Lehrstuhl für Entwurfsautomatisierung
der Technischen Universität München

Performance Estimation in HW/SW Co-simulation

Kun Lu

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. Eckehard Steinbach
Prüfer der Dissertation:	1.	Univ.-Prof. Dr.-Ing. Ulf Schlichtmann
	2.	Univ.-Prof. Dr.-Ing. Oliver Bringmann, Eberhard-Karls-Universität Tübingen

Die Dissertation wurde am 20. 01. 2015 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 09.11.2015 angenommen.

Abstract

Facing the high and growing design complexity in nowadays electronic systems, the need of more efficient modeling and simulation techniques arises in the domain of virtual prototypes. In the frame of this work, two major aspects of the non-functional performance estimation in fast hardware and software co-simulation are considered. In simulating the software, a method for annotating the source code with performance modeling codes is proposed. This method features rich control-flow analysis of both the source code and the cross-compiled target binary code. Based on this, it is able to annotate the source code reliably even for an optimized target binary code. Furthermore, by considering the memory allocation principles, memory access addresses in the target binary code are reconstructed in the annotated source code. These two techniques combined lead to appropriated performance estimation in the so-called host-compiled software simulation. The second aspect concerns the simulation of transaction-level models. As a modeling technique, the transfer of a large data block can be modeled by a vary abstract transaction. This work proposes a means to extract timing profiles of the highly abstract transactions so that they can be timed appropriately. Besides that, temporal decoupling can be used for fast simulation of transaction-level models. In order to correctly estimate the durations of the concurrent processes which are simulated in a temporally decoupled way, this work proposes analytical formulas to model the delays due to access conflicts at shared resources. Together with an efficient scheduling algorithm, this analytical method can dynamically predict and adjust the durations of concurrent processes.

Contents

Remember the mistakes

Chapter 1

Introduction and Background

The design of electronic systems has seen ever increasing complexity for decades. With the advent of multi-processor and heterogeneous architectures, the trajectory of such growing complexity will continue for many more years to come. Traditionally, the software development task is conducted on a hardware prototype, and the techniques such as in-circuit emulation or debugging can be used. However, to handle the growing complexity, a paradigm shift has been considered necessary since the 90's of last century [1–4]. The new paradigm promotes simulation model based development, giving rise to the HW/SW co-simulation.

The following sections in this chapter are organized as follows. Firstly, Section 1.1 introduces the HW/SW co-simulation and its contribution to the design and development of nowadays embedded systems. Secondly, Section 1.2 focuses on the aspects related to SW simulation in a co-simulation environment. It compares popular techniques of simulating the SW and puts forth the challenges considered in this work. Thirdly, Section 2.5.2 focuses on the aspects related to HW modeling, especially the inter-module communication and timing estimation. It discusses the popular hardware description language SystemC and the technique of transaction-level modeling. Finally, the academic and commercial progress in the domain of HW/SW co-simulation is briefly surveyed.

1.1 Motivation

According to [1], hardware and software co-simulation

> "refers to verifying that hardware and software function correctly together."

As the authors further put:

> "With hardware-software co-design and embedded processors within large single ICs, it is more necessary to verify correct functionality before the hardware is built."

With its popularization, the usage of HW/SW co-simulation is no longer limited to functional verification. As shall be seen, it can bolster a rich set of design tasks, including performance analysis, design-space exploration, etc. In the context of this work,

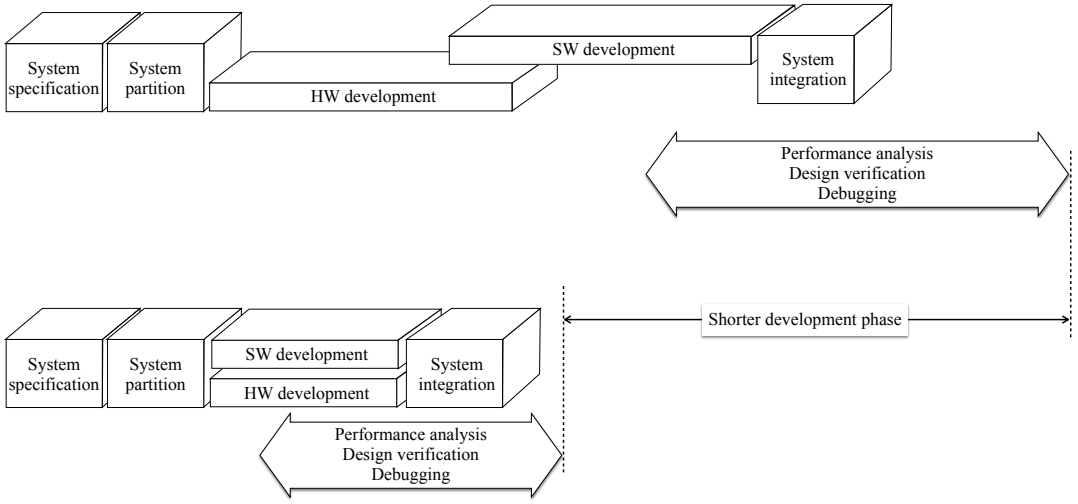


FIGURE 1.1: Co-simulation can shorten the design flow.

HW/SW co-simulation refers to the adoption of simulation platforms in aiding any design tasks of electronic systems. In the following, the benefits of using co-simulation will be discussed. Afterwards, the co-simulation tools and environments from academia and industry will be briefly outlined respectively.

1.1.1 Virtual Prototypes

HW/SW co-simulation often requires the availability of virtual prototypes. A virtual prototype refers to a simulation model that is used as the target system under development. It is different from an emulative model which prototypes the target system on a hardware platform such as an FPGA board. The terminology varies in literature. For example, CoFluent Studio [5] further distinguishes the system modeling by the abstraction level, using terms such as virtual prototypes, virtual platforms or virtual systems. In the context of this work, the term virtual prototype is used in a general sense without such distinction.

1.1.2 Benefits of Using HW/SW Co-Simulation

Compared to emulation, simulation-based approaches are endowed with a variety of advantages to the designers. They offer better flexibility, more cost-efficiency, higher controllability, and better system visibility. Furthermore, they are also easier and faster to develop. The multifaceted benefits of using co-simulation are broadly categorized in the following.

- It shortens the development phase:

  Firstly, the hardware and software development can be carried out on a simulation platform which is available in the early development phase. Many tasks therefore

can be started earlier, such as architecture exploration, performance analysis, system validation and HW/SW co-verification. Secondly, using virtual prototypes, the HW and SW design flow can be parallelized, as shown in Figure 1.1. SW designers do not need to wait for a working HW platform to test the SW programs or port them to the HW. Instead, they could conduct a large portion of the development task using virtual prototypes in the simulation platform. As seen in industrial practice, parallelizing the HW and SW design flows indeed shortens the development cycle to a large degree.

- It contributes to a high design quality:

  A few important reasons are listed here, as to why co-simulation can contribute to a higher design quality.

  1. Due to increased visibility, debugging the HW and SW becomes easier and can reach a fairly fine-grained level. Mechanisms can be implemented to trace detailed system status such as the register values, events of interests, etc.
  2. It facilitates the cooperation between the HW and SW development teams. More iterations between the HW and SW design groups can be achieved, since the effort of doing so in a co-simulation environment is much lower than that on a real product or on an emulative prototype. It also makes the full system integration easier to the HW and SW design groups. Besides, the virtual prototype can help the SW designers to better understand the HW system and develop the HW/SW interface. This eventually may reduce the design defects within a constrained development phase.
  3. Exploration of a larger design space is made feasible, which contributes to a potentially more optimal design. This is because modeling and simulating a new system candidate can be fast. For example, to evaluate a new HW/SW partition, the designers may be able to re-run the simulation by only modifying a configuration file.

- It reduces the overall development cost:

  Using simulation, the cost of modeling is very low. There is no need to manufacture the system. Modeling new HW and SW components may be cheap by re-using legacy codes. Verification and performance analysis can also be carried out. Therefore evaluating different design options is cheap. The cost of trial-and-error can also be greatly reduced. Further, another feature of simulation is its high flexibility. For example, it is easy to switch to new IP components, e.g. by simulating and verifying certain interface functions. This makes it easier to use new IPs and technologies which are changing fast.

1.2 SW Simulation

In this work, two main variants of software simulation that can be used in a co-simulation environment are considered. The first one is the instruction set simulation (ISS) based software simulation. The second is host-compiled software simulation. If the software is annotated with performance information, the latter is then called annotated host-compiled software simulation. This section introduces the basic principles of these software simulation approaches.

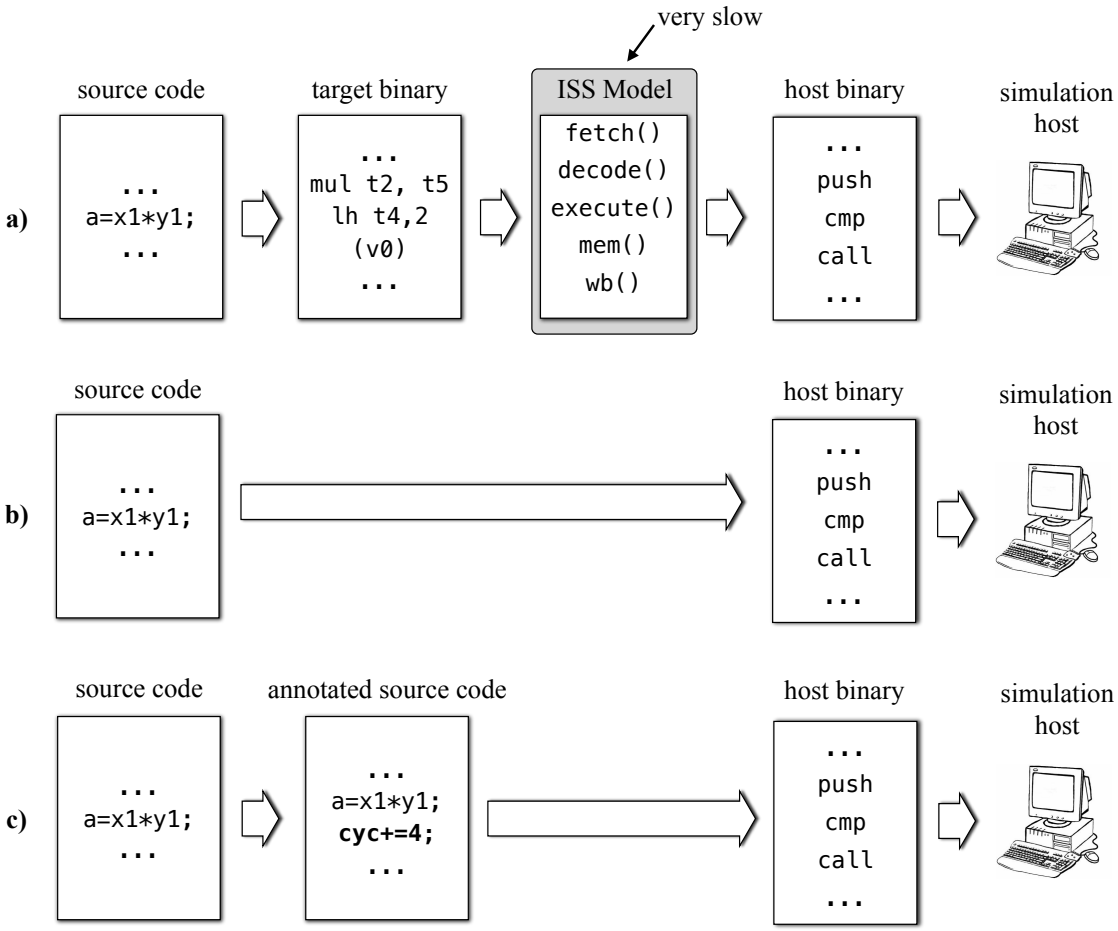


FIGURE 1.2: Basic steps in different SW simulation approaches. (a) ISS-based SW simulation; (b) Host-compiled SW simulation; (c) Host-compiled SW simulation procedure with annotated source code.

1.2.1 ISS-Based Software Simulation

An instruction set simulator (ISS) is a simulation model of the target processor. It models the internal processing steps of a processor in interpreting and executing one instruction, as can be seen in Figure 1.2(a). Therefore, an ISS interprets the target binary code as the target processor would do. Being a detailed model, cycle-accurate timing can be provided by an ISS. Before the actual hardware is available, ISS-based software simulation can be used to provide relatively accurate performance estimation. Even after the target hardware platform is ready, it can still be used as an alternative, e.g. for debugging or exploring different configurations.

Several basic terms are defined in the following before a more detailed description.

The **target processor** is the processor that will be used in the HW system of the final product. Common examples of a target processor include the MIPS processor [6], the ARM processor [7], the OpenRISC processor [8], etc.

The **simulation host** or **host machine** is the machine on which the simulation is carried out. It may use a different processor and hence the instruction set architecture from the target processor. For example, the simulation host can be an Intel machine with an *x86_64* processor.

Target binary or target binary code refers to the binary code that can be executed by the target processor.

Cross-compilation is the generation of the target binary using a cross-compiler installed on a simulation host. This compiler usually is different from the compiler used by the simulation host.

As is depicted in Figure 1.2(a), the following steps are performed in ISS-based SW simulation.

- **Step 1:** The source code firstly needs to be cross-compiled into the target binary. This step therefore requires the availability of the cross-compiler.
- **Step 2:** The ISS is provided with the target binary image.
- **Step 3:** The whole system model consisting of the ISS and other HW modules is compiled into a host executable.
- **Step 4:** Finally the simulation is performed by running the host executable. In the simulation, the ISS interprets each binary instruction of the target binary. It can model both the pipeline stages and corresponding memory accesses as would be performed by the target processor.

1.2.1.1 Disadvantages of ISS-Based SW Simulation

Besides the high modeling effort, the major disadvantage of traditional ISS-based SW simulation is that the simulation speed is relatively slow, which can make it very expensive to use in some cases such as the simulation of long software scenarios. This problem also limits the application of ISS-based SW simulation in design tasks such as design space exploration or real-time simulation, where high simulation speed is required.

The simulation cost of ISS-based simulation can be roughly assessed by considering the required host machine instructions to simulate one line of the source code. Several target binary instructions can be generated from cross-compiling one source code line. Simulating each of these instruction requires the ISS model to perform a chain of tasks corresponding to the pipeline stages of the target processor. This translates to up to several thousands of host machine instructions in simulating one source code line. As a result of this high cost, usual ISS models can simulate a few millions of target instructions on a host machine with a CPU clocked at GHz. Although there exist approaches toward faster ISS models [9–12], another line of research based on host compilation has received increasing popularity.

1.2.2 Host-Compiled SW Simulation

The basic steps in host-compiled SW simulation are shown in Figure 1.2(b). The source code is directly compiled to a host executable for the simulation host. A model of the target processor such as an ISS is completely by-passed. Such host-compiled simulation is very fast and yet able to verify the functional correctness of the simulated software. However, it can not provide non-functional performance estimation such as the timing information of the software. Specifically, the missing information includes the following:

- For the computation aspect of the software, the timing related to the execution time of the simulated SW becomes unknown.
- For the communication aspect of the software, the memory accesses caused by the store and load instructions become invisible, because the accessed addresses can not be statically obtained from the target binary code.

In the very early design phase, host-compiled simulation without performance estimation could be used for fast functional verification. As the design proceeds, performance estimation may become mandatory for many design tasks, including the design space exploration, timing verification, etc. To cover these design tasks, an improved version of host-compiled simulation has been proposed, which annotates performance modeling codes in the original software program. This new approach is discussed in the next section.

1.2.3 Annotated Host-Compiled SW Simulation

The basic principle of annotating the source code for host-compiled SW simulation is to augment the source code with performance modeling codes. It aims at providing performance analysis with sufficient accuracy while keeping high simulation speed. The annotated codes usually include execution cycles and memory accesses, corresponding to the computation and communication aspects of the target binary code, respectively. After annotation, the source code can be used as a *performance model* of the target binary. Therefore, through executing the annotated source code, performance estimation and analysis can be provided in host-compiled simulation as well.

1.2.3.1 Basic Block

A basic block is the largest piece of code that has a single entry and a single exit point, between which there exits neither a branch nor a target of a branch. Once a basic block is entered, the contained code will definitely be executed until its exit point. For annotated host-compiled simulation, most existing approaches perform the annotation at the granularity of the basic blocks.

1.2.3.2 Line Reference

Using the utility from the cross-compiler tool chain, debugging information such as the line reference can be obtained. The *line reference file* lists the reference line of the source code from which an instruction in the target binary is compiled. An excerpt of a sample line reference file is shown in the following:

```
CU: App.c:
File name      Line number    Starting address
App.c                  125               0x104
App.c                  124               0x108
App.c                  125               0x118
App.c                  119               0x12c
```

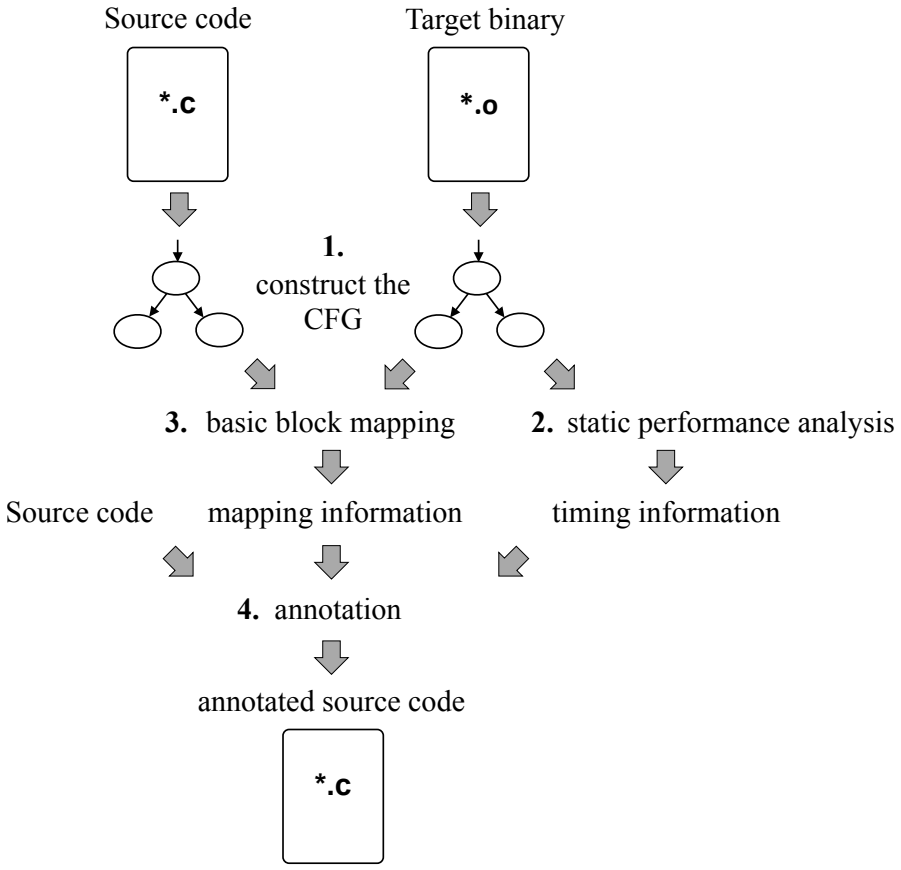


FIGURE 1.3: Basic steps in annotating the source code.

```
App.c                   116                 0x138
...                     ...                 ...
```

For example, from this line reference file, it can be known that the instructions within `[0x108, 0x118)`, i.e. `0x108, 0x10c, 0x110 and 0x114`, are compiled from line 124 in the source code.

1.2.3.3 The Annotation Procedure

The basic steps in annotating the source code are shown in Figure 1.3. These steps are described in the following.

1. The control-flow graphs are constructed for the source code and the target binary.
2. For each basic block in the target binary, performance information such as the execution cycles can be extracted through static analysis.
3. With the line reference, the annotation process can reason about the mapping from the basic blocks in the target binary to those in the source code.
4. After the mapping is constructed, the statically estimated performance modeling code of a target binary basic block is annotated into its counterpart basic block in the source code.

Afterwards, the annotated source code can be directly compiled and executed on the simulation host. In the simulation, performance estimation is achieved by executing the annotated codes for performance modeling.

1.2.3.4 Annotated Codes for Performance Modeling

The statically extracted codes for performance modeling cover two main aspects regarding the execution of a software. One is the computation aspect and the other is the communication aspect.

1. The computation aspect is represented by the estimated execution time or cycles of a piece of software code. For example, if the execution of a basic block is estimated to take be 8 cycles, then

   ```
   cyc+=8
   ```

   can be annotated in the counterpart basic block in the source code. The variable `cyc` is initialized to zero.

2. The communication aspect is represented by the annotated memory accesses. For example, assume the address of the accessed instruction in a target binary basic block is `0x2100`, then

   ```
   iCacheRead(0x2100)
   ```

   can be annotated in the counterpart basic block in the source code[1].

   Here, special attention needs to be given to the fact that the cache simulation is **non-functional**. This means that, for a given address, the cache model only checks whether the access causes a cache hit or miss, but does not cache any functional data. Upon a cache miss, a transaction can be initiated to access the memory over bus. This transaction is used to simulate the on-chip communication related to memory accesses, thus modeling the communication aspect of the target binary code. But, no functional data are actually transferred by this transaction.

   Similar annotation holds for simulating data cache. However, the addresses of the data memory accesses can not be statically extracted. This problem will be detailed in Section 2.2.

1.2.4 Comparison of the Above Approaches

The main pros and cons of the previously discussed software simulation approaches can be briefly summarized in Figure 1.4. As is shown, ISS-based SW simulation offers

[1] If the size of a cache-line is fixed, then the sequentially executed instructions that fit in the same cache-line require the instruction cache simulation for only once. For example, assume the cache-line size is $16 bytes$, then for a target binary basic block with 6 instructions starting from the instruction address $0x2100$, the annotation `iCacheRead(0x2100, 6)` suffices to simulate the instruction cache behaviour since the instructions fit in the same cache-line corresponding to the address $0x2100$. This reduces the overhead of instruction cache simulation.

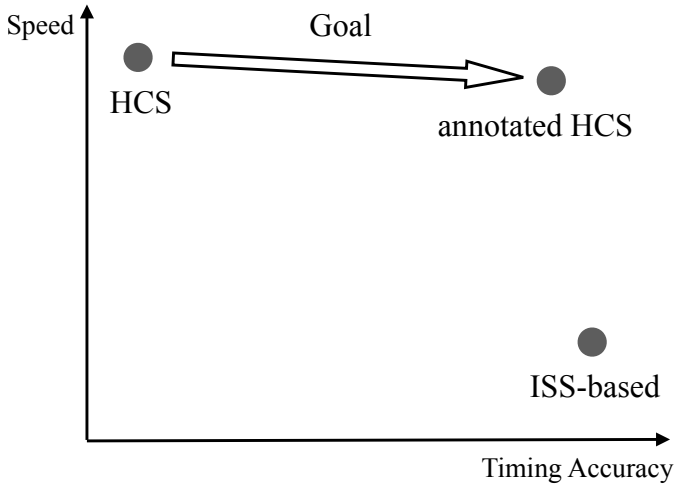


FIGURE 1.4: Compare different methods of SW simulation

the highest timing accuracy but relatively low simulation speed. Host-compiled SW simulation (HCS) is very fast but can not provide performance analysis. Therefore, the goal is to obtain an annotated source code that can be used as an accurate performance model of the target binary. A good annotation should preserve the correct *execution order* of the annotated performance modeling codes. This is hard to achieve due to compiler optimization. For optimized target binary, the line references become neither sufficient nor reliable. The concrete challenges are detailed in Section 2.2.

1.3 HW Modeling and Simulation

Recent hardware system modeling has witnessed the trend of moving to electronic system level (ESL), in the need of better handling the growing design complexity. Joint force from industry and academia has promoted the design shift to ESL. As a result of this effort, SystemC and transaction level modeling (TLM) have been developed and standardized. Both of them are now widely accepted. In the following, the principles of SystemC and TLM are briefly introduced. Afterwards, the often mentioned term abstraction level is discussed to clarify conceptual ambiguity.

1.3.1 SystemC

SystemC was defined by the Open SystemC Initiative (OSCI) and standardized by IEEE in 2005 [13]. It is a HW description language that is developed upon C++. Specifically, as its name suggests, SystemC is suitable in design tasks such as system-level modeling, system-wide analysis, component-based modeling, etc. The core ingredients and features of SystemC are summarized in the following.

- **Modular design:** With a few macros, it is very easy to describe and instantiate HW modules in SystemC. The granularity of a module can vary, e.g. ranging from an adder to a processor. A module encapsulates its internal computation and reduces the complexity of modeling a whole system to modeling its components.

- **Inter-module communication:** In its early version, the inter-module connection is modeled by *ports*, *interfaces* and *signals*. Now, with TLM 2.0, the connection is often modeled by *sockets*. By connecting the modules in a top-down manner, it is easy to model the system in a hierarchical manner.
- **Modeling of computation:** SystemC abstracts the modeling of the computation in the HW modules into *threads* and *methods*. They are referred to as processes in general unless distinction is necessary. This simplifies the system modeling task when lower level details are not relevant. For example, the computation in an encryption module can be modeled by a process containing the encryption algorithm written in C++, without modeling the underlying adders and multipliers.
- **Notion of time:** Modeling of time in SystemC is straight-forward. There are two main types of *wait* statements for this purpose.

  1. Wait on a time variable t, such as in *wait(t)*. Supported time units include *SC_FS*, *SC_PS*, *SC_NS*, *SC_MS*, and *SC_SEC*.
  2. Wait on an event e, such as in *wait(e)*. The execution of the process that calls this wait statement will resume when the event occurs. This time can be set by notifying the event as in *e.notify(t)*, e.g. by some other process. It is worthy of pointing out that it is possible to cancel the notified time of an event and re-notify it to a new time.

  A process calling *wait()* without argument will stall forever and will resume only when a default event occurs that this process is sensitive to.
- **Process scheduling:** SystemC uses a central scheduler to schedule concurrent processes. Each time a *wait* statement is called, the scheduling algorithm will be performed. This scheduler inspects the queues of stalled processes and selects the next process that is ready to run.
- **Timing synchronization cost:** After a *wait* statement is issued, a context switch will be performed between the current process and the SystemC scheduler. After checking the process queues and selecting the next runnable process, another context switch will be performed from the scheduler to the next process to resume. Such context switches are computationally very **expensive**, therefore frequent timing synchronization can slow down the simulation to a great degree.

1.3.2 Transaction-Level Modeling (TLM)

Transaction-level modeling (TLM) has been introduced to simplify the modeling of inter-module communication. To support models written in SystemC, TLM 1.0 and TLM 2.0 have been published in 2001 and 2009 respectively. The hardware virtual prototypes modeled with the TLM technique are referred to as transaction level models (TL models), or TLMs for short. This abbreviation is also often used in literature, but it ought not to be confused with the term TLM.

In TLMs, transactions are used to model the data transfer between HW modules, while the underlying complex signal protocols are abstracted away. One module can be completely agnostic to the signal interface of another module that it connects to. This not

only simplifies the modeling effort but also greatly improves the simulation speed as compared to RTL models. Therefore TLMs are suitable for fast and early SW development, system verification and design space exploration.

In essence, TLM is data-flow oriented modeling. Although the granularity of the data-flow being modeled can be arbitrary, two main categories of transactions have been used in the literature corresponding to two different abstraction levels.

1. Bus-word transactions: The data transferred by one transaction is a unit of datum supported by the bus protocol. For example, a bus-word transaction can transfer a byte, a word, or a burst of words. It can be regarded as a primitive transaction that transfers a basic data unit. Handshaking signal protocols, that are respected by the modules in the hardware implementation, are abstracted away.
2. Block transactions: The data transferred by one transaction can be increased in size. For example, a whole data block or packet can be transferred. According to the TLM 2.0 standard, a *generic payload* is used in implementing a transaction. Within this generic payload, a pointer is passed around that points to a data block, together with the size of the data to be transferred. One block transaction abstracts away the software protocols of the corresponding driver functions that implements the data transfer. Further details regarding such transactions are explained in the next section.

1.3.3 TLM+

Initially, TLM+ [14] is proposed as a further abstraction from TLM. It regards the inter-module communication as data-flows and provides support to bypass the underlying software protocols. In terms of modeling, it introduces the following changes:

1. A transaction can transfer not only a bus-word, but also a large data block. This data block can have arbitrary data type and size.
2. The corresponding driver function in the SW is simplified in a way that it calls a single transaction to transfer a large data block. This transaction replaces a long sequence of bus-word transactions that are invoked in the original driver function for transferring the data block.

From now on, the term *TLM+ transaction* [14] is used when referring to a transaction that transfers a large data block. TLM+ is in fact compatible to the later TLM 2.0 standard, which implements a TLM+ transaction by passing a pointer and the data size.

In the standard TLM simulation, the transfer of a data block such as a buffer initiates a long sequence of bus-word transactions. Expensive timing synchronization needs to be performed before and during each bus-word transaction. Such effort is greatly reduced in TLM+. One TLM+ transaction abstracts the sequence of bus-word transactions into one single transaction. The changes in the driver function are exemplified using *write_uart()* which writes a buffer to the UART module. The snippet in List 1.1 shows that, for each data word in the buffer, there is one iteration of handshaking protocol involved in the standard TLM simulation. Executing this driver function will evoke a long sequence of bus-word transactions to complete the data transfer.

```
1   int write_uart(buffer, size) {
2     ...
3     while(i<size){
4       get_uart_status_reg();
5       ...
6       set_uart_txd_reg(buffer[i]);
7       ...
8       wait_irq(...);
9       set_uart_irq_reg(0);
10    }
11  }
12
```

LISTING 1.1: Example of a driver function in standard TLM simulation

For comparison, List 1.2 shows the driver function for TLM+ simulation. The complete *while* loop in List 1.1 is replaced by a single transaction as shown in line 4.

```
1   int write_uart(buffer, size) {
2     get_uart_status_reg();
3     ...
4     set_uart_txd_reg(buffer, size);
5     ...
6     wait_irq(...);
7     set_uart_irq_reg(0);
8   }
9
```

LISTING 1.2: Example of a driver function in TLM+ simulation

1.3.4 Temporal Decoupling

1.3.4.1 Timing Simulation in Standard TLMs

Before the introduction of TLM 2.0, functional simulation and timing simulation are intermingled with fine granularity. A process calls the *wait* statements before initiating a bus-word transaction, so that the inter-module communication is correctly synchronized. A depiction of such synchronization can be seen in Figure 1.5(a). With such fine-grained timing synchronization, all bus-word transactions are performed at the correct global time, therefore the simulation can capture access conflicts at shared HW modules in a multiprocessor simulation. During a bus-word transaction, more *wait* statements can be called when needed. For example, a bus can be cycle-accurate and contain a process that synchronizes at each clock cycle. This process can arbitrate the on-going transactions according to the bus protocols. A bus can also be un-clocked, i.e. it does not need to wait for each clock cycle. The transactions are arbitrated with estimated durations. This leads to a faster simulation than the cycle-accurate bus model, and timing can still be sufficiently accurate.

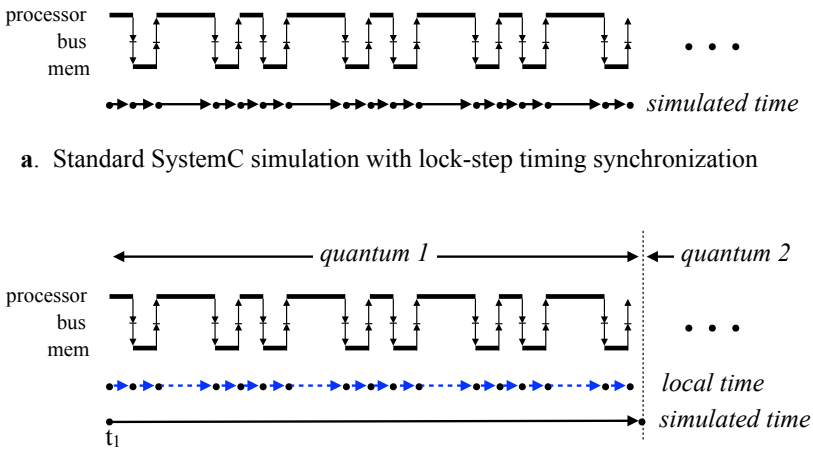


FIGURE 1.5: Timing synchronization before and after using temporal decoupling.

1.3.4.2 Temporal Decoupling in TLMs

Fine-grained synchronization as in Figure 1.5(a) is computationally expensive due to the incurred context switches. If the bus-word transactions need to be evoked very frequently, then the simulation is heavily slowed down due to the timing synchronization overhead. This can be the case if the simulated software requires frequent I/O communication or the cache miss rate is high.

To reduce the simulation overhead, temporal decoupling (TD) is introduced in the TLM 2.0 standard [15]. In fact, this concept has already been explored in earlier works [16, 17]. With this concept, TLM 2.0 proposes the notion of *local time* and *global time*. With temporal decoupling, the local time of a process can be larger than the global time. A process can be simulated ahead without the need to synchronize, even before it issues a transaction. In other words, timing synchronization is performed with a much coarser granularity, e.g. after the functional simulation of a large piece of code or a large quantum is reached. The basic principle of temporal decoupling is illustrated in 1.5(b). As can be seen, a local time variable is used to register the local time. If it does not exceed a pre-defined value of the global quantum, then no call to the *wait* statement is issued. In this way, the functional simulation is decoupled with the timing simulation. Using temporal decoupling, the number of synchronizations per thread is reduced to one within each global quantum. Setting a larger global quantum may therefore lead to more speed-up.

1.3.5 Transaction Level: Rethink the Nomenclature

A transaction per se may correspond to different levels of abstraction. Therefore, the term transaction-level can cause ambiguity when describing models at different abstraction levels. In fact, since its advent, there have been opposing opinions on the appropriateness of its terminology. Some [18] hold that "TLM does not denote a single level of abstraction but rather a modeling technique", therefore it is more appropriate to name

is as transaction-based modeling. Authors in [19] suggest that transactions in essence are data-flows. They consider their modeling methodology to be *data-flow* abstracted or oriented virtual prototyping. Another thought provoking work [20] even challenges whether TLM is a *buzz-word* and whether it is useful enough in handling nowadays design task.

From the perspective of inter-module communication, an abstraction level can correspond to the granularity of data transfer being modeled. If the transferred data correspond to the values of signals, then this abstraction is at the pin level. If the transferred data correspond to a byte or word that fit the bus protocols, then this abstraction level can be said to be at the bus-word level. If the transferred data correspond to a large data block or packet, then the abstraction is at the data block or packet level. In this work, the term transaction-level modeling is still used, despite its ambiguity. But to distinguish the different abstraction levels, the data granularity is explicitly mentioned together with the transaction, such as a bus-word transaction or a block transaction.

1.4 Recent Development in HW/SW Co-Simulation

The last two decades have seen prevalent development and application of the co-simulation environments, both in academia and in industry. Being an important part in the design task, a good co-simulation environment should meet certain requirements to effectively aid the design process. It should cover a wide range of system configurations, in order to support various processor types and system architectures which may be required by the emerging heterogeneous systems. It should be able to simulate different applications. It should also adapt to the demands of the system designers, e.g. switching among different abstraction level. It is challenging to develop a general co-simulation environment that meets all the requirements. This section broadly surveys both the academic and industrial progress in bringing out new co-simulation environments.

1.4.1 Academic Research and Tools

An early work (*Becker,92* [2]) proposes a co-simulation environment to enable concurrent development of software and hardware which are written in C++ and Verilog respectively. It uses Unix interprocess communication to interact software simulation with hardware simulation.

Poseidon [3] can synthesize the hardware and software components from a predefined system description. For timing simulation, clock cycles for the functional models need to be provided as input to the tool.

Ptolemy [4] is a tool that precedes and shares many similarities with SystemC. It targets the platform modeling for heterogeneous components, where the concept of modular component based modeling is raised. Computation in the modules is modeled by finite state machines. Inter-module communication is modeled by wires and ports as in SystemC.

Pia [16], developed upon Ptolemy, has several features. Firstly, it provides multiple communication models. Secondly, it proposes an advanced scheduling mechanism. Its scheduler supports the use of *global time* and *local time*. Therefore it is among the

first to propose the concept of temporal decoupling. Besides, it offers conservative and optimistic scheduling, corresponding to two modes of timing simulation. Thirdly, it enables the option of modeling the processor at different level of abstraction. This also includes host compiled simulation, in which the source code is compiled directly for the host.

The approach in [21] co-simulates HW components in VHDL and SW components in C. It adopts a so-called *multi-view* library concept to encapsulate various HW/SW implementations. This is conceptually similar to raising the abstraction level.

Miami [22] is another co-simulation environment which integrates an event driven hardware simulator with an instruction set simulator.

COSMOS [23, 24] is a co-simulation tool that features hardware and software interface generation. It supports software programs written in C and hardware models written in VHDL. Different levels of abstraction can be simulated as well. It also provides modeling and simulation support for heterogeneous multiprocessor systems.

The approach in [25] combines an instruction set simulator (ISS) with an event-based system simulator, so that the timing can be measured online instead of estimated offline. In addition, it uses the so-called cached timing to improve the simulation speed of the ISS. This is in effect equivalent to temporally decoupling the simulation of one system component.

The approach in [26] aims to integrate different IP models using a common high level communication semantic. It can be used in a platform of a heterogeneous system, where different IP models may be configured and simulated at different abstraction levels.

Giano [27] is another HW/SW co-simulation framework. It supports the SW execution in two modes: (1) executing the target binary code using a simulated microprocessor; and (2) executing VHDL code using a simulated FPGA. The simulated target system can interact with the external environment in real time. By attaching a HW simulator, it can simulate a HW system written in Verilog, VHDL, SystemC, and C.

Authors in [28] aim to reduce the communication overhead between simulator and emulators, by avoiding communication in a period if no transactions occur in it. This period is predicted by inspecting the software and hardware modules. The problem targeted by this approach is automatically attenuated in system simulation, because the overhead is only induced when transactions take place. Therefore the prediction is not necessary any more.

A component-based simulation platform is adopted in [29]. Both the HW and the SW are abstracted as components. Bridge components are used to ease the interface the HW/SW components. The co-simulator can be configured by the specifications of the components.

The simulation in [30] can be configured to provide multi-accuracy power and performance modeling. Transaction-level modeling is used for the hardware models. The authors modify the SystemC kernel and TLM library to enable the switching among different trade-offs between simulation accuracy and speed.

The approach in [31] annotates the software program with performance modeling codes before simulating it on the hardware modeled at transaction level. It predicts the synchronization point during the simulation and uses this prediction to reduce the synchronization overhead. This technique is a common practice in host-compiled software simulation with annotated performance.

The co-simulation framework in [32] is written in SystemC. It supports the integration of a many-core architecture, with each core modeled by an instruction set simulator.

The approach in [33] also targets many-core simulation up to the scale of a thousand cores. For such computation-intense simulation, techniques are required to boost the simulation speed. Adopted techniques by the authors include host-compiled execution, distributed simulation and course-grained synchronization.

1.4.2 Commercial Tools

Seamless [34] from Mentor Graphics is an early commercial co-simulation tool. It provides the possibility to switch between ISS simulation and host-compiled simulation. The HW system can be described at register-transfer level or at transaction level, depending on the design stage. In Seamless, HW/SW communication is mediated by a bus interface model, which handles the data transfer between the processor and memory or I/O ports. Timing, in terms of bus cycles, can be modeled by the bus interface model.

ModelSim [35] supports hardware design languages such as Verilog, VHDL, and SystemC. Basic debugging utilities such as signal-level tracing and code coverage measurement are provided. Another integrated feature in ModelSim is the assertion-based verification, using the IEEE Property Specification Language (PSL) [36].

Carbon [37] allows the plug-in of an accurate CPU model in an RTL environment. It emphasizes on its software debugging feature that enables the designer to interactively debug both the software and the hardware such as the processor registers.

Gezel [38–40] provides a co-simulation engine that supports a variety of processors, including ARM cores, 8051 micro-controllers and picoblaze micro-controllers. Users can configure multiple cores in the simulation and evaluate the design trade-offs by using coprocessors.

VMLab [41] is a co-simulation engine mainly for virtual prototyping and simulating the AVR models. It integrates the simulation of analog signals and provides SPICE-like hardware description language.

CoCentric System Studio [42] from Synopsys is a development suite for system-level design. It promotes the use of data-flow models in system modeling. The design focus of CoCentric is on two aspects. One is to produce algorithmic models that aim to verify the functional correctness in a very fast way. The other is the architecture modeling at multiple abstraction levels, from system level down to pin-accurate level.

CoFluent Studio [43] from Intel specializes itself in the very early phase of the design flow. At such an early phase, system modeling at a very abstract level can be used. For example, it uses message-level data transfer for communication modeling and ISS-free SW simulation for computation modeling. A typical application scenario with CoFluent is the modeling and simulation of use-cases, from which preliminary performance

prediction and design-space exploration could be performed. The performance metrics that can be simulated by CoFluent include timing, power, system loads, etc. An essential value proposition of CoFluent is that, through executable specifications and highly abstract modeling, design decisions can be made in the initial design phase before any HW and SW development take place, thereby shortening the overall design phase.

Platform Architect [43], previously Coware now Synopsys, features system level modeling and performance analysis. The inter-module communication is implemented using transaction-level modeling techniques. The simulation is claimed to be cycle-accurate. Adaptors are provided to interface transaction-level modules and RTL modules. It is applied in the early architectural optimization of a multi-core system, by using task-level or trace-driven simulation.

Chapter 2

Challenges and Contributions

In the beginning, this chapter gives an overview of the targeted challenges to tackle in the domain of co-simulation. This overview can be regarded as the scope of this work within which the proposed approaches can be applied. Details of each challenge will be described. Afterwards, related approaches are discussed. Following that, proposed solutions are summarized as the contributions of this work.

2.1 The Scope of This Work

To understand the challenges that this work aims to tackle, it is helpful to consult Figure 2.1 which coherently overviews the considered cases in a co-simulation environment, together with existing techniques that have been applied by previous researchers in each case to expedite the simulation.

The first case depicts the simulation of the software codes corresponding to high-level applications. They usually reflect the computation aspect of the software. The computation is mainly performed at the target processor in interpreting the instructions and initiating memory accesses after cache misses. The technique for faster software simulation in this case is the host-compiled simulation, which has been introduced in the previous chapter.

The second case considers the codes at the hardware abstraction layer, with the emphasis on device drivers. These drivers implement the communication between the CPU and the I/O devices. In the real implementation, a driver function performs the low level software handshake protocols that transfer data among the CPU, the memory, and the registers of the corresponding I/O device. From a modeling perspective, a driver function can be implemented at a high abstraction level for fast simulation. Therefore, the technique for a faster simulation in this case is the adoption of the abstract TLM+ transactions for modeling the transfer of data blocks.

The third case relates to the simulation of a multi-processor system. Because synchronization before each bus-word transaction is expensive, thus the technique for a faster simulation in this case is to temporally decouple the simulation of the software code on each processor, so that timing synchronization is performed only once after simulating a

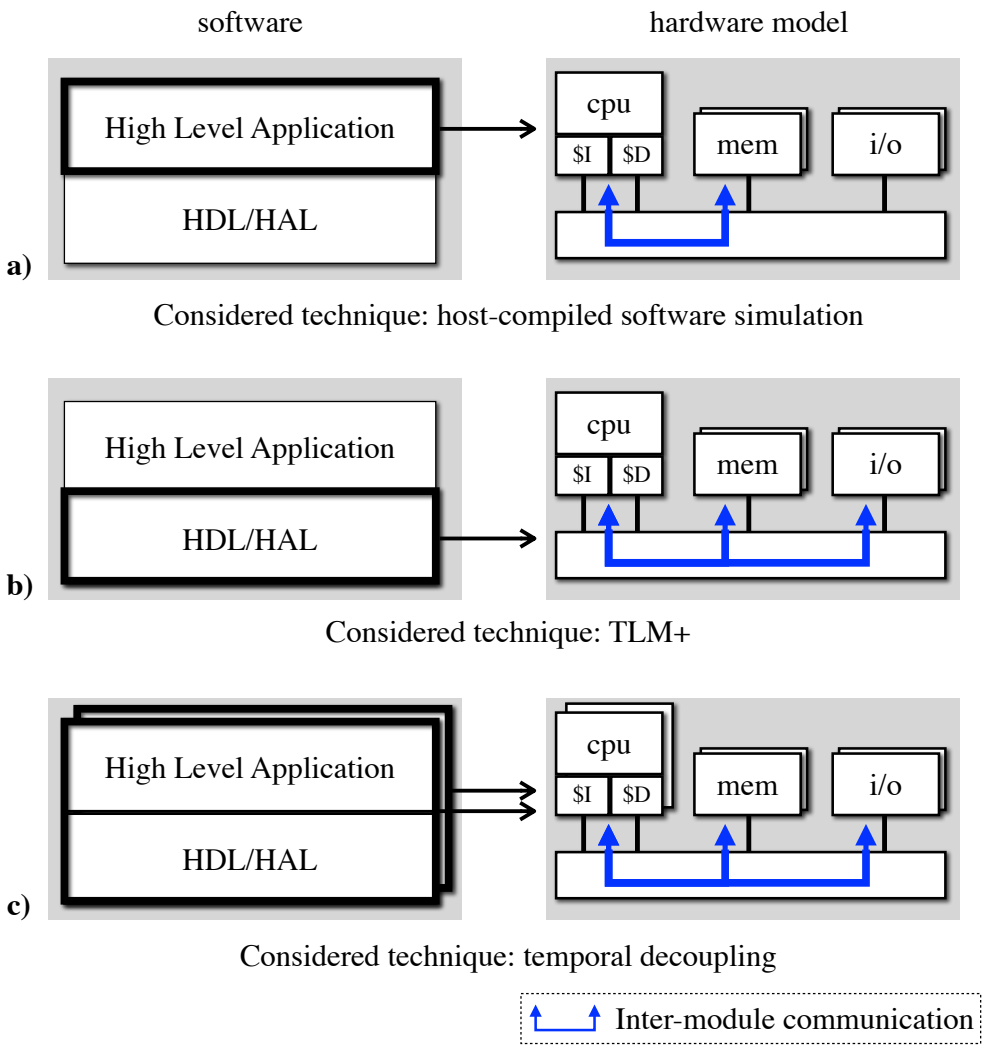


FIGURE 2.1: Overview of the considered cases of system modeling and simulation. a) Simulation of high level application codes; b) Modeling and simulation of driver functions at the hardware dependent or hardware abstraction layer; c) Multi-processor simulation in which temporal decoupling can be used. For these cases, the simulation speed can be improved by using existing techniques as listed below each figure.

large piece of the software code. This technique can be used, in addition to host-compiled simulation and TLM+ transactions, to further improve the simulation speed.

Unfortunately, the above techniques applied in each case also induce several problems that lower the simulation accuracy. Describing these problems will be the subject of the following sections.

2.2 Challenge in Annotating the Source Code

For performance estimation, host-compiled software simulation requires the source code to be annotated with performance modeling codes. The basic steps in doing this have been introduced in Figure 1.2. For a successful annotation, there are two major challenges:

The first one is to resolve the correct position in the source code, so that the performance modeling codes could be annotated in that position. In practice, this translates to mapping the basic blocks in the target binary to those in the source code. The difficulty arises as a result of compiler optimization, due to which it becomes ambiguous to determine a correct basic block mapping.

The second one is to determine the data memory access addresses, so that they can also be annotated to enable the simulation of cache and memory accesses. These addresses are obfuscated to extract because they can not be statically obtained from the target binary. Following sections will detail each challenge and point out the concrete problem to solve.

2.2.1 Timing Annotation and Basic Block Mapping

The aim of the annotation is to use the annotated source code as an accurate performance model of the target binary. To achieve this, the following criterion should be met:

The annotated performance modeling codes in the host-compiled simulation should be executed in a similar order as that in the ISS-based simulation of the target binary.

Such an annotation in effect *embeds* the control flow of the target binary code into the annotated program, which then becomes a good performance model of the target binary code. In this way, directly compiling and executing the annotated source code on the simulation host will accumulate the estimated performance in a proper order. Therefore the total execution count of each part of the annotated codes will also be correct. As a result, host-compiled simulation will yield similar performance estimation as compared to simulating the target binary code with an ISS.

The Mapping Problem

In the ideal case, the control-flow graph of the target binary resembles that of the source code, leading to a one to one mapping between the basic blocks of these two codes. In this case the annotation criterion can be largely satisfied, and the resulted annotated source code can be used as a good performance model of the target binary. However, it is very difficult to meet the annotation criterion if the target binary has been optimized by the compiler. Commonly seen optimizations include code motion, branch elimination, and loop optimization. With compiler optimization, the control-flow graphs of the source code and the optimized target binary can be very different. It becomes obscure how to map the basic blocks of the target binary to those of the source code, therefore a correct annotation position of the performance modeling codes can not be determined in a straightforward way.

The problem of an altered control-flow in an optimized target binary can be seen in the ambiguous and erroneous line reference file. An example is shown in Figure 2.2 to illustrate the problem. According to the line reference, the instructions in the basic block *bb2* of the target binary are compiled from several lines in the source code. These lines correspond to the basic block *sb1*, *sb2*, and *sb3*. So the question is where should the performance modeling code of *bb2* be annotated in the source code? Similar problem holds for *bb3*. As can be seen, *bb2* is the entry basic block of the outer loop in the target binary, therefore the correct counterpart for it should be *sb2*. But this mapping can not

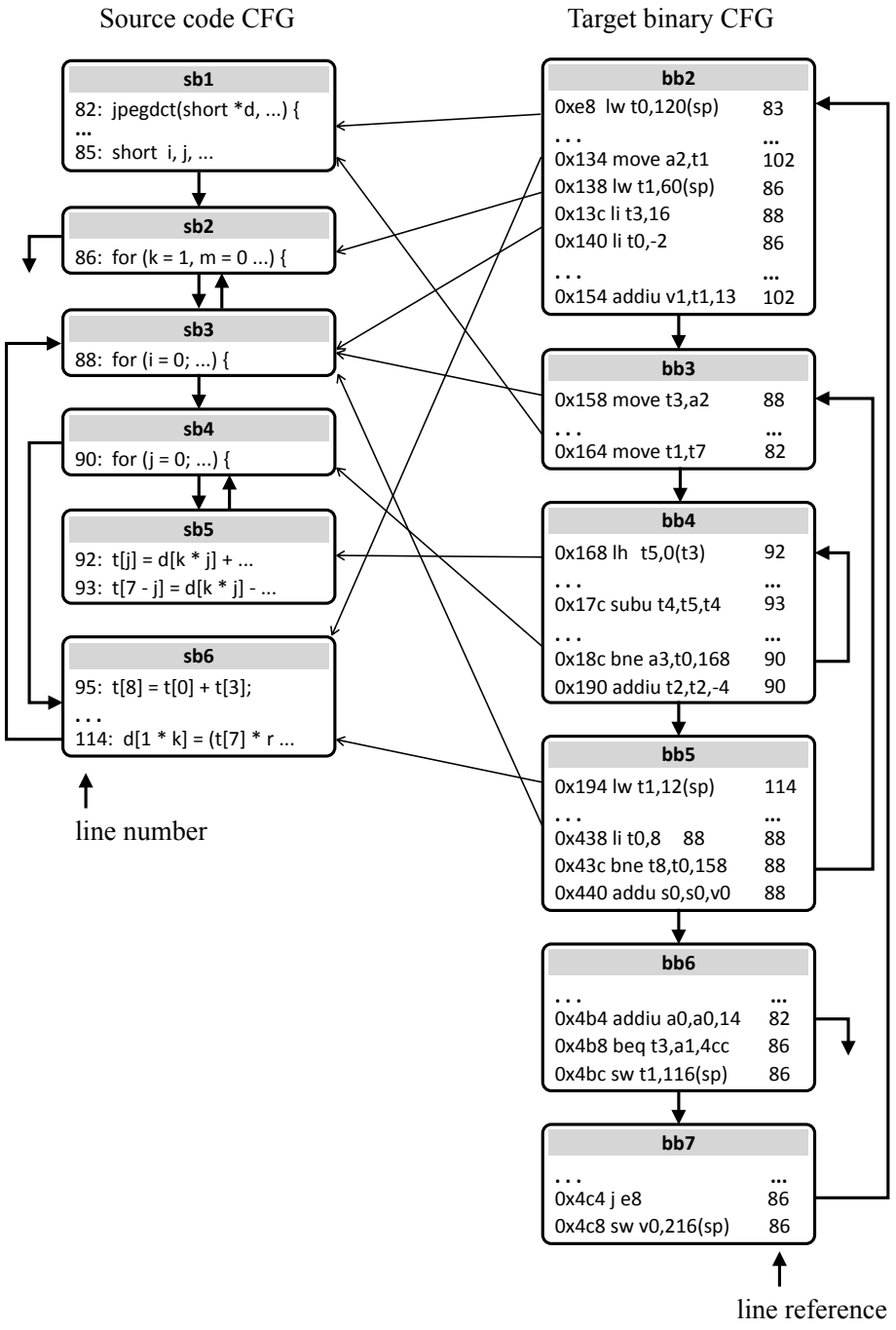


FIGURE 2.2: Ambiguity problem in using the line reference for resolving an annotation position. The line number corresponding to each instruction in the target binary is the reference line from which this instruction is compiled from.

be reliably constructed by inspecting the line reference alone. Structural analysis of the control-flow is required.

2.2.2 Annotate Memory Accesses

Codes related to memory accesses should also be annotated in the source code to enable realistic performance analysis. With them, cache simulation can be performed. At cache

misses, accesses to memory over bus can be simulated, e.g. by initiating transactions. Therefore, annotated memory accesses enable the simulation of inter-module communication. On one hand, this helps in achieving high timing estimation accuracy. On the other hand, it contributes to a HW/SW co-simulation and may aid certain design decisions such as the architectural exploration.

However, the addresses of data memory accesses can not be statically resolved from the target binary code. For example, consider the following instruction in the target binary:

```
sw r1 t1
```

The only extractable information is that it corresponds to a write memory operation to the address represented by `t1`. However the address can not be determined in host-compiled simulation, since it does not interpret the target binary instructions and is thus unable to compute the values of the registers. Without those addresses, data cache simulation can not be performed, leading to a lowered performance estimation accuracy. This problem has been long aware in the area of host-compiled simulation. Yet, it still remained as a major challenge.

2.3 Timing Estimation for TLM+ Transactions

A TLM+ transaction [14] transfers a large data block, abstracting the underlying software protocols. Fundamentals of TLM+ modeling have been introduced in Section 1.3.3. Due to the raised abstraction level, TLM+ complicates the timing estimation. The arising timing problem is twofold.

1. Consider the illustration in Figure 2.3. One TLM+ transaction in fact corresponds to a long sequence of bus-word transactions. But since the underlying software protocols are not represented any more after such abstraction, it is unclear how to estimate the duration of a TLM+ transaction. For example, assuming a TLM+ transaction that transfers 100 bytes to the UART module, what should be its duration? Furthermore, there are various types of driver functions and software protocols. The resulted TLM+ transactions should be timed differently for different cases.

   To tackle this problem, it is required to extract the timing characteristics of the driver functions that implement the low-level software protocols. These timing characteristics can be used later to time the TLM+ transactions.

2. Even if the duration of each TLM+ transaction is known, the bus-word transactions within a TLM+ transaction and their occurrence time are still unknown. These bus-word transactions may cause access conflicts with other bus-word transactions at shared modules. Therefore, it is difficult to estimate the timing of the TLM+ transactions if they overlap with other concurrent processes.

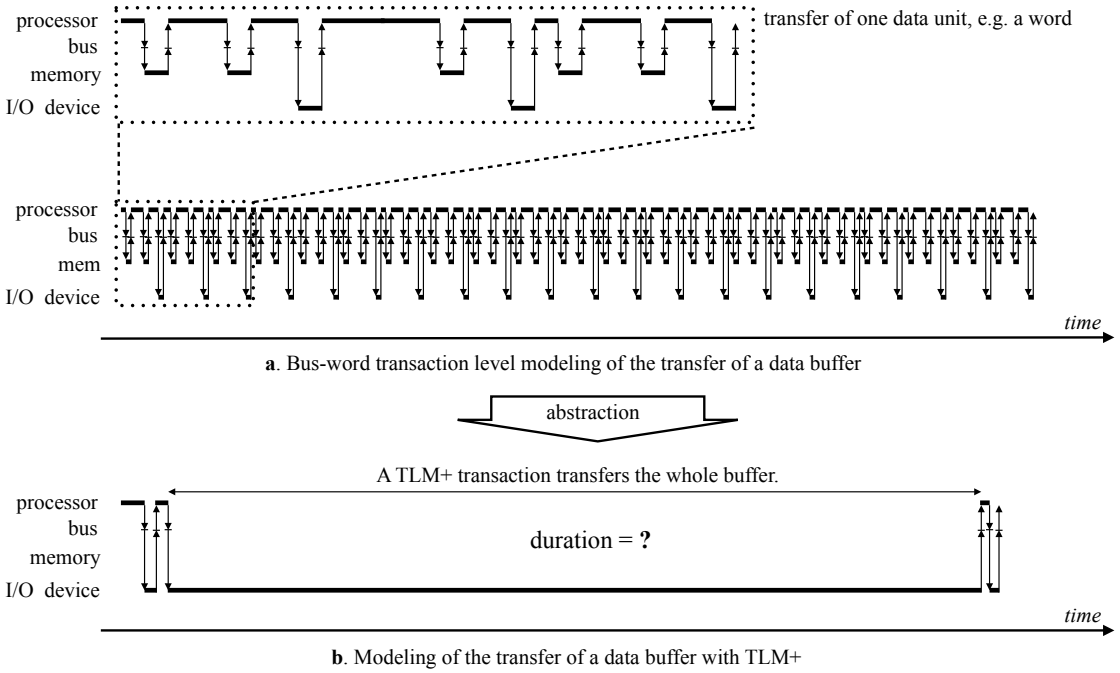


FIGURE 2.3: TLM+ complicates the timing estimation. a) The transfer of a buffer by simulating a driver function that implements the low level software protocols. b) The same transfer is implemented using a TLM+ transaction. As is shown in the dashed box in a), multiple bus-word transactions are required to transfer a single unit of data in this buffer. These bus-word transactions include those for memory accesses, checking the status register of the I/O device, etc. A long sequence of bus-word transactions need to be simulated to complete the data transfer. The larger this buffer is, the more bus-word transactions are evoked.

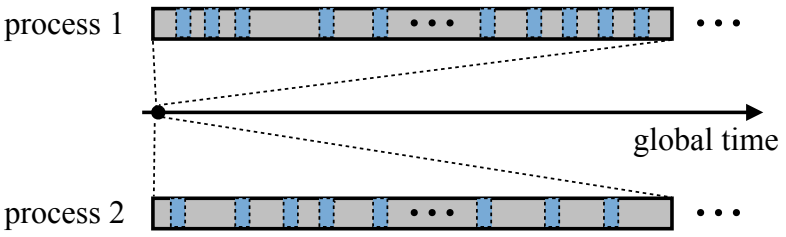


FIGURE 2.4: Timing estimation challenge for TLM+ transactions. Long gray bar (thick line) represents a TLM+ transaction. Blue bars with dashed line represent the bus-word transactions from which the TLM+ transaction is abstracted. A white bar with dashed line represents the delay of a TLM+ transaction.

2.4 Timing Estimation in Temporally Decoupled TLMs

It is straightforward to see the timing problem induced by temporal decoupling. Consider the example in Figure 2.4 that shows two concurrent processes. When temporal decoupling is used, the bus-word transactions are no longer synchronized with the global time. From the viewpoint of the global time, all the bus-word transactions and the functional simulation within the same quantum occur in *zero time*. For concurrent processes such as in multiple processor simulation, it becomes infeasible to detect which bus-word transactions of these processes overlap e.g. at the shared bus. As a result, timing of the conflicting bus-word transactions *can not be arbitrated* by conventional arbitration policy. Timing simulation thus becomes inaccurate. If the transactions are initiated

more frequently, the degree of conflicts may increase and the timing inaccuracy thus becomes higher.

Additionally, using TLM+ transactions implicitly applies temporal decoupling, because timing synchronization is performed only once for a TLM+ transaction that corresponds to a very long time period. But the bus-word transactions within this period are invisible, thus it is not possible to perform standard arbitration at a shared module such as the bus. New mechanisms are required for estimating the delay induced by conflicts at shared modules.

2.5 State of the Art

2.5.1 Annotated Host-Compiled SW Simulation

Almost two decades ago, the idea of using host-compiled software simulation was proposed by Zivojnovic and Meyr [44]. This idea has come to provide an alternative way of software performance analysis that is faster than ISS-based simulation. Since the advent, host-compiled software simulation has been continually researched [45–63].

The related work is chronologically summarized in Table 2.1. A majority of existing approaches aim at annotating performance aspects into a code that will be directly compiled for the host machine. The annotated code thus becomes a performance model of the target binary code. Depending on the format of the annotated code, there are mainly three categories of approaches: (1) binary-level annotation, (2) intermediate-level annotation, and (3) source-level annotation. Besides, there also exist profiling-based approaches. Instead of modeling a specific target binary, they use performance statistics to annotate the source code. All of these approaches are surveyed in the following.

2.5.1.1 Binary-Level Annotation

Approaches using binary-level annotation directly transform the target binary into a performance model for host compiled simulation [45, 46, 69]. Usually, the following steps are involved. Firstly, the source code is cross-compiled into the target assembly instructions. Then, the assembly code is translated into a C code. This C code can be annotated and used as a performance model. Afterwards, the annotated C code can be directly simulated on the host machine. The accuracy of binary-level annotation is not affected by compiler optimization, because the performance is both extracted from and annotated to the assembly code. This is the main reason of the proposal of binary-level annotation.

2.5.1.2 Intermediate-Level Annotation

Some approaches annotate intermediate codes generated by the compiler, as proposed by [47, 50, 54, 55]. The performance aspects can be estimated using the intermediate code. They can also be extracted from the target binary and annotated in the intermediate code. Because the intermediate code is partially optimized, the annotation is relatively robust against compiler optimization. However, when compiling the intermediate code to

TABLE 2.1: Approaches for timing estimation in host-compiled simulation. Cells colored blue are approaches using binary level annotation. Cells colored pink are approaches using intermediate level annotation. Cells colored green are approaches using source level annotation. Cells colored brown are approaches using statistical profiling.

author	96	99	00	01	02	03	04	05	06	07	08	09	10	11	12
Meyr et.al	[44]							[49]	[50]						
Vincentelli et.al		[45]									[52]				
Lee					[47]										
Posadas et.al							[48]						[59]		
Cheug										[51]					
Hwang											[54]				
Rosenstiel et.al											[53]				[62]
Gerin and Petrot												[55]			
Wang												[64]			
Lin													[58]		
Müller and Lu														[60]	
Brandolese				[65]											
Cai								[66]							
Schirner										[67]					
Xu															[68]

the target assembly, the compiler can still apply target-specific optimization. Therefore there still exist mismatches in the control-flow graphs of the intermediate code and the target binary.

2.5.1.3 Source-Level Annotation

Recently, annotating performance information in the original source code has drawn increasing research effort [52, 53, 58, 60, 62]. This is also termed as source-level simulation (SLS) by some literatures. Compared to binary-level annotation or intermediate-level annotation, source-level annotation offers better transparency and is easier to analyze for software designers. It also directly shows the performance figures associated with each part of the source code. As introduced in Section 1.2.3, the annotation is usually performed at the granularity of basic blocks. The early works on source-level annotation do not perform structural analysis in mapping the basic blocks [52, 53, 58], The annotation consults primarily the line reference file. Under the presence of compiler optimization, such annotation can not preserve the correct execution order of the annotated performance modeling codes. The annotated source code thus is not an accurate performance model. To address compiler optimization, researchers have recently resorted to structural analysis [60, 62]. For example, the approach in [62] tries to preserve the dominance relation in mapping a binary basic block to a source basic block. However, compiler optimization often alters the dominance relation among the basic blocks. Therefore preservation of the dominance relation can be insufficient to provide a robust basic block mapping. Besides the dominance relation, this work also examines control-dependency and loop membership of a basic block. They are more adequate in providing a constraint regarding the execution order of a basic block.

2.5.1.4 Timing Extraction Based On Profiling

Besides the work above mentioned, some other approaches model the performance statistically by means of profiling [65–68, 70–72]. The profiling can be constructed at C operation level [65, 66, 71], or task level [67, 68, 72]. For example, a recent approach [68] performs task-based timing estimation. This means the timing information is extracted not from the basic blocks but from the software tasks. The authors first profile each task using Monte-Carlo simulation from which they extract the average timing for each task.

2.5.1.5 Reconstruction of Memory Accesses

Despite some progress, there still lacks a reliable methodology regarding the reconstruction of the memory accesses from the target binary. The addresses for those accesses should be resolved so that data cache simulation can be evoked within the annotated source code. At cache misses, transactions can be initiated to simulate on chip communication, leading to the so-called software TLMs [51, 54, 73–75]. In [48, 60, 61, 76], this problem is alleviated by disabling the use of data cache. Approaches in [54, 58] use random cache misses. But cache miss rate is specific to the executed program and hard to predict. Using random cache misses therefore can lead to timing estimation inaccuracy. In [57], only the addresses of global data are handled. Approaches in [50, 59] use the

addresses of the simulation host to emulate the target memory accesses. However, data locality can be very different in the target binary and the host binary. For example, the stack and static variables can be compiled to very different locations for the host and target machines. Such locality discrepancy can result in large timing errors in case the cache misses are inappropriately simulated. Besides, the data types can be different as well for the host and target machine. For example, the integer may be 8 bytes for the host machine but 4 for the target machine. Therefore if an integer array is sequentially accessed, the number of estimated cache misses using host-compilation would be approximately twice as that on the target machine. Further, many data memory accesses can not be emulated by the addresses of the simulation host, because they are ISA specific and only visible in the cross-compiled binary. For example, when the register values are temporally stored at the stack as in register spilling, the corresponding memory accesses can not be emulated by host machine addresses. In [63], an abstract cache model from worst case execution time (WCET) analysis is used. It uses a range of possible addresses to annotate one data memory access, leading to pessimistic timing estimation. Resolving the memory accesses is further complicated by pointer dereference [55], especially if the pointers are passed as function arguments. To the author's best knowledge, no approach has explicitly considered the pointer dereference problem.

2.5.2 Faster and More Abstract HW Communication Simulation

In standard TLMs which use bus-word transactions, timing is synchronized before a transaction is evoked, so that the transactions of concurrent processes can be arbitrated properly. As the timing synchronization causes costly context switches, there have been ideas toward faster simulation of transaction-level models by means of reducing the number of timing synchronization. For a simulation with such course-grained synchronization, some approaches have been proposed to maintain the timing estimation accuracy. This section surveys the works that aim at expediting the simulation of transaction-level models and corresponding methods regarding the timing simulation.

2.5.2.1 Faster Simulation of Transaction-Level Models

Generally speaking, increasing the abstraction level can be used to achieve higher simulation speed. With regard to the simulation of transaction-level models, the concept proposed in TLM+ increases the level of data-flow modeling from bus-word level to packet or data block level [14]. Conceptually similar to TLM+, other approaches [77–80] also model the hardware system at higher abstraction level. A single abstract transaction is used in [77] to implement end-to-end communication for fast simulation of wormhole switched NoC. As mentioned by the authors, timing accuracy is reduced due to abstraction, since the timing of the abstract transaction is obtained empirically. Approaches in [78–80] further abstract the models at the granularity of tasks. They use task graphs based simulation for evaluating task response times, memory utilization, etc. These approaches often target early course-grained performance estimation and architecture exploration. Usually, they can not adequately estimate the delay due to resource conflicts in TLM+ simulation or TL simulation with temporal decoupling.

In addition to increasing the abstraction level, another way to speed up the simulation is to coarsen the synchronization granularity. This is implemented in the TLM 2.0 standard [15] as a temporal decoupling technique, as introduced in Section 1.3.4.2. Readers are referred to Section 2.4 for the timing problem induced by temporal decoupling.

2.5.2.2 Arbitrated Timing Simulation

Arbitration based approaches usually trace and store the timing of individual transactions. Then they perform arbitration on long sequences of transactions, such as those occurred within the current quantum.

Schirner et al. [81] propose a concept of result-oriented modeling. A conflict-free optimistic duration is firstly used for a long transaction sequence. Afterward, retroactive timing correction is performed successively, until the actual duration is reached. Indrusiak et al.[82] apply the same concept to wormhole network-on-chip, where they store the timing of packet flows and perform timing estimation thereupon.

When temporal decoupling is used, Stattelmann et al. [76] store the occurrence times of all transactions in lists and retroactively traverse the list to arbitrate individual transactions in order to correct the timing and adjust the quantum allocation. Conceptually, this approach is similar to [81]. When the timing lists are very large due to frequent cache misses or I/O accesses, retroactively arbitrating each bus-word transaction may become very expensive. Besides, since the computation and communication models in TLMs often do not provide cycle accuracy, one may argue that, in a timing inaccurate system, it is not always advantageous to perform cycle-accurate arbitration of the bus-word transactions. For example, suppose there exists no timing conflict between two concurrent bus-word transactions in the simulation of a cycle-accurate model. However, in the simulation of a transaction-level model, the occurrence times of these two transactions may be different from those in the cycle-accurate simulation. Therefore, timing conflicts may be observed between these two transactions. In this case, the arbitration may delay one of the transaction which is not delayed in the cycle-accurate simulation.

Another problem for arbitration based approaches is that, retroactive arbitrating bus-word transactions may not always be feasible. For example, a TLM+ transaction is in fact an abstraction of a long sequence of bus-word transactions, whose occurrence times might not be known. Consequently, arbitrating each individual bus-word transaction is not possible. For timing estimation in TLM+ simulation, the authors assume that each TLM+ transaction fully consumes the resources. Due to this assumption, the arbitration shifts an overlapping lower priority transaction to the end of the TLM+ transaction. In the actual case, this transaction can be interleaved with the TLM+ transaction as depicted in Figure 2.5. Clearly, such arbitration leads to over-pessimistic results.

Similar to arbitration based approaches, authors in [83] investigate the predictive timing estimation, for the scheduling of periodic real-time tasks in host-compiled OS modeling. Given the periods of the tasks, this approach predicts the preemption time and the duration of the tasks. Compared to this approach, the present work does not require the periodicity for prediction and is capable of scheduling TLM+ transactions of arbitrary durations.

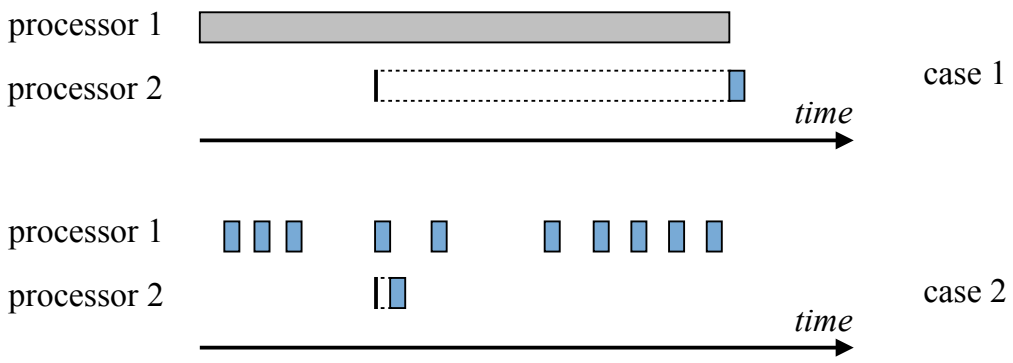


FIGURE 2.5: Timing arbitration of TLM+ transactions. Case 1: assuming fully resource consumption of a TLM+ transaction as in [14]; Case 2: the actual case.

2.5.2.3 Hybrid Simulative and Analytical Approaches

Hybrid approaches combine analytical performance model with simulation model [84–87]. Bobrek et al. [84] analytically estimate the penalty during simulation for concurrent threads that may have resource contention. Details regarding the analytical model however are not given. Authors in [86] interface the simulation model with a formal analysis tool for investigating system-level performance, e.g. the arrival curves, delay, etc. Lahiri et al. [85] evaluate the communication performance from timing parameters extracted in simulation for fast design exploration. Although not proposed for TL models, these approaches shed light on the idea of integrating analytical approaches into simulation models. Still, there exists the need of an effective delay model and a mechanism to handle timing dynamically for TLMs with temporal decoupling.

2.5.2.4 Analytical NoCs

For performance analysis, analytical approaches have gained much popularity in the domain of network-on-chips (NoCs), as can be seen in a large body of research [82, 88–98]. Some of the NoCs are modeled at transaction level. At the granularity of network flows, these approaches aim to analyze or predict the average or worst-case packet latency, router latency, buffer utilization, etc, under certain given traffic pattern. A majority of these approaches apply queuing theory to model the NoC components such as routers [82, 88–97], some apply the Markov chain theory [99]. Besides, a relevant work worth mentioning is the SystemQ proposed by Sonntag et al. [100, 101]. SystemQ is also based on queueing theory and can be used as a superset of SystemC. It enables performance estimation for message passing architectures such as NoCs. Initially, it is used to aid the architectural decision making at an early design phase. With later development, it incorporates the transaction modeling to support TLMs. For this, adaptors are inserted to collect a group of transactions into a message, which is queued at a server and sent as a whole.

When applied to TL models of multiprocessor SoCs, the granularity of packets may be too coarse to achieve appropriate timing estimation accuracy. This is because the transfer of a large data block such as a packet often consists of a sequence of many bus-word transactions, between the two of which there exists a certain time interval. Therefore the packet transfers by different processors can be interleaved at shared resources. Besides, the arbitration in detailed TL models is performed at the granularity

of bus-word transactions instead of TLM+ transactions. Another difference between SoCs and NoCs is that a bus often does not have buffers, while routers usually do. With the above consideration, a more adequate approach is required to estimate the timing of TLMs in bus-based SoCs that are simulated with temporal decoupling. This approach should take into account the traffic patterns within the TLM+ transactions and the corresponding resource access characteristics to analytically estimate the delay due to resource conflicts.

2.5.2.5 Statistical Approaches

Statistical approaches use the statistics traced during simulation to perform timing estimation that is approximated in a statistical sense. Bobrek et al. [102] train a statistical regression model with samples of resource contention. The training effort may be high to accurately model all the contention scenarios. The model also needs to be re-trained for each different design. In application, their approach is used for host-compiled SW simulation on a multi-processor TLM. But a scheduler for dynamic timing rectification is not investigated. It assumes known resource utilization in simulation, which is not usually the case due to cache effects and the abstraction of buffer transactions.

Other approaches analyze the system performance under certain traffic distribution [103, 104]. The goal is to achieve fast and early architecture exploration. For this purpose, it suffices to estimate the timing approximately. A clear statistical model of the resource conflicts is not proposed. Another approach in [105] integrates a SystemC-based framework with a statistical memory access model to predict the performance. Authors in [106] statistically characterize the workload to handle the increased design complexity of heterogeneous multiprocessors. The resource conflicts at shared resources are not explicitly modeled. These approaches are not directly applied to simulate a specific application program or use-case, for which the actual traffic pattern should be simulated and not statistically assumed.

Qian et al. [107] propose a machine learning based performance analysis framework for NoCs. Specifically, they use a support vector regression model to predict traffic flow latency. They need to train their model to learn the typical traffic patterns and the average packet latency. For accurate prediction, the main features of traffic patterns should be covered in the training data. The learning curve and the implementation effort of porting to existing TLMs may be high.

2.5.2.6 Parallel Simulation

Also aiming at expediting the simulation speed, another line of research explores parallel simulation [108–114]. Authors in [108] present their implementation of general mechanisms such as synchronization to simulate a single SystemC model in parallel. Adjustable quantum size is implemented, although no specific handling of timing conflicts is mentioned. In [109], parallel scheduling of multiple simulator is studied, with the focus is on ensuring the correct temporal order of the discrete events received by each simulator. In addition to scheduling, they [110] further consider the effect of load balancing for parallel simulation. TLM with distributed timing (TLM-TD) is conceptualized in [111] as a means to keep the notion of timing in parallel simulation. Loosely-timed TLM with temporal decoupling is also supported, but no contention modeling is considered. Static

analysis at compile time is applied in [113] in order to achieve an out-of-order parallel simulation, where the execution of parallel threads does not strictly follow the order of events that those threads are sensitive to. In [114], more course grained parallel simulation is used at task level, where each task is assigned a duration. Authors' emphasis is on an unmodified SystemC kernel.

2.5.2.7 Consider the Data Dependency

A general issue of transaction-level models with temporal decoupling and parallel simulation is the detection and handling of data-dependency [115–118]. Authors in [115] put forth that certain codes can not be simulated with temporal decoupling, since functional correctness would otherwise be lost. By identifying those codes, they propose a hybrid simulator which can integrates both TLM-TD and standard TLM simulation, in which host-compiled simulation and ISS simulation are used respectively. A data-dependency table in [117] is used to predict the memory accesses, in order to avoid unnecessary synchronization. Used in trace-driven simulation [116], data-dependency is checked to determine the necessity of synchronization and scheduling of the processes. Other approaches investigate TLMs with adaptive timing accuracy [119, 120]. A formal method [118] uses model checking to identify race condition in SystemC models. The results can then be used in scheduling the processes. In the present approach, it is assumed that there is no data dependency in the processes that are simulated in a temporally decoupled way. In fact, even for standard TLM simulation without temporal decoupling, data dependency can not be guaranteed to be captured, since TLMs can not be absolutely timing accurate anyway. However, if data dependency needs to be considered, the above mentioned approaches can be investigated.

2.6 Contributions

Corresponding to the scope of this work shown in Figure 2.1 and the confronted challenges, the contributions of this work focus on tackling these challenges. An overview of the proposed methodologies is given in the following.

- For host-compiled software simulation, a methodology is presented that can reliably annotate the source code with the performance modeling codes extracted from an optimized target binary code. This tackles the challenges in Section 2.2.
- For timing estimation of TLM+ transactions, a method is proposed that can provide accurate timing characteristics of the TLM+ transactions. This tackles the challenges in Section 2.3.
- For resolving the delay due to resource conflicts in the simulation with temporal decoupling, an analytical timing model and an efficient scheduler is developed. This tackles the challenges in Section 2.4.

These methodologies proposed in this work contribute to the improvement of the simulation accuracy while maintaining high simulation speed. In the following, each contribution is described in more detail.

2.6.1 A Methodology for Annotating the Source Code

To enable performance estimation in host-compiled software simulation, performance modeling codes need to be annotated in the source code. Example of the annotated codes include the execution cycles and the memory accesses. A confronted challenge is to resolve a correct position in the source code to annotate the performance modeling codes extracted from an optimized binary code. This work performs detailed structural analysis on the control-flows of both the source code and the target binary. The performed analysis can identify the dominance relation, the post-dominance relation, the control dependency, and the loop membership for the basic blocks. Similar analyses are also performed by the compiler in the code optimization. After these analyses, structural properties are extracted for the basic blocks in each control-flow graph. Extracted structural properties are used as constraints in mapping the basic blocks in the source code and the target binary code. Because the control dependency represents the condition under which a basic block will be visited, matching the control dependency thus ensures that the annotated codes in the source code are executed under the same condition as the corresponding basic blocks in the target binary code. The resulting source code annotation is thus robust against compiler optimization. Previous approaches mainly rely on the line reference that is unreliable under compiler optimization. To the best of the author's knowledge, this work is the first of its kind to utilize such thorough structural analysis of the control-flows in annotating the source code. Details of the control-flow analysis are shown in Section 3.2, and the subsequent basic block mapping is given in Section 3.3.

Another difficulty in annotating the source code is that the addresses of data memory accesses can not be determined by static analysis of the target binary code. This challenge is tackled in this work by exploiting the memory allocation mechanism. The memory accesses are differentiated into several cases corresponding to the accesses of variables in the stack, the variables in the heap, and the global or static variables. These variables are located in different memory sections. Their addresses are constructed according to such memory allocation principle. This results in high agreement to those in the target binary. Besides, a method for resolving the addresses of the memory accesses caused by pointer dereferences is also proposed. The effectiveness of the presented method is that it exploits the same memory allocation mechanism that is respected by both the compiler and the execution of a program, thus guaranteeing the correctness of the reconstructed memory accesses. The work is elaborated in Section 3.4.

2.6.2 Construct Timing Profiles for TLM+ Transactions

This dissertation presents a simple and yet efficient method to time the TLM+ transactions. This method begins with a tracing and profiling step in which the timing characteristics of the driver functions that implement the low-level software protocols in the data transfer. The contribution is a simple idea regarding the tracing of the start and end of those driver functions. This idea is to use the *entry* and *exit* instruction addresses of those driver functions. These addresses can be extracted from the target binary with the help of debuggers. A tracer in the ISS simulation checks the address of the simulated instruction to accurately determine start and end of the considered driver functions. Based on this, a timing library can be constructed for those driver functions. A TLM+ transaction therefore can be timed properly by querying this library. Besides,

this tracing and profiling mechanism is general and can provide versatile results for other purposes. The developed tool-chain for the tracing and profiling is described in Section 4.3.

2.6.3 Analytical Timing Estimation for Temporally Decoupled TLMs

The challenge of timing estimation for TLMs with temporal decoupling is tackled in this dissertation by a novel analytical timing estimation mechanism. It introduces a timing model, in which

- The HW modules are modeled as resources.
- The access of a resource is modeled by resource utilization.
- The allocation of a resource is modeled by resource availability.
- The timing conflicts at shared resources are modeled by a delay formula that takes into account the resource utilization and resource availability.

To combine the analytical timing estimation with TLM simulation, a fast scheduling algorithm is implemented. The scheduler is provided within an additional resource model library that can be easily ported to existing TLMs. Due to the use of dynamic event cancellation, this scheduler needs only one *wait* statement to schedule one timing synchronization request of a very long time period. Therefore the simulation overhead introduced by the scheduling algorithm is very low. This work in effect achieves hybrid analytical and simulative timing estimation, where timing characteristics are extracted online and analytical formula are then used for fast timing prediction. Describing this analytical timing estimation methodology is the subject of Chapter 4.

2.6.4 Summary of Contributions

The key novelty and uniqueness of the proposed methodologies are briefly summarized:

- Thorough control-flow analysis for accurate source code annotation
- Efficient tracing and profiling for TLM+ transactions
- Novel formulas and a scheduling algorithm for fast timing estimation

Each of these ideas is among the first in the corresponding area of research.

2.6.5 Previous Publications

Initially, the work in this dissertation was part of the SANITAS project. The objective is to solve the problems in timing estimation of TLM+ transactions. A preliminary solution of this problem is proposed in [121]. More detailed solution and results are published in [122]. A resulted tracing and profiling tool can provide timing characteristics for TLM+ transformations. This part of the work basically accomplished the

original objective. In addition, the timing estimation technique is further extended for TLMs with concurrent processes and thus resource conflicts. An analytical approach is developed that applies formulas to model resource utilization, resource availability and the delay [123]. A scheduler based on the resource model [124] integrates the analytical estimation into the simulation of TLM+ transactions or TLMs with temporal decoupling.

In parallel to the above work for the project, another body of this dissertation contributes to the topic of source code annotation for host-compiled software simulation. The paper [60] proposes a novel idea of exploring control-flow analysis in tackling the compiler optimization problem. For the nodes in a control-flow graph, it can extracts structural properties such as the dominators, post-dominators, control dependency, and loop membership. Based upon this paper, follow-up work has been done to consider the reconstruction of data memory accesses for data cache simulation [125], incorporation of specific compiler optimizations [126], faster cache simulation [127], and application to a two-wheeled robot simulation platform [128].

The above two lines of work are compatible and can be incorporated together in the practical software simulation or full system simulation. The application codes, which represent the computation aspect of the software, can be simulated with host-compiled simulation, while the TLM+ transactions can be called for data transfers which represent the communication aspect. Another publication [129], although not tightly related to the above ones, modifies the SystemC kernel and aims at removing unnecessary context switches.

Chapter 3

Source Code Annotation for Host-Compiled SW Simulation

The aim of annotating the source code is to turn the annotated source code into a performance model of the target binary. This requires the embedding or reconstruction of the control flow of the target binary into the source code, the execution of which hence can resemble that of the target binary. To do this, a mapping between the basic blocks of the target binary code and the source code needs to be constructed. It may become difficult to find this mapping if the target binary has been optimized by the compiler. Reliably handling a target binary code with a structurally different CFG requires structural exploration of both the CFGs of the source code and the target binary code. Besides, even for a perfect mapping, it is still difficult to annotate the memory accesses for cache simulation, since the accessed addresses can not be statically known.

This chapter tackles the above mentioned challenges. Firstly, Section 3.1 introduces the considered structural analysis and terminologies. Then, Section 3.2 discusses the examined structural properties. Following that, Section 3.3 details the basic block mapping procedure using the structural properties. Afterwards, Section 3.4 presents a novel approach that exploits the memory allocation mechanism in determining the addresses of the memory accesses. Finally, Section 3.4 elaborates the implementation and usage of the tool-chain developed in this work.

3.1 Structural Control Flow Analysis

The control flow graph $G(N, E)$ is a directed graph representing the control flow of a program, with N being the set of nodes representing the basic blocks and E being the set of edges representing the execution order of those basic blocks. For a given CFG, structural analysis known from compiler theory [130, 131] lays a foundation of the proposed approach. Therefore, as a preliminary, this section introduces the investigated structural analysis and the extracted structural information, which will be used in constructing the basic block mapping in the next section. The sample CFG is shown in Figure 3.1(a). There is an auxiliary edge between the *entry* and *exit* node

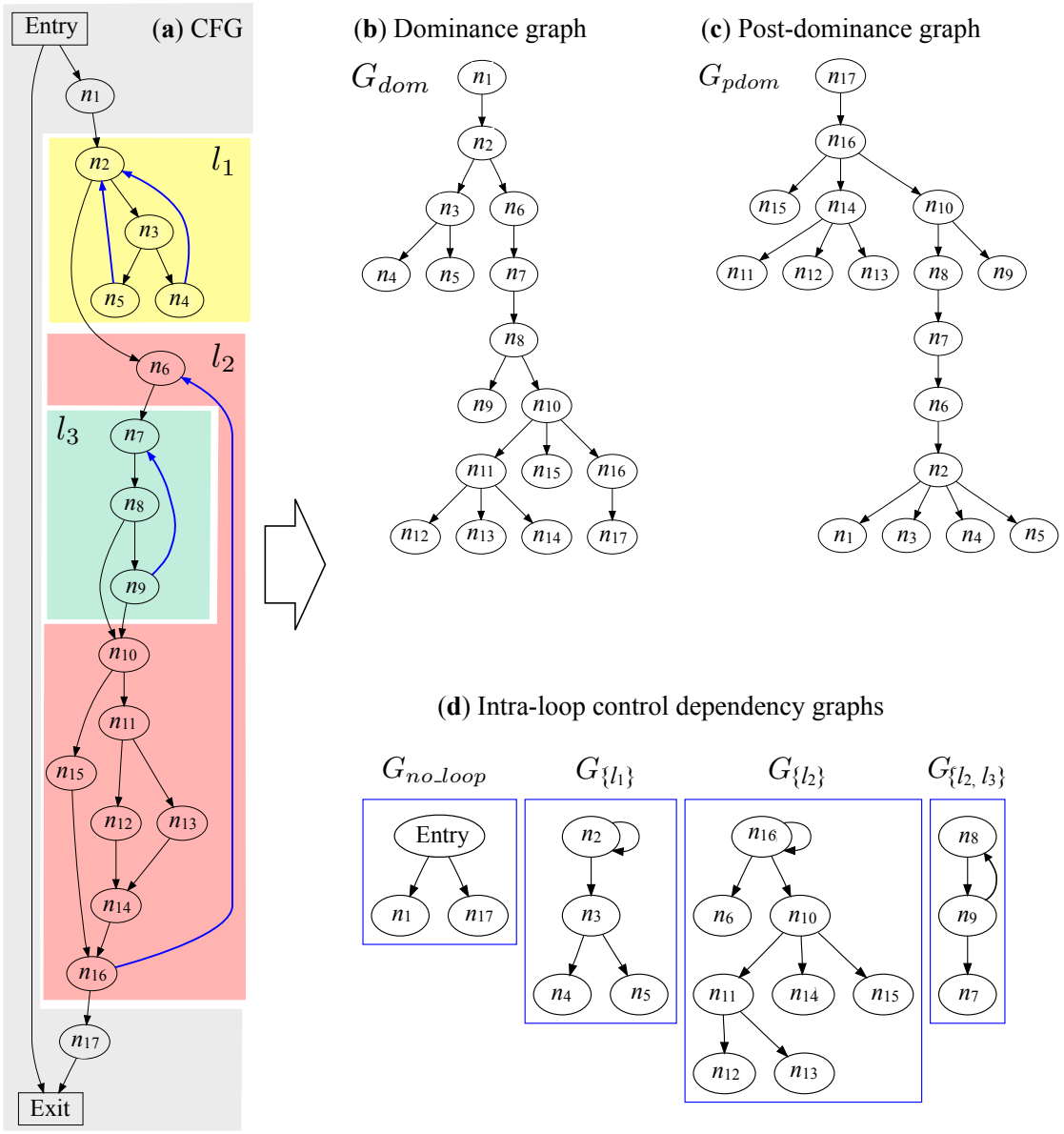


FIGURE 3.1: Sample CFG and the resultant graphs given by structural analysis.

3.1.1 Dominance Analysis

Definition 3.1.1. Node n_k dominates node n_i, if n_k is part of all paths in the CFG from the *entry* node to node n_i.

By definition, each node dominates itself. The set of all nodes that dominate node n_i is denoted by

$$dom(n_i).$$

The dominator sets for all the nodes are computed with the algorithm in [132]. If node n_i dominates node n_j and node n_j dominates node n_k, then node n_i also dominates node n_k. Therefore the domination relation is transitive. The resulting dominance graph G_{dom} is a tree, as shown in Figure 3.1(b). For simplicity, the reflexive edges corresponding to the self-dominance relation are omitted in the dominance graph.

Definition 3.1.2. *Immediate dominator* of node n_i is its parent node in the dominance graph.

It is denoted by

$$idom(n_i).$$

The immediate dominator of a node is either unique or nil (for the entry node).

3.1.2 Post-Dominance Analysis

Definition 3.1.3. Node n_k post-dominates node n_i, if n_k is part of all paths from the node n_i to the *exit* node.

By definition, each node post-dominates itself. The set of all nodes that post-dominate node n_i is denoted by

$$pdom(n_i).$$

The post-dominator sets can be obtained by applying domination analysis to the reverse CFG. If node n_i post-dominates node n_j and node n_j post-dominates node n_k, then node n_i also post-dominates node n_k. Therefore the post-domination relation is transitive. The resultant post-dominance graph G_{pdom} is a tree, as shown in Figure 3.1(c). For simplicity, the reflexive edges corresponding to the self-post-dominance relation are omitted in the post-dominance graph.

Definition 3.1.4. *Immediate post-dominator* of node n_i is its parent node in the post-dominator tree.

It is denoted by

$$ipdom(n_i).$$

The immediate post dominator of a node is either unique or nil (for the exit node).

3.1.3 Loop Analysis

The considered loops of a CFG in the present work correspond to the iterative program structures that are declared by specific programming instructions such as *for* and *while* statements. A loop represents a part of the code that can be repeatedly executed. A loop in the CFG is denoted by l_p. There are several terms associated with the loop analysis. Details of these terms can be found in [130, 131] They are briefly explained in the following:

Definition 3.1.5. Back edge: An edge $(n_i, n_j) \in E$ is a back edge, if $n_j \in dom(n_i)$.

Each back edge corresponds to a loop, while a loop may have several back edges.

Definition 3.1.6. The target node of a back edge is called the *loop head*.

Definition 3.1.7. The source node of a back edge is called the *latching node*.

For example, (n_5, n_2) and (n_{16}, n_6) are two back edges corresponding to loop l_1 and l_2 respectively.

Definition 3.1.8. Natural loop: The natural loop, i.e. loop body, identifies those nodes that are part of the loop. It consists of the set of nodes that can reach the latching node without passing through the head node plus the latching node and the head node.

For loop l_p, its natural loop is denoted as

$$N(l_p)$$

For example, $N(l_3) = \{n_7, n_8, n_9\}$.

Definition 3.1.9. *Exit edge:* Edge (n_i, n_j) exits a loop l_p, if $n_i \in N(l_p)$ and $n_j \notin N(l_p)$.

For example, (n_{16}, n_{17}) exits loop l_2.

Definition 3.1.10. Exit nodes: The exit nodes of a loop are the source nodes of all its exit edges.

For loop l_p, its exit nodes are denoted as:

$$X(l_p).$$

They represent a set of branch conditions that can exit a loop. In Figure 3.1, $X(l_1) = \{n_2\}$, $X(l_2) = \{n_{16}\}$, $X(l_3) = \{n_8, n_9\}$

Definition 3.1.11. Entry edge: Edge (n_i, n_j) enters a loop l_p if $n_i \notin N(l_p)$ and $n_j \in N(l_p)$.

For example, (n_1, n_2) enters loop l_1.

Definition 3.1.12. *Nested loops:* l_p is nested in l_n if $N(l_p) \subset N(l_n)$.

Definition 3.1.13. *The parent loop* of a node is the innermost loop that encloses this node. For example in Figure 3.1, the parent loop of n_7 is l_3.

3.1.4 Control Dependency Analysis

Definition 3.1.14. An edge $(n_i, n_j) \in E$ is a **control edge**, if $n_j \notin pdom(n_i)$.

In Figure 3.1, (n_2, n_3) is a control edge since $n_3 \notin pdom(n_2)$, but (n_2, n_6) is not since $n_6 \in pdom(n_2)$.

Definition 3.1.15. The source node of a control edge is called the *controlling node.*

With respect to the program execution, the controlling node represents a branch condition, whose evaluation controls whether its target basic blocks will be executed. Therefore it effectively constrains the execution order of the basic blocks.

Definition 3.1.16. The **control dependent nodes** for a given control edge are those nodes that will be visited if and only if that control edge is taken.

For control edge (n_i, n_j), its control dependent nodes can be determined by the algorithm in [131], which inspects the post-dominator tree and checks for two cases:

1. If n_i is an ancestor of n_j, then the dependent nodes include the nodes on the path from n_i to n_j in G_{pdom} plus n_i and n_j;

TABLE 3.1: Example of the determination of control dependent nodes

case	control edge	dependent nodes
(1)	(n_{16}, n_6)	$\{n_6, n_7, n_8, n_{10}, n_{16}\}$
(2)	(n_{10}, n_{11})	$\{n_{11}, n_{14}\}$

TABLE 3.2: Example properties of nodes in Figure 3.1

nodes	$P_m(n_i)$	$P_c(n_i)$	$P_b(n_i)$
n_1	$\{\}$	$\{Entry\}$	$Entry$
n_4	$\{l_1\}$	$\{n_3\}$	n_3
n_8	$\{l_2, l_3\}$	$\{n_9\}$	n_2
n_9	$\{l_2, l_3\}$	$\{n_8\}$	n_8
n_6	$\{l_2\}$	$\{n_{16}\}$	n_2
n_{10}	$\{l_2\}$	$\{n_{16}\}$	n_8
n_{16}	$\{l_2\}$	$\{n_{16}\}$	n_{10}

2. If n_i is not an ancestor of n_j, then the dependent nodes include the nodes on the path from n_j to the first common ancestor n_a of n_i and n_j including n_j but not n_a.

Examples corresponding to these two cases are given in table 3.1. Further, given a branch node n_i, the nodes that are control dependent on n_i are the union of the control dependent nodes for all the control edges that branch from n_i. For example, n_{11}, n_{14} and n_{15} are control dependent on n_{10} in Figure 3.1.

3.2 Structural Properties

Notice that the superscript s and b are used to distinguish the notation corresponding to the source code and the target binary code respectively. For example, G^s and G^b denote the CFG of the source code and the target binary code respectively. n_i^s denotes node i in G^s, and n_j^b denotes node j in G^b. After the structural control-flow analysis, extracted structural properties for a node in the CFG include its loop membership, its intra-loop controlling nodes, and its immediate branch dominator. Derivation of each property is given in the following.

3.2.1 Loop Membership

Node n_i is enclosed in loop l_p if it is in the loop body (natural loop) of l_p. Loop membership of a node is all the loops that enclose this node. It is used as a structural

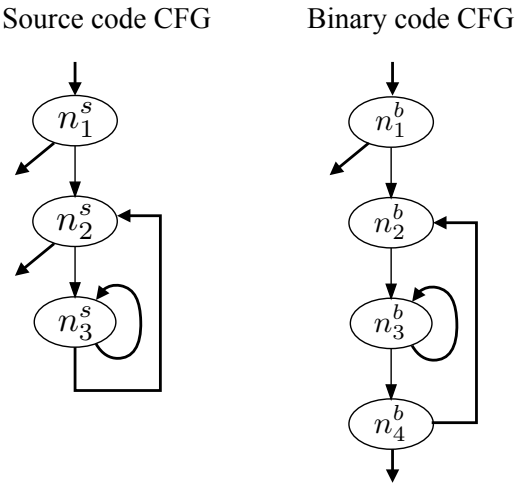


FIGURE 3.2: Using intra-loop control dependency as a mapping constraint is more robust against compiler optimization than the raw control dependency

property of a node and is denoted as:

$$P_m(n_i) = \{l_p | n_i \in N(l_p)\}. \tag{3.1}$$

Examples are given in the second column Table 3.2.

3.2.2 Intra-Loop Control Dependency

The classic control dependency graph is obtained by connecting a controlling node to all its control dependent nodes [131]. The obtained control dependency of a node is its dependency on the controlling nodes in the whole control-flow graph. Different from this, this work proposes a new type of control dependency. It takes the loop membership of the nodes into account and derives the intra-loop control dependency.

Definition 3.2.1. *Intra-loop control dependency* is the control dependency of node n_j on a controlling node n_i, if they have the same loop membership, i.e., $P_m(n_i) = P_m(n_j)$.

To show the difference between the control dependency and intra-loop control dependency, consider the controlling node n_{16} in Figure 3.1, for which

- the control dependent nodes are $\{n_6, n_7, n_8, n_{10}, n_{16}\}$, according to [131];
- the intra-loop control dependent nodes are $\{n_6, n_{10}, n_{16}\}$.

Applying intra-loop control dependency in effect confines the consideration of control dependency to those nodes with the same loop membership. Compared to the raw control dependency, it is more robust against compiler optimization in terms of mapping the basic blocks. The reason is that control dependency for nodes within a loop can be easily altered if this loop or or another nesting loop have been transformed by the compiler. Consider the example in Figure 3.2. The controlling nodes for n_3^s in the source code CFG are $\{n_2^s, n_3^s\}$. But for n_3^b in the binary code CFG, they are $\{n_1^b, n_3^b, n_4^b\}$. This unmatched control dependency impedes the mapping from n_3^b to n_3^s.

Intra-loop Control Dependency Graph

Based on intra-loop control dependency, the control dependency graph can be obtained by connecting a controlling node n_i to its intra-loop dependent nodes. As can be seen in Figure 3.1(d), the result is a set of control dependency graphs corresponding to the set of loop memberships.

The notation $G_{P_m(n_i)}$ is used to refer to a control dependency graph that corresponds to the loop membership of node n_i. In Figure 3.1(d), the loop membership of n_3 is $P_m(n_3) = \{l_1\}$, hence $G_{\{l_1\}}$ denotes the control dependency graph that encompasses all the nodes that have the loop membership $\{l_1\}$.

Intra-loop Controlling Nodes

Using intra-loop control dependency, another structural property can be obtained for each node:

Definition 3.2.2. Intra-loop controlling nodes $P_c(n_i)$ of n_i is simply its parent nodes in the control dependent graph $G_{P_m(n_i)}$, expressed as

$$P_c(n_i) = \{n_j | n_i \in edge(G_{P_m(n_i)})\}, \tag{3.2}$$

where $edge(G_{P_m(n_i)})$ is the set of edges in $G_{P_m(n_i)}$. Usually, $P_c(n_i)$ contains only one node. But it can contain more than one node, if a loop has multiple break statements or if branches have a joint target in the optimized binary code. Examples of the intra-loop controlling nodes are given in Table 3.2.

3.2.3 Immediate Branch Dominator

For the mapping process in next section, another structural property is extracted:

Definition 3.2.3. The **immediate branch dominator** $P_b(n_i)$ is the first encountered ancestor of n_i in G_{dom} that is a branch node in the original CFG.

The following recursive function calculates $P_b(n_i)$:

function $selectBranchDominator(n_i)$:
$n_j = parent(n_i)$ in G_{dom}
if $is\_branch(n_j)$ in G **then**
  return n_j
else
  return $selectBranchDominator(n_j)$
end if

Branch dominator restricts the position of a node among other nodes with the same intra-loop control dependency, thereby facilitating the mapping procedure in the next section. For example in Figure 3.1, n_6, n_{10} and n_{16} have the same intra-loop controlling nodes, namely $\{n_{16}\}$. But they have different immediate branch dominators, as can be seen in Table 3.2.

3.3 Basic Block Mapping Procedure

The mapping procedure consists of two main steps. Firstly, it checks the line reference to relate the structural properties of nodes in N^s and N^b. Secondly, it implements an algorithm to successively select the appropriate mapping.

3.3.1 Line Reference From Debug Information

Debugging information provides a line reference file that gives the sources lines from which an instruction is compiled, as depicted in Figure 3.3. The line references can be expressed by a bipartite graph

$$G_R(N^s, N^b, E_R),$$

where:

- E_R is the set of mapping edges from N^s to N^b according to the line reference. Among the edges in E_R, a subset is distinguished:
- $E_B \subset E_R$ denotes those mapping edges that map the branch nodes in N^b by the line reference of the branch instructions.

For the example in Figure 3.3, it follows

$$E_R = \{(n_1^b, n_2^s), (n_1^b, n_3^s), (n_2^b, n_1^s), (n_3^b, n_2^s), (n_3^b, n_4^s)\}$$

$$E_B = \{(n_1^b, n_2^s), (n_3^b, n_4^s)\}.$$

3.3.2 Matching Loops

Loop l_m^b matches loop l_n^s if the branch instructions contained in the exit nodes of l_m^b are compiled from the exit nodes of l_n^s, formally written as

$$\forall n_i^b \in X(l_m^b)\ \exists n_j^s \in X(l_n^s)\ ((n_i^b, n_j^s) \in E_B) \tag{3.3}$$

In other words, two loops are matched if they are exited under the same condition. In Figure 3.3, l_1^b matches l_1^s because the branch instruction (at address `0xac`) in the exit node of l_1 is compiled from line `102` in the exit node of l_1^s. The matched loop l_m^b is denoted by a mapping M_L:

$$l_m^b \mapsto M_L(l_m^b) \tag{3.4}$$

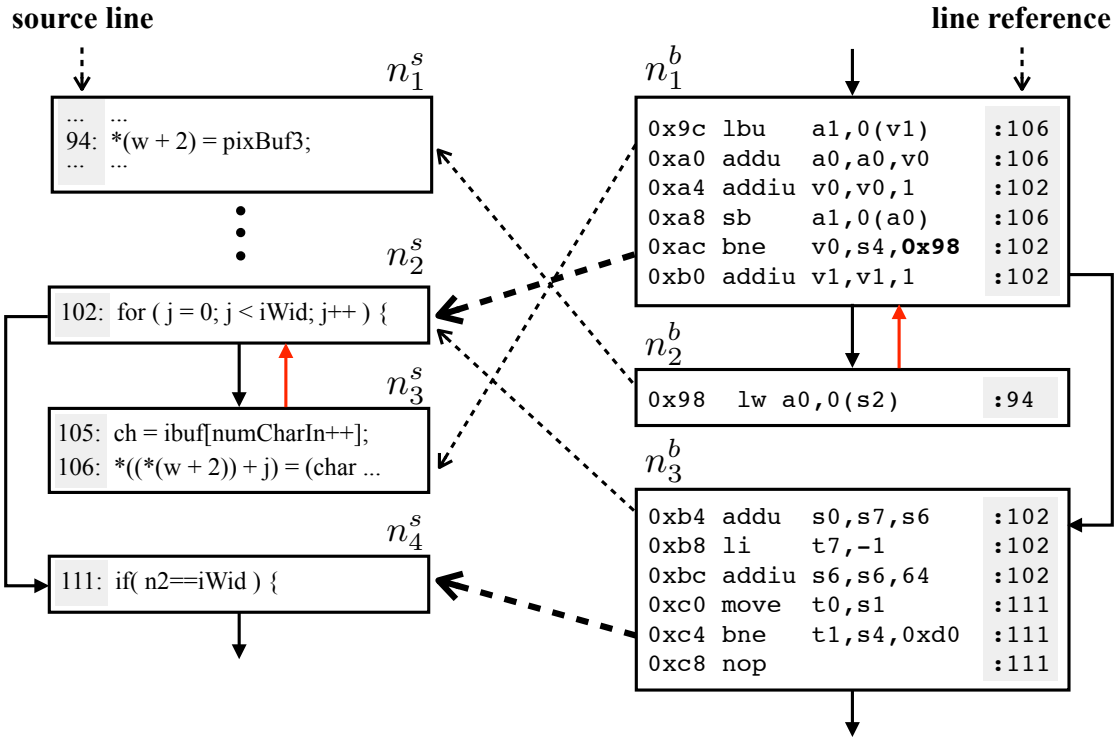


FIGURE 3.3: Example of the line reference provided by the debugger. (left) CFG of source code; (right) CFG of the target binary code. Dashed lines: mapping by the line reference (E_R). Thick dashed lines: line reference mapping (E_B) for the branch instructions.

3.3.3 Translate the Properties of Binary Basic Blocks

Using E_B and M_L, the structural properties of a binary basic block can be translated into a representation using basic blocks and loops in the source code.

$$P_m^s(n_i^b) = \{l_n^s \mid (l_m^b, l_n^s) \in M_L \wedge l_m^b \in P_m(n_i^b)\} \tag{3.5}$$

$$P_c^s(n_i^b) = \{n_j^s \mid (n_k^b, n_j^s) \in E_B \wedge n_k^b \in P_c(n_i^b)\} \tag{3.6}$$

$$P_b^s(n_i^b) = \{n_j^s \mid (n_k^b, n_j^s) \in E_B \wedge n_k^b = P_b(n_i^b)\} \tag{3.7}$$

The translated properties can be subsequently used in the comparison with the properties of a source basic block.

3.3.4 Selection Using Matching Rules

Determining the map for n_i^b is guided by several selection processes. Standard notation from relational algebra [133] is used to define a selection:

Definition 3.3.1. *Selection* is a unary operation written as

$$\sigma_\phi(N_a^s), \tag{3.8}$$

where N_a^s is a set of source basic blocks to select from, and ϕ is a propositional formula representing a matching rule of a specific structural property. The selection in (3.8)

selects all n_i^s from N_a^s for which ϕ holds. If ϕ matches a specific structural property, then applying (3.8) will select a subset from N_a^s as a corresponding matching set for this property. The examined matching rules include:

- Selection rule ϕ_m matches the loop membership:

$$\phi_m := P_m(n_j^s) = P_m^s(n_i^b) \tag{3.9}$$

  Matching loop membership ensures that the delay due to executing a basic block is accumulated correctly over iterations. An often seen case is that compiler can move iteration invariant code before the loop body. By checking loop membership, timing of these binary code will be annotated in the correct loop region.

- Selection rule ϕ_c matches the intra-loop controlling nodes:

$$\phi_c := P_c(n_j^s) = P_c^s(n_i^b) \tag{3.10}$$

  This rule is important in preserving the correct execution order of the annotated codes, because control dependency implies when a node is reached. Matching the intra-loop controlling nodes ensures that the annotated performance modeling code is executed under the same condition as in the target binary.

- Selection rule ϕ_b matches the immediate branch dominator:

$$\phi_b := P_b(n_j^s) = P_b^s(n_i^b) \tag{3.11}$$

  After applying the selection rules (3.10) and (3.9), this rule may be further checked to resolve a unique mapping node in case ambiguity still exists. For an example of when this rule is needed, the reader is referred to Section3.2.3.

- Selection rule ϕ_r matches line reference:

$$\phi_r := (n_i^b, n_j^s) \in E_R \tag{3.12}$$

  After previous selections that match structural properties, more than one source basic block candidates may exist. Line reference could be used as an auxiliary constraint in determining the final mapping.

3.3.5 The Mapping Procedure

For a node n_i^b in $G^b(N^b, E^b)$, the task is to find a mapping $\mathcal{M}_*(n_i^b)$ in $G^s(N^s, E^s)$, so that the performance modeling code of n_i^b could be annotated to the source node in $\mathcal{M}_*(n_i^b)$. The steps in determining this mapping are outlined and discussed in the following.

Step 1a: From all the source basic blocks N^s, select those basic blocks by matching the intra-loop controlling nodes (rule ϕ_c):

$$N_c \leftarrow \sigma_{\phi_c}(N^s)$$

Step 1b: From N_c, select those basic blocks by matching loop membership (rule ϕ_m):

$$N_m \leftarrow \sigma_{\phi_m}(N_c)$$

Step 1c: If $|N_m| = 1$, a unique match is found, thus $\mathcal{M}_*(n_i^b) \leftarrow N_m$, and the algorithm stops. If $|N_m| = 0$, go to step 4. If $|N_m| > 1$, there are multiple source basic blocks with the same intra-loop controlling nodes and loop membership as n_i^b. Further select from N_m those basic blocks by matching the immediate branch dominator (rule ϕ_b), giving a refined matching set:

$$N_b \leftarrow \sigma_{\phi_b}(N_m)$$

Step 2: Inspect N_b: If $|N_b| = 1$, then $\mathcal{M}_*(n_i^b) \leftarrow N_b$ and the algorithm stops. If $|N_b| > 1$, go to step 3. If $|N_b| < 1$, overwrite N_b with N_m and go to step 3.

Step 3: Further select from N_b those basic blocks by line reference (rule ϕ_r):

$$N_r \leftarrow \sigma_{\phi_r}(N_b)$$

After previous selection, N_b contains only those basic blocks that are executed under similar conditions as n_i^b. In other words, it has ruled out those inappropriate basic blocks that may be provided by the incorrect line references of the instructions contained in n_i^b. The resulting N_r from this selection therefore disregards those incorrect line references. If $|N_r| = 1$, a unique match is found, thus $\mathcal{M}_*(n_i^b) \leftarrow N_r$. If $|N_r| = 0$ or $|N_r| > 1$, n_i is added to a pending set for later processing. How this pending process works is explained with an example in Figure 3.4. Suppose $N_b = \{n_2^s, n_3^s\}$ for n_2^b after previous selections. Since both $(n_2^b, n_2^s) \in E_R$ and $(n_2^b, n_3^s) \in E_R$, N_r is still $\{n_2^s, n_3^s\}$. So n_2^b is added to the pending set. Afterwards the pending set is checked. Since n_3^b is mapped to n_3^s and n_2^b is mutually exclusive to n_3^b, n_2^b can only be mapped to n_2^b, hence $\mathcal{M}_*(n_2^b) = \{n_2^s\}$.

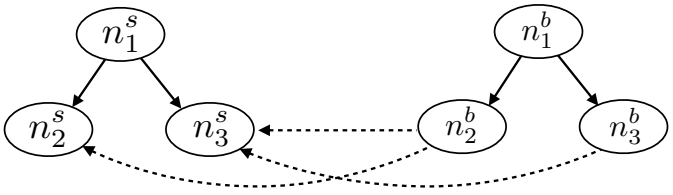


FIGURE 3.4: Handling ambiguous line reference

Step 4: No match can be found after the selection process. The reason for this case can be the following:

- *Ambiguous control dependencies:* Compiler optimization can merge multiple control edges at a single node n_i^b, leading to control dependency ambiguity. Consider the example in Figure 3.5(a). In the source code, a part of n_2^s is identical to n_4^s, therefore it is optimized into a single node n_4^b in the target binary. Consequently, n_4^b has more than one controlling nodes, namely $P_c(n_4^b) = \{n_1^b, n_3^b\}$. To handle this case, each controlling node is separately checked. This results in a *one-to-many* mapping. For example, $\mathcal{M}_*(n_4^b) = \{n_2^s, n_4^s\}$. This means that the performance modeling code of n_4^b should be annotated in both n_2^s and n_4^s.

- *Missing basic blocks in N^s:* Compiler can add a branch target to a branch basic block in N^b, therefore no counterpart for the added basic block could be found in N^s. One of such added nodes in Figure 3.5(b) is n_3^b. Code of n_3^s is pushed into both n_2^b and n_3^b. No match for n_3^b could be found. To handle this case, the source

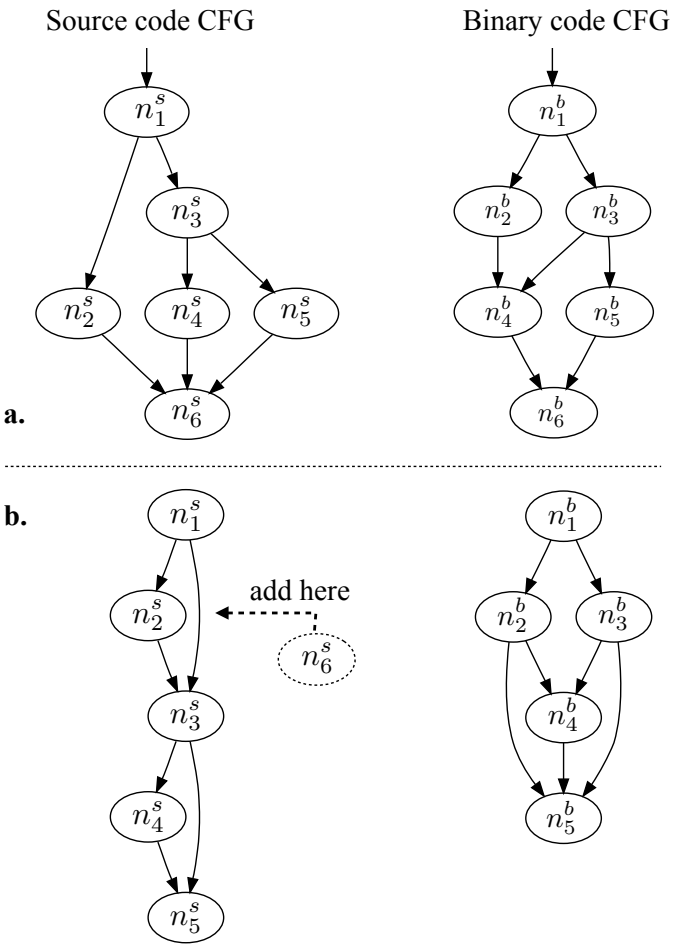


FIGURE 3.5: Resolve ambiguous control dependencies

code needs to be augmented by adding a code block accordingly. As is shown by the dashed arrow, an *else* block n_6^s is added, which is merely a placeholder for annotating the performance information of n_3^b.

Above selection steps give a general procedure in determining the basic block mapping. In practice, this procedure suffices in handling of most basic blocks in the optimized binary code. Yet, for certain specific optimizations, additional considerations besides this general procedure need to be taken into account. The considered cases are described in Section 3.3.7.

3.3.6 Comparison with Other Mapping Methods

The feature of the present approach is that it applies detailed structural analysis. It matches the extracted structural properties in constraining the basic block mapping and ensuring a correct execution order of the annotated performance modeling codes. Its advantages over existing mapping methods can be seen in the following comparison.

Comparison with the mapping based on line reference alone

Line references relate the instruction with the corresponding source code line. No structural information is contained in it. Therefore, for the basic block mapping, it is the least robust approach against the structure alteration of the CFG induced by compiler optimization. If only line reference is used, the correct execution order of the annotated performance modeling codes can not preserved. For example, n_2^b in Figure 3.3 would be wrongly mapped to n_1^s which is outside of the loop.

Comparison with the mapping based on dominance matching

The advantages over dominance preserving mapping are threefold. (1) Dominance relation is not sufficient to represent the CFG's control structure. Two different CFGs can have the same dominance graph, as shown in the upper part of Figure 3.6. (2) Dominance relation is often changed by the compiler. A simple example is given in the lower part of Figure 3.6. Due to altered dominance relation, the dominance graphs for the source code and the binary code become different. For cases as such, only investigating dominance relation is insufficient to resolve a suitable mapping between the basic blocks in the two control flow graphs. In contrast, the intra-loop control dependency graphs of these two CFGs are still similar. Although compiler optimization has changed the dominance relation among the basic blocks, it still respect the control dependency among them. Therefore, a correct mapping can be provided by inspecting the intra-loop control dependency. (3) Dominance mapping is less restrictive than control dependency mapping. It therefore yields more often ambiguous results.

Comparison with the raw control dependency based mapping

Consider the example in Figure 3.2. If the raw control dependency is used as in [60], the controlling nodes for n_3^s in the source code CFG are $\{n_2^s, n_3^s\}$, while those of n_3^b in the binary code CFG are $\{n_1^b, n_3^b, n_4^b\}$. This discrepancy impedes the mapping from n_3^b to n_3^s. This is not a problem if the intra-loop control dependency is used, which excludes the control dependency of n_3^s or n_3^b on nodes with different loop memberships.

3.3.7 Consider Other Specific Compiler Optimizations

Besides the basic block mapping methodology discussed previously, further measures are to be taken targeting specific compiler optimizations. These measures take into account specific structural alterations caused by certain optimizations, so that the source code can be annotated accordingly.

3.3.7.1 Handle Optimized Loops

To examine the influence of loop optimization to the basic block mapping, consider several common cases of loop optimization shown in Figure 3.7.
Case 1: A loop pre-header (n_1^b) is inserted before entering the loop. Loop invariant codes can be placed in the pre-header. In addition, the loop is transformed from a *for* loop into a *do-while* loop, because the compiler identifies that the loop will be entered.
Case 2: Because the compiler detects that the loop will be entered, it peels the first iteration which is executed before entering the loop for the remaining iterations. This is termed as *first-iteration splitting*.
Case 3: A *pre-test* node is inserted to check whether the loop will be entered. If so, the loop is transformed as in case 1.

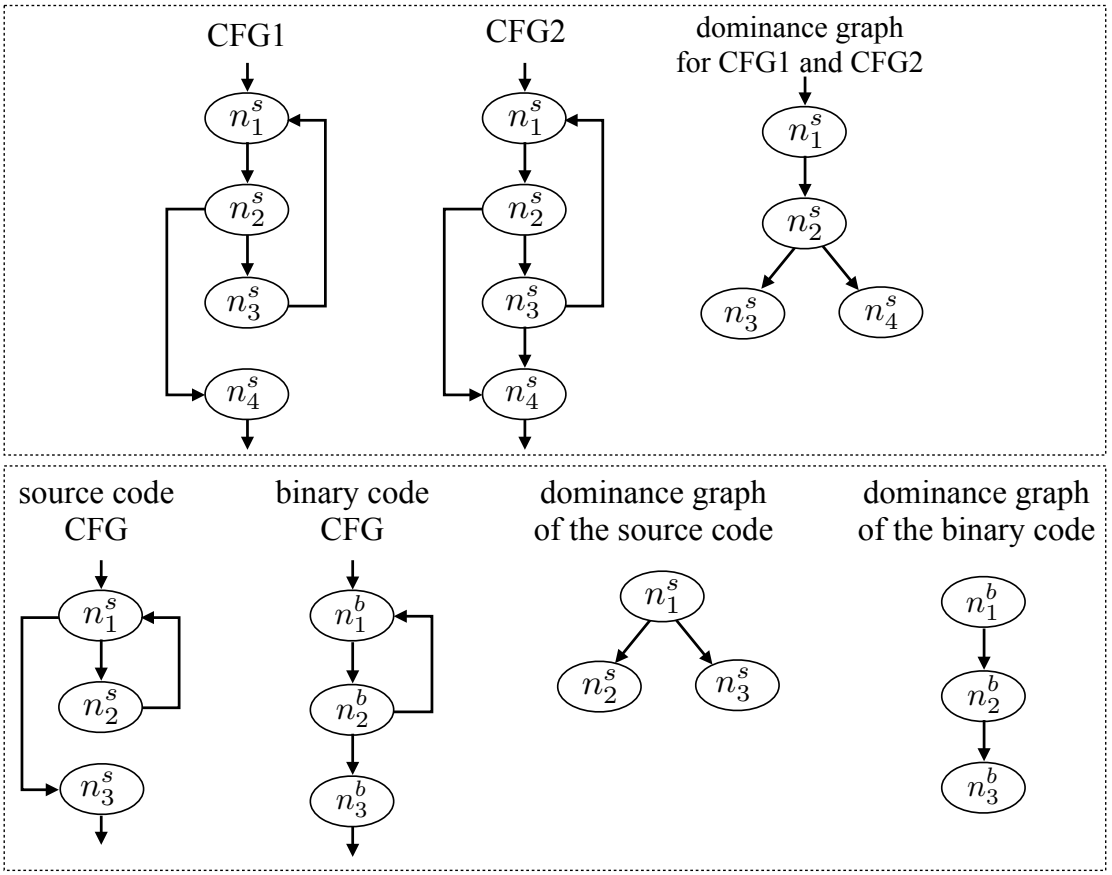


FIGURE 3.6: Checking dominance relation alone is insufficient

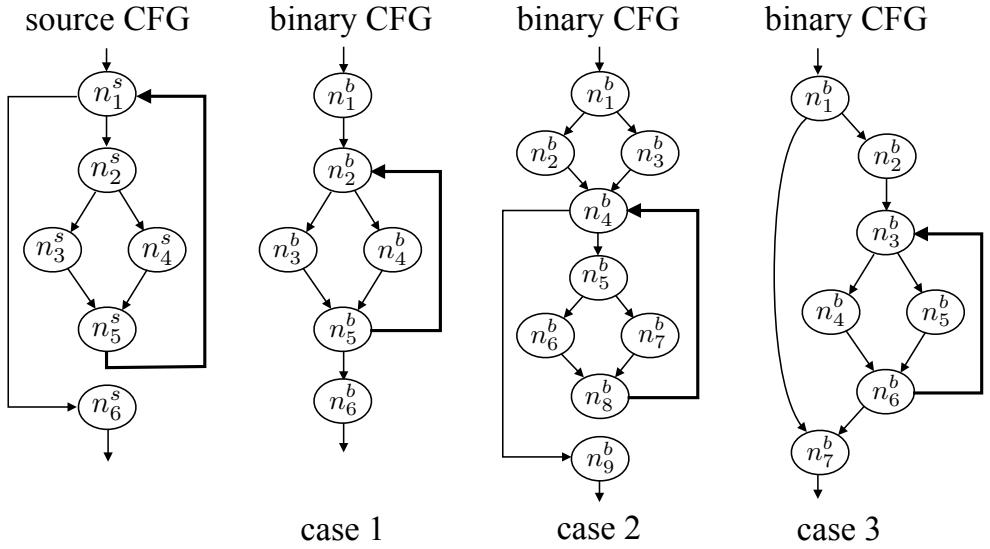


FIGURE 3.7: Common cases of loop optimizations.

These cases can be handled in the following way. The loop pre-header basic block is unproblematic to the present approach, because matching intra-loop control dependency will automatically map it to a correct basic block that is outside the loop. As for the loop pre-test basic block, a corresponding basic block can be inserted in the source code as an annotation placeholder. The *do-while* transformation is already covered by the

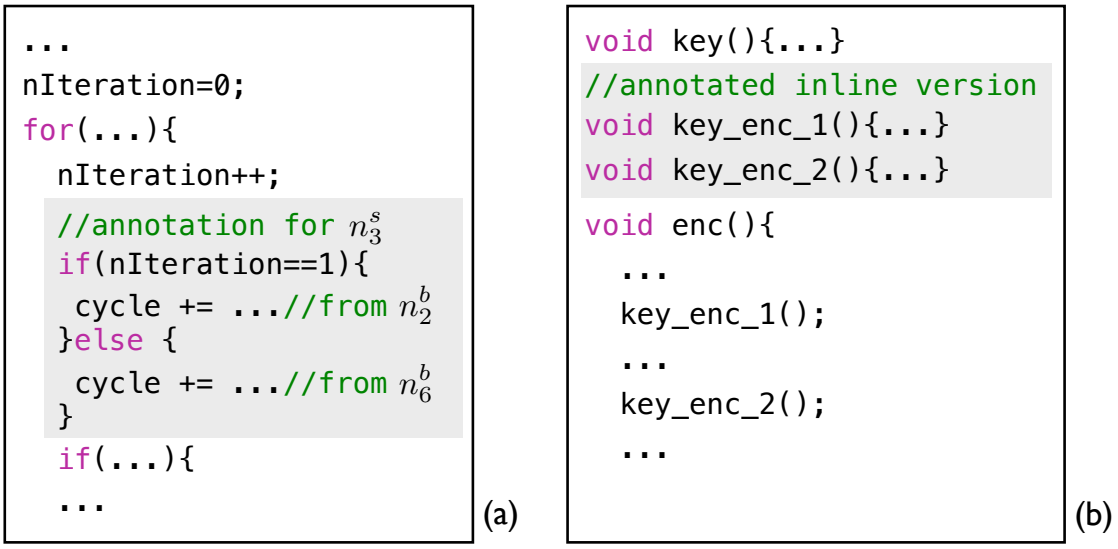


FIGURE 3.8: Annotate the source code in the case of (a) loop splitting and (b) function inlining.

selection algorithm, e.g. n_5^b in case 1 will be correctly mapped to n_1^s. Relatively more difficult to handle is the loop splitting in case 2, in which a source code basic block within a loop body is compiled to multiple counterparts. Each counterpart binary basic block is executed during certain iterations. To correctly annotate the source code, the so-called *iteration count sensitive annotation* is used. For this, an additional variable as the iteration counter is annotated in the source code. Within a source code basic block, a control statement selects among different performance modeling codes according to the iteration counter. Take n_3^s in Figure 3.7 for example, the annotation in case of first-iteration splitting (case 2) can be conducted as shown in Figure 3.8(a). For unrolled loops, two cases are differentiated. In the first case, the loop in the binary code is unrolled into a single basic block. Therefore the performance modeling code for this basic block can be annotated simply outside the corresponding loop in the source code. In the second case, there exits branch basic blocks in each unrolled iteration, indicating that a source basic block is compiled to one binary basic block within each iteration. Therefore control statements are annotated to perform iteration sensitive annotation as discussed before.

3.3.7.2 Handle Function Inlining

An inlined function is duplicated in the binary code where it is called. For annotation, inlined versions of the function are created, each of which is annotated from the corresponding part of the binary code where the function is inlined. Then, instead of calling the original function, the caller calls the annotated version accordingly such as in Figure 3.8(b). Here, `key_enc_1()` and `key_enc_2()` are the annotated versions for the 1st and 2nd calls to the inlined function `key()` in `enc()`.

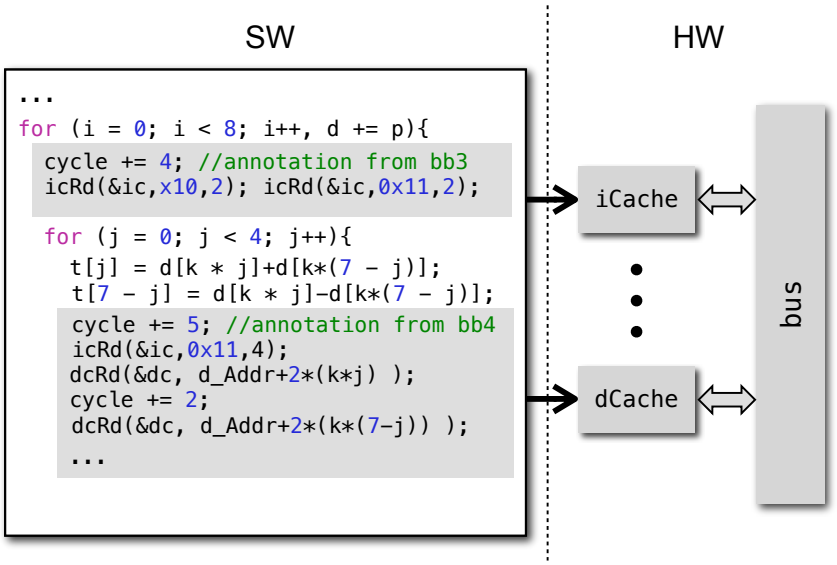


FIGURE 3.9: Example of annotated source code for HW/SW co-simulation.

3.3.7.3 Consider Compound Branches

One additional problem can be caused by complex compound branches. In the source code, one basic block containing a compound branch such as

```
if(weight==1&&(k1>1||k2<-1)),
```

may be compiled into several branch basic blocks in the binary code. All these branch basic blocks are compiled from the same line, hence it is difficult to relate the binary branch basic block to the actual comparison operation in the compound condition. On the other hand, it would also be fairly intrusive to annotate each comparison in the compound condition. To handle this case, the branch basic blocks in the binary code are considered as a single basic block, from which the worst-case performance is extracted to annotate the corresponding compound branch in the source code.

3.4 Reconstruction of Data Memory Accesses

To enable HW/SW co-simulation, memory accesses also need to be extracted from the target binary and annotated into the source code for cache simulation. One example of such annotated source code is shown in Figure 3.9. The variables `ic` and `dc` denote the instruction and data cache respectively. When cache misses occur, transactions over the bus can be invoked for data communication modeling. It is straightforward to extract the addresses for instruction memory accesses, since they are statically known. However, the problem of resolving the addresses of data memory accesses persists, even though the control-flow of the target binary can be accurately modeled in the source code. A solution of this problem has been pending for a long time. This work tackles this problem by exploiting the underlying memory allocation mechanism. Since compiler also respects this mechanism, the present approach provides a robust and precise way to reconstruct the memory accesses, which contributes to the high accuracy of data cache simulation.

Three major sections are considered in allocating memory to a program: (1) the section stack for local variables, (2) the data section for global/static variables, and (3) the heap section for dynamically allocated data. Furthermore, pointer dereference and pointer arithmetic can also be handled, This is a problem considered hard for host-compiled SW simulation [55]. In the tool-chain, an enhanced *cparser* [134] is used to parse the source code and extract syntax information, such as function signatures and local variables. Also, binary utilities are used to obtain debug information, such as the symbol table containing the addresses of static variables. Details of how each case is handled are given in the following.

3.4.1 Addresses of the Variables in the Stack

Stack memory stores primarily local variables, spilled registers and function arguments. Register spilling, due to the limited number of registers, often happens when entering or exiting a function. The addresses of the stack data are referenced by the stack pointer (*sp*). The value of *sp* is decreased or increased when a function is entered or exited. So the stack address of a function is dynamic, depending on when the function is called. To trace such dynamics, a variable *sp* is annotated in the source code. Its value is updated at the entrance and exit of a function based on the local stack size of the function. As shown in the annotated source code in Figure 3.10, the local stack size of the function is 80 as indicated in the target binary code. Thus the variable *sp* is added and subtracted by 80 respectively at the function entrance and exit. To determine the addresses for stack data, two cases are considered:

1. The data are explicitly addressed by the `sp` register in the binary instructions, as in Figure 3.10.
2. The data are not explicitly addressed by the `sp` register, as in the instruction "`sw v0 v1`" in Figure 3.11.

For case 1, the value of the annotated variable *sp* can be directly used to calculate the accessed addresses (see Figure 3.10). The initial value `0x1000` of *sp* is defined in the boot-loader by the designer.
For case 2, the syntax of the source code needs to be examined. For example in Figure 3.11, the instruction "`sw v0 v1`" in the binary code is compiled from the source code to store a variable to $b[i]$. Although the accessed address in the instruction is *sp* implicit, $b[i]$ is a local variable and thus locates in the local stack of the function. Thus, the accessed address is calculated as $sp+40+4*i$, where $sp+40$, provided by the *cparser*, is the base address of the array $b[8]$ and $4*i$ is the offset address of $b[i]$, assuming the size of an integer for the target machine is 4 bytes.

3.4.2 Addresses of Static and Global Variables

The data section stores static and global variables. Their addresses are stored in a *symbol table* in the ELF file, which is provided by the debugger. The symbol table is used to resolve the addresses related to static and global variables. In the example of Figure 3.12, the instruction at $0x500$ loads the variable $arr[i]$. Since *arr* is a global variable, the *symbol table* is queried to obtain the base address $0x3010$ of *arr*. Then, with the help of a parser of the source code, the actual address is calculated as $0x3010+4*i$.

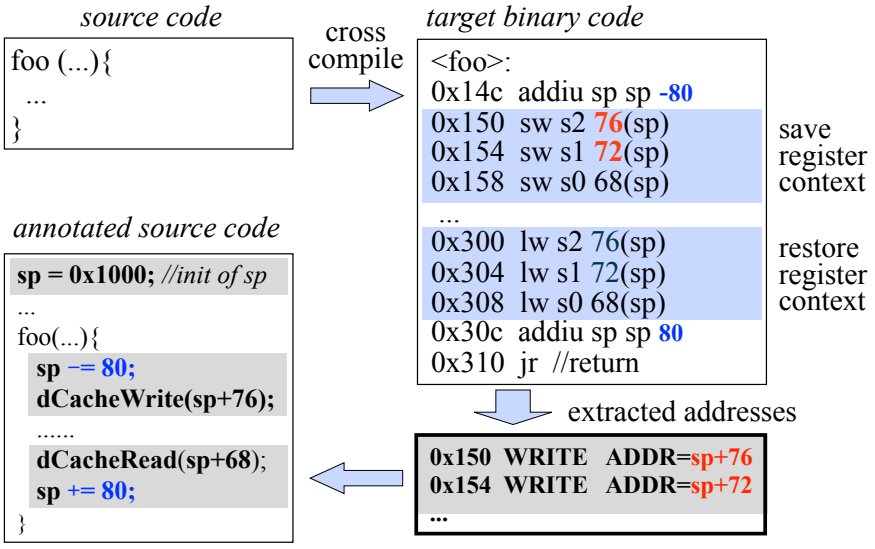


FIGURE 3.10: Stack data with *sp*-explicit addresses.

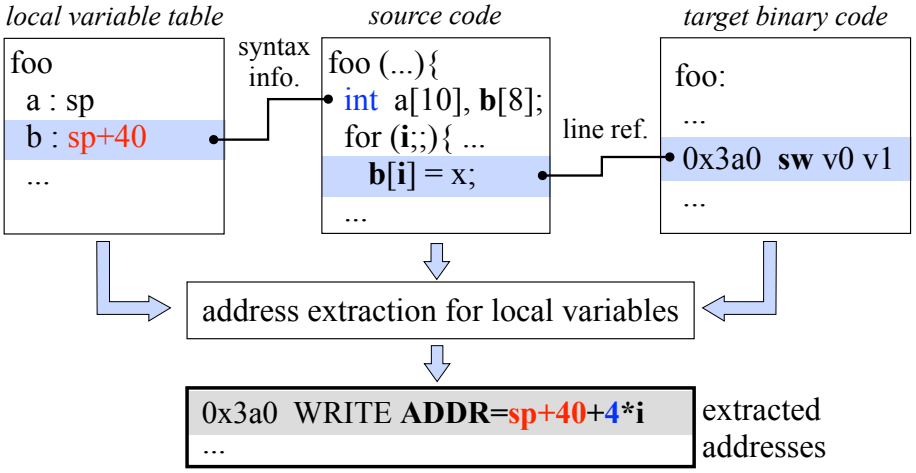


FIGURE 3.11: Stack data with *sp*-implicit addresses.

3.4.3 Addresses of the Variables in the Heap

The heap section stores the dynamically allocated variables. This involves the OS support such as *malloc* and *free*, which have not been considered previously [53, 58, 60, 61]. To handle this case, the growth of the heap in the target memory space is simulated, by the same algorithms of heap initialization and allocation used in the target operating system. As a result, the addresses returned by *malloc* can be directly used as the addresses of dynamically allocated heap variables in the target binary. For example, consider the following code snippet:

```
1 //extract addresses of the variables in the heap
2 1  ptr = malloc(100);
3 2  ptr_addr = OSMalloc(100); //annotated
4 3  ptr[i] = weight;
5 4  dCacheWrite(ptr_addr+i);  //annotated
```

The `malloc` in line 1 will call the normal memory allocation algorithm of the host machine. The `OSMalloc` in line 2 will call the same heap allocation algorithm used by

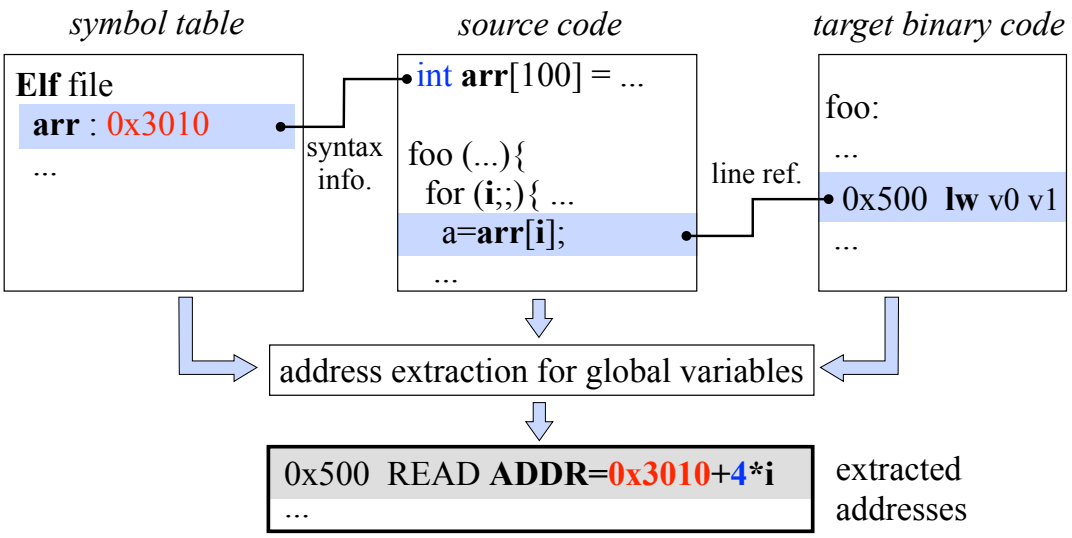


FIGURE 3.12: Extracted addresses for memory accesses corresponding to static or global variables.

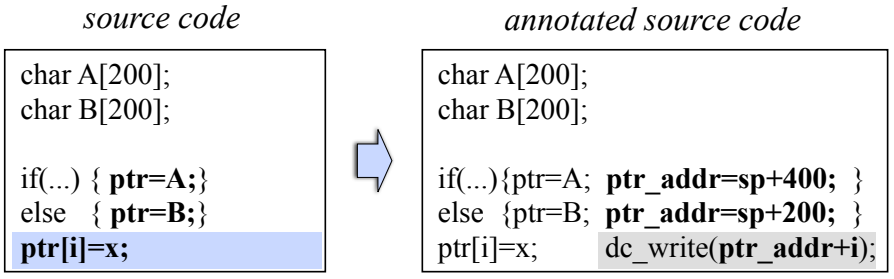
the OS of the target machine. The resultant variable `ptr_addr` is per se the address of the allocated memory in the target memory space. It is then used to calculate the address of the considered heap variable. The calculated address is then used for data cache simulation as shown in line 4.

3.4.4 Handling Pointers

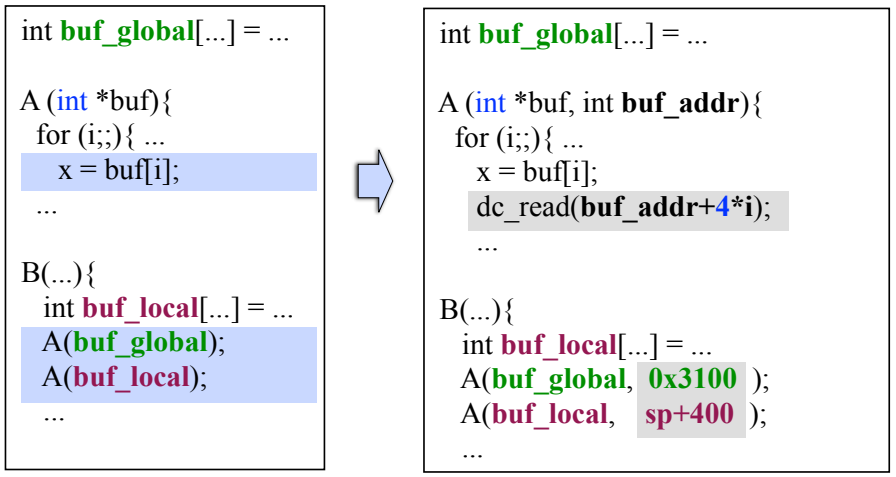
It was believed in [55] that addresses related to pointer dereference are very difficult to be determined. Here a method to extract the addresses related to pointer dereferences is shown. In this method, a variable is annotated to simulate the pointed address, as exemplified by the variable `ptr_addr` in Figure 3.13(a). At pointer assignment, the simulated pointer address is updated. At pointer dereference, the address can thus be annotated by using the simulated address.

The addresses become statically indefinite when they are passed as function arguments or pointer arithmetic is used. One example of the first issue is given in Figure 3.13(b). Function `A(int *buf)` takes a pointer as its input argument, which is then used to address the variable. However, `A(int *buf)` can be called by different callers with different pointer values. This means the address pointed by `buf` can be resolved only by the callers during the execution of the program. To handle this, the function signature is augmented with additional arguments. These additional arguments are placeholders for the actual addresses, which are to be provided by the caller according to each function call. As exemplified in the right part of Figure 3.13(b), `B()` calls `A(int *buf)` twice by passing pointers to a local and a global variable respectively. In the annotated source code, `B()` also passes corresponding addresses in each call. With them, the actual memory accesses can be subsequently constructed in `A()`.

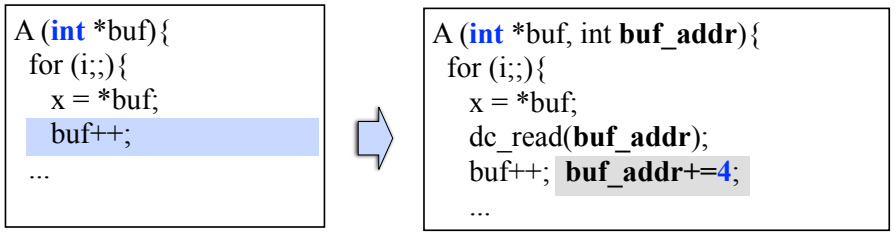
When pointer arithmetic is used, the pointed address is changed. To trace the pointed address, the value of the pointer has to be changed accordingly. In the given example of Figure 3.13(c), `buf_addr` holds the value of `buf` in the target binary. Corresponding



(a) Update the pointer value



(b) Pass the pointer value as a function argument



(c) Simulate the pointer arithmetic

FIGURE 3.13: Handle different cases of pointer dereference.

to `buf++`, the annotated `buf_addr+=4` simulates the new address after the pointer increment, therefore `buf_addr` represents the actual accessed address which can be used in the data cache simulation.

3.5 Experimental Results

The experimental results will be presented in three parts. First, from an implementation perspective, Section 3.5.1 describes the developed tool for automatic source code annotation. It will detail the usage of this tool from the required input files, to the performed analysis and the corresponding class diagram. Examples of generated reports will also be given. Second, from a verification perspective, Section 3.5.2 uses the tool to annotate

benchmark programs. ISS-based simulation and the host-compiled simulation are carried out. The estimated performances in different simulations are compared to evaluate the present approach. Finally, from an application perspective, Section 3.5.3 presents a complex virtual platform for an autonomous two-wheeled robot. Control algorithms run on the hardware system to balance the robot and to follow a red line. Host-compiled simulation is applied so that the simulation can run in real-time, while the performance of the control algorithms can be estimated with suitable accuracy.

3.5.1 The Tool Chain and the Generated Files

The main steps when using the tool chain are depicted in Figure 3.14. Firstly, several input files are made available for further analysis. Secondly, structural analysis is performed for the CFGs of both the source code and the target binary. Thirdly, basic block mapping takes place. Finally, the source code is annotated based on previous results. These steps are detailed in the following.

3.5.1.1 Input Files

The tool chain requires the following files to be available:

- The original source code files to be annotated. In the examined case, the source codes are written in C.
- The assembly code of the target binary code, which is obtained by cross-compiling the source code.
- The line reference file provided by the binary utilities of the cross-compiler.

3.5.1.2 Performed Analysis

As shown in Figure 3.14, following analyses are performed:

- The source code file is provided to a customized C code parser based on the pycparser [134]. This parser primarily performs two tasks:
  1. It constructs the control-flow graphs of the parsed source code functions.
  2. It extracts syntax information such as the function signature and the local variables.
- The target assembly code is also parsed by a parser which primarily performs two tasks:
  1. It constructs the control-flow graphs of the parsed binary code functions. To do this, it parses the target instructions for the branch instructions and subsequently identifies the entry and exit instructions for all the basic blocks.
  2. It performs static analysis for each basic block of the target binary code. This provides the estimated execution cycles of each binary basic block.

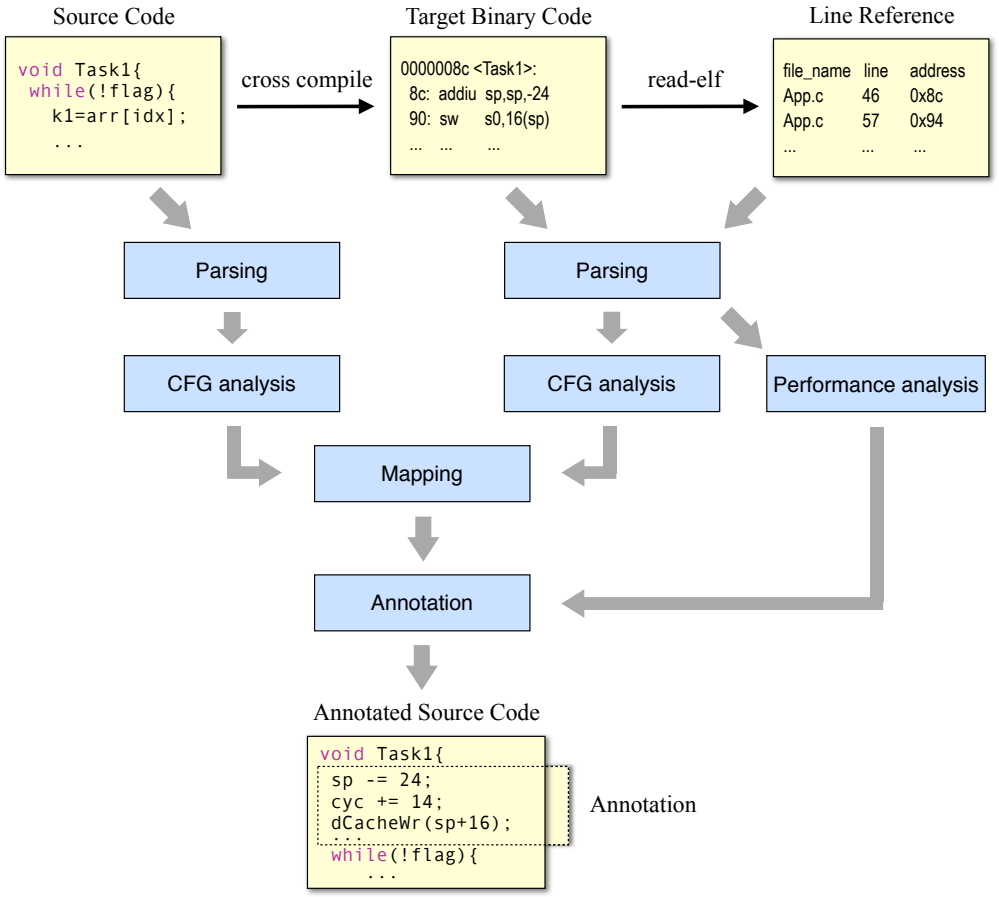


FIGURE 3.14: Main structure of the tool chain

- Given a control-flow graph (CFG) of the source code or the target binary code, the CFG analyzer performs structural analysis of the CFGs. It then and assigns corresponding structural properties to each node of a CFG. Performed analyses include:

  (a) Dominance analysis.

  (b) Post-dominance analysis.

  (c) Loop analysis.

  (d) Control dependency and intra-loop control dependency analysis.

  (e) Extracting the structural properties for each basic block based on the above structural analysis.

- With the extracted structural properties, the mapper is used to complete the basic block mapping by matching those properties as described in Section 3.2. Finally, based on the basic block mapping and the static performance analysis, the annotator generates the annotated source code.

For introducing the implementation details, a diagram of several main classes used in the tool-chain is illustrated in Figure 3.15. In this diagram, the B-parser instantiates a set of functions that are parsed from the target binary code. Each function instance contains a set of basic blocks which further consists of a set of instructions. A function instance also owns a control flow graph instance whose nodes and edges represent the

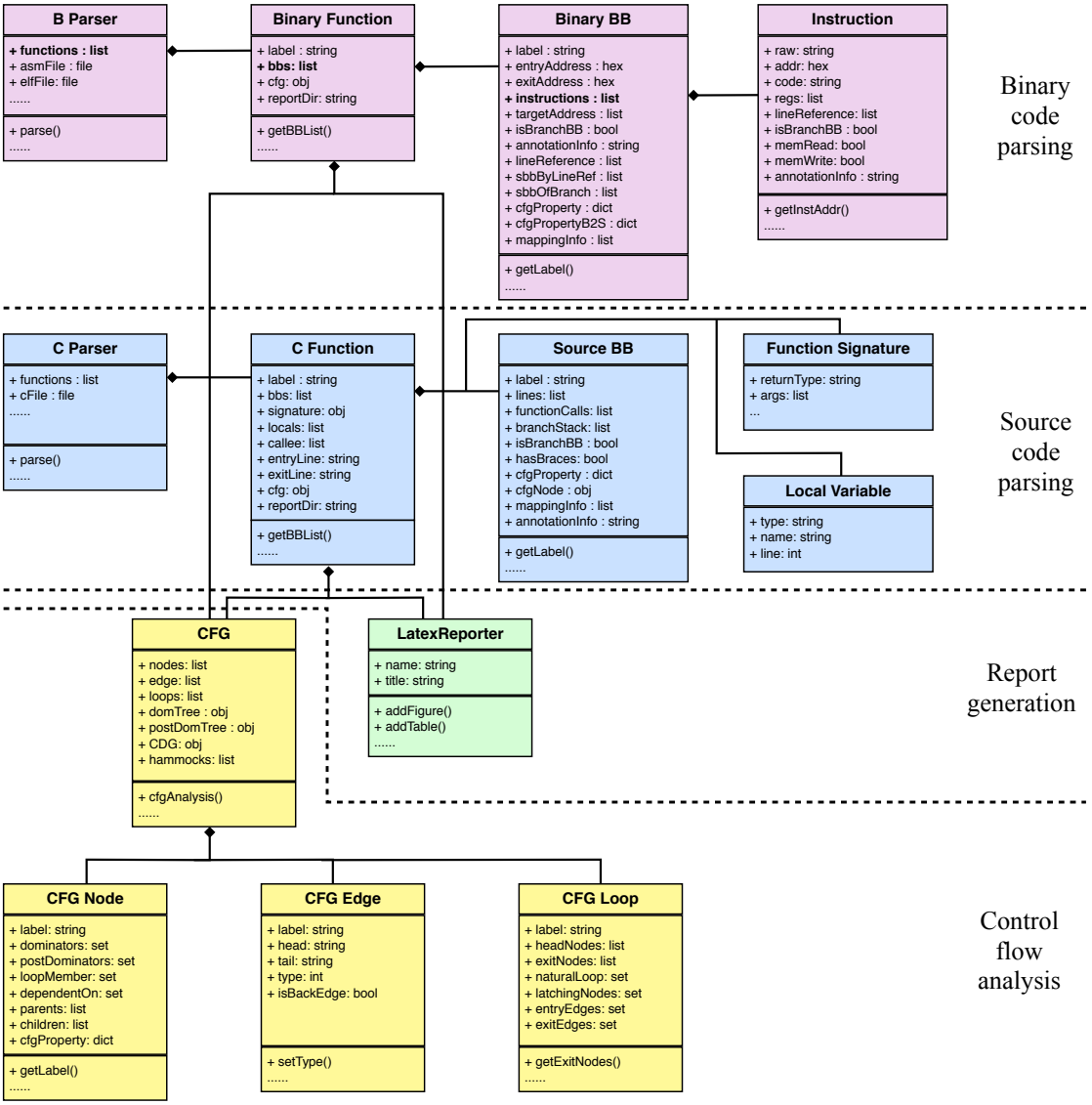


FIGURE 3.15: A diagram showing the main classes for the parsing and CFG analysis.

basic blocks of this function and their connections. The CFG class implements the structural analyses previously described. The report class in the diagram can be used by a function instance to convert the analysis results into figures or tables in a latex file that is further compiled a *pdf* format report. Similar steps also hold for the source code parsing and analysis. Examples of the generated reports are given in the following Section.

3.5.1.3 Automatically Generated Reports

The generated reports are exemplified here with the analysis of a function called *edge*. In the following reports, basic block i in the source code is denoted by sbi. A loop in the source code is denoted by loop_sbi where sbi is the head node of this loop. Similarly, basic block j in the binary code is denoted by bbj.

From the source code, the parser extracts syntactic information such as the local variables and the branch conditions. Corresponding reports are given in Table 3.3 and Table 3.4 respectively. The local variables are used in determining the memory access addresses for data located in the local stack of the function as described in Section 3.4. The compound branch conditions are used in handling the complex branches in the target binary code. Several branch basic block in the CFG of the target binary code can be regarded as a single basic block if they are all compiled from a compound branch in the source code.

After parsing, the control-flow graph of this function can be generated, as shown in Figure 3.16. Structural analysis of this control-flow graph is performed, resulting in several reports including a dominance graph, a post-dominance graph, the loop analysis, the intra-loop control dependency graph, and the structural properties of the basic blocks. These reports are given in Figure 3.17, Figure 3.18, Table 3.5, Figure 3.19, and Table 3.7, respectively. In the property Table 3.7, the columns P_m, P_b, and P_c correspond to the examined structural properties defined in Section 3.2. The column P_{sc} gives the controlling nodes obtained according to the standard control dependency. Notice that the loop analysis can merge two loops in case they have the same head node. This is the case for the merged loop with the head node $sb8$. The natural loop of the merged loop is the union of that of the two loops.

The CFG of the target binary code is shown in Figure 3.20. As has been shown for the source code CFG, analysis of the control-flows is performed. The obtained structural properties of the basic blocks are listed in Table 3.8. The translated properties are shown in Table 3.9. They are derived according to the procedure in Section 3.3.2 and

TABLE 3.3: Parsed local variables

variable type	variable name	line number
int	i	73
int	j	73
int	n1	73
int	n2	73
int	iWid	74
int	iLen	74
int	k1	75
int	k2	75
char	ch	76
char unsigned **	w	77
char unsigned *	temp	78
char unsigned	tmp	78
int	y1	79
int	y2	79
int unsigned	zn2	80
int [3]	buf	90
int	numCharIn	97
int	numCharOut	98

TABLE 3.4: Control node list in the function

node	type	condition	compound	line
sb2	for	n1 < iLen	No	99
sb3	for	j < iWid	No	102
sb5	for	n2 < iWid	No	111
sb7	for	k1 <= 1	No	116
sb8	for	k2 <= 1	No	119
sb9	if	((n2 + k2) < 0) || (((n2 + k2) - 1) > iWid)	Yes	120
sb12	if	y1 < 0	No	129
sb14	if	y2 < 0	No	132
sb16	if	y1 > y2	No	135

TABLE 3.5: Loop analysis results

loop	back edges	natural loop	entry edges	exit edges
loop_sb3	(sb4,sb3)	{sb3,sb4}	(sb2,sb3)	(sb3,sb5)
loop_sb7	(sb8,sb7)	{sb8,sb9,sb10,sb7,sb11}	(sb6,sb7)	(sb7,sb12)
loop_sb8	(sb10,sb8)(sb11,sb8)	{sb10,sb11,sb8,sb9}	(sb7,sb8)	(sb8,sb7)
loop_sb5	(sb19,sb5)	{sb6,sb7,sb5,sb8,sb9,sb14, sb15,sb16,sb17,sb10,sb11, sb12,sb13,sb18,sb19}	(sb3,sb5)	(sb5,sb20)
loop_sb2	(sb20,sb2)	{sb2,sb3,sb6,sb7,sb4,sb5, sb8,sb9,sb20,sb14,sb15, sb16,sb17,sb10,sb11 sb12,sb13,sb18,sb19}	(sb1,sb2)	(sb2,sb21)

Section 3.3.3. As can be seen, these properties are represented using the source code basic blocks or loops, thus they can be compared against those in Table 3.7. Now, the procedure in Section 3.3 can be carried out. It checks the properties for the considered basic blocks in Table 3.7 and Table 3.9, and determines the basic block mapping. Results of the successively selected matching sets in this mapping procedure are given in Table 3.10. They correspond to the selection process described in Section 3.3.5.

Several remarks can be made for the above analysis and results. Firstly, the standard control dependency is very different from the intra-loop control dependency introduced in this work. This difference can be seen in Table 3.7 and Table 3.8, where the last two columns in each table differ from each other. The reason for this difference is that, the standard control dependency is derived from the overall CFG and takes into account the control dependency of a basic block on another one which can have a different loop membership. For an optimized binary code, this CFG-wide control dependency often does not match to that in the source code, as shown in this example. Therefore, if the mapping procedure compares the standard control dependency, the mapping of many binary basic blocks will not be found. In contrast, the intra-loop control dependency is

well preserved across the source code and the optimized binary code, thus contributing to a successful basic block mapping.

Secondly, after applying all the selection steps, there can still be multiple candidate source code blocks. For example, the basic block bb4 has the same properties with sb3 and sb4. It also has line references from both of them. In this case, bb4 will be mapped to only one of them, but not both. The final mapping for bb4 is sb3, since both of them are the exit nodes of the corresponding loops. In other cases, if the multiple candidates are sequential to each other, only one of them will be selected. Two basic blocks are sequential to each other, if any path from the entry to the exit of the code either visits both or none of them. This condition can be identified, if the first one dominates the second one and the second one post-dominates the first one. From the perspective of performance estimation, putting the annotated codes in either of the two sequential basic blocks would provide same result.

Thirdly, although a mapping is usually found after matching the properties, there may be exceptions. One reason is due to erroneous line references. As can be seen in Figure 3.21, the line reference for bb12, bb13, and bb19 is sb7, which is wrong. Therefore, matching the intra-loop control dependency of bb19 and bb13 would map them to sb7, as shown in the first column of Table 3.10. But this mapping is revoked after further checking the loop membership as seen in the second column of Table 3.10, since bb19 and sb7 have different loop memberships. In the end, according to step 4 in Section 3.3.5, they are mapped to sb13 and sb13a respectively, where sb13a is an added *else* block as a place-holder for annotation. The wrong line references for bb13 and bb19 also lead to a mapping from bb14 to sb12, instead of to the correct one sb16. This would not affect the accumulated performance estimation, since sb12 and sb16 are sequential to each other. It has been observed in practice that the line reference of one instruction can incorrectly overwrite that of another instruction, as a side effect after certain optimization technique has been applied. This problem of wrong line reference is believed to be a bug of the compiler or the debugging utilities. Some researchers have considered this issue by correcting those incorrect line references [135]. Another seemingly more effective way to fix this bug is to directly modify the tool-chain of the cross compiler.

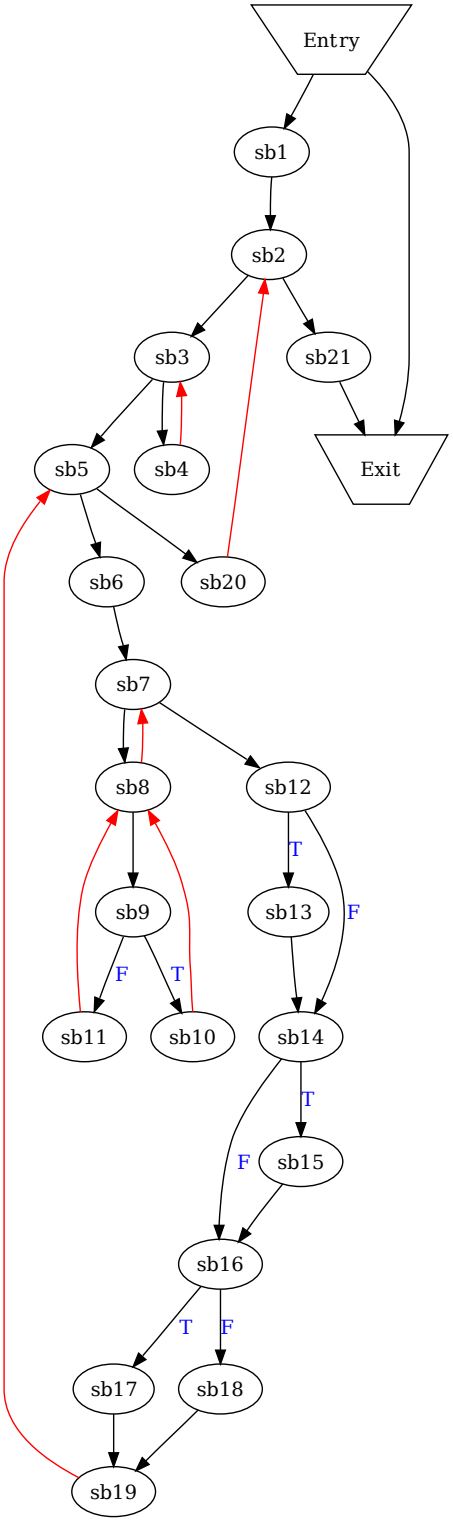


FIGURE 3.16: Control-flow graph of the source code

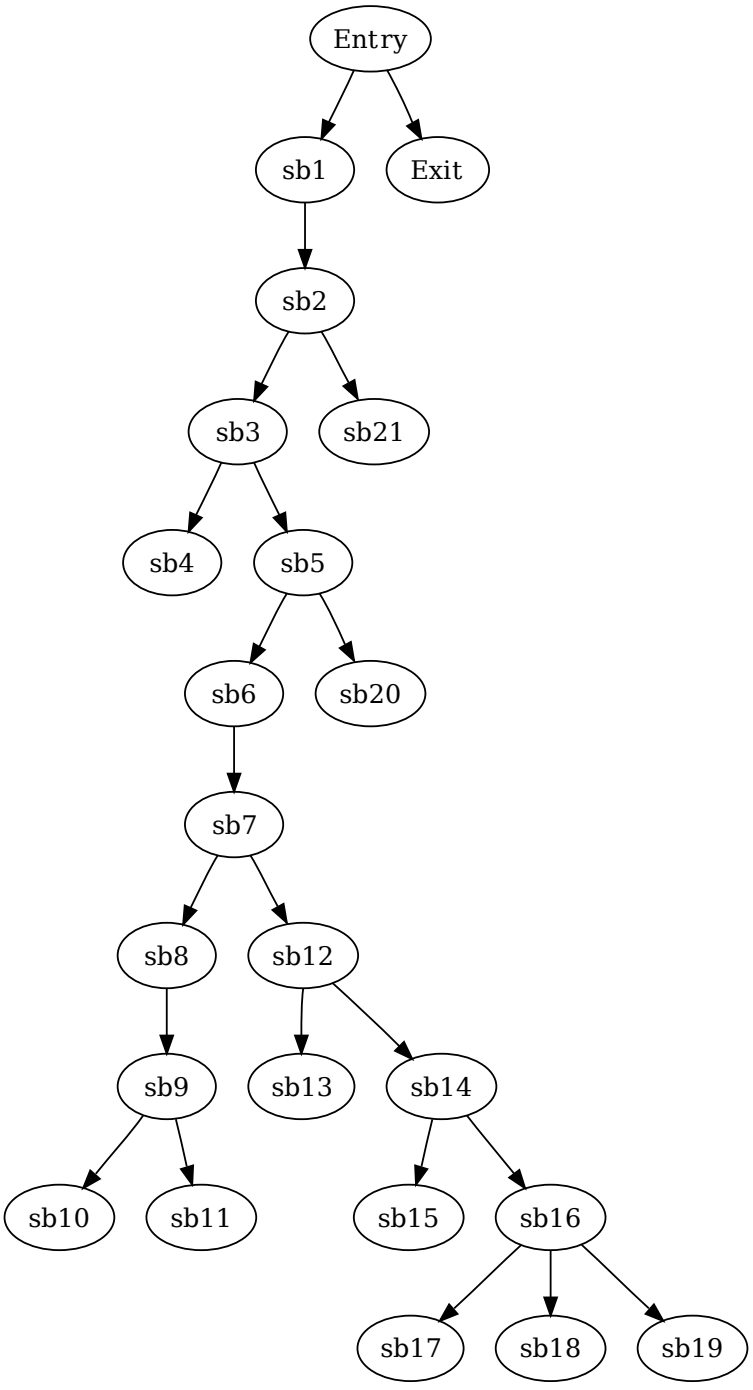


FIGURE 3.17: The dominance graph

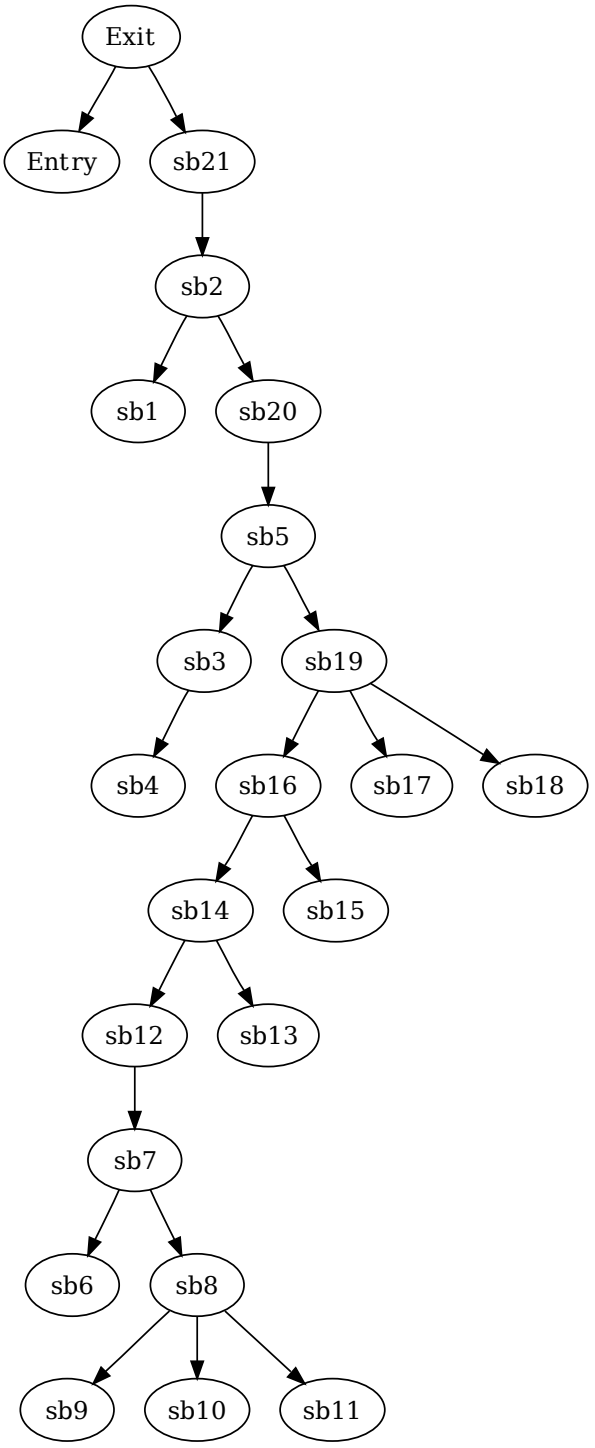


FIGURE 3.18: The post-dominance graph

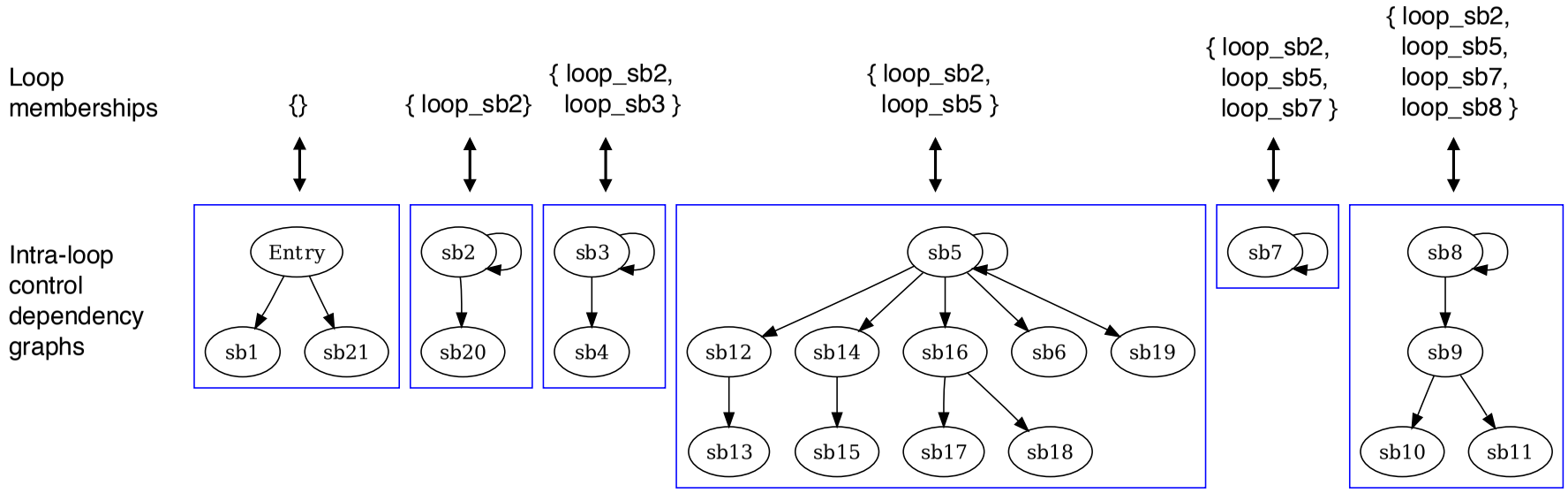


FIGURE 3.19: A set of generated intra-loop control dependency graphs.

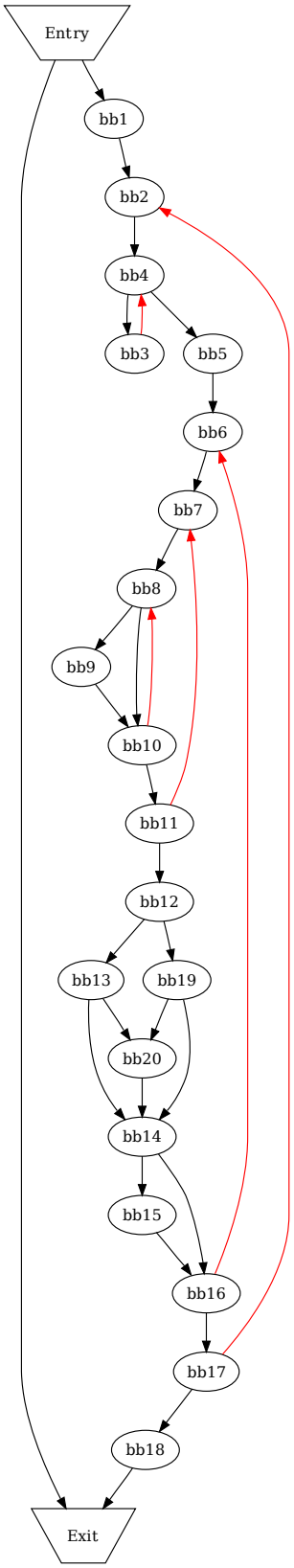


FIGURE 3.20: Control-flow graph of the target binary code

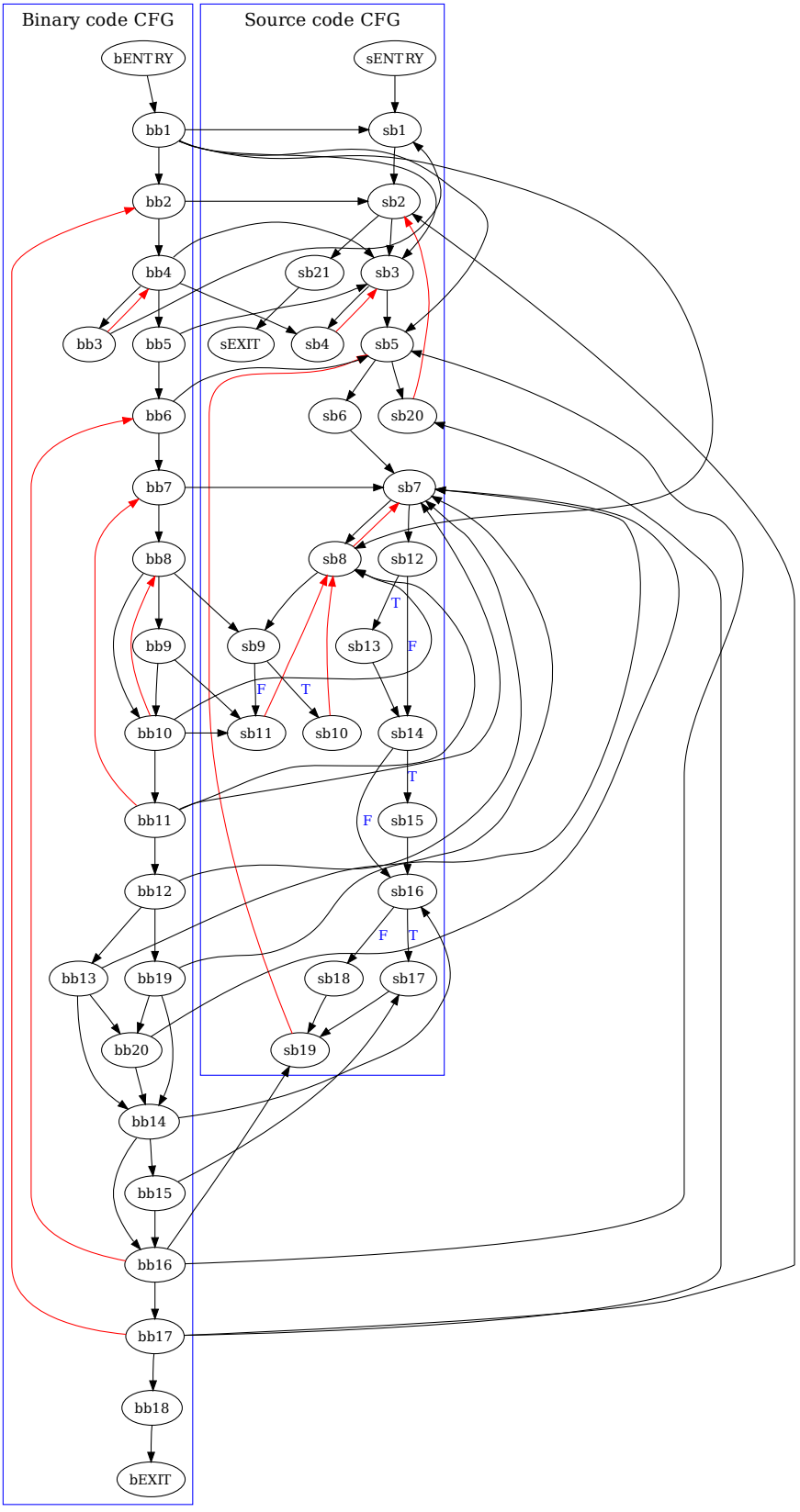


FIGURE 3.21: Line references

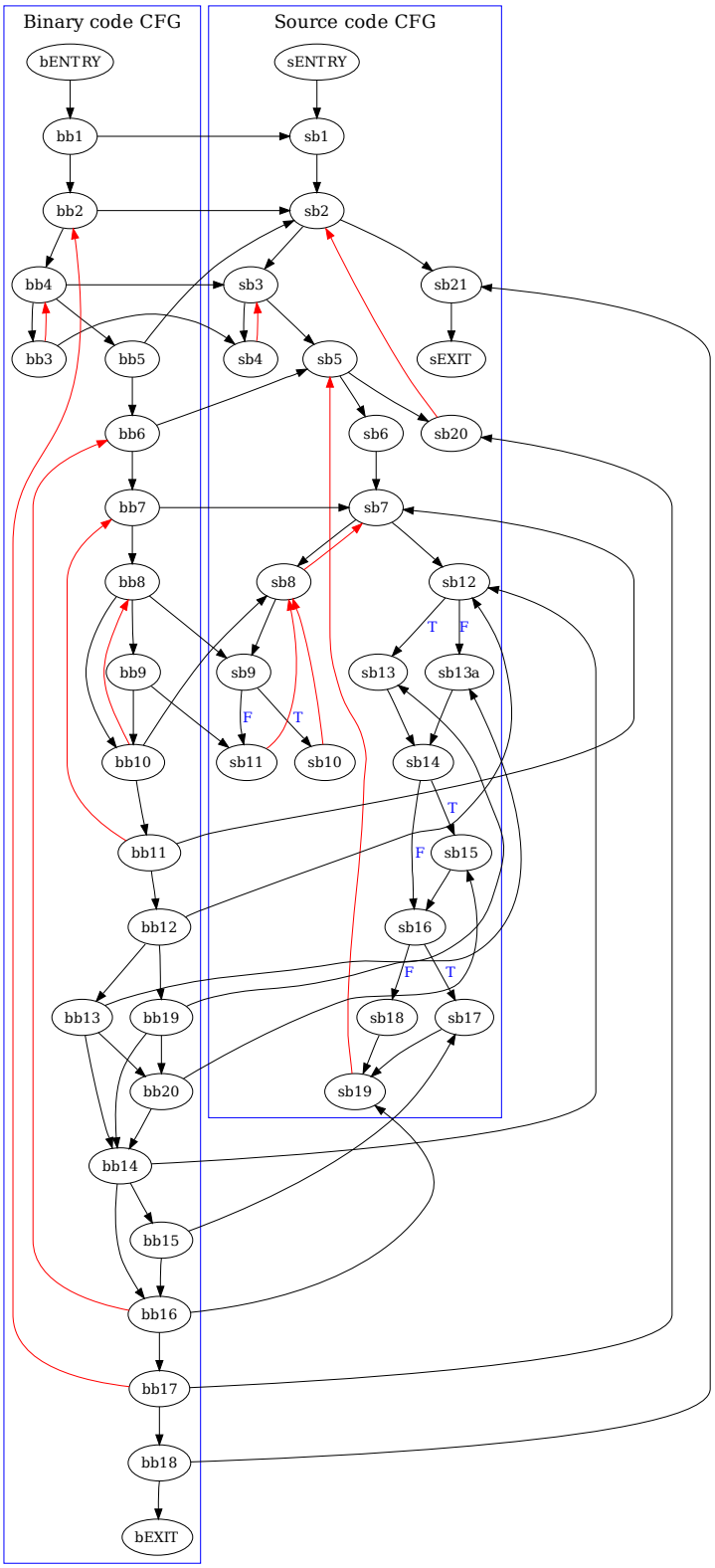


FIGURE 3.22: Resulted basic block mapping graph

TABLE 3.6: Structural properties of the basic blocks

node label	P_m	P_c	P_b
Entry	{}	{}	Entry
sb1	{}	{Entry}	Entry
sb2	{loop_sb2}	{sb2,Entry}	Entry
sb3	{loop_sb2,loop_sb3}	{sb2,sb3}	sb2
sb4	{loop_sb2,loop_sb3}	{sb3}	sb3
sb5	{loop_sb5,loop_sb2}	{sb2,sb5}	sb3
sb6	{loop_sb5,loop_sb2}	{sb5}	sb5
sb7	{loop_sb7,loop_sb5,loop_sb2}	{sb7,sb5}	sb5
sb8	{loop_sb7,loop_sb5,loop_sb2,loop_sb8}	{sb8,sb7}	sb7
sb9	{loop_sb7,loop_sb5,loop_sb2,loop_sb8}	{sb8}	sb8
sb10	{loop_sb7,loop_sb5,loop_sb2,loop_sb8}	{sb9}	sb9
sb11	{loop_sb7,loop_sb5,loop_sb2,loop_sb8}	{sb9}	sb9
sb12	{loop_sb5,loop_sb2}	{sb5}	sb7
sb13	{loop_sb5,loop_sb2}	{sb12}	sb12
sb14	{loop_sb5,loop_sb2}	{sb5}	sb12
sb15	{loop_sb5,loop_sb2}	{sb14}	sb14
sb16	{loop_sb5,loop_sb2}	{sb5}	sb14
sb17	{loop_sb5,loop_sb2}	{sb16}	sb16
sb18	{loop_sb5,loop_sb2}	{sb16}	sb16
sb19	{loop_sb5,loop_sb2}	{sb5}	sb16
sb20	{loop_sb2}	{sb2}	sb5
sb21	{}	{Entry}	sb2
Exit	{}	{}	Entry

TABLE 3.7: Source node properties

label	P_m	P_b	P_{sc}	P_c
sb1	{}	Entry	{Entry}	{Entry}
sb2	{loop_sb2}	Entry	{sb2, Entry}	{sb2}
sb3	{loop_sb2, loop_sb3}	sb2	{sb2, sb3}	{sb3}
sb4	{loop_sb2, loop_sb3}	sb3	{sb3}	{sb3}
sb5	{loop_sb5, loop_sb2}	sb3	{sb2, sb5}	{sb5}
sb6	{loop_sb5, loop_sb2}	sb5	{sb5}	{sb5}
sb7	{loop_sb7, loop_sb5, loop_sb2}	sb5	{sb7, sb5}	{sb7}
sb8	{loop_sb7, loop_sb5, loop_sb2, loop_sb8}	sb7	{sb8, sb7}	{sb8}
sb9	{loop_sb7, loop_sb5, loop_sb2, loop_sb8}	sb8	{sb8}	{sb8}
sb10	{loop_sb7, loop_sb5, loop_sb2, loop_sb8}	sb9	{sb9}	{sb9}
sb11	{loop_sb7, loop_sb5, loop_sb2, loop_sb8}	sb9	{sb9}	{sb9}
sb12	{loop_sb5, loop_sb2}	sb7	{sb5}	{sb5}
sb13	{loop_sb5, loop_sb2}	sb12	{sb12}	{sb12}
sb14	{loop_sb5, loop_sb2}	sb12	{sb5}	{sb5}
sb15	{loop_sb5, loop_sb2}	sb14	{sb14}	{sb14}
sb16	{loop_sb5, loop_sb2}	sb14	{sb5}	{sb5}
sb17	{loop_sb5, loop_sb2}	sb16	{sb16}	{sb16}
sb18	{loop_sb5, loop_sb2}	sb16	{sb16}	{sb16}
sb19	{loop_sb5, loop_sb2}	sb16	{sb5}	{sb5}
sb20	{loop_sb2}	sb5	{sb2}	{sb2}
sb21	{}	sb2	{Entry}	{Entry}

TABLE 3.8: Binary node properties

label	P_m	P_b	P_{sc}	P_c
bb1	{}	Entry	{Entry}	{Entry}
bb2	{loop_bb2}	Entry	{Entry, bb17}	{bb17}
bb3	{loop_bb2, loop_bb4}	bb4	{bb4}	{bb4}
bb4	{loop_bb2, loop_bb4}	Entry	{Entry, bb4, bb17}	{bb4}
bb5	{loop_bb2}	bb4	{Entry, bb17}	{bb17}
bb6	{loop_bb2, loop_bb6}	bb4	{Entry, bb17, bb16}	{bb16}
bb7	{loop_bb2, loop_bb7, loop_bb6}	bb4	{Entry, bb11, bb17, bb16}	{bb11}
bb8	{loop_bb2, loop_bb8, loop_bb7, loop_bb6}	bb4	{Entry, bb11, bb10, bb17, bb16}	{bb10}
bb9	{loop_bb2, loop_bb8, loop_bb7, loop_bb6}	bb8	{bb8}	{bb8}
bb10	{loop_bb2, loop_bb8, loop_bb7, loop_bb6}	bb8	{Entry, bb11, bb10, bb17, bb16}	{bb10}
bb11	{loop_bb2, loop_bb7, loop_bb6}	bb10	{Entry, bb11, bb17, bb16}	{bb11}
bb12	{loop_bb2, loop_bb6}	bb11	{Entry, bb17, bb16}	{bb16}
bb13	{loop_bb2, loop_bb6}	bb12	{bb12}	{bb12}
bb14	{loop_bb2, loop_bb6}	bb12	{Entry, bb17, bb16}	{bb16}
bb15	{loop_bb2, loop_bb6}	bb14	{bb14}	{bb14}
bb16	{loop_bb2, loop_bb6}	bb14	{Entry, bb17, bb16}	{bb16}
bb17	{loop_bb2}	bb16	{Entry, bb17}	{bb17}
bb18	{}	bb17	{Entry}	{Entry}
bb19	{loop_bb2, loop_bb6}	bb12	{bb12}	{bb12}
bb20	{loop_bb2, loop_bb6}	bb12	{bb13, bb19}	{bb13, bb19}

TABLE 3.9: Binary node properties translated according to Section 3.3.2 and 3.3.3

label	P_m	P_b	P_{sc}	P_c
bb1	{}	Entry	{Entry}	{Entry}
bb2	{loop_sb2}	Entry	{Entry, sb2}	{sb2}
bb3	{loop_sb2, loop_sb3}	sb3	{sb3}	{sb3}
bb4	{loop_sb2, loop_sb3}	Entry	{Entry, sb3, sb2}	{sb3}
bb5	{loop_sb2}	sb3	{Entry, sb2}	{sb2}
bb6	{loop_sb2, loop_sb5}	sb3	{Entry, sb2, sb5}	{sb5}
bb7	{loop_sb2, loop_sb7, loop_sb5}	sb3	{Entry, sb7, sb2, sb5}	{sb7}
bb8	{loop_sb2, loop_sb8, loop_sb7, loop_sb5}	sb3	{Entry, sb7, sb8, sb2, sb5}	{sb8}
bb9	{loop_sb2, loop_sb8, loop_sb7, loop_sb5}	sb9	{sb9}	{sb9}
bb10	{loop_sb2, loop_sb8, loop_sb7, loop_sb5}	sb9	{Entry, sb7, sb8, sb2, sb5}	{sb8}
bb11	{loop_sb2, loop_sb7, loop_sb5}	sb8	{Entry, sb7, sb2, sb5}	{sb7}
bb12	{loop_sb2, loop_sb5}	sb7	{Entry, sb2, sb5}	{sb5}
bb13	{loop_sb2, loop_sb5}	sb7	{sb7}	{sb7}
bb14	{loop_sb2, loop_sb5}	sb7	{Entry, sb2, sb5}	{sb5}
bb15	{loop_sb2, loop_sb5}	sb16	{sb16}	{sb16}
bb16	{loop_sb2, loop_sb5}	sb16	{Entry, sb2, sb5}	{sb5}
bb17	{loop_sb2}	sb5	{Entry, sb2}	{sb2}
bb18	{}	sb2	{Entry}	{Entry}
bb19	{loop_sb2, loop_sb5}	sb7	{sb7}	{sb7}
bb20	{loop_sb2, loop_sb5}	sb7	{sb7, sb7}	{sb7, sb7}

TABLE 3.10: Progressively selected mapping sets

label	σ_{ϕ_c}	σ_{ϕ_m}	σ_{ϕ_b}	σ_{ϕ_r}	final mapping
bb1	sb1, sb21	sb1, sb21	sb1		sb1
bb2	sb2, sb20	sb2, sb20	sb2		sb2
bb3	sb3, sb4	sb3, sb4	sb4		sb4
bb4	sb3, sb4	sb3, sb4		sb3, sb4	sb3
bb5	sb2, sb20	sb2, sb20			sb2
bb6	sb5, sb6, sb12, sb14, sb16, sb19	sb5, sb6, sb12, sb14, sb16, sb19	sb5		sb5
bb7	sb7	sb7			sb7
bb8	sb8, sb9	sb8, sb9		sb9	sb9
bb9	sb10, sb11	sb10, sb11	sb10, sb11	sb11	sb11
bb10	sb8, sb9	sb8, sb9		sb8	sb8
bb11	sb7	sb7			sb7
bb12	sb5, sb6, sb12, sb14, sb16, sb19	sb5, sb6, sb12, sb14, sb16, sb19	sb12		sb12
bb13	sb7				
bb14	sb5, sb6, sb12, sb14, sb16, sb19	sb5, sb6, sb12, sb14, sb16, sb19	sb12		sb12
bb15	sb17, sb18	sb17, sb18	sb17, sb18	sb17	sb17
bb16	sb5, sb6, sb12, sb14, sb16, sb19	sb5, sb6, sb12, sb14, sb16, sb19	sb19		sb19
bb17	sb2, sb20	sb2, sb20	sb20		sb20
bb18	sb1, sb21	sb1, sb21	sb21		sb21
bb19	sb7				
bb20	sb7				

3.5.2 Benchmark Simulation

3.5.2.1 Evaluation of the Method for Basic Block Mapping

An example of generated basic block mapping is shown in Figure 3.23. Two mapping results generated by two different methods for the same *jdct* program are given in Figure 3.23(a) and Figure 3.23(b) respectively. In both parts, the sub-graph G^s and G^b represent the CFG of the source code and the binary code respectively. The edges from the nodes in G^b to the nodes in G^s represent the mapping edges. As can be seen in the left part, the line reference becomes ambiguous due to compiler optimization. For example, the basic block n_2^b and n_5^b in the target binary code contains instructions compiled from multiple source code blocks that are in different loops. By matching the intra-loop control dependency, the annotation position of each binary basic block is correctly resolved as seen in the right part. Matching the dominators would not yield a good mapping. For n_3^b, its dominator is n_2^b which is not a branch node. Most instructions of n_2^b are compiled from n_3^s which dominates n_4^s. However, mapping n_3^b to n_4^s to preserve the dominance is incorrect, because n_4^s is in the innermost loop while n_3^b is not. Simply using the line reference to map n_3^b to n_3^s is not always reliable, because it could also have line references to n_4^s in other cases. It is only through checking the intra-loop control dependency that a robust mapping can be achieved.

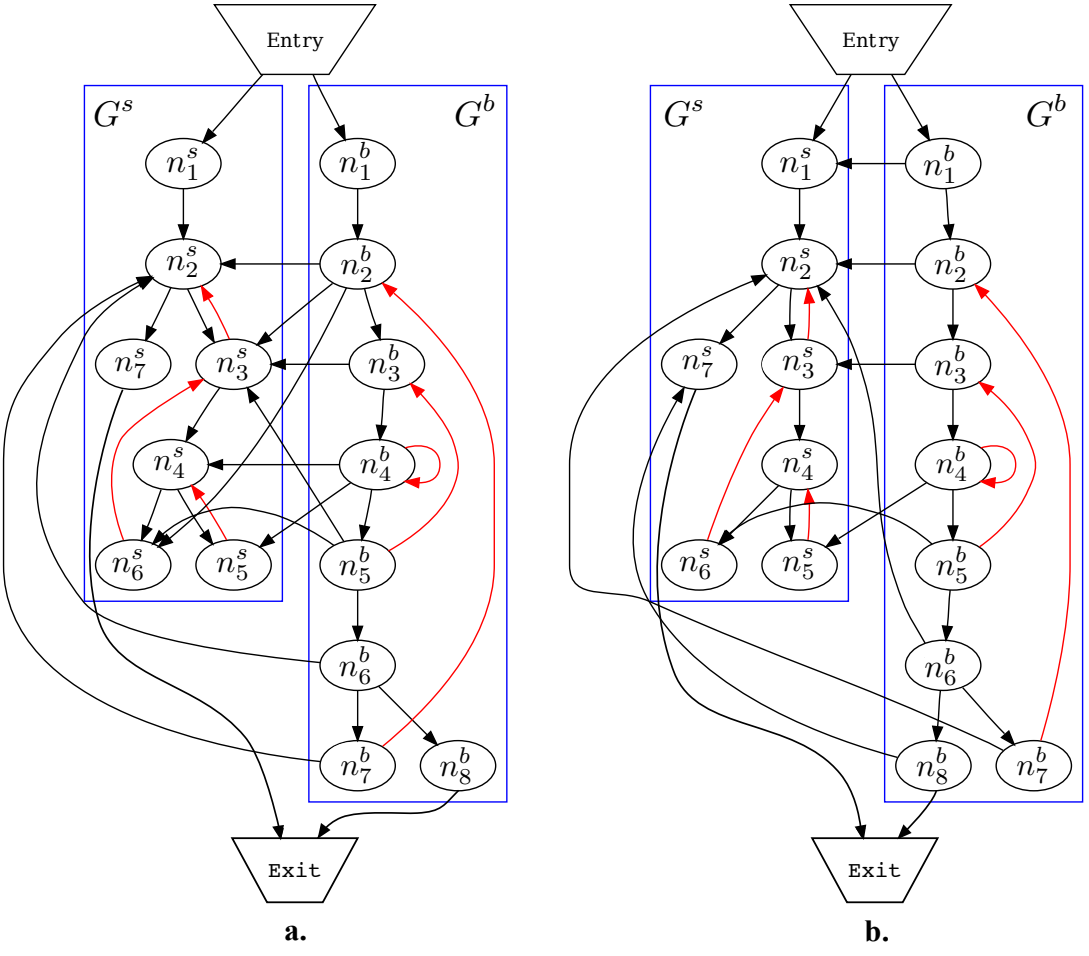


FIGURE 3.23: Mapping graph for the *jdct* program: a) the mapping edges according to the line reference file; b) the mapping graph generated by the present approach

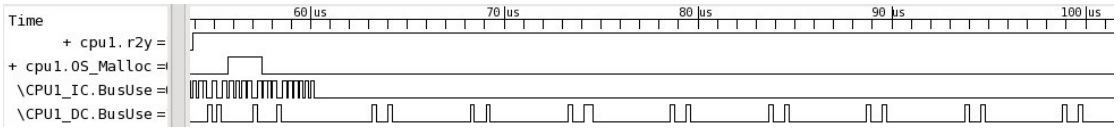
In the simulation, the HW system is modeled in SystemC at transaction-level. As the reference model, an interpretive ISS from [136] is used, which supports MIPS ISA. Firstly, the programs are cross-compiled with optimization level $O2$ and simulated by the ISS. Then the annotated programs provided by the present approach are directly compiled for the simulation host, which is an Intel quadcore 2.33GHz machine running Linux. The simulated cycles in the ISS simulation and host-compiled simulation are compared, which are summarized in Table 3.11. The column rHCS, dHCS and cHCS correspond to the host-compiled simulation (HCS) using line reference based mapping, dominance relation based mapping and the intra-loop control dependency based mapping respectively. The line reference based mapping maps a binary basic block to the source basic block from which most its instructions are compiled from. Compared to the ISS simulation, the simulated cycles for certain simple programs such as *iir* and *r2y* can be well estimated by all three types of mapping. However, for more complex programs, the line reference mapping gives large estimation error. Dominance based mapping is not robust against compiler optimization, and the corresponding annotated source code can not be used as a reliable performance model. This can be seen by the large estimation error for the *edge* program. For the *aes* program, although the overall estimated cycles are close to the ISS simulation for the dominance based mapping, the timing is actually not well estimated for the internally called functions. In the simulation using the present approach, not only the overall timing but also the internal timing are correctly estimated. This is proved by a vcd file tracing all the start and end times of the called functions.

TABLE 3.11: Comparison of simulated cycles.

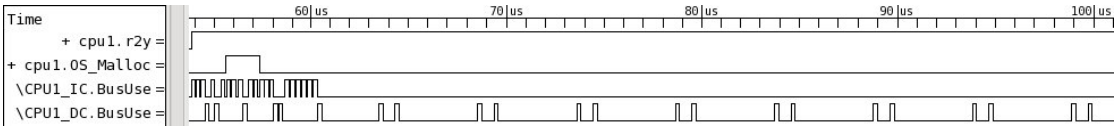
SW	ISS	rHCS	err.(%)	dHCS	err.(%)	cHCS	err.(%)
iir	98590	98481	-0.1	98481	-0.1	98481	-0.1
r2y	134159	132919	-0.92	132919	-0.92	132919	-0.92
fir	233939	248910	-12.9	226740	0.57	233448	-0.2
jpegdct	97418	69247	-27.4	101078	5.9	97475	0.06
isort	89910	109295	22.3	90469	1.2	89996	0.1
aes	12896	10947	15.1	12952	0.43	12867	0.22
edge	1046074	1316336	26.04	804189	-23	1050527	-1.32

TABLE 3.12: Comparison of data cache simulation accuracy

	\$I: N_{access}/N_{miss}		\$D: $N_{read}/N_{write}/\mathbf{N}_{miss}$	
SW	ISS	cHCS	ISS	cHCS
iir	76229/13	76226/13	15257 / 5256 / **71**	15257 / 5256 / **69**
r2y	114772/19	114772/18	2057 / 2063 / **517**	2057 / 2061 / **518**
fir	189980/13	189731/13	30254 / 1903 / **424**	30254 / 1902 / **427**
jpegdct	66877/45	66895/45	16261 / 11256 / **99**	16261 / 11203 / **100**
isort	73139/6	73177/6	8146 / 7953 / **26**	8152 / 7953 / **27**
aes	7551/71	7564/70	1665 / 1160 / **49**	1676 / 1159 / **51**
edge	879263/16	880443/15	155019 / 8397 / **281**	155081 / 8396 / **279**



(a) ISS-based simulation of the *rgb2yuv* program.



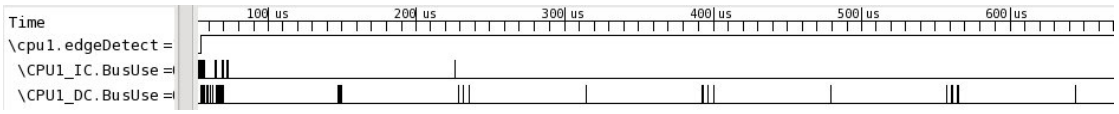
(b) cHCS simulation of the *rgb2yuv* program.

FIGURE 3.24: Traced transactions over bus during the execution of *rgb2yuv*.

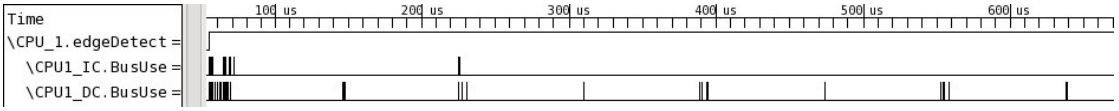
3.5.2.2 Reconstructed Memory Accesses

To examine the annotated memory accesses and cache simulation, the transactions over the bus are traced. Other statistics are also examined, including the memory access count and cache miss count. The statistics are given in Table 3.12. From these results, it can be seen that host-compilation with annotated source code achieves almost identical cache access numbers and miss rates as compared to the ISS simulation. This indicates that the memory accesses are annotated at the proper positions. For example, ISS simulation shows that there are 30254 and 1903 data cache read and write operation in the program `fir()`. Similar numbers are measured in host-compiled simulation. Moreover, it also indicates that the memory access addresses are properly reconstructed. To further prove that the reconstructed memory addresses are correct, the incidence times of the cache misses are examined in the traced transactions. Similar bus access patterns are observed in both the ISS and host-compiled simulation of all the programs. This feature is also useful to arbitrate conflicting transactions in multi-processor simulation. For example, Figure. 3.24 shows the traced transactions for the program *rgb2yuv*, together with the start and end of the executed functions. In this program, a color conversion algorithm is performed on an image buffer. Then the converted frame is written to an output buffer, which is dynamically allocated in the heap. It can be seen that the bus access patterns in both simulations show high agreement. Another example is given in Figure 3.25, showing the traced transactions when simulating the *edge* program, which detects edges from an image frame. High resemblance of the pattern of transactions is again observed in both simulations. Similar observation also holds for other programs.

Additionally, the accessed memory addresses have been traced in the ISS-based simulation and the host-compiled simulation. Comparing the addresses in the two simulations shows that in most cases the annotated addresses in host-compiled simulation equal those in ISS simulation. A few address discrepancies occur within some inlined functions where the binary codes of an inlined function and its caller are together optimized and the debugger gives wrong source-line references for a few instructions. However, overall the estimated cache miss rate and pattern exhibit suitably accuracy. This proves that the memory accesses are successfully reconstructed. The simulation speed-up is shown in Table 3.13. The speed-up over ISS simulation ranges from 16 up to 87. If it is not necessary to trace the occurrence time of the transactions, then the source code can be simulated without calling the wait statement for timing synchronization before invoking a transactions. Doing so will further improve the simulation speed-up. Designers can decide how the simulation should be performed.



(a) ISS-based simulation of the *edge* program.



(b) cHCS simulation of the *edge* program.

FIGURE 3.25: Traced transactions over bus during the execution of *edge*.

TABLE 3.13: Speed-up of host-compiled simulation over ISS simulation.

SW	fir	iir	jdct	isort	aes	r2y	edge
speed-up	55	61	36	46	16	44	87

3.5.3 Case Study: An Autonomous Two-Wheeled Robot

A virtual platform is developed in this case study to simulate an autonomous two-wheeled robot. The robot needs to navigate in a virtual 3D environment by following a red line. This is realized by several tasks that control the direction and balance of the robot. Host-compiled simulation is applied for fast development and simulation of the controlling tasks in the software. The main components of the simulation platform are given in Figure 3.26(a). The dashed block shows the hardware and software components, which are prototyped in SystemC and TLM 2.0. The employed ISS supports the OpenRISC architecture. It is wrapped in a TLM model of the hardware system. Details of the implementation can be found in [137].

The main controlling tasks are shown in Figure 3.26(b). The `pathControl` task determines the new heading direction and velocity of the robot. It is executed after each received camera frame. During its executions, transactions are sent over the bus to read the sensor values. The `motionControl` [138] task balances the robot and calculates the torque values of the two wheels to achieve the targeted heading direction and velocity. It is executed periodically, for example every $3ms$. During its execution, it reads the sensors and writes the actuators via transactions. The SW is compiled with the optimization level O2. The overall temporal order of the executed software tasks is sketched in Figure 3.26(c). The most complex and timing critical task is the `pathControl`, which contains 7 loops and 100+ basic blocks. Once it completes, the new heading direction and velocity are updated, which will subsequently be used in the `motionControl` to calculate the new torque values sent to the actuator of each wheel. In this case, timing becomes indispensable to assess the effectiveness and quality of the control task.

3.5.3.1 Simulation Results

In the study of the control tasks of the robot, the designer needs to simulate many different shapes of lines, environments and system configurations. Each simulation takes millions of cycles to simulate, therefore timed host-compiled simulation is especially

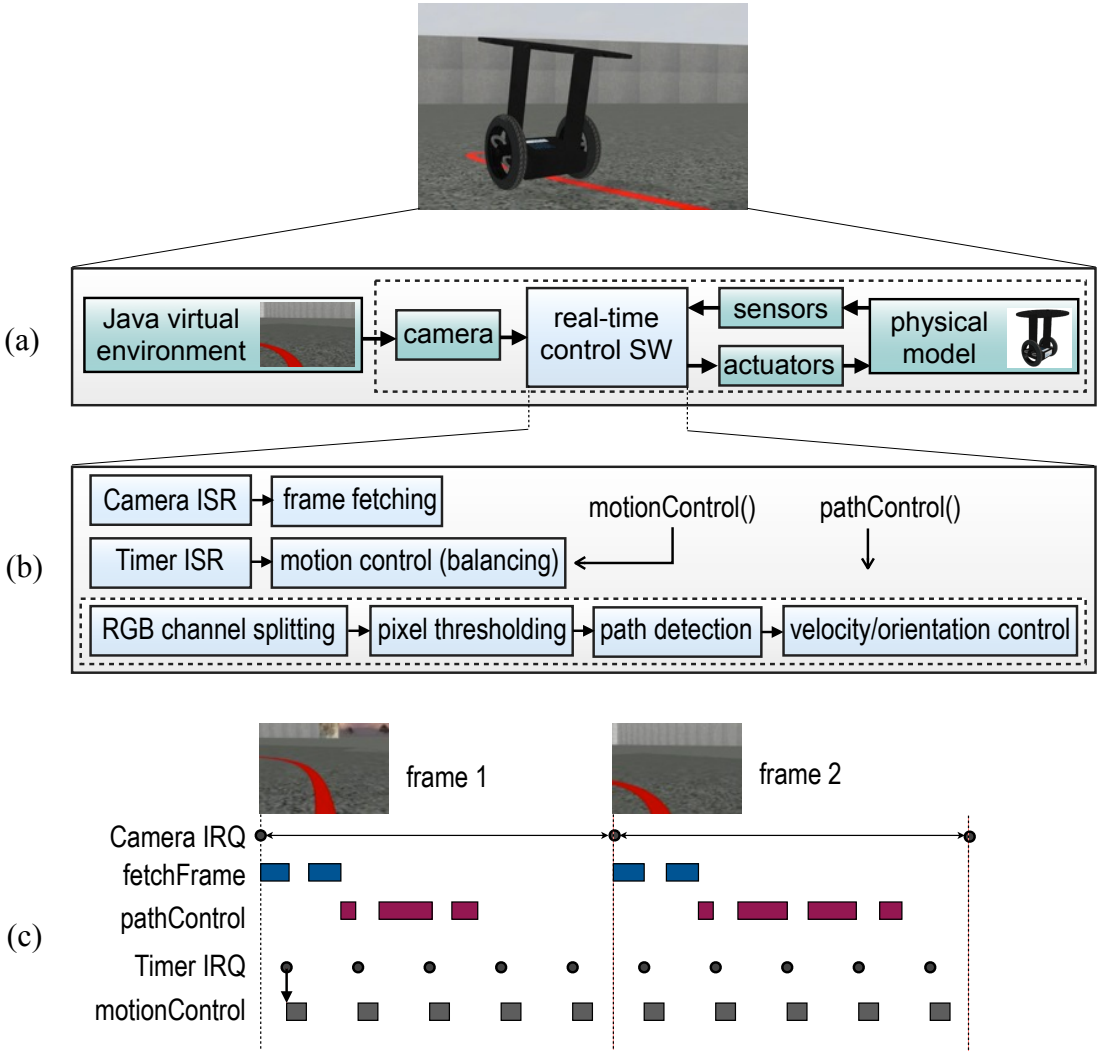
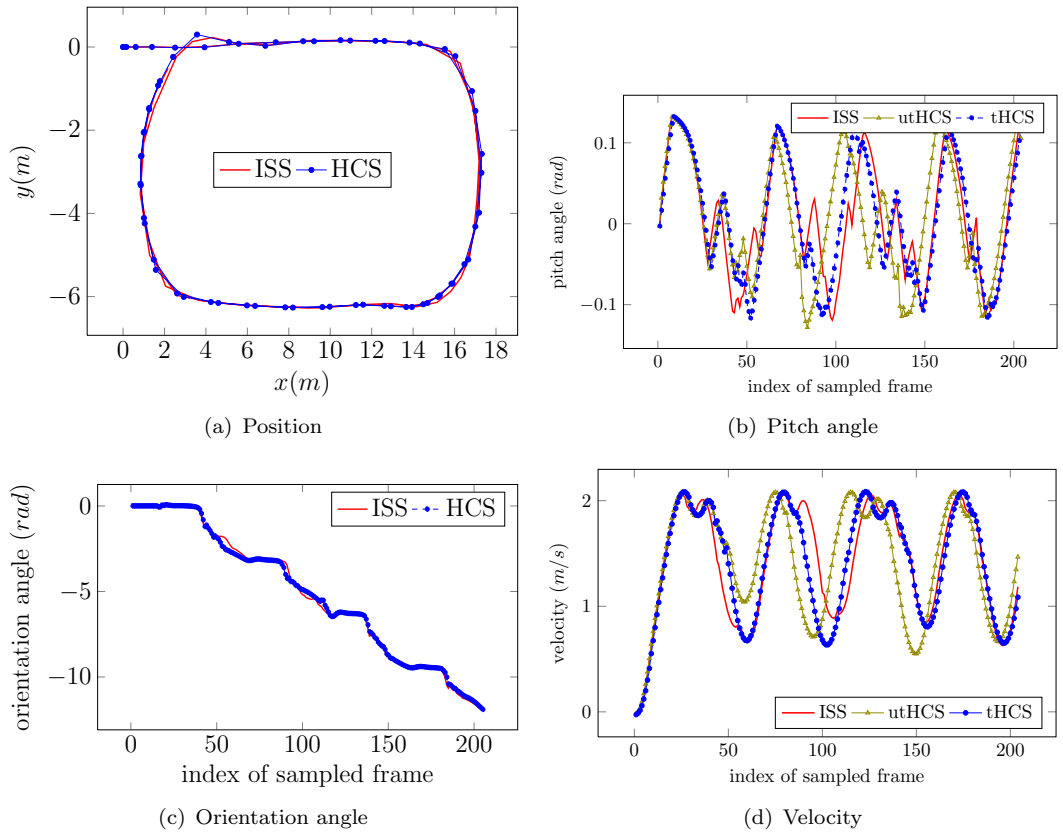


FIGURE 3.26: The virtual simulation platform: (a) Main components in the platform; (b) Software tasks that run on the underlying hardware system; (c) Illustration of the execution order of the software tasks

necessary in this long-scenario simulation. To evaluate the usefulness of timed host-compiled simulation, the following aspects are considered: (1) It should be able to reveal the correct system dynamics of the robot. (2) It should be able to provide approximate timing estimation. (3) It should be sufficiently fast, so that the simulation can be carried out in real time. In the experiment, the physical status parameters of the robot and the timing of the control tasks are traced in both ISS-based simulation and timed host-compiled simulation (HCS) for comparison.

In the first experiment, the robot needs to follow a line in an in-door environment. The simulated x-y coordinates of the robot are given in Figure 3.27(a). These curves lie in close proximity to the red line that the robot is supposed to follow. This verifies the functional correctness of the line recognition and path control algorithm. Besides the functional verification, designers also need to inspect physical status of the robot to better evaluate the quality of the control algorithm. For example the pitch angle can provide information regarding the stability of the robot. The results regarding the physical status parameters of the robot are given in Figure 3.27(b), Figure 3.27(c), and



(a) Position (b) Pitch angle

(c) Orientation angle (d) Velocity

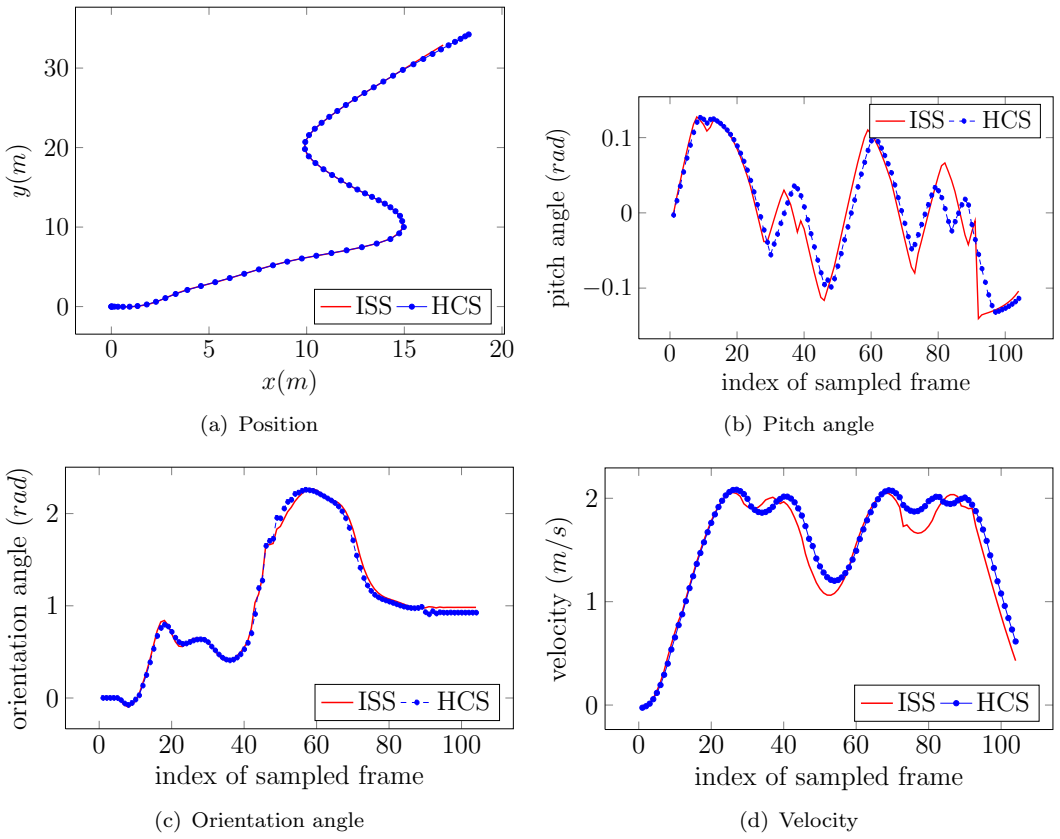
FIGURE 3.27: Experiment using line 1: Simulated status parameters of the robot.

Figure 3.27(d), which plot the pitch angle, the orientation angle and the velocity of the robot, respectively. It can been seen that the fluctuation of the curves shows suitable agreement with that in ISS-based simulation. Similar consistency is observed for other parameters, including the torque values sent by the motion control task to the actuators.

In the second experiment, the robot needs to follow a line with different curvature in an out-door environment. The simulated x-y coordinates of the robot are given in Figure 3.28(a). The traced physical status parameters are shown in Figure 3.28(b), Figure 3.28(c), and Figure 3.28(d). As in the previous experiment, the results in timed host-compiled simulation exhibit similar system dynamics as those in ISS-based simulation.

To examine the correctness of the annotated timing, the measured execution cycles for the path control task frames in ISS-based simulation and timed host-compiled simulation (HCS) are given in Figure 3.29. In most cases, the estimated cycles in HCS match well with those in ISS-based simulation. The maximum estimation error is about 12% for the sampled frame 210.

To test the simulation speed, a 60 seconds scenario is simulated. This corresponds to a simulated time period of 60 seconds. The simulation finishes in about 20 seconds in timed host-compiled simulation. However, it takes around 10 minutes in ISS-based simulation. This implies a speed-up of about 30x, which is very high considering that the virtual environment rendering is also performed in HCS. Therefore, real-time simulation with well estimated timing is achieved in the HCS using the proposed approach.



(a) Position (b) Pitch angle

(c) Orientation angle (d) Velocity

FIGURE 3.28: Experiment using line 2: Simulated status parameters of the robot.

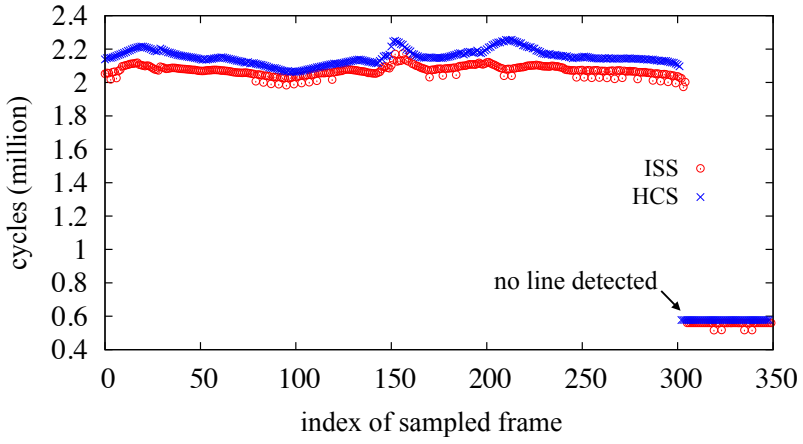


FIGURE 3.29: Estimated cycles for the path control task in ISS based and host-compiled simulation.

To conclude, the experimental results show that the timed host-compiled simulation can simulate both timing and system dynamics with suitable accuracy. This satisfies the aforementioned criterion, proving its usefulness. Additionally, due to its fast speed, simulating long software scenarios becomes affordable. Different design options, such as changing the task scheduling algorithm, clock frequencies, and control parameters, can be validated much faster than using ISS-based simulation.

Chapter 4

Analytical Timing Estimation for Faster TLMs

Subsequent to the new modeling techniques such as TLM+ and temporal decoupling which aim at faster and more abstract TLMs, the confronted challenge of maintaining timing simulation accuracy needs to be tackled. In this chapter, Section 4.1 briefly summarizes the advantages of the present approach. Then, the challenge of timing estimation is formally described in Section 4.2, in which it is decomposed into three sub-problems. Following that, the solution to each sub-problem is presented. Firstly, Section 4.3 elaborates the extraction of resource utilization. Secondly, Section 4.4 details the calculation of resource availability. Thirdly, using these two terms, Section 4.5 applies a novel analytical formula in estimating the delay induced by resource conflicts. Furthermore, Section 4.6 discusses the implementation of the proposed approach in an on-line scheduler, so that it can be used in combination with existing transaction-level models. At last, Section 4.7 applies the approach for experimental evaluation.

4.1 Contributions and Advantages

This work presents an analytical approach for timing estimation. It features analytical formulas that model the resource utilization, resource availability and delay due to resource conflicts. The timing estimation is implemented by a scheduler which enables dynamic timing adjustment during simulation. The advantages of the present analytical timing estimation are summarized as follows:

- Timing estimation is provided by formulas that take into account the average timing characteristics, instead of performing conventional arbitration of bus-word transactions.
- Because it does not arbitrate the bus-word transactions, it does not require that the occurrence times of the bus-word transactions should be known. This not only saves the effort of tracing the bus-word transactions, but also is especially advantageous when it is not feasible to do so. For example, in the case where TLM+ transactions are used, the underlying bus-word transactions are not modeled and their exact occurrence times are therefore unknown.

- Granularity of timing synchronization can be determined *on demand* during simulation. This is achieved by the scheduling algorithm that dynamically adjusts the end time of each timing synchronization. Therefore, it does not require a global quantum with pre-configured value.
- The computation overhead of the scheduling algorithm is very low. Therefore the speed-up gained by using temporal decoupling can be preserved. Rescheduling is performed by dynamic event cancellation and re-notification. Therefore, only a single SystemC *wait* is required to schedule a synchronization request of a very long period.
- This methodology is implemented as a library that conforms to the TLM 2.0 standard. Existing TLM VPs can be easily ported.
- It proposes a methodology for extracting the timing profiles of TLM+ transactions. These timing profiles are used in estimating the duration of a TLM+ transaction and its resource utilization.

4.2 Overview of the Timing Estimation Problem

4.2.1 Terms and Symbols

Using temporal decoupling, in effect, divides a SystemC process into large code parts and piecewise simulates each code part. The simulation of each code part is separated into two phases regarding the functional and temporal aspects:

- In the first phase, functional simulation is carried out in a *un-timed* way. During this phase, timing information may be collected without actually resorting to timing synchronization. The transactions that take place in this phase are also un-timed.
- In the second phase, timing simulation is performed by issuing a timing synchronization request for the duration of the codes simulated in the first phase. Now, after such a request is issued, the task is to estimate additional delay due to resource conflicts. According to the estimated delay, timing adjustment can be made to the actual duration of the corresponding phase of functional simulation.

The present timing estimation methodology features analytical formulas that model the timing conflicts at shared resources. This methodology is agnostic to when each individual bus-word transaction occurs during the functional simulation. It requires only the timing information such as the accumulated access time collected during simulation.

The terms and symbols used by the analytical timing estimation methodology are given in the following. Afterwards, the considered timing estimation problem will be decomposed into three sub-problems.

R is a set of resources corresponding to the set of system modules in a transaction-level model. One resource is denoted as R, with $R \in \mathbf{R}$. The total number of resources is denoted by N_r.

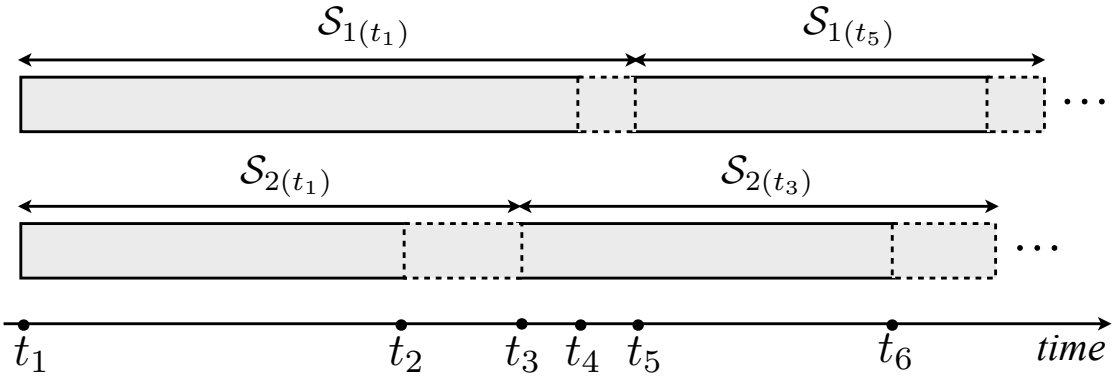


FIGURE 4.1: Example scenario of considered abstract timing requests.

Abstract timing request: An abstract timing request models the timing aspects of a piece of code. In contrast to the standard SystemC *wait* statement, an abstract timing request does not only contain a simple time interval, but also other timing information related to the computation and communication aspects such as the execution time and resource access time. The timing information may be gathered during the un-timed simulation, which corresponds to a very long time period including a sequence of bus-word transactions. Or, it may correspond to a TLM+ transaction that transfers a large data block.

$\mathcal{S}_i$ denotes one abstract timing request that is initiated by initiator i. Additionally, the notation $\mathcal{S}_{i(t)}$ is used to explicitly refer to $\mathcal{S}_i$ that starts at time t. In Figure 4.1, $\mathcal{S}_{1(t_1)}$ and $\mathcal{S}_{1(t_5)}$ represent two synchronization requests starting at t_1 and t_5 respectively.

The considered temporal properties for $\mathcal{S}_i$ include:

p_i denotes the requested duration, assuming there is no delay caused by timing conflicts at shared resources. For $\mathcal{S}_{2(t_1)}$ in Figure 4.1, $p_2 = t_2 - t_1$.

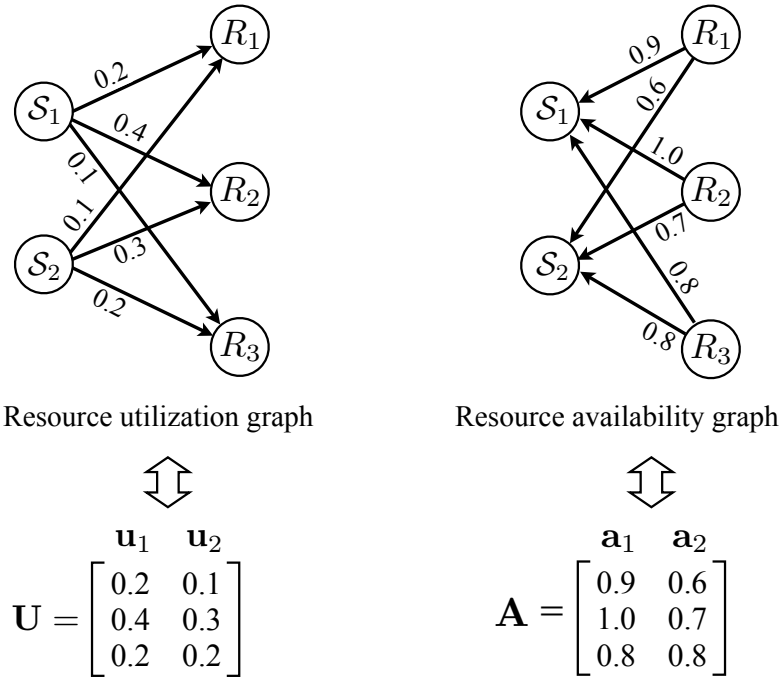
d_i denotes the overall delay that is experienced by $\mathcal{S}_i$. For $\mathcal{S}_{2(t_1)}$ in Figure 4.1, $d_2 = t_3 - t_2$.

$d_{i,m}$ represents the delay for $\mathcal{S}_i$ at resource R_m.

$\mathbf{u}_i$ is a vector of resource utilization demanded by $\mathcal{S}_i$, expressed as $\mathbf{u}_i = [u_{i,1}\ u_{i,2}\ ...\ u_{i,N_r}]^T$. The utilization $u_{i,m} \in [0, 1]$ models how much a resource R_m is accessed by $\mathcal{S}_i$. For example, for the given resource utilization graph in Figure 4.2, it follows that $\mathbf{u}_1 = [0.2\ 0.4\ 0.2]^T$. This vector can also be interpreted as a use case of the resources in terms of timing.

$\mathbf{a}_i$ is a vector of resource availability for $\mathcal{S}_i$, expressed as: $\mathbf{a}_i = [a_{i,1}, a_{i,2}, ..., a_{i,N_r}]$. The availability $a_{i,m} \in [0, 1]$ quantifies how much a resource is accessible to $\mathcal{S}_i$ under the presence of resource conflicts. In Figure 4.2, for the given resource availability graph, it follows that $\mathbf{a}_2 = [0.6\ 0.7\ 0.8]^T$.

The set of ongoing abstract timing requests seen at time t is denoted by $\mathbf{S}_{(t)}$. In Figure 4.1, $\mathbf{S}_{(t_1)} = \{\mathcal{S}_{1(t_1)}, \mathcal{S}_{2(t_1)}\}$, where both $\mathcal{S}_{1(t_1)}$ and $\mathcal{S}_{2(t_1)}$ start at t_1. Likewise, at time t_5, $\mathbf{S}_{(t_5)} = \{\mathcal{S}_{1(t_5)}, \mathcal{S}_{2(t_3)}\}$, where $\mathcal{S}_{1(t_5)}$ starts at t_5 and $\mathcal{S}_{2(t_3)}$ starts at t_3. With a given $\mathbf{S}_{(t)}$, following terms are described.



$$\mathbf{U} = \begin{array}{c} \begin{matrix} \mathbf{u}_1 & \mathbf{u}_2 \end{matrix} \\ \begin{bmatrix} 0.2 & 0.1 \\ 0.4 & 0.3 \\ 0.2 & 0.2 \end{bmatrix} \end{array} \qquad \mathbf{A} = \begin{array}{c} \begin{matrix} \mathbf{a}_1 & \mathbf{a}_2 \end{matrix} \\ \begin{bmatrix} 0.9 & 0.6 \\ 1.0 & 0.7 \\ 0.8 & 0.8 \end{bmatrix} \end{array}$$

FIGURE 4.2: Exemple resource utilization and availability matrices.

U is a resource utilization matrix, with $\mathbf{U} = [\mathbf{u_1}, \mathbf{u_2}, ..., \mathbf{u_{N_S}}]$, where N_s is the number of ongoing abstract timing requests. **U** is a matrix representation of an equivalent resource utilization graph. An example is given in Figure 4.2-left.

A is a resource availability matrix composed of the resource availability vectors, with $\mathbf{A} = [\mathbf{a}_1, \mathbf{a}_2, ..., \mathbf{a}_{N_S}]$. An example is given in Figure 4.2-right.

p is a vector of the requested durations assuming no access conflicts, written as $\mathbf{p} = [p_1, p_2, ..., p_{N_S}]$.

d is a vector of the delays of the abstract timing requests, written as $\mathbf{d} = [d_1, d_2, ..., d_{N_S}]$.

4.2.2 Problem Description

Now the problem can be stated as such: Given a set of abstract timing requests $\mathbf{S}_{(t)}$ at time t, estimate the additional delays due to resource conflicts for all abstract timing requests and hence determine their actual durations. The task of delay estimation is further decomposed into three sub-tasks, as outlined in Figure 4.3. Details of each sub-task are described below.

1. In step 1, timing characteristics are extracted during the functional simulation, in order to calculate the resource utilization for each abstract timing request and the utilization matrix **U** of $\mathbf{S}_{(t)}$. The extracted timing parameters for an abstract timing request include the requested duration without conflicts and the total access time to each resource. Notice that it does not store the incidence time of each individual bus-word transaction.

2. In step 2, the task is to estimate the resource availability for each abstract timing request, in order to obtain the availability matrix **A** of $\mathbf{S}_{(t)}$. Instead of being

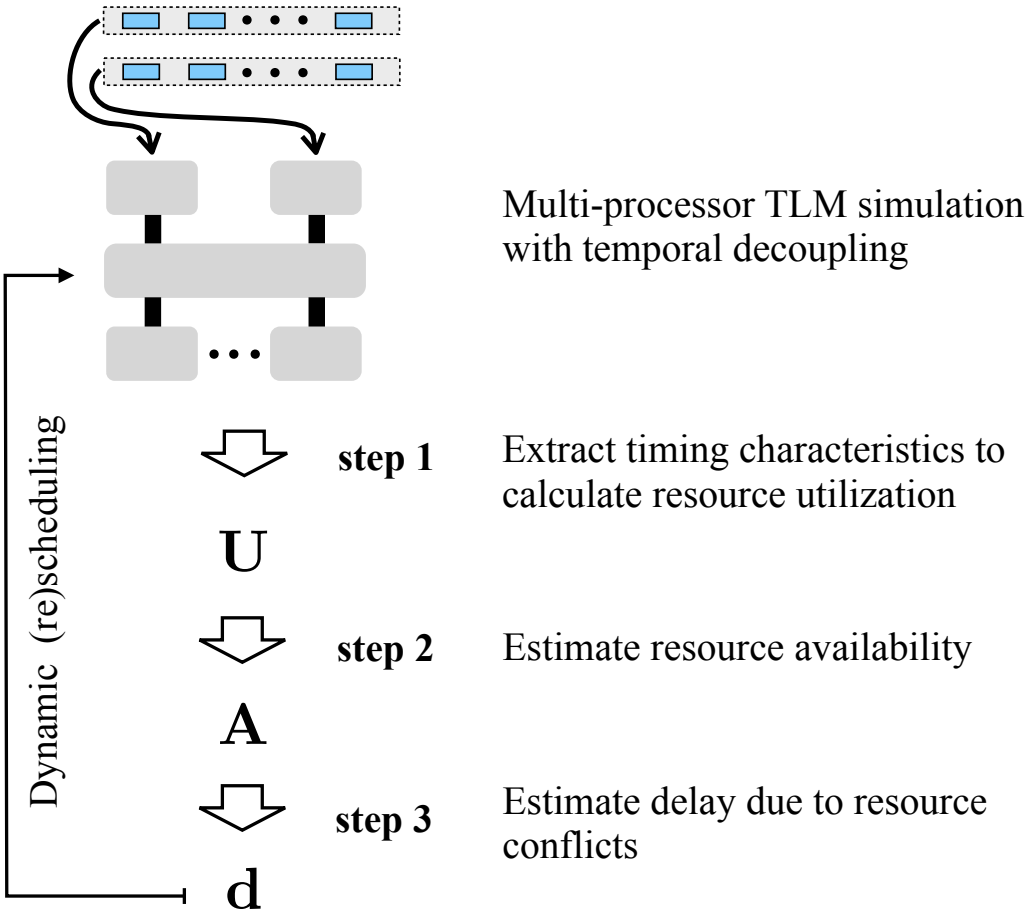


FIGURE 4.3: Decomposition of the problem into three sub-problems.

explicitly simulated, resource availability is derived from the overall resource utilization of the ongoing abstract timing requests. The derivation should consider aspects that may influence the way a resource is granted, such as the arbitration policy, bus protocol, etc.

3. In step 3, the task is to estimate the delay for each $\mathcal{S}_i$ in $\mathbf{S}_{(t)}$. This estimation takes as input parameters the remaining conflict-free duration, resource utilization and resource availability. First, it estimates the delay $d_{i,m}$ of $\mathcal{S}_i$ at each individual resource R_m:

$$d_{i,m} = \phi(p_i, u_{i,m}, a_{i,m}), \tag{4.1}$$

where ϕ is a delay estimator that models the delay due to resource conflicts. Then it calculates the overall delay d_i. The estimated duration of $\mathcal{S}_i$ is therefore $p_i + d_i$. Combining these operations, this step can be written as:

$$\mathbf{d} = \Phi(\mathbf{p}, \mathbf{U}, \mathbf{A}), \tag{4.2}$$

meaning that the delays $\mathbf{d}$ for all abstract timing requests are estimated from the timing parameters $\mathbf{p}, \mathbf{U}$ and $\mathbf{A}$. With the estimated delay, the end time of each abstract timing request can be subsequently adjusted.

During simulation, the above tasks need to be performed each time an abstract timing request is issued, which can change the scenario of resource availability for other ongoing

abstract timing requests. Therefore, as depicted in the figure, an additional requirement is to perform dynamic rescheduling, through which the analytical timing estimation can be integrated with simulation.

4.3 Calculation of Resource Utilization

Resource utilization represents the degree of timing usage of a certain resource. It is averaged as the total access time over a certain time period. Extracting these timing parameters further depends on the way how the system is modeled. It is necessary to distinguish the levels of abstraction. For a TLM+ transaction that transfers a large data block, timing characteristics of the corresponding driver function should be extracted off-line, e.g. by profiling. When using bus-word transactions and temporal decoupling, timing characteristics should be collected on-line. There two cases are elaborated in the following.

4.3.1 Simulation Using Bus-Word Transactions

One bus-word transaction transfers a unit of data that fits the bus protocol, such as a byte, a word or a burst of words. Usually, bus-word transactions occur when data are fetched from or stored to memory due to cache misses or when the processor directly communicates with other system modules. Let T denote a a bus-word transaction, and $\tau_m(T)$ denote T's access time of resource R_m. For $\mathcal{S}_i$ consisting of a long sequence of bus-word transactions, a trace $\mathcal{T}$ can represent those transactions:

$$\mathcal{T} = \{T_1, T_2, ..., T_N\}.$$

Notice that it is not required to trace when exactly each bus-word transaction occurs. Next, the total access time Γ_m of a resource R_m during $\mathcal{S}_i$ is calculated as:

$$\Gamma_m = \sum_{j=1}^{N} \tau_m(T_j)$$

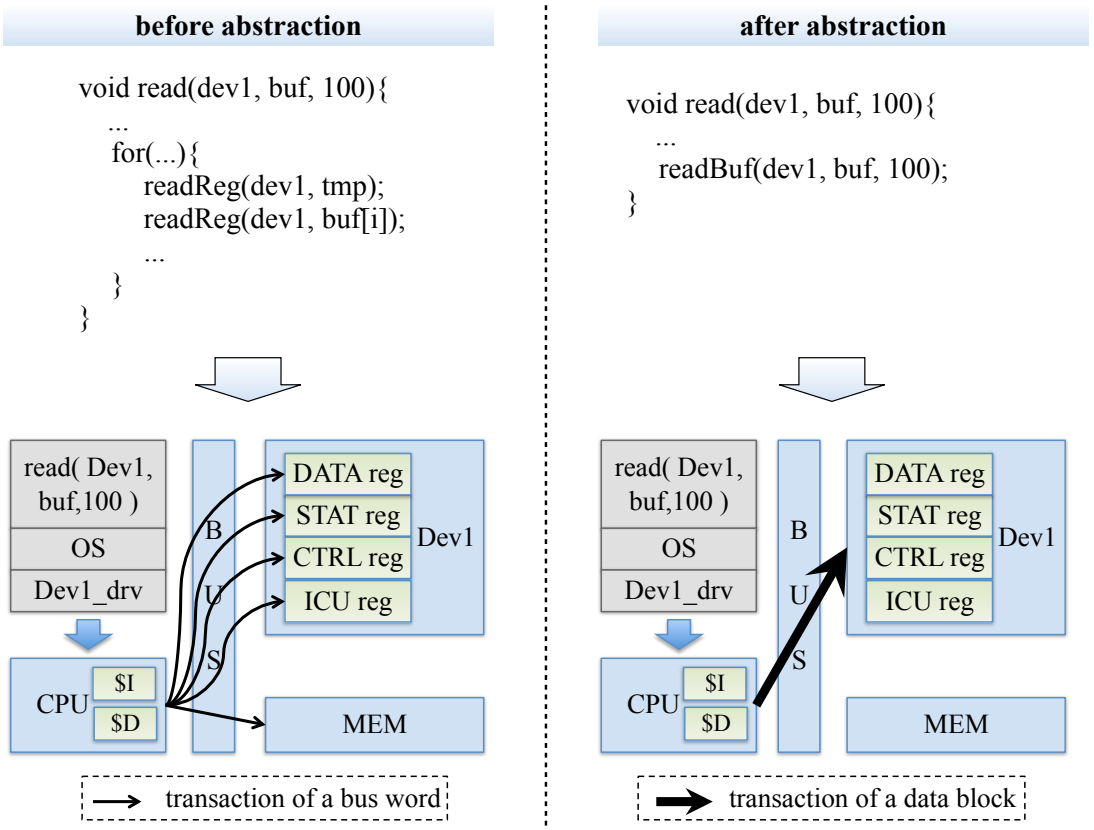
On the other hand, the duration of $\mathcal{S}_i$ assuming no resource conflicts is given by:

$$p_i = t_{comp} + t_{comm} = t_{comp} + \sum_{i=1}^{N} \tau(T_i),$$

where t_{comp} is the locally accumulated computation time and $\tau(T_i)$ is the conflict-free duration of the bus-word transaction T_i.

With the above results, the utilization $u_{i,m}$ of R_m demanded by $\mathcal{S}_i$ is obtained:

$$u_{i,m} = \frac{\Gamma_m}{p_i}.$$



(a) **Before abstraction**: many word transactions evoked by the driver function

(b) **After abstraction**: complete data block transferred in a *single* TLM2.0 transaction.

FIGURE 4.4: TLM+ transactions: concept and implementation.

4.3.2 Simulation Using TLM+ Transactions

The concept of more abstract transactions, termed as TLM+ transactions, has been recently proposed [14]. As shown in Figure 4.4, one single TLM+ transaction can transfer a very large data block. Such abstraction bypasses the underlying driver functions, thus saving modeling and computation effort. Compared to the bus-word transactions which abstract away the HW signal handshaking between two modules, the TLM+ transactions abstract away the SW protocols between two modules.

However, it becomes difficult to extract timing information of the TLM+ transactions, since a great amount of detail is missing due to abstraction. For example, to transfer a data block of 100 words from memory to an I/O module, there may be thousands of instructions executed by the corresponding driver function and hundreds of bus-word transactions initiated during the transfer. After abstraction, the transfer is implemented by a single transaction, within which the lower level timing information is invisible. As a result, it is unclear how to calculate the duration and resource utilization for such a TLM+ transaction. Previously, it has not been proposed how to extract sufficient timing characteristics for the block transactions. Designers often count on empirical values for timing estimation [14, 139]. In addition, because the underlying bus-word transactions are invisible in a TLM+ transaction, it becomes difficult to arbitrate the timing if resource conflicts exist between a TLM+ transaction and other overlapping processes. In the present approach, timing estimation is based on the overall resource

access time and does not require the availability of the occurrence time of individual bus-word transactions. Therefore it can tackle the challenge of timing estimation in case TLM+ transactions are used.

In the following, a method that can extract the timing information from the original driver functions is presented. With these information, the TLM+ transactions can be timed with sufficient accuracy. There are two main steps performed in this method.

4.3.2.1 Extracting Timing Profiles of TLM+ Transactions

In the scope of this work, TLM+ transactions are abstracted from driver functions that implement the low-level software protocols for the data transfer. The profiling process will extract timing characteristics of the corresponding driver functions. The tool chain is given in Figure 4.5. As marked in the figure, each step is described in the following.

1. A profile function is constructed, which calls the driver functions of different types to transfer different data blocks. These driver functions implement the detailed SW protocols for the data transfer.
2. **Finding the entry and exit addresses:** Using debugging information, the addresses of the instruction in the cross-compiled binary are known. Then this tool parses this address file and extracts the addresses of the *entry* and *exit* instruction of all functions, including the driver functions.
   - The *entry* instruction is the first instruction of the function body, as the one at address *0x100* in List 4.1.
   - The *exit* instructions are those corresponding to the return of the function, such as those at address *0x340* and *0x4a0* in List 4.1. [1]

   Here, the address *0x100* is the offset address of this function in the binary code compiled from the source code file. The translation unit corresponding to the source code file will be compiled to a base address in the final executable. The base address can be obtained by the *read-elf* utility. Assuming a base address *0x2000*, then the entry and exist addresses for the example function are therefore *0x2100* and *0x2344, 0x24ac*.

```
1  00000100 <foo>:
2   100: 8f890000    lw     t1,0(gp)          <-- entry
3   104: 27bdffe8    addiu  sp,sp,-24
4   108: 308400ff    andi   a0,a0,0xff
5   10c: 000419c2    srl    v1,a0,0x7
6
7   . . . . . .
8
9   340: 03e00008    jr     ra
10  344: 00000000    nop                      <-- exit
11
12  . . . . . .
```

[1] For a MIPS ISA, the instruction immediately following a branch instruction will always be executed once the branch instruction is executed. This is because the compiler swaps the branch instruction and the one preceding it in the original code for performance improvement. Therefore, although the *jr* is the return instruction, the real exit instruction is the one after it.

```
13
14   4a8: 03e00008    jr     ra
15   4ac: a043000e    sb     v1,14(v0)       <-- exit
16
17   . . . . . .
```

LISTING 4.1: Entry and exit instructions.

The entry and exit addresses are referred to as the instruction space boundaries (*ISB*). A exemplary file of the extracted (*ISB*) is given in List 4.2.

```
1  // function address boundaries of the driver functions
2  Function   read_aes       entry: 0x1034   exit: 0x10a4
3  Function   write_aes      entry: 0x10b4   exit: 0x1124
4  Function   read_camera    entry: 0x13b0   exit: 0x13fc
5  Function   write_camera   entry: 0x140c   exit: 0x147c
6  Function   read_lcd       entry: 0x2ee4   exit: 0x2f54
7  Function   write_lcd      entry: 0x2f64   exit: 0x2fa4
8  Function   write_sif      entry: 0x35bc   exit: 0x364c
9  Function   read_sif       entry: 0x3650   exit: 0x3738
10 Function   read_dmac      entry: 0x1718   exit: 0x171c
11 Function   write_dmac     entry: 0x1720   exit: 0x1724
12 // function address boundaries of other functions
13 Function   OS_ISR         entry: 0x95c    exit: 0x960
14 Function   init_device    entry: 0x964    exit: 0x9b8
15 Function   init_isr       entry: 0x9bc    exit: 0x9cc
16 Function   openf          entry: 0xa94    exit: 0xab4
17 Function   closef         entry: 0xab8    exit: 0xad8
18 Function   initf          entry: 0xadc    exit: 0xafc
19 Function   ioctlf         entry: 0xb00    exit: 0xb20
20 Function   writef         entry: 0xb24    exit: 0xb44
21 Function   readf          entry: 0xb48    exit: 0xb68
22 Function   strncat        entry: 0x3bd8   exit: 0x3c20   0x3c4c
23 Function   memcpy         entry: 0x3d58   exit: 0x3d90   0x3dc0
24 Function   memmove        entry: 0x3dc4   exit: 0x3df4   0x3e20
25 Function   memcmp         entry: 0x3e24   exit: 0x3e50   0x3e58
26 Function   memset         entry: 0x3e5c   exit: 0x3e84
27 Function   strcpy         entry: 0x3b40   exit: 0x3b5c
28 . . . . . .
```

LISTING 4.2: Extracted entry and exit addressses of all functions.

3. A timing accurate ISS is used to simulate the profile function. For the tracing purpose, a SW monitor is added to the ISS. At the start of simulation, the SW monitor reads in all the *ISBs*.

4. During the simulation, if the SW monitor detects that the instruction interpreted by the ISS enters or exits a function (e.g. `jal` or `jr` instruction for a MIPS CPU), then it checks the address against those extracted *ISBs*. In this way, the start and end of the driver functions, can be traced exactly.

   In addition, the bus-word transactions are traced so that the accesses to the resources during the driver functions are obtained.

5. As a result, a trace file is generated from which timing characteristics are extracted during the execution of each driver function. This trace file provides timing models

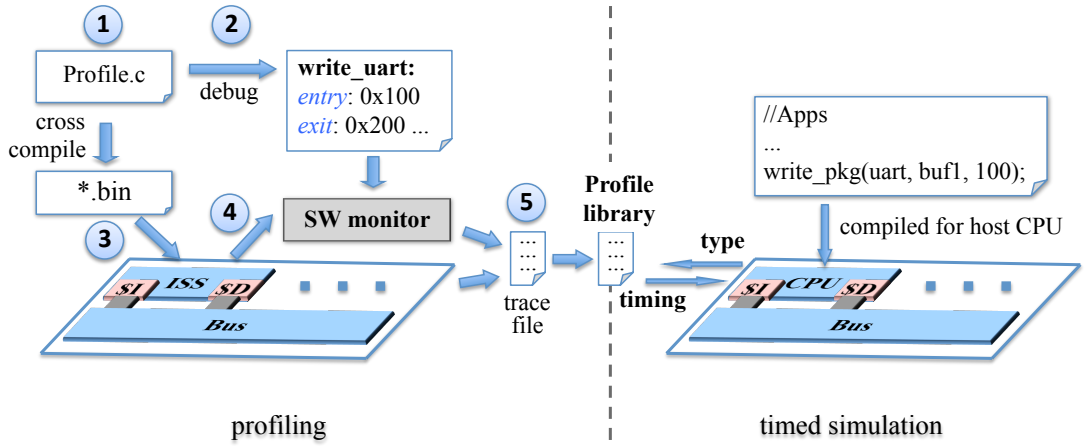


FIGURE 4.5: A tool-chain to profile the driver functions and extract timing characteristics of the TLM+ transactions.

of the driver functions and hence can be used to time the TLM+ transactions. For each type of driver functions, this profile contains:

- λ_{ave}: the average time to transfer one unit of data in the block.
- $\{\lambda_1, \lambda_2, ..., \lambda_{N_R}\}$, with λ_i being the average time a resource R_i is accessed during the transfer of one unit of data.
- $\{addr_1, addr_2, ...\}$: the addresses of instructions during the execution of the driver function. These addresses will be used later for instruction cache simulation.

Notice that the driver function involves the transfer but not the processing of the data block. Therefore, it can be assumed that the duration of the driver function is approximately linear to the size of data block, since it is not dependent on the value of the transferred data. This is in accordance with the general idea of separating the computation and communication in TLMs.

```
1  write(dev1, buf, size){
2    //1. a TLM2.0 transaction
3    ...
4    gp->set_data_ptr(reinterpret_cast<UCHAR*>(buf) );
5    gp->set_data_length(size);
6    socket->b_transport( *gp, delay );
7
8    //2. get timing from the profile library (plib)
9    ty=getDriverType('drv1 write');
10   p = plib.getStaticDuration(ty,size);
11   timeIMem = plib.penaltyICache();
12   timeDMem = plib.penaltyDCache();
13   p += timeDMem + timeIMem;
14   timeBus = plib.getAccessTime(ty, busID, size);
15   timeBus += timeDMem+timeIMem;
16   ...
17 }
```

LISTING 4.3: Sample of a TLM+ transaction with annotated timing.

4.3.2.2 Estimated Duration of TLM+ Transactions

With the profile of the timing characteristics, the timing of the TLM+ transactions can be annotated and estimated in simulation. An example is given in List 4.3. As can be seen, in line 6, a single TLM+ transaction is used to transfer a complete data block, based on the OSCI TLM 2.0 standard. After performing this un-timed TLM+ transaction, timing is estimated by additional codes in line 9 to 15. The type of the driver functions is used to query the profile library. The required duration p_i of a TLM+ transaction is given by

$$p_i = p_{static} + p_{dynamic}, \tag{4.3}$$

where p_{static} is a statically estimated part of the duration and $p_{dynamic}$ is a dynamically estimated part capturing the cache miss penalty. These two terms are computed in the following way.

- p_{static} is computed as

  $$p_{static} = T_{ave} * N_{size}, \tag{4.4}$$

  where N_{size} is the size of the data block. For the examined driver functions in the code inherited from [124], the transfer time is approximately linear to the size of the transferred data block. It is also possible that the statically estimated duration is dependent on the data size. For example, the duration can increase differently within different ranges of the data size. This case can also be handled by modifying 4.4 accordingly:

  $$p_{static} = \sum_{i=1}^{r} (T_{ave_i} * N_i), \tag{4.5}$$

  where i is the index of each range, T_{ave_i} is the averaged transfer time for one data unit in a certain range, and N_i is the size of each range with $N_{size} = \sum_{i=1}^{r} N_i$. Before the transfer, the driver function may need to complete certain initialization handshaking with device. If necessary, this additional time can be added as an offset time to the estimated duration.

- $p_{dynamic}$ is the timing penalty given by the instruction and data cache simulation. The accessed addresses for the instruction cache simulation are those that are extracted in the timing profile. The accessed addresses for the data cache simulation are obtained by means of the method discussed in Section 3.4.

4.3.2.3 Compute the Resource Utilization

After the estimation of the duration of a TLM+ transaction, the utilization of each resource for this transaction can be obtained by dividing the total resource access time by the estimated duration.

For a resource R_m corresponding to a peripheral I/O module, the access time is not influenced by the cache penalty. Its total access time is estimated as

$$\Gamma_m = \lambda_m \cdot N_{size}. \tag{4.6}$$

The utilization of this resource is

$$u_{i,m} = \frac{\Gamma_m}{p_i}. \tag{4.7}$$

For a resource R_n such as the bus, its total access time is increased by reading or writing a cache-line at cache misses. So this time is calculated as follows:

$$\Gamma_n = \lambda_n \cdot N_{size} + p_{dynamic}. \tag{4.8}$$

The resource utilization is given by:

$$u_{i,n} = \frac{\Gamma_n}{p_i}. \tag{4.9}$$

Similar calculation holds for the resource utilization of instruction or data memory.

4.3.3 A Versatile Tracing and Profiling Tool

The trace and profile tool-chain as described in Figure 4.5 is not limited to providing the timing characteristics of the driver functions. Instead, this tool has been made very versatile in its features. It can provide non-intrusive, multi-level tracing and profiling results in ISS-based software simulation. The comprehensive results may be helpful in expediting the design process. For example, a traced dynamic function call graph can identify hot-spot in the subroutines; memory access pattern can aid the decision of cache configuration; and the resource contention tracing can be used to select the arbitration policy. Details and results regarding the features and usage of this tool are given in Appendix B.

4.4 Calculation of Resource Availability

Resource availability aims to provide a metric that measures how much a resource is accessible within a period of time. Its value is not explicitly simulated but rather approximated using other timing parameters. In this work, the calculation of resource availability depends on the resource utilization of ongoing abstract timing requests. Further, several other aspects also exert influence on resource availability, such as arbitration policy, interrupt handling and bus protocols. They are also considered in the calculation.

4.4.1 Arbitration Policy with Preemptive Fixed Priorities

Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be issued by initiator 1 and 2 respectively. Assume initiator 1 has higher priority than initiator 2, then the resource availability at R_m for $\mathcal{S}_1$ and $\mathcal{S}_2$ are calculated as:

$$a_{1,m} = 1; \qquad a_{2,m} = 1 - u_{1,m}. \tag{4.10}$$

For $\mathcal{S}_1$, resource R_m is fully available. For $\mathcal{S}_2$, resource R_m is less available if $\mathcal{S}_1$ demands more resource utilization, indicating more delay to $\mathcal{S}_2$ at R_m as shall be seen later.

Now suppose a third initiator with lower priority issues a timing synchronization request for $\mathcal{S}_3$. To calculate its resource availability, the combined resource utilization $u_{(1,2),m}$ of R_m demanded by initiator 1 and initiator 2 is computed:

$$u_{(1,2),m} = u_{1,m} + u'_{2,m}, \tag{4.11}$$

where $u_{1,m}$ remains as before and $u'_{2,m}$ is a modified resource utilization of $\mathcal{S}_2$ considering its prolonged duration. The calculation of $u'_{2,m}$ and $u_{(1,2),m}$ makes use of the delay formula. Therefore it is given later in (4.18) after the introduction of the delay formula in Section 4.5. The resource availability at R_m for $\mathcal{S}_3$ is $a_{3,m} = 1 - u_{(1,2),m}$, which is then substituted in (4.14) for delay estimation.

4.4.2 Arbitration Policy with FIFO Arbitration Scheme

This section considers the first-in-first-out (FIFO) arbitration policy. Assume initiator 1 and 2 have the same priority. For FIFO arbitration scheme, the transactions of initiator 1 may delay or may be delayed by the transactions of initiator 2. For a fixed resource utilization of $\mathcal{S}_2$ at R_m, the more $\mathcal{S}_1$ accesses R_m, the more often its transactions will win in the arbitration. In accordance with this consideration, the resource availability of $\mathcal{S}_1$ and $\mathcal{S}_2$ at R_m is symmetrically derived as:

$$\begin{aligned} a_{1,m} &= (1 - u_{2,m}) + \frac{u_{1,m}}{u_{1,m} + u_{2,m}} \cdot u_{2,m} \\ a_{2,m} &= (1 - u_{1,m}) + \frac{u_{2,m}}{u_{2,m} + u_{1,m}} \cdot u_{1,m} \end{aligned} \tag{4.12}$$

In the above equation, $\frac{u_{1,m}}{u_{1,m}+u_{2,m}}$ can be thought of as a weight that measures how often the access to R_m is granted to $\mathcal{S}_1$ against $\mathcal{S}_2$. For example, assume $u_{1,m} = 0.6$ and $u_{2,m} = 0.3$, it follows $a_{1,m} = 1 - 0.3 + \frac{0.6}{0.6+0.3} \cdot 0.3 = 0.9$.

If there are three initiators with the same priority, the resource availability for their abstract timing requests can be symmetrically calculated in a similar way:

$$\begin{aligned} a_{1,m} &= 1 - u_{(2,3),m} + \frac{u_{1,m}}{u_{1,m} + u_{2,m} + u_{3,m}} \cdot u_{(2,3),m} \\ a_{2,m} &= 1 - u_{(1,3),m} + \frac{u_{2,m}}{u_{1,m} + u_{2,m} + u_{3,m}} \cdot u_{(1,3),m} \\ a_{3,m} &= 1 - u_{(1,2),m} + \frac{u_{3,m}}{u_{1,m} + u_{2,m} + u_{3,m}} \cdot u_{(1,2),m}, \end{aligned} \tag{4.13}$$

where $u_{(i,j),m} = min(u_{i,m} + u_{j,m}, 1)$ is the combined resource utilization of R_m for $\mathcal{S}_i$ and $\mathcal{S}_j$. For example, assume $u_{1,m} = 0.5, u_{2,m} = 0.3, u_{3,m} = 0.2$, then $a_{1,m} = 1 - (0.2 + 0.3) + \frac{0.5}{0.5+0.2+0.3} \cdot (0.3 + 0.2) = 0.75$.

4.4.3 Generalization of the Model

Analytical modeling techniques offer the advantage that the formulas can be modified or generalized to incorporate the modeling of different aspects of the system. For the present model, this can be achieved by reasoning about the influence of the considered aspect on the resource utilization and resource availability. Section 4.4.1 and Section 4.4.2

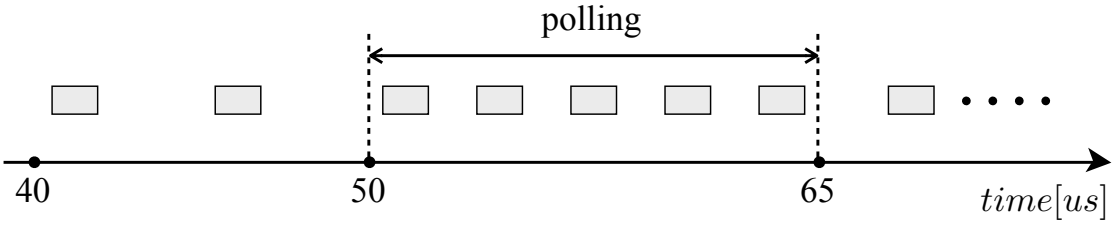


FIGURE 4.6: Handle ISR in calculating resource availability.

have demonstrated the modeling of arbitration policies by using corresponding formulas. This section further gives another two examples regarding the generalization of the present analytical model in handling particular cases in practice.

4.4.3.1 Consideration of Register Polling

In transferring one data unit of a data block, a driver function may need to use an interrupt service routine (ISR). If a polling policy is implemented in handling the data communication with peripheral devices, the ISR keeps invoking transactions to check the status register of the device. After the ready state is set in the status register, the transfer of the next data unit can be initiated and the polling is carried out in a similar way. The duration it takes for the device to become ready upon receiving the data depends on the data processing algorithm implemented on the device. This duration is independent of the ISR. Therefore even the transactions for register polling are delayed, the duration of the ISR may be unaffected. For example, assume the 3rd to 7th transactions in Figure 4.6 correspond to register polling. Also, assume the HW device being polled will set its status register to ready at $65us$. Even if the polling transactions are delayed by the transactions of other higher priority initiators, the polling in the ISR will still end at $65us$.

Based on these considerations, the calculation of resource utilization and availability should be adjusted accordingly. For a lower priority initiator, the transactions related to register polling do not count when calculating its resource utilization, thus not contributing to the delay. For a higher priority initiator, the transactions related to register polling within are considered as its resource utilization when calculating the resource availability and delay of other abstract timing requests issued by lower priority initiators.

4.4.4 Consideration of Bus Protocols

In most common cases, the system bus is a shared resource. When advanced bus protocols are used, the calculation of resource utilization needs to be adjusted. For example, according to AMBA AHB bus protocols, slow and split-capable slaves may split the transactions and thus free the bus. Further, an initiator can lock the bus and thus can not be preempted after it has been granted to use the bus. Also, address and data phases are pipelined in AHB protocols. Preemption of lower priority initiator happens when the data is ready for the current beat of its burst transaction.

To handle these protocols, it needs to be distinguished how the resource utilization and resource availability are affected by one protocol. In case of split transactions, the split slots are not considered when calculating the bus utilization of higher priority initiators.

As for bus-locking transactions of a lower priority initiator, the resource availability for a higher priority initiator needs to be reduced by the resource utilization due to those bus-locking transactions. Similarly, this availability needs also to be reduced by the average percentage of a beat in the resource utilization of a lower priority initiator.

4.5 The Delay Formula

Due to timing conflicts at shared resources, the actual duration of a abstract timing request is usually longer than its optimal duration. To model this, a formula is proposed for estimating the delay given the resource utilization and the resource availability.

Before deriving the formula, the influences of resource utilization and availability on the delay are reasoned about. Roughly speaking, for a fixed resource availability less than 1, the delay increases if the resource utilization increases, since more resource conflicts may occur. For a fixed resource utilization larger than 0, the delay increases if the resource availability decreases. More specifically, it is hypothesized in deriving the delay formula that the delay increases linearly with respect to the resource utilization, and hyperbolically with respect to the resource availability.

To model the above two relations, let's consider the delay $d_{i,m}$ at a resource R_m for a timing request $\mathcal{S}_i$. The delay formula is expressed as:

$$\begin{aligned} d_{i,m} &= \phi(p_i, u_{i,m}, a_{i,m}) \\ &= \frac{1}{a_{i,m}} \cdot u_{i,m} \cdot p_i - u_{i,m} \cdot p_i \\ &= (\frac{1}{a_{i,m}} - 1) \cdot u_{i,m} \cdot p_i \end{aligned} \tag{4.14}$$

In the above formula, $u_{i,m} \cdot p_i$ can be thought of as the total access time of resource R_m required by $\mathcal{S}_i$ within its required duration p_i. Since the resource R_m is not fully available ($a_{i,m} < 1$), the actual access time of this resource time is prolonged to $\frac{1}{a_{i,m}} \cdot u_{i,m} \cdot p_i$. The delay at this resource is therefore given by the difference between the prolonged access time and the optimal access time. As can be seen from the formula, the delay is linear to the resource utilization and hyperbolic to the resource availability, which complies with the relations that have previously reasoned about. The overall delay for $\mathcal{S}_i$ is

$$d_i = \sum_{i=1}^{r} d_{i,m}, \tag{4.15}$$

with r being the number of shared resources. This delay formula is simple and yet efficient in terms of complexity and accuracy, as will be demonstrated in the experiments.

Consider a simple example: let $p_i = 1ms, u_{i,m} = 0.3$, and $a_{i,m} = 0.6$, then it follows:

$d_{i,m} = (\frac{1}{0.6} - 1) \cdot 0.3 \cdot 1ms = \frac{0.4}{0.6} \cdot 0.3 \cdot 1ms = 0.2ms.$

Assume R_m is the only shared resource, then $d_i = 0.2ms$. Therefore the duration of $\mathcal{S}_i$ will be prolonged to $p_i + d_i = 1ms + 0.2ms = 1.2ms$.

For an arbitration scheme with fixed priorities, assume initiator 2 has lower priority than initiator 1. Substituting the resource availability $a_{2,m}$ from (4.10) into the delay

formula gives

$$\begin{aligned} d_{2,m} &= (\frac{1}{a_{2,m}} - 1) \cdot u_{2,m} \cdot p_2 \\ &= \frac{u_{1,m}}{1 - u_{1,m}} \cdot u_{2,m} \cdot p_2 \end{aligned} \tag{4.16}$$

It can be seen that, for a larger resource utilization $u_{1,m}$ demanded by the higher priority initiator, delay $d_{2,m}$ becomes larger as expected.

Suppose there exists another request from $\mathcal{S}_3$ which has the lowest priority. To estimate its delay, the combined resource utilization for $\mathcal{S}_1$ and $\mathcal{S}_2$ at resource R_m is calculated as

$$u_{(1,2),m} = u_{1,m} + u_{2,m} \cdot \frac{p_2}{p_2 + d_2} \tag{4.17}$$

The second term in the right side of the above equation represents the resource utilization of $\mathcal{S}_2$ seen by $\mathcal{S}_3$. It is less than $u_{2,m}$, since certain resource accesses of $\mathcal{S}_2$ are blocked by those of $\mathcal{S}_1$, leading to a reduced resource utilization of $\mathcal{S}_2$ experienced by $\mathcal{S}_3$. Assume R_m is the only shared resource, thus $d_2 = d_{2,m}$. Substituting (4.16) in (4.17) gives

$$\begin{aligned} u_{(1,2),m} &= u_{1,m} + u_{2,m} \cdot \frac{p_2}{p_2 + d_2} \\ &= u_{1,m} + \frac{u_{2,m} \cdot p_2}{p_2 + \frac{u_{1,m}}{1-u_{1,m}} \cdot u_{2,m} \cdot p_2} \\ &= u_{1,m} + \frac{1}{\frac{1}{u_{2,m}} + \frac{u_{1,m}}{1-u_{1,m}}} \\ &\leq u_{1,m} + \frac{1}{1 + \frac{u_{1,m}}{1-u_{1,m}}} = 1 \end{aligned} \tag{4.18}$$

Similarly, for FIFO arbitration policy, substitute the resource availability from (4.12) into (4.14) for delay estimation. For example, assume $u_{1,m} = u_{2,m} = 0.4$, therefore $a_{1,m} = 0.8$ according to (4.12). Therefore, based on (4.14), it follows

$$d_{1,m} = (\frac{1}{0.8} - 1) \cdot 0.4 \cdot p_1 = 0.1 \cdot p_1.$$

To handle other specific traffic distributions or specific bus protocols, corresponding adjustments can be made in the calculation of the resource utilization or availability. The same delay formula as in (4.14) can still be applied.

4.6 Incorporate Analytical Timing Estimation in Simulation

The next step is to integrate the present analytical timing estimation method into a simulation framework for transaction-level models. Realizing this integration raises several requirements. First, the simulation overhead of applying the timing estimation should be low. This requirement is met by an efficient scheduling algorithm in Section 4.6.1. Second, the implementation should provide flexible usability. This is achieved by using a stand-alone *resource model* library, which handles the non-functional aspects such as

timing, making timing estimation transparent to functional simulation. In addition, this library is compatible to the SystemC and TLM standards, therefore it can be easily ported to existing models. Details are described in 4.6.2.

4.6.1 The Scheduling Algorithm

The scheduling algorithm is implemented by a central scheduler, which is contained in the resource model layer [14]. It is a central scheduler in the sense that the utilizations of all resources and all ongoing abstract timing requests are visible to the scheduler. Since one abstract timing request involves not just one but multiple resources, a central scheduler is therefore a more efficient and straightforward implementation than distributed schedulers. The task of this central scheduler is to ensure that the abstract timing requests finish at the correct time. Each time the synchronization of a new abstract timing request is issued, the scheduler is called to estimate its duration and adjust the durations of other ongoing abstract timing requests, by calculating the delay due to the resource conflicts among them. For the description of the algorithm, the following term is introduced:

Synchronization point: it refers to the time when the scheduler is called to perform the scheduling algorithm.

A synchronization point occurs each time the communication scenario is changed, i.e. when a new abstract timing request is issued. To describe the principle of the scheduling algorithm, recall the problem to be solved in (4.2) of Section 4.2.2:

$$\mathbf{d} = \Phi(\mathbf{p}, \mathbf{U}, \mathbf{A}).$$

It is this task that the scheduling algorithm needs to perform. In doing so, the scheduler performs two main sub-tasks.

1. **Timing parameter update:** At each synchronization point, the timing parameters of the ongoing abstract timing requests need to be updated before performing timing estimation. Updated timing parameters include the remaining durations and the resource utilization matrix $\mathbf{U}$. The calculation takes into account how much time has elapsed since the last synchronization point. Subsequently, the resource availability matrix $\mathbf{A}$ is recalculated, by using the formulas in Section 4.4.
2. **Delay estimation and rescheduling:** New delay and duration are estimated for each abstract timing request, by substituting the updated resource utilization and availability in the delay formula. Finally, the abstract timing requests are *dynamically rescheduled*, so that they will terminate at the correct time.

To illustrate the timing dynamics in the scheduling process, consider a hypothesized system with two initiators. Assume that a fixed priority arbitration scheme is used, with initiator 1 having higher priority than initiator 2. The communication scenarios under consideration are shown in Figure 4.7. In the following, the steps performed by the scheduling algorithm at each synchronization point are described.

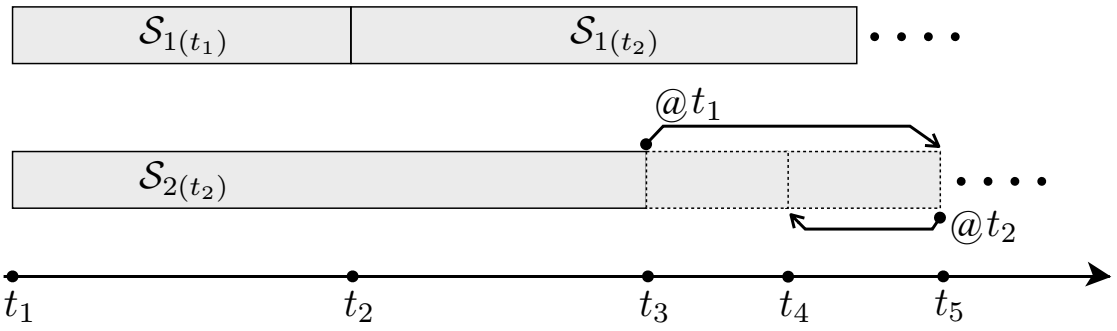


FIGURE 4.7: Example of the dynamic rescheduling feature.

- At time t_1, two abstract timing requests $\mathcal{S}_{1(t_1)}$ and $\mathcal{S}_{2(t_1)}$ are initiated by initiator 1 and 2 respectively. $\mathcal{S}_{1(t_1)}$ requires a duration p_1, with $p_1 = (t_2 - t_1)$. $\mathcal{S}_{2(t_1)}$ requires a duration p_2, with $p_2 = (t_3 - t_1)$. Suppose initiator 1 calls the scheduling algorithm first to schedule $\mathcal{S}_{1(t_1)}$. Because of its higher priority, the end time of $\mathcal{S}_{1(t_1)}$ is scheduled to t_2, i.e. an optimal duration with no delay. Next, initiator 2 calls the scheduling algorithm to schedule $\mathcal{S}_{2(t_1)}$. At this time, $\mathbf{S}_{(t_1)} = \{\mathcal{S}_{1(t_1)}, \mathcal{S}_{2(t_1)}\}$. For illustrative purpose, assume there are conflicts at two shared resources and the utilization matrix reads as follows:

$$\mathbf{U} = \begin{array}{c} \begin{matrix} \mathbf{u}_1 & \mathbf{u}_2 \end{matrix} \\ \begin{bmatrix} 0.4 & 0.2 \\ 0.3 & 0.5 \end{bmatrix} \end{array} \tag{4.19}$$

where $\mathbf{u}_1 = [0.4, 0.3]^T$ and $\mathbf{u}_2 = [0.2, 0.5]^T$ are the resource utilization of $\mathcal{S}_{1(t_1)}$ and $\mathcal{S}_{1(t_1)}$ respectively. Substituting (4.19) in the formula (4.10) gives the resource availability matrix:

$$\mathbf{A} = \begin{array}{c} \begin{matrix} \mathbf{a}_1 & \mathbf{a}_2 \end{matrix} \\ \begin{bmatrix} 1 & 0.6 \\ 1 & 0.7 \end{bmatrix} \end{array} \tag{4.20}$$

For $\mathcal{S}_{2(t_1)}$, its delays at each resource can be obtained using the formula (4.14):

$$\begin{bmatrix} d_{2,1} \\ d_{2,2} \end{bmatrix} = \begin{bmatrix} \phi(p_2, u_{2,1}, a_{2,1}) \\ \phi(p_2, u_{2,2}, a_{2,2}) \end{bmatrix} \approx \begin{bmatrix} 0.13 \cdot p_2 \\ 0.21 \cdot p_2 \end{bmatrix} \tag{4.21}$$

Here $d_{2,1} = \phi(p_2, u_{2,1}, a_{2,1}) = (\frac{1}{a_{2,1}} - 1) \cdot u_{2,1} \cdot p_2 \approx 0.13 \cdot p_2$ is the delay of $\mathcal{S}_{2(t_1)}$ at resource 1 and $d_{2,2} = 0.21 \cdot p_2$ is the delay at resource 2. The total delay $d_2 = d_{2,1} + d_{2,2} \approx 0.34 \cdot p_2$. Therefore, the end time of $\mathcal{S}_{2(t_1)}$ is scheduled to t_5, with $t_5 \approx t_1 + p_2 + 0.34 \cdot p_2$.

- At time t_2, the end event of the abstract timing request $\mathcal{S}_{1(t_1)}$ expires. Then initiator 1 resumes its execution. It executes the next piece of code in a temporally decoupled way, after which it issues a new abstract timing request $\mathcal{S}_{1(t_2)}$ for synchronization. Now the scheduling algorithm is called again, with $\mathbf{S}_{(t_2)} = \{\mathcal{S}_{1(t_2)}, \mathcal{S}_{2(t_1)}\}$. Assuming the resource utilization of $\mathcal{S}_{1(t_2)}$ is $\mathbf{u}_1 = [0.2, 0.1]^T$, the utilization matrix

at t_2 is:

$$\mathbf{U} = \begin{array}{c} \begin{matrix} \mathbf{u}_1 & \mathbf{u}_2 \end{matrix} \\ \begin{bmatrix} 0.2 & 0.2 \\ 0.1 & 0.5 \end{bmatrix} \end{array} \tag{4.22}$$

Update the resource availability using (4.22) and (4.10):

$$\mathbf{A} = \begin{array}{c} \begin{matrix} \mathbf{a}_1 & \mathbf{a}_2 \end{matrix} \\ \begin{bmatrix} 1 & 0.8 \\ 1 & 0.9 \end{bmatrix} \end{array} \tag{4.23}$$

Besides, the remaining duration of $\mathcal{S}_{2(t_1)}$ also needs to be updated as well:

$$p_2 \leftarrow \frac{t_5 - t_2}{t_5 - t_1} \cdot p_2,$$

where t_2 and t_1 represent the current and previous synchronization points respectively, t_5 represents the currently scheduled end time. The interpretation of the update of the remaining duration is that, it finds out how much the duration of this request has been effectively conducted from the previous to the current synchronization point, thus only taking into account the remaining duration in the following calculation. Now, similar as in (4.21), the delay for $\mathcal{S}_{2(t_1)}$ is recalculated as $[d_{2,1}, d_{2,2}]^T = [0.05 \cdot p_2,\ 0.06 \cdot p_2]^T$, resulting in $d_2 = 0.11 \cdot p_2$. Therefore the end event of $\mathcal{S}_{2(t_1)}$ is scheduled at t_4, with $t_4 = t_2 + p_2 + 0.11 \cdot p_2$.

It is worth pointing out that the end event of a abstract timing request can be rescheduled to an earlier time, if the resource utilization of other abstract timing requests decreases, such as in the illustrated case.

- At time t_4, the end event of $\mathcal{S}_{2(t_1)}$ expires. Simulation of the process on initiator 2 resumes, leading to a new abstract timing request $\mathcal{S}_{2(t_4)}$. Afterwards the scheduling algorithm will run again to determine the timing of this and other ongoing abstract timing requests in a similar manner as discussed above.

  No synchronization is performed at time t_3 or t_5. Time t_3 is the optimal end time for $\mathcal{S}_{2(t_1)}$. The time difference $t_4 - t_3$ measures the final delay for $\mathcal{S}_{2(t_1)}$. Time t_5 is the estimated expire time for the end event of $\mathcal{S}_{2(t_1)}$. It is set at t_1 but canceled at t_2.

Formally, the scheduling algorithm is outlined in Algorithm 4.1. It is called each time a new abstract timing request needs to be scheduled. The task for updating on-going abstract timing requests is implemented in lines 6 to 15.

Firstly, the resource utilization matrix is updated. The remaining durations of all on-going abstract timing requests are re-estimated.

Secondly, using the updated resource utilization, the schedule recalculates the resource availability, according to the method in Sec 4.4.

The task for delay and duration estimation is implemented in lines 17 to 27. Using the just updated parameters, it estimates the delay (line 21) according to the delay formula described in Sec 4.5.

Additionally, there is a SystemC event associated with an abstract timing request. This event represents the time that the abstract timing request terminates. It is dynamically canceled and re-notified (line 25 and 26), effectively adjusting the actual duration.

Finally, a *wait* statement is called to wait until the end event of this abstract timing request expires. This algorithm needs to be performed only once to schedule one abstract timing request. Notably, no matter how many times the re-scheduling is performed, only one *wait* statement is needed to schedule one abstract timing request due to the use of dynamic event cancellation. Because the time-consuming *wait* statement is used only once, the implemented algorithm imposes very low computation overhead, thus guaranteeing the efficiency of the scheduling algorithm.

Algorithm 4.1 The Scheduling Algorithm

1: // called to schedule a new abstract timing request
2: $\mathcal{S}_{new}$:= the newly issued abstract timing request
3: $\mathbf{S}_{(t)}$:= ongoing abstract timing requests seen at time t
4: $\mathcal{S}_i.endEvent$:= a SystemC event associated with $\mathcal{S}_i$
5:
6: // update ongoing abstract timing requests
7: **for** $\mathcal{S}_i \in \mathbf{S}_{(t_1)}$ **do**
8: $p_i \leftarrow$ UpdateDuration($\mathcal{S}_i$)
9: $\mathbf{u}_i \leftarrow$ UpdateResourceUtilization($\mathcal{S}_i$)
10: **end for**
11: **for** $\mathcal{S}_i \in \mathbf{S}_{(t)}$ **do**
12: **for** $R_m \in \mathbf{R}$ **do**
13: $a_{i,m} \leftarrow$ GetAvailability($\mathbf{U}, i, m$)
14: **end for**
15: **end for**
16:
17: // timing estimation, resolve conflicts
18: **for** $\mathcal{S}_i \in \mathbf{S}$ **do**
19: $p_i :=$ remaining duration of $\mathcal{S}_i$
20: **for** $R_m \in \mathbf{R}$ **do**
21: $d_{i,m} :=$ GetDelay($p_i, u_{i,m}, a_{i,m}$) // as in (4.14)
22: **end for**
23: $D_i \leftarrow \sum_{m=1}^{|\mathbf{R}|} d_{i,m}$
24: $p_i \leftarrow p_i + D_i$
25: $\mathcal{S}_i.endEvent$.cancel() // dynamic re-scheduling
26: $\mathcal{S}_i.endEvent$.notify(p_i)
27: **end for**
28:
29: // the single call to the expensive *wait()* statement
30: **wait**($\mathcal{S}_{new}.endEvent$)
31:
32: // finally $\mathcal{S}_{new}.endEvent$ expires.
33: **return**

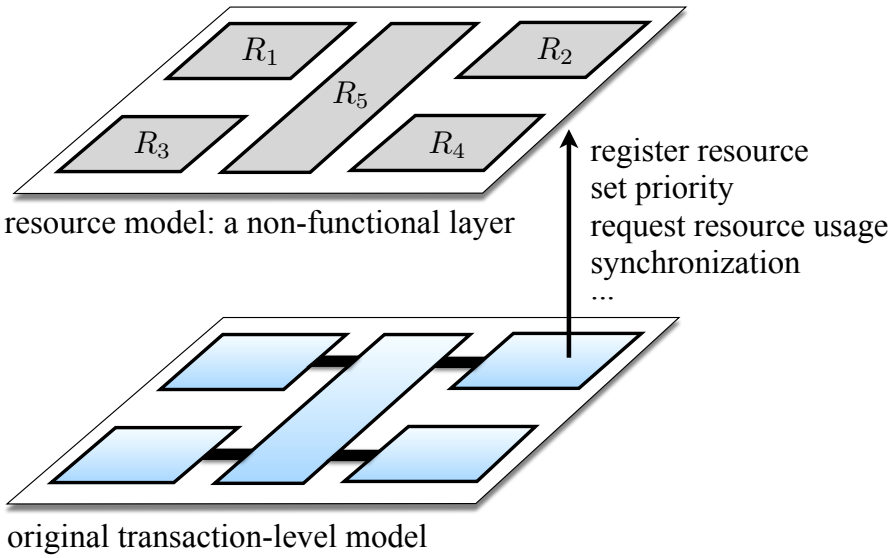


FIGURE 4.8: A resource model library ported to TLMs as a non-functional layer for timing estimation.

4.6.2 Modeling Support - Integrating the Resource Model

The central scheduling algorithm is implemented in a *resource model* layer. The resource model encapsulates the handling of non-functional properties such as timing. In this sense, it can be viewed as an additional timing layer parallel to the functional simulation of the original transaction-level model (see Figure 4.8). The resource model layer is coded into a stand-alone library, thus it is easily portable to existing transactions-level models. Besides, it is compatible to the SystemC standards and does not require any modification to the SystemC kernel or TLM library.

Integrating the resource model to existing transaction-level models requires fairly low coding effort. Each system module simply derives itself from a base resource class `rm_resource`, as in List 4.4. Then it is able to call a set of functions provided by the resource model library. At instantiation, the base resource class registers itself to the central resource layer, receives a resource id, and sets its priority if needed. If the system module is an initiator, it additionally owns an instance of an abstract timing request, for which the timing parameters need to be set for synchronization purpose. Now, the simulation can be carried out with clear separation of the functional and timing aspects. In the functional simulation, an initiator can execute a large code block of its thread without calling the *wait* statements, e.g. in line 5 of Listing 4.4. During the thread execution, the local time and the access time of the resources can either be accumulated (Section 4.3.1) or queried from a timing profile library (Section 4.3.2.1). Then, for the timing simulation, the initiator issues a synchronization request to the resource model, e.g. in line 8 of Listing 4.4. The resource model maintains a set of the ongoing abstract timing requests. Upon receiving a synchronization request, it calls the scheduling algorithm described in Algorithm 4.1, which dynamically determines and adjusts the actual durations of all ongoing abstract timing requests. When to issue the synchronization request can be determined by the programmer. For example, synchronization points can be set before and after the execution of certain functions.

```
1  class ModuleX: public sc_module, public rm_resource
2    ...
```

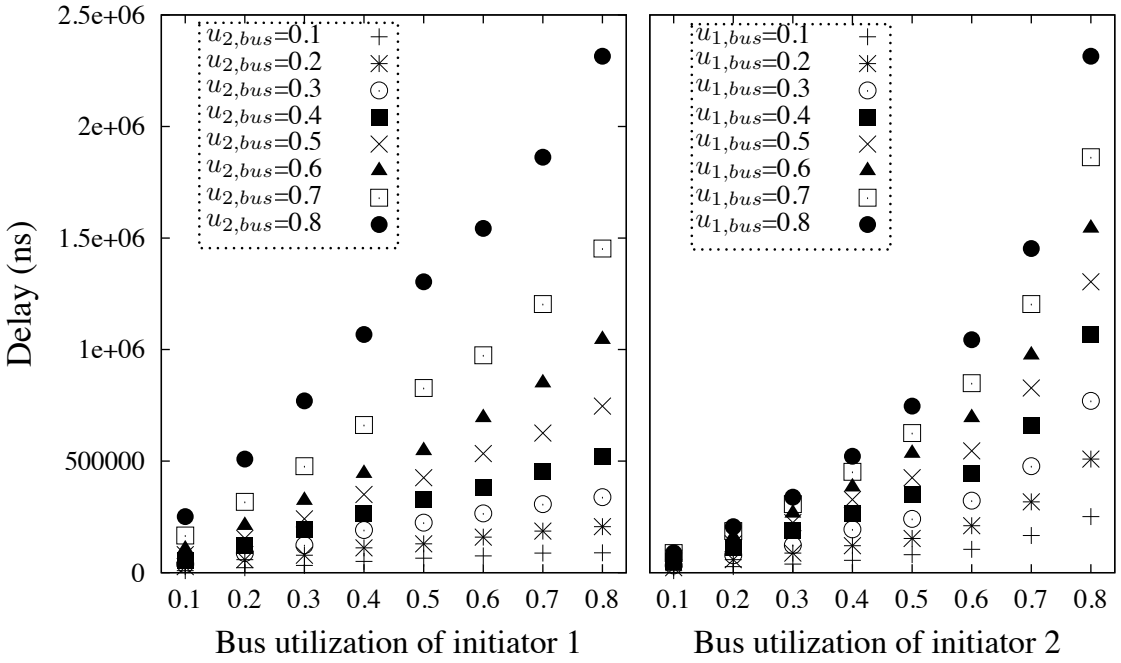


FIGURE 4.9: Delay of the synchronization request of processor 1 with respect to different resource usage scenarios

```
3   void thread_1{
4     ...
5     functionA(); //functional simulation
6     rm_useResource(bus_id, sumBusAccessTime);
7     ...
8     rm_sync(duration); //timing
9     ...
```

LISTING 4.4: Porting the resource model to existing transactions-level models.

4.6.3 Comparison with TLM2.0 Quantum Mechanism:

The TLM2.0 standard also has a built-in support for using temporal decoupling. For this, a global quantum needs to be predefined. Usually, the predefined value of the global quantum is quite large, when compared with the clock period. During the simulation, synchronization between different processes is performed at the granularity of the global quantum. That means a process keeps its execution until its local time exceeds the global quantum. Only then the context-switching is performed so that another process can resume its execution. In contrast to the quantum mechanism, the present approach supports *on-demand* synchronization, meaning that one process can call the scheduling algorithm only when it needs to. It does not require a pre-determined granularity of synchronization. Such on-demand synchronization offers more modeling flexibility and simulation efficiency.

4.7 Experimental Results

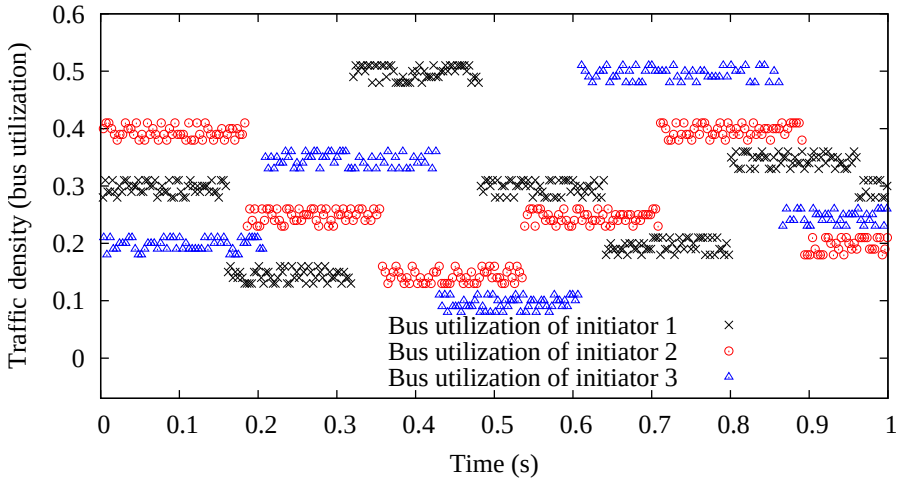
4.7.1 RTL Simulation as a Proof of Concept

RTL models are usually clocked, thus temporal decoupling is not used. Nevertheless, cycle accurate RTL simulation is used here to demonstrate the influence of resource utilization and availability on the delay. Two initiators are connected to an AMBA AHB bus. They use a random traffic generator to transfer data over the bus. Preemptive arbitration is used, where initiator 2 has higher priority.

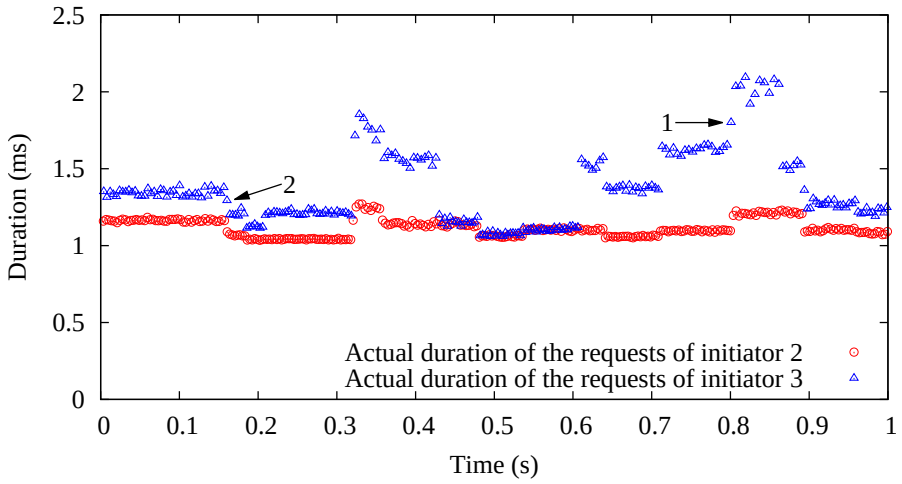
With no resource conflicts, the number of transactions sent by initiator 1 within $1ms$ is approximately constant for a given traffic density. With resource conflicts, the delay is measured as the additional time for initiator 1 to finish the same number of transactions. This measurement is conducted in various traffic scenarios, by varying the bus utilization of each initiator. As shown in Figure 4.9(a), for a fixed bus utilization of initiator 2, the delay is approximately linear to the bus utilization of initiator 1. As shown in Figure 4.9(b), for a fixed bus utilization of initiator 1, the delay is approximately hyperbolic to the bus utilization of initiator 2. Both the linear and hyperbolic curvatures conform to the delay model as in (4.10)and (4.14). The results therefore validate the adoption of the analytical delay estimation.

4.7.2 Hypothetical Scenarios

In this experiment, hypothetical scenarios of different resource conflicts are simulated. The simulation results are used as illustrations of the fundamental principles of the proposed delay estimation approach and the scheduling algorithm. Three initiators are connected to a shared resource that uses fixed priority arbitration with a priority configuration: initiator 1 > initiator 2 > initiator 3. Concurrently, each initiator issues a synchronization request with a requested duration of $1ms$. The resource utilization of these requests varies around a pre-defined value within certain time periods. For example, within the time interval $[0s, 0.1s]$ in Figure 4.10(a), the resource utilization for each synchronization request of initiator 2 is around 40%. For the lower priority initiator 2 and initiator 3, their synchronization requests will be delayed. Their actual durations are plotted in Figure 4.10(b), which is obtained by applying the delay formula in the scheduler. It can be seen that, for a lower priority initiator, its synchronization request is delayed more if the resource utilization of this initiator or another higher priority initiator increases. Moreover, this figure also shows that the duration is adjusted dynamically. Take initiator 3's the synchronization request pointed by the arrow for example. At first, it overlaps with initiator 1's synchronization request with a resource utilization around 0.2. Before it finishes, the next synchronization request of initiator 1 begins with a resource utilization around 0.35. Because each synchronization request will cause the scheduling algorithm in Algorithm 4.1 to update and reschedule all ongoing requests, the request of initiator 3 is therefore rescheduled to a later time, indicating more delay. Details regarding such dynamic adjustment have been given in Section 4.6.1. Similar observation can also be made for the synchronization request pointed by arrow 2.



(a) Different resource utilization scenarios.



(b) Actual durations of the synchronization requests.

FIGURE 4.10: Estimated timing in hypothetical scenarios.

4.7.3 Applied To HW/SW Co-Simulation

This experiment demonstrates how the analytical timing estimation can be used in a co-simulation environment, with host-compiled software simulation on a multiprocessor TLM. The architecture of the employed TLM virtual prototype is sketched in Figure 4.11.

4.7.3.1 Description of the SW Simulation

Section 2.1 has already briefly listed the considered cases of software simulation in the scope of this work. For the sake of clarity, this section gives a more detailed description regarding how the software and its performance are simulated in the present approach, before presenting the experimental results.

There are two types of temporal decoupling applied in the software simulation. The first type corresponds to the simulation of application software. In the domain of host-compiled SW simulation, previous approaches use bus-word transactions to simulate the

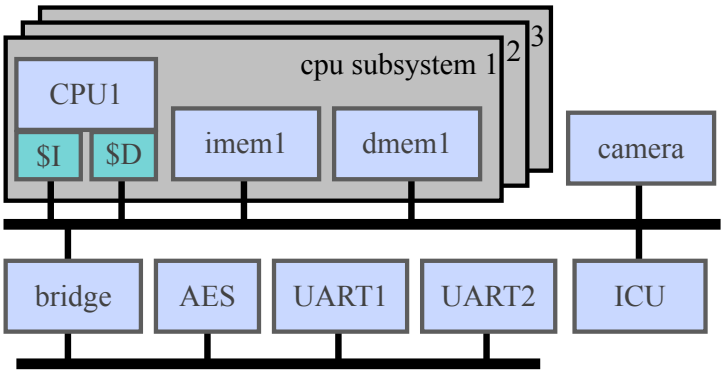


FIGURE 4.11: VP architecture modeled with TLM.

memory accesses after cache misses or the accesses to other hardware modules. An example is given in Figure 4.12(a). As can be seen, the expensive *wait* statement is called to synchronize the estimated cycles before inter-module communication, such as cache line refilling or accesses to peripheral devices. Then the transactions are evoked. Arbitration can be performed for each bus-word transaction to resolve timing conflicts. Such fine-grained synchronization may incur large simulation overhead, in case of frequent cache misses or peripheral accesses. For this reason, it is necessary to simulate the SW in a temporally decoupled way. As shown in Figure 4.12(b), neither *wait* statements nor transactions take place at cache misses. Leaving out these transactions will not alter the functional correctness of the simulation, since they do not carry functional data as already explained in Section 1.2.3.4. The number of cache misses is accumulated. In the present approach, the requested duration of a large code block is calculated by summing the accumulated cycles and the cache miss penalty. The accumulated cache miss counts are also used to calculate the utilization of a resource such as the bus. Then, with these timing parameters, a synchronization request is issued to the scheduler, which performs the proposed scheduling algorithm described in Section 4.6.1 to determine the delay and hence the actual duration.

The second type corresponds to the usage of TLM+ transactions, which implicitly apply temporal decoupling. The requested durations of these TLM+ transactions can be obtained by using their timing profiles, which are constructed according to the method in Section 4.3.2.1. In simulation, after a TLM+ transaction is carried out, a timing synchronization request will be issued to the scheduler. The scheduler estimates its actual duration by taking into account the resource utilization of other ongoing synchronization requests as described in Section 4.6.1.

In practice, the above two cases can happen in one software program. For example, the program calls a function to encrypt a data block and then transfers the encrypted data to a UART module. Firstly, the encryption function is simulated with temporal decoupling. Then, the duration of this encryption function is estimated by issuing a timing synchronization request to the scheduler. Secondly, the data transfer is simulated by a TLM+ transaction, which is timed by a subsequent timing synchronization request issued to the scheduler as well.

In the following experiments, different communication scenarios and priority schemes are tested. From the experimental data, timing accuracy and simulation speed-up are examined to evaluate the efficacy of the proposed approach. In particular, it will be shown that (i) the analytical timing estimation can ensure high timing accuracy when

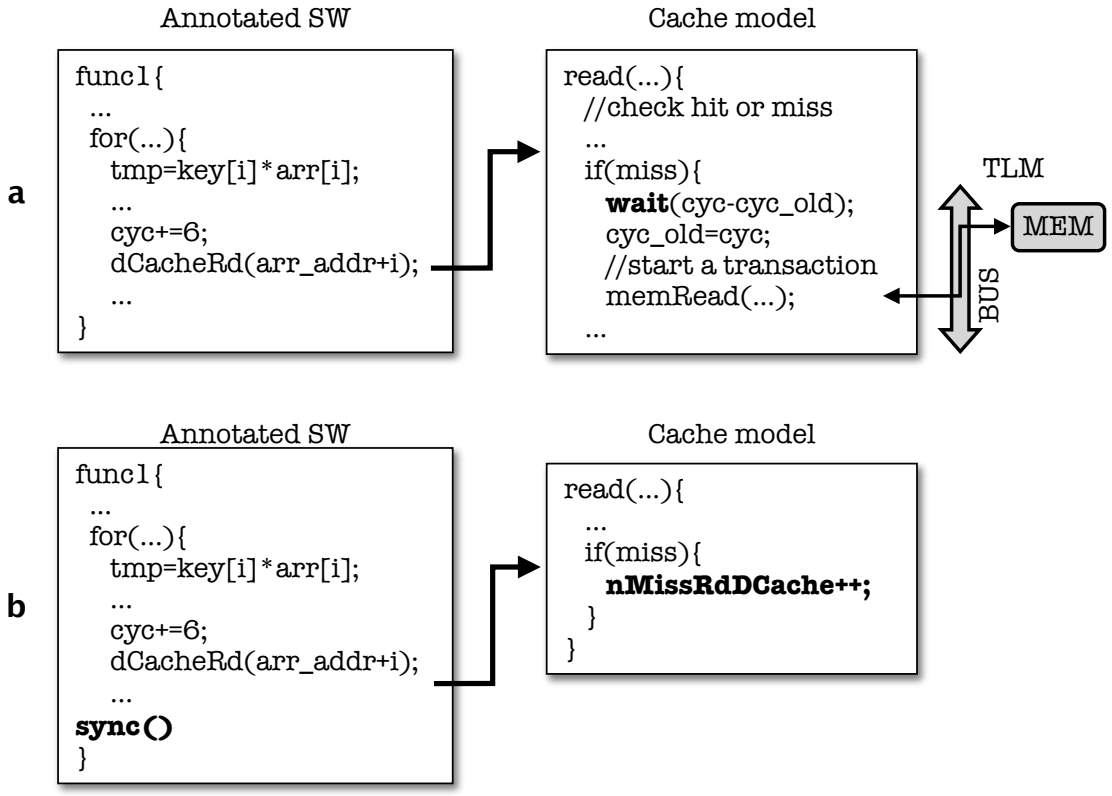


FIGURE 4.12: Host-compiled SW simulation (a) conventional simulation using standard TLM without using temporal decoupling (b) modified for fast simulation with analytical timing estimation.

temporal decoupling is used and (ii) it introduces very low simulation overhead thus preserving the high speed-up.

4.7.3.2 Simulation of Two Processors

The experiment is set up as such: processor 1 filters a new buffer by using the *fir* algorithm, and then writes the results to the *UART* module, so on and so forth. Concurrently, processor 2 reads a frame from the camera, performs color conversion (*rgb2yuv*) on each pixel, and then continues with the next frame. A fixed priority scheme is used, with processor 2 having a higher priority than processor 1. The simulation is performed in 3 modes: (1) In TLM mode, the conventional host-compiled simulation is used, as in Figure 4.12(a). (2) In TLM+TD mode, temporal decoupling is used as in Figure 4.12(b). But there is no consideration of the delay due to the access conflicts at shared resources. (3) In TLM+TD+DM mode, temporal decoupling is used. The proposed delay model (DM) and scheduling algorithm are applied for timing estimation.

For in-depth analysis, the functions and corresponding bus accesses of processor 1 and processor 2 in TLM simulation are traced, as shown in Figure 4.13. In the zoomed-in figure, processor 1 is executing the driver function *write_uart*, during which it initiates transactions for communicating with the memory or UART. Concurrently, processor 2 is executing the driver function *readCamera* in the first half of the figure and the application *rgb2yuv* in the second. It can be seen that, when executing *read_camera*, processor 2 initiates transactions more frequently than executing *rgb2yuv*, due to heavy

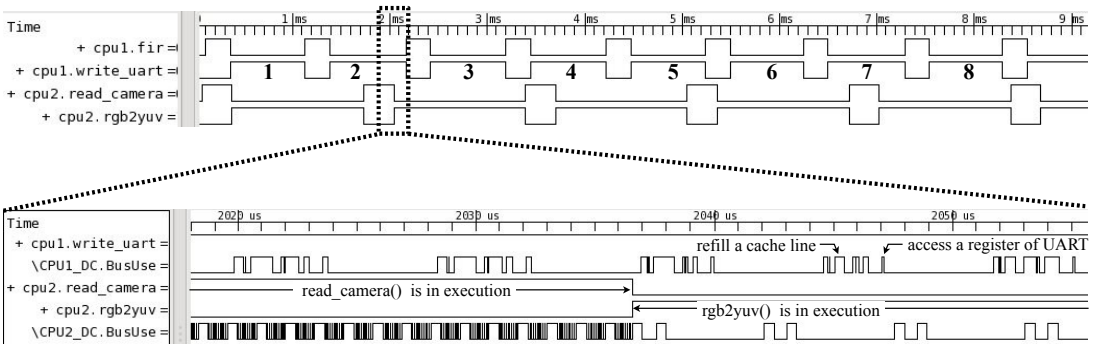


FIGURE 4.13: Traced functions and HW accesses in standard TLM simulation using bus-word transactions.

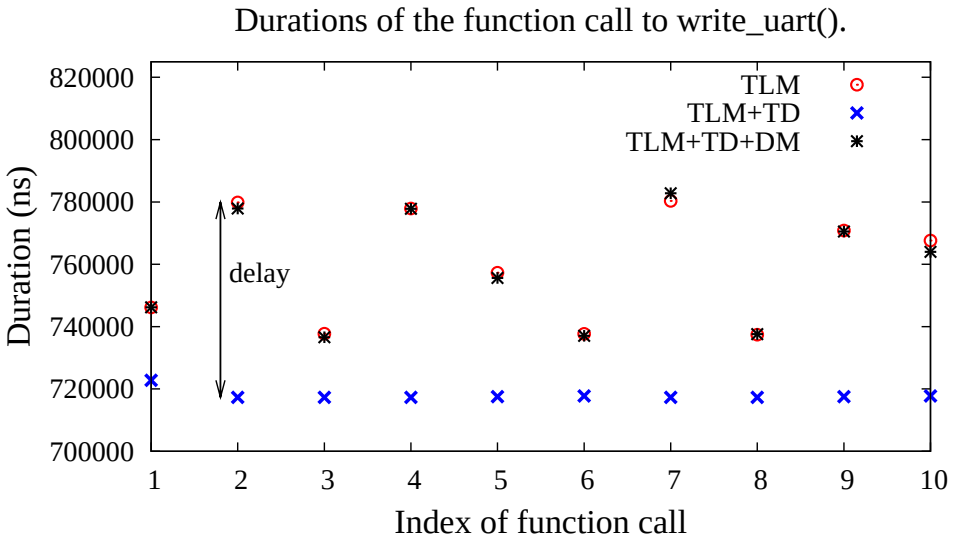


FIGURE 4.14: Timing comparison for *write_uart()*.

I/O communication and frequent cache-line refilling. Therefore, transactions of processor 1 are delayed more when processor 2 is executing *read_camera*. This can be reflected in the measured durations of processor 1's calls to *write_uart* in Figure 4.14. The 2nd, 4th and 7th calls are delayed more, since they overlap more with the execution of *read_camera* by processor 2. Furthermore, it can also be seen that the estimated delays in TLM+TD+DM simulation match very well with those in TLM simulation. This means the proposed delay formula and the scheduling algorithm have successfully modeled the timing related to resource conflicts and dynamically resolved the duration of each code block simulated with temporal decoupling.

The overall simulation results are given in Table 4.1. In TLM+TD mode, the simulation is 19 times faster than TLM mode, demonstrating the speed-up due to temporal decoupling. However, the timing is underestimated by 7%, because the transactions of the lower priority processors are not delayed by the resource conflicts. In TLM+TD+DM mode, the reduced timing estimation error (-0.8%) verifies the efficacy of the proposed analytical timing estimation approach. At the same time, the simulation speed is close to that of TLM+TD simulation. This implies an negligible overhead caused by the scheduling algorithm. To sum up, this experiment shows that integrating analytical timing estimation into the simulation with temporal decoupling is capable of retaining both the timing accuracy and the simulation speed-up.

TABLE 4.1: Multiprocessor simulation results.

Sim. mode	**TLM**	**TLM+TD**	**TLM+TD+DM**
Cycles	306M	288M	303M
Err(%)	-	-7	-0.8
Exe. time (s)	8.81	0.47	0.48
Speed-up.	-	19	18.4

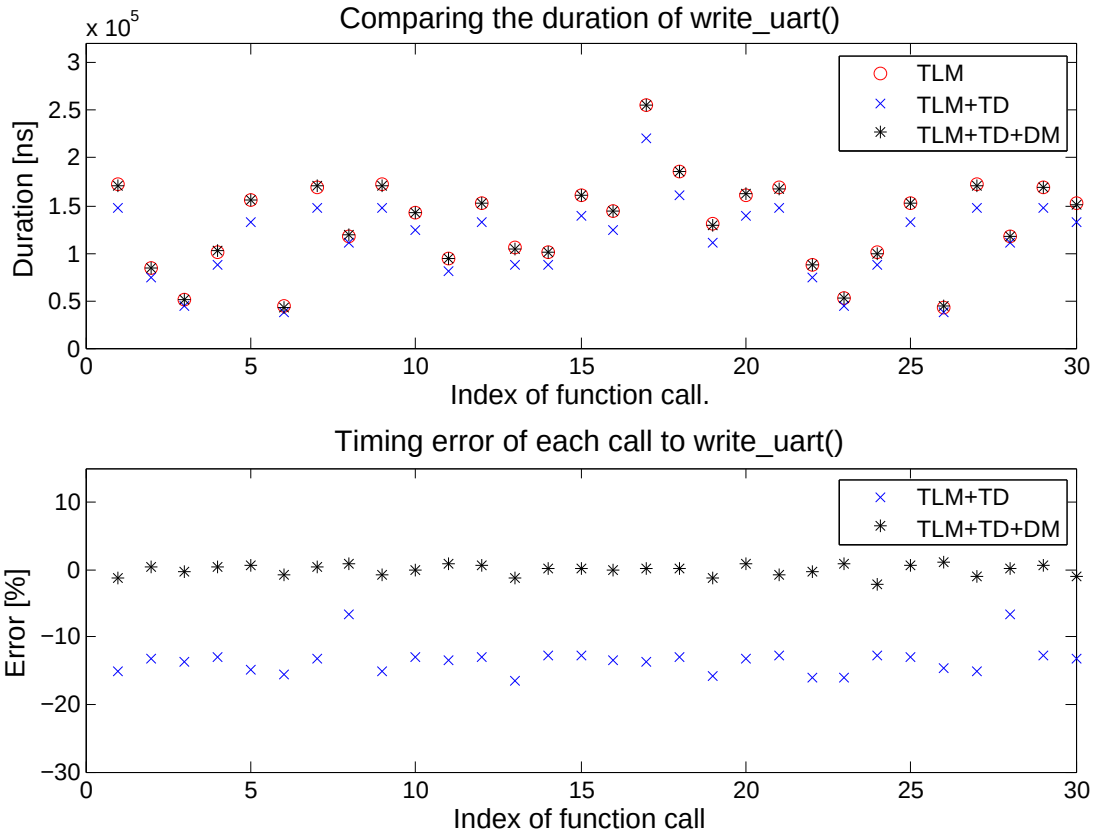


FIGURE 4.15: Durations of processor 3's calls to write_uart() under FIFO arbitration scheme.

4.7.3.3 Simulation with Three Processors

This experiment examines the delay estimation under different arbitration schemes with relative high degree of conflicts. It is set up as such: processor 1 keeps writing new buffers of various lengths to the UART1 module; processor 2 keeps reading a buffer from the AES module; processor 3 keeps writing new buffers of variant lengths to the UART2 module. To stress the proposed approach, the data cache is disabled and polling is used by the interrupt service routine in the driver functions for the I/O devices. This results in heavy traffic and thus frequent access conflicts at the shared bus.

Test scenario 1: A FIFO arbitration scheme is used. Resource availability is calculated according to (4.13). For each processor, the durations of the function calls are measured in each simulation mode. For processor 3, the measured durations of its calls to the *write_uart()* function are given in Figure 4.15. As can be seen, timing in TLM+TD simulation is quite underestimated, with the errors fluctuating around 14%. In contrast,

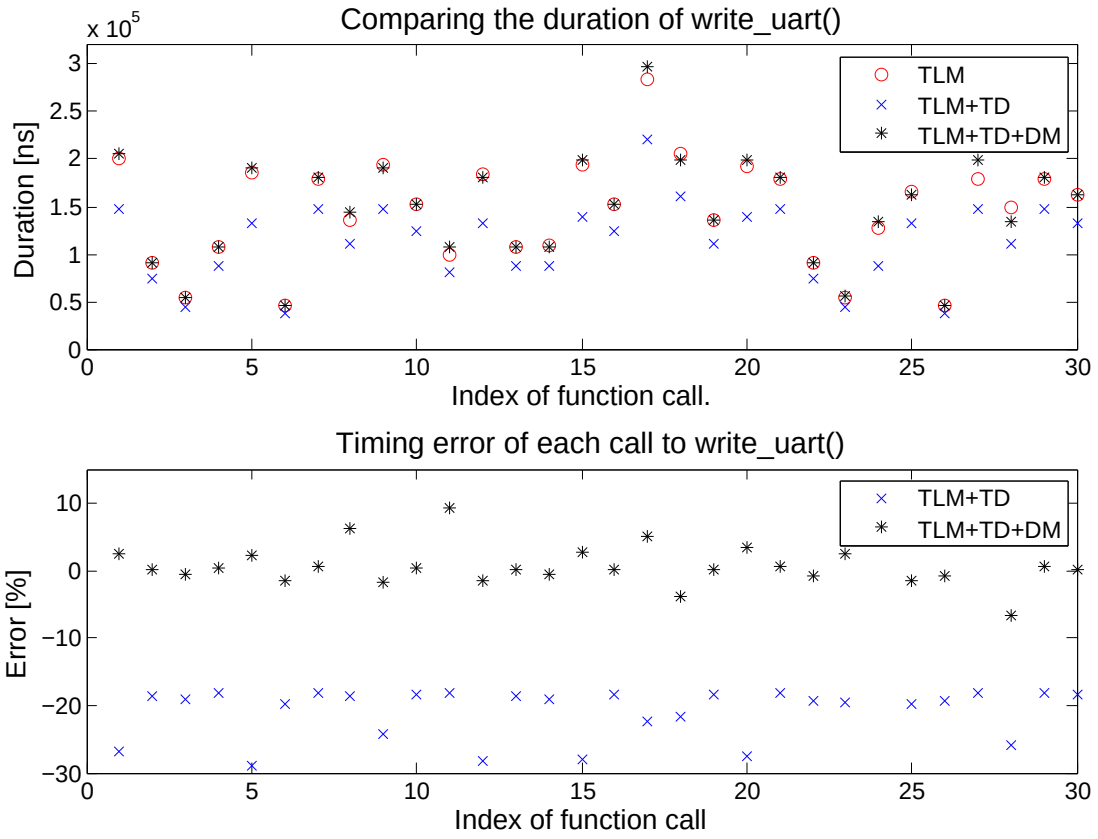


FIGURE 4.16: Durations of processor 3's calls to write_uart() under fixed priority arbitration scheme.

timing in TLM+TD+DM simulation is very well estimated, with most errors in the range of $(-2\%, 2\%)$. Over a long time period, the timing error for the accumulated duration in TLM+TD+DM simulation is below 1%. Similar results are also obtained for the estimated durations of the function calls of processor 1 and processor 2.

Test scenario 2: A fixed priority arbitration scheme is used, with the priority assignment processor 1>processor 2>processor 3. In this case, the function calls of processor 3 should be delayed more, compared with the previous test scenario. The measured durations are given in Figure 4.16. In TLM+TD simulation, the timing is now underestimated by around 20% for most of the calls and up to 30% for a few of them. In TLM+TD+DM simulation, the absolute timing errors in most cases are below 5%, with a few approaching 10%. The relative large timing error for an individual call is because there can be certain offset to the start time of this call, which leads to variation of the resource availability in this offset period. But speaking from the accumulated duration over a long period, the timing estimation exhibits high accuracy with an error below 3%.

The simulation speed-up offered by using TLM+ transactions depends on two factors. The first is represented by the simulation cost in transferring one unit of data in the data block, where the software protocol for this transfer is implemented based on bus-word transactions. The second is the size of the data block. Since the whole data block can be transferred by just a single TLM+ transaction, therefore the larger the data block is, the more speed-up a TLM+ transaction can lead to. Assume there are K bus-word transactions required to transfer one unit of data in the simulation using

bus-word transactions. This corresponds to at least K synchronization effort needed to synchronize before and during the bus-word transactions. Further, assume the size of the data block is N. As a result, when a TLM+ transaction is used to transfer this data block, the simulation speed in terms of modeling this transfer will be improved by approximately $K \times N$ times. The value of K depends on the software protocols. In practice, it can be up to a few hundreds or thousands [14]. The value of N depends on the application. It can range from a few bytes as in transferring data to an encryption module, to a few hundreds of bytes as in transferring data to a serial interface. In some examined cases, a speed-up of three orders of magnitude or more can be reached [14, 121]. The net contribution of TLM+ transactions to the overall simulation efficacy depends on the proportion of the data transfer in the simulated software. This contribution is high in those applications where a large amount of data needs to be frequently transferred such as in the simulation of a communication network.

Chapter 5

Conclusion

The last few decades has seen a design trend of using simulative models to handle the ever increasing design complexity. More and more design tasks are carried out on virtual prototypes. Using virtual protocols expedites the development, reduces the cost and offers high flexibility. Following this need, techniques and tools have emerged for efficient modeling and simulation of the software and hardware of the system under design. The work in this dissertation has put forth several methodologies. They cover two principal aspects in simulating the software and the underlying hardware model. One corresponds to the performance estimation in faster software simulation. For this, there has been a line of research investigating host-compiled software simulation. The other corresponds to the timing estimation in faster modeling of the inter-module communication. Representative work includes the proposal of highly abstract TLM+ transactions and the application of temporal decoupling. Although these lines of research contribute to a faster simulation, problems of degraded simulation accuracy are encountered. The proposed methodologies tackle these problems.

Performance Estimation in Host-Compiled Software Simulation

Driven by detailed structural analysis of the control-flows, this work proposes a methodology that can reliably annotate the source code from an target binary code that has been optimized by the compiler. Through control-flow analysis, structural properties are extracted for the basic blocks of both the source code and the target binary code. Examined structural properties include loop membership, control dependency, the dominance relation, and the post-dominance relation. The control dependency analysis is combined with the loop analysis to identify the intra-loop control dependency. These properties provide useful insight regarding the execution order of the basic blocks. For each basic block in the binary code, its structural properties are progressively checked in the selection of a most appropriate counterpart basic block in the source code. This resolves the correct position in the source code to annotate the performance estimation codes extracted from a basic block in the binary code. The annotated source code by this means can serve as a good performance model of the target binary. Therefore, in the host-compiled simulation, the codes related to performance estimation will be accumulated in an appropriate oder. Compared to the annotation approaches that only resort to the line reference file or the dominance relation, the present methodology is more robust against compiler optimization and hence leads to more accurate performance estimation of the simulated software.

To accurately resolve the addresses of data memory accesses, the key idea in this work is to exploit the memory allocation mechanism. The compiler respects this mechanism in assigning addresses to the binary load and store instructions. This lays a solid theoretical basis for the present approach and therefore ensures the preciseness of the extracted memory addresses. This approach considers three major sections accounting for data memory accesses during the execution of a program, namely the stack, data and heap sections. The address is resolved by considering in which section the accessed variable is located. For variables in the stack, a variable is annotated in the source code to simulate the stack pointer. When a function is called or returned, this variable is modified according to the stack size of the function. Hence, this simulated stack pointer can be used to dereference the address of the accessed variable. For static or global variables, their addresses are obtained explicitly from the debugging information. These addresses can be directly used for cache simulation. For variables in the heap, the growth of the heap is simulated by using the same memory allocation algorithm as in the target binary code. Therefore the value returned by this algorithm can be directly used as the allocated address in the heap. The present approach is also able to handle pointer dereferences, even if the pointers are used as function arguments. With accurately reconstructed data memory accesses, it is possible for the data cache model to simulate when a cache miss occurs. This is helpful in the simulation of a multi-processor model, which requires the availability of the occurrence times of the transactions after cache misses, so that the access conflicts at shared resources can be realistically resolved.

The control-flow analysis and the source code annotation are automated by a tool in this work. Further development of this tool may involve the addition of new features to handle very specific compiler optimizations or path-based annotation. For path-based annotation, the performance modeling codes are extracted from a long path, instead of a single basic block. Path-based annotation may also be necessary to handle certain complex compiler optimizations. One example is the optimization of multiple branch basic blocks such as joining their target basic blocks. The code region containing these branch basic blocks can be identified by finding the hammocks. Then, the paths within this region are enumerated. Each path corresponds to a sequence of branch conditions. For annotation, a path in the source code corresponding to a same sequence of the branch conditions needs to be found. Finally, the performance estimation extracted for the path in the binary code is annotated in the corresponding path of the source code. For other specific optimizations such as loop-tiling, the corresponding loop in the source code may need to be transformed for annotation. Another alternative could be to directly modify the compiler, so that it provides the necessary information regarding what optimization has been performed and which basic blocks are affected. This information could make it much easier to annotate the source code.

A Method to Time TLM+ Transactions

To time TLM+ transactions, a straightforward solution is to find out the timing characteristics of the corresponding driver functions that implement the low-level software protocols, from which the TLM+ transactions are abstracted. This was the task of this work at the beginning of the SANITAS project. The difficulty is to identify the start and end of the driver functions when the software is executed by an ISS. The proposed idea is to inspect the entry and exit instruction addresses of those driver functions. A tool is developed that can provide the addresses of the entry and exit instructions of any functions, through static analyzing the target binary code. By comparing the instruction addresses with those entry and exit addresses, this method is able to trace the exact

start and end of the executed driver functions in ISS-based simulation. As a result, the timing profiles of them can be constructed.

Analytical Timing Simulation for Temporally Decoupled TLMs

For a transaction-level model that is simulated using temporal decoupling, the durations of the concurrent processes are hard to simulate. Timing estimation becomes even more difficult, if TLM+ transactions are used concurrently with other processes. The underlying bus-word transactions in a TLM+ transaction are invisible, since they have been abstracted away. Therefore, using temporal decoupling or TLM+ transactions hides the timing information that is required by the conventional arbitration scheme to resolving timing conflicts at shared resources. An analytical timing estimation methodology is proposed for the timing estimation problem. It abstracts the accesses of hardware modules as resource utilization. Then the resource availability is calculated. Taking the resource utilization and resource availability, a delay formula is proposed to model the effect of timing conflicts at a shared resource. Additionally, an efficient scheduling algorithm integrated in a resource model library dynamically adjusts the duration of a synchronization request. Even this synchronization corresponds to a very long period, the expensive SystemC wait statement is called only once. With this methodology, timing is simulated without the need of tracing the bus-word transactions or performing bus-word transaction based arbitration. It also supports on-demand timing synchronization, and is free from the use of a pre-defined global quantum. Currently, the analytical timing estimation is used for concurrent processes among which there exists no data dependency. If this is not the case, new treatment needs to be added. For example, if two TLM+ transactions access a shared memory region, one of them may need to wait until the other one finishes. This can be modeled by reducing the resource availability to zero for the waiting transaction.

General Consideration and outlook

These methodologies can be used in a combined way in the hardware and software co-simulation. Certain parts of the software can be simulated using host-compiled simulation. Other parts that involve data transfer between the memory and other peripheral devices can be simulated using TLM+ transactions. In the simulated of a multi-processor model, each software on a processor can be simulated in a temporal decoupled way, whereby timing is dynamically and analytically estimated by the present scheduler.

The tools developed in this work can also be applied in a broader way for system modeling and performance analysis. Task level performance modeling can be incorporated in the annotation process. Traffic patterns for the tasks may be parameterized. Resource utilization for a given task can be derived from the traffic density among the resources. Because the analytical timing estimation is not restricted to transaction-level simulation, performance metrics can also be estimated without simulation, given the extracted timing characteristics. To incorporate specific design aspects, the analytical formulas, such as the one for modeling resource availability, may need to be modified. Such extension should not be difficult to achieve in the analytical timing estimation framework, which is flexible in terms of adding new features.

Appendix A

Algorithms in the CFG Analysis

The dominance analysis implemented in the present tool-chain adopts a simple and fast algorithm by Cooper [140], to which the readers are referred for detailed description or explanation. The pseudo code of this algorithm is given in Algorithm A.1. Before identifying the dominators, it first constructs a reverse post-order node set. This set is denoted by N^R. Then it initializes the dominator set for each node to be the whole node set. Following this, the *while* loop completes the dominator construction (from line 1 to line 30). Until this step, the dominator set for each node has already been completely found. However, this dominator set is unordered. Based on this, the following step until line 44 orders the node according to their dominance relation. This results in a dominator tree. It does so by progressively inspecting the nodes at a larger depth. When finished, the ancestors for a node in this tree are thus its ordered dominator set.

The post-dominance analysis is exactly the same as the dominance analysis, but performed on a reversed CFG where the directions of the edges are reversed. So, the root node of the obtained post-dominance tree is the *Exit* node of the original CFG.

Algorithm A.1 Find the ordered dominator set for all nodes in a CFG

```
1: N := the set of nodes in the CFG
2: N^R := a reverse post-order of the nodes in N, see [140]
3:
4: for n_i ∈ N do
5:    dom(n_i) ← N // initialized as the full node set
6: end for
7:
8: changed ← True
9: while changed do
10:   changed ← False
11:   for n_i ∈ N^R do
12:      firstprec ← True
13:      newset ← ∅
14:      // intersecting all dominator sets of the predecessors of the node
15:      for n_k ∈ parent(n_i) do
16:         if firstprec == True then
17:            newset ← dom(n_k)
18:            firstprec ← False
19:         else
20:            newset ← newset ∩ dom(n_k)
21:         end if
22:      end for
23:      newset.add(n_i) //A node dominates itself by definition
24:      //Check if the domination set has changed in this iteration
25:      if newset ≠ dom(n_i) then
26:         dom(n_i) ← newset //Update the dominator set
27:         changed ← True
28:      end if
29:   end for
30: end while
31:
32: // connect the nodes according to the dominance relation to form a tree
33: T_d := a dominance tree initialized with a single root node
34: height = max([dom(n_i).size, for n_i ∈ N])
35: for i ∈ [2, hight + 1] do
36:   for n_i ∈ N do
37:      if i == dom(n_i).size then
38:         N_d ← T_d.getNodesAtDepth(i − 2)
39:         dominator ← dom(n_i) ∩ N_d
40:         T_d.addEdge(dominator, n_i)
41:      end if
42:   end for
43: end for
44: return
```

Appendix B

Details of the Trace and Profile Tool

B.1 The Tracing Mechanism

The class hierarchy of this tracing and profiling mechanism is shown in Figure B.1. A *tracer* class is used to model different events of interest such as the execution of a function, the access to a hardware module, etc. Each tracer contains a profiler that enables online profiling. A tracer can be classified either as state-less or state-based. A state-less tracer is mainly for tracing activities related to a single event such as a time-out event. A state-based tracer is associated with several variable related to state registration. A bi-state tracer is used to model activities that have a start event and a stop event. Examples include a transaction, the execution of a software or hardware function. A multi-state tracer can trace the accesses to a queue where different states corresponds to the current queue length. It can also trace multiple concurrent accesses to a shared resource. In addition, some timing verification tasks can be performed as well by using the states of a tracer. For example, a bi-state tracer is not allowed to start again if it is already in the *start* state. Also, the *end* state can be specified to happen within a certain time interval after the *start* state.

B.2 Tracing the SW Execution

This tool supports multi-level software tracing and profiling, including the function level, basic block level and instruction level. Such panoramic analysis has not been reported by previous tools. To trace the software execution, the ISS forwards the address and type of the executed instruction to a monitor. By checking the type and address, the monitor further calls a corresponding tracer. Following SW tracers are considered.

1. At instruction level: the execution frequency of instructions is profiled according to their types. This profiling can be done for the whole application program or for a specific function.

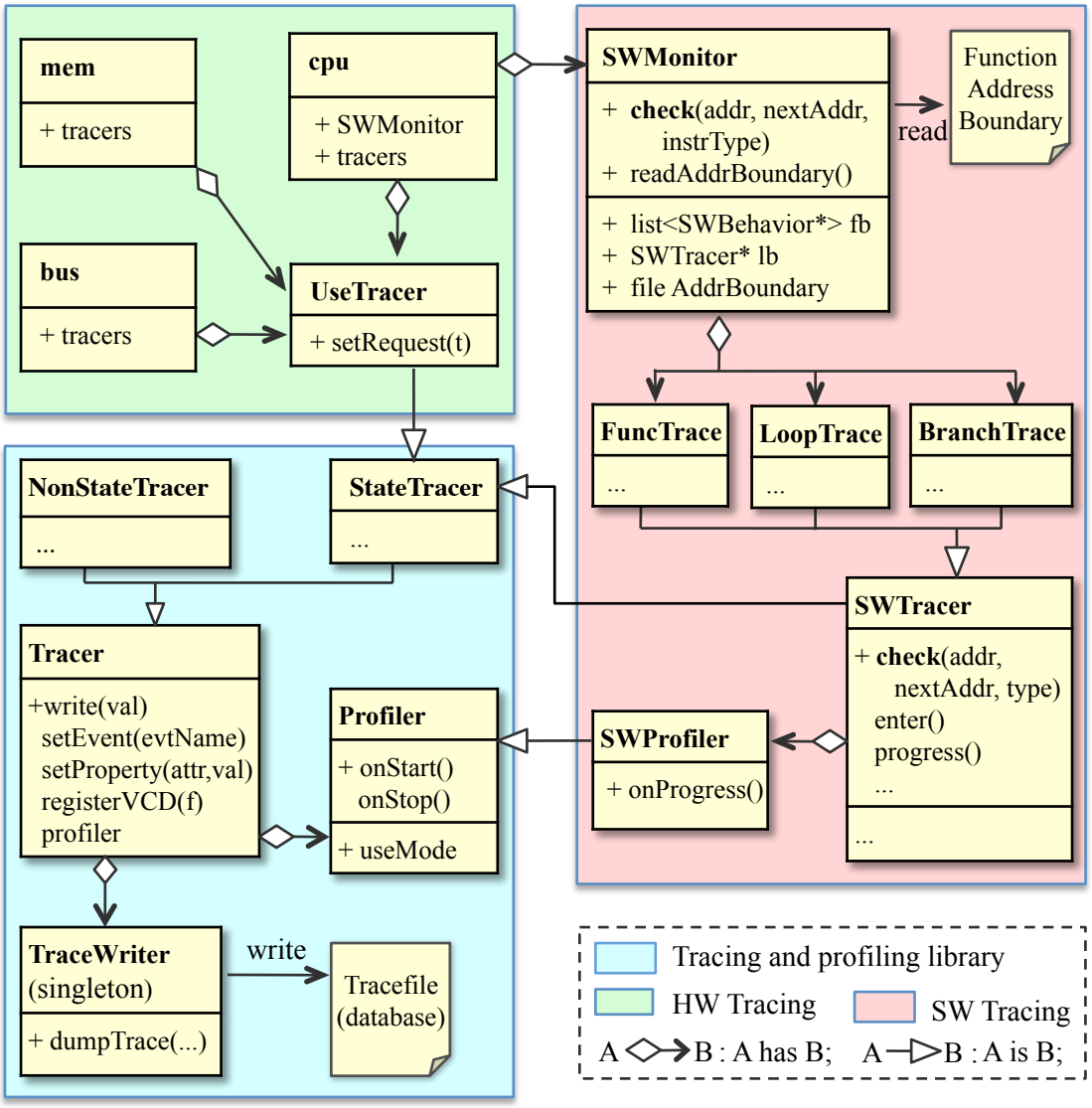


FIGURE B.1: Overall class hierarchy of TraPro.

2. At loop or basic block level: timing statistics of the loops are profiled such as their activation counts and looped count per activation. The traces of branch instructions include its taken/non-taken status, which is also profiled. Additionally, the dynamic control flow graph can be constructed.

3. At function level: The start and end of each function can be traced. A dynamic function call graph can be constructed. In this graph, for a caller function, the number of calls and accumulated time of execution of its callee are provided. This information is not available with static call graph analysis and is of great importance for the designer to inspect and analyze the performance of individual tasks.

Tracing the functions: From the target assembly code, entry and exit addresses of the functions can be parsed. There is only one entrance of a function, which is its first instruction. There can be multiple exits of a function. Taking a MIPS instruction set for

example, the exits correspond to the *jr* (jump and return) instructions, taking a MIPS instruction set for example. As is shown in Figure B.1, the SW monitor traces the executed functions by checking the address and type of each executed instruction. If the instruction is a function call *jal*, then the monitor compares the address to be jumped to against all the function entry addresses. If the instruction is the return of a function, then the monitor compares its address with all the exit addresses of the functions. If neither case is true, then it accumulates the number of instructions for the functions that are currently being executed.

Tracing the loops: From the control-flow graph of the target binary code, all loops can be statically identified. Therefore the instruction addresses of the loop head instructions, loop latch instructions, and the next instruction when a loop exits are known. With these addresses, the software monitor can detect the entry and exit of a loop, similar to the case of function tracing. Besides, each time a loop latch instruction is executed, one iteration of the loop is finished and the loop tracer will be informed. The executed loops are stored in a stack-like structure. A loop is popped from the stack when it exits.

Tracing the branches: To trace the *taken* or *not-taken* status of the branch instructions, the monitor inspects the next instruction address after a branch instruction. If it is the target address of the branch, then this branch is taken, otherwise not taken.

B.3 Tracing the HW Activities

To trace the access to a hardware module, a *UseTracer* is used (see the upper left in Figure B.1). It can trace timing information such as the request time, delay and duration of a transaction that accesses a certain hardware module. In TLM2.0, a transaction is initiated as $transport(gp, t)$, where gp and t represent a generic payload and a time variable respectively. The variable t is used to store the request time. The transaction can be delayed, resulting in a longer duration than the request time. A *UseTracer* has a method *useRequest(t)* which is called to register the request time before a transaction starts. When a transaction stops, the *UseTracer* is informed to automatically calculate the delay.

An example of using the *UseTracer* to trace the access to a hardware in case of a transaction is demonstrated in List B.1. The original *transport()* call is wrapped by the methods of the *UseTracer* class. The method *write(1)* is called after a transaction is received. The method *write(-1)* is called after a transaction is complete. The number of overlapping requests at a shared resource can be traced.

```
1  class CPU: public sc_module...
2    UseBhv u; //behavior declaration
3    ...
4    int write(addr, data){
5        tlm::tlm_generic_payload x;
6        ...
7      u.setProperty("Addr", addr);
8      u.write(1);       //start: will set start time
9        busport.transport(x,t);
```

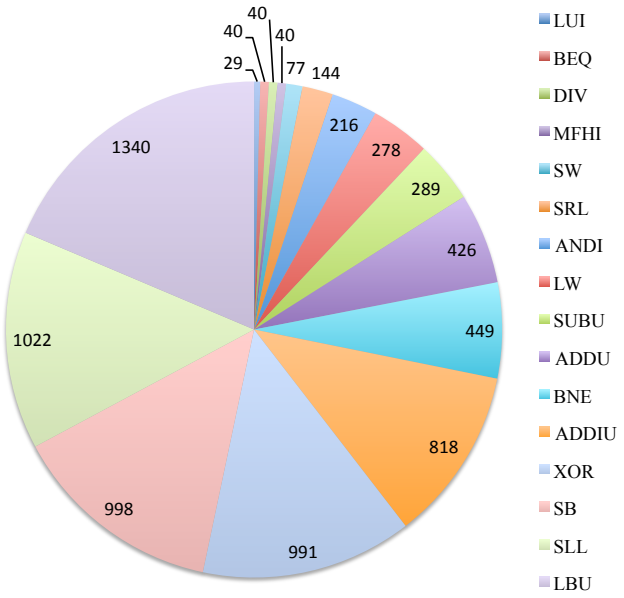


FIGURE B.2: Instruction profile of the application program.

```
10      u.useRequest(t);   //set request time
11      u.write(-1);       //stop: will calculate delay
12      ...
13    }
14    //At the initiation of the cpu class
15    CPU(...):
16      //set owner and type of the behavior
17      u(_name,"X_WRITE")
18    ...
```

LISTING B.1: Example of tracing resource contention

B.4 Application of the Tracing Tool

B.4.1 Results of Traced Software Execution

To show the results generated by this trace and profile tool, an AES encryption algorithm is simulated on a ISS-based model. Results related to the computation aspects of the program are given in this section, while those related to the communication aspects are in the next section.

The instruction profiling is shown in Figure B.2. As is shown, the *LBU, SLL, SB, XOR* instructions consume over 50% in the overall executed instructions. Such information can be useful in selecting the CPU's micro architecture.

The traced loops are shown in Table B.1. The first row gives the addresses of the loop head instructions. The second row shows how many times the loop is activated. The third row shows the total number of executed instructions for each loop. From the

Addr	1114	1c8	750	9c0	90c	6e0	6ec	1240	a04	be0	898	7dc	250	8ac	650	**2d4**	354
lCnt	2	3	3	3	3	12	3	31	3	15	27	27	120	8	27	39	10
iCnt	12	34	58	58	70	76	91	93	94	151	522	630	760	944	1656	1939	2400

TABLE B.1: Instruction and iteration counts for different loops.

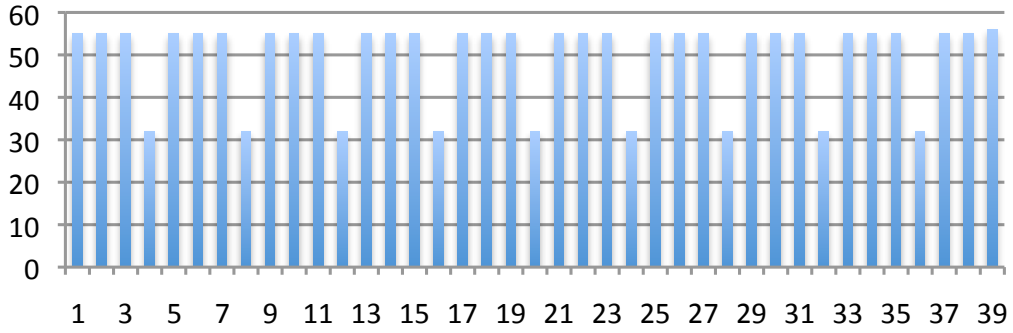


FIGURE B.3: Iteration count profile. The x-axis is the number of entrance to this loop. The y-axis gives the iterations count each time this loop is entered.

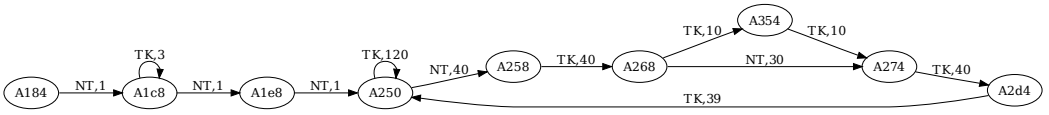


FIGURE B.4: Dynamic control flow of *KeyExpansion()*.

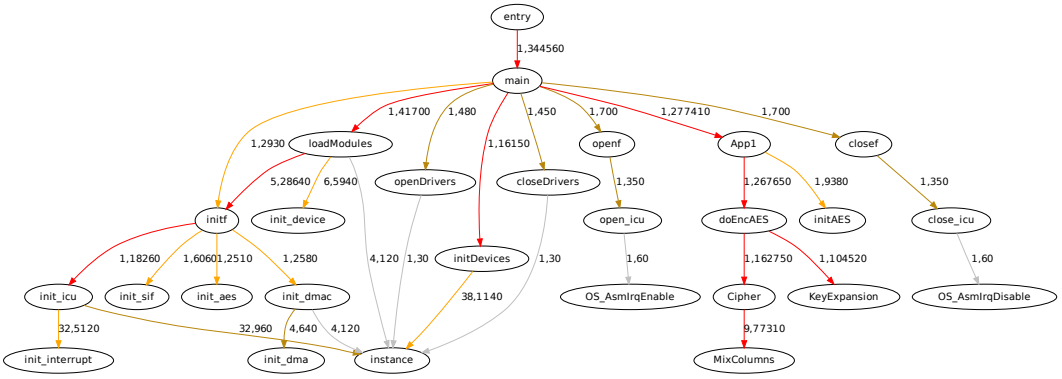


FIGURE B.5: The dynamic function call graph, with edges colored by execution time.

table, the loops that have high activation counts or executed instruction counts can be identified. These loops can be examined more closely. For example, the iteration count profile for a loop with its head address at $0x2d4$ is given in Figure B.3. It can be seen that this loop is entered 39 times. These information may help the designers to revise the loops in the source code.

From tracing the taken/non-taken status of branch instructions, a dynamic control-flow graph can be constructed. An example is given in Figure B.4. In this figure, a node consist of the address of a branch instruction. The label of edge indicates how many times the branch is taken (TK) or not taken (NT).

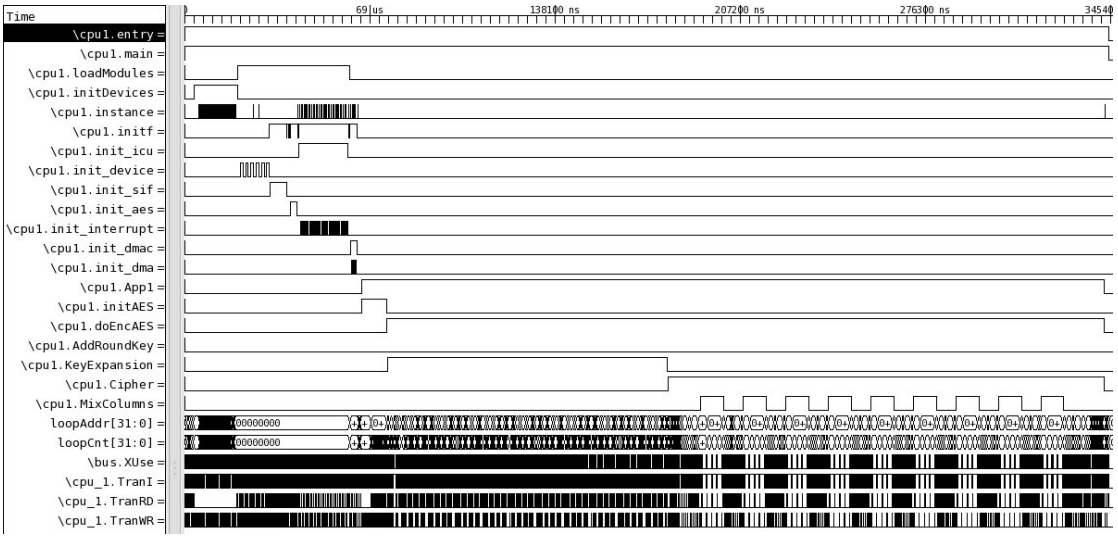


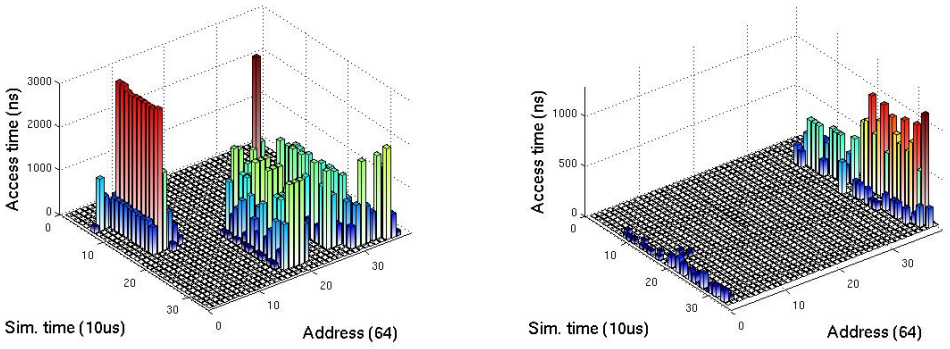
FIGURE B.6: The traced functions.

Based on the function tracing, a dynamic function call graph during the software simulation can be constructed. An example is given in Figure B.5. The label of an edge denotes the number of function calls and the execution time of the called function. The edges are colored according to the execution time of the callee function. This figure shows the dynamics of function activation and can be used to examine the computational cost of the sub-routines. A waveform-style figure of all traced functions are shown in Figure B.6. This figure provided by the software monitor. It sets the value of a corresponding signal to high or low when it detects the entry or exit of a function by inspecting the address of the executed instruction.

B.4.2 Results of Traced Hardware Accesses

The traced memory accesses are profiled over a two dimensional time-address plane. For a resolution of $(10us, 64B)$, the resulted profile shows the memory access time within every time interval of $10us$ and an address interval $64Bytes$. The resulted profile visualizes memory access patterns in different memory sections, including the instruction memory, the static data section, the stack and the heap section. Example results are given in Figure B.7. For clarity, the address is offset to the minimum address in the corresponding profile. Such memory access patterns are useful in aiding the design of memory hierarchy such as determining the size and associativity of the cache.

The concurrent accesses to a hardware module can be examined after the simulation. An experiment with 3 CPUs connected to a shared bus is conducted. A sample waveform is given in Fig B.8. The first signal in this figure is the number of access requests on this bus. A value larger than 1 indicate access conflicts. The other signals represent the transaction durations of each CPU. For example, at time $22.52us$, there are 3 access requests on the bus, due to CPU1's write data request, CPU2's write data request and CPU3's read data request, respectively.



(a) Instruction memory access profile.

(b) Static data access profile.

FIGURE B.7: Memory access profiles

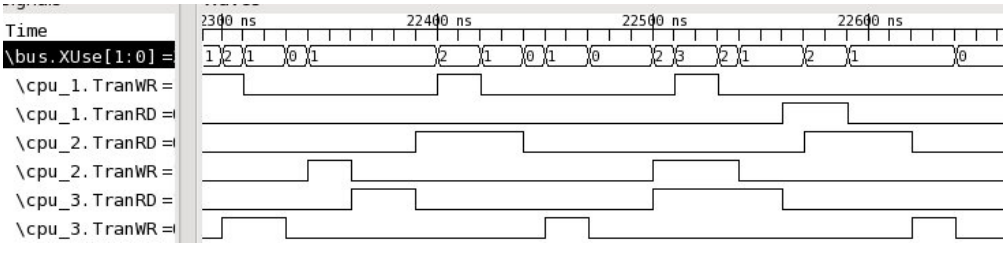


FIGURE B.8: Bus conflicts when using 3 CPUs.

List of Figures

List of Tables

Symbols

n_i	a node in a CFG
$G^s(N^s, E^s)$	CFG of the source code
$G^b(N^b, E^b)$	CFG of the binary code
N^s	a set of the source code basic block
N^b	a set of the binary code basic block
n_i^s	basic block in the source code CFG
n_i^b	basic block in the binary code CFG
$dom(n_i)$	dominator set
$pdom(n_i)$	dominator set
$idom(n_i)$	immediate dominator
$ipdom(n_i)$	immediate post-dominator
l_i	a loop in the CFG
$N(l_i)$	the natural loop of l_i
E_{l_i}	the entry edge of l_i
X_{l_i}	the exit edge(s) of l_i
P_m	loop membership property
P_c	intra-loop controlling node property
P_b	immediate dominate property
E_R	mapping edges line reference
E_B	mapping edges line reference for the branch instruction
ϕ_c	selection by matching control dependency
ϕ_m	selection by matching loop membership
ϕ_m	selection by matching branch dominator
ϕ_r	selection by matching line reference
$\mathcal{S}_i$	a timing synchronization request
u_{ij}	resource utilization of $\mathcal{S}_i$ at resource R_j
a_{ij}	resource availability of $\mathcal{S}_i$ at resource R_j
d_{ij}	delay of $\mathcal{S}_i$ at resource R_j

Index

Bibliography

[1] J.A. Rowson. Hardware/software co-simulation. In *Design Automation, 1994. 31st Conference on*, pages 439–440, 1994.

[2] D. Becker, R. K. Singh, and S. G. Tell. An engineering environment for hardware/software co-simulation. In *Design Automation Conference*, DAC '92. ACM, 1992.

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

[4] J.T. Buck, S. Ha, E.A. Lee, and D.G. Messerschmitt. Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems. *Int. Journal of Computer Simulation*, 1994.

[5] Value Proposition of CoFluent Studio. http://www.intel.de/content/www/de/de/cofluent/cofluent-difference.html. [Online; accessed 20-July-2013].

[6] Plasma CPU. http://opencores.org/. [Online; accessed 19-July-2013].

[7] RealView ARMulator Instruction Set Simulator. http://arm.com/. [Online; accessed 19-July-2013].

[8] OpenRISC 1200. http://openrisc.net/or1200-spec.html. [Online; accessed 19-July-2013].

[9] Jianwen Zhu and Daniel Gajski. A retargetable, ultra-fast instruction set simulator. In *Design, Automation and Test in Europe (DATE)*, 1999.

[10] Mehrdad Reshadi, Prabhat Mishra, and Nikil Dutt. nstruction set compiled simulation: a technique for fast and flexible instruction set simulation. In *ACM/IEEE Design Automation Conference (DAC)*, 2003.

[11] Rainer Leupers, Johann Elste, and Birger Landwehr. Generation of interpretive and compiled instruction set simulators. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*, 1999.

[12] Achim Nohl, Gunnar Braun, Oliver Schliebusch, Rainer Leupers, Heinrich Meyr, and Andreas Hoffmann. A universal technique for fast and flexible instruction-set architecture simulation. In *ACM/IEEE Design Automation Conference (DAC)*, 2002.

[13] IEEE. *IEEE Standard SystemC Language Reference Manual*, 2005.

[14] W. Ecker, V. Esen, and M. Velten. TLM+ modeling of embedded HW/SW systems. In *Design, Automation and Test in Europe (DATE)*, 2010.

[15] OSCI. *OSCI TLM-2.0 Language Reference Manual*, 2009.

[16] Ken Hines. Pia: A framework for embedded system co-simulation with dynamic communication support. Technical report, University of Washington, 1996.

[17] K. Hines and G. Borriello. Optimizing communication in embedded system co-simulation. In *Hardware/Software Codesign, 1997. (CODES/CASHE '97), Proceedings of the Fifth International Workshop on*, pages 121–125, 1997.

[18] Wikipedia. Transaction-level modeling. http://en.wikipedia.org/wiki/Transaction-level_modeling. [Online; accessed 21-July-2013].

[19] Wolfgang Ecker, Stefan Heinen, and Michael Velten. Using a dataflow abstracted virtual prototype for hds-design. In *Design Automation Conference, 2009. ASP-DAC 2009. Asia and South Pacific*, pages 293–300. IEEE, 2009.

[20] H-J Schlebusch, Gary Smith, Donatella Sciuto, Daniel Gajski, Carsten Mielenz, Christopher K Lennard, Frank Ghenassia, Stuart Swan, and Joachim Kunkel. Transaction based design: Another buzzword or the solution to a design problem? In *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pages 876–877. IEEE, 2003.

[21] C.A. Valderrama, A. Changuel, P. V. Raghavan, M. Abid, T. Ben Ismail, and A.A. Jerraya. A unified model for co-simulation and co-synthesis of mixed hardware/software systems. In *European Design and Test Conference, 1995. ED TC 1995, Proceedings.*, pages 180–184, 1995.

[22] R. Klein. Miami: a hardware software co-simulation environment. In *Rapid System Prototyping, 1996. Proceedings., Seventh IEEE International Workshop on*, pages 173–177, 1996.

[23] C.A. Valderrama, F. Nacabal, P. Paulin, and A.A. Jerraya. Automatic generation of interfaces for distributed c-vhdl cosimulation of embedded systems: an industrial experience. In *Rapid System Prototyping, 1996. Proceedings., Seventh IEEE International Workshop on*, pages 72–77, 1996.

[24] C. Liem, F. Nacabal, C. Valderrama, P. Paulin, and A. Jerraya. System-on-a-chip cosimulation and compilation. *Design Test of Computers, IEEE*, 14(2):16–25, 1997.

[25] Jie Liu, Marcello Lajolo, and Alberto Sangiovanni-Vincentelli. Software timing analysis using hw/sw cosimulation and instruction set simulator. In *Proceedings of the 6th international workshop on Hardware/software codesign*, CODES/CASHE '98, pages 65–69. IEEE Computer Society, 1998. ISBN 0-8186-8442-9.

[26] Guang Yang, Xi Chen, F. Balarin, H. Hsieh, and A. Sangiovanni-Vincentelli. Communication and co-simulation infrastructure for heterogeneous system integration. In *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, volume 1, pages 1–6, 2006.

[27] Alessandro Forin, Behnam Neekzad, and Nathaniel L. Lynch. Giano: The two-headed system simulator. Technical report, Microsoft Research, 2006.

[28] Moo-Kyoung Chung and Chong-Min Kyung. Enhancing performance of hw/sw cosimulation and coemulation by reducing communication overhead. *Computers, IEEE Transactions on*, 55(2):125–136, 2006.

[29] Ping Hang Cheung, Kecheng Hao, and Fei Xie. Component-based hardware/software co-simulation. In *Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on*, pages 265–270, 2007.

[30] G. Beltrame, D. Sciuto, and C. Silvano. Multi-accuracy power and performance transaction-level modeling. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(10):1830–1842, 2007.

[31] Jinyong Jung, Sungjoo Yoo, and Kiyoung Choi. Fast cycle-approximate mpsoc simulation based on synchronization time-point prediction. *Design Automation for Embedded Systems*, 2007.

[32] S. Cordibella, F. Fummi, G. Perbellini, and D. Quaglia. A hw/sw co-simulation framework for the verification of multi-cpu systems. In *High Level Design Validation and Test Workshop, 2008. HLDVT '08. IEEE International*, pages 125–131, 2008.

[33] J.E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12, 2010.

[34] Heiko Huebert. Seamless co-verification environment user's and reference manual, 1996.

[35] Mentor Graphics. ModelSim. http://www.mentor.com/products/fpga/model. [Online; accessed 20-July-2013].

[36] IEEE. IEEE Standard for Property Specification Language (PSL), 2010. [Online; accessed 20-July-2013].

[37] Carbon simulator. http://www.carbondesignsystems.com/. [Online; accessed 20-July-2013].

[38] GEZEL. GEZEL Hardware/Software Codesign Environment. http://rijndael.ece.vt.edu/gezel2/index.html, 2005. [Online; accessed 19-July-2013].

[39] Patrick Schaumont and Ingrid Verbauwhede. A component-based design environment for esl design. *IEEE Design and Test of Computers*, pages 338–347, 2006.

[40] Patrick Schaumont, Doris Ching, and Ingrid Verbauwhede. An interactive codesign environment for domain-specific coprocessors. *ACM Trans. Des. Autom. Electron. Syst.*, pages 70–87, 2006.

[41] AMcTools. VMLAB: a virtual prototyping IDE. http://www.amctools.com/vmlab.htm, 2009. [Online; accessed 19-July-2013].

[42] Synopsis. CoCentric System Studio. http://www.synopsys.com/Tools/Pages/default.aspx. [Online; accessed 20-July-2013].

[43] Intel. CoFluent Studio. http://www.intel.com/content/www/us/en/cofluent/intel-cofluent-studio.html. [Online; accessed 20-July-2013].

[44] Vojin Zivojnovic and Heinrich Meyr. Compiled HW/SW co-simulation . In *ACM/IEEE Design Automation Conference (DAC)*, 1996.

[45] M. Lajolo, M. Lazarescu, and A. L. Sangiovanni-Vincentelli. A compilation-based software estimation scheme for hardware/software co-simulation. In *International Conference on Hardware Software Codesign*, 1999.

[46] J. R. Bammi, W. Kruijtzer, L. Lavagno, E. Harcourt, and M. T. Lazarescu. Software performance estimation strategies in a system-level design tool. In *International Workshop on Hardware/Software Codesign (CODES)*, 2000.

[47] J. Y. Lee and I. C. Park. Timed compiled-code simulation of embedded software for performance analysis of SOC design. In *ACM/IEEE Design Automation Conference (DAC)*, 2002.

[48] H. Posadas, F. Herrera, P. Sanchez, E. Villar, and F. Blasco. System-level performance analysis in SystemC. In *Design, Automation and Test in Europe (DATE)*, 2004.

[49] Kingshuk Karuri, M.A.A.Faruque, Stefan Kraemer, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. Fine-grained Application Source Code Profiling for ASIP Design. In *ACM/IEEE Design Automation Conference (DAC)*, 2005.

[50] Torsten Kempf, Kingshuk Karuri, Stefan Wallentowitz, Gerd Ascheid, Rainer Leupers, and Heinrich Meyr. A SW performance estimation framework for early system-level-design using fine-grained instrumentation. In *Design, Automation and Test in Europe (DATE)*, 2006.

[51] E. Cheung, H. Hsieh, and F. Balarin. Framework for fast and accurate performance simulation of multiprocessor systems. In *IEEE International High Level Design Validation and Test Workshop*, 2007.

[52] Trevor Meyerowitz, Alberto Sangiovanni-Vincentelli, Mirko Sauermann, and Dominik Langen. Source-Level Timing Annotation and Simulation for a Heterogeneous Multiprocessor. In *Design, Automation and Test in Europe (DATE)*, 2008.

[53] J. Schnerr, O. Bringmann, A. Viehl, and W. Rosenstiel. High-performance timing simulation of embedded software. In *ACM/IEEE Design Automation Conference (DAC)*, 2008.

[54] Yonghyun Hwang, S. Abdi, and D. Gajski. Cycle-approximate Retargetable Performance Estimation at the Transaction Level. In *Design, Automation and Test in Europe (DATE)*, 2008.

[55] P. Gerin, M. M. Hamayun, and F. Petrot. Native MPSoC co-simulation environment for software performance estimation. In *International conference on Hardware/Software codesign and system synthesis*, 2009.

[56] Lei Gao, Jia Huang, Jianjiang Ceng, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. Totalprof: A Fast and Accurate Retargetable Source Code Profiler. In *CODES ISSS*, 2009.

[57] A. Pedram, D. Craven, and A. Gerstlauer. Modeling Cache Effects at the Transaction Level. In *IESS*, 2009.

[58] Kai-Li Lin, Chen-Kang Lo, and Ren-Song Tsay. Source-Level Timing Annotation for Fast and Accurate TLM Computation Model Generation. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2010.

[59] H. Posadas, L. Diaz, and E. Villar. Fast Data-Cache Modeling for Native Co-Simulation. *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2010.

[60] Daniel Mueller-Gritschneder, Kun Lu, and Ulf Schlichtmann. Control-flow-driven Source Level Timing Annotation for Embedded Software Models on Transaction Level. In *EUROMICRO Conference on Digital System Design (DSD)*, September 2011.

[61] Stefan Stattelmann, Oliver Bringmann, and Wolfgang Rosenstiel. Fast and Accurate Source-Level Simulation of Software Timing Considering Complex Code Optimizations. *Design Automation Conference (DAC)*, 2011.

[62] S. Stattelmann, O. Bringmann, and W. Rosenstiel. Dominator homomorphism based code matching for source-level simulation of embedded software. In *International conference on Hardware/Software codesign and system synthesis (CODES+ISSS)*, 2011.

[63] S. Stattelmann, G. Gebhard, C. Cullmann, O. Bringmann, and W. Rosenstiel. Hybrid Source-Level Simulation of Data Caches Using Abstract Cache Models. In *Design, Automation and Test in Europe (DATE)*, 2012.

[64] Z. Wang and Andreas Herkersdorf. An efficient approach for system-level timing simulation of compiler-optimized embedded software. In *ACM/IEEE Design Automation Conference (DAC)*, July 2009.

[65] C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto. Source-Level Execution Time Estimation of C Programs. In *CODES*, 2001.

[66] L. Cai, A. Gerstlauer, and D. Gajski. Multi-Metric and Multi-Entity Characterization of Application for Early System Design Exploration. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2005.

[67] G. Schirner, A. Gerstlauer, and R. Doemer. Abstract, Multifaceted Modeling of Embedded Processors for System Level Design. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2007.

[68] Yang Xu, Bo Wang, Ralph Hasholzner, Rafael Rosales, and Juergen Teich. On Robust Task-Accurate Performance Estimation. In *ACM/IEEE Design Automation Conference (DAC)*, 2013.

[69] A. Bouchhima, P. Gerin, and F. Petrot. Automatic Instrumentation of Embedded Software for High Level Hardware/Software Co-Simulation. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2009.

[70] P. Giusto, G. Martin, and E. Harcourt. Reliable estimation of execution time of embedded software. In *Design, Automation and Test in Europe (DATE)*, 2001.

[71] M. S. Oyamada, F. Zschornack, and F. R. Wagner. Applying neural networks to performance estimation of embedded software. *Journal of Systems Architecture*, 2008.

[72] Henning Zabel and Wolfgang Mueller. Increased Accuracy through Noise Injection in Abstract RTOS Simulation. In *Design, Automation and Test in Europe (DATE)*, 2009.

[73] I. Moussa, T. Grellier, and G. Nguyen. Exploring SW performance using SoC transaction-level modeling . In *Design, Automation and Test in Europe (DATE)*, 2003.

[74] S. Yoo, G. Nicolescu, L. Gauthier, and A.A. Jerraya. Building Fast and Accurate SW Simulation Models Based on Hardware Abstraction Layer and Simulation Environment Abstraction Layer. In *Design, Automation and Test in Europe (DATE)*, 2003.

[75] Eric Cheung, Harry Hsieh, and Felice Balarin. Memory subsystem simulation in software TLM/T models. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2009.

[76] Stefan Stattelmann, Oliver Bringmann, and Wolfgang Rosenstiel. Fast and Accurate Resource Conflict Simulation for Performance Analysis of Multi-Core Systems. In *Design, Automation and Test in Europe (DATE)*, 2011.

[77] A. Kohler and Martin Radetzki. A SystemC TLM2 model of communication in wormhole switched Networks-On-Chip. In *Forum on Specification and Design Languages (FDL)*, 2009.

[78] T. Arpinen, Erno Salminen, Timo Hamalainen, and Marko Hnnikaninen. Performance evaluation of UML2-modeled embedded streaming applications with system-level simulation. *EURASIP Journal on Embedded Systems*, 2009.

[79] Chi-Fu Chang and YarSun Hsu. A System Exploration Platform for Network-on-Chip. In *International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, 2010.

[80] S. Le Nours, A. Barreteau, and O. Pasquier. A state-based modeling approach for fast performance evaluation of embedded system architectures. In *IEEE International Symposium on Rapid System Prototyping (RSP)*, 2011.

[81] G. Schirner and R. Doemer. Fast and Accurate Transaction Level Models using Result Oriented Modeling. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2006.

[82] Leandro S. Indrusiak and Osmar M. dos Santos. Fast and accurate transaction-level model of a wormhole network-on-chip with priority preemptive virtual channel arbitration. In *Design, Automation and Test in Europe (DATE)*, 2011.

[83] Parisa Razaghi and Andreas Gerstlauer. Predictive OS Modeling for Host-Compiled Simulation of Periodic Real-Time Task Sets. *IEEE Embedded System Letters (ESL)*, 2012.

[84] Alex Bobrek, Joshua Pieper, Jeffrey Nelson, JoAnn M. Paul, and Donald E. Thomas. Modeling shared resource contention using a hybrid simulation/analytical approach. In *Design, Automation and Test in Europe (DATE)*, 2004.

[85] K. Lahiri, A. Raghunathan, and S. Dey. Design space explo- ration for optimizing on-chip communication architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2004.

[86] Simon Kuenzli, Francesco Poletti, Luca Benini, and Lothar Thiele. Combining simulation and formal methods for system-level performance analysis. In *Design, Automation and Test in Europe (DATE)*, 2006.

[87] Alex Bobrek, JoAnn M. Paul, and Donald E. Thomas. Stochastic contention level simulation for single-chip heterogeneous multiprocessors. *IEEE Transactions on Computers*, 2010.

[88] Umit Ogras and Radu Marculescu. Analytical Router Modeling for Networks-on-Chip Performance Analysis. In *Design, Automation and Test in Europe (DATE)*, 2007.

[89] Umit Ogras, Paul Bogdan, and Radu Marculescu. An Analytical Approach for Network-on-Chip Performance Analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2010.

[90] Mingche Lai, Lei Gao, Nong Xiao, and Zhiying Wang. An accurate and efficient performance analysis approach based on queuing model for network on chip. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2009.

[91] Nikita Nikitin and Jordi Cortadella. A performance analytical model for Network-on-Chip with constant service time routers . In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2009.

[92] S. Foroutan, Y. Thonnart, R. Hersemeule, and A. A. Jerraya. An analytical method for evaluating Network-on-Chip performance. In *Design, Automation and Test in Europe (DATE)*, 2010.

[93] Youhui Zhang, Xiaoguo Dong, and Weimin Zheng. A Performance Model for Network-on-Chip Wormhole Routers. *Journal of Computers*, 2012.

[94] Jih-Ching Chiu, Kai-Ming Yang, and Chen-Ang Wong. Analytical Modeling for Multi-transaction Bus on Distributed Systems. *Algorithms and Architectures for Parallel Processing*, 2012.

[95] S Foroutan, Y. Thonnart, and F. Petrot. An Iterative Computational Technique for Performance Evaluation of Networks-on-Chip. *IEEE Transactions on Computers*, 2012.

[96] Erik Fischer, Albrecht Fehske, and Gerhard P. Fettweis. A Flexible Analytic Model for the Design Space Exploration of Many-Core Network-on-Chips Based on Queueing Theory . In *International Conference on Advances in System Simulation (SIMUL)*, 2012.

[97] Vinitha A. Palaniveloo and Arcot Sowmya. Formal estimation of worst-case communication latency in a Network-on-chip. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2012.

[98] Abbas Eslami Kiasari, Zhonghai Lu, and Axel Jantsch. An Analytical Latency Model for Networks-on-Chip. *IEEE Transactions on VERY LARGE SCALE INTEGRATION Systems (TVLSI)*, 2013.

[99] S. Foroutan, Y. Thonnart, R. Hersemeule, and A. A. Jerraya. A Markov chain based method for NoC end-to-end latency evaluation. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 2010.

[100] Soeren Sonntag, Matthias Gries, and Christian Sauer. SystemQ: A Queuing-Based Approach to Architecture Performance Evaluation with SystemC. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, 2005.

[101] Soeren Sonntag, Matthias Gries, and Christian Sauer. SystemQ: Bridging the gap between queuing-based performance evaluation and SystemC. *Design Automation for Embedded Systems*, 2007.

[102] A. Bobrek, JoAnn M. Paul, and Donald E. Thomas. Shared resource access attributes for high-level contention models. In *ACM/IEEE Design Automation Conference (DAC)*, 2007.

[103] Kalle Holma, Mikko Setaelae, Erno Salminen, Marko Haennikaeinen, and Timo D. Haemaelaeinen. Evaluating the model accuracy in automated design space exploration. *Microprocessors and Microsystems*, August 2008.

[104] Chafic Jaber, Andreas Kanstein, Ludovic Apvrille, Amer Baghdadi, and Renaud Pacalet. High-level system modeling for rapid hw/sw architecture exploration. In *IEEE International Symposium on Rapid System Prototyping (RSP)*, 2009.

[105] Hans-Peter Loeb and Christian Sauer. Fast SystemC Performance Models for the Exploration of Embedded Memories. *Advances in Design Methods from Modeling Languages for Embedded Systems and SoC*, 2010.

[106] M. Otoom and J. M. Paul. Workload Mode Identification for Chip Heterogeneous Multiprocessors. *International Journal of Parallel Programming*, 2012.

[107] Zhiliang Qian, Dacheng Juan, Paul Bogdan, and Radu Marculescu. SVR-NoC: A Performance Analysis Tool for Network-on-Chips Using Learning-based Support Vector Regression Model. In *Design, Automation and Test in Europe (DATE)*, March 2013.

[108] Christoph Roth, Oliver Sander, Matthias Kuehnle, and Juergen Becker. HLA-based simulation environment for distributed SystemC simulation . In *International ICST Conference on Simulation Tools and Techniques (SIMUTools)*, 2011.

[109] Rauf Khaligh and Martin Radetzki. Efficient Parallel Transaction Level Simulation by Exploiting Temporal Decoupling. *Analysis, Architectures and Modelling of Embedded Systems*, 2009.

[110] Rauf S. Khaligh and Martin Radetzki. A dynamic load balancing method for parallel simulation of accuracy adaptive TLMs. In *Forum on Specification and Design Languages (FDL)*, 2010.

[111] A. Mello, I. Maia, A. Greiner, and F. Pecheux. Parallel simulation of systemC TLM 2.0 compliant MPSoC on SMP workstations. In *Design, Automation and Test in Europe (DATE)*, 2010.

[112] Rainer Doemer, Weiwei Chen, Xu Han, and Andreas Gerstlauer. Multi-core parallel simulation of system-level description languages . In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2011.

[113] Weiwei Chen, Xu Han, and Rainer Doemer. Out-of-order parallel simulation for ESL design. In *Design, Automation and Test in Europe (DATE)*, 2012.

[114] Giovanni Funchal and Matthieu Moy. Modeling of time in discrete-event simulation of systems-on-chip. In *IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, 2011.

[115] L.G. Murillo, J. Eusse, J. Jovic, and Rainer Leupers. Synchronization for hybrid MPSoC full-system simulation. In *ACM/IEEE Design Automation Conference (DAC)*, 2012.

[116] Kuen-Huei Lin, Siao-Jie Cai, and Chung-Yang Huang. Speeding up SoC virtual platform simulation by data-dependency-aware synchronization and scheduling . In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2010.

[117] Yu-Fu Yeh, Shao-Lun Huang, Chi-An Wu, and Hsin-Cheng Lin. Speeding Up MPSoC virtual platform simulation by Ultra Synchronization Checking Method. In *Design, Automation and Test in Europe (DATE)*, 2011.

[118] Nicolas Blan and Daniel Kroening. Race analysis for SystemC using model checking. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2010.

[119] Rauf S. Khaligh and Martin Radetzki. Modeling constructs and kernel for parallel simulation of accuracy adaptive TLMs . In *Design, Automation and Test in Europe (DATE)*, 2010.

[120] Parisa Razaghi and Andreas Gerstlauer. Automatic timing granularity adjustment for host-compiled software simulation. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2012.

[121] Kun Lu, Daniel Mueller-Gritschneder, Wolfgang Ecker, Volkan Esen, Michael Velten, and Ulf Schlichtmann. An Approach toward Accurately Timed TLM+ for Embedded System Models . In *edaWorkshop*, May 2011.

[122] Kun Lu, Daniel Mueller-Gritschneder, and Ulf Schlichtmann. Accurately Timed Transaction Level Models for Virtual Prototyping at High Abstraction Level. In *Design, Automation and Test in Europe (DATE)*, March 2012.

[123] Kun Lu, Daniel Mueller-Gritschneder, and Ulf Schlichtmann. Analytical Timing Estimation for Temporally Decoupled TLMs Considering Resource Conflicts. In *Design, Automation and Test in Europe (DATE)*, March 2013.

[124] W. Ecker, V. Esen, and M. Velten. TLM+ modeling of embedded HW/SW systems. In *Design, Automation and Test in Europe (DATE)*, 2010.

[125] Kun Lu, Daniel Mueller-Gritschneder, and Ulf Schlichtmann. Memory Access Reconstruction Based on Memory Allocation Mechanism for Source-Level Simulation of Embedded Software. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*, January 2013.

[126] Kun Lu, Daniel Mueller-Gritschneder, and Ulf Schlichtmann. Hierarchical Control Flow Matching for Source-level Simulation of Embedded Software. In *IEEE International Symposium on System-on-Chip*, October 2012.

[127] Kun Lu, Daniel Mueller-Gritschneder, and Ulf Schlichtmann. Fast Cache Simulation for Host-Compiled Simulation of Embedded Software. In *Design, Automation and Test in Europe (DATE)*, March 2013.

[128] Daniel Mueller-Gritschneder, Kun Lu, Erik Wallander, Marc Greim, and Ulf Schlichtmann. A Virtual Prototyping Platform for Real-time Systems with a Case Study for a Two-wheeled Robot. In *Design, Automation and Test in Europe (DATE)*, March 2013.

[129] Kun Lu, Daniel Mueller-Gritschneder, and Ulf Schlichtmann. Removal of Unnecessary Context Switches from the SystemC Simulation Kernel for Fast VP Simulation. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, July 2011.

[130] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass., 1975.

[131] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The Program Depedence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319.–349, 1987.

[132] Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. A Simple, Fast Dominance Algorithm. *Software Practice and Experience*, 4:1–10, 2001.

[133] T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Computer science. Springer, 2011. ISBN 9781441988348.

[134] pycparser. http://code.google.com/p/pycparser, 2011.

[135] Stefan Stattelmann, Alexander Viehl, Oliver Bringmann, and Wolfgang Rosenstiel. Reconstructing Line References from Optimized Binary Code for Source-Level Annotation. In *Forum on specification and Design Languages*, 2010.

[136] Plasma CPU Model. http://www.opencores.org/projects/mips, 2011.

[137] Embecosm Limited. The OpenCores OpenRISC 1000 Simulator and Tool Chain: Installation Guide., 2008.

[138] K. Pathak, J. Franch, and S. K. Agrawal. Velocity and position control of a wheeled inverted pendulum by partial feedback linearization. *IEEE Transactions on Robotics*, 2005.

[139] Manoj Ariyamparambath, Denis Bussaglia, Bernd Reinkemeier, Tim Kogel, and Torsten Kempf. A Highly Efficient Modeling Style for Heterogeneous Bus Architectures. In *IEEE International Symposium on System-on-Chip*, 2003.

[140] Keith D Cooper, Timothy J Harvey, and Ken Kennedy. A simple, fast dominance algorithm. *Software Practice and Experience*, 4:1–10, 2001.