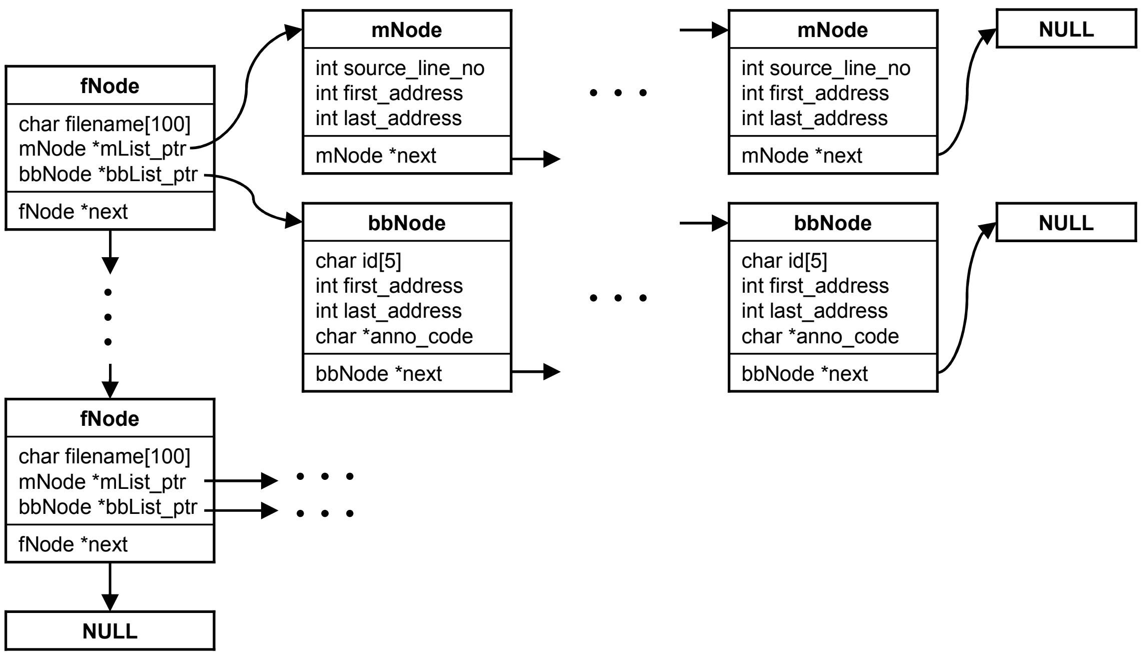


Figure 5.5: Data Structure of Mapping Information and Timing Information

the last instruction address of this instruction block are the same. One source line might correspond to more than one instruction block. As shown in Figure 5.4(c), the machine code generated from source line 15 are separated into three instruction blocks. This is the result of code motion made by the compiler back-end.

To facilitate later use, the mapping information of each program file is stored in a linked list data structure, called *mapping list* (*mList* for short). As shown in Figure 5.5, each node of a mList, called *mNode*, contains four members: *source_line_no*, *first_address* and *last_address* record the source line number, the first and the last instruction addresses of the instruction block, respectively; *next* points to the next mNode in the list. The list is sorted in the ascending order of source line numbers.

5.3.2 Basic Block List Construction

A basic block is a sequence of instructions that has only one entry point and one exit point. This means that besides the first and the last instructions this sequence of instructions contains neither a branch nor a branch target. Because the basic blocks form the nodes of a program's control flow graph and the sequence of instructions of a basic block is known at compile-time, we use basic blocks as basic units for timing information generation. The obtained timing information of each program file is also stored in a linked list data structure, called *basic block list* (*bbList* for short), as shown in Figure 5.5. Each node of a bbList, called *bbNode*, corresponds to a basic block and contains five members: *id*, *first_address*, and *last_address* hold the id of the basic block, the addresses of its first and last instructions, respectively; *anno_code* points to the annotation code that expresses the timing information of the basic block; *next* points to the next bbNode. The list is sorted in the ascending order of instruction addresses.

Algorithm 1 Basic Block List Construction

```
1: $current\_address = first\_address\_of\_program$
2: while $current\_address \neq last\_address\_of\_program$ do
3:     $instruction = \mathsf{getInstruction}(current\_address)$
4:     $\mathsf{instructionDecoding}(instruction)$
5:     if instruction is a branch then
6:         $\mathsf{insertBBList}(bbList, current\_address)$
7:         $target\_address = \mathsf{getTargetAddress}(instruction)$
8:         if $target\_address \neq first\_address\_of\_program$ then
9:             $\mathsf{insertBBList}(bbList, target\_address - 4)$
10:        end if
11:    end if
12:    $current\_address = current\_address + 4$
13: end while
14: $current\_bbNode = first\_node\_of\_bbList$
15: $current\_id = 0$
16: $current\_bbNode \rightarrow first\_address = first\_address\_of\_program$
17: while $current\_bbNode \neq \mathsf{NULL}$ do
18:     if $current\_id \neq 0$ then
19:         $current\_bbNode \rightarrow first\_address = previous\_bbNode \rightarrow last\_address + 4$
20:     end if
21:     $current\_bbNode \rightarrow id = current\_id$
22:     $\mathsf{generateAnnotationCode}(current\_bbNode)$
23:     $current\_id = current\_id + 1$
24:     $current\_bbNode = current\_bbNode \rightarrow next$
25: end while
```

We use the following algorithm to set up such a basic block list for each program: first, the binary code is scanned to find exit points of basic blocks. Instructions that may end a basic block include primarily conditional branches, unconditional branches, and the instruction before the target instruction of a branch. When an exit point is found in the scanning, a new node is created and inserted into the list. After the basic block list is created, each node contains only the value of *last_address*. Next, the list is traversed to assign an id and the value of *first_address* to each node. In the traversal, an id is assigned to each node according to the position of the node in the list. As the first instruction of each basic block, except the first basic block, is actually the instruction after the last instruction of its previous basic block, the value of *first_address* can be calculated by "`previous_bbNode->last_address + 4`". The value of *first_address* of the first basic block is assigned with the start address of the program. Meanwhile, the annotation code of each basic block is generated. During the annotation code generation, static timing analysis is performed. The whole algorithm of basic block list construction is described in Algorithm 1.

In the running example, only one basic block of machine code is generated from the shown ISC. It is the second basic block of the whole program. Therefore, it gets an id of "*bb2*". Figure 5.4(d) shows the timing information generated from this basic block. The timing information entry "$bb2 : 0x1800098 - 0x18000b4 \rightarrow < annotation\ code\ of\ bb2 >$" denotes that the instructions of the basic block *bb2* are stored in the memory block "$0x1800098 - 0x18000b4$" and the timing information of this basic block is expressed by the annotation code "$< annotation\ code\ of\ bb2 >$".

```
cycle_counter += 8;
instruction_counter += 8;
```

(a) Annotation Code 1

```
cycle_counter += 8;
iCache(0x1800098, 0x18000b4);
```

(b) Annotation Code 2

Figure 5.6: Annotation Code of the Running Example

As discussed, the annotation code of each basic block might contain not only timing values obtained by static timing analysis but also the code that is used to trigger dynamic timing analysis, depending on which timing effects are to be taken into account. Figure 5.6 shows two examples of annotation code generated for *bb2*. In the annotation code shown in Figure 5.6(a), the delay of the basic block (8 cycles), estimated by analyzing the timing effects of a superscalar pipeline, is added to the cycle counter *cycle_counter*. In the simulation, the cycle counter aggregates such timing values to get the timing of the whole program. In addition to timing values, we can also extract other useful information from the binary level for other statistics of interest. For example, the number of instructions is also counted in Figure 5.6(a). Static analysis of pipeline effects is introduced later in Section 5.3.3. During pipeline analysis, we assume optimistically all memory accesses hit cache and each memory access consumes one CPU cycle. The penalties of cache misses can be compensated at simulation run-time by means of dynamic cache simulation. In Figure 5.6(b), we show the annotation code to trigger dynamic instruction cache simulation. More details about dynamic timing analysis are presented in Section 5.4.

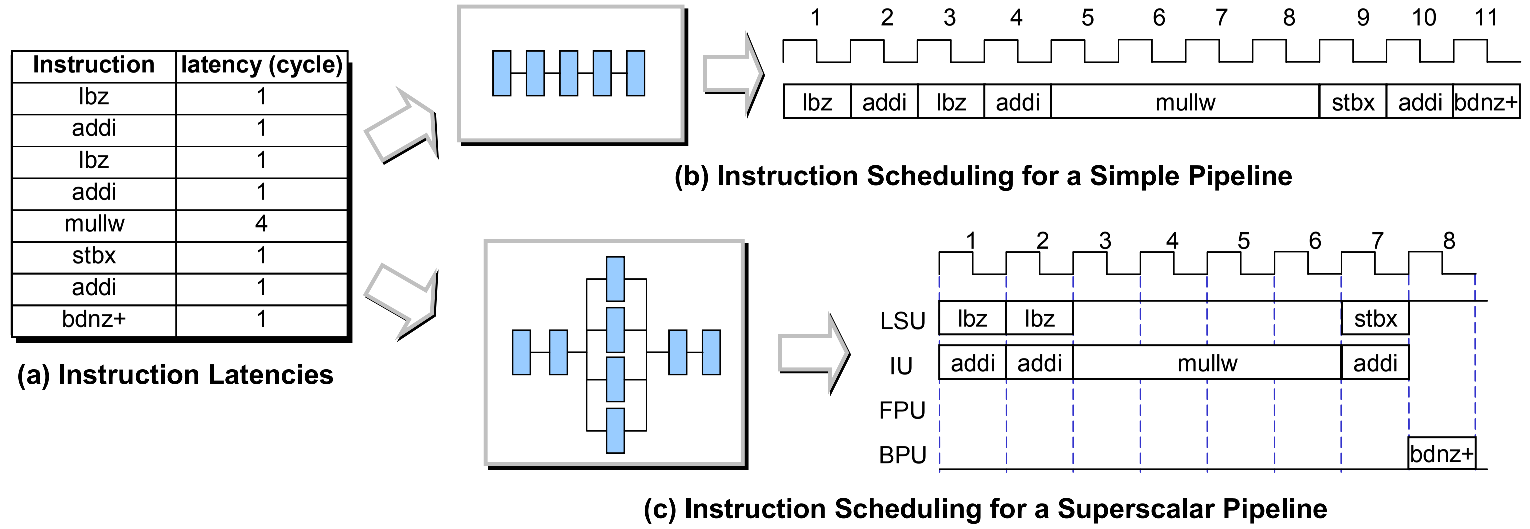
5.3.3 Static Timing Analysis

Basically, each instruction is associated with a latency. Such instruction latencies are usually specified in the processor manual. However, the execution time of a sequence of instructions cannot be estimated simply by summing up the instruction latencies, because low-level timing effects of the processor microarchitecture, such as superscalarity, caching and branch prediction, all can change the instruction timing.

As already mentioned, we use a hybrid approach for timing analysis, namely a mixture of static analysis and dynamic analysis. Correspondingly, the performance model is divided into two parts: an offline performance model and an online performance model to capture *local timing effects* and *global timing effects*, respectively. Pipeline effects are a typical example of local timing effects. When dispatched into a pipeline, only adjacent instructions affect each other, but remote instructions do not affect their respective executions. The sequence of instructions in a basic block is known at compile-time, so pipeline effects like data hazards, structural hazards and superscalarity can be analyzed statically in the scope of a basic block. The only approximation is made for the starting instructions of a basic block, the timing of which depends on the execution context set by the previous basic block. For example, we can assume that the pipeline is flushed before the instructions of each basic block are dispatched onto the pipeline model for scheduling. The advantage of static analysis is that the analysis is needed only once for each instruction, while dynamic analysis must be repeated every time when the same instruction is executed.

We take the basic block of the running example to explain the static pipeline analysis. The latencies of all the instructions are shown in Figure 5.7(a). If the instructions are scheduled onto a simple pipeline model where the instructions are issued one after another, the total execution time of this sequence of instructions is calculated by summing up their latencies, as shown in Figure 5.7(b). The result is 11 CPU cycles. For superscalar architectures, this simple, sequential instruction scheduling is no longer feasible. Instead, a model that takes the timing effect of superscalarity into account is required. Figure 5.7(c) shows an example of such a superscalar pipeline model, which contains four parallel execution units. In each CPU cycle, up to 2 instructions can be fetched and issued to the execution units. Since the pipeline is modeled at a high abstraction level, the parameters, e.g., the issue width and the number of execution units, and the penalty of hazards, can be configured easily for a specific processor. Still, with this pipeline model, the most important timing effects, such as timing overlapping between parallel execution instructions, structural hazards, and data dependencies among adjacent instructions, can be identified. For example, in Figure 5.7(c), the first instruction *lbz* and the second instruction *addi* are issued to two execution units in the same clock cycle, while the instruction *stbx* is issued after *mullw* because it must wait until *mullw* finishes writing the result to the destination register. The analysis results in an estimate of 8 CPU cycles. We presented more details about static pipeline analysis in [100].

Global timing effects are highly context-related and different execution paths set different context scenarios for an instruction. They cannot be analyzed statically without complex control flow analysis. Because the control flow depends on concrete input data during simulation, these global timing effects should be analyzed dynamically for an accurate estimation. The cache effect and the branch prediction effect are typical global timing effects.



Instruction	latency (cycle)
lbz	1
addi	1
lbz	1
addi	1
mullw	4
stbx	1
addi	1
bdnz+	1

(a) Instruction Latencies

(b) Instruction Scheduling for a Simple Pipeline

(c) Instruction Scheduling for a Superscalar Pipeline

Figure 5.7: Static Instruction Scheduling

In most cases, reasonable approximations can be made for the global timing effects to avoid dynamic timing analysis. For example, we can assume optimistically that all memory accesses hit the cache. If an execution of a program has a high enough cache hit rate, this approximation will cause only a small error, which is acceptable in system level design. Otherwise, dynamic cache simulation is needed. More details about dynamic simulation of global timing effects are presented later in Section 5.4.

5.3.4 Back-Annotation of Timing Information

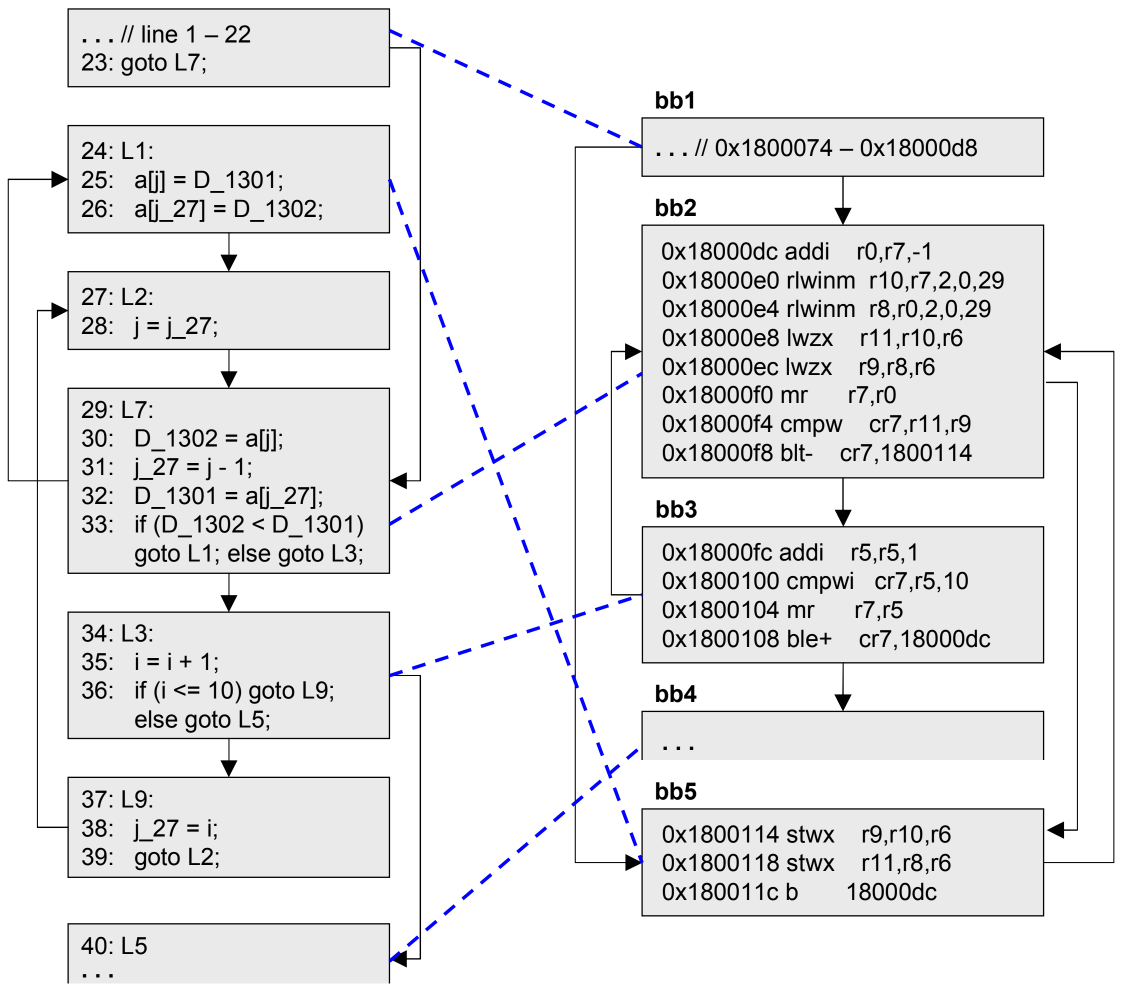
The previous steps of work can be regarded as preparation of the information needed for instrumentation. Before instrumentation starts, all the information has been stored in a well-organized data structure shown in Figure 5.5. Each program file of an application corresponds to a node in a so-called *file list*. Each node, called *fNode*, contains two pointers to the mapping list and the basic block list of the corresponding file and also a pointer that points to the next fNode in the list.

In this step, the instrumentation tool makes use of the mapping information to insert the annotation code of each basic block into ISC. We use a simple algorithm to realize the instrumentation. It works as follows: the tool traverses both the mapping list and the basic block list and marks the mNode if this mNode corresponds to an instruction block that is the last segment of a basic block. After that, the ISC is read line by line and printed out into an output file until a line that is recorded in a marked mNode is reached. Then, the annotation code of the corresponding basic block is printed out in the output file before this ISC line is printed. After the instrumentation is over, the output file contains both the original ISC and the timing annotations. For the running example, the mNode that corresponds to the last segment of *bb2* is "19→0x18000b4 - 0x18000b4". Hence, the ISC is read and printed line by line until line 19 is reached. "$<$ *annotation code of bb*2 $>$" is printed before line 19 is printed. Figure 5.4(e) shows the instrumented ISC.

This simple instrumentation algorithm may cause estimation error, if there is no exact one-to-one mapping between ISC-level basic blocks and binary-level basic blocks. Nevertheless, this error is very small in most cases. According to our experiment results presented in Section 5.6, this error was only 0.53% on average for the selected six programs. This error can also be eliminated by flow analysis that is able to find an exact mapping between ISC-level control flows and binary-level control flows. This is gained at the expense of the complexity of the instrumentation tool and the time spent in instrumentation.

```
20  → basic block 1
26  → basic block 5
33  → basic block 2
36  → basic block 3
43  → basic block 4
```

(a) Mapping Information

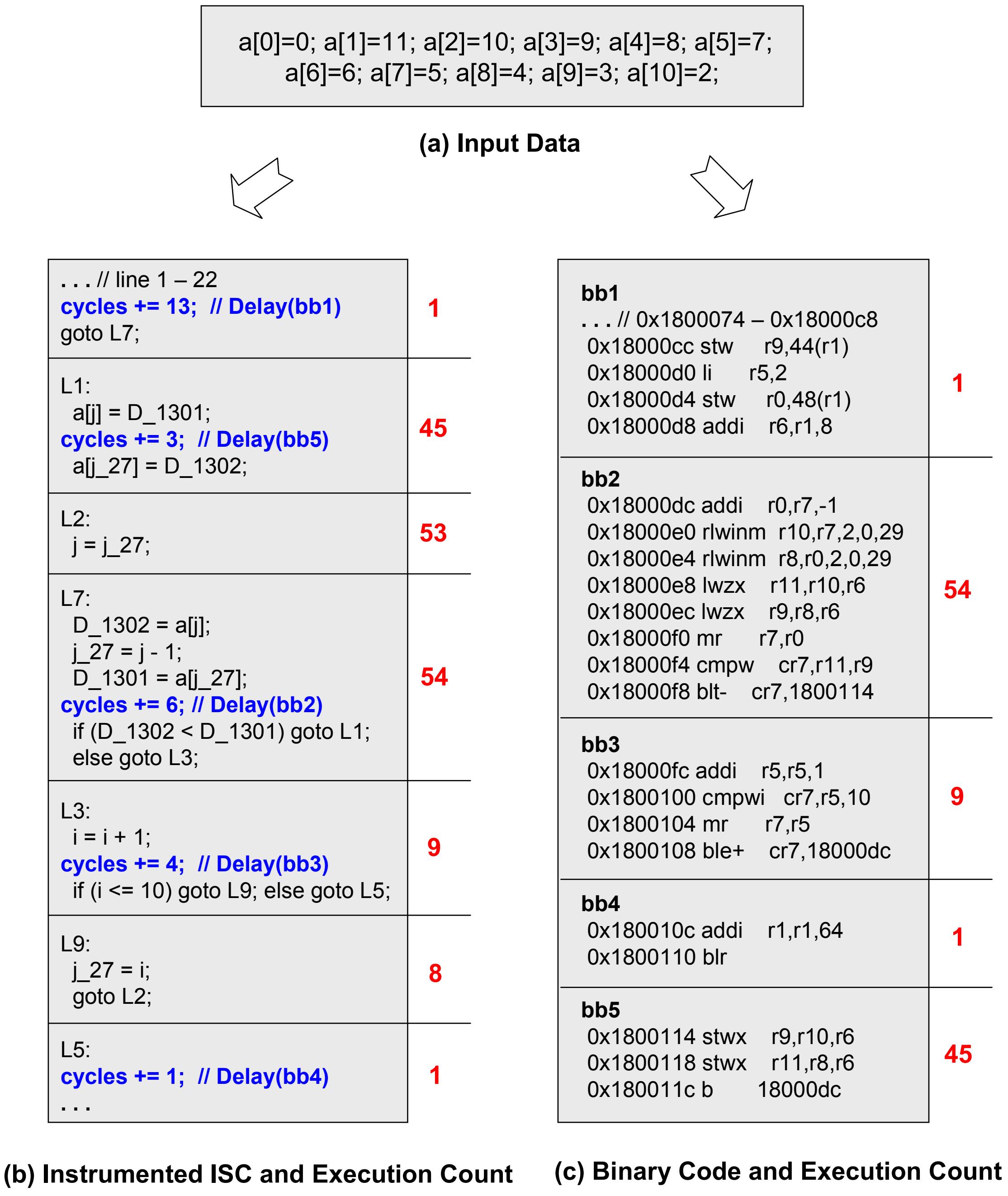


```
. . . // line 1 – 22
23: goto L7;

24: L1:
25:   a[j] = D_1301;
26:   a[j_27] = D_1302;

27: L2:
28:   j = j_27;

29: L7:
30:   D_1302 = a[j];
31:   j_27 = j - 1;
32:   D_1301 = a[j_27];
33:   if (D_1302 < D_1301)
      goto L1; else goto L3;

34: L3:
35:   i = i + 1;
36:   if (i <= 10) goto L9;
      else goto L5;

37: L9:
38:   j_27 = i;
39:   goto L2;

40: L5
. . .
```

```
bb1
. . . // 0x1800074 – 0x18000d8

bb2
0x18000dc addi    r0,r7,-1
0x18000e0 rlwinm  r10,r7,2,0,29
0x18000e4 rlwinm  r8,r0,2,0,29
0x18000e8 lwzx    r11,r10,r6
0x18000ec lwzx    r9,r8,r6
0x18000f0 mr      r7,r0
0x18000f4 cmpw    cr7,r11,r9
0x18000f8 blt-    cr7,1800114

bb3
0x18000fc addi    r5,r5,1
0x1800100 cmpwi   cr7,r5,10
0x1800104 mr      r7,r5
0x1800108 ble+    cr7,18000dc

bb4
. . .

bb5
0x1800114 stwx    r9,r10,r6
0x1800118 stwx    r11,r8,r6
0x180011c b       18000dc
```

(b) Mapping Established between ISC and Binary Code

Figure 5.8: Mapping between ISC and Binary Code of *insertsort*



```
a[0]=0; a[1]=11; a[2]=10; a[3]=9; a[4]=8; a[5]=7;
a[6]=6; a[7]=5; a[8]=4; a[9]=3; a[10]=2;
```

(a) Input Data

```
. . . // line 1 – 22
cycles += 13;  // Delay(bb1)                      1
goto L7;

L1:
  a[j] = D_1301;
cycles += 3;  // Delay(bb5)                       45
  a[j_27] = D_1302;

L2:
  j = j_27;                                       53

L7:
  D_1302 = a[j];
  j_27 = j - 1;
  D_1301 = a[j_27];                               54
cycles += 6; // Delay(bb2)
  if (D_1302 < D_1301) goto L1;
  else goto L3;

L3:
  i = i + 1;
cycles += 4;  // Delay(bb3)                       9
  if (i <= 10) goto L9; else goto L5;

L9:
  j_27 = i;                                       8
  goto L2;

L5:
cycles += 1;  // Delay(bb4)                       1
. . .
```

(b) Instrumented ISC and Execution Count

```
bb1
. . . // 0x1800074 – 0x18000c8
 0x18000cc stw     r9,44(r1)
 0x18000d0 li      r5,2                           1
 0x18000d4 stw     r0,48(r1)
 0x18000d8 addi    r6,r1,8

bb2
 0x18000dc addi    r0,r7,-1
 0x18000e0 rlwinm  r10,r7,2,0,29
 0x18000e4 rlwinm  r8,r0,2,0,29
 0x18000e8 lwzx    r11,r10,r6                     54
 0x18000ec lwzx    r9,r8,r6
 0x18000f0 mr      r7,r0
 0x18000f4 cmpw    cr7,r11,r9
 0x18000f8 blt-    cr7,1800114

bb3
 0x18000fc addi    r5,r5,1
 0x1800100 cmpwi   cr7,r5,10                      9
 0x1800104 mr      r7,r5
 0x1800108 ble+    cr7,18000dc

bb4
 0x180010c addi    r1,r1,64                       1
 0x1800110 blr

bb5
 0x1800114 stwx    r9,r10,r6
 0x1800118 stwx    r11,r8,r6                      45
 0x180011c b       18000dc
```

(c) Binary Code and Execution Count

Figure 5.9: Instrumented ISC and Binary Code of *insertsort*

5.3.5 An Example: insertsort

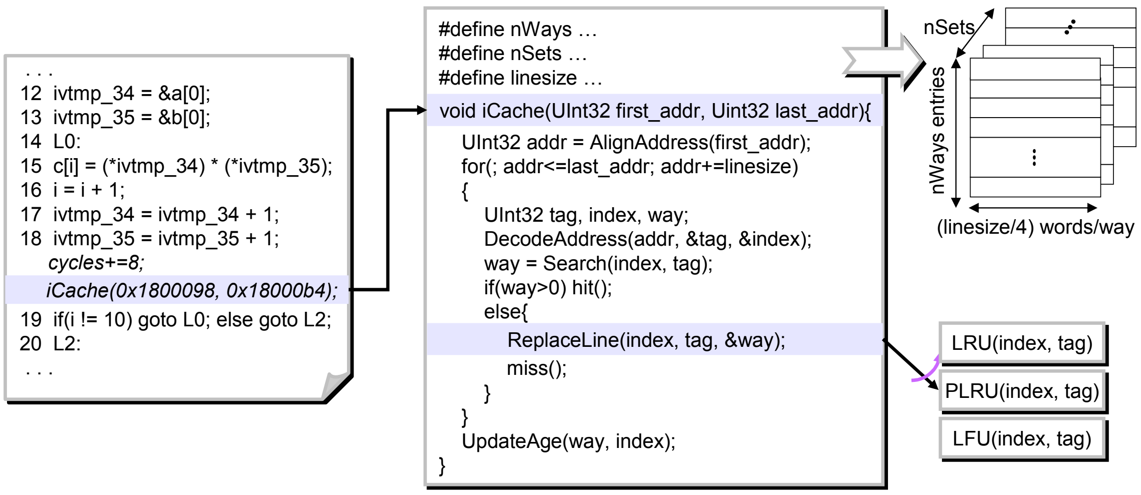
In Chapter 4, we use the program *insertsort* to show the mapping problems of SciSim. Now, we use the same example to show how these problems are solved using ISC. Figure 5.8(b) shows the ISC of *insertsort* and the binary code generated from the ISC. Figure 5.8(a) gives the mapping between the ISC level basic blocks and the binary level basic blocks, established by parsing the mapping list and the basic block list and using the simple algorithm described in Section 5.3.4. The mapping is also depicted in Figure 5.8(b). We have manually proved its correctness.

According to the mapping, the timing information of each binary level basic block can be back-annotated straightforwardly. For example, basic block 2 is mapped to ISC line 33, so the execution time of basic block 2, which is 6 cycles obtained by static timing analysis, is simply inserted before line 33. In the same way, the execution time of basic block 3 is inserted before line 36. Finally, we get the instrumented ISC of *insertsort* as shown in Figure 5.9(b). We measured the execution count of each ISC level basic block and binary level basic block, given a set of input data as shown in Figure 5.9(a). The execution count is shown at the right hand side of each basic block. It is found that all the timing information is aggregated the same number of times as the execution count of the corresponding binary level basic blocks. For example, basic block 2 is executed 54 times in the real execution. Its timing information annotated in the ISC is also aggregated 54 times.

This example proves that: (1) we can establish an accurate mapping between ISC level basic blocks and binary level basic blocks, (2) we can straightforwardly back-annotate the timing information obtained from binary level basic blocks into ISC according to the mapping, and (3) the back-annotated timing information can be correctly aggregated along the ISC level control flows during the simulation. These are validated with more programs in our experiment. The experimental results are shown in Section 5.6.

5.4 Dynamic Simulation of Global Timing Effects

To couple ISC with the online performance model, code that triggers dynamic timing analysis is annotated into the ISC. During simulation, the annotated code passes run-time data to the performance model for timing analysis. As an example, we show in Figure 5.10 how to simulate instruction cache behaviors. At runtime, the addresses of the first instruction and the last instruction of the basic block are sent to an instruction cache simulator by calling *icache(UInt32 first_addr, UInt32 last_addr)*. The cache simulator then decides how many cache hits and how many cache misses are caused by this basic block according to the present state of the cache. As shown in Figure 5.10, the function iCache models the instruction cache's behavior. DecodeAddress extracts the tag part and the index of cache set from an instruction address. Search is the function, which, when called, compares the extracted tag with the tag of each cache entry in the specified set, and returns the way number if one match is found, i.e., in the case of cache hit. Otherwise, a negative value is returned by Search and then ReplaceLine is called. ReplaceLine is defined as a function pointer, switching between different functions that implement the widely used replacement strategies, such as LRU (least recently used), PLRU (Pseudo-LRU), and LFU (least frequently used). Finally, the bits representing the access history of each cache entry are updated by calling UpdateAge which is also a function pointer pointing to the function that performs the corresponding update.



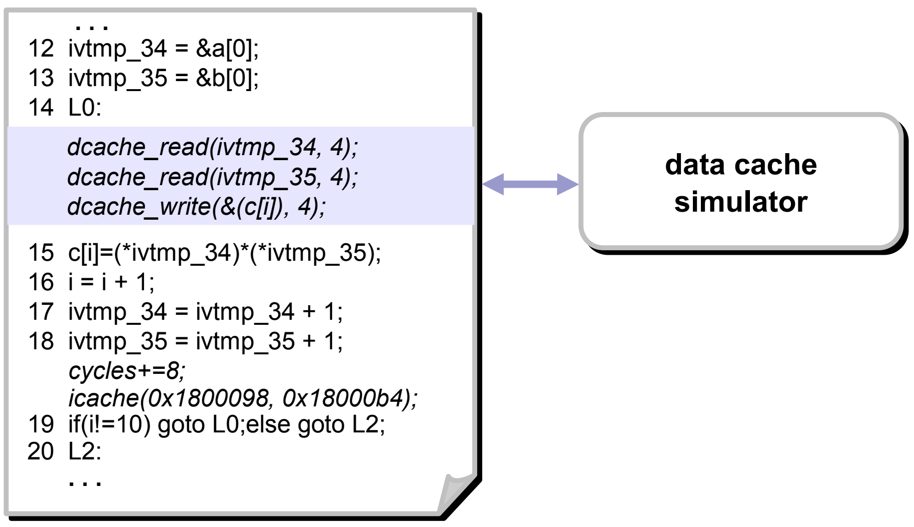
```
. . .
ivtmp_34 = &a[0];
ivtmp_35 = &b[0];
L0:
c[i] = (*ivtmp_34) * (*ivtmp_35);
i = i + 1;
ivtmp_34 = ivtmp_34 + 1;
ivtmp_35 = ivtmp_35 + 1;
  cycles+=8;
  iCache(0x1800098, 0x18000b4);
if(i != 10) goto L0; else goto L2;
L2:
. . .
```

```
#define nWays ...
#define nSets ...
#define linesize ...
void iCache(UInt32 first_addr, Uint32 last_addr){
    UInt32 addr = AlignAddress(first_addr);
    for(; addr<=last_addr; addr+=linesize)
    {
        UInt32 tag, index, way;
        DecodeAddress(addr, &tag, &index);
        way = Search(index, tag);
        if(way>0) hit();
        else{
            ReplaceLine(index, tag, &way);
            miss();
        }
    }
    UpdateAge(way, index);
}
```

Figure 5.10: Instruction Cache Simulation

Data cache simulation is more complex, because target data addresses are not visible in ISC. A simple solution proposed in [69] is to use a statistical data cache model, which generates cache misses randomly, without the need of data addresses. An alternative is to annotate code, describing register operations, to generate target data addresses at runtime, as proposed in [100]. Here, we propose another method, which is more accurate than the former solution and allows for much faster simulation than the latter one. We use data addresses in the host memory space for data cache simulation. It has been validated that the data of a program in the host memory and in the target memory have similar spatial and temporal localities, if similar compilers are used for host compilation and cross-compilation. As shown in Figure 5.11, *dcache_read(_write)(UInt32 addr, int data_size)* sends host data address and data size for data cache simulation.

In [100], dynamic simulation of the branch prediction effect is also presented. These methods of dynamic timing analysis proposed for SLS are also applicable for iSciSim. Using dynamic timing analysis, simulation accuracy is gained at the expense of simulation speed. Users should handle this trade-off and decide which timing effects are worth dynamic analysis.



```
...
ivtmp_34 = &a[0];
ivtmp_35 = &b[0];
L0:
dcache_read(ivtmp_34, 4);
dcache_read(ivtmp_35, 4);
dcache_write(&(c[i]), 4);
c[i]=(*ivtmp_34)*(*ivtmp_35);
i = i + 1;
ivtmp_34 = ivtmp_34 + 1;
ivtmp_35 = ivtmp_35 + 1;
cycles+=8;
icache(0x1800098, 0x18000b4);
if(i!=10) goto L0;else goto L2;
L2:
...
```

Figure 5.11: Data Cache Simulation

5.5 Software TLM Generation using iSciSim for Multiprocessor Simulation in SystemC

So far, we have introduced the approach of generating high-level software simulation models and the way of simulating timing effects of uniprocessor microarchitecture. Whereas, our target is to use the generated software simulation models for simulation of complex multiprocessor systems, in order to accelerate the simulation speed and shorten the time needed for design space exploration.

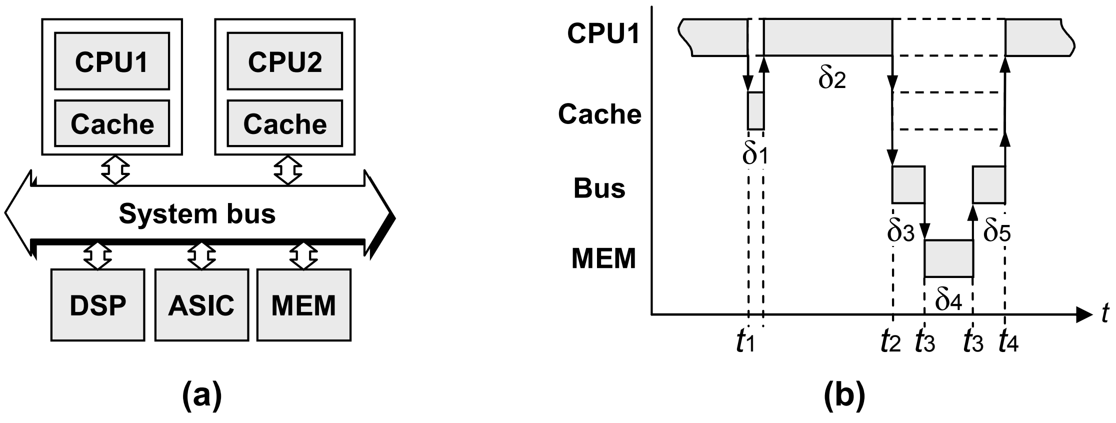


Figure 5.12: An Example of MPSoC: (a) Architecture, (b) Part of Timing Behavior

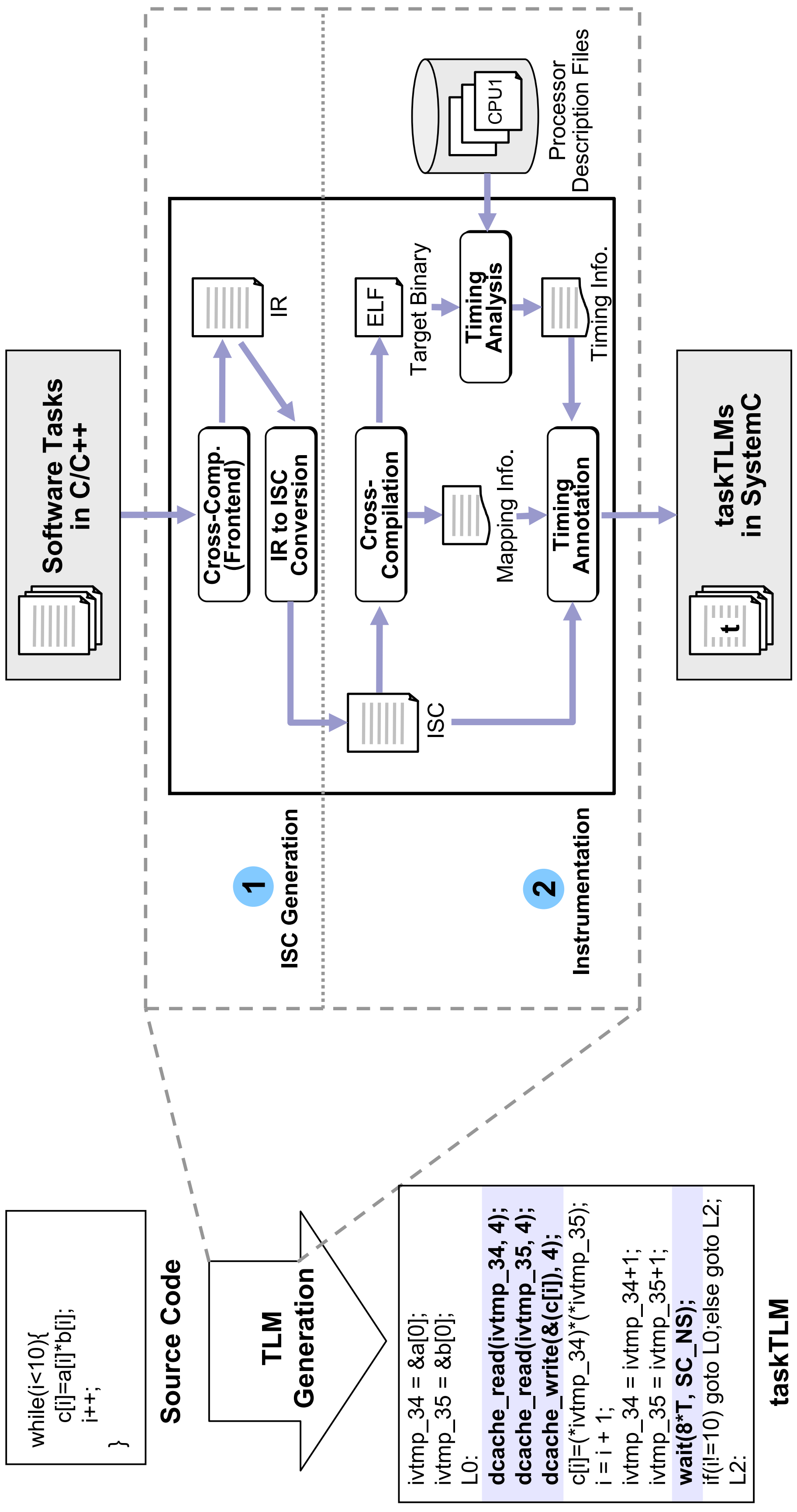
During the design space exploration of a multiprocessor system, it is important to model the system's temporal behaviors and the dynamic interactions among system components. Figure 5.12(a) shows a simplified view of an MPSoC (Multiprocessor System-on-Chip), which consists of two processor cores, a DSP, an ASIC, a shared memory and a system bus. Figure 5.12(b) illustrates part of the temporal behavior of the MPSoC. It involves two memory accesses from CPU1: one is hit in the cache, while the other is missed, resulting in different delays. Simulation is aimed to capture such temporal behaviors to generate performance statistics such as utilization of the system components, cache miss rate, and the number of bus contentions. These statistics are then used for design improvement.

In a multiprocessor simulation, the software simulation models generated by iSciSim can not only represent the functional behavior but also capture the workload of the system. They can estimate accurately the delays caused by software execution such as δ_2 in Figure 5.12(b), in order to study the influence of software execution on the system performance. To explore the memory hierarchy, they send the address and size of data, with which the cache simulator can decide whether a hit or a miss occurs. Knowing the accurate access times, we can model the resource contentions caused by simultaneous accesses to the same resource. For example, a bus contention will occur, if CPU2 also requires a bus transfer at t_2. This will lead to a longer delay on the bus.

To model such complex systems, where multiple processors and hardware components run in parallel, SystemC [55] provides a desirable simulation framework. To be integrated into a SystemC simulator, the instrumented ISC of each software task is also written in SystemC. Such a simulation model is actually a transaction level model (TLM) of the task. We call it a *taskTLM* for short in the following discussion. There is actually no large difference between the instrumented ISC in C and the taskTLM in SystemC. The latter has more code for declaring itself as a SystemC thread. In addition, in order to advance SystemC simulation time, the annotated timing information is expressed by SystemC *wait()* statements in the taskTLM. The taskTLM generation approach is illustrated in Figure 5.13, along with the running example. As shown, in the taskTLM "cycles += 8" is replaced by "wait(8*T, SC_NS)", where T is the duration of a CPU cycle with nanosecond as the unit (SC_NS). This *wait()* statement advances the SystemC simulation time for the execution time of the basic back, i.e., 8 cycles. This means that for each basic block of binary code there is a *wait()* statement explicitly annotated.

When a taskTLM is executed, after the native execution of a part of source code, there is a *wait()* statement to advance the SystemC simulation time for the corresponding basic block of binary code. Hence, the execution of a taskTLM can be divided into many segments, each consisting of the execution of a segment of source code and a *wait()* statement. The number of execution segments is equal to that of *wait()* calls.

Here, we want to discuss about two different terms of time: *simulation time* and *native execution time*. Simulation time is the time recorded in the SystemC simulation kernel to present the time consumed by executing an application on the target platform. Native execution time is the physical time spent in executing the simulation models on the simulation host. The relation between simulation time and native execution time is illustrated in Figure 5.14 using a simple example.



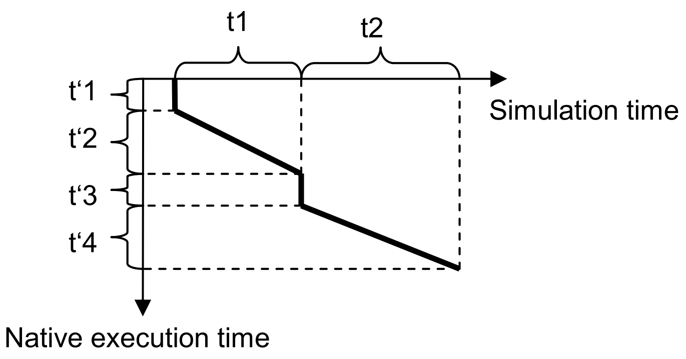
```
while(i<10){
    c[i]=a[i]*b[i];
    i++;
}
```

```
ivtmp_34 = &a[0];
ivtmp_35 = &b[0];
L0:
dcache_read(ivtmp_34, 4);
dcache_read(ivtmp_35, 4);
dcache_write(&(c[i]), 4);
c[i]=(*ivtmp_34)*(*ivtmp_35);
i = i + 1;
ivtmp_34 = ivtmp_34+1;
ivtmp_35 = ivtmp_35+1;
wait(8*T, SC_NS);
if(i!=10) goto L0;else goto L2;
L2:
```

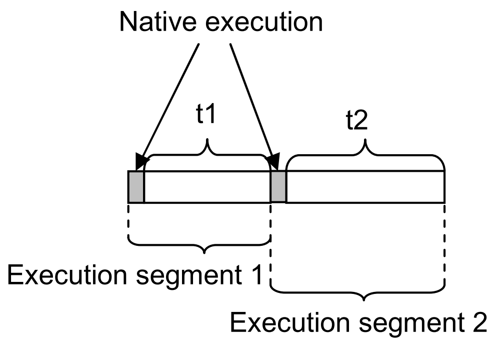
Figure 5.13: Timed taskTLM Generation

```
void Func(){
… // code segment 1 executes for t'1
wait(t1);  // executes for t'2
… // code segment 2 executes for t'3
wait(t2);  // executes for t'4
}
```

(a) An Example of taskTLM



(b) Simulation Time vs. Native Execution Time



(c) Representation of Execution Segments

Figure 5.14: Segmented Execution of taskTLM

Figure 5.14(a) is an example of taskTLM consisting of two segments. Figure 5.14(b) shows the relation between simulation time and native execution time. The source code in the first segment consumes *t'1* to execute on the host machine, whereas it does not advance any simulation time. It is estimated that the execution of this code segment takes *t1* on the target processor. Therefore, *wait(t1)* is called to advance the simulation time for *t1*. We know, a wait() statement also takes a significant amount of native execution time. As shown in Figure 5.14(b), *wait(t1)* advances simulation time = *t1* and takes native execution time = *t'2*, resulting in a segment of ramp. The second execution segment is also represented in the same manner. For convenience, we use two rectangles to represent an execution segment: a short gray rectangle representing the native execution of source code and a white rectangle representing the simulation time advancement, as shown in Figure 5.14(c). The length of the white rectangle corresponds to the amount of simulation time being advanced.

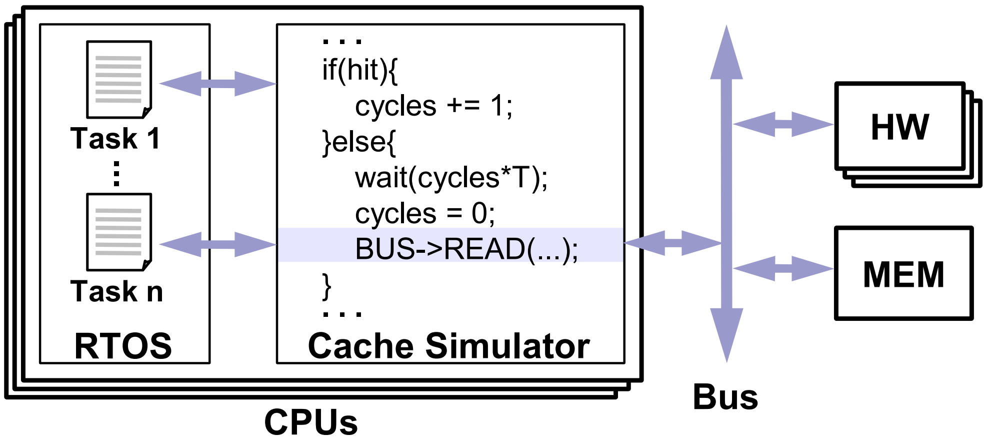


Figure 5.15: Multiprocessor Simulation

Compared with simulation in C, simulation in SystemC has the major disadvantage of low performance. The *wait()* statements annotated in the source code can cause very time-consuming context switches between the software threads and the SystemC simulation kernel. Our experiment in Section 5.6.4 shows that the *wait()* statements cause software simulation slowdown of a factor of 280. To avoid this, our solution is to keep aggregating timing information using variables. A *wait()* statement is generated to advance the simulation time, only when the software task has to interact with external devices. For example, as illustrated in Figure 5.15, a *wait()* statement is generated, when there is an access to an external memory in the case of a cache miss. In this way, the number of *wait()* statements is reduced significantly.

Figure 5.16 illustrates this optimization. After optimization, the execution segments between two IO or memory accesses are merged to one and the number of *wait()* statements is reduced from 22 to 3. According to our experiment in Section 5.6.4, the software simulation with the optimized taskTLMs has only a slowdown factor of 5 compared with simulation in C.

For simulation of the whole system including other system components, such as RTOS, peripherals and bus, additional code must be annotated during instrumentation to trigger interactions between software tasks and these system components. Figure 5.15 gives a highly simplified view of a multiprocessor system and shows an example of modeling accesses to the shared memory. Of course, to simulate such a system, many other complex issues have to be tackled. Here, we focus only on software simulation.

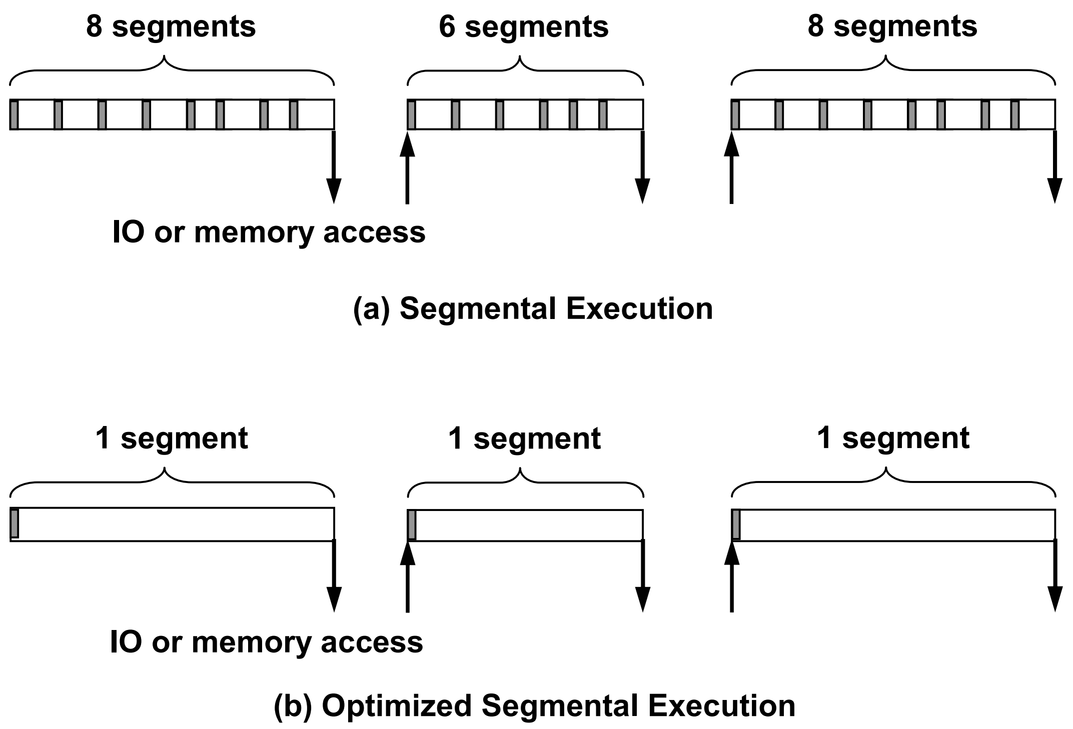


Figure 5.16: wait() Statements Reduction

5.6 Experimental Results

The experiments have been carried out mainly to compare the discussed software simulation strategies, to show the benefits of iSciSim, and to demonstrate how iSciSim facilitates MPSoC design. The simulation strategies discussed in this dissertation include instruction set simulation (ISS), binary level simulation (BLS), source level simulation (SLS) and intermediate representation level simulation (IRLS).

The experiment setup was the same as in the last chapter. The same 6 benchmark programs were used. All the programs were cross-compiled using a GCC compiler with the optimization level -O2. The simulation performance was measured on a 3 GH Intel CPU based PC with 4 GB memory, running Linux. PowerPC 603e was chosen as the target processor. The estimates from an interpretive ISS were used as a reference.

In order to get a representative of the BLS technique, we also implemented a BLS approach introduced in [77], which can represent the state-of-the-art. SciSim was used as the representative of the SLS technique. iSciSim can be regarded as an IRLS approach. We evaluated the four approaches quantitatively, in terms of simulation accuracy and speed. Simulation accuracy was measured by both instruction count accuracy and cycle count accuracy. In addition, we compared the execution times of source code and ISC, studied how cache simulation affects

Table 5.1: Source Code vs. ISC

		ISS(Source Code)	**ISS(ISC)**	
		Estimates	**Estimates**	**Difference**
fibcall	instructions	4384	4384	0.00%
	cycles	3307	3307	0.00%
insertsort	instructions	630	631	0.16%
	cycles	516	508	-1.55%
bsearch	instructions	59010	59010	0.00%
	cycles	46057	46057	0.00%
crc	instructions	17201	17457	1.49%
	cycles	10288	10289	0.01%
blowfish	instructions	262434	260913	-0.58%
	cycles	178610	179141	0.30%
AES	instructions	3624426960	3544426960	-2.21%
	cycles	2292275341	2248279321	-1.92%

the estimation accuracy and simulation performance of iSciSim, and also did simulation in SystemC. Finally, we did a case study of designing an MPSoC for a M-JPEG decoder.

5.6.1 Source Code vs. ISC

As timing analysis is performed on binary code generated from ISC, a justified concern is whether this binary code is the same as the one generated from the original source code. In the first experiment we have compared the execution times of ISC and source code, using the ISS. We tested all the six benchmarks. The estimates and the difference between execution times of source code and ISC are shown in Table 5.1. The difference is also illustrated in Figure 5.17. As shown, the ISC programs of *fibcall* and *bsearch* take exactly the same number of CPU cycles to execute as their source programs. In the simulation of the other four programs, a maximal error of -1.92% is found. The mean absolute error is 0.63%. This means that the ISC generation may change the temporal behavior of a program, but this change is small enough to be ignored.

5.6.2 Benchmarking SW Simulation Strategies

All the estimates of the six programs using the four simulation approaches are shown in Table 5.2. The estimates obtained by the ISS have taken the most important timing effects into account using a simplified performance model, which

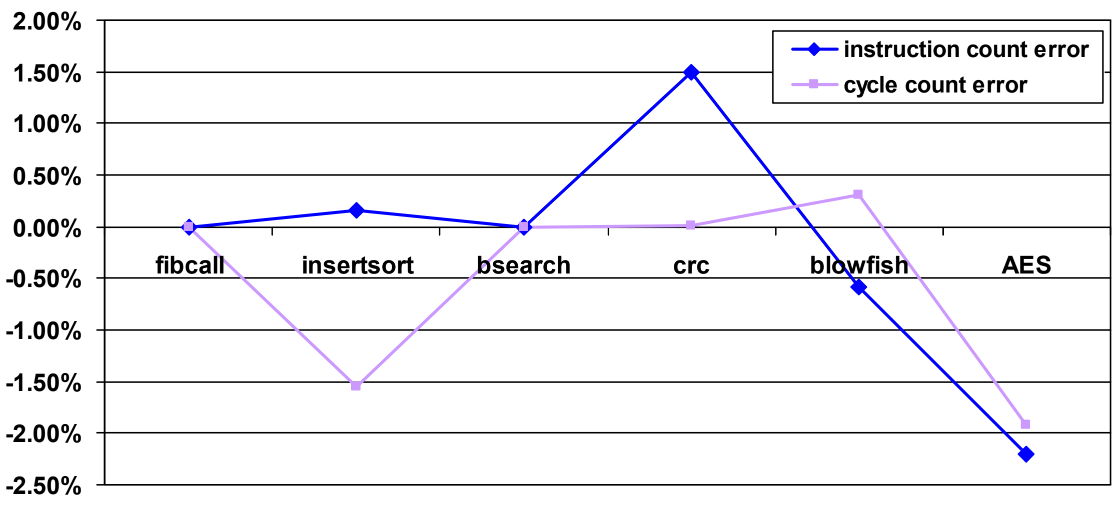


Figure 5.17: Difference between Execution Times of Source Code and ISC

consists of a pipeline model, a data cache simulator, and an instruction cache simulator. As there are still some approximations made in the performance model, the estimates from the ISS are not exactly the same as but close to the real execution times. Assume the estimates from the ISS are accurate enough and can be used as a reference to evaluate the simulation accuracies of the other tools. The estimation error (EE) of a simulation and the mean absolute error (MAE) of simulating N programs are given by:

$$EE = \frac{(Estimate_{OtherTool} - Estimate_{ISS})}{Estimate_{ISS}} \times 100\%$$

and

$$MAE = \frac{1}{N} \sum_{i=1}^{N} |EE_i|$$

The other tools follow the hybrid timing analysis approach. Namely, they analyze pipeline effects statically at compile-time and simulate the global timing effects dynamically during the simulation. Since all the selected programs are of small size and the target processor has large caches (a 16 KB data cache and a 16 KB instruction cache), the execution of these programs on the target processor has a very small cache miss rate. In this case, the cache effects are not worth dynamic analysis. Therefore, in the estimates from BLS, SLS, and iSciSim, we have taken only the pipeline effects into account and ignored the cache effects by assuming optimistically that all memory accesses hit cache. The pipeline effects were analyzed using the same pipeline model as the ISS's.

As the mapping between binary level functional representation and binary code is straightforward, BLS should allow for instruction counting with an accuracy of 100%. This was proved by the experiment. As shown in Table 5.2, BLS has an accuracy of 100% in counting the number of executed instructions of all the six programs. In the estimates of CPU cycles, BLS has an error introduced by the approximation made in the performance model in comparison to the ISS. The MAE of cycle counting is 0.83%. The same error exists in the timing analyses in SLS and iSciSim.

SLS is able to count instructions and cycles accurately for data-intensive applications, like *blowfish* and *AES*, where control flows are relatively simple. For the other programs, large errors are seen, because the control flows of these programs were changed by compiler optimizations and the mapping between binary level control flows and source level control flows was destroyed. The MAEs are 48.6% and 65.1% in instruction counting and cycle counting, respectively.

As expected, iSciSim allows for very accurate simulation with MAE of 0.99% in instruction counting and MAE of 1.04% in cycle counting. The simulation errors are also shown in Figure 5.18. The largest error is found in the simulation of *AES*, which is -2.21% and -1.74% in instruction counting and cycle counting, respectively. The sources of the estimation error in iSciSim are threefold: error caused during the ISC generation, error because of the approximations made in the timing analysis, and error introduced by the back-annotation of timing information using the simple algorithm. The three sources of error in the estimates of the six programs are illustrated in Figure 5.19. It has already been discussed above that, for the selected programs, there are MAEs of 0.63% and 0.83% caused by the ISC generation and the simplification of timing analysis, respectively. The mapping error introduced by back-annotation can be measured by comparing the execution times of ISC estimated by BLS and iSciSim. For the given programs, the MAE caused by the back-annotation is only 0.53%.

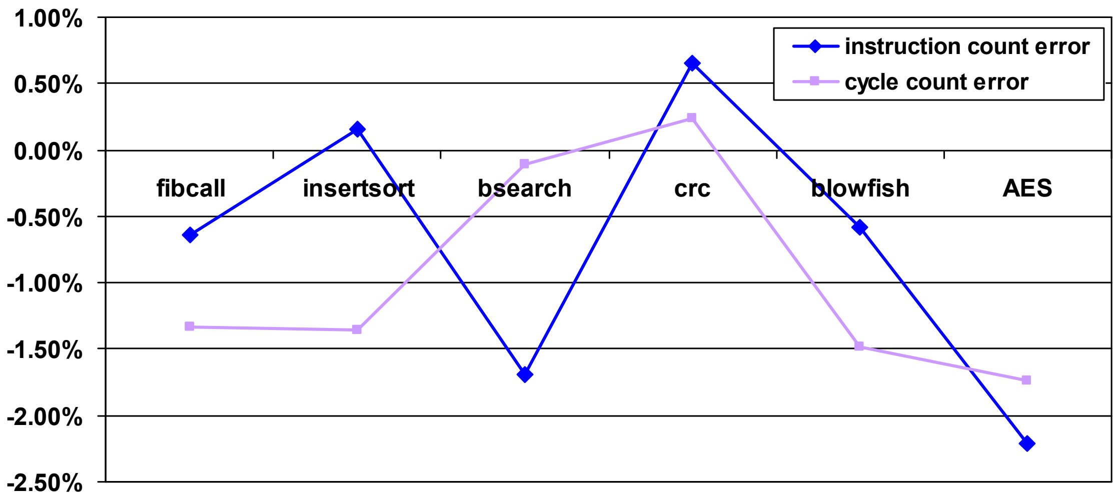


Figure 5.18: The Simulation Error of iSciSim

Table 5.2: Benchmarking Simulation Strategies

		ISS	**BLS**		**SLS**		**iSciSim**	
		Estimates	**Estimates**	**Error**	**Estimates**	**Error**	**Estimates**	**Error**
fibcall	instructions	4384	4384	0.00%	12973	195.92%	4356	-0.64%
	cycles	3307	3281	-0.79%	12996	292.98%	3263	-1.33%
insertsort	instructions	630	630	0.00%	975	54.76%	631	0.16%
	cycles	516	518	0.39%	672	30.23%	509	-1.36%
bsearch	instructions	59010	59010	0.00%	75013	27.12%	58010	-1.69%
	cycles	46057	47008	2.06%	51008	10.75%	46008	-0.11%
crc	instructions	17201	17201	0.00%	19385	12.70%	17313	0.65%
	cycles	10288	10284	-0.04%	15430	49.98%	10313	0.24%
blowfish	instructions	262434	262434	0.00%	265735	1.26%	260899	-0.58%
	cycles	178610	175919	-1.51%	187184	4.80%	175975	-1.48%
AES	instructions	3624426960	3624426960	0.00%	3624685359	0.01%	3544422940	-2.21%
	cycles	2292275341	2296272902	0.17%	2336484895	1.93%	2252276071	-1.74%

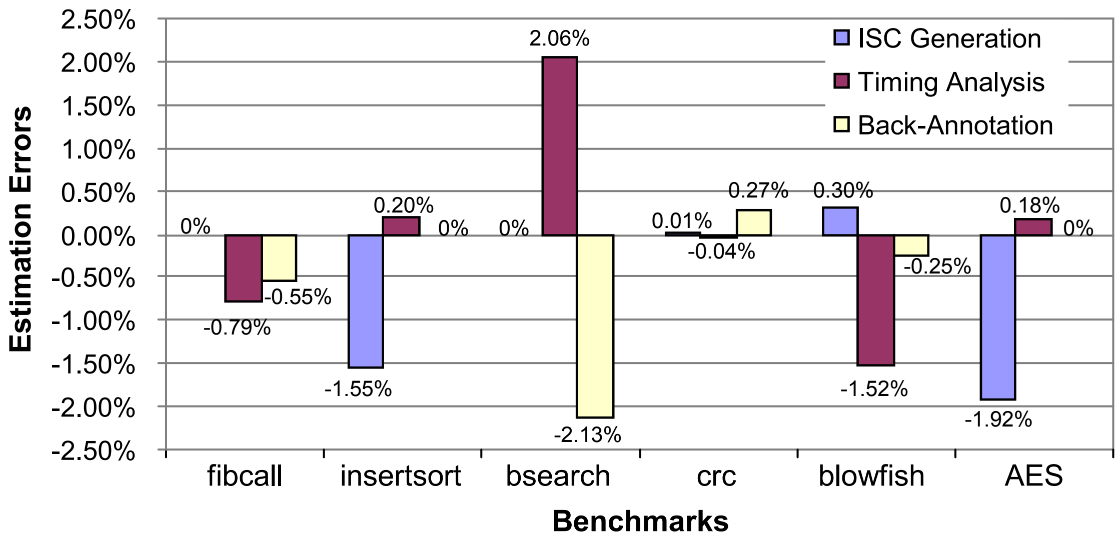


Figure 5.19: Estimation Errors Introduced by iSciSim Working Steps

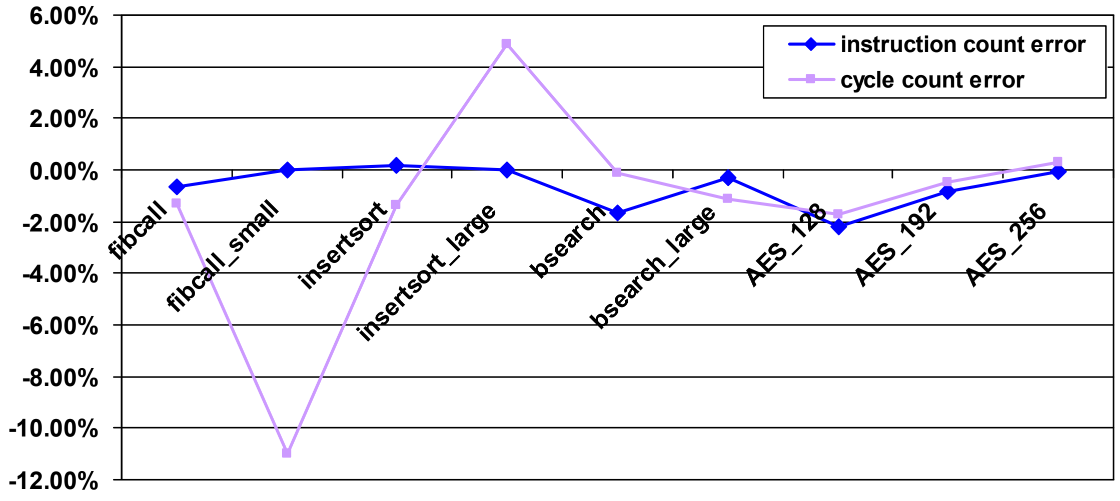


Figure 5.20: The Influence of Input Data on Simulation Accuracy of iSciSim

In addition, we also studied how input data affects the simulation accuracy. First, we changed the input data size of *insertsort*, *fibcall*, *bsearch*, and *AES*. The original *insertsort* sorts an array of length 10. We changed the array length to 50 and got *insertsort_large*. Similarly, *bsearch* calls the function *binary_search(int x)* 1000 times, while *bsearch_large* calls the function 18000 times. In contrast, input data of *fibcall* was reduced. *fibcall* calls the function *fib(int n)* seven times, each with a different value of *n*, while *fibcall_small* calls it only once. In the previous experiment, we simulated *AES* with the key length of 128 bits. Here, we also simulated it with the key length of 192 bits and 256 bits. The estimates of all these programs are shown in Table 5.3. We can see that although the input data was changed, the simulation accuracy of iSciSim is still very high for most programs. The absolute cycle count error is kept within 1.4%, except for *fibcall_small* and *insertsort_large*.

Table 5.3: Influence of Input Data on Simulation Accuracy

		ISS	BLS		SLS		iSciSim	
		Estimates	**Estimates**	**Error**	**Estimates**	**Error**	**Estimates**	**Error**
fibcall	instructions	4384	4384	0.00%	12973	195.92%	4356	-0.64%
	cycles	3307	3281	-0.79%	12996	292.98%	3263	-1.33%
fibcall_small	instructions	128	128	0.00%	296	131.25%	128	0.00%
	cycles	109	97	-11.01%	237	117.43%	97	-11.01%
insertsort	instructions	630	630	0.00%	975	54.76%	631	0.16%
	cycles	516	518	0.39%	672	30.23%	509	-1.36%
insertsort_large	instructions	14170	14170	0.00%	22258	57.08%	14171	0.01%
	cycles	11031	11616	5.30%	17256	56.43%	11569	4.88%
bsearch	instructions	59010	59010	0.00%	75013	27.12%	58010	-1.69%
	cycles	46057	47008	2.06%	51008	10.75%	46008	-0.11%
bsearch_large	instructions	722010	722010	0.00%	999010	38.37%	720010	-0.28%
	cycles	552092	548008	-0.74%	840008	52.15%	546008	-1.10%
AES_128	instructions	3624426960	3624426960	0.00%	3624685359	0.01%	3544422940	-2.21%
	cycles	2292275341	2296272902	0.17%	2336484895	1.93%	2252276071	-1.74%
AES_192	instructions	4180503763	4180503763	0.00%	4168559080	-0.29%	4144502563	-0.86%
	cycles	2640319037	2644325683	0.15%	2636361263	-0.15%	2628327283	-0.45%
AES_256	instructions	4736618963	4736618963	0.00%	4724682680	0.25%	4732620163	-0.08%
	cycles	2988396337	2996398883	0.27%	2988443263	0.00%	2996400883	0.27%

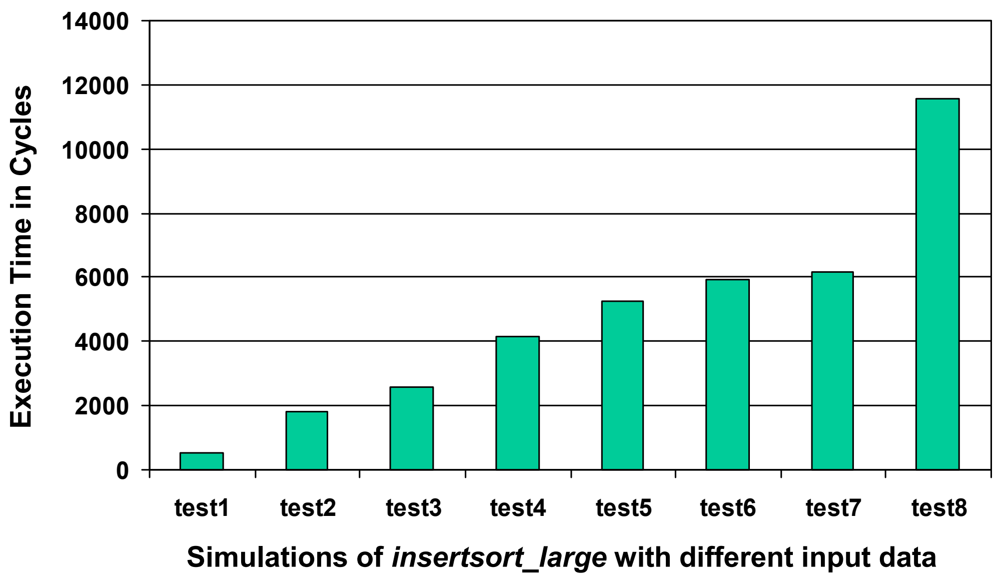


Figure 5.21: Simulation of *insertsort_large* Using iSciSim: Execution Times with Respect to Test Cases

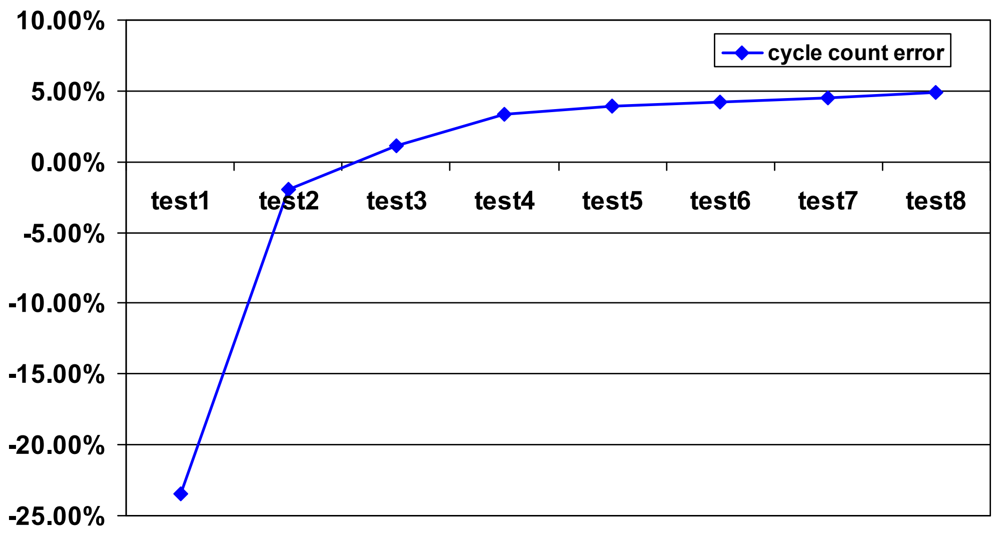


Figure 5.22: Simulation of *insertsort_large* Using iSciSim: Errors with Respect to Test Cases

The cycle count error of *fibcall_small* is -11.01%, much larger than that of *fibcall*. This is because that the simulation of both *fibcall* and *fibcall_small* had three instruction cache misses and the cache effects were not taken into account in the simulation using iSciSim. *fibcall_small* executed only 109 cycles. Ignoring the cache effects had a large influence on its simulation accuracy. The simulation of *insertsort_large* using iSciSim has an error of 4.88%, larger than the simulation error of *insertsort*. That is mainly because of the error introduced in pipeline analysis. The cycle count errors of iSciSim are also illustrated in Figure 5.20.

We further simulated *insertsort_large* with 8 different test cases using both the ISS and iSciSim. Figure 5.21 shows the execution times estimated by iSciSim with respect to the test cases. The columns are sorted in the ascending order of execution times. The first test case leaded to the best case execution time of *insertsort_large*, while the last test case caused the worst case execution time. The corresponding cycle count errors are presented in Figure 5.22. It is interesting to see a smooth curve. Test 1 has a negative error of -23.49%. This is because that the execution of *insertsort_large* had 17 instruction cache misses and 7 data cache misses and the delays caused by these cache misses were not taken into account in the simulation using iSciSim. As *insertsort_large* executed only 711 cycles in Test 1, the delays caused by cache misses accounted for a large portion of time and ignoring them resulted in a large negative error. In Test 2, the execution time of *insertsort_large* increased to 1849 cycles, because the second test case caused more loop iterations. Nevertheless, the number of cache misses did not change, so the influence of cache effects became smaller. The error is reduced to -1.95%. Then, the error becomes positive and increases slowly to 4.88%. The positive error was mainly caused by approximations made in pipeline analysis. The error was actually very small, but, with the number of loop iterations increased from Test 3 to Test 8, the error was more and more accumulated. Nevertheless, for the worst case, the error is still within 5%, acceptable in system level design.

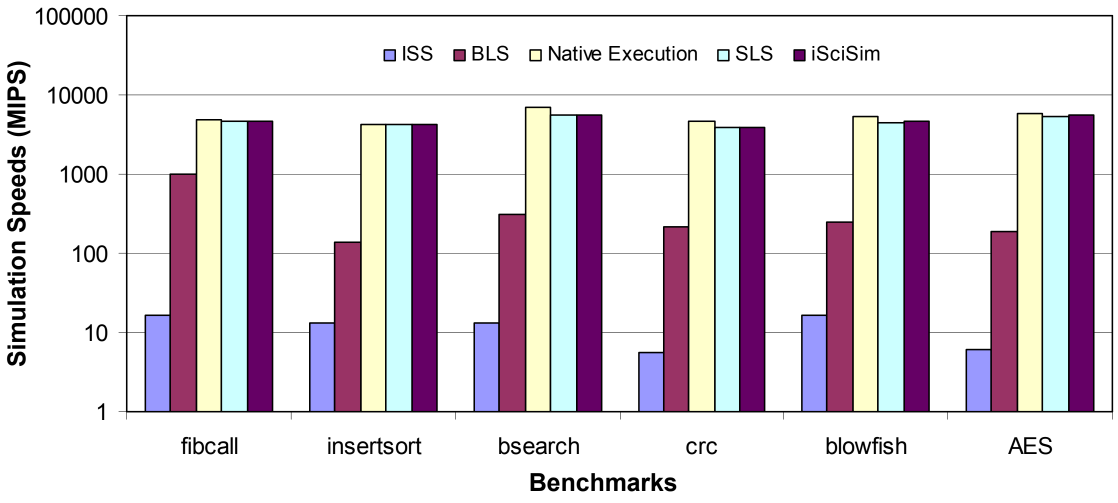


Figure 5.23: The Simulation Speeds and the Native Execution Speed

Figure 5.23 shows the speeds of native execution and simulations, measured in MIPS (million instructions per second), in a log scale. The simulation speeds as well as the native execution speed are given by

$$simulation\ speed = \frac{C_{instruction}}{(T_{simulation} \times 1000000)}$$

where $C_{instruction}$ is the total number of instructions executed when running the program on the target processor, estimated by simulation, and $T_{simulation}$ is the simulation duration.

The simulation speed of the ISS is very low, only 11.9 MIPS on average. SLS and iSciSim allow for simulation with average simulation speeds of 4669.0 MIPS and 4765.6 MIPS, respectively, almost as fast as the native execution (5198.2 MIPS on average) and much faster than BLS (349.4 MIPS on average).

5.6.3 Dynamic Cache Simulation

In this experiment, we first studied how dynamic cache simulation affects simulation accuracy of iSciSim. As discussed in the last sub-section, cache simulation will not increase the simulation accuracy, if the programs are executed on the target processor with the default data cache and instruction cache configurations (both are $4 \times 128 \times 32$ bytes). A cache configuration is expressed with $nWays \times nSets \times linesize$, where *nWays*, *nSets* and *linesize* represent the number of ways, the number of sets and the cache line size in byte, respectively. When both data cache and instruction cache configurations are changed to $4 \times 64 \times 16$ (4K) bytes, there is an error of 35.2% in the estimate of *AES* from iSciSim without cache simulation, as shown in Figure 5.24. The simulation errors of the other programs are still within 10%. In Figure 5.24, we also show how the errors are reduced with only data cache simulation, with only instruction cache simulation, and with both cache simulations. Further, when both cache configurations are changed to $4 \times 64 \times 8$ (2K) bytes, the simulation errors of *AES*, *insertsort*, and *blowfish* exceed 10%, as shown in Figure 5.25. The error in the estimate of *AES* is even larger than 80%. The error in the estimate of *blowfish* was mainly caused by ignoring the penalties of data cache misses, so iSciSim with data cache simulation alone reduces the estimation error from 35.8% down to 1.1%. In contrast to *blowfish*, the errors in the estimates of *AES* and *insertsort* were caused mainly by ignoring the instruction cache effect. Nevertheless, in the estimate of *AES*, the error caused by ignoring the data cache effect is also very large.

Further, we studied how cache simulation reduces the simulation performance of iSciSim. The performance reduction due to data cache simulation depends mainly on the percentage of load/store instructions in a program and the implementation of the data cache simulator. The performance reduction due to instruction cache simulation is dependent on the granularity of basic blocks and the implementation of the instruction cache simulator. According to our implementation of both cache simulators, a cache configuration that causes more cache misses will lead to a lower simulation speed. A smaller value of *nSets* will lead to a little shorter time for data/instruction address decoding.

In the experiment, we chose randomly four cache configurations. The simulation speeds of iSciSim with data cache simulation and instruction cache simulation are

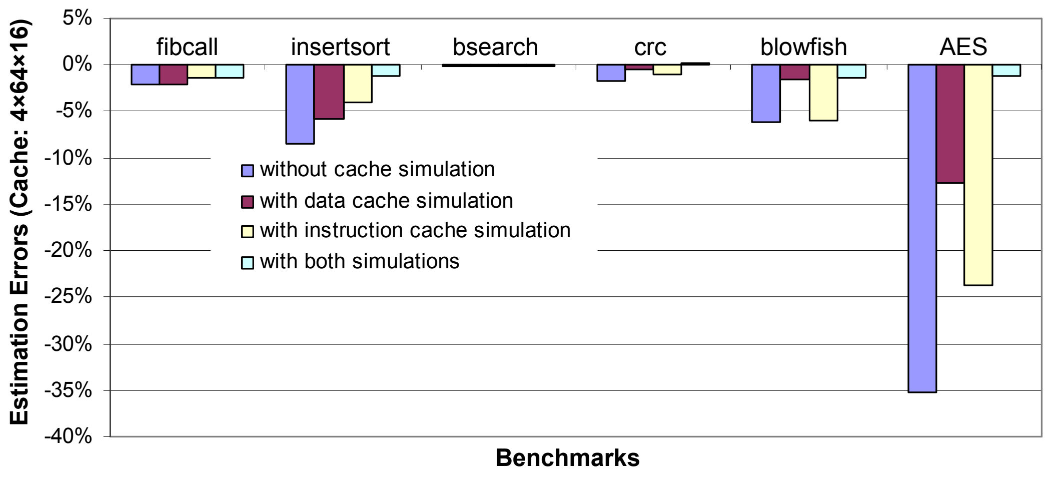


Figure 5.24: Estimation Errors of iSciSim with and without Cache Simulation (Cache Configuration: 4 × 64 × 16)

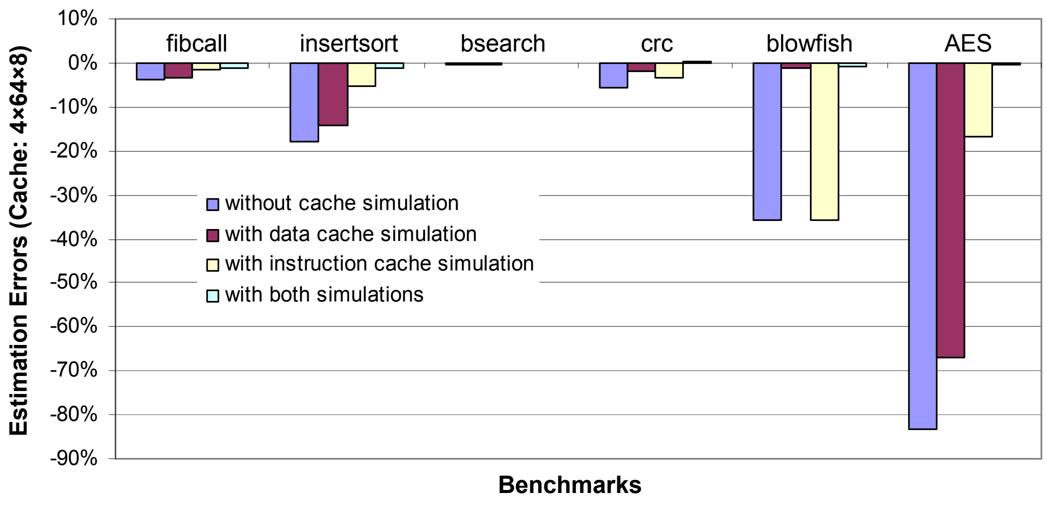


Figure 5.25: Estimation Errors of iSciSim with and without Cache Simulation (Cache Configuration: 4 × 64 × 8)

shown in Figure 5.26 and Figure 5.27, respectively. When the data cache of the default configuration is simulated, the average simulation speed is reduced from 4765.6 MIPS down to 1344.5 MIPS. In *fibcall*, the executed load/store instructions account for only 0.1% of the total instructions, so the dynamic data cache simulation has only around 5.1% overhead. In contrast to *fibcall*, *insertsort* and *AES* are two programs, where the executed load/store instructions have very high percentages of the total instructions, which are 33.3% and 31.1%, respectively. Therefore, the simulations of the two programs with data cache simulation have 91.8% and 93.9% performance reductions, respectively, in comparison to the simulations without data cache simulation. As the data cache size is reduced, the cache miss rates in the simulations of *blowfish* and *AES* increase significantly. As a result, the simulation performances are also reduced. In the simulations of the other programs, the different data cache configurations do not have large impact on cache miss rates, and therefore, only very slight changes of simulation performance are observed.

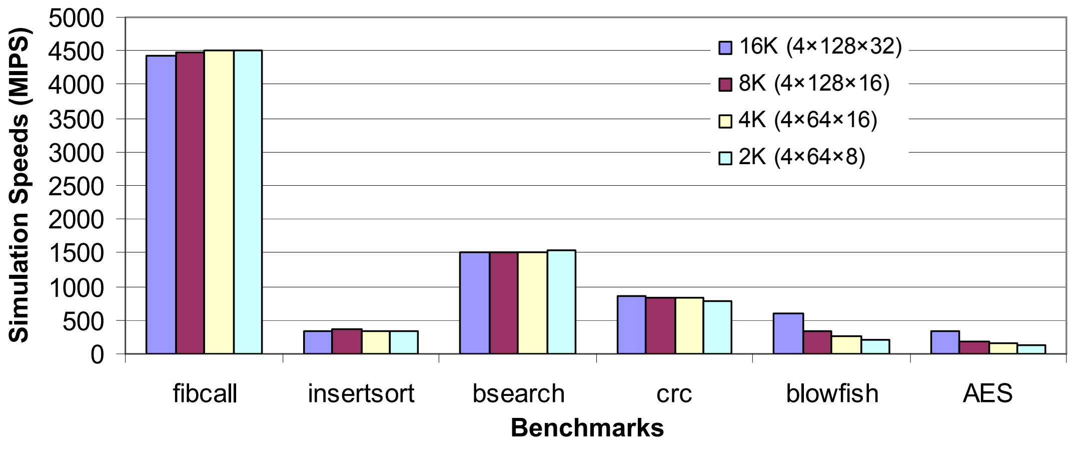


Figure 5.26: Simulation Performance of iSciSim with Data Cache Simulation

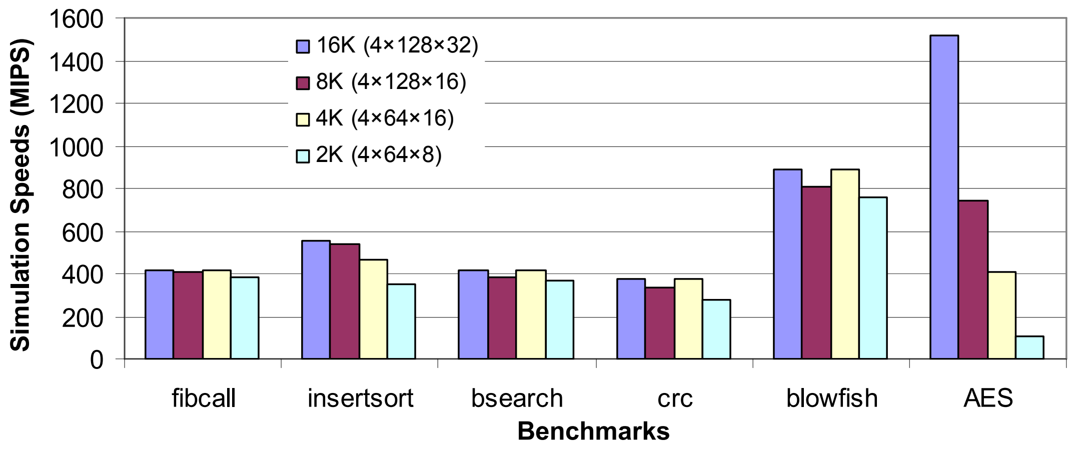


Figure 5.27: Simulation Performance of iSciSim with Instruction Cache Simulation

When the instruction cache of the default configuration is simulated, the average simulation speed of iSciSim is reduced from 4765.6 MIPS down to 695.6 MIPS. In the simulations of *AES*, when the instruction cache configuration is changed from $4 \times 128 \times 32$ stepwise to $4 \times 64 \times 8$, obvious simulation performance reductions are seen, because the reduction of instruction cache size increases the instruction cache miss rate significantly, from the one close to 0.0% to 50.2%. Slight performance reductions are also observed in the simulations of *insertsort*. The change of instruction cache configuration has relatively small impact on simulation performance in the simulations of the other programs.

Table 5.4: Simulation Duration and Factors That Determine Simulation Duration

	C_{bb}	$C_{dcacheAccess}$	$C_{dcacheMiss}$	T_{simInC}	$T_{simInSystemC}1$	$T_{simInSystemC}2$	$T_{wait()}1$	$T_{wait()}2$
fibcall	1101	5	2	1.0 us	22.7 us	7642.1 us	21.7 us	7641.1 us
insertsort	110	210	3	1.9 us	28.6 us	760.2 us	26.7 us	758.3 us
bsearch	18002	5005	4	39.0 us	72.2 us	120854.2 us	33.2 us	120815.2 us
crc	5430	1237	33	20.3 us	251.3 us	36836.2 us	231.0 us	36815.9 us
blowfish	28315	56986	270	457.0 us	2325.3 us	190437.2 us	1868.3 us	189980.2 us
AES	20022660	1104180835	446	10.5 s	10.5 s	142.9 s	0.004 s	132.4 s

5.6.4 Simulation in SystemC

This experiment is to study the slowdown of simulation speed caused by *wait()* statements, when software simulation models are wrapped in SystemC for simulation of a whole system. If cache is also simulated, the native execution time consists of three parts: (1) the execution time of the application code on the host machine, $T_{appCode}$, (2) the time needed for cache simulation, $T_{cacheSim}$, and (3) the time consumed by *wait()* statements, $T_{wait()}$. So, the simulation speed is expressed by

$$simulation\ speed\ =\ \frac{C_{instruction}}{(T_{appCode}\ +\ T_{cacheSim}\ +\ T_{wait()})\ \times\ 1000000}$$

As shown in Figure 5.28, if one *wait()* statement is annotated for each basic block, the average simulation speed is slowed down to 4.8 MIPS. If a *wait()* statement is generated only before an access to the external memory in the case of a data cache miss, the simulation runs at an average speed of 254.5 MIPS. Compared with the simulation in C, which is at 1344.5 MIPS on average, there is a slowdown factor of 5.

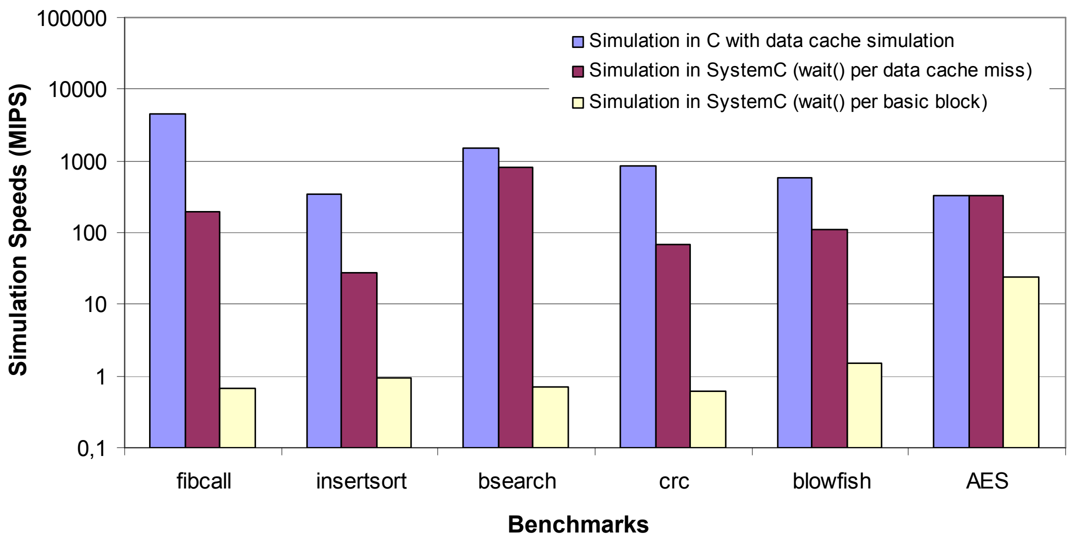


Figure 5.28: Simulation in C vs. Simulation in SystemC

The data shown in Table 5.4 helps us to understand the factors that determine the simulation speeds. C_{bb}, $C_{dcacheAccess}$, and $C_{dcacheMiss}$ are the number of basic blocks, the number of data reads/writes, and the number of data cache misses, respectively. T_{simInC}, $T_{simInSystemC}1$, and $T_{simInSystemC}2$ are respective times spent in simulation in C, simulation in SystemC with one *wait()* executed while a cache miss occurs, and simulation in SystemC with one *wait()* for each basic block. As simulation in C has no overhead due to *wait()* statements, T_{simInC} is equal to $T_{appCode} + T_{cacheSim}$. Therefore, by subtracting T_{simInC} from $T_{simInSystemC}1$ and $T_{simInSystemC}2$, we get the times consumed by *wait()* statements $T_{wait()}1$ and $T_{wait()}2$. Assuming that each *wait()* consumes a constant time $t_{wait()}$, we get $T_{wait()} = t_{wait()} \times N_{wait()}$, where $N_{wait()}$ is the number of *wait()* statements. If one *wait()* corresponds to each basic block, $N_{wait()}$ is equal to the number of basic blocks C_{bb}. So, we get $T_{wait()}2 = t_{wait()} \times C_{bb}$. According to the measured data, a single *wait()* takes around 7 us to execute on the experiment PC. Compared with $T_{wait()}2$, $T_{appCode} + T_{cacheSim}$ is almost ignorable. If one *wait()* corresponds to each data cache miss, $N_{wait()}$ is equal to the number of cache misses plus one *wait()* that is used to advance the simulation time for the code after the last cache miss. Thus, $T_{wait()}1 = t_{wait()} \times (C_{dcacheMiss} + 1)$. Usually, $C_{dcacheMiss} + 1$ is much smaller than C_{bb}. That is the reason why this strategy can significantly improve the simulation speed.

5.6.5 Case Study of MPSoC Simulation: A Motion-JPEG Decoder

We also did a case study to show how iSciSim facilitates MPSoC design space exploration. In the case study we designed an MPSoC for a Motion-JPEG (M-JPEG) decoder. We have chosen this application because it is a real-life application. Although it is not too complicated but has enough features to illustrate the use of our simulation method. This experiment was carried out on a laptop equipped with an Intel Core2 Duo CPU at 2.26 GHz and 2 GB memory. The execution environment was Cygwin on Windows XP.

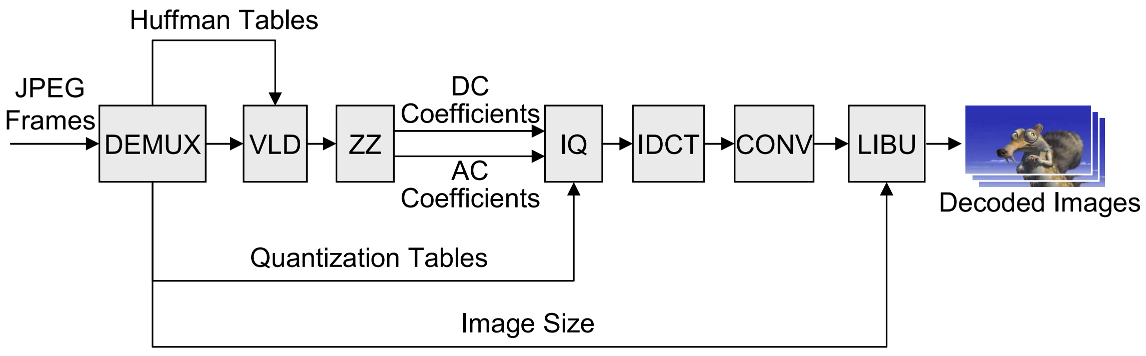


Figure 5.29: M-JPEG Decoder Block Diagram

M-JPEG is a multimedia format, where each video frame is compressed as a JPEG image, and the M-JPEG decoder is a typical multimedia application. The most building blocks of a M-JPEG decoder are also used in many other image/video processing applications. The block diagram of the M-JPEG decoder used in this case study is shown in Figure 5.29. The input is a stream of JPEG images. DEMUX extracts some information data from each image, such as *Huffmann tables*, *quantization tables*, image blocks, and image size, and sends them to the corresponding blocks. VLD, ZZ, IQ, and IDCT perform *variable length decoding*, *Zigzag scanning*, *inverse quantization*, and *inverse discrete cosine transformation*, respectively. Then, CONV converts the image format from *YUV* to *RGB*. LIBU receives RGB blocks and orders them in lines. At last, the RGB lines are sent for displaying.

The video used in the case study has a resolution of 256x144 pixels per frame. The design was aimed at archiving 25 frames per second (fps) decoding rate. We used the proposed simulation method to perform system level design space exploration to find the best design from the design alternatives. In principle, the design space for such an application could be very large. The target platform could be a high-performance uniprocessor or a multiprocessor system-on-chip (MPSoC) that is either homogeneous or heterogeneous. However, we considered only homogeneous MPSoCs consisting of multiple identical PowerPC cores to simplify the design problem. Each processor has a 16 KB instruction cache and a 16 KB data cache. This kind of simplification does not hinder the demonstration of the key concepts. The same simulation methodology is surely applicable for exploring a larger design space.

The described M-JPEG decoder was first implemented in a sequential C program. Then, the C program was transformed to an ISC program. We measured the execution times of both the source program and the ISC program on the experiment PC. For a sequence of 200 frames of size 256x144 pixels, the execution times were 1.47 s and 1.54 s, respectively. This proves again that the execution time of an ISC program is close to its original source program. Hence, it is fully feasible to use the ISC program for implementation instead of the source program.

Table 5.5: Performance Profiling of the M-JPEG Decoder

Module	Execution time in cycles	Time percentage
All	5612778816	100%
VLD	590901512	10.5%
IQ+ZZ	578880000	10.3%
IDCT	3177253144	56.6%
CONV	1175923358	21.0%
LIBU	71614200	1.3%
DEMUX and Others	18206602	0.3%

Before parallelizing the sequential program, we profiled the execution time of the application on the target processor and got the contribution of each module to the total execution time, as shown in Table 5.5. The data is also illustrated in Figure 5.30. The execution times were obtained by running the instrumented ISC of the whole program. The execution of the instrumented ISC took only 2.12 s.

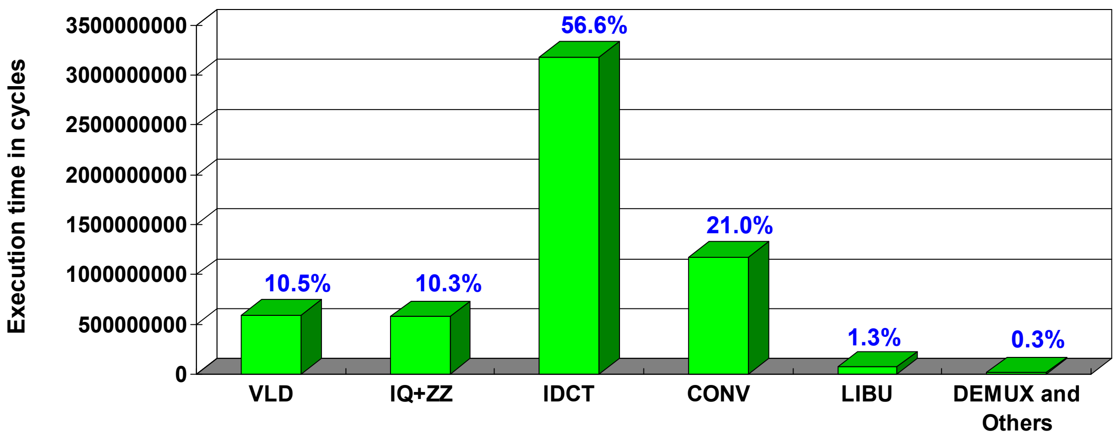


Figure 5.30: Performance Profiling of the M-JPEG Decoder

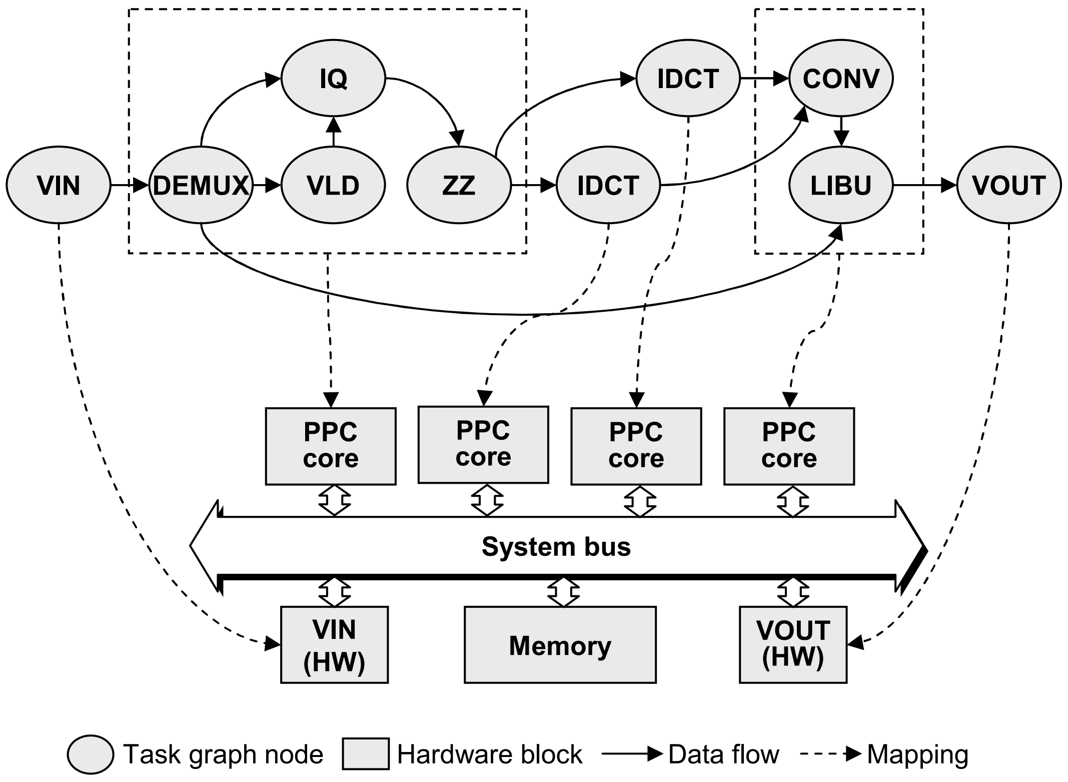


Figure 5.31: Mapping of the M-JPEG Application onto the MPSoC Architecture

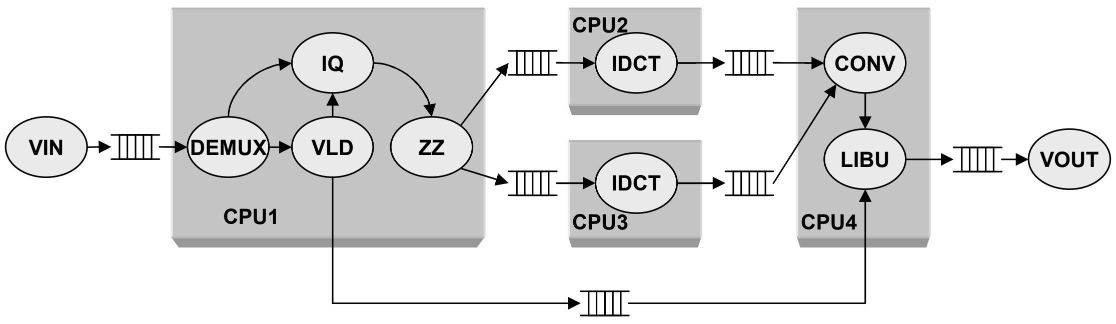


Figure 5.32: Point-to-Point Communication Using FIFOs

This fast program performance profiling is also an advantage of our approach, because profiling using iSciSim is much faster and easier than using an ISS.

The performance data shown in Table 5.5 is very useful for parallelizing the application model and its mapping onto the target platform. As IDCT is the most computationally-intensive block and contributes 56.6% to the total execution time, we should either run IDCT on a high-performance processor core or on more than one cores in parallel or implement it as a hardware co-processor, to achieve the desired performance. However, as mentioned, we considered only MPSoC architectures with a set of identical PowerPC cores, so we chose to parallelize IDCT on multiple cores. In contrast, some other blocks that are less computationally-intensive were grouped together. In this way, we can achieve load-balancing between the processor cores. An example of parallelization of the application model and its mapping onto four processor cores is illustrated in Figure 5.31.

Table 5.6: M-JPEG Decoder Performances of Different Configurations (Point-to-Point Communication with Untimed FIFOs)

Configuration	Frames/Sec	Simulation Duration
1: Three Cores @ 100 MHz	6.29	7.7 sec
2: Four Cores @ 100 MHz	12.58	7.4 sec
3: Five Cores @ 100 MHz	16.01	7.2 sec
4: Four Cores @ 200 MHz	25.16	7.5 sec
5: Five Cores @ 200MHz	32.03	7.4 sec

Then, each task was modeled at transaction level in SystemC, annotated with accurate timing information. Each processor was modeled as a SystemC module and each task corresponded to a thread in the processor module that the task was mapped to. After getting the computation model we can start to refine the communication. At the beginning of the communication refinement, we first used FIFOs to realize point-to-point communications among the processing elements, as shown in Figure 5.32. Using fast simulation at this high abstraction level, we can get a rapid validation of system's functionality after the partitioning and mapping, and at the same time, get the first idea of the system's performance. In Table 5.6, we show the performances of five selected design alternatives. The first three designs are architectures consisting of three, four, and five PowerPC cores, respectively, all running at 100 MHz. As shown, even the implementation on five cores achieves only the performance of 16.01 fps, far away from the desired throughput. By increasing the frequency of the processor cores from 100 MHz to 200 MHz, the performances of the implementations on four cores and five cores are increased to 25.16 fps and 32.03 fps, respectively, both higher than the requirement.

As the communication timing has not been taken into account, the performance data is not accurate enough to make the final choice. However, we know that the real performance can only be worse than the estimation, so we can for sure exclude the first three designs and consider only Design 4 and Design 5 that fulfill the performance requirement in the following design phases.

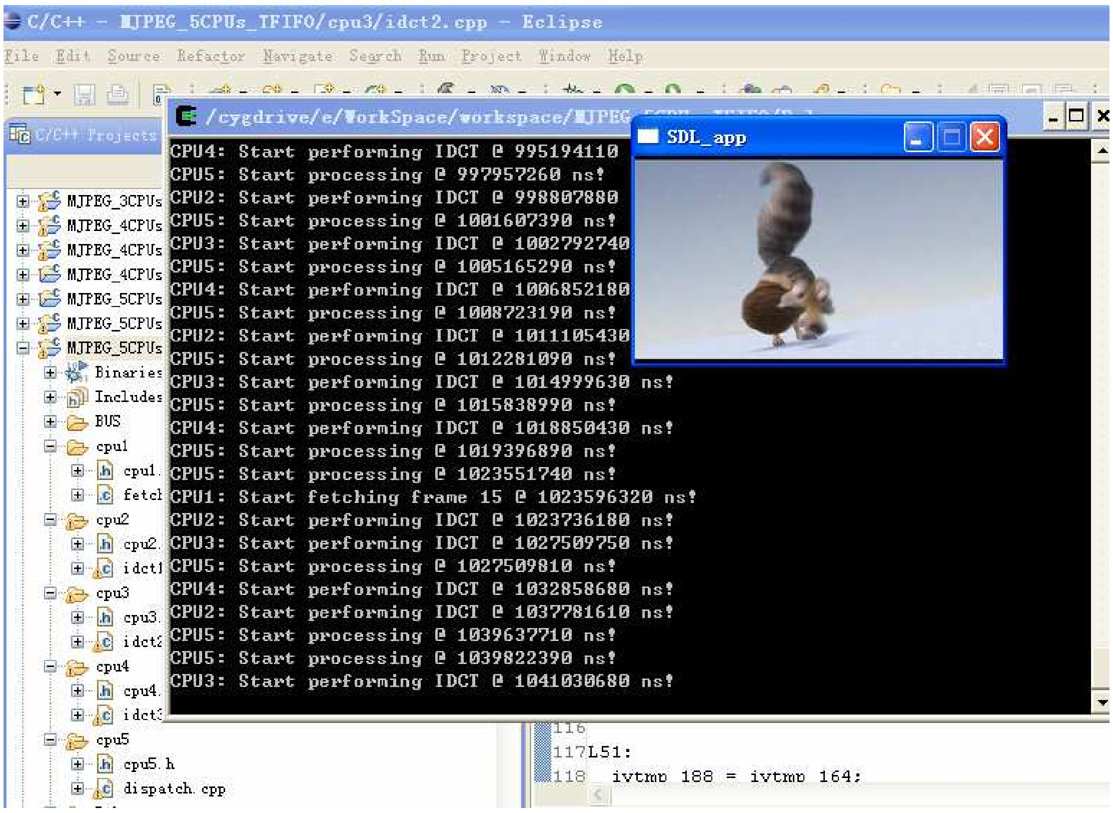
As shown in Table 5.6, the simulations at this level are very fast. For a sequence of 200 video frames, each simulation takes only around 7 seconds, as shown in Table 5.6. It takes only less than one minute to evaluate the five design alternatives and exclude three of them. Hence, the simulation at this abstraction level can narrow the design space significantly in a very short time.

Table 5.7: Communication Workload Profiling

From	To	Communicated data size (MB)
VIN	**CPU1**	0.91
CPU1	**CPU2**	42.19
CPU1	**CPU3**	42.19
CPU2	**CPU4**	10.55
CPU3	**CPU4**	10.55
CPU4	**VOUT**	21.10

In addition, we can analyze the communication workload among the processing elements. The obtained workload statistics can help in communication refinement, for example, to make the choice of an appropriate communication architecture and to optimize the communication protocol. Table 5.7 shows the communication workload obtained for the mapping shown in Figure 5.31. The most data transferred among the processing elements is in form of large data blocks. If we choose a conventional bus as the communication fabric, it is better to use a wide data bus that supports burst transfer. The burst length should be set as large as possible.

As the task models capture both functional and timing behaviors, during the simulation we can not only get the performance statistics but also see the decoded



```
CPU4: Start performing IDCT @ 995194110
CPU5: Start processing @ 997957260 ns!
CPU2: Start performing IDCT @ 998807880
CPU5: Start processing @ 1001607390 ns!
CPU3: Start performing IDCT @ 1002792740
CPU5: Start processing @ 1005165290 ns!
CPU4: Start performing IDCT @ 1006852180
CPU5: Start processing @ 1008723190 ns!
CPU2: Start performing IDCT @ 1011105430
CPU5: Start processing @ 1012281090 ns!
CPU3: Start performing IDCT @ 1014999630 ns!
CPU5: Start processing @ 1015838990 ns!
CPU4: Start performing IDCT @ 1018850430 ns!
CPU5: Start processing @ 1019396890 ns!
CPU5: Start processing @ 1023551740 ns!
CPU1: Start fetching frame 15 @ 1023596320 ns!
CPU2: Start performing IDCT @ 1023736180 ns!
CPU3: Start performing IDCT @ 1027509750 ns!
CPU5: Start processing @ 1027509810 ns!
CPU4: Start performing IDCT @ 1032858680 ns!
CPU2: Start performing IDCT @ 1037781610 ns!
CPU5: Start processing @ 1039637710 ns!
CPU5: Start processing @ 1039822390 ns!
CPU3: Start performing IDCT @ 1041030680 ns!
```

```
L51:
  ivtmp_188 = ivtmp_164;
```

Figure 5.33: Execution Environment

video frames to validate the system's functionality, as shown in Figure 5.33. After each step of communication refinement, it is necessary to verify whether the system's functionality is still correctly modeled.

Table 5.8: M-JPEG Decoder Performances of Different Configurations (Point-to-Point Communication with Timed FIFOs)

Configuration	Frames/Sec	Simulation Duration
1: Three Cores @ 100 MHz	5.98	17.7 sec
2: Four Cores @ 100 MHz	11.96	17.1 sec
3: Five Cores @ 100 MHz	14.91	16.9 sec
4: Four Cores @ 200 MHz	22.77	18.1 sec
5: Five Cores @ 200MHz	26.93	17.5 sec

In the next step, we annotated a delay for each inter-processor transfer of a buffer of data. For example, assume a bus is used and it does not support burst mode. Let the bus width be BUS_WIDTH bytes. To transfer a buffer of data with a size of n bytes, n/BUS_WIDTH bus transactions are needed. Let the time needed by the processor to set up a bus transaction be T_SETUP ns. Assume the transfer of each word consumes a constant time $T\_TRANSFER$ ns on the bus. Then,

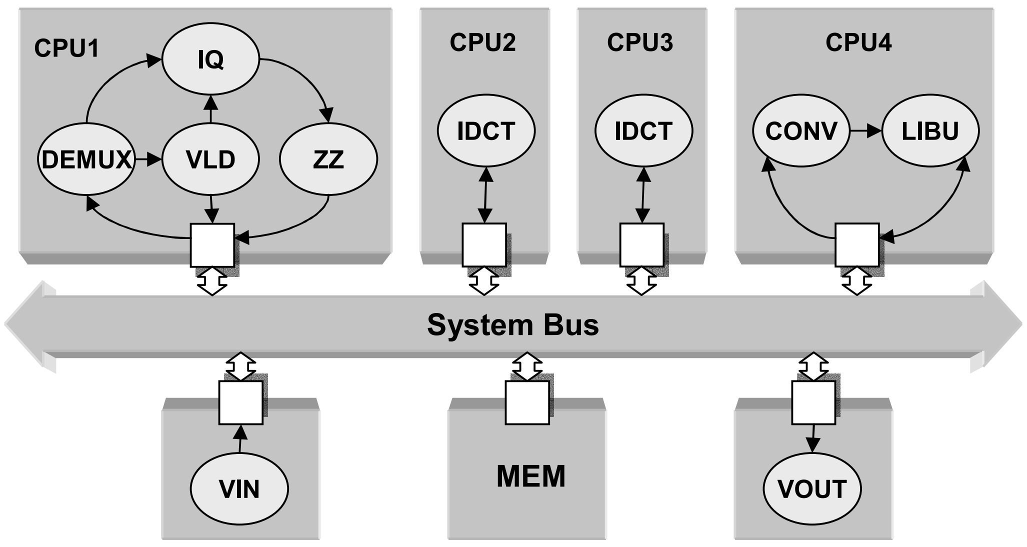


Figure 5.34: Communication Using a System Bus

we annotate "wait((n/BUS_WIDTH)*(T_TRANSFER+T_SETUP), SC_NS)" for a transfer of n bytes data.

With the simulation using timed FIFOs, we can test the parameters of the communication architecture, such as buffer size, bus width, and burst length. In the case study, a simple bus that is 32 bits wide and does not support burst mode was tested. As discussed above, in this step we need only to evaluate Design 4 and Design 5. Nevertheless, in order to get a complete comparison of different simulation levels, we still simulated all the design alternatives. The simulation results are shown in Table 5.8. We can see that when the communication timing is taken into account, the performances of all the designs are reduced. The performance of Design 4 is reduced from 25.16 fps to 22.77 fps and does not fulfill the requirement any more. As the communication was only approximately simulated and bus contentions have not been taken into account, the real performances should still be worse than the simulation results. Hence, only Design 5 needs to be considered in the next refinement step.

As shown in Table 5.8, delay annotations for data transfers bring a large overhead in simulation performance. For a sequence of 200 frames, the simulation duration increases from 7 seconds to 17.5 seconds on average. This simulation speed is still very fast.

At last, the timed FIFOs were replaced by a bus simulation model and a shared memory model. The processing elements are connected to the bus using the interface provided by the bus model, as shown in Figure 5.34. The bus model simulates each bus transaction accurately and takes the bus contentions into account. We first set the bus width to 32 bits. To get a complete comparison of the simulation levels, we simulated all the five designs. The simulation results are shown in Table 5.9. We can see that the estimated decoding performances are further reduced.

The estimated performance of Design 5 is reduced from 26.93 fps to 21.67 fps and does not fulfill the requirement either.

The simulation using the accurate bus model is very slow. For a sequence of 200 frames, each simulation takes around 22 minutes. Fortunately, at this design step, the design space has been narrowed down. We do not need to run too many such slow simulations.

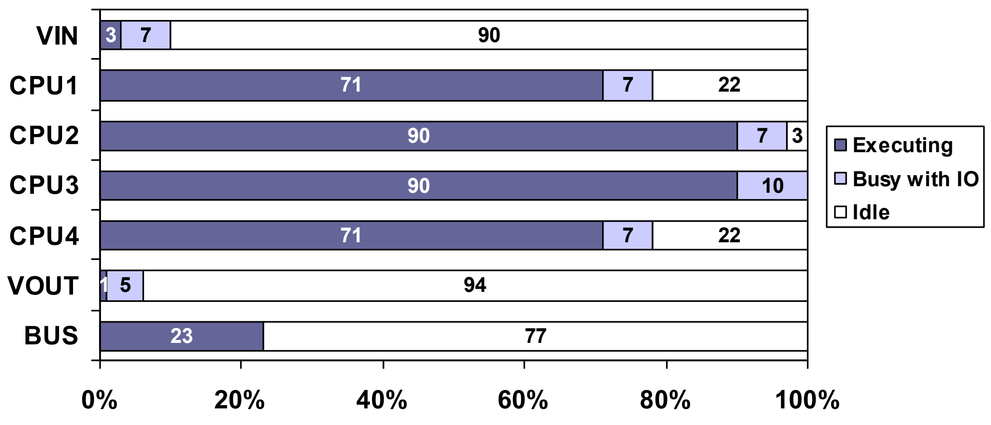
So far, we still do not get a design that fulfills the requirement, although the performances of Design 4 and Design 5 are close to the requirement. To further improve the designs, it is useful to get utilizations of system components to find the performance bottlenecks. Figure 5.35 and Figure 5.36 show the utilizations of system components of 4-CPU and 5-CPU architectures, respectively. If the decoder is realized on the architecture with 5 processor running at 200 MHz and a 32-bit bus, the utilizations of CPU 1, CPU 2, CPU 3, CPU 4, and CPU 5 are 99%, 70%, 71%, 71%, and 84%, respectively. CPU 1 is fully loaded, with 65% of time busy with execution and 34% of time busy with I/O. The bus utilization is 44%. As the processors spent a large portion of time being busy with IO, we decided to improve the bus. We first increase the bus width. As shown in Table 5.10, when the bus is widened to 64 bits, the performances of Design 4 and Design 5 are increased to 22.70 fps and 26.47 fps, respectively. Finally, we got a design that fulfills the requirement. We can also see that when the bus width is increased, there are less bus transactions to be simulated and thus the simulation performance is higher. The two simulations using the 64-bit bus model take 11 min 34 sec and 10 min, respectively, much less than the simulations using the 32-bit bus model.

Table 5.9: M-JPEG Decoder Performances of Different Configurations (Communication with 32-bit Bus)

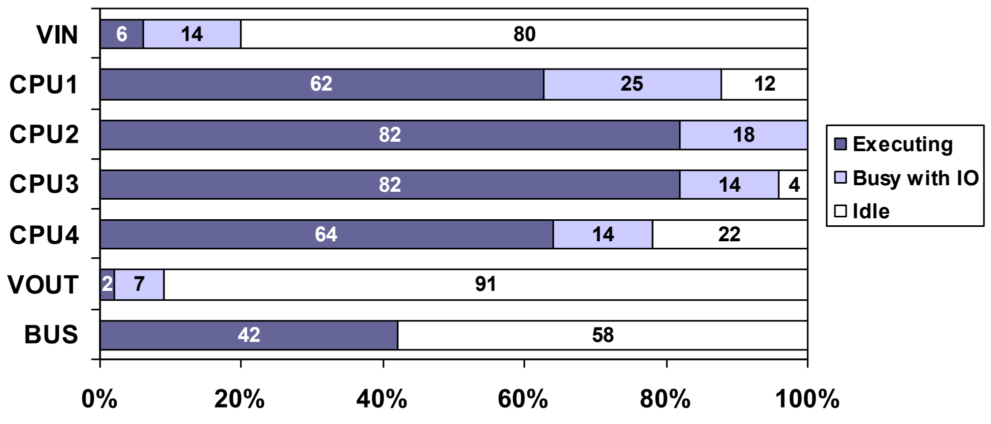
Configuration	Frames/Sec	Simulation Duration
1: Three Cores @ 100 MHz	5.70	18 min 36 sec
2: Four Cores @ 100 MHz	11.33	22 min 19 sec
3: Five Cores @ 100 MHz	13.37	22 min 35 sec
4: Four Cores @ 200 MHz	20.72	22 min 29 sec
5: Five Cores @ 200MHz	21.67	22 min 49 sec

Table 5.10: M-JPEG Decoder Performances of Different Configurations (Communication with 64-bit Bus)

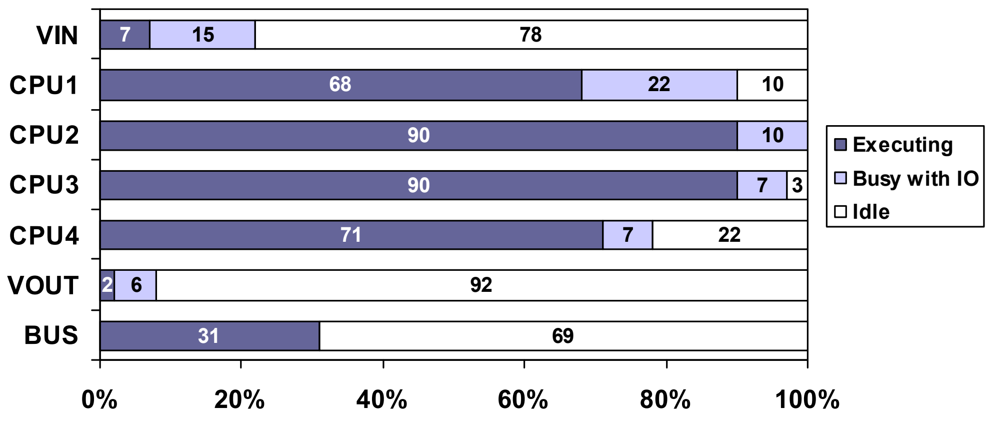
Configuration	Frames/Sec	Simulation Duration
4: Four Cores @ 200 MHz	22.70	11 min 34 sec
5: Five Cores @ 200 MHz	26.47	9 min 57 sec



(a) 4 CPU Cores Running at 100 MHz and 32 bit Data Bus

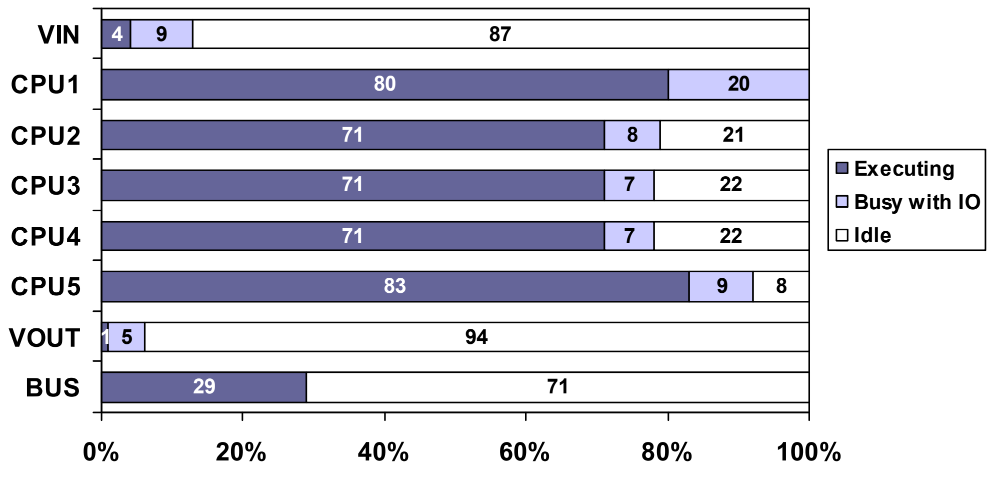


(b) 4 CPU Cores Running at 200 MHz and 32 bit Data Bus

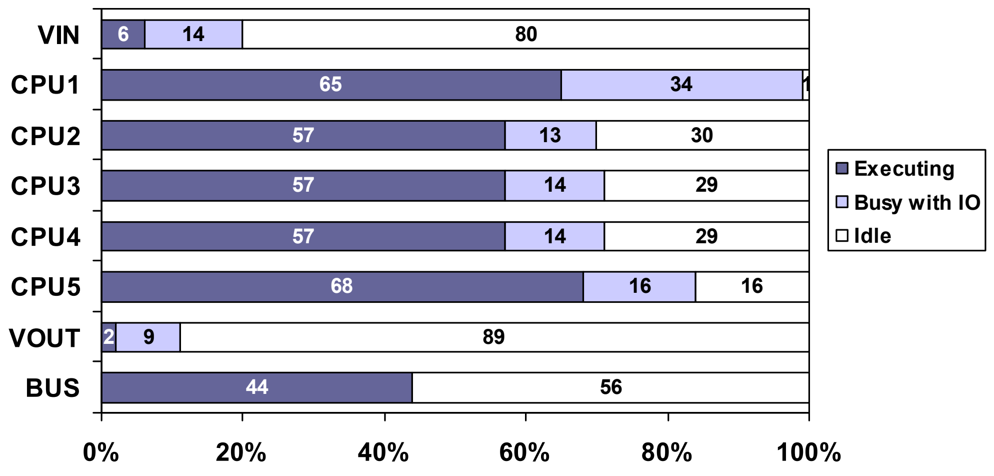


(c) 4 CPU Cores Running at 200 MHz and 64 bit Data Bus

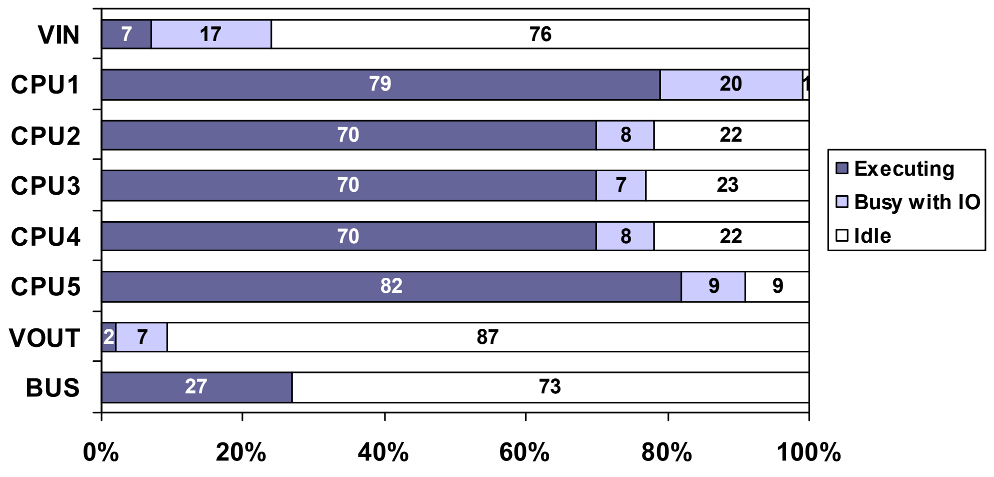
Figure 5.35: Utilizations of Processing Components of the 4-CPU Architecture



(a) 5 CPU Cores Running at 100 MHz and 32 bit Data Bus



(b) 5 CPU Cores Running at 200 MHz and 32 bit Data Bus



(c) 5 CPU Cores Running at 200 MHz and 64 bit Data Bus

Figure 5.36: Utilizations of Processing Components of the 5-CPU Architecture

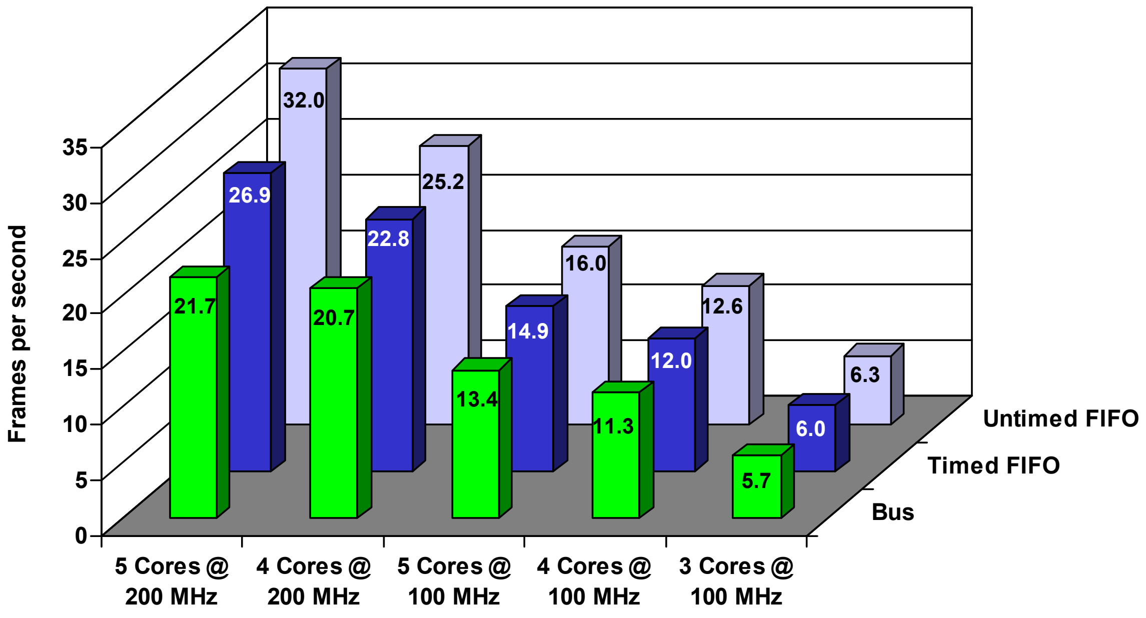


Figure 5.37: MJPEG-Decoder Performances Obtained by Simulations at Different Levels

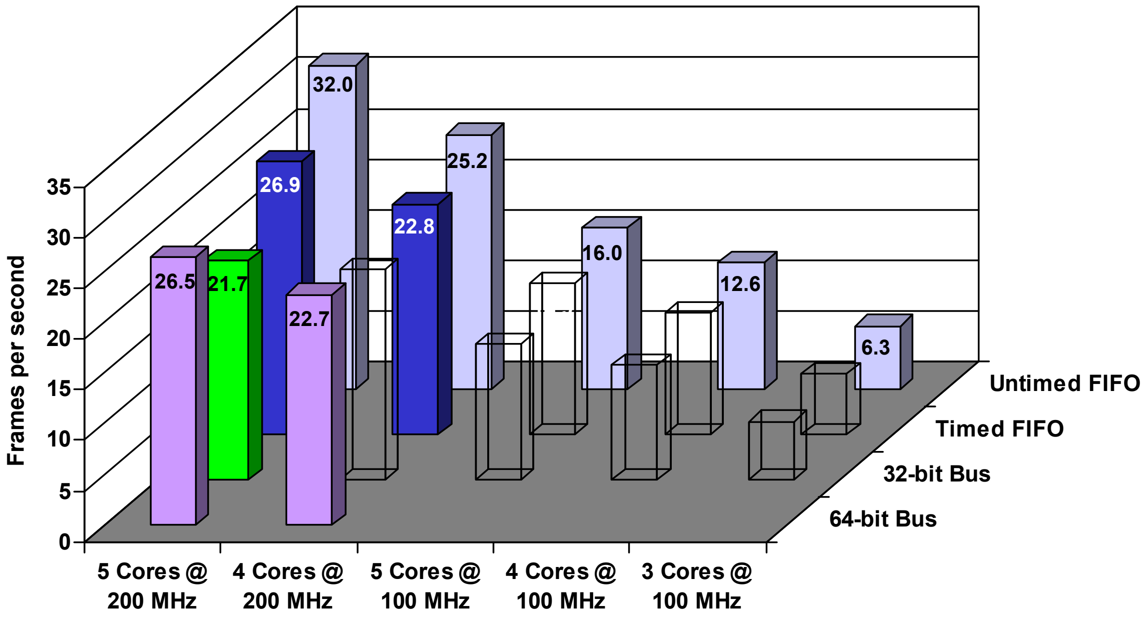


Figure 5.38: Necessary Simulations at Different Levels

Now, we make a short conclusion. This case study demonstrated how iSciSim is used to facilitate MPSoC design. Given a sequential implementation of an application, iSciSim can be used for performance profiling to get statistics for application model partitioning. After partitioning and mapping, we use iSciSim to generate software TLMs annotated with accurate timing information. The timed software TLMs allow for very fast software performance simulation. With stepwise refinement of the communication architecture, the software TLMs can be connected to the communication model without modification. In the case study, we presented three levels of communication model: FIFOs, timed FIFOs and a timing-accurate bus model. We selected five design alternatives to present the process of system refinement. We evaluated all the five designs using the three simulation levels. Figure 5.37 gives an overview of the simulation results. With FIFOs realizing point-to-point communications among the processors, communication timing cannot be taken into account. We got very optimistic estimates. When each data transfer is annotated with a delay, we can get more accurate estimates. However, as resource contentions cannot be modeled, the estimates are still optimistic. Simulations at these two abstraction levels are very fast. When a timing-accurate bus model was used, we got very accurate statistics but the simulation speed was slowed down significantly. Although we simulated all the design alternatives at different levels to get a complete comparison, actually only part of them is necessary in the real design. With the fast simulations at the highest abstraction level, we can already narrow the design space and leave only two designs still under consideration: architectures consisting of 4 and 5 cores running at 200 MHz. Next, the simulations at a lower level can provide more accurate performance data, with which it is found that only 5-core system can fulfill the performance requirement. Then, we need to simulate only the 5-core system with the accurate bus model and find that the decoder performance is still under the requirement when the communication overhead is accurately taken into account. As the decoder performance is close to the requirement, we improve the communication performance by increasing the width of the data bus from 32 bits to 64 bits. With two more simulations we can obtain the system that fulfills the requirement: 5 PowerPC cores running at 200 MHz connected by a 64-bit bus. All the simulations performed are illustrated in Figure 5.38, with the unnecessary ones shown with transparent columns.

Chapter 6

Multi-Task Simulation in SystemC with an Abstract RTOS Model

In many embedded systems, especially in real time systems, a complex application is splitted into tasks, each responsible for a portion of the work. The tasks are executed according to their real-time requirements. The sequence of task execution on a single processor is managed by a real time operating system (RTOS). Different scheduling policies of the RTOS may result in very different timing behaviors of the system. Therefore, it is desired that the scheduling behavior can also be captured during system-level design space exploration. Since a particular RTOS may not yet be decided at early design stages, an abstract RTOS model that provides common RTOS services, such as scheduling, synchronization and interrupt handling, and captures the most important RTOS temporal behaviors, such as context switching time and scheduling latency, could be very useful.

In this chapter, we describe the way of modeling an RTOS at the system level with SystemC and present how to use the abstract RTOS model to schedule the timed task models generated by iSciSim. Section 6.1 shows the necessity of modeling an RTOS at system level. Then, in Section 6.2, the common services and temporal behaviors of RTOSs are introduced. An implementation of RTOS model and preemption modeling are described in detail in Section 6.3 and 6.4, respectively. Section 6.5 presents a more modular design of RTOS model. Finally, we show some experimental results in Section 6.6.

6.1 Unscheduled Execution of Task Models

As already discussed before, a design in a typical system-level design process starts with a specification model. Then, the partitioning and mapping step decides how to implement the computation and communication, captured in the specification model, on a set of processing elements (PEs) connected with a bus or a network. After partitioning, the system is represented as a set of concurrent, communicating tasks, each consisting of a piece of code and realizing a part of the system's functionality.

During mapping, more than one task may be allocated onto a single PE. We model a PE with a SystemC module and the transaction level task models (taskTLMs) mapped to the PE are declared as SystemC threads. The relation among the taskTLMs, the SystemC kernel and the simulation host is shown in Figure 6.1. As the taskTLMs are not scheduled, the independent tasks can run fully concurrently.

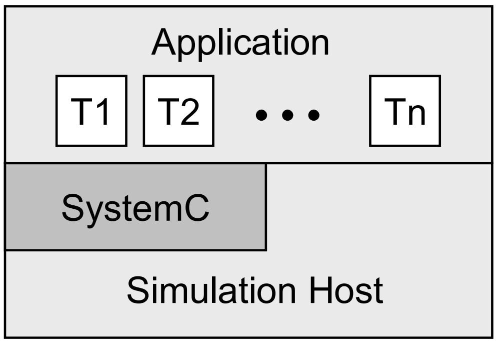


Figure 6.1: Layers of the Simulation Setup

In Figure 6.2, we use an example of a priority based scheduling to motivate scheduling simulation with an RTOS model. First, the real execution trace is shown in Figure 6.2(a). The example involves two tasks running on a target processor. Task2 has a higher priority than Task1. Both tasks access a global variable *gv*. Task1 starts its execution at 0 us. It writes *value1* to the global variable at 8 us. At 14 us, Task2 is activated and preempts Task1. It reads *value1* from *gv* and then executes until blocked at 22 us by waiting for some I/O data. At the same time, Task1 is resumed. It writes *value2* to *gv* at 26 us. At 38 us, Task2 gets the required I/O data and is activated again. At the same time, Task1 is preempted again. Task2 gets *value2* from *gv* at 46 us. Task1 is kept in the ready state until Task2 finishes its current cycle at 50 us. Then Task1 resumes and terminates at 58 us. In the real execution, each external interrupt will trigger an interrupt service routine (ISR). The ISR will then activate a ready task. For clarity, the execution of ISRs is not considered in the example.

Figure 6.2(b) illustrates the simulation trace of the unscheduled taskTLMs. As in a taskTLM a *wait()* statement is generated only before an access to the global variable, the taskTLMs of Task1 and Task2 both are divided into three execution segments. For more information about segmental execution of taskTLM, please refer to Section 5.5. We can see, taskTLM1 and taskTLM2 execute truly in parallel and their simulated delays fully overlap. Both taskTLMs run to the end without being interrupted. The total simulated delay is only 38 us. Compared with the real delay 58 us, the error is large. Even worse, the order of the global variable accesses is also different from the real execution. This might lead to wrong functional results. In the example, taskTLM2 gets a wrong value of *gv* in the first read. It should get *value1* but it gets *value2*.

The above example shows the importance of modeling scheduling behaviors in the system level design space exploration.

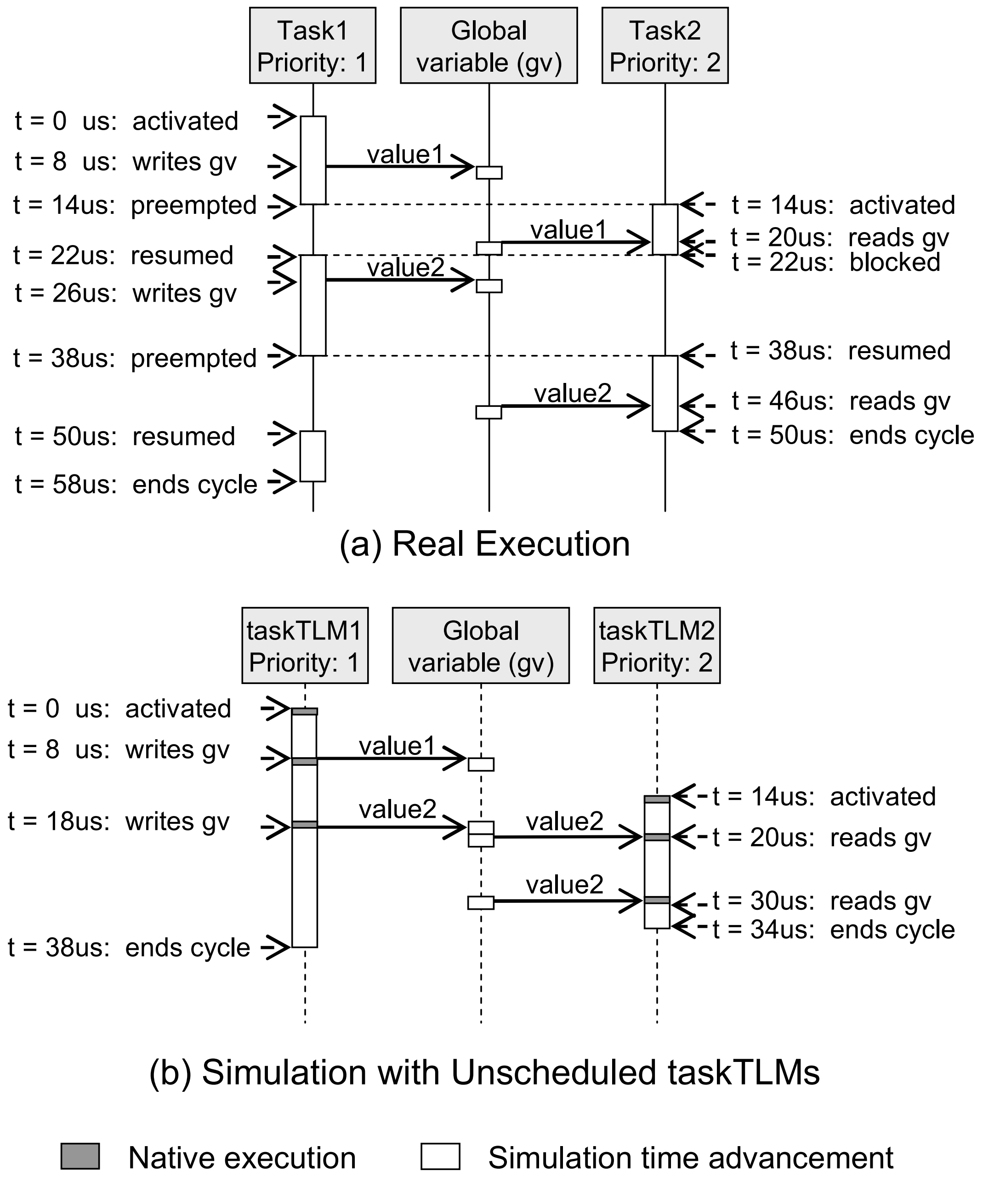


Figure 6.2: An Example of Multi-Task Execution

6.2 The RTOS's Functionality

An RTOS performs basically two primary functions, (1) providing a consistent and convenient interface to the application software and the resources, and (2) scheduling the software tasks and managing the hardware resources. At system level we are not interested in the exact functionality at the implementation level, but rather how its scheduling policies, management schemes, and timing parameters etc. affect the system performance. In this context, we introduce some basic information on task and scheduler of a typical RTOS in this section.

6.2.1 Task

Typically, there are several tasks competing for the processor at the same time, but the processor can only execute one task at a time. To solve this problem, each task is modeled as a state machine. The scheduler organizes the task execution by means of task state transitions. The most basic task state model consists of three states, as shown in Figure 6.3:

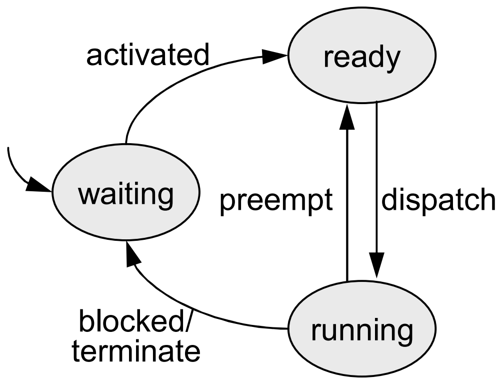


Figure 6.3: 3-State Task Model

- ready: the task in this state is already runnable. However, it must wait for allocation of the processor, because the processor is still occupied by another task.
- running: the task in this state is actually using the processor. As the processor can execute one task at any time, there is only one task in the *running* state.
- waiting: the task in this state is unable to run until some external event happens

The initial state of a task is *waiting*. After a task is created, there are four transitions possible among the three states. The transition from *ready* to *running* occurs when the task is selected by the scheduler to execute next. Note that only *ready* tasks can transit into the *running* state. There is no direct transition from *waiting* to *running*. There may be more than one *ready* task. It's the duty of the scheduler to decide which *ready* task can get control of the processor next. Typically, the ready task with the highest priority is selected. In contrast, the transition from *running* to *ready* occurs, when the running task is preempted by a higher priority task.

The transition from *running* to *waiting* occurs, when the running task terminates normally or it is blocked by waiting for an external event (e.g., waiting for an I/O operation to complete, a shared resource to be available, or time to expire etc.). In contrast, when the external event for which a task was waiting happens, the task transits from *waiting* to *ready*.

In specific designs, a task may have more states. For example, in the OSEK OS [10], the *waiting* state introduced above is subdivided into two states, *waiting* and *suspended*, as shown in Figure 6.4. An *extended task* of the OSEK OS transits from *running* to *waiting* when being blocked by waiting for an event. It transits from *running* to *suspended* when it terminates normally. A task of μC/OS-II [67] has five states. It has the *ISR* state and the *dormant* state in addition to the three basic states.

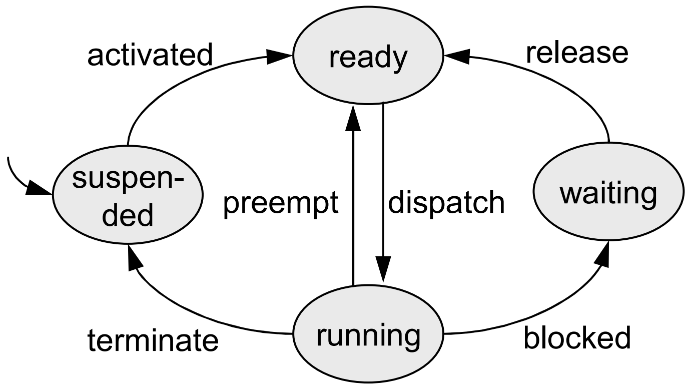


Figure 6.4: 4-State Task Model of the OSEK OS

When a task is created, it is assigned a *task context block* (TCB). A TCB is a data structure containing the information needed to manage a task. This information includes typically the task id, the task priority, hardware register values, and the task address space etc. When a running task is preempted, the RTOS first stops the execution of the running task and saves its context (the values in hardware registers) to its TCB. Then, the RTOS updates the hardware registers with the context of the new running task from its TCB. This process is called a *context switch*. A context switch is necessary to make a preempted task can resume execution exactly where it is interrupted.

6.2.2 Scheduler

As mentioned above, the scheduler is responsible for determining which ready task will run next. This decision is made according to a specific scheduling algorithm. There are several scheduling algorithms, including *round-robin scheduling*, *fixed priority based scheduling*, and *earliest deadline first approach* etc. Among these algorithms, fixed priority based scheduling is the most widely used one. A scheduling algorithm can be either non-preemptive or preemptive.

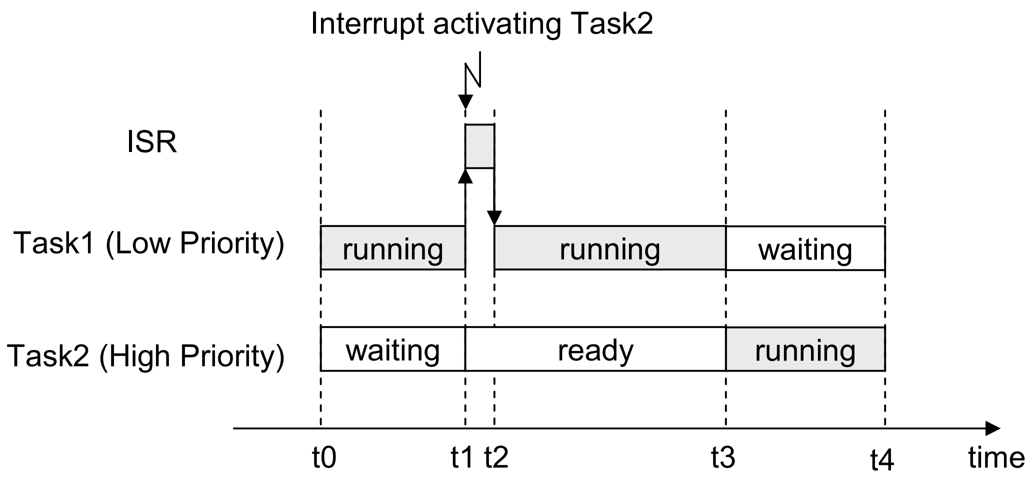


Figure 6.5: Non-Preemptive Scheduling

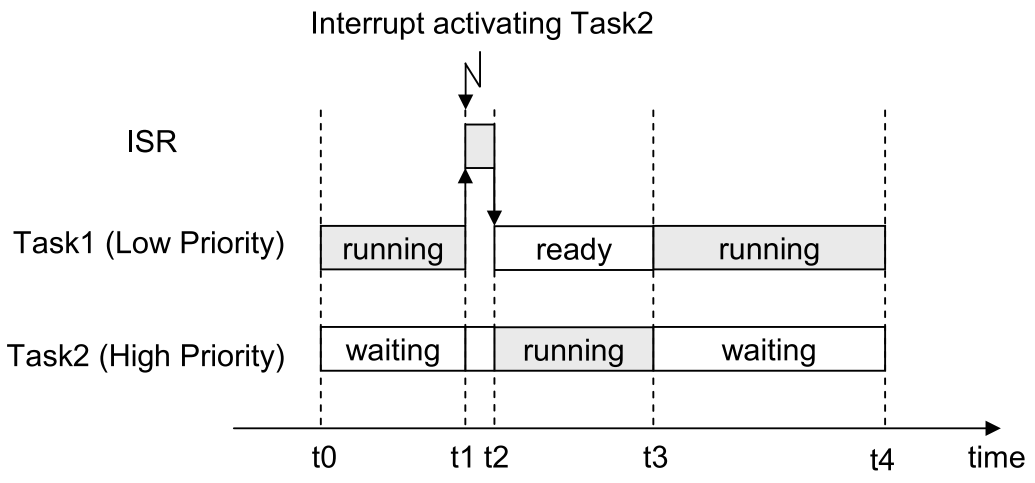


Figure 6.6: Preemptive Scheduling

Using non-preemptive scheduling, a running task can not be interrupted until it terminates, unless it explicitly gives up control of the processor. A simple priority-based, non-preemptive scheduling scenario is illustrated in Figure 6.5. Task1 is first running but gets interrupted at *t1* by an event, which the higher priority task Task2 is waiting for. The processor jumps to an ISR (interrupt service routine) to handle the event. The ISR makes Task2 ready to run. At the completion of the

ISR, the processor returns to Task1. Task1 resumes at the instruction following the interrupted point. It gives up control of the processor when terminating normally at *t3*. Then, Task2 gets control of the processor to the event signaled by the ISR.

The main drawback of non-preemptive scheduling is its low responsiveness. A higher priority task that has been made ready has still to wait a long time for a lower priority task to finish. It also makes the response time of a safety-critical application non-deterministic, because you never really know when the highest priority task will get control of the processor. Due to this drawback of non-preemptive scheduling, most existing RTOSs use preemptive scheduling.

In contrast to non-preemptive scheduling, when preemptive scheduling is used, the ready task with the highest priority can always get control of the processor. Therefore, with preemptive scheduling, execution of the highest priority task is deterministic. Figure 6.6 shows an execution scenario of preemptive scheduling. As illustrated, with preemptive scheduling, the high priority task Task2 is made ready to run and then immediately gets control of the processor at the completion of the ISR, while the interrupted low priority task Task1 is suspended and transits into the *ready* state. Task1 resumes, after Task2 terminates.

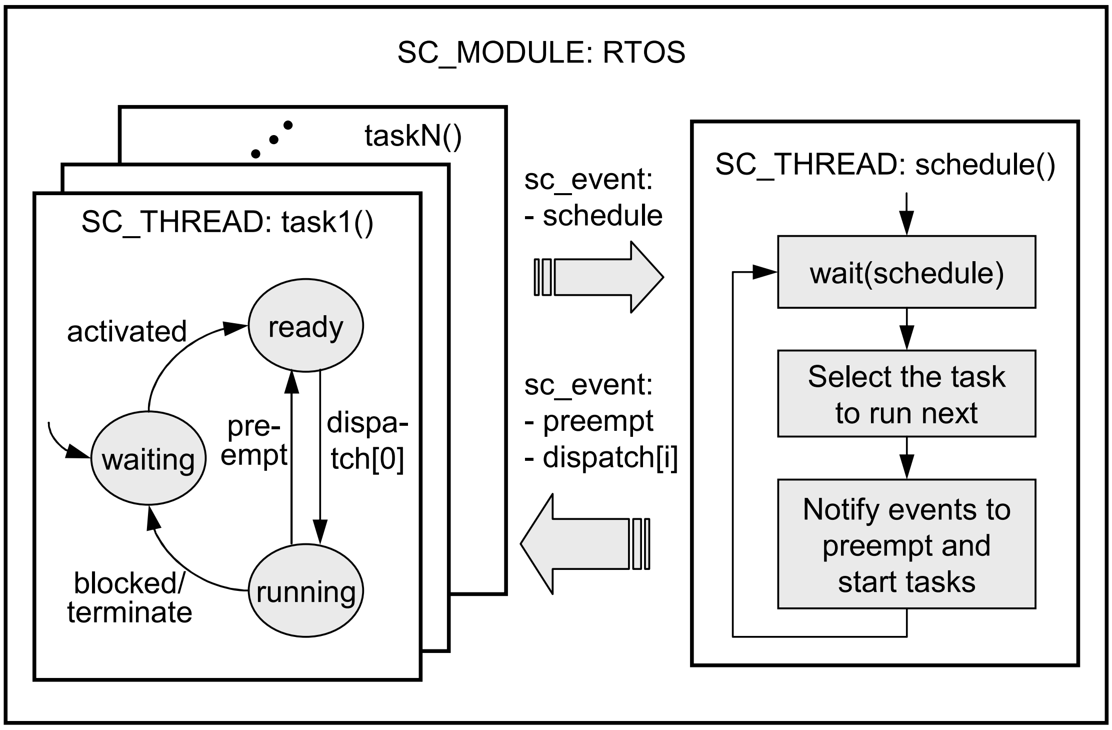


Figure 6.7: Implementing the RTOS Model and Task Models in a Single Module

6.3 The RTOS Model in SystemC

In our first implementation, both tasks and the scheduler are modeled as SystemC threads in a single module. The synchronization between the tasks and the scheduler is realized by SystemC events. This implementation is depicted in Figure 6.7. Before a particular RTOS has been decided, we use the most general task state model with three basic states. When an RTOS with a specific state model is chosen later, the state model used in simulation can be easily extended. As shown, the task state transitions are triggered by events from the scheduler. To manage the execution of the tasks with the three basic states, the scheduler needs only the *preempt* event and a *dispatch* event for each task. All the *dispatch* events are put in an array. Each task is assigned an unique id, $id \in \mathbb{Z}\ and\ 0 \leq id < N$, where N is the number of tasks allocated to the processor. The scheduler notifies the event *dispatch[i]* to dispatch the corresponding task, which has $id = i$. There is only one running task, so the scheduler needs only one *preempt* event to interrupt the running task.

The scheduler performs a scheduling when there is a new task gets ready or the running task terminates. Hence, the scheduler process is triggered by the *schedule* event notified by a task at its beginning and completion. With the RTOS model refined, more events can be added to handle more complex synchronizations.

The following code gives a basic idea of the RTOS module definition.

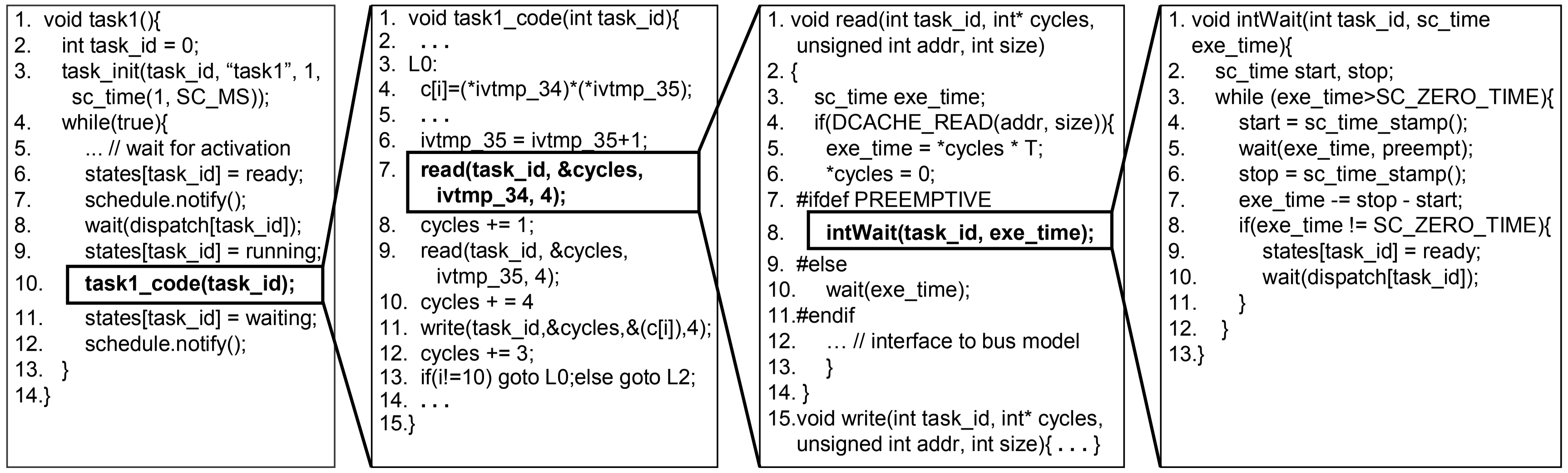
```
enum TASK_STATE{waiting, ready, running};

SC_MODULE(cpu_model){
private:
  int *priorities;
  TASK_STATE *states;
  sc_event preempt, schedule, *disptach;
  ... // other variables

public:
  int task_number;
  void scheduler(){ ... }
  void task_init(...){ ... }
  void task1(){ ... }
  void task2(){ ... }
  ... // other functions

  SC_CTOR(cpu_model){
    task_number = 2;
    priorities = new int[task_number];
    states = new TASK_STATE[task_number];
22.     dispatch = new sc_event[task_number];
23.     SC_THREAD(task1);
24.     SC_THREAD(task2);
25.     SC_THREAD(scheduler);
26.     ... // other thread declarations
27.   }
28. };
```

In a real RTOS, TCBs are usually implemented as a linked list. In our RTOS model, we do not simulate software at the assembler level, so all the register operations are abstracted away. Therefore, there is no need to copy hardware register values during a context switch. Some other information in a TCB such as task address space is also not required during the abstract RTOS simulation. Only the information like task id, task state, and priority needs to be recorded. To increase simulation performance, we do not construct a TCB list as a real RTOS does, but store the necessary information in variables or arrays. As shown in the above code, we declare two arrays *priorities* and *states* to store the priorities and states of all the tasks (line 5 and 6), respectively. Each task accesses its own priority and state using its id as the index.

The events used for task management are declared in line 7. The size of the arrays priorities, states, and dispatch is equal to the number of tasks. These arrays are instantiated in the constructor of the module (line 20 through line 22). Thus, we need only to change the value of task_number when the number of tasks changes. All the tasks and the scheduler are declared as SystemC threads (line 23 through line 25). In the shown code, there are two tasks declared. The task state array is declared as an enumerated type TASK_STATE, the elements of which are the task states (line 1).

6.3.1 Task Model

According to the discussion in Section 6.2, a task realizes a portion of the application's work and must have the notion of states to compete the processor with other tasks. Therefore, a task model contains three parts of code: behavioral code, timing annotations, and the code that realizes state transitions and synchronizations. An example of task model is shown in Figure 6.8(a) and gives a basic idea of task model definition. As already discussed, a task model is implemented as a *SC_THREAD*. At the beginning of the thread, the task is assigned an id (line 2) and calls the function *task_init(int task_id, char* task_name, int priority, sc_time deadline)* (line 3), which initializes the task's context (the parameters such as task name, priority and deadline) and sets the task the *waiting* state.



```
void task1(){
   int task_id = 0;
   task_init(task_id, “task1”, 1,
     sc_time(1, SC_MS));
   while(true){
      ... // wait for activation
      states[task_id] = ready;
      schedule.notify();
      wait(dispatch[task_id]);
      states[task_id] = running;
      task1_code(task_id);
      states[task_id] = waiting;
      schedule.notify();
   }
}
```

(a) Task Model

```
void task1_code(int task_id){
  . . .
L0:
  c[i]=(*ivtmp_34)*(*ivtmp_35);
  . . .
  ivtmp_35 = ivtmp_35+1;
  read(task_id, &cycles,
    ivtmp_34, 4);
  cycles += 1;
  read(task_id, &cycles,
    ivtmp_35, 4);
  cycles + = 4
  write(task_id,&cycles,&(c[i]),4);
  cycles += 3;
  if(i!=10) goto L0;else goto L2;
  . . .
}
```

(b) Instrumented ISC

```
void read(int task_id, int* cycles,
  unsigned int addr, int size)
{
   sc_time exe_time;
   if(DCACHE_READ(addr, size)){
     exe_time = *cycles * T;
     *cycles = 0;
#ifdef PREEMPTIVE
     intWait(task_id, exe_time);
#else
     wait(exe_time);
#endif
     … // interface to bus model
   }
}
void write(int task_id, int* cycles,
  unsigned int addr, int size){ . . . }
```

(c) Read/Write Wrapper

```
void intWait(int task_id, sc_time
  exe_time){
   sc_time start, stop;
   while (exe_time>SC_ZERO_TIME){
      start = sc_time_stamp();
      wait(exe_time, preempt);
      stop = sc_time_stamp();
      exe_time -= stop - start;
      if(exe_time != SC_ZERO_TIME){
         states[task_id] = ready;
         wait(dispatch[task_id]);
      }
   }
}
```

(d) Interruptible Wait

Figure 6.8: Task Model and Preemptive Wait

The task body is an infinite loop. As the task is in the *waiting* state, it is unable to run until being activated by some external event (line 5). When the event for which the task is waiting occurs, the task transits from *waiting* to *ready* (line 6). Then, the task notifies the *schedule* event to trigger a rescheduling (line 7) and waits for its turn of execution (line 8). Because the task has the id = 0, it transits into the *running* state and gets control of the processor when it receives the event *dispatch[0]* from the scheduler (line 9). Then, it calls the function *task1_code()* to execute the instrumented ISC generated by iSciSim (line 10). When the task terminates normally, it transits into the *waiting* state again (line 11) and requests the scheduler to determine the task to run next (line 12). Then, the task waits for the next activation.

Figure 6.8(b) shows an example of instrumented ISC. It is generated by iSciSim, given the source code of the task. The instrumented ISC is integrated into the task model without any change, by means of a function call. In the shown code, there are no explicit I/O accesses. The only accesses to external modules happen, when memory accesses due to *load/store* instructions are missed in the cache. During ISC instrumentation, we annotate calls to read/write wrapper functions: *read(int task_id, int* cycles, unsigned int addr, int size)* and *write(int task_id, int* cycles, unsigned int addr, int size)*, instead of direct calls to the cache simulator. The two functions are responsible to advance the SystemC simulation time and realize interruptible waits.

Figure 6.8(c) shows the implementation of the *read/write* wrapper functions. In the *read* function, the values of *addr* and *size* are forwarded to the cache simulator for data cache simulation by calling *DCACHE_READ(unsigned int addr, int size)* (line 4). When a cache miss occurs, the function returns the *true* value. In this case, the aggregated cycle value is converted to *sc_time* by multiplying the cycle value with *T*, the duration of one CPU cycle (line 5). The cycle counter is cleared for cycle counting of the next execution segment (line 6). When non-preemptive scheduling is used, this execution segment can run to the end, so we just use a standard SystemC *wait* statement to advance the simulation time (line 10). In the case of preemptive scheduling, a function realizing an *interruptible wait intWait()* is called instead (line 8). The preemption modeling will be discussed in separate in Section 6.4. Details about *interruptible wait* are also described there. At the completion of the execution segment, a memory access through communication architecture is simulated (line 12). The code of *write()* is similar to *read()* and is not shown.

6.3.2 Scheduler Model

The scheduler thread waits initially for the event schedule from the task threads. Once it receives the notification of the event, it starts to select the task to run next according a specific scheduling policy. If a priority based scheduling policy is used, the task with the highest priority will be chosen from the ready tasks. If the selected task has a higher priority than the running task, the event preempt is notified to interrupt the running task. At the same time, the event dispatch[task_id] is sent out to start this selected task. Then, the scheduler thread is blocked by waiting for the next notification of the event schedule.

6.3.3 Timing Parameters

During the RTOS simulation, it is important to take the RTOS overhead into account. Therefore, the abstract RTOS model should capture the time required by the RTOS to perform typical services, such as task scheduling, context switching, task synchronization, and interrupt handling. The following timing parameters are considered as most important:

- The scheduling duration: the time required by the RTOS to select from the ready queue the task to run next. This overhead depends on both the scheduling algorithm and the number of ready tasks.
- The context switching duration: the time required by the RTOS to save the context of the running task and then update the processor registers with the context of the new running task from its TCB.
- The interrupt handling duration: the time spent by the RTOS to handle an interrupt.

The value of these timing parameters depends on the type of RTOS and the processor running the RTOS. To get these values, we can use the RTOS characteristics provided by the vendor. Another option is to measure the RTOS overhead on the target processor or on a cycle-accurate processor simulator, as done in [54]. However, this requires both the RTOS and the target processor or a cycle-accurate processor simulator available. This is often not the case in an early design phase. In this case, reasonable assumptions can be made according the designer's experience. For example, we can assign each RTOS operation with a reasonable delay. This loosely-timed RTOS model can give a first idea how the RTOS overhead influences the system performance. When the model is refined, the timing parameters can be updated with more accurate timing values.

6.4 Preemption Modeling

As most RTOS schedulers support preemptive scheduling, it is very important to get an efficient method for preemption modeling in abstract RTOS simulation. However, the simulation method introduced so far does not support preemption modeling. That is, the simulation time advancement of an execution segment

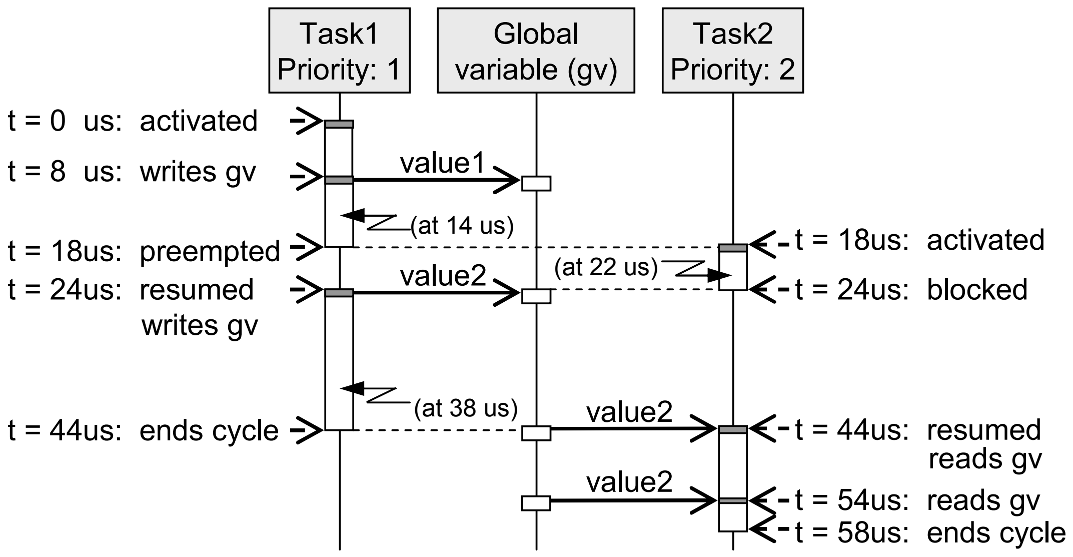


Figure 6.9: Example of Preemption Modeling

cannot be interrupted. Figure 6.9 shows the simulation trace of simulating the example in Figure 6.2(a) using the simulation method introduced so far. The execution order of the two tasks is scheduled. However, the simulation trace is not exactly the same as the real execution, because the preemptions are not correctly handled. As we can see, an interrupt occurs at 14 us, while a *wait()* is running. This interrupt cannot be handled until the *wait()* call returns. As a result, although the activation event of Task2 occurs at 14 us, Task2 can actually execute until Task1 finishes the second execution segment at 18 us. During the first execution segment of Task2, at 22 us, the event that blocks Task2 occurs. This event is reacted after Task2 finishes its first execution segment at 24 us. As soon as Task1 is resumed, it writes *value2* to *gv*. The event that unblocks Task2 occurs at 38 us, during the third execution segment of Task1. In the same manner, only at 44 us, this event can be reacted and Task2 is actually resumed. The second and third segments of Task2 are then executed one after another without breaking. Task2 gets *value2* in the both reads of *gv* at 44 us and 58 us.

Modeled in this way, both the order of execution segments and the order of global variable accesses are different from the real execution shown in Figure 6.2(a). Due to the wrong order of global variable accesses, Task2 gets a wrong value of *gv* in the first read. This wrong value can lead to different execution times of the following execution segments, although this effect is not depicted in the figure. Even worse, the functional results will be wrong. As a result, in the simulation the whole system behaves totally different from the real implementation. Such simulation results will mislead the design.

If such interrupts are predictable at the design time, a solution is to split the *wait()* call of an execution segment, during which an interrupt will occur, into two *wait()* calls. The first *wait()* advances the simulation time to the time point when the interrupt occurs. After the interrupt is reacted, the second *wait()* advances the rest simulation time of the execution segment.

However, most interrupts from the environment are not predictable. In this case, several other solutions have been proposed in previous work including *time-slicing method* [80] and *result oriented method* [89]. In the following subsections, we will introduce the two solutions. After that, the method we use for preemption modeling is introduced.

6.4.1 Static Time-Slicing Method

The time-slicing method (TSM) is proposed in [80]. The idea of TSM is to split one *wait()* statement in each execution segment into several *wait()* statements, each advancing a smaller interval of simulation time. Thus, there are more time points to detect interrupts. This idea is depicted in Figure 6.10. We can see, as the simulation time intervals are shortened, the interrupt can be earlier detected and reacted, and as a result, the simulation error is smaller. However, as the simulation time intervals are refined, more *wait()* statements are needed. As discussed before, a *wait()* can cause a very time-consuming context switch between the thread and the SystemC simulation kernel and thus introduces a very large simulation overhead.

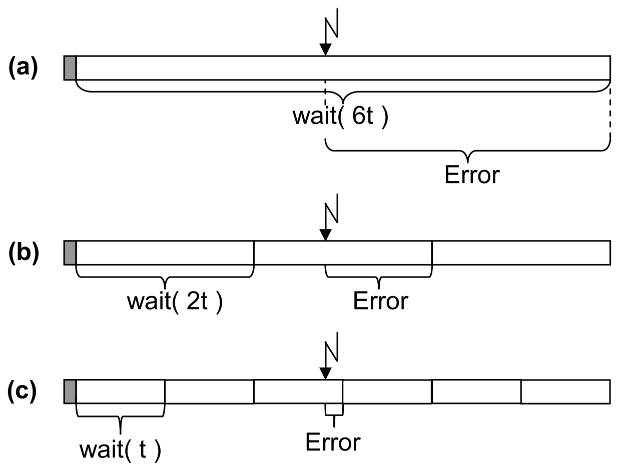


Figure 6.10: Time-Slicing Method for Preemption Modeling

Another large disadvantage of this method is that the possibility of wrong functional result still exists. In some cases, even a very small simulation error can lead to wrong functional results. For example, if two tasks communicate with each other using global variables, a mall timing error may change the access order of a global variable and thus lead to functional errors.

Figure 6.11 shows the simulation trace of simulating the example in Figure 6.2(a) using TSM. As shown, if the time interval is reduced to 2 us, all the three interrupts

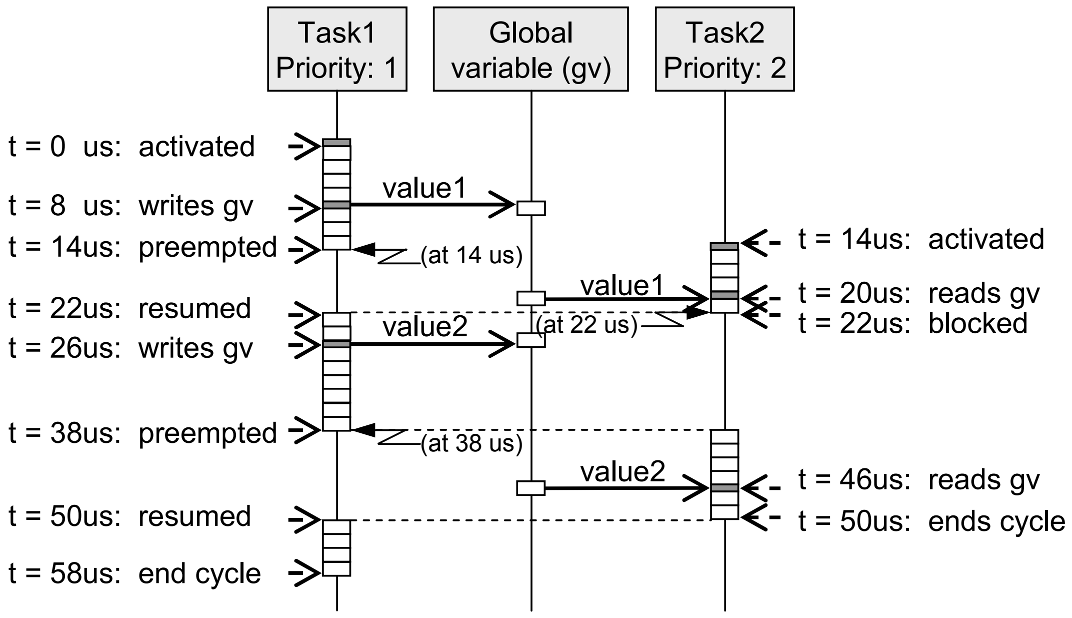


Figure 6.11: Example of Preemption Modeling Using TSM

can be detected and reacted in time without any error. Modeled in this way, the number of *wait()* statements used in the whole execution is increased from 6 to 29. This reduces the simulation performance significantly. Furthermore, in the simulation of a real system, there are much more interrupts and each interrupt can occur at any random time. It's very hard to find a time interval value to guarantee that all the interrupts can be handled in time. The only way to eliminate the error is to refine the simulation time intervals to the cycle level, i.e., to use one *wait()* to advance the simulation time for one CPU cycle. Using so many *wait()* statements, the software simulation performance will be reduced to an unacceptable level.

To summarize, TSM reduces the simulation error caused by synchronizations at the expense of simulation performance. As the simulation error cannot be eliminated, it is only suitable for the cases where small synchronization errors will not lead to serious consequences like functional errors.

6.4.2 Result Oriented Method

The result oriented method (ROM) is introduced by Gunar Schirner in [89]. ROM is a general modeling concept that achieves simulation optimization by hiding intermediate states of the system execution and focusing only on the final simulation outcome. In [89] this modeling concept is applied for preemption modeling in an abstract RTOS model. This solution is very simple but allows for accurate preemption modeling.

The concept of ROM is depicted in Figure 6.12. In the example, there are two tasks. Task1 with a lower priority is activated first. During its execution, an

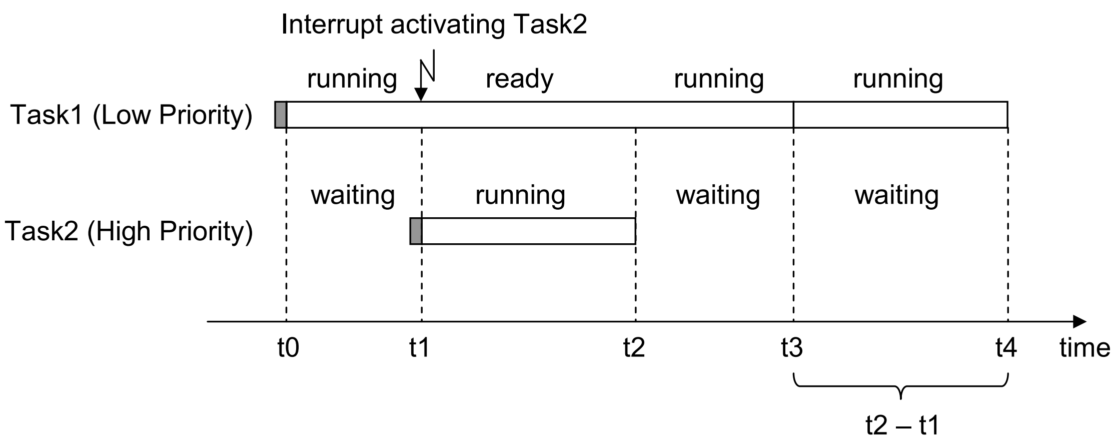


Figure 6.12: Result Oriented Method for Preemption Modeling

interrupt that activates Task2 with a higher priority occurs at *t1*. Although the interrupt does not break the execution segment of Task1, it changes the state of Task1 from *running* to *ready* and records the time point when the interrupt occurs. Once the state of Task1 changes, the scheduler is triggered and selects the next running task. In the example, Task2 is activated. When Task2 finishes the execution, the time point *t2* is also recorded. After that, the state of Task1 changes back to *running*. Once the current execution segment of Task1 ends, it is calculated how the execution of the preempted task would have been effected by the preemption. For the shown example, Task1 should have been delayed for the execution duration of Task2. With the recorded time points, it is easy to calculate this duration, which is $t2 - t1$. This time is then compensated to Task1 using an extra *wait()*. Modeled in this way, all the time stamps including the start times and stop times of both tasks match the real execution.

Figure 6.13 shows the simulation trace of simulating the example in Figure 6.2(a) using ROM. In the figure, the waits compensated to preempted tasks are referred to as *ROM waits*.

6.4.3 Dynamic Time-Slicing Method

The method we use is called *dynamic time-slicing method* (DTSM). The concept of this method is illustrated in Figure 6.14. As shown, the execution segment of Task1 breaks, when the interrupt occurs. The rest time is recorded. Meanwhile, the state of Task1 is changed from *running* to *ready*. Then, Task2 can start to run. After the execution segment of Task2 ends, Task1 resumes and the rest time of its execution segment is converted to the SystemC simulation time using an extra *wait()*. In this way, the system can be simulated exactly the same as the real execution. From the programmer's view, the execution segment of Task1 is splitted dynamically into two parts at the point where the interrupt occurs. That's why the method is called dynamic time-slicing method.

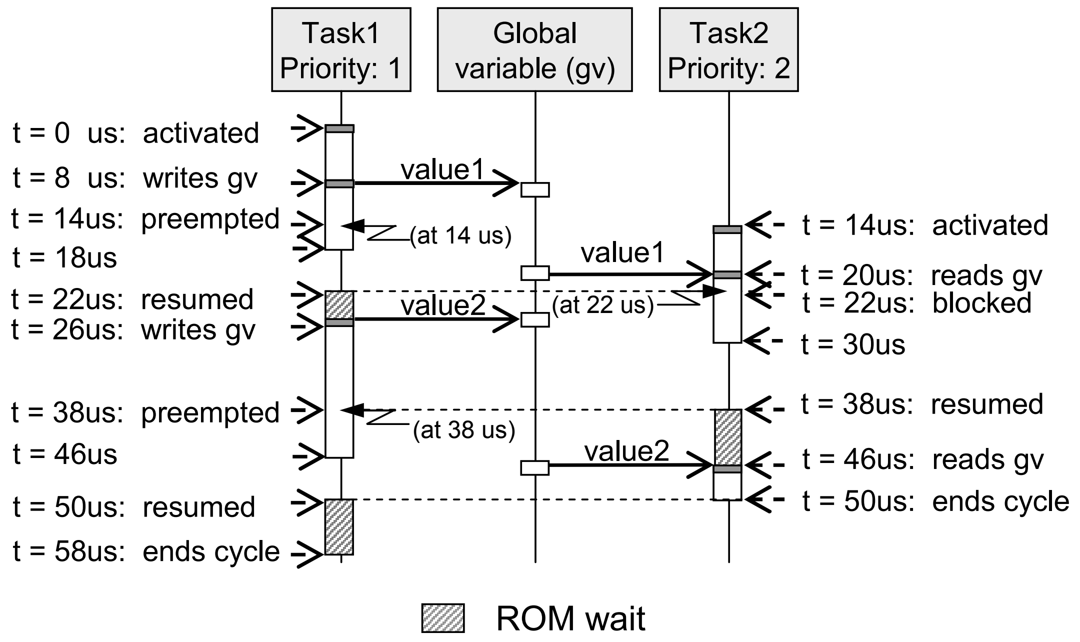


Figure 6.13: Example of Preemption Modeling Using ROM

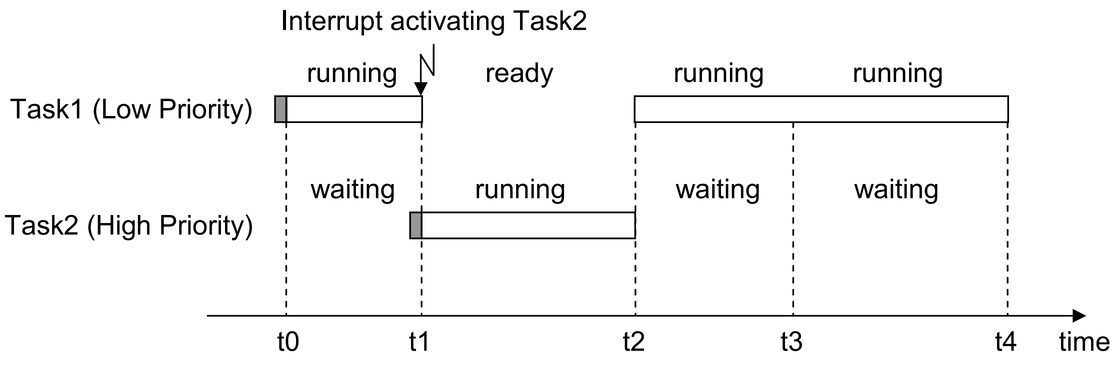


Figure 6.14: Dynamic Time-Slicing Method

The implementation of this method is shown in Figure 6.8. Using DTSM, there is no need of modifying task models. We just need to define the macro PREEMPTIVE. Then, the call to the interruptible wait *intWait(int task_id, sc_time exe_time)* is compiled instead of the standard *wait()* call (Figure 6.8(c), line 8).

Figure 6.8(d) shows the implementation of *intWait*. In line 5, we use a wait statement *wait(exe_time, preempt)* with both a time and an event as arguments. This statement is provided by SystemC. It suspends a thread by waiting for the event. As soon as the event is notified, the thread will resume. If the event does not occur within the given time, the thread will give up on the wait and resume. In our case, the time is the simulation time of an execution segment and the event is *preempt* from the scheduler. In line 4 and 6, the simulation time stamps before and after the *wait()* statement are recorded. Their difference *stop - start* is the actual simulation time advanced by the *wait()* statement. By subtracting this

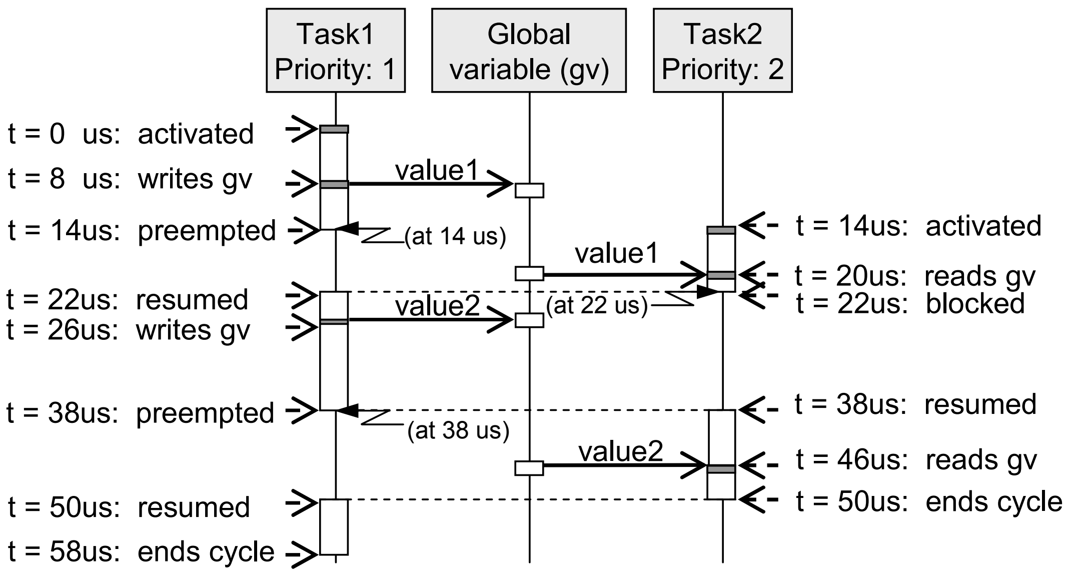


Figure 6.15: Example of Preemption Modeling Using Interruptible Waits

actually advanced simulation time from *exe_time*, we get the remaining time, i.e., *exe_time -= (stop - start)* (line 7). If the remaining time is zero, this means that the simulation time has been advanced without being interrupted. Otherwise, it indicates the occurrence of an interrupt. In this case, the task is preempted and its state is changed from *running* to *ready*. The code in line 9 and 10 changes the state of the task and suspends the thread by waiting for the notification of the dispatch[task_id] event from the scheduler.

Figure 6.15 shows the simulation trace of simulating the example in Figure 6.2(a) using the proposed method. As shown, the simulation trace is exactly the same as the real execution.

6.5 A More Modular Software Organization of Transaction Level Processor Model

So far, we have described the transaction level models of application task, cache, and RTOS, which are primary components of a processor. Hence, we can get a transaction level processor model by combining these models. We call a transaction level processor model *cpuTLM* for short in the following discussion. A cpuTLM is functionally identical to the target processor, and meanwhile, can accurately estimate the delays caused by software execution on the target processor. Further, to simulate a multiprocessor system, several cpuTLMs simulating the software processors are connected to HW simulation models and memory models through communication architecture. This is illustrated in Figure 6.16. As shown, a cpuTLM is modeled as a single SystemC module, where application tasks are

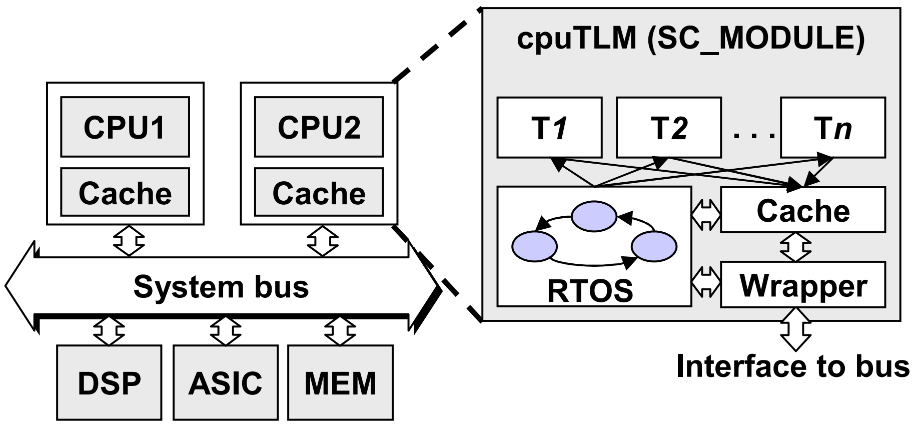


Figure 6.16: Transaction Level Processor Model

modeled as SystemC threads and the services of the RTOS and cache are invoked by function calls. Modeled in this way, the advantage is that in a single SystemC module the synchronization between different components can be easily realized by events. The simple synchronization can result in high simulation performance. Nevertheless, this implementation has the disadvantage of less reusability and modularity of the model components. In a multiprocessor system simulation, a copy of RTOS model and cache model is needed in each cpuTLM. A modification or refinement of the RTOS model or the cache model will lead to the change of all the processor models.

To achieve a more modular software organization, we implement both cache and RTOS as SystemC hierarchical channels. The software task threads are wrapped in a separate module *taskTLM*. The threads use the services provided by RTOS and cache through the interfaces of the RTOS and cache channels. Nevertheless, this software organization has the difficulty of synchronizing the task threads in the taskTLM module using functions or threads in the RTOS module, because the communication between two modules cannot be realized easily using SystemC events. In other words, the scheduler in the RTOS module is not able to schedule the threads in the taskTLM module directly.

Our solution to this problem is to divide an original task model into two parts: a task model and a *shadow task model*, as shown in Figure 6.17. The task model executes the functional code and aggregates the annotated cycle values for each execution segment and the shadow task model advances the simulation time given the aggregated cycle value of the execution segment. The shadow task model is located in the RTOS channel and therefore can be easily controlled by the scheduler, while the task model is in the taskTLM module. The task model and the shadow task model are tightly coupled using FIFOs. There are three FIFOs connecting each pair of task model and shadow task model, namely, *fifo_start*, *fifo_cycles*, and *fifo_ack*. They are all one-stage FIFOs, accessed by blocking read

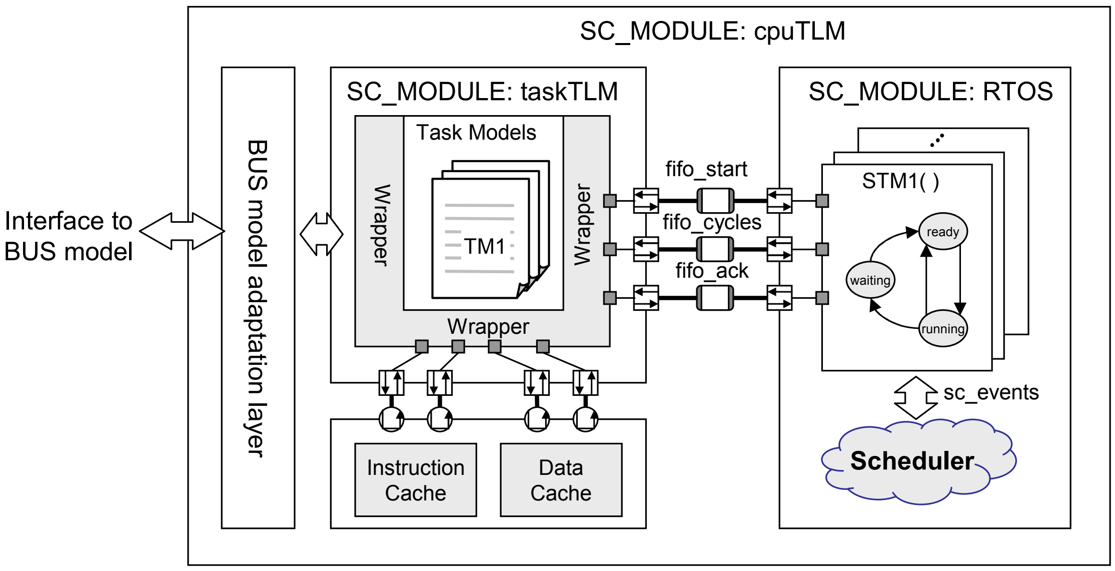
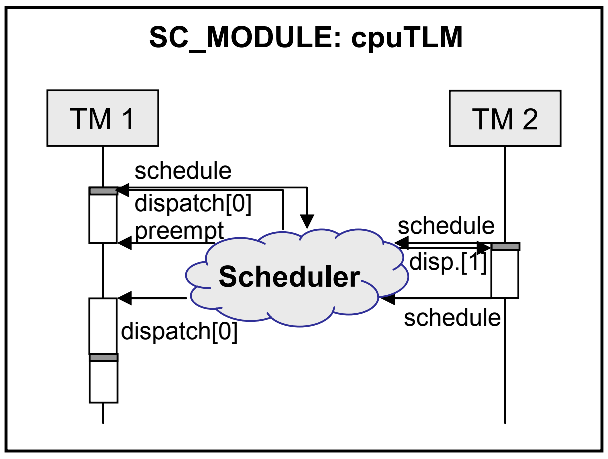


Figure 6.17: A More Modular Software Organization

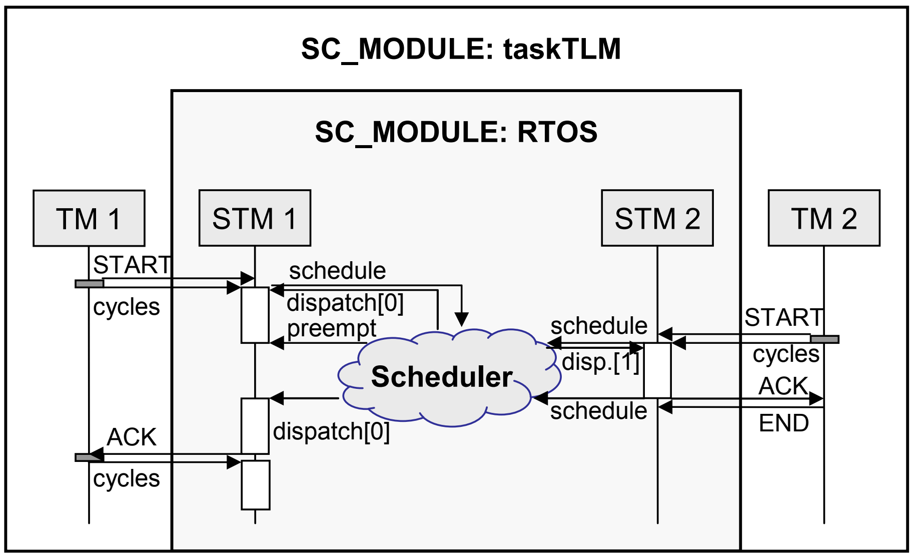
and write, for the sake of correct synchronization between the task model and the shadow task model.

The proposed strategy is depicted in Figure 6.18 with an example. The example is kept simple for the purpose of clarity. Figure 6.18(a) shows the simulation trace using the way introduced in the last section. In the example, there are two tasks. The first task model (TM1) has a lower priority than the second task model (TM2) and starts first. It is first inserted into the ready queue and requests for a scheduling by sending the *schedule* event to the scheduler. As at that time there is only one task in the ready queue, TM1 is immediately activated after receiving the *dispatch[0]* event from the scheduler. During its execution, an event that activates TM2 occurs. In the same way, TM2 is inserted in the ready queue and requests for a scheduling. As TM2 has a higher priority, it can run next and therefore TM1 is preempted. When TM2 finishes its execution, it sends the *schedule* event to trigger a scheduling again. As TM1 is the only ready task left, it can resume its execution. Totally, TM1 contains two execution segments and TM2 has only one.

Figure 6.18(b) shows the simulation trace using the proposed strategy. As shown, each task model in the taskTLM module has a corresponding shadow task model in the RTOS channel. As all the shadow task models do the same thing and therefore can reuse the same code. The code is shown in Figure 6.19(c). The task models still play the role in reacting with the environment. In the illustrated example, at the beginning, both task models wait for being activated by an external event, while both shadow task models are also blocked by reading the blocking FIFO *fifo_start*, which is initially empty (Figure 6.19(c), line 6). Then, TM1 can start first. It writes START to *fifo_start*. Its shadow task model STM1 gets the data and is also activated. It then changes the task state from *waiting* to *ready* (line 7). Next, it reads the cycle value of the current execution segment from *fifo_cycles*



(a) Modeling Using a Single SystemC Module



(b) Modeling RTOS Using a SystemC Channel

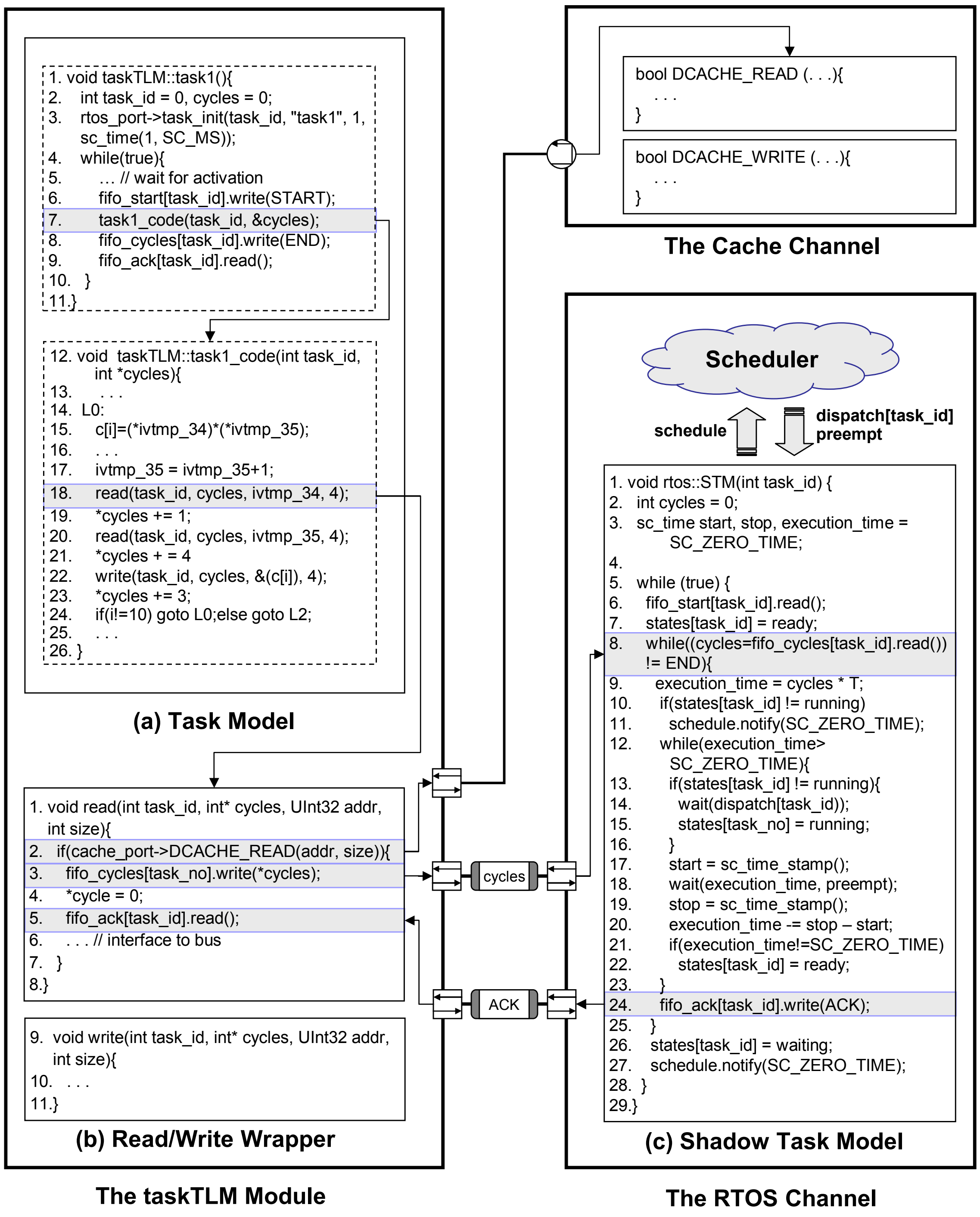
Figure 6.18: An Example of Multi-Task Simulation Using Abstract RTOS Models

sent by TM1. If the cycle value does not exist, STM1 is blocked. Meanwhile, TM1 executes the functional code and calculates the cycles that the current execution segment takes. An execution segment ends when there is an access to an external device. Before modeling the access, the aggregated cycle value must be converted to the simulation time. Therefore, TM1 sends the cycle value to *fifo_cycles* and then is blocked by waiting for an acknowledgment from STM1, which indicates the completion of the simulation time advancement of the current execution segment.

As soon as STM1 gets the cycle value, it is really ready to run. The cycle value is first converted to *sc_time* by multiplying with the duration of each CPU cycle T (line 9). The resulting time is assigned to a variable *execution_time*. Then, it is checked whether the task is already running (line 10). If the current execution segment is the first execution segment of the task, the task state is not *running* but *ready* and hence we have to notify the *schedule* event to request a scheduling (line 11). For the first execution segment, STM1 sends the *schedule* event to the scheduler and waits for its turn to run (line 14). The scheduler then decides which task can run next. As currently there is only one task ready, STM1 gets the *dispatch[0]* immediately and changes its state from *ready* to *running* (line 15). The code from line 17 to line 22 advance the simulation time, taking preemption into account. *execution_time* is the time to be converted to the simulation time, while the difference between the two time stamps *stop* and *start* calculates the really advanced simulation time. Therefore, at line 20, "*execution_time -= stop - start*" calculates the rest of time to be converted to simulation time. If the rest of time is zero, this means that the execution time of the current execution segment has been completely converted to the simulation time without being interrupted. Otherwise, the task has been preempted and its state is changed to *ready* (line 22). Then, the loop begins the next iteration from line 12. It iterates until the simulation time of the current execution segment is completely advanced. In the example, during the first execution segment, STM1 is preempted and suspended by waiting for being dispatched (line 14). STM2 runs in the same manner. When it finishes its execution, STM1 resumes and converts the rest of execution time to the simulation time. At the completion of the first execution segment, STM1 sends an acknowledgment ACK to *fifo_ack* (line 24) and then jumps back to line 8, waiting for the cycle value of the next execution segment.

At the TM1 side, when it gets ACK from *fifo_ack*, it starts the next execution segment. TM1 has only two execution segments. After finishing the last execution segment, it sends END to *fifo_cycles*. In STM1, as the variable *cycles* gets END, the loop ends (Figure 6.19(c), line 8). The task state is changed to *waiting* (line 26) and a rescheduling is requested (line 27). STM1 then waits for the next round of activation (line 6).

As described above, using one-stage FIFOs, the task model and the shadow task model are well synchronized. An execution segment in the task model starts from functional code execution and ends with receiving an acknowledgment from the shadow task model. An execution segment in the shadow task model starts



```
void taskTLM::task1(){
  int task_id = 0, cycles = 0;
  rtos_port->task_init(task_id, "task1", 1,
  sc_time(1, SC_MS));
  while(true){
     … // wait for activation
     fifo_start[task_id].write(START);
     task1_code(task_id, &cycles);
     fifo_cycles[task_id].write(END);
     fifo_ack[task_id].read();
  }
}
```

```
void  taskTLM::task1_code(int task_id,
        int *cycles){
     . . .
 L0:
    c[i]=(*ivtmp_34)*(*ivtmp_35);
    . . .
    ivtmp_35 = ivtmp_35+1;
    read(task_id, cycles, ivtmp_34, 4);
    *cycles += 1;
    read(task_id, cycles, ivtmp_35, 4);
    *cycles + = 4
    write(task_id, cycles, &(c[i]), 4);
    *cycles += 3;
    if(i!=10) goto L0;else goto L2;
    . . .
}
```

```
bool DCACHE_READ (. . .){
   . . .
}
```

```
bool DCACHE_WRITE (. . .){
   . . .
}
```

```
void read(int task_id, int* cycles, UInt32 addr,
  int size){
  if(cache_port->DCACHE_READ(addr, size)){
    fifo_cycles[task_no].write(*cycles);
    *cycle = 0;
    fifo_ack[task_id].read();
    . . . // interface to bus
  }
}
```

```
void write(int task_id, int* cycles, UInt32 addr,
  int size){
  . . .
}
```

```
void rtos::STM(int task_id) {
 int cycles = 0;
 sc_time start, stop, execution_time =
       SC_ZERO_TIME;

 while (true) {
   fifo_start[task_id].read();
   states[task_id] = ready;
   while((cycles=fifo_cycles[task_id].read())
   != END){
     execution_time = cycles * T;
      if(states[task_id] != running)
        schedule.notify(SC_ZERO_TIME);
      while(execution_time>
        SC_ZERO_TIME){
        if(states[task_id] != running){
          wait(dispatch[task_id]);
          states[task_no] = running;
        }
        start = sc_time_stamp();
        wait(execution_time, preempt);
        stop = sc_time_stamp();
        execution_time -= stop – start;
        if(execution_time!=SC_ZERO_TIME)
          states[task_id] = ready;
      }
      fifo_ack[task_id].write(ACK);
   }
   states[task_id] = waiting;
   schedule.notify(SC_ZERO_TIME);
 }
}
```

Figure 6.19: The Code Structure of a cpuTLM

from receiving the cycle value from the task model and ends with sending an acknowledgment to the task model.

In Figure 6.19, we show the code structure of a cpuTLM. The illustrated example of task model is the same as the one in Figure 6.8. A task model is declared as a thread in the taskTLM module. At the beginning of the thread, the task is assigned an id (line 2) and calls the interface method *task_init()* implemented in the RTOS channel through the port *rtos_port*, through which taskTLM is connected to the RTOS channel (line 3). This interface method is responsible to spawn a thread for the shadow task model and initializes the task's context. The code of *task_init()* is shown as follows:

```
1. int rtos::task_init(int task_id, char *task_name, int priority,
   sc_time deadline){
2.   sc_spawn( sc_bind(&rtos::STM,this,task_id) );
3.   task_names[task_id] = task_name;
4.   priorities[task_id] = priority;
5.   states[task_id] = waiting;
6.   deadlines[task_id] = deadline;
7.   return 0;
8. }
```

In the above code, the function *sc_spawn()* is called to spawn a SystemC thread for a shadow task model (line 2). Because the shadow task models of different tasks do exactly the same thing, the same function *STM(int task_id)* shown in Figure 6.19(c) is attached to all the threads of shadow task models. Given *task_id* as a parameter, the code in *STM()* can access the correct FIFOs and task parameters, which are stored in arrays indexed by the task id. *sc_spawn()* relies on an additional function *sc_bind()* to attach the function *STM()*. This is done by passing the address of *STM()* as an argument. Another argument *this* is used to indicate that *STM()* is a member function. Finally, as the function *STM()* has one input parameter *task_id* of type *int*, we set the actual task id of a task inside the *sc_bind()* call.

The task body is an infinite loop. At the beginning of the loop, the task waits for being activated (Figure 6.19(a), line 5). Once it is activated, it sends START to *fifo_start* to notify the shadow task model (line 6). Then, it runs the instrumented ISC (line 12 through line 26), which is exactly the same as in Figure 6.8(b). This means that for the new RTOS model the instrumented ISC generated by iSciSim does not need any modification. During the execution of the instrumented ISC, the functional code is executed and meanwhile the software execution delays are aggregated using the variable *cycle*. As an example of memory access, at line 18, a wrapper function *read()* is called. In the *read()* function, the interface method *DCACHE_READ()* implemented in the cache channel is called through *cache_port* for cache simulation (Figure 6.19(b), line 2). In the case of a cache miss, the aggregated cycle value is sent to *fifo_cycles* (line 3). Then, the cycle counter is cleared (line 4) and an access to the memory starts (line 6) after an acknowledgment from the shadow task model is received (line 5). The *write()* function is implemented in the same way. After the execution of the instrumented ISC, the task model sends END to notify the shadow task model of the termination of the task (Figure 6.19(a), line 8). After getting an acknowledgment, the task really terminates.

6.6 Experimental Results

In the experiment, we evaluated our modeling method with simulating a simple example: a single processor running 6 independent, periodic tasks. The tasks are the same as those used in the experiment in Chapter 5. A PowerPC 100 MHz processor was selected the target processor. The simulations ran on a laptop equipped with an Intel Core 2 Duo CPU at 2.26 GHz and 2 GB memory. The execution environment was Cygwin on Windows XP.

Table 6.1: Task Parameters

	Period	**Priority**	**Deadline**
fibcall	1 ms	4	1 ms
insertsort	0.1 ms	6	0.1 ms
bsearch	3 ms	5	3 ms
crc	1 ms	3	1 ms
blowfish	5 ms	2	5 ms
AES	50 s	1	50 s

We used iSciSim to generate fast and accurate transaction level task models for all the tasks. It has been validated in Chapter 5 that all these taskTLMs allow for very accurate performance simulation, comparable to simulation using an ISS.

First, we tested one of the most common static-priority scheduling algorithms: the priority of each task is pre-defined by the user. In Table 6.1, we define the parameters of these tasks, including deadline, period, and priority. The deadline is equal to the period. We ran the simulation for a simulated time of 150 seconds. The host execution time was 138 seconds. This means that the simulation performance was around 109 Mcycles/sec. It is a very high simulation speed, which allows for exploring various design alternatives in a short time. Table 6.2 shows the simulation results, including the total execution time (Total ET) and the worst-case response time (WCRT) of each task. We can see that except *AES* all the other tasks finish execution before the deadline. In the period of 150 seconds, *AES* runs only once. The worst-case response time is 65.18 seconds, larger than the deadline 50 seconds. The processor was already fully loaded with a utilization rate of 91.9%.

Table 6.2: Simulation Results

	Total ET		**WCRT**		**Deadline**
	FP	**EDF**	**FP**	**EDF**	
fibcall	4.88 s	4.91 s	484.19 us	145.89 us	1 ms
insertsort	5.99 s	5.99 s	4.34 us	4.34 us	0.1 ms
bsearch	21.48 s	21.50 s	451.33 us	592.88 us	3 ms
crc	15.43 s	15.06 s	592.78 us	139.85 us	1 ms
blowfish	50.29 s	50.29 s	2.63 ms	2.63 ms	5 ms
AES	39.81 s	39.85 s	65.18 s	65.00 s	50 s

We also tested the EDF scheduling algorithm. For the same scenario, we got similar results as FP: *AES* missed the deadline and the other tasks all could complete before the respective deadlines. From Table 6.2, we find that with EDF *fibcall* and *crc* have significantly shorter WCRTs than with FP. This is because the two tasks have both relatively short deadlines. In the simulation using EDF, they can get higher priorities than the priorities set in the simulation using FP. In contrast, *bsearch* gets a longer WCRT using EDF. The simulation results show that changing the scheduling algorithm can not help in getting a safe design. There are two solutions: either to use a processor with higher computing power or to implement *AES* in hardware.

So far, we tested the RTOS model only with this simple scenario. It is our future work to apply it in practical designs.

Chapter 7

Flow Analysis on Intermediate Source Code for WCET Estimation

Constraining the possible control flows in a program is essential for a tight WCET bound. The results of flow analysis are pieces of control flow information on loop bounds and infeasible paths, which are often called *flow facts*. It is most convenient to extract flow facts from the source level, where the program is developed. However, as timing analysis and WCET calculation are usually performed on the binary code that will be executed on the target processor, the source level flow facts must be transformed down to the binary level. Due to optimizing compilation, the problem of this transformation is nontrivial.

This chapter introduces an efficient WCET approach that performs flow analysis on intermediate source code (ISC). On one hand, ISC is high level enough for manual annotations and automatic flow facts extraction. On the other hand, it is low level enough to keep flow facts transformation easy. The approach has another advantage that it is easy to realize and to use and does not require any modification of standard tools for software development.

The concrete contributions of this dissertation to WCET estimation are threefold:

1. We show how the usage of ISC facilitates the extraction of flow facts. In ISC, complex statements are unstructured to simple statements, which makes analyses like *program slicing*, *syntax analysis*, and *abstract interpretation* much easier.
2. We show how to simply transform the ISC level flow facts to the low level by using debugging information.
3. We propose an experiment method to demonstrate only the effectiveness of flow analysis. This allows us to evaluate a flow analysis method and a timing analysis method separately.

This chapter is organized as follows: An overview of the proposed approach as well as the tool architecture is presented in Section 7.2 after an overview of related work on flow analysis in Section 7.1. The whole work flow consists of four steps:

ISC generation, flow analysis, flow facts transformation, and timing analysis and WCET calculation. ISC generation has been introduced in Chapter 5. The other three steps are described in Section 7.3, 7.4, and 7.5, respectively. Experiment methodology and results are presented in Section 7.6.

7.1 Flow Analysis and Related Work

In this section, we discuss only about previous work on flow analysis. There are also many research activities that focus on other aspects of WCET analysis. It is outside the scope of this dissertation to give an overview of these works. Instead, some survey papers like [103] can be referred to.

According to the representation level of programs, where flow analysis is performed, the existing flow analysis approaches can be grouped into three categories, as shown in Figure 7.1. In the first category of approaches (Figure 7.1(a)), both flow analysis and timing analysis are performed on binary code. The obtained flow facts can be directly forwarded to timing analysis and WCET calculation without the need of any transformation. The approach described in [72] and the well-known commercial tool aiT [1] are in this category. They apply *symbolic execution* and *abstract interpretation* [36] for flow analysis, respectively. The approaches in this category have a common limitation that the automatic flow analysis succeeds only for simple control flows that do not depend on input data. Otherwise, additional flow facts have to be given manually at the binary level, which is a very error-prone task.

It is much easier to do manual annotations at the source level, where programs are developed. Also, it is advantageous to do automatic flow analysis at this level, as the programmer can easily check the correctness of the obtained flow facts. To support manual annotations in source programs, some previous works propose to use special annotation languages such as MARS-C [84] and wcetC [59]. Using state-of-the-art flow analysis techniques, most input data independent flow facts can be extracted automatically without manual annotations. The approach for source level flow analysis proposed by Healy et al. in [51] is able to bound the number of iterations automatically but is limited to three special types of loops. Ermedahl and Gustafsson [43] use *abstract execution*, which is a type of abstract interpretation, to discover false paths and loop bounds. In abstract execution, loops are "rolled out" dynamically and each iteration is analyzed individually. However, the high level flow facts obtained using these approaches cannot be used directly for low level timing analysis and WCET calculation, because optimizing compilation can change the control flows of a program significantly. Therefore, a mechanism to map the high level flow facts to the binary level is necessary. As shown in Figure 7.1(b), the flow analysis in this category of approaches consists of two phases: (1) automatic flow facts extraction and manual annotation at the source level and (2) flow facts transformation to the binary level.

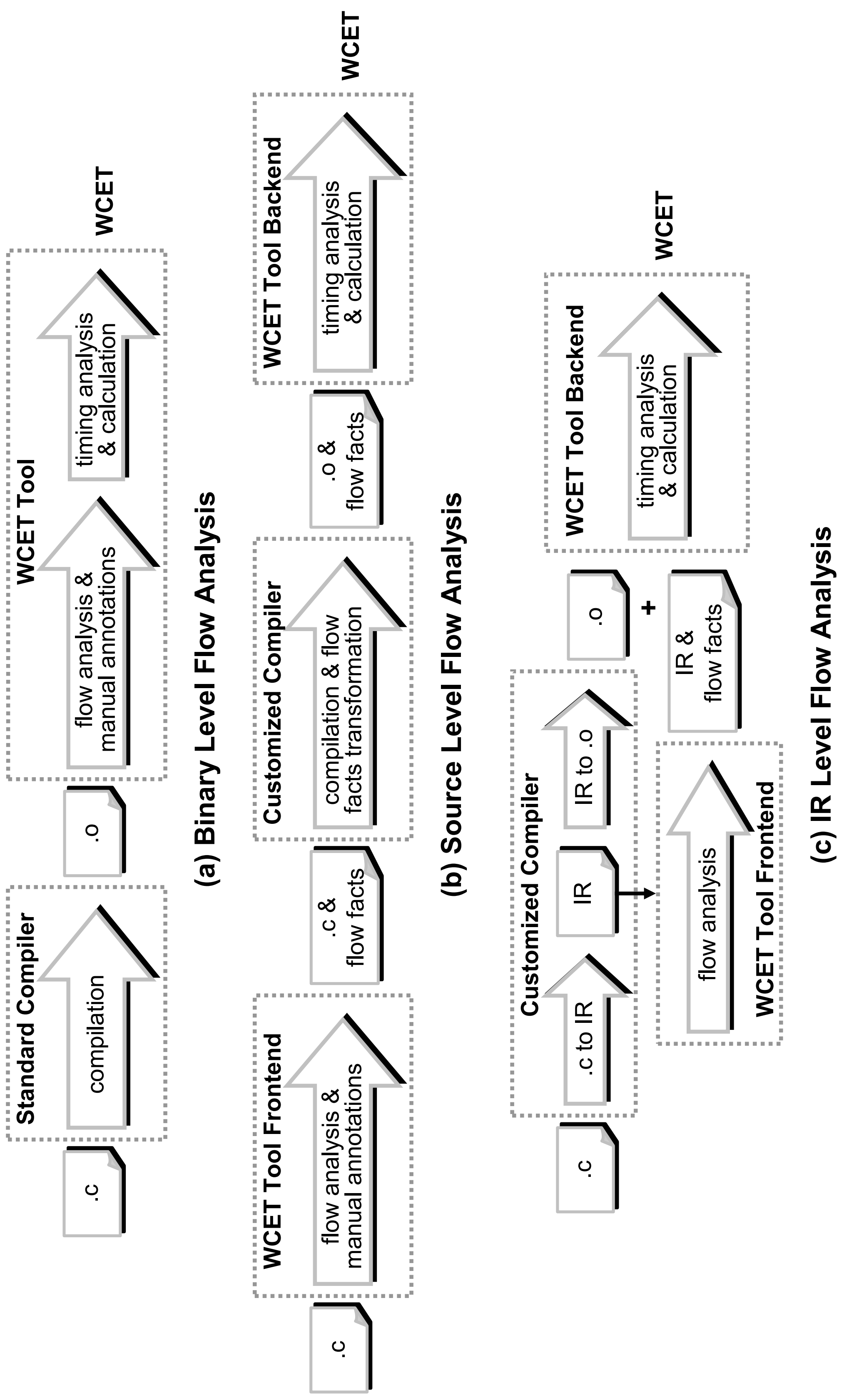

Standard Compiler
.c
compilation
.o
WCET Tool
flow analysis & manual annotations
timing analysis & calculation
WCET
(a) Binary Level Flow Analysis
WCET Tool Frontend
.c
flow analysis & manual annotations
.c & flow facts
Customized Compiler
compilation & flow facts transformation
.o & flow facts
WCET Tool Backend
timing analysis & calculation
WCET
(b) Source Level Flow Analysis
Customized Compiler
.c
.c to IR
IR
IR to .o
.o
+
IR & flow facts
WCET Tool Frontend
flow analysis
WCET Tool Backend
timing analysis & calculation
WCET
(c) IR Level Flow Analysis

Figure 7.1: Typical Work Flows of WCET Analysis Tools

The problem of flow facts transformation is nontrivial. Puschner presents a low-cost approach in [83] to transform source level flow facts to the binary level, but this approach is limited to moderately optimized programs. To support WCET analysis of programs compiled with usual, standard optimizations, one feasible solution is to integrate flow facts transformation within program compilation to keep track of the influence of compiler optimizations on the flow facts. This is achieved using two separate tools in [42]: a modified compiler tracing the compiler optimizations and, given the traces, a *co-transformer* performing flow facts transformation. In [58] Kirner and Puschner integrate a similar transformation approach fully into a GCC compiler. The tool TuBound [81] uses abstract interpretation for automatic flow analysis and Kirner's method [58] for flow facts transformation.

The third category of approaches shown in Figure 7.1(c) performs flow analysis on a low-level IR. It achieves a better trade-off between visibility of flow facts and simplicity of their transformation to the binary level. The tool SWEET [48] belongs to this category. It uses a special compiler to transform C programs to a new IR called *new intermediate code* (NIC), which can be regarded as a high-level assembler code format and is written in a LISP-like syntax. Flow analysis is performed on the NIC code. As the NIC code has a structure close to the structure of the binary code, the generated flow facts can easily be mapped to the binary level. Nevertheless, this approach relies on the special compiler and the special IR format and thus has limited usability in practice. In addition, NIC code has very limited readability according to the example shown in [48].

7.2 Overview of the Proposed Approach and Tool Architecture

From our point of view, a good WCET analysis approach should have the following properties:

- To get a tight WCET bound, the representation level of programs for flow analysis, on one hand, should be high enough to facilitate flow facts extraction, and on the other hand, should be low enough, so that the effort for flow facts transformation is kept low.
- Timing analysis should be performed on the binary code, which is to be executed on the target hardware, for accurate timing modeling.
- The WCET analysis approach should be general enough and not limited to a special compiler.

According to the discussion in Section 7.1, none of the existing approaches achieve all the above properties. Motivated by this fact, we developed a new approach that

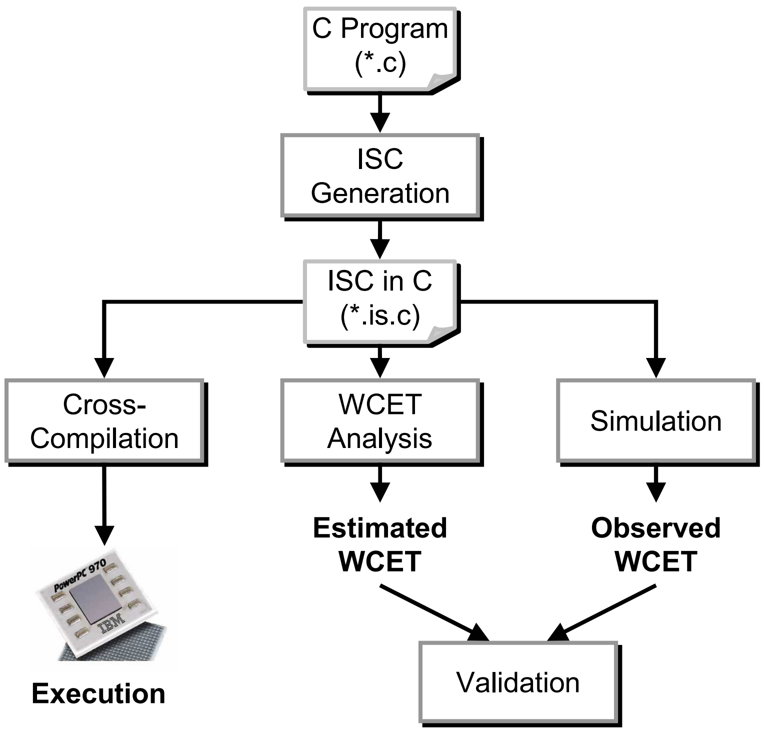
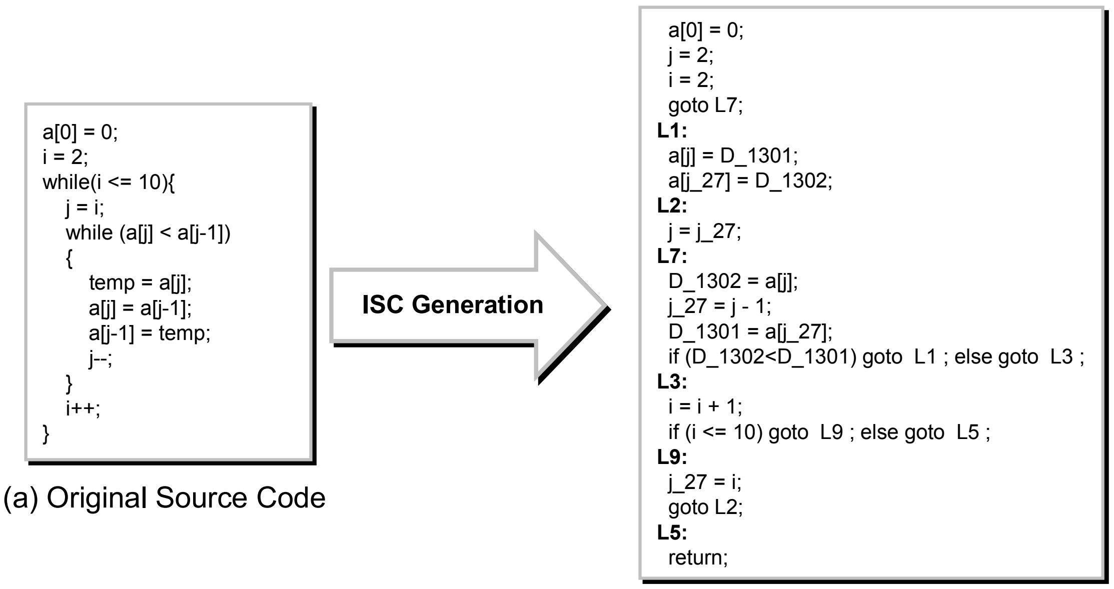


Figure 7.2: The Usage of ISC

performs WCET analysis on ISC. According to the descriptions in Chapter 5, ISC has the following features: (1) it is written in a high-level programming language, and thus, is compilable; (2) it has the same functional behavior as the original source code; (3) it contains enough high-level information for necessary manual annotations; (4) it has a structure close to that of the binary code for easy flow facts transformation.

In our current implementation, ISC is written in the C programming language and generated by slight modification of standard IRs of standard compilers. The ISC generation has been introduced in Chapter 5. Here, we do not repeat it but just illustrate it with another example, the program *insertsort* from the Mälardalen WCET benchmark suite [9], which will be used to illustrate the whole proposed approach through this chapter. Figure 7.3 shows this example. Figure 7.3(b) contains the generated ISC. As many high-level information of the original source program has been retained in the ISC, the programmer can easily specify the value intervals of the input variables, which cannot be computed automatically by the tool.

ISC retains exactly the same semantics and executability of its original source code. When compiled with a C compiler, it shows exactly the same functionality as the original C program. Nevertheless, there might be a slight difference between the temporal behaviors of the ISC and the original source code. As we perform timing analysis on the binary code generated by the ISC, this binary code should also be



```
a[0] = 0;
i = 2;
while(i <= 10){
   j = i;
   while (a[j] < a[j-1])
   {
      temp = a[j];
      a[j] = a[j-1];
      a[j-1] = temp;
      j--;
   }
   i++;
}
```

(a) Original Source Code

```
  a[0] = 0;
  j = 2;
  i = 2;
  goto L7;
L1:
  a[j] = D_1301;
  a[j_27] = D_1302;
L2:
  j = j_27;
L7:
  D_1302 = a[j];
  j_27 = j - 1;
  D_1301 = a[j_27];
  if (D_1302<D_1301) goto  L1 ; else goto  L3 ;
L3:
  i = i + 1;
  if (i <= 10) goto  L9 ; else goto  L5 ;
L9:
  j_27 = i;
  goto L2;
L5:
  return;
```

(b) Intermediate Source Code

Figure 7.3: An Example of ISC Generation

finally executed on the target processor, as shown in Figure 7.2. In our experiments presented in Chapter 5, we have already shown that the binary code generated from an ISC program has similar performance as the one generated from the original source program. This validated that the usage of ISC for execution on the target hardware will not reduce the system performance. As shown in Figure 7.2, we also get *observed WCET* by means of simulation. This observed WCET is used to validate the estimated WCET. This is discussed more in Section 7.6.

The work flow of WCET analysis on ISC is illustrated in Figure 7.4. The whole approach consists of five steps: (1) First, flow facts are extracted automatically from ISC. (2) Next, if necessary, we can give manually additional flow information that cannot be computed by the tool. (3) After that, the program is compiled to generate binary code and debugging information. The debugging information is then used to map the flow facts to the binary level. (4) Timing analysis is performed on the binary code based on a hardware model. (5) Given the low-level flow facts and timing information, the WCET is finally calculated. As shown, the whole approach is performed fully outside the standard compilation approach.

7.3 Flow Analysis

In this section, we present the approach for bounding loop iterations and detecting infeasible paths. The whole approach consists of four steps: (1) First, a control flow graph (CFG) is constructed for each program at the ISC level. (2) Next, the unstructured loops are identified and the conditional branches within a loop

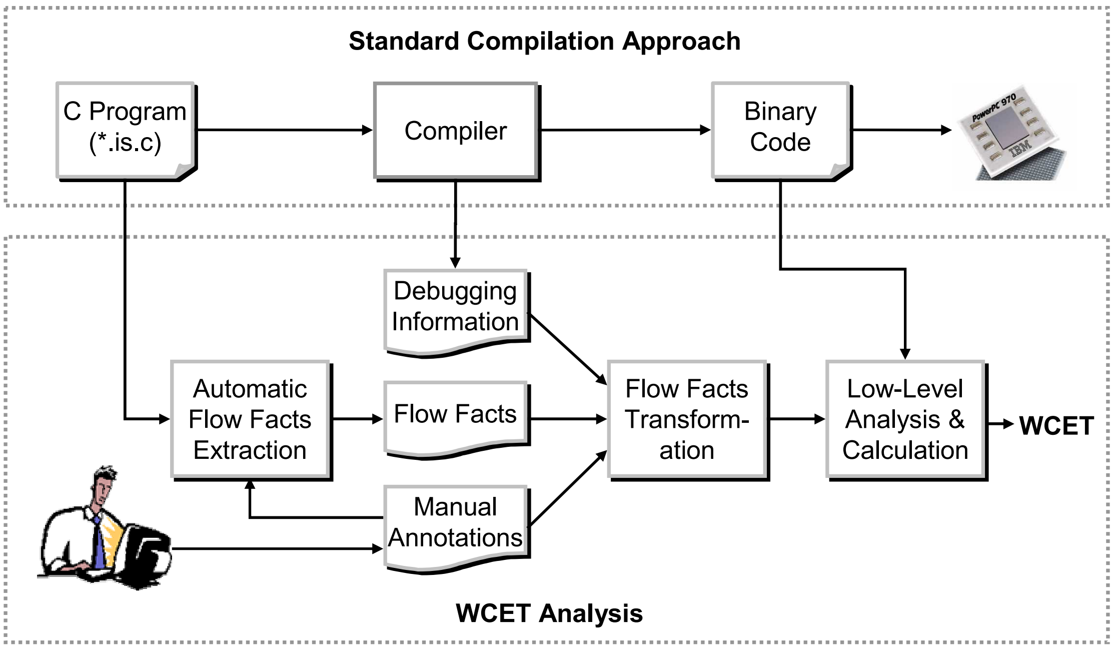

Standard Compilation Approach
C Program (*.is.c)
Compiler
Binary Code
Debugging Information
Automatic Flow Facts Extraction
Flow Facts
Flow Facts Transform-ation
Low-Level Analysis & Calculation
WCET
Manual Annotations
WCET Analysis

Figure 7.4: The Work Flow of Our WCET Analysis Tool

that can affect the number of loop iterations are identified. (3) Then, the CFG is reduced for special analyses, in order to reduce the analysis complexity. (4) Finally, loops are bounded and infeasible paths are identified. These working steps are introduced in the following sub-sections.

7.3.1 Constructing Control Flow Graph

A CFG is basically a graphical representation of a program. It is essential to many compiler optimizations and static analysis tools. Each node of the CFG corresponds to a *basic block*, which consists of a sequence of instructions that has only one entry point and one exit point. Basic blocks are connected with *edges*, which represent jumps in the control flow. An edge in the CFG is directed. A *back edge* is a special edge that points to an ancestor in a depth-first traversal of the CFG.

According to our implementation, a CFG is represented using a linked list. The labels contained in an ISC program simplify the work of constructing its CFG list. When a label is encountered while parsing the program, one new node is to be added into the CFG list. By parsing the ISC statements between two labels we can get all the needed information about the corresponding basic block. The CFG of the running example is shown in Figure 7.5(a).

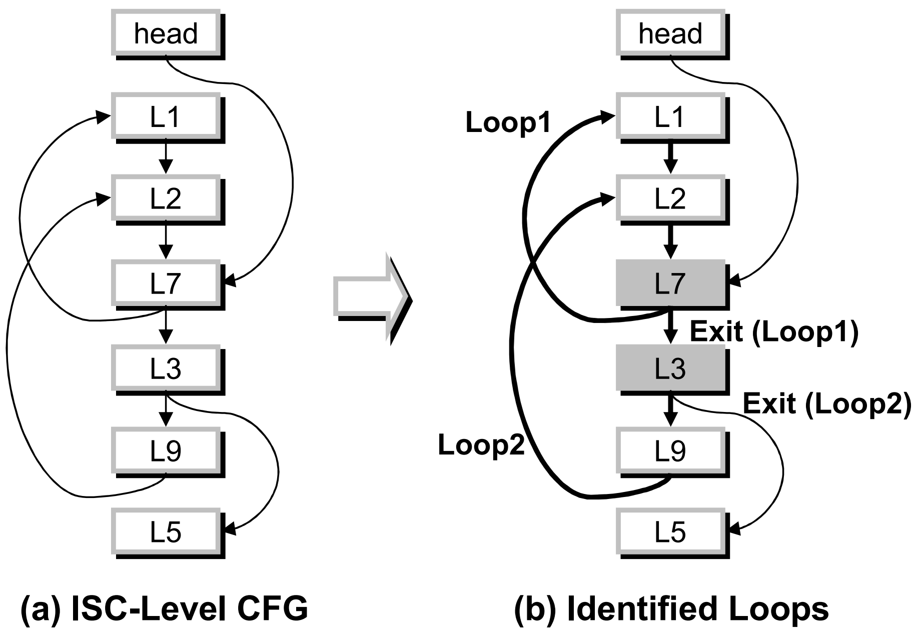


Figure 7.5: Identifying Loops

7.3.2 Identifying Loops

Each loop in a CFG contains at least one back edge. We define this kind of back edges as *loop-forming back edges*. Finding a back edge is considered as a necessary condition but not the sufficient condition of identifying a loop. To make sure that a back edge is really a loop-forming back edge, it must be checked whether there is a control flow leads the destination block of the back edge to its source block, i.e., to find a closed graph. For the running example, two nested loops are identified, as shown in Figure 7.5(b).

Next, the conditional branches that can directly or indirectly affect the number of loop iterations are identified. Such conditional branches are called *iteration branches*. Identifying iteration branches is mandatory for reducing CFG and bounding loop iterations. A branch that can leads the control flow to an exit edge is identified as a *direct* iteration branch, while a branch that leads to two different paths, affecting the number of iterations differently, is identified as an *indirect* iteration branch. The blocks containing iteration branches are shown in gray in Figure 7.5(b).

7.3.3 Reducing Control Flow Graph

In the ISC of a program, there might be many assignments that do not affect the conditions of conditional branches and thus do not affect the control flows. If these assignments are removed, the CFG will be reduced and the flow analysis complexity will be reduced accordingly, but the outcome of the analysis will still be the same.

In the running example, none of the assignments can be removed. To show the effect of CFG reduction, we give another example in Figure 7.6. The CFG in Figure 7.6(a) is constructed for the function *icrc()* in the benchmark *crc* [9]. In Figure 7.6(b), the blocks that do not affect the iteration branches are identified, shown in gray. For loop bounding these blocks can be removed. Figure 7.6(c) shows the reduced CFG. In addition, in each remaining block, the assignments that do not affect the iteration branches are removed. Analysis on this reduced CFG will be much easier and faster.

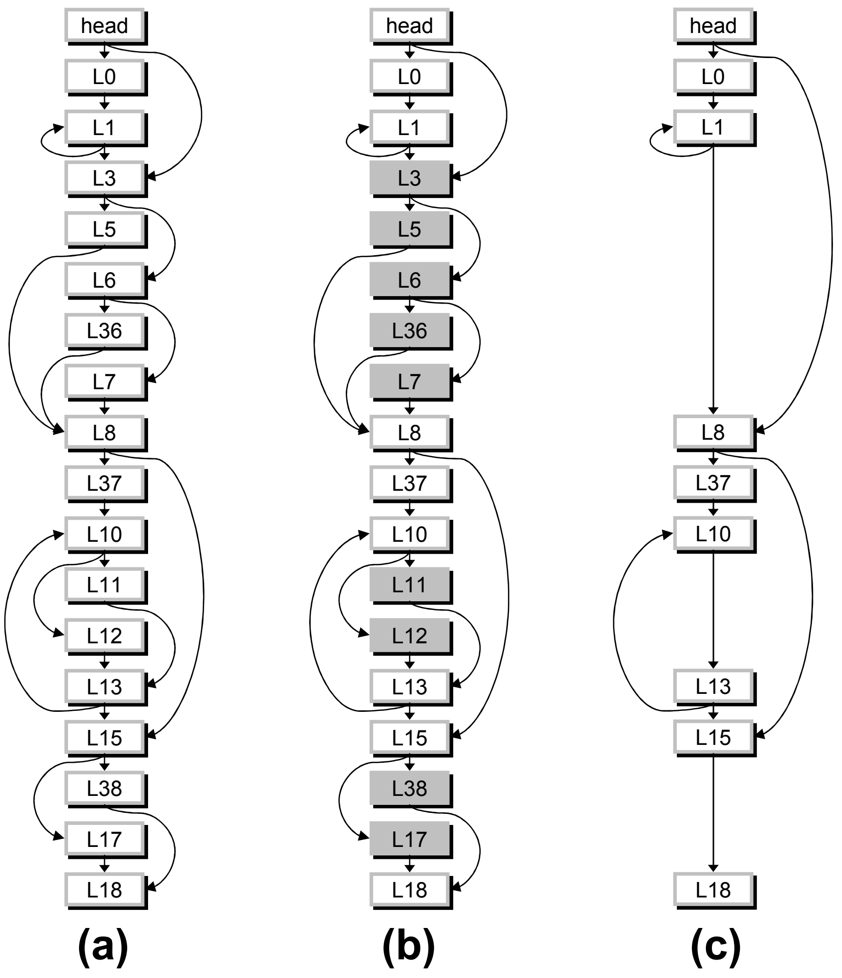


Figure 7.6: CFG Reduction

7.3.4 Bounding Loops and Detecting Infeasible Paths

Given the reduced CFG, we use three methods to bound loops and detect infeasible paths: *pattern analysis*, *code instrumentation*, and *abstract execution*. Pattern

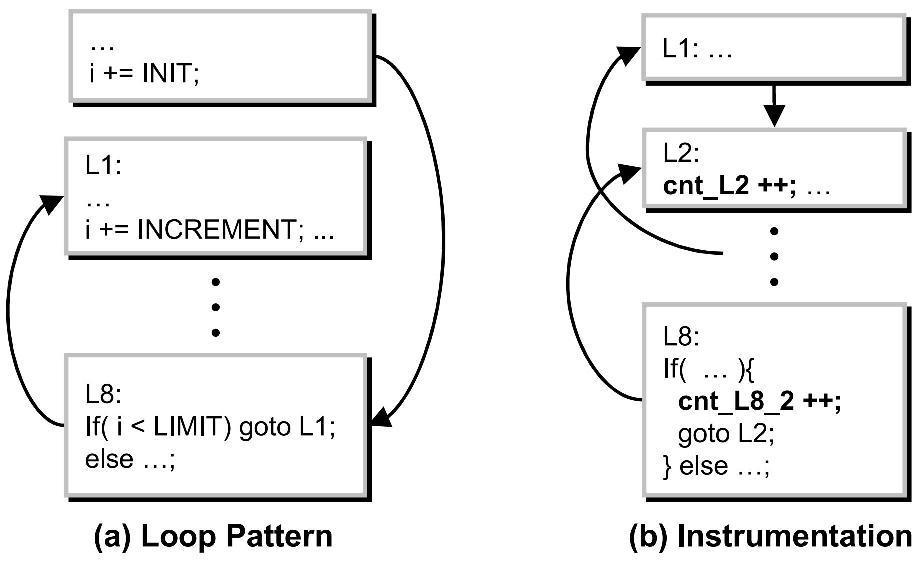


Figure 7.7: Loop Bounding Methods

analysis is used for bounding the loops that match some simple patterns. One of the simplest patterns is shown in Figure 7.7(a), where i is the loop counter and INIT, INCREMENT, and LIMIT are all constants. The number of iterations can be calculated simply using the equation:

$$N = \lfloor (LIMIT - INIT)/INCREMENT \rfloor.$$

Many loops like nested loops have more complicated structures and do not match any such simple patterns. If the number of iterations of such a complicated loop does not depend on input data, i.e, is statically known, we can apply the instrumentation technique to get its bound. Figure 7.7(b) illustrates this technique with an example. When the program is executed, the annotated variables *cnt_L2* and *cnt_L8_2* will count how many times L2 is executed and the number of jumps from L8 to L2, respectively.

The first two methods are not applicable to the loops, the iteration numbers of which depend on input data. To bound such loops, we can use the *abstract execution* technique, which is introduced in detail in [43, 48]. Using abstract execution, the infeasible paths can also be exploited. Our own work on abstract execution is still ongoing.

We output obtained flow facts in a separate file and the flow facts are expressed in a simple form. A loop bound is expressed by bounding the execution count of a block containing an iteration branch. For the running example, two loop bounds are obtained: (1) $L7 \leq 54$, (2) $L3 = 9$.

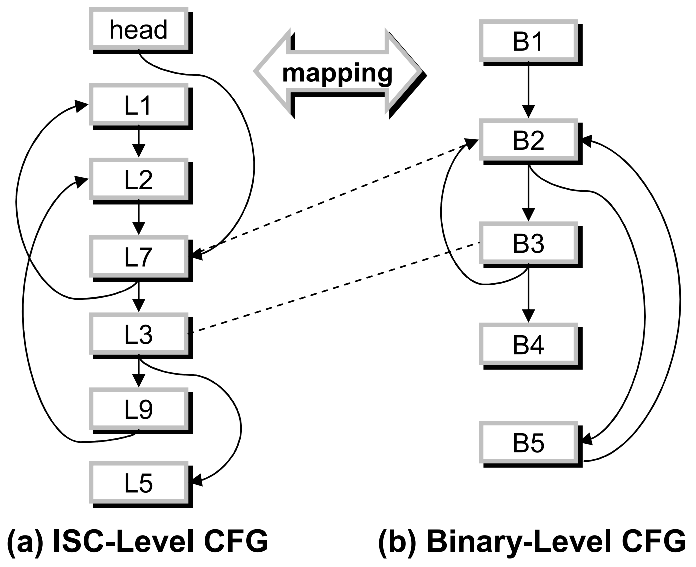


Figure 7.8: Mapping between ISC-Level CFG and Binary-Level CFG

7.4 Flow Facts Transformation

The structure of ISC is already highly optimized by the compiler frontend. If the ISC is compiled using the same compiler, the frontend will not change its structure any more. However, when the IR is lowered to the binary code, the compiler backend will further optimize the code structure to adapt to the special features of the target machine, e.g., by means of code motion and elimination. As shown in Figure 7.8, the two CFGs from the ISC level and the binary level look different. This difference makes it impossible to get a straightforward mapping, but fortunately, the two CFGs are still similar enough for flow facts transformation. As shown in Figure 7.9, if we reorganize the CFGs, the similarity is quite obvious.

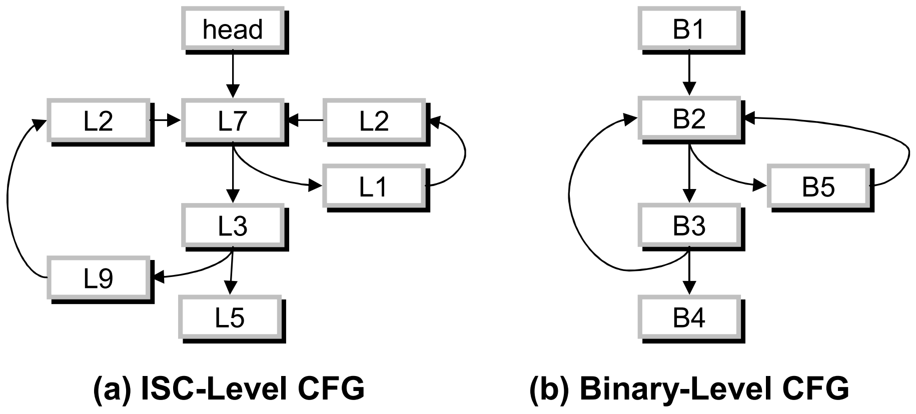


Figure 7.9: CFGs after Re-organization

The structure of a CFG is decided mainly by conditional branches. Such conditional branches can be regarded as *mapping points* to map the control flows. Our

ISC	Code Segment	Basic Block
	. . .	
L1: ...	...	B5, B2
L2: j = j_27;	NULL	NULL
L7: ... if (D_1302 < D_1301) goto...	... 0x18000f4 - 0x18000f8	... B2

L3:
i = i + 1;
if (i <= 10) goto L9 ; else ...
	...
	0x1800100 - 0x1800108
		...
		B3

L9: ...	NULL	NULL

Figure 7.10: Mapping Information

method for finding the counterparts of ISC-level conditional branches in the binary code is very simple: we make use *line table*, a kind of debugging information, which describes the correspondence between the ISC statements and the generated binary code. The line table of a program can be generated automatically during program compilation.

To transform loop bounds from the ISC level to the binary level, we just need to get a mapping of iteration branches. In each loop, there is at least one iteration branch that can find its counterpart in the binary code. That's the reason why we express a loop bound by bounding the execution count of a block containing an iteration branch.

In the following, we explain how the first flow fact $L7 \leq 54$ of the running example is transformed. Figure 7.10 shows all the information used in the description, including the ISC code, the binary code segments generated by some important ISC statements, and the basic blocks these code segments belong to. First, we find the iteration branch corresponding to the flow fact: "`if(D_1302<D_1301) goto ...`". Next, we get the line number of this iteration branch in the ISC program. Using this line number, the code segment "0x18000f4 - 0x18000f8" generated by the iteration branch is found by looking up the line table. Here, a code segment is usually either a basic block or part of a basic block. Thus, we get a mapping between the ISC-level basic block L7 that contains the iteration branch and the binary level basic block B2 that contains the code segment generated from the iteration branch. Based on this mapping, the loop bound is transformed to a low level loop constraint $x2 \leq 54$. In the same way, a mapping between L3 and B3 can be established and the second flow fact $L3 = 9$ is transformed to $x3 = 9$.

Figure 7.10 also shows that no code is generated by L2 and L9 and the code generated by L1 is partly in B2 and partly in B5, indicating that their structure has been changed by the machine-dependent optimizations. Flow facts transformation based on the mapping of these blocks instead of the iteration branches will result in a wrong estimate.

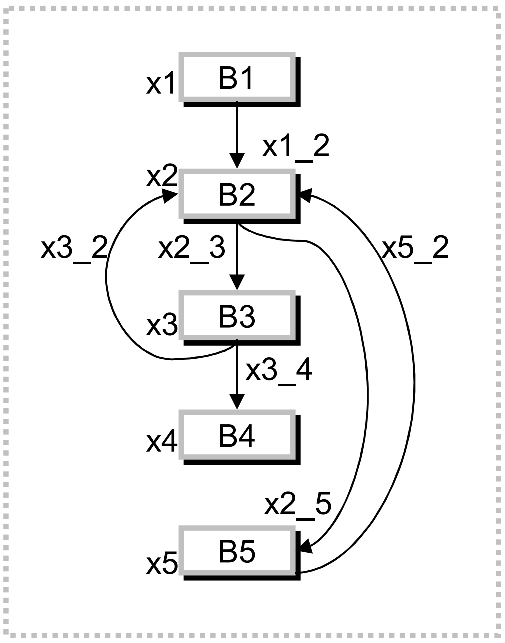
Flow facts on infeasible paths can be transformed to the low level in a similar manner. The only difference is that an ISC level flow fact on an infeasible path should be expressed as a constraint on an edge in the infeasible path, instead of a block. This edge can find its corresponding edge at the binary level using the line table. With this mapping the flow fact can be transformed.

7.5 Timing Analysis and WCET Calculation

Our flow analysis approach is fairly general to be adapted to any low level analysis. There are three main classes of WCET calculation methods: *structure-based*, *path-based*, and techniques using *implicit path enumeration* (IPET). Our current implementation of timing analysis and WCET calculation is based on IPET, which is first proposed by Li and Malik [71] and is recently the most widely used method for WCET calculation.

In IPET, finding a WCET bound of a task is expressed by an *integer linear programming* (ILP) problem: the total execution time of the task is represented as a linear expression $\sum_{i \in B} x_i * t_i$, where the time coefficient t_i expresses the execution time bound of the basic block i and the count variable x_i corresponds to the number of times the block is executed. The WCET is calculated by maximizing the expression, given a set of arithmetic constraints reflecting the structure of the task and possible flows. This formula can also be extended to express more complex flows and more complex timing effects. For example, Li has made IPET capable of expressing the timing effects of out-of-order pipeline, branch predictor and instruction cache in his Ph.D. thesis [70].

Figure 7.11(c) shows the ILP problem and the constraints generated by the IPET-based WCET calculation method for the running example. Figure 7.11(b) shows the execution time bounds of all the five basic blocks. The count variables are marked in the CFG in Figure 7.11(a), where x_n expresses the number of times the block n is executed and x_{m_n} expresses how many times the control is transferred from the block m to the block n. There are three classes of constraints shown in Figure 7.11(c): The *start and exit constraints* state that the task must be started and exited once. The structural constraints reflect the possible program flow, based on the fact that a basic block is entered the same number of times as it is exited. The loop bounds obtained by flow analysis constrain the number of times the loops are executed. With these constraints, the ILP problem can be solved using an ILP solver to output a WCET bound.



```
t1 = 26;  t2 = 8; t3 = 4;  t4 = 2; t5 = 3;
```

(b) Timing Information

```
max: 26*x1+8*x2+4*x3+2*x4+3*x5;

// Start and exit constraints
x1 = 1; x4 = 1;
// Structural constraints
x1 = x1_2;
x2 = x1_2+x3_2+x5_2 = x2_5+x2_3;
x3 = x2_3 = x3_2+x3_4;
x4 = x3_4;
x5 = x2_5 = x5_2;
// Loop bounds
x3 = 9; x2 <= 54;
```

(a) Low-Level CFG **(c) ILP Problem and Constraints**

Figure 7.11: WCET Calculation Using IPET

7.6 Experiment

7.6.1 Experiment Methodology

As it is very hard (if not impossible) to find the actual WCET of a program, the most widely used way to evaluate a WCET estimation method is to compare the estimated WCET with the observed WCET measured by simulation. Knowing $t_{est_i} \geq t_{act_i} \geq t_{sim_i}$ and $x_{est_i} \geq x_{act_i} \geq x_{sim_i}$ for each basic block i, it yields:

$$Estimated\ WCET\ \geq Actual\ WCET \geq Observed\ WCET$$

Assuming that the observed WCET is very close to the actual WCET, the comparison between the estimated WCET and the observed WCET is able to show the tightness of WCET estimation. However, if an overestimation is found, it is unable to know whether this overestimation is caused by flow analysis or timing analysis.

To show only the effectiveness of flow analysis, we introduce another observed WCET: $observedWCET' = \sum_{i \in B} x_{sim_i} * t_{est_i}$. It is a combination of simulated control flows and estimated timing information. Given a set benchmarks, the worst test cases of which are known, we get: $x_{sim_i} = x_{act_i}$. Therefore, we can evaluate the quality of our flow analysis approach very accurately, by comparing $estimatedWCET$ and $observedWCET'$.

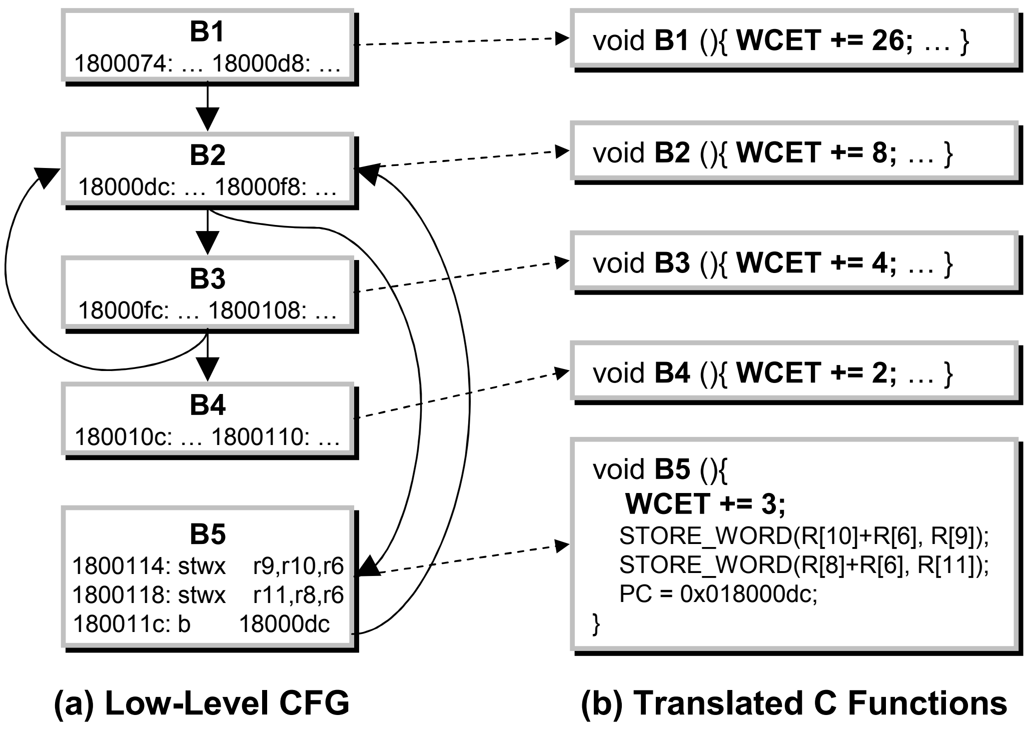


Figure 7.12: Simulation Strategy

To get *observedWCET′*, we make use of binary level simulation (BLS). In BLS, the compiled binary code of a program is translated back to C code, which has exactly the same functionality as the original program. One basic block corresponds to a translated C function, which contains both the behavioral code and the execution time bound of this basic block.

Figure 7.12 shows the five C functions translated from the five basic blocks of the running example. The execution time bounds shown in Figure 7.11(b) are annotated. The name of each function indicates the corresponding basic block. The C function corresponding to B5 is shown in detail.

The control flows between the basic blocks are constructed using a *switch* statement in a *while* loop. Each *case* of the *switch* construct corresponds to a basic block. After executing the C function corresponding to a basic block, the PC value is updated and the control is transferred to the *case* branch corresponding to the next basic block. For the shown example, the code looks as follows:

```
1: PC = 0x1800074;
2: while((PC>=0x1800074) && (PC<=0x180011c)){
3:   switch(PC){
4:     case 0x1800074: B1(); break;
5:     case 0x18000dc: B2(); break;
6:     case 0x18000fc: B3(); break;
```

```
    case 0x180010c: B4(); break;
    case 0x1800114: B5(); break;
    default: sim_end(); break;
  }
}
```

7.6.2 Experiment Results

The experiments were carried out mainly to evaluate our flow analysis approach. A PowerPC processor was chosen as the target processor. 10 programs were randomly selected from the Mälardalen WCET benchmark suite [9]. All the programs were cross-compiled using a GCC compiler with the optimization level -O2.

All the 10 programs were first converted to the corresponding ISC programs. Both WCET analysis and simulation were performed on the ISC programs. Before WCET analysis on ISC, the correctness of the ISC generation was verified by executing both the source programs and the ISC programs with a set of input data and comparing the output.

Table 7.1 shows both the estimated WCETs and the observed WCETs obtained by the proposed simulation technique. The estimated WCETs of 9 programs are exactly the same as their observed WCETs. This means that our flow analysis method has successfully extracted and transformed all the flow facts in 9 programs. The only overestimation is found by the estimate of the program *janne_complex*. In the following, we discuss about the cause of this overestimation. The source code of *janne_complex* is shown as follows:

```
while(a < 30){
  while(b < a){
    cnt_in++;
    if(b > 5)
      b = b * 3;
    else
      b = b + 2;
    if(b >= 10 && b <= 12)
      a = a + 10;
    else
      a = a + 1;
  }
  a = a + 2;
  b = b - 10;
}
```

Table 7.1: WCET Estimates (in cycles)

	Obs.WCET'	*Est.WCET*	**Difference**
ns	5230	5230	0%
matmul	129700	129700	0%
janne_complex	186	214	15%
crc	17587	17587	0%
insertsort	631	631	0%
edn	66526	66526	0%
cnt	3648	3648	0%
adpcm	330632	330632	0%
compress	4062	4062	0%
cover	2130	2130	0%

In the program, the two variables `a` and `b` are supposed to have the same value interval $[1, 30]$. Although this program has a very small code size, but has a fairly complex structure. Our flow analysis found the bounds of both the outer loop and the inner loop, which are 11 and 13, respectively. In the WCET calculation, the both bounds were used to constrain control flows. This causes the overestimation, because, in reality, there is probably no input data that leads to the maximum iterations of both the outer loop and the inner loop. Table 7.2 shows the three test cases that might lead to the WCET. None of these test cases cause maximum iterations of both the outer loop and the inner loop. Which one finally results in the WCET depends on the instruction timing. The simulation results of the three test cases show that the test case `a=1, b=7` leads to the observed WCET.

This example was originally used in [43]. However, in [43] a different result is shown. Our simulation proved the correctness of our analysis.

Table 7.2: Simulation Results of *janne_complex*

a	**b**	**max. loop_out**	**max. loop_in**	*Obs.WCET'*
1	19	11	9	162
1	7	9	13	186
1	5	10	11	174

Part III

Software Performance Estimation in a System-Level Design Flow

Chapter 8

The SysCOLA Framework

A modeling language with formal semantics is able to capture a system's functionality unambiguously, without concerning implementation details. Such a formal language is well-suited for a design process that employs formal techniques and supports hardware/software synthesis. On the other hand, SystemC is a widely used system level design language with hardware-oriented modeling features. It provides a desirable simulation framework for system architecture design and exploration. This chapter introduces a design framework, called *SysCOLA*, that makes use of the unique advantages of both a new formal modeling language, *COLA*, and SystemC, and allows for parallel development of application software and system platform. Here, the term *system platform* represents the whole platform lying under the application software layer, including a *software platform* and a *hardware platform*. The software platform contains software layers that provide services required for realizing the designated functionality. In SysCOLA, function design and architecture exploration are done in the COLA based modeling environment and the SystemC based virtual prototyping environment, respectively. The concepts of *abstract platform* and *virtual platform abstraction layer* facilitate the orthogonalization of functionality and architecture by means of mapping and integration in the respective environments. Currently, the framework is targeted at the automotive domain.

The framework was developed in the scope of the BASE.XT project, a cooperation project between Technical University of Munich and BMW Forschung und Technik GmbH, where mainly 7 Ph.D. students were involved. My own contributions to the framework include the software performance estimation tools and the virtual prototyping environment. This chapter introduces the framework as a whole. To get an overview of who has done which part of work, please refer to Section 1.1.4.

The chapter is organized as follows: first, in Section 8.1, we give an overview of design process. Section 8.2 presents some details of the SysCOLA framework. A case study of designing an automatic parking system is shown in Section 8.3. Finally, the software performance estimation techniques employed in the framework are summarized in Section 8.4.

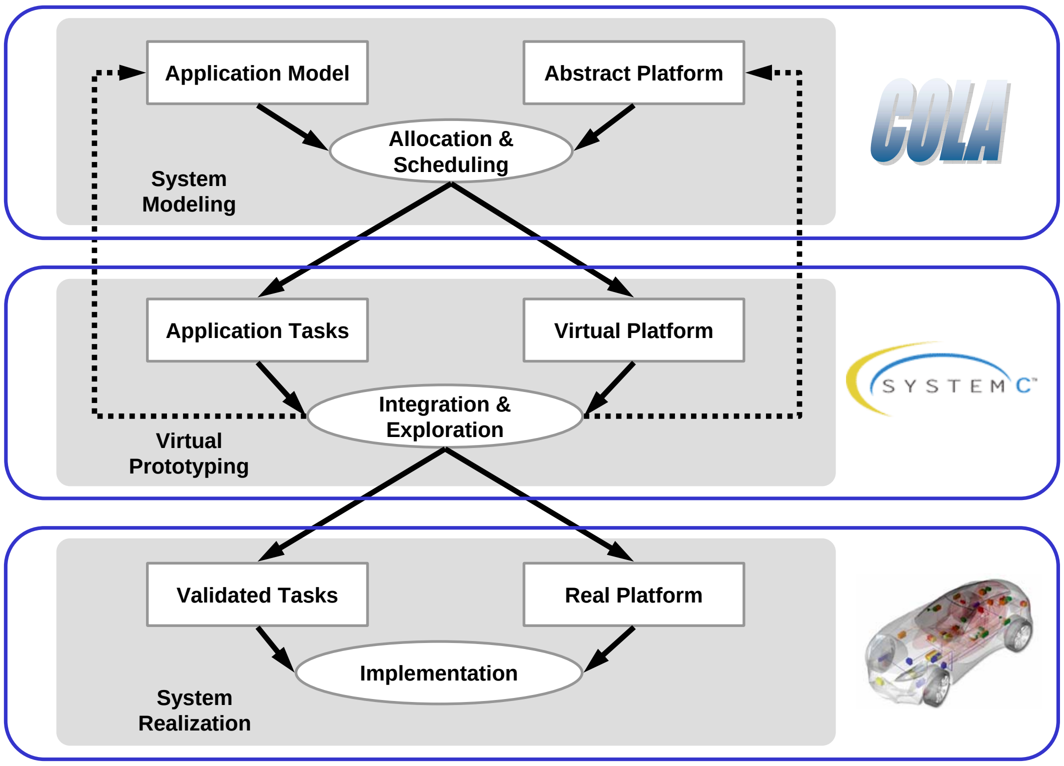


Figure 8.1: Design Process

8.1 Design Process Overview

The proposed design process is divided into three stages, namely system modeling, virtual prototyping, and system realization, as depicted in Figure 8.1. There exist different understandings of the term *virtual prototyping*. In this dissertation, it is defined as a process of generating an executable model that emulates the whole system under design. System modeling and virtual prototyping are carried out in two separate environments of SysCOLA. A newly developed formal language, *the component language* (COLA) serves as the formal foundation of the modeling environment, while the virtual prototyping environment is based on SystemC. Corresponding to the three design stages, three abstraction levels of a system platform are defined: the *abstract platform* at the model level, the *virtual platform* in SystemC and finally the implemented *real platform*.

In the modeling environment, the system's functionality is captured with an application model. The model is refined and then partitioned into *clusters*, which are tasks from an operational point of view. The system platform is modeled as an abstract platform, allowing for automatic mapping of the software clusters and automatic code generation. However, the abstract platform is specified solely with a set of resource figures and thus cannot provide an insight into the system architecture for design space exploration. Therefore, it is only used to find suitable topologies of the system platform and fix some parameters. Rather, the detailed exploration is done on a virtual platform, using a simulative approach. Different from the abstract platform, the virtual platform is an executable SystemC model that captures the dynamic behavior of the system platform under design and can execute the generated software tasks. It gives an environment for functional validation and allows early system integration as well as performance evaluation. For system realization, the real platform is implemented, according to design decisions made on the virtual platform, and the validated tasks are then executed on the finished real platform.

Within SysCOLA, the application software and the system platform are developed in parallel, in a tightly coupled loop. Software errors found by testing on the virtual platform or a change of the virtual platform architecture may lead to a change at the model level. Then, newly generated code is tested again on the virtual platform. Such iterations reduce the possibility of errors in the final system and avoid costly redesigns.

8.1.1 Related Work

There exist many high level design frameworks that cover both application modeling and system platform design, such as SystemClick [88], CoFluent Studio [4], and Metropolis [25]. These frameworks as well as ours are based on the similar ideas but are different from each other in terms of target application domains and conceptual approaches. SystemClick is specially targeted at WLAN design. Unlike COLA, the modeling language Click, used in SystemClick, defines the causal order of tasks and their resource requirements, but does not deal with specifying their implementation. Tasks are implemented by hand. CoFluent Studio and other purely SystemC-based design solutions model the system's functionality using C/C++ semantics. They rule out formal techniques and hinder automated synthesis. Simulink-based solutions like [53] have the same limitation. Among these design frameworks, the most similar one to ours is Metropolis, which supports both function and architecture modeling in an integrated environment. In contrast, we prefer separation of function design and architecture exploration in two different environments to make use of the unique advantages of a formal language like COLA and a system level design language like SystemC. The orthogonalization of functionality and architecture is achieved by means of mapping and integration in the respective environments. Also, some key concepts like abstract platform, logical communication, and virtual platform abstraction layer (VPAL) make our methodology different from Metropolis's.

The AUTOSAR project [14] also focuses on automotive systems design and has similar objectives as ours. However, it is aimed more at defining a standard for automotive systems design, while we focus on establishing a concrete design framework. When needed, SysCOLA can be adapted to generate AUTOSAR compatible designs.

8.2 The SysCOLA Framework

SysCOLA employs techniques of graphical modeling, model-level simulation, formal verification, automatic mapping and scheduling, and hardware/software code generation etc., supported by a well-integrated tool-chain in the modeling environment. Within this environment a system's functionality is modeled, verified and then realized in hardware or software. We focus more on the software-based implementation because of its flexibility and low cost. The main work in the virtual prototyping environment is to generate a virtual prototype that emulates the system under design. With such a virtual prototype at hand, the functionality of the generated tasks is validated and the design space of the system platform is explored. The concept of VPAL is introduced to facilitate the integration of the application software and the virtual platform and reduce design effort.

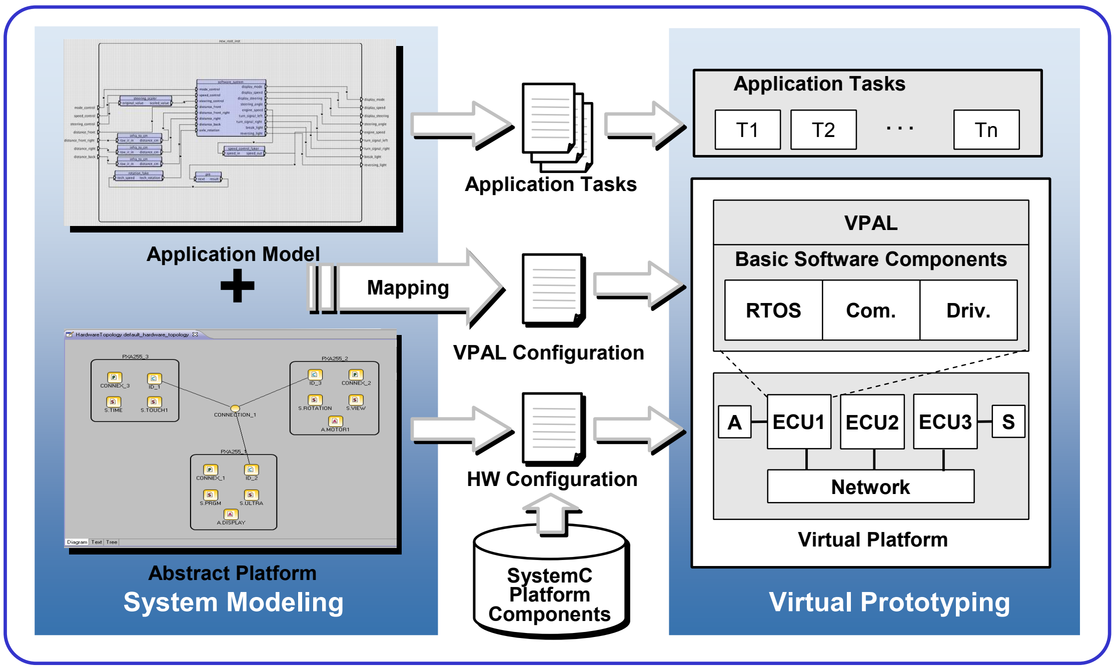


Figure 8.2: The SysCOLA Framework

Figure 8.2 shows the architecture of the SysCOLA framework. A prototype of the framework was implemented based on the Eclipse platform. As shown in Figure 8.2, the application behavior and important information captured in the COLA model are forwarded to the virtual prototyping environment in the form of application tasks and configuration files. A detailed introduction to both design environments is given in the following sub-sections.

8.2.1 System Modeling

Modeling Language and Semantics

Model-driven development, based on a formally defined language, allows for a clear and unambiguous description of a system's functionality, without concerning implementation details. To set up a modeling environment, one critical issue is to find a suitable model of computation, which can capture the target system's characteristics efficiently. We aim at automotive control systems as our target systems. *Data flow*, as used e.g. in MATLAB/Simulink, is the state-of-the-art concept for models in this domain. Unfortunately, the tools like MATLAB/Simulink lack either formally defined semantics or integrated tool support throughout the entire development process. Therefore, a new modeling language, COLA, that follows the paradigm of synchronous data flow was developed as the formal foundation of our modeling environment. COLA offers an easily understandable graphical syntax and a formally defined semantics. Being a synchronous formalism, COLA follows the *hypothesis of perfect synchrony* [30]. Basically, this asserts that computations and communication take no time. Components in a synchronous dataflow language then operate in parallel, processing input and output signals at discrete instants of time. This discrete uniform time-base allows for a deterministic description of concurrency by abstracting from concrete implementation details, such as physical bus communication, or signal jitter. We describe the modeling concept and basic units of COLA in Chapter 9. For more details about COLA syntax and semantics, please refer to our technical report [65].

Application Modeling

In COLA, design starts from modeling functional requirements in a *feature model*. The feature model is converted to a *functional model* throughout several steps of model transformation and rearrangement. In a functional model, the data flows are expressed by networks of COLA *units*. A unit can be further composed hierarchically, or occurs in terms of *blocks* that define the basic (arithmetic and logical) operations. The communication between units is realized by *channels*. In addition to the hierarchy of networks, COLA provides *automata* to express control flows in a functional model. With automata complex networks of units can be partitioned into disjoint operating modes, the activation of which depends on the input events.

COLA allows for modeling in a component-based flavor, which facilitates model reusability. The correctness of modeling and model refinement is checked by model-level simulation [94] and formal verification.

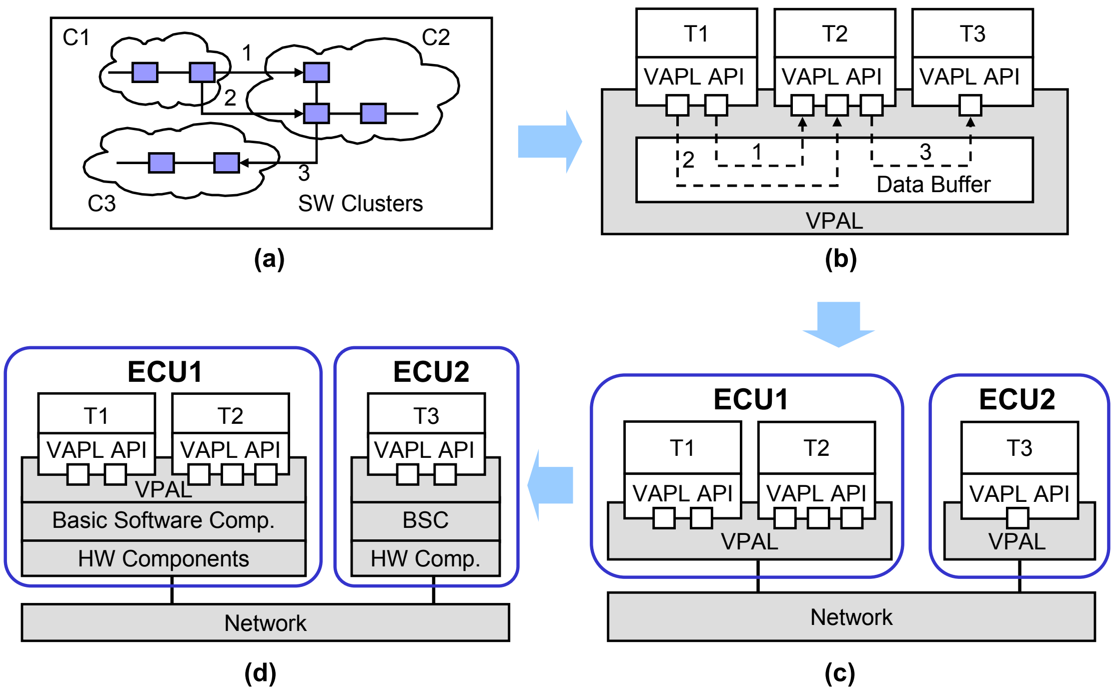


Figure 8.3: Logical Communication and Virtual Platform Refinement

Clustering and Logical Communication

Before mapping model entities to computing nodes of the platform, units of the functional model are grouped into *clusters*—the model representation of software tasks, each annotated with resource requirements on each possible processing element the task might be mapped to. Among the resource requirements, worst-case execution time (WCET) is the most important one. A clustering is derived from an optimized software architecture w.r.t. reusability, maintainability, design guidelines, documentation and others.

To enable software development independent on the system platform, we introduce the notion of *logical communication*. According to the concept of logical communication, two clusters exchange data by means of a logical address (i.e., data id) assigned to each channel between them, as shown in Figure 8.3(a). After mapped to a networked system, this logical communication is retained. The generated software tasks communicate with each other still by logical addresses, without caring whether they are located at the same node or not. The realization of the logical communication on a concrete platform is introduced later.

Automatic Mapping and Scheduling

An abstract platform is constructed using the hardware modeling capabilities of COLA. The purpose of the abstract platform is to specify the topology and capabilities of the target platform and thus give a basis for mapping of clusters. We have developed an automatic mapping algorithm, which takes the application clusters and the abstract platform as its input and tries to find a valid mapping with the resource requirements of the tasks and the platform capabilities as the foundation of its decision. We describe this mapping algorithm in [64].

The execution order of the tasks can be decided either at runtime by a dynamic scheduling algorithm or pre-runtime by a static scheduling algorithm. To guarantee that the resulting system behaves the same as modeled, we propose to use static scheduling. An algorithm for static scheduling of COLA tasks is also introduced in [64].

Code Generation

After finding a proper mapping that fulfills the requirements, C code is generated [49]. COLA also allows for hardware implementation of the modeled functionality for higher performance [99]. The code generation relies on a set of template-like translation rules. As COLA has a slender syntax, the number of translation rules is small. The translation rules are well documented in [49] and [99]. As an example, the task generated from the cluster C2 (Figure 8.3) is shown in Figure 8.4. It reads the input data from the logical addresses 1 and 2 and sends the output data to the logical address 3. The behavioral code is not shown here.

```
void T2(){
 Int in_0, in_1; Int out_0;
 receive_data(&in_0, sizeof(in_0), 1);
 receive_data(&in_1, sizeof(in_1), 2);
 ... // behavioral code
 send_data(&out_0, sizeof(out_0), 3);
}
```

Figure 8.4: An Example of Generated Code

8.2.2 Virtual Prototyping

Virtual Platform Abstraction Layer

The VPAL is aimed at wrapping the whole virtual platform. Using this layer, the virtual platform can be regarded as a functional entity that provides a set of services, from the application's perspective. The software tasks use these services by calling a fixed VPAL API. In this way, the tasks can be simulated on different virtual platforms without the need of changing any code. Only adaptation of the VPAL to the new platform is needed. To achieve a small and simple VPAL API, the VPAL provides four basic functions, as shown in Figure 8.5. The API is easily extendable. The functions *send_data()* and *receive_data()* are defined for sending and receiving data, respectively. The first argument is a pointer to a variable the data is read from or written to. The second argument represents the length of the data. The logical address is passed to the function as the third argument. We define a task as a *stateful* task, if the execution of this task depends on its history (*state*). This state data is read at the beginning of the task's execution and saved in the end, by calling *read_state()* and *save_state()*, respectively.

```
send_data(void *buf, ssize_t len, unsigned short int dataid);
receive_data(void *buf, ssize_t len, unsigned short int dataid);
read_state(void *buf, ssize_t len, unsigned short int taskid);
save_state(void *buf, ssize_t len, unsigned short int taskid);
```

Figure 8.5: VPAL API

It is one of the duties of the VPAL to retain the logical communication between software tasks on the virtual platform. It can be realized by a shared data buffer, where the tasks exchange data using logical addresses, as shown in Figure 8.3(b). To go one step further, the logical communication between tasks on different nodes is mapped to the physical communication that is emulated by a network simulation model, as shown in Figure 8.3(c). To set up exactly the same logical communication as the model level, configuration information generated from the model is used. Using this configuration information, the VPAL is able to allocate buffer space during the startup phase, initialize sensor and actuator hardware for operation, and distinguish between local and remote communications. A segment of configuration information is shown in Figure 8.6. It is used to configure the VPAL for the task shown in Figure 8.4.

```
<node id="1">
 <task id="2" name="T2">
  <data id="1" channel="c_in_0" name="in_0" type="INT" len="4"/>
  <data id="2" channel="c_in_1" name="in_1" type="INT" len="4"/>
  <data id="3" channel="c_out" name="out_0" type="INT" len="4"/>
 </task>
 ...
</node>
```

Figure 8.6: VPAL Configuration Information

System Platform Design

The system platform design can be regarded as the process of building and refining a virtual platform. It starts with defining the critical platform services in the VPAL, including communication mechanisms, device interaction, and scheduling.

These services are then refined and realized in the corresponding basic software components (BSCs), as shown in Figure 8.3(d). For example, an RTOS is chosen and modeled to take charge of scheduling and device drivers are developed to interact with devices. The VPAL lies between the application tasks and the basic software components and controls the interaction between these software components. Meanwhile, the hardware platform is designed to realize these services in hardware components including processing nodes, sensors and actuators, interconnected by a network. During the design, both functional and performance simulations are needed. Functional simulation is used to verify each change or refinement of the virtual platform, while performance simulation provides performance statistics to explore the system architecture.

To which level a platform component should be refined depends on how many details we need to explore. If a new component is to be developed, its model should be refined down to implementation. In contrast, if we decide to use an existing one, it is unnecessary to simulate all the details as accurate as the real one. To achieve low simulation complexity and high simulation speed, we just need an abstract model that captures the most important features of the existing component and is accurate enough to make important design decisions.

Functional Simulation

In SysCOLA, functional simulation can be performed early during modeling, by means of model-level simulation [94]. This model-level simulation tests a functional model before mapping and code generation and thus cannot take the underlying platform into account. Different from the model-level simulation, the simulation on a virtual platform allows for testing of both generated code and the services provided by the target platform.

Thanks to the VPAL, the details of the virtual platform are transparent from the application software's point of view, and thus the software tasks can run on the virtual platform at an arbitrary abstraction level. Validation of application software is made possible early in the design cycle, even when there is only the VPAL available. The refinement of the virtual platform can be verified by comparing functional simulation results from the refined virtual platform and an earlier one. For example, the network design can be verified by comparing simulation results from the setup shown in Figure 8.3(b) and the one shown in Figure 8.3(c).

Performance Simulation

During embedded systems design, performance is one of the most important factors to be considered. If performance requirements cannot be met, this could lead to system failure. It is necessary to handle performance issues from early design phases until system implementation.

We estimate the WCET of each task, which serves as a basis for automatic mapping and scheduling. However, these static analyses have limited ability to capture real workload scenarios and often result in an over-designed system. Therefore, we use performance simulation to validate whether the performance requirements are really satisfied and show how pessimistic the static analyses are.

At system level, we use iSciSim to generate software performance simulation models for fast architectural exploration. The iSciSim tool has been integrated into the SysCOLA framework, and thus, C code generated from COLA models is automatically forwarded to the iSciSim tool for ISC generation and timing annotation. This means that we just need one button click to generate timed simulation models from COLA. The temporal behavior of the system platform can also be abstracted with delay values, to be annotated into the functional simulation model. For example, a network model is annotated with delays, determined with respect to the bus protocol and the data length. Similarly, as already introduced in Chapter 6, each RTOS overhead due to, e.g. scheduling or context switches, is also associated with a delay. Such delay values can be obtained by means of measurement. The simulation models of these system components are linked with the simulation models of software tasks to generate a simulator of the whole system, which we call a virtual prototype. Depending on the simulation results, system architecture will then be adjusted and refined iteratively. With the stepwise refinement of the virtual platform, the granularity of the timing values can also be refined, down to cycle-accurate if necessary.

8.3 Case Study: An Automatic Parking System

The presented approach has been evaluated using a case study. As the framework is targeted at the automotive domain, we chose an upcoming feature of premium cars, namely an automatic parking system, to imitate the concerns and requirements of automotive design. We chose a model car with a scale of 1:10 as the basis for system realization.

8.3.1 Functionality and Design Space

The intended functionality includes (i) manual steering via remote control, (ii) automatic discovery and parking into a parking space of sufficient length, and (iii) ability to run alongside a, not necessarily straight, wall while going around all convexities.

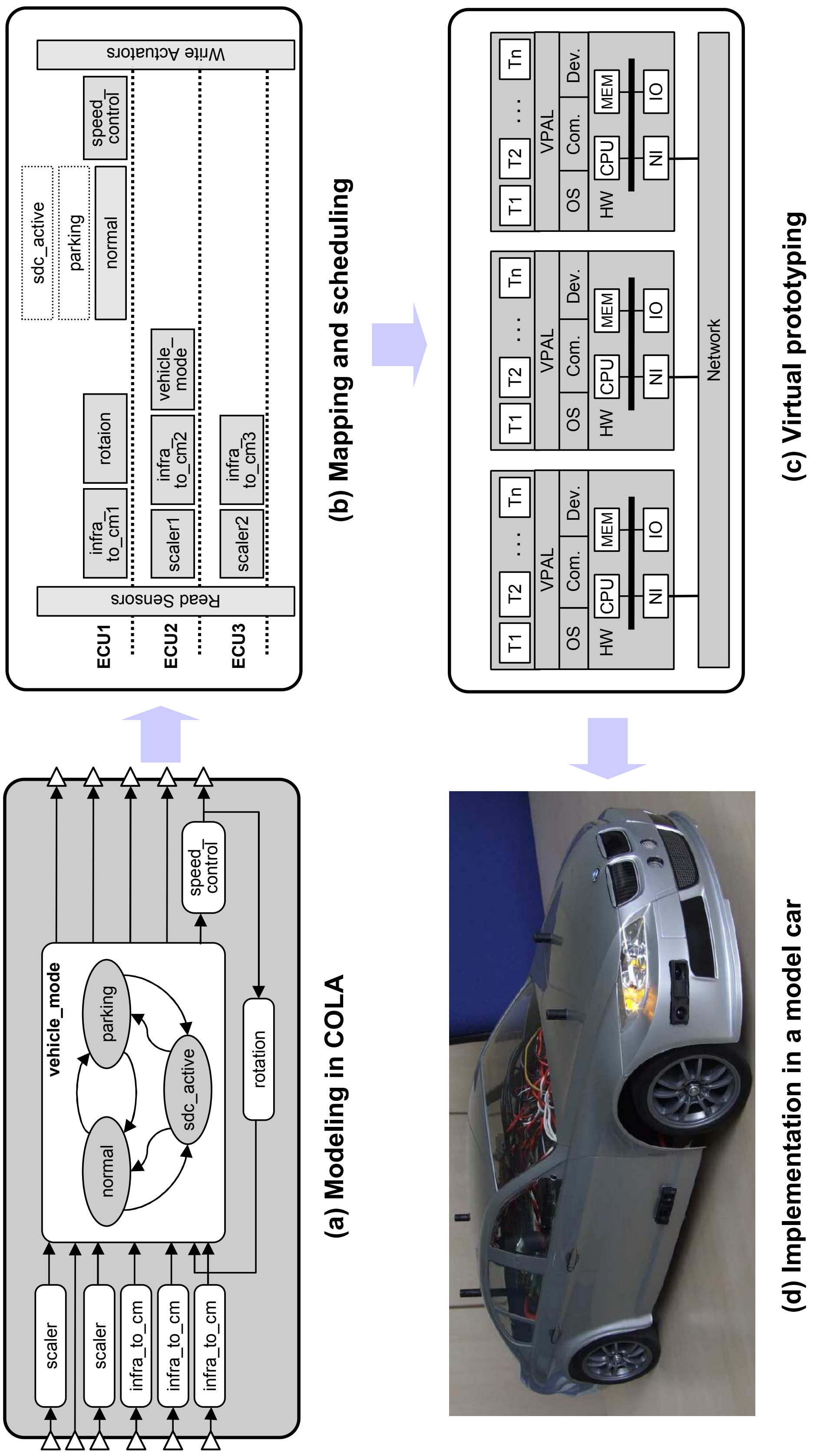


Figure 8.7: Case Study

In principle, the design space for the designated functionality could be very large. The target platform could be an MPSoC or a networked embedded system. Implementations could vary from application-specific integrated circuits (ASICs) to general purpose micro-controllers. Also, different micro-controllers, communication fabrics and BSCs are free to choose. However, in our case study, the design space was constrained by the availability of resources and our research interests. We chose Gumstix micro computers [8] as processing nodes. An Ethernet network with RTnet protocol [11] was chosen as the communication fabric to study the real-time capability of the Ethernet. RTnet is based on Xenomai [17], which implements an RTOS emulation framework on Linux. These constraints reduce the design space but do not hinder the demonstration of the key concepts. The same methodology is surely applicable for exploring a larger design space.

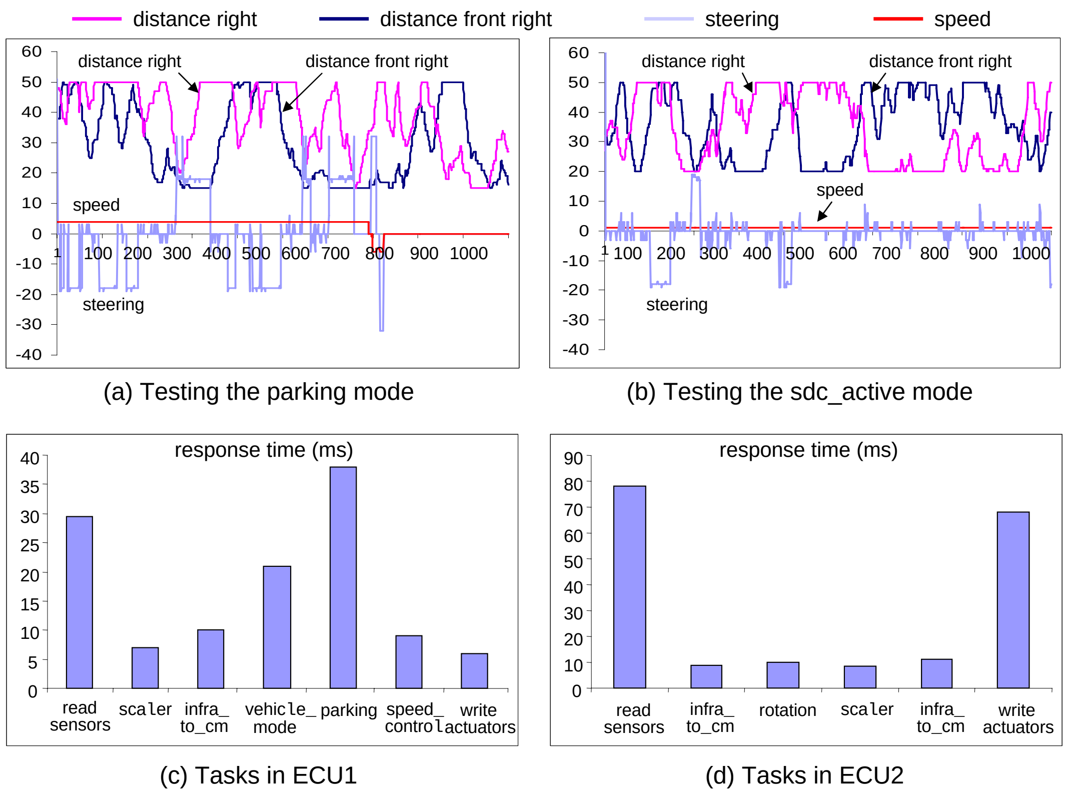
8.3.2 System Modeling

The discussed functionality was transferred into a COLA model. Eleven distributable clusters are defined, as shown in Figure 8.7(a). The system is separated into three operating modes, namely **normal**, **parking**, and **sdc_active**, each corresponding to one defined function. In the **normal** mode, the user can modify speed and direction using the remote control. The **parking** mode makes the model car run in parallel to a wall and search for a gap of sufficient size to park. In the **sdc_active mode**, the car just runs parallel to a given wall and adjusts its distance to all convexities found. The mode cluster **vehicle_mode** controls the transitions between the three operating modes, which are triggered using the remote control.

After specification of the software model and an abstract platform, the clusters were distributed onto available computing nodes and a scheduling plan was generated automatically. Figure 8.7(b) shows an exemplified result of mapping and scheduling on a network of three computing nodes. As shown, the scheduling cycle starts from reading input data from sensors and ends with writing output data to actuators. The tasks are scheduled in between, with the starting time explicitly specified. The tasks that can be executed in parallel are mapped to different nodes. Whereas, **normal**, **parking**, and **sdc_active** must be delayed until the mode decision is made by **vehicle_mode**. As they execute exclusively, they are mapped to the same node. In the same way, mapping and scheduling were repeated for two-node and four-node systems. At last, software tasks and configuration files were generated from the model.

8.3.3 Virtual Prototyping

As the design space was narrowed down by the constraints discussed before, the work in this phase was significantly eased, including: (i) refining VPAL services



(a) Testing the parking mode

(b) Testing the sdc_active mode

(c) Tasks in ECU1

(d) Tasks in ECU2

Figure 8.8: Simulation Results

down to implementation, (ii) writing device drivers, (iii) deciding the number of computing nodes, (iv) system integration, (v) functional validation and performance evaluation.

By integrating the generated code into a virtual platform, an executable virtual prototype was generated for functional validation. By the time of functional validation, the virtual platform was at a high level of abstraction, containing the VPAL and the functional models of RTOS, devices and network. Figure 8.8(a) and (b) show the exemplified simulation results. The two simulations, each running with 1000 test cases, took only 1.92 and 1.98 seconds, respectively, with useful logging information generated for error detection. The first simulation tested the system in the **parking** mode, while the second one tested the system in the **sdc_active** mode. The distances from the wall to the right side and the front-right side of the car were calculated using randomly generated data representing the sensor data. Both the parking and steering algorithms depend on the two distances. The distances from the car to the front, back and left obstacles were set constants in the value range of safe distances. In both figures, we can see the changes of the steering value with respect to the distance values. In Figure 8.8(a), we can also see the parking behavior after around 700 simulation steps.

The parking system must react to the environment fast enough, so we defined that the scheduling cycle could not extend 200 ms. To decide the number of

ECUs needed to satisfy this performance requirement, the virtual platform was refined for fine-grained timing annotation and performance simulation. To connect all the sensors, at least two ECUs were needed. Figure 8.8(c) and (d) show the respective maximal response times of all the tasks mapped to two ECUs, obtained by simulation with 10000 test cases. The simulation took less than one minute. According to the simulation results, the scheduling cycle was adjusted, which resulted in a cycle length of 238 ms. In the same way, the systems with 3 ECUs and 4 ECUs were evaluated and resulted in cycle lengths of 178.5 ms and 145 ms, respectively. The increase of the number of ECUs cannot improve the performance anymore, because there are five tasks that must run sequentially due to data dependencies. For higher performance and lower cost, these five tasks should be allocated to an ECU with higher computing power, while the other tasks can run on cheaper ECUs. Another solution is to reduce the overhead of the basic software components. According to the exploration result, we decided to implement the three-node system. After all the high-level design decisions were made, the VPAL was refined and implemented as a middleware. All the device drivers were also implemented.

8.3.4 System Realization

For system realization, three Gumstix micro-computers were embedded into the model car shown in Figure 8.7(d). The infrared and supersonic sensors were used for distance sensing and connected via serial and I2C buses, respectively. A cellular phone was connected via bluetooth and served as the user interface for remote control. Control of steering, motor, and light system was realized by an interface board, equipped with a serial interface. Executing the application code, the middleware and the basic software components on the real platform confirmed the simulation results obtained in the virtual prototyping phase.

8.4 Summary of the Employed Software Performance Estimation Techniques

Important performance issues along with the process are shown in the right-hand side of Figure 8.9. Among them, the software performance estimation techniques are discussed in the following.

8.4.1 Model-Level Performance Estimation

The performance estimation, early at the model level, is aimed at determining the approximate temporal behavior of the system by associating each COLA unit

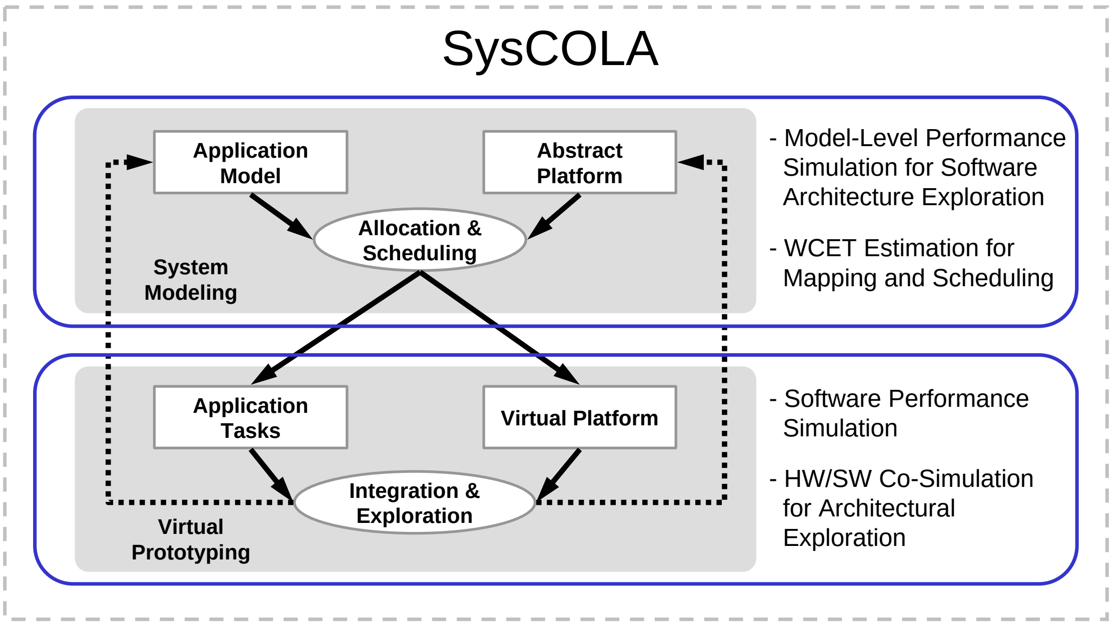

SysCOLA
Application Model
Abstract Platform
Allocation & Scheduling
System Modeling
- Model-Level Performance Simulation for Software Architecture Exploration
- WCET Estimation for Mapping and Scheduling
Application Tasks
Virtual Platform
Integration & Exploration
Virtual Prototyping
- Software Performance Simulation
- HW/SW Co-Simulation for Architectural Exploration

Figure 8.9: The Performance Estimation Techniques Applied in the Design Framework

with a timing value, which is obtained by means of statistical measurements. It can help in optimizing software architecture and making initial decisions regarding hardware/software partitioning. It is especially useful when the application model is still under development and C code cannot be generated.

Because COLA models are inherently abstract and cannot be executed immediately. SystemC is chosen to serve as a back-end simulation framework, because of its similar syntactic structures to COLA's. We established an automated translation procedure from COLA models to SystemC. Please note the difference between SystemC based model-level simulation and SystemC based virtual prototyping. The latter is used to simulate the whole system, executing application code generated from COLA, while model-level simulation is performed before code generation for some initial decisions in an early design phase.

8.4.2 WCET Estimation

After getting an initial partitioning of the software architecture, software tasks are generated. Since SysCOLA is targeted at the automotive domain, where most applications are real-time applications, knowing the WCET of each task is mandatory for task allocation and schedulability analysis.

We apply the static analysis technique introduced in Chapter 7 for WCET estimation. As introduced before, static WCET analysis consists of three phases: flow analysis, timing analysis, and WCET calculation. So far, we focused more on flow analysis. For timing analysis, we established only a very simple model that does not take into account the complex timing effects, such as the timing effects of out-of-order pipeline, cache, and branch predictor. It is our future work to improve timing analysis in our WCET tool. WCETs estimated by the current tool cannot guarantee the safety. The mapping and scheduling based on such WCETs is probably inaccurate. In this situation, it is especially important to validate the design using simulative methods.

8.4.3 Software Performance Simulation

After software tasks are mapped to the available resources and a suitable scheduling is found, accurate HW/SW co-simulation is performed for design space exploration and design refinement. A virtual platform in SystemC captures the dynamic behavior of the system platform under design and can execute the generated software tasks. For fast software performance simulation, we integrate iSciSim in the framework to generate software simulation models directly from COLA models.

Both WCET estimation and software performance simulation have already been described. Model-level simulation of COLA is presented in the next chapter.

Chapter 9

Model-Level Simulation of COLA

As introduced in the last chapter, COLA is the formal foundation of the SysCOLA design framework. It offers a graphical representation to specify the functional behavior of the modeled application. To take the resulting models to the designated hardware platform, automated code generation is supported by our established toolchain.

Still, not only execution on the target hardware is sought after, but also *model-level simulation* plays an important role in the design process. Model-level simulation is aimed at support for early error detection and performance analysis. Following the nomenclature described in [82], this allows for behavioral simulation based on the logical architecture, without knowing the technical architecture a priori. This is important as the latter may not yet be available in early process stages. Furthermore, once the target platform of the application is known or shall be chosen among several alternatives, performance analysis based on model-level simulation is also possible. This is realized either by annotating timing information to corresponding software models or by combining software models and hardware models.

However, COLA models are inherently abstract and cannot be executed immediately. This calls for a translation process to translate COLA models to executable models supported by a suitable simulation framework. SystemC is the one that appears to be the most suitable for our purpose. It closely matches the common syntactic elements of many component-based modeling languages, including components, ports and hierarchical composition. It has been made use of, e.g., in the analysis of AUTOSAR models in [61]. As an SLDL (system level design language) SystemC has become a standard in system level design [55].

We established an automated translation procedure from COLA models to SystemC to enable model-level behavioral simulation. The translation requires not only a close match of the syntactic structure of SystemC- and COLA models, but also the preservation of the COLA's formal semantics in SystemC models. This implies a methodology for modeling the zero execution and communication time assumption of such a synchronous language.

This chapter is organized as follows: Section 9.1 gives a brief introduction to COLA and SystemC. Following this, Section 9.2 contains a detailed description of the translation procedure from COLA to SystemC and a brief introduction to the simulation using SystemC. In Section 9.3 a case study from the automotive domain is considered to validate the proposed approach.

9.1 Overview of COLA and SystemC

The key concept of COLA is that of *units*. They can be composed hierarchically, or occur in terms of *blocks* that define the basic (arithmetic and logical) operations of a system.

Each unit has a set of typed *ports* describing the interface, which form the *signature* of the unit. Ports are categorized into input and output ports. Units can be used to build more complex components by building a *network* of units and defining an interface to such a network. *Channels* are used to connect the sub-units in a network.

In addition to the hierarchy of networks, COLA provides a decomposition into *automata* (i. e., finite state machines, similar to Statecharts [31]). If a unit is decomposed into an automaton, each state of the automaton is associated with a corresponding sub-unit, which determines the behavior in that particular state. This definition of an automaton is therefore well-suited to partition complex networks of units into disjoint *operating modes* (cf. [27]), whose respective activation depends on the input signals of the automaton.

A further introduction to individual COLA elements is given in the next section, based on their graphical representations. For more information about COLA's syntax and semantics refer to [65].

SystemC supports system modeling at different levels of abstraction, from system level to register-transfer level. Essentially, it is a C++ class library featuring methods for building and composition of SystemC elements. In order to model concurrent system behavior, SystemC extends C++ with concepts used by hardware modeling languages, like VHDL and Verilog, making use of the notion of *delta-cycles*. A delta-cycle lasts for an infinitesimal amount of time, such that no simulation time advances when processing delta-cycles (*zero-delay semantics*). Thus the processes triggered by events that occur in the same sequence of delta-cycles are said to run simultaneously. The simulation time can be advanced either by triggering processes with events ordered by time or by suspending processes for an amount of time.

The basic building blocks within a SystemC design are *modules*. Each module contains ports, through which the module communicates with its environment, and at least one process to describe the behavior of the module. A SystemC module can consist hierarchically of other modules connected by channels.

There are two kinds of processes in SystemC, namely *method processes* and *thread processes*. Once triggered, a method process cannot be suspended, whereas a thread process may be suspended by some special operations, e.g., accessing a full (empty) FIFO with blocking writes (reads). More often, *wait()* statements are used to suspend thread processes, in order to advance simulation time by waiting for a defined amount of time or for some events to happen. As *wait()* statements in concurrently running modules advance simulation time in parallel, they are well-suited to describe a system's temporal behavior.

9.2 COLA to SystemC Translation

There are obvious similarities between COLA units and SystemC modules in terms of their structure. Both contain input ports, output ports and an implementation of the intended functionality. A higher level element may be composed hierarchically of several such elements which are connected by channels. The simulation semantics of SystemC is also able to retain COLA semantics. In this section, we describe how COLA semantics is followed by SystemC and present in detail the mapping between COLA and SystemC elements. Following this, a brief introduction to simulation using SystemC is given.

9.2.1 Synchronous Dataflow in SystemC

Being a synchronous formalism, COLA asserts that computation and communication occur instantly in a system, i.e., take no time. The COLA components then operate in parallel at discrete instants of time, only constrained by the causality induced by data dependencies. This behavior of COLA designs can be efficiently modeled in SystemC by means of its delta-cycles (i.e., zero-delay) mechanism (cf. Section 9.1). Nevertheless, effort must be spent on mapping COLA channels onto SystemC elements. Since SystemC is a modeling language in a discrete event-driven paradigm, the primitive channels in SystemC have specific events associated with updating of the channels. In COLA, however, communication has no notion of events. Channels propagate data from source ports to destination ports without delay. Therefore, data going through paths of possibly different lengths arrive at the destination ports of a unit at the same time.

As the first attempt, we mapped COLA channels onto SystemC signal channels, each having an event associated with a change in its value. The process sensitive to this event is triggered whenever one of the channels bound to input ports of the module is updated. This results in the module being triggered more than once in each computation, if the channels are not updated in the same delta-cycle. Therefore, the COLA semantics cannot be followed exactly. One feasible solution to this problem is to model COLA channels by *one-stage FIFO channels*. We call

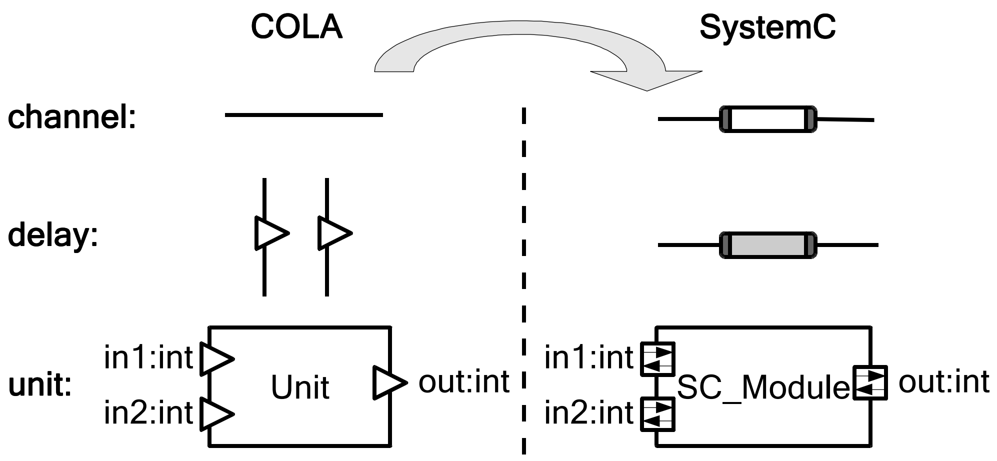


Figure 9.1: SystemC Modules Corresponding to the COLA Channel, Delay and Unit

the one-stage FIFO channels *FIFOs* for short. A FIFO has an event associated with the change from being empty to having data written to it, and another with the change from being full to having data read from it. In order to fulfill the causality requirement of COLA models, the FIFOs are accessed by *blocking* reads and writes. A blocking read will cause the calling process to suspend, until data is available, if a FIFO is empty. Likewise, a process will also suspend, if it accesses a full FIFO with a blocking write. In this way, the SystemC process that realizes the functionality of a COLA unit can perform computation only after all the FIFOs bound to its input ports are full.

9.2.2 Translation Rules

In this sub-section, we take a closer look at individual COLA elements. We give a description of each element and illustrate its mapping to SystemC.

Channels and Delays

Graphically, a COLA *channel* is simply represented by a line (cf. Figure 9.1) connecting the ports in question. As discussed previously, we map a COLA channel onto a SystemC FIFO, which is represented by a tube with the two ends in dark gray depicting interfaces of the FIFO.

In the current version of COLA the *delay* is the only element whose behavior is dependent on time. It is used to delay the dataflow for one tick. Intuitively this is a representation of memory, which is initialized with a given default value. At the first evaluation of the delay, the default value is written to the output port and the value present at the input port is used as the new internal state. In all further steps, the internal state is written to the output port and again the input port value is stored as the new internal state. A COLA delay is represented by two vertical lines drawn in parallel. It has exactly one input and one output port, represented by the triangles on the vertical lines. Modeled by a SystemC FIFO that is initialized with a default value before simulation, the semantics of the delay can be preserved. We call such a initially filled FIFO a *DFIFO*. The body of its graphical notation is colored in light gray, as shown in Figure 9.1.

The following simple example shows how a DFIFO models a COLA delay (Figure 9.2). The SystemC FIFOs' names are constructed using a combination of the prefix fifo_ and the corresponding channel name. Because the data in the DFIFO is already available at the beginning of simulation, the module is triggered once fifo_c1 is filled. At the first computation, the default data in the DFIFO is read by the process of the module. The resulting data is then output to both fifo_c2 and the DFIFO. The data stored in the DFIFO is then *delayed* to be used in the next step of computation.

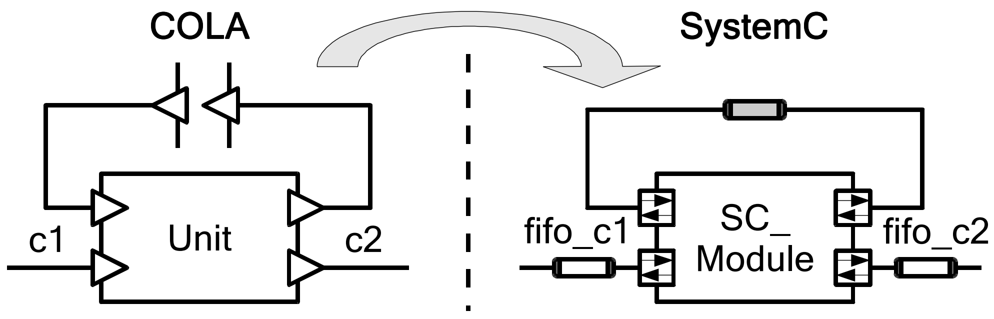


Figure 9.2: Example: Modeling a COLA Delay with a DFIFO

Units and Blocks

Units are the abstract computing elements within COLA. Each unit defines a relation on a set of input and output values. A unit is denoted by a box labeled with the identifier of the unit (cf. Figure 9.1). Its ports are represented by triangles on the border. When a COLA unit is mapped onto a SystemC module, its input and output ports correspond exactly to the respective input and output ports of the SystemC module and its functionality is described by a *thread process*. The generated SystemC code that represents the COLA unit depicted in Figure 9.1 is given in Figure 9.3.

Blocks are units that cannot be further decomposed. They define basic computational behaviors. Examples of blocks include arithmetic operations, logical operations, and constants. The graphical notation of a block is distinguished from those of the other units by drawing a black triangle in its upper right corner (cf. Figure 9.4). For each block we do not generate a SystemC module but generate one code line in the process of the module that includes the block. If several blocks are interconnected, the generated code is *inlined*. Figure 9.4 shows

```
SC_MODULE(module_name){
     // ports
     sc_fifo_in<int> in1;
     sc_fifo_in<int> in2;
     sc_fifo_out<int> out;
     ... // channels
     // the thread process that realizes the functionality
     void func_imp(){
          while(1){
               // blocking read
               int temp_in1 = in1.read();
               int temp_in2 = in2.read();
               ...
               // blocking write
               out.write(...);
          }
     }
     SC_CTOR(module_name){
          ... // body of constructor:
          SC_THREAD(func_imp); // declare the process
     }
};
```

Figure 9.3: SystemC Module Definition for a COLA Unit

an example of a composite unit that is composed of a set of blocks. The generated code describing the blocks' behavior is:

```
fifo_c5 = fifo_c1*(-1)/20;
```

As inlining is applied here, the FIFOs that interconnect the blocks are not specified.

Networks and Automata

A composite COLA unit can be either a *network* or an *automaton*. A network contains sub-units connected by channels. It is used to describe data flow. Figure 9.5 illustrates the mapping of a COLA network, which is composed of three sub-units and a delay, to a SystemC module.

Control flows are modeled using automata in COLA designs. The states of automata are also referred to as *operating modes*. Each state of an automaton represents an operating mode and is associated with a behavior. For each state a sub-unit is given to realize the state-associated behavior and computes the output

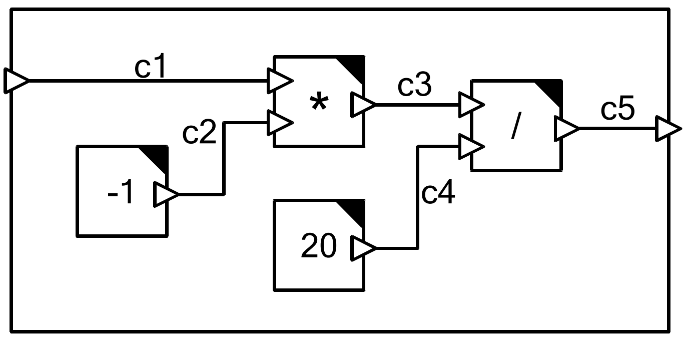


Figure 9.4: A Network Composed of Blocks

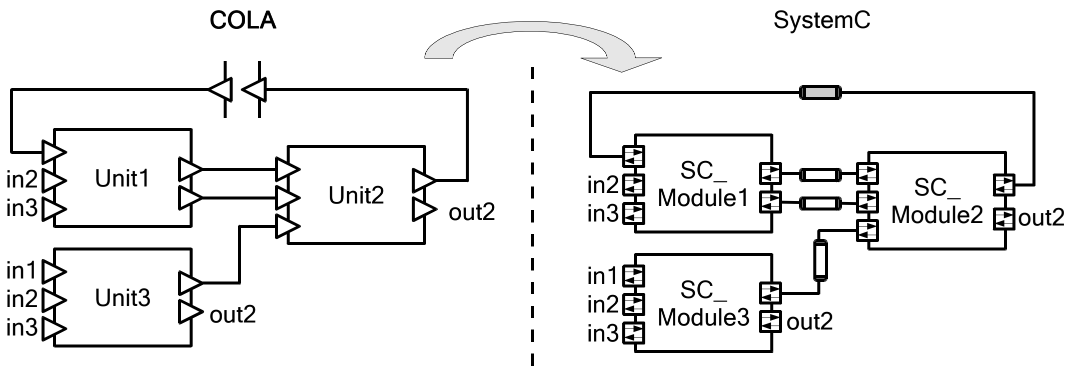


Figure 9.5: Mapping from a COLA Network onto a SystemC Hierarchical Module

of the automaton. There is only one active sub-unit in an automaton, namely the sub-unit which corresponds to the enabled state of the automaton. The passive sub-units freeze, i.e., retain their current state. The transitions between the states are guarded by *predicates*.

The mapping of a COLA automaton to a SystemC module follows the semantics described above. In the SystemC module, computation and communication are divided into several paths. Each path is associated with a state of the automaton to be modeled. Based on predicates, the flow of data is redirected to the sub-module implementing the enabled state. Figure 9.6 illustrates an example of a COLA automaton with two states, two input ports and one output port. The activation of the path depends on whether the input p is equal to *1* or not. All the input data are then forwarded to the active sub-module. The *switch structures* (in gray) drawn in the SystemC module are only for clarity. The real code of the process realizing the behavior of the given automaton is presented in Figure 9.7.

9.2.3 Simulation using SystemC

After an automated mapping as described above has been established, simulating realistic scenarios still requires a proper definition of external stimuli (environmental/user events). SystemC offers the flexibility to either specify when and

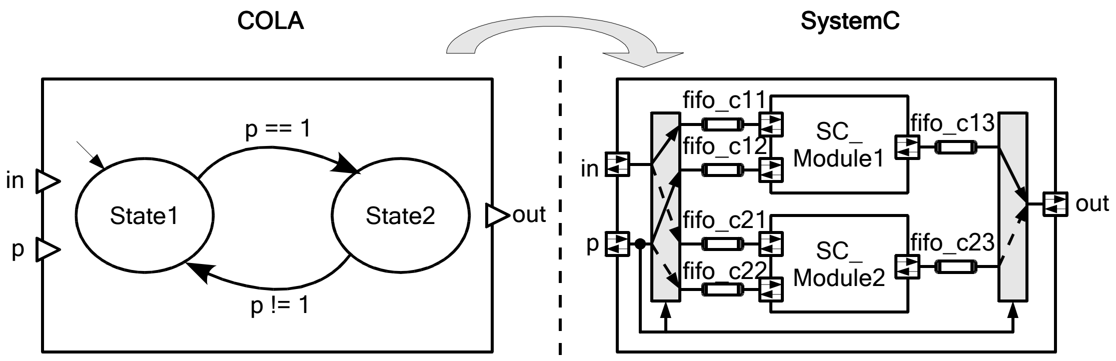


Figure 9.6: Mapping from a COLA Automaton to a SystemC Hierarchical Module

```
void automaton_imp(){
    int atm_state = 1;
    while(1){
        int temp_in = in.read();
        int temp_p = p.read();
        switch(atm_state){
            case 1:
                if((temp_p == 1)){
                    atm_state = 2;
                    fifo_c21 = temp_in;
                    fifo_c22 = temp_p;
                    out.write(fifo_c23.read());
                    break;
                }
                fifo_c11 = temp_in;
                fifo_c12 = temp_p;
                out.write(fifo_c13.read());
                break;
            case 2:
                ...
        }
    }
}
```

Figure 9.7: Part of Generated Code for the Given Automaton

which data is *measured* or simulate the interactions between the system and its environment. In this way the functionality of the system can be validated without knowing the target platform.

For the time being, the hardware platform related issues have not been taken into account. The whole system is modeled as concurrently running modules connected by FIFOs which implement the point-to-point communication scheme. Both computation and communication are modeled at the untimed level.

Once the target platform is known, approximate performance simulation can be performed at a high abstraction level. It is achieved by annotating timing information to software components and replacing the untimed FIFOs with timed FIFOs to take communication time into account. Such a performance simulation can help in making early decisions regarding task allocation and hardware/software partitioning. COLA units are annotated with their respective timing information and stored in a repository, to facilitate reuse. We propose two ways to get the approximate timing information. The first way is to run C code that corresponds to individual components on processor simulators. The execution times are measured for best, worst and/or average cases with the most widely used processors as targets. The second way is to annotate timing-related properties which are defined general enough for all the processors. For example, the number of COLA blocks and sub-units included in a COLA network can be put in relation to the execution time of the network, since basic operations and function calls will be generated for COLA blocks and sub-units, respectively, during C code generation. Later, each timing-related property will be associated with a timing value, depending on the implementation of the code generator, the compiler and the target processor. Because the tasks mapped onto different processing elements may run concurrently while all the sub-modules of a task must run sequentially, we define a global variable to aggregate the execution times of the modules contained in the same task and make use of *wait()* statements to represent timing of concurrent tasks.

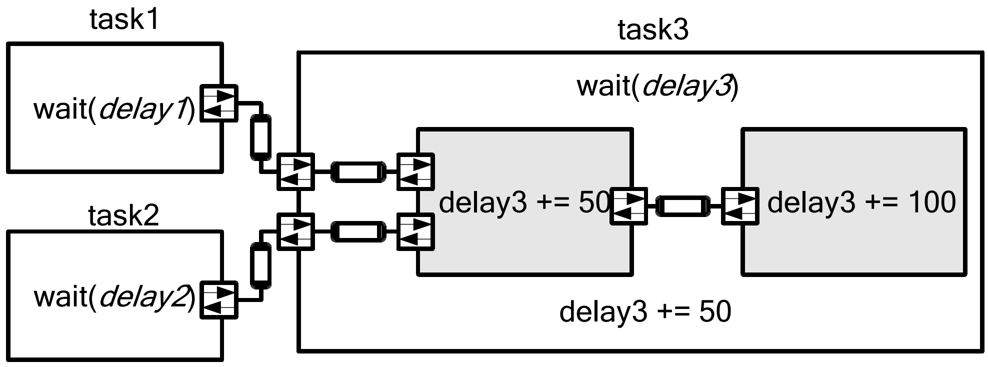


Figure 9.8: Timing Annotation

Consider the example shown in Figure 9.8, where an application with three tasks is illustrated. Suppose that a task is suspended until all its inputs have arrived.

The module that corresponds to task3 has two sub-modules, annotated with the respective execution times. Because the two sub-modules run one after the other, their execution times are aggregated using a global variable *delay3*. At the task level, three tasks are annotated with *wait()* statements. As task1 and task2 run concurrently, the advance in simulation time is the maximum of *delay1* and *delay2*. Once the two tasks finish, task3 is triggered. Its delay is the sum of the top-level module's and its sub-modules' computation times, with the communication time between the sub-modules ignored. Therefore, in the example, task3 delays for 200 time units.

9.3 Case Study

We now show how the proposed simulation approach can be applied, using a case study taken from the automotive domain. The modeled adaptive cruise control (ACC) enables cars to keep their speed at a constant value set by the driver, while maintaining a minimum distance to the car running ahead. This example is an imitation of the concerns and requirements of automotive design, and does not represent a real set of control algorithms for an actual product or prototype.

When implementing such a control system, usually an informal specification in natural language is given. The intended functionality of the ACC is described as follows. When the ACC is turned on, the speed and distance regulation is activated. This includes the measurement and comparison of the pace set by the user and the actual measured car velocity. If the desired user speed differs from the actual speed, the value for the motor control is corrected accordingly. This regulation is used as long as no object is detected within a safety distance ahead of the car. We chose 35 units for our example. If the distance drops below this threshold, the actual speed is continuously decreased by 5 percent. The minimum distance allowed is set to 15 units. If the actual distance is below 15 units, the car must perform an emergency stop.

According to this specification, the system was modeled using COLA. The design was then simulated at model level for both functional validation and performance evaluation.

9.3.1 The COLA Model of the Adaptive Cruise Control System

The top-level network, i.e., the COLA system representing the ACC model, is shown in Figure 9.9. The main components are the user interface (net_ui), which realizes the control actions, the display (DEV_A_DISPLAY), the computation of the actual speed (net_rotation), the distance sensing (net_radar), the connection to the engine (DEV_A_MOTOR), and the main control code (net_acc_on_off).

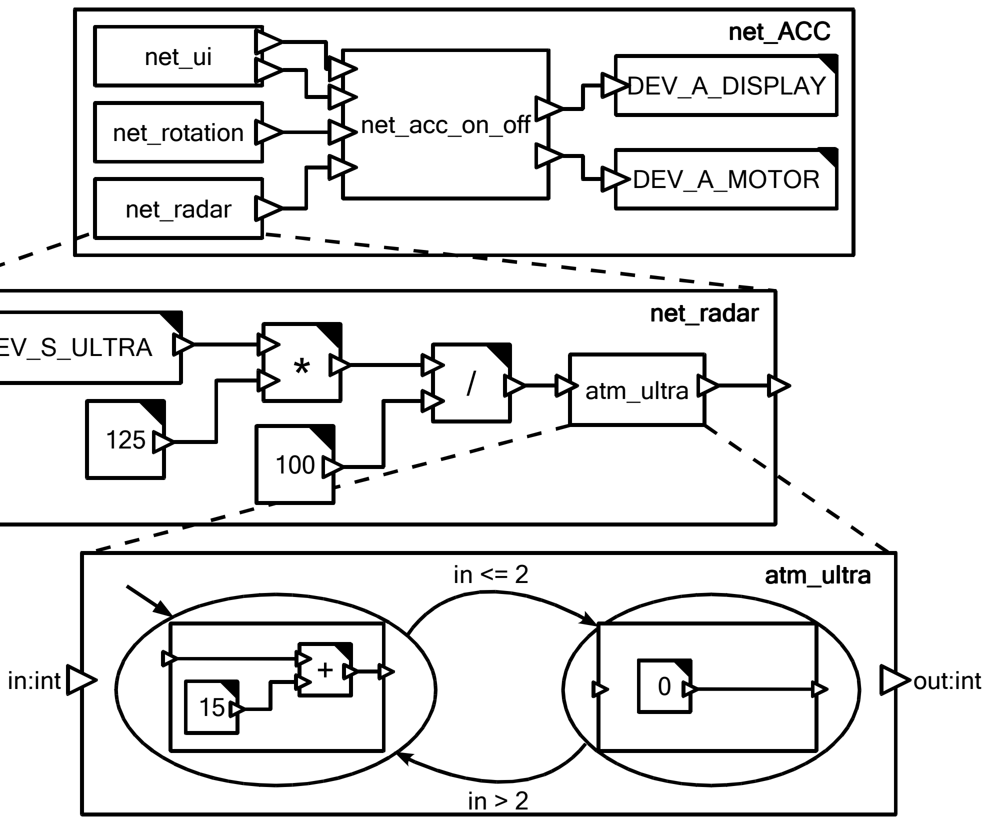


Figure 9.9: ACC Top-Level View and Decomposition of the Component *net_radar*

As an example, we present the decomposition of net_radar in Figure 9.9. The network is implemented by constants and basic arithmetic operations on data provided by the ultrasonic sensor (DEV_S_ULTRA). The interface of the network consists of a single output port, whereas all sensor specific data manipulations are encapsulated within this network. The characteristics of the employed hardware require further computation, performed within an automaton (atm_ultra). Depending on whether the value resulting from the post-processing of the data provided by the ultrasonic sensor is greater than 2 or not, either the constant value 15 must be added, or the constant value 0 is returned. This function is implemented using an automaton with two states. For brevity, in Figure 9.9 the sub-units are drawn inside the respective states. In a similar manner, the other components of the ACC can be decomposed.

9.3.2 Simulation using SystemC

The complete COLA model was mapped onto SystemC for simulation (cf. Figure 9.10). The test data were defined before simulation, except for the test data of the rotation sensor. As shown in the figure, we add a module rotation_gen to simulate feedback between motor and rotation in order to generate more realistic test data. rotation_gen delays the motor's speed and converts it into appropriate rotation data, which will be forwarded to the rotation sensor.

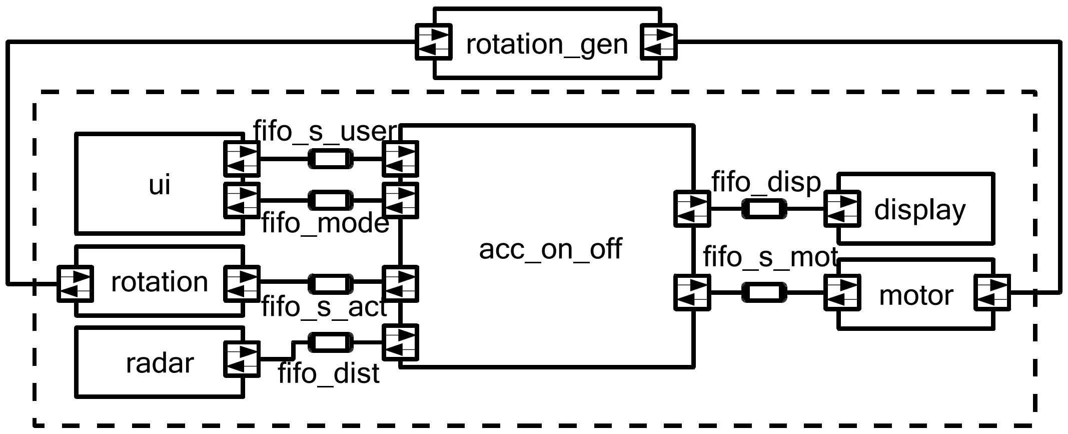


Figure 9.10: The ACC Modeled Using SystemC

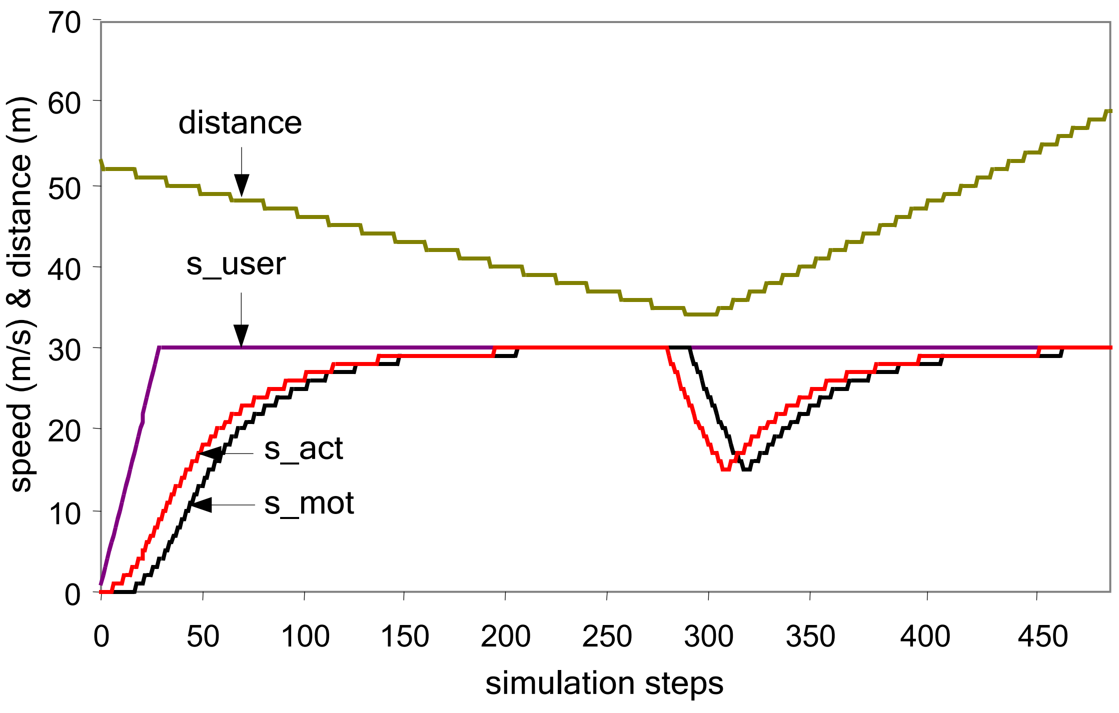


Figure 9.11: Simulation of the Modeled ACC

In Figure 9.11 the results of simulating the SystemC model are displayed. The simulation is run for 500 steps in this example. The ACC is enabled permanently during the simulation. The diagram features the intermediate data as well as the output value. The motor speed (s_mot) is the only output value shown in the diagram. Intermediate data include the desired speed (s_user), the actual speed (s_act) and the distance (distance) that are generated by ui, rotation and radar, respectively. s_user is increased or decreased by 1 in each simulation step, controlled by two *triggers* in ui. In the diagram, s_user is increased from 0 to 30 during the first 30 steps. distance is defined arbitrarily in the example. It decreases linearly during the first 300 steps and increases during the last 200 steps. s_act is calculated interactively using the data fed back from the motor. The delay in the increase of s_mot, visible in the diagram, results from the soft start functionality of the ACC. s_mot is reduced in some steps when distance is lower than 35 and increases smoothly again after distance exceeds 35 units. As can be seen, the functional behavior of the designed system can be monitored well using SystemC.

Further, the performance simulation with a PowerPC processor as target was performed. We assumed that the whole application is mapped to this single processor. The observed worst case execution times of the top-level components are given in Table 9.1.

Table 9.1: Timing Simulation Results

application components	**ET (cycles)**
net_ui	235
net_rotation	75
net_radar	141
net_acc_on_off(acc off)	96
net_acc_on_off(acc on, distance $<$ 35)	698
net_acc_on_off(acc on, 15 $<$ distance $\leq$ 35)	568
net_acc_on_off(acc on, distance $\leq$ 15)	342

Part IV

Summary

Chapter 10

Conclusions and Future Work

This thesis is focused on software performance estimation methods for system-level design (SLD) of embedded systems. We studied both software performance simulation and WCET estimation for simulative and analytical design space exploration methods, respectively. Software performance simulation is the main focus. To say more precisely, about 60% of effort was dedicated to software performance simulation. We proposed two software performance simulation approaches, called SciSim and iSciSim. They generate high-level software simulation models by annotating source code and intermediate source code (ISC) with timing information. For multiprocessor simulation, both can generate software TLMs in SystemC. We proposed an abstract RTOS model in SystemC to schedule the execution of software TLMs. In our work on WCET analysis, we have attacked the problem of flow analysis. We proposed to perform flow analysis on ISC to get an efficient WCET estimation approach. In addition, we presented a system-level design framework for automotive systems, called SysCOLA. We showed how the introduced software performance estimation methods are used in SysCOLA.

So, the work can be divided to three parts: software performance simulation, WCET estimation, and the work done in the scope of SysCOLA. In the following sections, the three parts of work are summarized and their respective future works are outlined.

10.1 Software Performance Simulation

10.1.1 Conclusions

For design space exploration of a complex embedded system, simulation is one of the most efficient ways to capture real workload scenarios and get dynamic performance statistics. As the importance of software and its impact on the overall system performance are steadily increasing, an efficient technique for fast and accurate software performance simulation is especially important. Accurate software performance simulation can help study the influence of software on the whole system performance and make decisions of hardware/software partitioning, task mapping and scheduling in early design phases. The simulation must also be fast enough for iterative design space exploration loops. Therefore, it is important to concern the trade-off between simulation accuracy and simulation performance while choosing a simulation tool. Besides simulation accuracy and performance, low complexity is also an important criterion to evaluate a simulation technique. A good simulation technique should be easy to realize and to use.

In this thesis, we focused only on execution-based simulations, which can capture both functional and temporal behaviors of systems. We introduced four execution-based simulation strategies and provided a discussion about their advantages and limitations. The four simulation strategies are interpretive instruction set simulation (ISS), binary level simulation (BLS), source level simulation (SLS) and intermediate representation level simulation (IRLS). In the four simulation techniques, ISS allows for the most accurate simulation but has the lowest speed and the highest complexity. BLS offers up to two orders of magnitude faster simulation than ISS. This speedup is achieved by the idea of performing time-consuming instruction fetching and decoding prior to simulation. BLS has moderate complexity and simulation speed. Its simulation accuracy is high enough for design space exploration. Both SLS and IRLS allow for ultrafast simulation based on the idea of using source code or its IR as functional representation. Simulation using the IRLS approaches proposed in previous work is not accurate enough. Some approaches are even compiler-intrusive and are therefore complex to realize. That is why the IRLS approaches are not widely used in practical system design.

We first developed a SLS approach, called SciSim. Compared to other existing SLS approaches, SciSim allows for more accurate performance simulation using a hybrid timing analysis method. SciSim has many advantages. It allows for ultrafast and accurate simulation. The simulation models are highly readable. This helps in debugging a complex system by means of simulation. The whole approach is very easy to realize. It takes only days to extend the tool to support a new processor. Nevertheless, disadvantages are also many. SciSim is restricted in simulation of programs in the C language, written in a required coding style. Maybe, this is not a big problem, because most embedded applications are written in C and the coding style problem can also be solved by getting a tool that converts original programs to programs in a supported coding style. Nevertheless, the mapping problems raised by compiler optimizations are more serious. They are summarized as follows:

1. The problem of finding accurate mapping between source code and optimized binary code: For some programs, where complex code structures like nested loops are used, it is hard to find an accurate mapping between source code and binary code after optimizing compilation. In many cases, the easy way of describing the mapping using debugging information no longer works.

2. The problem of correct back-annotation of timing information: Assume the correct mapping between source code and binary code is found using a sophisticated method. If the timing information is annotated straightforwardly before or after the corresponding source code statements, it cannot be aggregated correctly along the source level control flows during the simulation, because, after optimizing compilation, no constant instrumentation rule can be found for a complex C statement.

Even if the above problems can be solved, a large error still exists in the simulation of some programs like *insertsort*, as discussed in Section 4.4. Therefore, instead of making effort to solve these problems in source code instrumentation, we proposed a more efficient approach, called iSciSim, which converts source code to a lower-level representation that has a structure close to the structure of binary code and annotates timing information to this lower-level representation instead of source code. This lower-level code is called intermediate source code (ISC) to be differentiated from the original source code. ISC is formalized in the C language and is compilable. Because of the similar structures between ISC and binary code, the mapping between ISC and binary code can be accurately described using debugging information. In the current implementation, ISC is obtained by modifying standard IRs from standard compilers. In this sense, iSciSim can be regarded as an IRLS approach.

To show the advantages of iSciSim, we presented a quantitative comparison of all the discussed simulation strategies, in terms of simulation speed and accuracy using six benchmark programs with different workloads. For this purpose, we implemented representative approaches of ISS and BLS in the experiment. SciSim was used to represent the state-of-the-art of the SLS technique. According to the experiment results, iSciSim allowed for simulation with an average accuracy of 98.6% at an average speed of 4765.6 MIPS, close to the native execution (5198.2 MIPS on average). It achieved the best trade-off concerning simulation speed and accuracy in comparison to the other approaches.

In the iSciSim approach, ISC generation, timing analysis, and back-annotation of timing information might cause errors in the final simulation. According to the experimental results, the error caused by ISC generation is very small. We can eliminate this error by using the binary code generated by ISC also for final execution on the target processor. The error caused by static timing analysis could be large, but this error can be compensated with dynamic timing analysis. As dynamic timing analysis will reduce the simulation speed, users have to handle the trade-off between simulation accuracy and speed. The error caused by back-annotation was also proved to be small. We can eliminate this error by using flow analysis to get a mapping between ISC level control flows and binary level control flows, instead of a mapping between ISC level basic blocks and binary level basic blocks.

iSciSim is also very easy to realize, without the need of modifying the compiler. The tool is easily retargetable to a new processor by adding the processor's performance model and a new instruction decoder. For someone who is familiar with the tool, the work of retargeting takes only 2–3 days.

The advantages of iSciSim make it well-suited to be employed in system level design space exploration of complex systems. Nevertheless, iSciSim has also a limitation that it requires source code to be available, and therefore, is not applicable for systems design with software in the form of IPs. In this case, a hybrid solution that combines iSciSim for simulating software with available source code and another technique, either binary level simulation or instruction set simulation, for simulating software IPs would be useful. Furthermore, compared to source level simulation models, simulation models generated by iSciSim are less readable.

For multiprocessor simulation, software TLMs in SystemC can be generated by iSciSim. Software TLMs are then combined with TLMs of other system components to generate a simulator of the whole system. We demonstrated a case study of designing MPSoC for a Motion-JPEG decoder to show how iSciSim facilitates multiprocessor systems design. We also proposed an abstract RTOS model that can efficiently schedule the execution of software TLMs to take into account scheduling effects in system-level design.

10.1.2 Future Work

All the discussed software simulation techniques have both advantages and disadvantages. Different techniques are useful in different situations.

- ISSs are very accurate and can be used to calibrate high-level simulation models.
- Source level simulation models are very fast and are highly readable. If source level simulation models are accurate enough, they should be the first choice for system-level design.
- Simulation models generated by iSciSim allow for very fast and accurate simulation. Compared to source level simulation models, they have the major advantage in that they can simulate compiler-optimized software accurately. However, compared to SciSim, iSciSim has also some minor disadvantages that an extra step of ISC generation is needed and generated simulation models are less readable.
- Both SciSim and iSciSim require source code available. In case that the source code of some programs is not available, binary level simulation models are a good alternative. Binary level simulation models have moderate complexity and simulation speed. They are also very accurate.

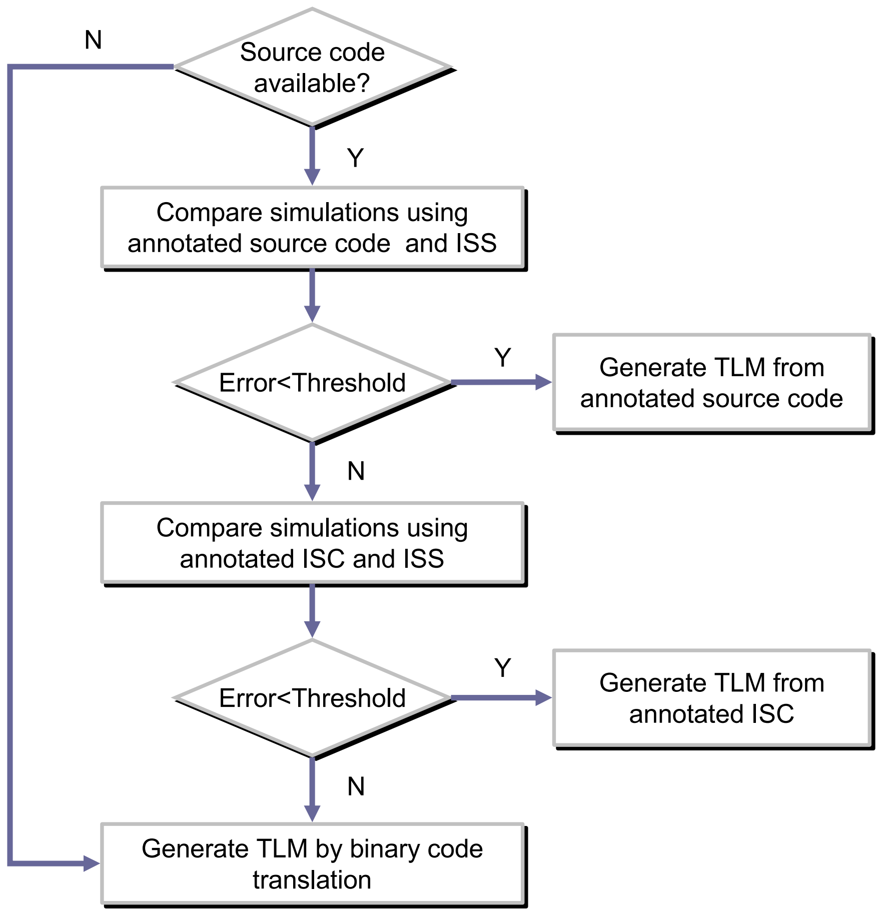

Source code available?
N
Y
Compare simulations using annotated source code and ISS
Error<Threshold
Y
Generate TLM from annotated source code
N
Compare simulations using annotated ISC and ISS
Error<Threshold
Y
Generate TLM from annotated ISC
N
Generate TLM by binary code translation

Figure 10.1: Future Work on Software TLM Generation

One of our future works is to get a software TLM generation framework that combines the advantages of all these simulation techniques. This can be achieved in a way illustrated in Figure 10.1. Given a program, it is first checked if the source code is available. If not available, the software TLM is generated by binary code translation. Otherwise, the SLS model is the first choice. We compare the simulations using an ISS and the SLS model. If the error is within the given threshold, the SLS model is used. Otherwise, the source code is converted to ISC. In most cases, the instrumented ISC can fulfill the accuracy requirement and the TLM is generated from the instrumented ISC. If in a very rare case the instrumented ISC still cannot fulfill the requirement, the BLS model is used instead.

Currently, SciSim only allows for accurate simulation of unoptimized programs or programs that do not contain complex control structures. In future we will also address the mapping problems of SciSim to make it can simulate more programs accurately.

In addition, we will continue the work on abstract RTOS modeling. In the thesis, we focused more on simulation of the scheduling behavior of RTOSs. In future, other important services of RTOSs, such as interprocess communication and memory management, will be realized in the abstract RTOS model. We will also study the way of getting accurate estimation of the RTOS overhead.

Our M-JPEG decoder case study showed that even when the software is simulated in a very high speed, the slow communication simulation will pull down the overall simulation performance significantly. It is also our future work to get an appropriate abstraction of communication models to speed up communication simulation without compromising accuracy.

10.2 Worst Case Execution Time Estimation

10.2.1 Conclusions

Bounding WCETs of software tasks is essential in hard real-time systems design. Today, static analysis still dominates the research on WCET estimation. A typical static WCET analysis approach consists of three steps: flow analysis, timing analysis and WCET calculation. Flow analysis, which constrains the possible control flows in a program, is essential for a tight WCET bound. In the state-of-the-art WCET analysis approaches, flow analysis faces some problems. If flow analysis is performed on the source level, the obtained source level flow facts must be transformed down to the binary level for WCET calculation. Due to optimizing compilation, the problem of this transformation is nontrivial. If it is performed on binary code, it is hard to get programmers' inputs, which are often mandatory, at the binary level.

In this thesis we proposed to perform flow analysis on intermediate source code to facilitate WCET estimation. ISC simplifies flow facts extraction because of its simple statements and allows for easy flow facts transformation thanks to its structure close to that of the binary code. The whole approach does not require modification of any standard tools and is not limited to a special compiler. The effectiveness of the approach was demonstrated with experiments using standard WCET benchmarks. The experiment was specially designed to show only the effectiveness of the flow analysis by removing the effect of timing analysis on the WCET estimates.

10.2.2 Future Work

In the thesis we focused more on flow analysis. For timing analysis we just created a very simple model that does not take into account the complex timing effects, such as the timing effects of out-of-order pipeline, cache, and branch predictor. It is our future work to improve timing analysis in our WCET tool. Some timing effects such as the branch prediction effect and the instruction cache effect have been frequently studied in previous work, while for the out-of-order pipeline effect and the data cache effect there are few efficient approaches proposed. In future, we will realize the state-of-the-art methods for the well-studied timing effects and research on timing analysis of the out-of-order pipeline effect and the data cache effect.

10.3 The SysCOLA Framework

10.3.1 Conclusions

In the thesis, we presented the SysCOLA framework, which consists of two design environments: a modeling environment and a virtual prototyping environment. The concepts of logical communication, abstract platform and virtual platform abstraction layer (VPAL) were introduced to facilitate parallel development of application software and system platform. In the modeling environment, a system's functionality is captured in a COLA model, verified and validated using formal techniques, and finally realized in software by means of automatic code generation. An abstract platform provides an abstract view of the system architecture and serves as the basis for automatic mapping of the software model onto the system platform. A system platform is designed in the form of a virtual platform in the SystemC based virtual prototyping environment. The VPAL wraps the virtual platform and enables early system integration. The generated code using the VPAL API can be executed on the virtual platform without caring the implementation details.

The software estimation techniques employed in SysCOLA include:

- Model-level simulation of COLA: it is aimed at early, approximate performance evaluation before source code can be generated. The approximate performance data can be used for software architecture exploration. As COLA models are essentially abstract and cannot be executed directly, executable SystemC code is generated for simulation, by means of one-to-one translation from COLA models to SystemC.
- WCET estimation: it is aimed at bounding a WCET for each COLA cluster. The WCETs are then used in the automatic mapping and scheduling algorithm. The whole process consists of generating C code from each COLA cluster, bounding a WCET of each task, and back-annotation of the WCETs to the corresponding COLA clusters to be used in mapping and scheduling.
- Software performance simulation: it is aimed at generating fast and accurate software simulation models to be used in virtual prototyping. SciSim or iSciSim is used to generate software simulation models.

10.3.2 Future Work

The BASE.XT project was successfully completed. There are several projects at BMW that follow up to either continue the work on the framework or apply it to design some practical automotive systems. I am no longer involved in these projects and will not directly work on SysCOLA, but I will continue the research on system-level design frameworks for different application domains. Here, I would like to point out some possible future works on software performance estimation and virtual prototyping in the scope of SysCOLA. The possible future work includes:

- Future work on model-level simulation: on this part of work most effort has been made to get a translation tool that generates SystemC code from COLA models. The generated SystemC code is compiled and executed outside the graphical modeling environment. It will be much more convenient for modelers, if model-level simulation can be integrated into the graphical environment. Therefore, it is the future work to realize this to get a better combination between COLA models and SystemC simulation models and to visualize the simulation on graphical models.
- Future work on WCET estimation: one work is to improve timing analysis in the WCET analysis tool, as already discussed in Section 10.2.2. Another possible work is to enable WCET estimation earlier at the model level. We can associate each COLA block or each basic operation with a pessimistic timing value for each possible target processor, and then, perform flow analysis to find the worst-case data path at the model level. This kind of model-level WCET estimation can provide the first idea about the system's worst case performance to help in software architecture exploration. Flow analysis at the model level is easier to get automated. The obtained model-level flow facts can be transformed down to facilitate WCET estimation of generated C programs.
- Future work on virtual prototyping: in the current version of virtual prototyping environment, the model library contains only simple RTOS and network models. To enable virtual prototyping of any real design, much effort should be made to create SystemC simulation models of RTOSs and networks used in practice, such as the OSEK RTOS, CAN bus, and FlexRay.

List of Abbreviations

3AC	3-Address Code
ACC	Adaptive Cruise Control
ALG	Algorithmic
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
bbList	Basic Block List
bbNode	Basic Block Node
BLS	Binary Level Simulation
BSC	Basic Software Component
CA	Cycle Accurate
CCE	Cycle Count Error
CFG	Control Flow Graph
CFSM	Codesign Finite State Machine
COLA	the Component Language
CP	Communicating Processes
CP+T	Communicating Processes with Time
cpuTLM	Transaction Level Processor Model
CT	Continuous-Time Modeling
DSE	Design Space Exploration
DPE	Data Processing Element
DSP	Digital Signal Processor
DTSM	Dynamic Time-Slicing Method
ECU	Electronic Control Unit
EDF	Earliest Deadline First
ELF	Executable and Linkable Format
ESE	Embedded System Environment

ESL	Electronic System Level
FIFO	First In First Out
fNode	File Node
FP	Fixed Priority
fps	Frame per Second
FSM	Finite State Machine
GCC	GNU Compiler Collection
HW	Hardware
IBS	International Business Strategies
IC	Integrated Circuit
ICE	Instruction Count Error
IDCT	Inverse Discrete Cosine Transformation
ILP	Integer Linear Programming
IPET	Implicit Path Enumeration Technique
IQ	Inverse Quantization
IR	Intermediate Representation
IRLS	Intermediate Representation Level Simulation
ISA	Instruction Set Architecture
ISC	Intermediate Source Code
iSciSim	Intermediate Source Code Instrumentation Based Simulation
ISR	Interrupt Service Routine
ISS	Instruction Set Simulator or Instruction Set Simulation
ITRS	International Technology Roadmap for Semiconductors
KIPS	Kiloinstructions per Second
KPN	Kahn Process Network
LFU	Least Frequently Used
LRU	Least Recently Used
M-JPEG	Motion-JPEG
MAE	Mean Absolute Error
MIPS	Million Instructions per Second
mList	Mapping List

mNode	Mapping Node
MOC	Model of Computation
MPSoC	Multiprocessor System-on-Chip
OS	Operating System
OSCI	Open SystemC Initiative
PE	Processing Element
PLRU	Pseudo Least Recently Used
PV	Programmer's View
PV+T	Programmer's View with Time
ROM	Result Oriented Method
RTL	Register Transfer Level or Register Transfer Language
RTOS	Real-Time Operating System
SALT	Structured Assertion Language for Temporal Logic
SciSim	Source Code Instrumentation Based Simulation
SDF	Synchronous Dataflow
SLD	System-Level Design
SLDL	System-Level Design Language
SLS	Source Level Simulation
SoC	System-on-Chip
SSA	Single Static Assignment
STM	Shadow Task Model
SW	Software
taskTLM	Transaction Level Task Model
TCB	Task Context Block
TLM	Transaction Level Modeling or Transaction Level Model
TM	Task Model
TSM	Time-Slicing Method
VLD	Variable Length Decoding
VLSI	Very Large Scale Integration
VPAL	Virtual Platform Abstraction Level
WCET	Worst Case Execution Time

WCRT	Worst Case Response Time
ZZ	Zigzag Scanning

List of Figures

Bibliography

[1] aiT: Worst-Case Execution Time Analyzers, http://www.absint.com/ait.

[2] AutoESL, http://www.autoesl.com.

[3] Cadence, http://www.cadence.com.

[4] CoFluent Design, http://www.cofluentdesign.com.

[5] CoWare, http://www.coware.com.

[6] Embedded System Environment (ESE), http://www.cecs.uci.edu/.

[7] Embedded Systems: Technologies and Markets. Available at http://www.electronics.ca/.

[8] Gumstix Homepage, http://www.gumstix.com.

[9] Mälardelaen WCET benchmark programs, http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.

[10] OSEK/VDX Operating System, http://www.osek-vdx.org.

[11] RTnet Homepage, http://www.rtnet.org.

[12] Synopsys, http://www.synopsys.com.

[13] SystemC Homepage, http://www.systemc.org.

[14] The AUTOSAR Development Partnership, http://www.autosar.org.

[15] The Ptolemy Project, http://ptolemy.eecs.berkeley.edu/.

[16] VaST Systems Technology, http://www.vastsystems.com.

[17] Xenomai: Real-Time Framework for Linux, http://www.xenomai.org.

[18] F.A.S.T. and TUM, study of worldwide trends and R&D programmes in embedded systems in view of maximising the impact of a technology platform in the area, 2005.

[19] Scientific and Technical Aerospace Reports, National Aeronautics and Space Administration (NASA). Available at http://www.nasa.gov, 2006.

[20] International technology roadmap for semiconductors: Design, 2007 edition. Available at http://www.itrs.net, 2007.

[21] International technology roadmap for semiconductors: Overview, 2008 update. Available at http://www.itrs.net, 2008.

[22] International Business Strategies, http://www.internationalbusinessstrategies.com/, 2009.

[23] B. Bailey, G. Martin, and A. Piziali. *ESL Design and Verification - A Prescription for Electronic System-Level Methodology.* Morgan Kaufmann, 2007.

[24] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara. *Hardware-Software Co-design of Embedded Systems: The POLIS Approach.* Kluwer Academic Publishers, 1997.

[25] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: An Integrated Electronic System Design Environment. *Computer*, pages 45–52, 2003.

[26] J. R. Bammi, W. Kruijtzer, L. Lavagno, E. Harcourt, and M. Lazarescu. Software performance estimation strategies in a system-level design tool. In *Proceedings of the International Workshop on Hardware/Software Codesign*, pages 82–86, San Diego, CA, USA, 2000.

[27] A. Bauer, M. Broy, J. Romberg, B. Schätz, P. Braun, U. Freund, N. Mata, R. Sandner, and D. Ziegenbein. AutoMoDe — Notations, Methods, and Tools for Model-Based Development of Automotive Software. In *Proceedings of the SAE 2005 World Congress*, Detroit, MI, April 2005. Society of Automotive Engineers.

[28] A. Bauer, M. Leucker, and J. Streit. SALT—structured assertion language for temporal logic. In *Proceedings of the Eighth International Conference on Formal Engineering Methods*, volume 4260 of *Lecture Notes in Computer Science*, pages 757–775, sep 2006.

[29] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri. MPARM: Exploring the Multi-Processor SoC Design Space with SystemC. *Springer Journal of VLSI Signal Processing*, 41(2):169–182, 2005.

[30] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1), January 2003.

[31] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, 1998.

[32] G. Braun, A. Nohl, A. Hoffmann, O. Schliebusch, R. Leupers, and H. Meyr. A universal technique for fast and flexible instruction-set architecture simulation. *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, pages 1625–1639, 2004.

[33] L. Cai and D. Gajski. Transaction level modeling: an overview. In *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS'03)*, pages 19–24, Newport Beach, CA, USA, 2003.

[34] E. Cheung, H. Hsieh, and F. Balarin. Framework for fast and accurate performance simulation of multiprocessor systems. In *Proceedings of IEEE International Workshop on High Level Design Validation and Test*, pages 21–28, 2007.

[35] M.-K. Chung, S. Yang, S.-H. Lee, and C.-M. Kyung. System-level HW/SW co-simulation framework for multiprocessor and multithread SoC. In *Proceedings of IEEE VLSI-TSA international symposium on VLSI Design, Automation and Test*, pages 177–180, 2005.

[36] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL '77)*, pages 238–252, 1977.

[37] A. Davare. *Automated Mapping for Heterogenous Multiprocessor Embedded Systems*. Ph.D. Thesis, Electrical Engineering and Computer Sciences, University of California at Berkeley, 2007.

[38] E. de Kock, G. Essink, W. Smits, P. van der Wolf, J. Brunel, W. Kruijtzer, P. Lieverse, and K. Vissers. YAPI: Application Modeling for Signal Processing Systems. In *Proceedings of the 37th Design Automation Conference (DAC'00)*, pages 402–405, Los Angeles, California, United States, 2000.

[39] D. Densmore, R. Passerone, and A. Sangiovanni-Vincentelli. A Platform-Based Taxonomy for ESL Design. *IEEE Des. Test*, 23(5):359–374, 2006.

[40] A. Donlin. Transaction level modeling: flows and use models. In *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS'04)*, pages 75–80, Stockholm, Sweden, 2004.

[41] M. J. Eager. Introduction to the DWARF debugging format, 2007.

[42] J. Engblom, A. Ermedahl, and P. Altenbernd. Facilitating worst-case execution times analysis for optimized code. In *Proceedings of the 10th Euromicro Workshop on Real-Time Systems*, pages 146–153, 1998.

[43] A. Ermedahl and J. Gustafsson. Deriving annotations for tight calculation of execution time. In *Proceedings of the Third International Euro-Par Conference on Parallel Processing (Euro-Par '97)*, pages 1298–1307, 1997.

[44] F. Fummi, G. Perbellini, M. Loghi, and M. Poncino. ISS-centric modular HW/SW co-simulation. In *Proceedings of the 16th ACM Great Lakes symposium on VLSI (GLSVLSI'06)*, pages 31–36, 2006.

[45] D. D. Gajski. System Level Design: Past, Present, and Future. In *Design, Automation, and Test in Europe: The Most Influential Papers of 10 Years DATE*. Springer, 2008.

[46] P. Giusto, G. Martin, and E. Harcourt. Reliable estimation of execution time of embedded software. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE'01)*, pages 580–589, 2001.

[47] B. J. Gough. *An Introduction to GCC*. Network Theory Limited, 2004.

[48] J. Gustafsson, A. Ermedahl, and B. Lisper. Towards a flow analysis for embedded system C programs. In *10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2005)*, pages 287–297, Feb. 2005.

[49] W. Haberl, M. Tautschnig, and U. Baumgarten. From COLA Models to Distributed Embedded Systems Code. *IAENG International Journal of Computer Science*, 2008.

[50] P. Hardee. CoWare CODES-ISSS Panel presentation. System Level Design Tools: Who needs them, who has them and how much should they cost?, 2003.

[51] C. Healy, M. Sjodin, V. Rustagi, and D. Whalley. Bounding loop iterations for timing analysis. In *Proceedings of IEEE Real-Time Technology and Applications Symposium*, pages 12–21, Jun 1998.

[52] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis - the SymTA/S approach. In *IEE Proceedings of Computers and Digital Techniques*, pages 148–166, Mar 2005.

[53] K. Huang, S.-i. Han, K. Popovici, L. Brisolara, X. Guerin, L. Li, X. Yan, S.-l. Chae, L. Carro, and A. A. Jerraya. Simulink-based MPSoC design flow: case study of Motion-JPEG and H.264. In *Proceedings of the 44th annual Design Automation Conference (DAC'07)*, pages 39–42, San Diego, California, 2007.

[54] Y. Hwang, G. Schirner, S. Abdi, and D. G. Gajski. Accurate Timed RTOS Model for Transaction Level Modeling. In *Proceedings of the conference on Design, automation and test in Europe (DATE'10)*, Dresden, Germany, 2010.

[55] Institute of Electrical and Electronics Engineers. IEEE Std 1666 - 2005 IEEE Standard SystemC Language Reference Manual. *IEEE Std 1666-2005*, 2006.

[56] T. Kempf, K. Karuri, S. Wallentowitz, G. Ascheid, R. Leupers, and H. Meyr. A SW performance estimation framework for early system-level-design using fine-grained instrumentation. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE'06)*, pages 468–475, 2006.

[57] K. Keutzer, S. Malik, A. R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System level design: Orthogonolization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12), December 2000.

[58] R. Kirner and P. Puschner. Transformation of path information for wcet analysis during compilation. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems (ECRTS '01)*, 2001.

[59] R. Kirner and P. Puschner. Classification of WCET Analysis Techniques. In *Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-time distributed Computing*, 2005.

[60] J. Krasner. Embedded market forecasters, embedded software development issues and challenges: Failure is not an option—it comes bundled with the software. Available at http://www.embeddedforecast.com, 2003.

[61] M. Krause, O. Bringmann, A. Hergenhan, G. Tabanoglu, and W. Rosenstiel. Timing simulation of interconnected AUTOSAR software-components. In *Proceedings of Design, Automation and Test in Europe Conference and Exposition (DATE 2007)*, pages 474–479, Nice, France, March 2007. EDAA.

[62] M. Krause, O. Bringmann, and W. Rosenstiel. Target software generation: an approach for automatic mapping of SystemC specifications onto real-time operating systems. *Journal of Design Automation for Embedded Systems*, 10(4), December 2005.

[63] S. Kugele and W. Haberl. Mapping Data-Flow Dependencies onto Distributed Embedded Systems. In *Proceedings of the 2008 International Conference on Software Engineering Research & Practice, SERP 2008*, Las Vegas Nevada, USA, July 2008.

[64] S. Kugele, W. Haberl, M. Tautschnig, and M. Wechs. Optimizing Automatic Deployment Using Non-Functional Requirement Annotations. In *Leveraging Applications of Formal Methods, Verification and Validation*, volume 17, pages 400–414. Springer Berlin Heidelberg, 2009.

[65] S. Kugele, M. Tautschnig, A. Bauer, C. Schallhart, S. Merenda, W. Haberl, C. Kühnel, F. Müller, Z. Wang, D. Wild, S. Rittmann, and M. Wechs. COLA – The component language. Technical Report TUM-I0714, Institut für Informatik, Technische Universität München, Sept. 2007.

[66] C. Kühnel, A. Bauer, and M. Tautschnig. Compatibility and reuse in component-based systems via type and unit inference. In *Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE Computer Society Press, 2007.

[67] J. J. Labrosse. *MicroC/OS-II: The Real-Time Kernel, Second Edition.* CMP Books, 1998.

[68] M. Lazarescu, M. Lajolo, J. Bammi, E. Harcourt, and L. Lavagno. Compilation-based software performance estimation for system level design. In *Proceedings of the International Workshop on Hardware/Software Codesign*, pages 167–172, 2000.

[69] J.-Y. Lee and I.-C. Park. Timed compiled-code simulation of embedded software for performance analysis of SOC design. In *Proceedings of the Design Automation Conference (DAC'02)*, pages 293–298, 2002.

[70] X. Li. *Microarchitecture Modeling for Timing Analysis of Embedded Software.* Ph.D. Thesis, School of Computing, National University of Singapore, 2005.

[71] Y.-T. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(12):1477–1487, Dec 1997.

[72] T. Lundqvist and P. Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems*, 17(2-3):183–207, 1999.

[73] B. Mehradadi. Why study for an embedded systems degree? Available at http://www.science-engineering.net/.

[74] T. Meyerowitz, M. Sauermann, D. Langen, and A. Sangiovanni-Vincentelli. Source-level timing annotation and simulation for a heterogeneous multiprocessor. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE'08)*, pages 276–279, Munich, Germany, 2008.

[75] C. Mills, S. C. Ahalt, and J. Fowler. Compiled instruction set simulation. *Software-Practice Experience*, 21(8):877–889, 1991.

[76] G. Mouchard. ppc750sim, http://microlib.org/projects/ppc750sim/, 2003.

[77] T. Nakada, T. Tsumura, and H. Nakashima. Design and implementation of a workload specific simulator. In *Proceedings of the 39th Annual Symposium on Simulation (ANSS'06)*, pages 230–243, 2006.

[78] A. D. Pimentel, M. Thompson, S. Polstra, and C. Erbas. Calibration of abstract performance models for system-level design space exploration. *Journal of Signal Processing Systems*, 50(2):99–114, 2008.

[79] K. Popovici, X. Guerin, F. Rousseau, P. S. Paolucci, and A. A. Jerraya. Platform-based software design flow for heterogeneous MPSoC. *ACM Transactions on Embedded Computing Systems*, 7(4):1–23, 2008.

[80] H. Posadas, E. Villar, and F. Blasco. Real-Time Operating System Modeling in SystemC for HW/SW Co-Simulation. In *Proceedings of DCIS, IST*, 2005.

[81] A. Prantl, M. Schordan, and J. Knoop. TuBound – A Conceptually New Tool for Worst-Case Execution Time Analysis. In *8th International Workshop on Worst-Case Execution Time Analysis (WCET 2008)*, pages 141–148, 2008.

[82] A. Pretschner, M. Broy, I. H. Krüger, and T. Stauner. Software engineering for automotive systems: A roadmap. In *Future of Software Engineering (FOSE '07)*, pages 55–71, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[83] P. Puschner. Worst-case execution time analysis at low cost. *Control Engineering Practice*, 6:129–135, 1998.

[84] P. Puschner and C. Koza. Calculating the maximum, execution time of real-time programs. *Real-Time Systems*, 1(2):159–176, 1989.

[85] W. C. Rhines. DVCon 2005 presentation: Verification discontinuities in the nanometer age, 2005.

[86] S. Rittmann. *SA methodology for modeling usage behavior of multi-functional systems*. PhD thesis, Technische Universität München, 2008.

[87] A. Sangiovanni-Vincentelli and G. Martin. Platform-based design and software design methodology for embedded systems. *Design & Test of Computers, IEEE*, 18(6):23–33, Nov/Dec 2001.

[88] C. Sauer, M. Gries, and H.-P. Löb. SystemClick: a domain-specific framework for early exploration using functional performance models. In *Proceedings of the 45th annual Design Automation Conference (DAC'08)*, pages 480–485, Anaheim, California, 2008.

[89] G. Schirner and R. Dömer. Introducing preemptive scheduling in abstract RTOS models using result oriented modeling. In *Proceedings of the conference on Design, automation and test in Europe (DATE'08)*, pages 122–127, Munich, Germany, 2008.

[90] J. Schnerr, O. Bringmann, A. Viehl, and W. Rosenstiel. High-performance timing simulation of embedded software. In *Proceedings of the Design Automation Conference (DAC'08)*, pages 290–295, Anaheim, CA, USA, Jun 2008.

[91] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprettere. System Design Using Kahn Process Networks: The Compaan/Laura Approach. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'04)*, pages 340–345, 2004.

[92] L. Thiele and E. Wandeler. Performance analysis of distributed systems. *Embedded Systems Handbook, R. Zurawski (Ed.), CRC Press.*

[93] Z. Wang, W. Haberl, A. Herkersdorf, and M. Wechs. A Simulation Approach for Performance Validation during Embedded Systems Design. In *Leveraging Applications of Formal Methods, Verification and Validation*, volume 17, pages 385–399. Springer Berlin Heidelberg, 2009.

[94] Z. Wang, W. Haberl, S. Kugele, and M. Tautschnig. Automatic generation of SystemC models from component-based designs for early design validation and performance analysis. In *Proceedings of the 7th International Workshop on Software and Performance (WOSP'08)*, pages 139–144, Princeton, NJ, USA, Jun 2008.

[95] 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'09)*, pages 220–225, San Francisco, California, July 2009.

[96] Z. Wang and A. Herkersdorf. Flow Analysis on Intermediate Source Code for WCET Estimation of Compiler-Optimized Programs. In *Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'09)*, pages 22–27, Beijing, China, August 2009. IEEE Computer Society.

[97] Z. Wang and A. Herkersdorf. Software performance simulation strategies for high-level embedded system design. *Performance Evaluation*, 67(8):717–739, 2010.

[98] Z. Wang, A. Herkersdorf, W. Haberl, and M. Wechs. SysCOLA: a framework for co-development of automotive software and system platform. In *Proceedings of the 46th Annual Design Automation Conference (DAC'09)*, pages 37–42, San Francisco, California, 2009.

[99] Z. Wang, S. Merenda, M. Tautschnig, and A. Herkersdorf. A Model Driven Development Approach for Implementing Reactive Systems in Hardware. In *Proceedings of International Forum on Specification and Design Languages (FDL'08)*, pages 197–202, Stuttgart, Germany, September 2008.

[100] 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 (WOSP'08)*, pages 33–42, Princeton, NJ, USA, Jun 2008.

[101] Z. Wang, A. Sanchez, A. Herkersdorf, and W. Stechele. Fast and Accurate Software Performance Estimation during High-Level Embedded System Design. In *Proceedings of edaWorkshop*, Hannover, Germany, May 2008.

[102] T. Wild, A. Herkersdorf, and G.-Y. Lee. TAPES—Trace-based architecture performance evaluation with SystemC. *Journal of Design Automation for Embedded Systems*, 10(2-3):157–179, 2005.

[103] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):1–53, 2008.

[104] H. Yu, A. Gerstlauer, and D. Gajski. RTOS scheduling in transaction level models. In *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS'03)*, pages 31–36, 2003.

[105] V. Zivojnovic and H. Meyr. Compiled HW/SW co-simulation. In *Proceedings of the Design Automation Conference (DAC'96)*, pages 690–695, Jun 1996.