Precise Software Timing Simulation Considering Execution Contexts 20

Oliver Bringmann, Sebastian Ottlik, and Alexander Viehl

Abstract
Context-sensitive software timing simulation enables a precise approximation of software timing at a high simulation speed. The number of cycles required to execute a sequence of instructions depends on the state of the microarchitecture prior to the execution of that sequence, which in turn heavily depends on the preceding instructions. This is exploited in context-sensitive timing simulation by selecting one of multiple pre-calculated cycle counts for an instruction sequence based on the control flow leading to a particular execution of the sequence. In this chapter, we give an overview of this concept and present our context-sensitive simulation framework. Experimental results demonstrate that our framework enables an accurate and fast timing simulation for software executing on current commercial embedded processors with complex high-performance microarchitectures without any slow, explicit modeling of components such as caches during simulation.

Acronyms

BB	Basic Block
BLS	Binary-Level Simulation
CFG	Control-Flow Graph
DSE	Design Space Exploration
FPGA	Field-Programmable Gate Array
ICFG	Interprocedural Control-Flow Graph

O. Bringmann (✉)
Wilhelm-Schickard-Institut, University of Tübingen, Tübingen, Germany

Embedded Systems, University of Tübingen, Tübingen, Germany
e-mail: oliver.bringmann@uni-tuebingen.de

S. Ottlik • A. Viehl
Microelectronic System Design, FZI Research Center for Information Technology, Karlsruhe, Germany
e-mail: ottlik@fzi.de; viehl@fzi.de

© Springer Science+Business Media Dordrecht 2017
S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_21

MIPS	Million Instructions Per Second
PSTC	Path Segment Timing Characterization
RTL	Register Transfer Level
SIMD	Single Instruction, Multiple Data
SLS	Source-Level Simulation
TDB	Timing Database
VIVU	Virtual Inlining and Virtual Unrolling
WCET	Worst-Case Execution Time

Contents

20.1 Introduction

Hardware/software cosimulation is an essential tool during the codesign process. In principle, well-known techniques such as Register Transfer Level (RTL) simulation could be used to create simulations that provide a level of detail (e.g., cycle-by-cycle exact timing) that is sufficient for most purposes. However, the low simulation performance makes them impractical in many scenarios while this level of detail is unnecessary for many use cases. For example, a developer analyzing software performance is likely not directly interested in the contents of the branch history table of the processor, but only indirectly in its influence on the time required to execute a particular function. A simulation used by this developer does not need to simulate the branch history table if it can still accurately approximate its influence on performance. Therefore, simulation performance can be improved by raising the abstraction level, if a reasonable accuracy can be maintained.

In this chapter, we discuss an approach that is capable of accurately approximating the timing influence of the full microarchitecture, yet at the same time allows for a highly efficient simulation as no components of the microarchitecture are explicitly simulated. This is achieved by preestimating target code instruction block timings in a Path Segment Timing Characterization (PSTC). For each block, timings are differentiated by the execution paths and thus the instructions executed before a particular execution of the block. So-called contexts serve as an abstraction of these paths. Context essentially enable an approximate consideration of the different possible states of the microarchitecture before and its impact on the execution time of a particular block execution. We implemented this approach in a highly flexible framework that support both static and dynamic methods for the PSTC. Results are stored in a common format, the so-called Timing Database (TDB). We integrated our timing simulation with QEMU [3], a fast functional simulator for binary code, which enables us to support a wide range of commercially embedded processors.

This chapter is organized as follows: In Sect. 20.2 we introduce fundamental concepts of context-sensitive timing simulation and give an overview of the current state of the art. In Sect. 20.3 we present our simulation framework and detail the main aspects of the context-sensitive timing simulation. Experimental results are presented in Sect. 20.4. A discussion of our main findings is provided in Sect. 20.5. This chapter is concluded in Sect. 20.6.

20.2 Context-Sensitive Simulation Fundamentals

In this section we first introduce the basic idea of context-sensitive timing simulation in Sect. 20.2.1. The necessary background on control-flow graphs and context mappings is introduced in Sects. 20.2.2 and 20.2.3. Related work is summarized in Sect. 20.2.4. We discuss various challenges that must be handled by context-sensitive simulation approaches in Sect. 20.2.5.

20.2.1 Basic Idea of Context-Sensitive Simulation

Binary-Level Simulation (BLS) and Source-Level Simulation (SLS), which are, respectively, discussed in ▶ Chaps. 18, "Multiprocessor System-on-Chip Prototyping Using Dynamic Binary Translation" and ▶ 19, "Host-Compiled Simulation", are well-known techniques to simulate software functionality in hardware/software cosimulation at a high performance. However, providing accurate approximations of time at this abstraction level is challenging. Typically software timing simulations consider the processor pipeline, branch prediction, and the memory subsystem – caches in particular. However, a detailed simulation of all relevant components and their interactions is extremely slow. While this approach is common in interpretative instruction-set simulation (e.g., Gem5 [4]), it degrades overall simulation performance of BLS to a level that makes the functional simulation performances nearly irrelevant [6, 22]. Such a detailed timing simulation is therefore not reasonable, if BLS or the significantly faster SLS is utilized to achieve a significant boost in simulation performance.

a

```
                 dependency

         ld r0, [iteration_count]  ; load iteration count into r0
loop:    sub r0, r0, 1             ; decrement r0
         …
         cmp r0, 0                 ; if r0 is greater than 0
         jg <loop>                 ; jump to loop
```

b

```
ld r0, [iteration_count]
sub r0, r0, 1            ; delayed until load complete
…
cmp r0, 0
jg <loop>
sub r0, r0, 1            ; not delayed
…
cmp r0, 0
jg <loop>
```

Fig. 20.1 Example of execution order/path dependent execution time. (**a**) Target code. (**b**) Instruction sequence executed by processor

A reasonable trade-off between timing simulation accuracy and overhead over a purely functional simulation can be achieved by advancing time by a number of processor cycles for each execution of an instruction block, for example, as discussed by Tach, Tamiya, Kuwamura, and Ike [26] for BLS. Simulation performance is improved by analyzing the processing of an instruction block by the processor pipeline only once. The results of this analysis are reused on each execution of a block. Further influences of the microarchitecture (e.g., caches) are accounted for by modeling the respective components during the simulation and applying timing penalties for certain events (e.g., cache misses). Usually, these events are assumed to not occur during the pipeline analysis, and only a single timing is calculated per block.

A major drawback of these approaches is that influences of preceding blocks (e.g., instruction dependencies) on the execution time of a block are not reflected due to the use of a single value and an isolated consideration of each block. The problem is illustrated in Fig. 20.1 for a simple loop. Before the loop is entered, the iteration count is loaded into a register, which is used as loop counter. The loop counter register is decremented by the first instruction in the loop body. In the first loop iteration, this instruction is directly preceded by the load in the instruction sequence executed by the processor and can only be processed once the loaded is completed. In subsequent iteration this is not an issue. The timing of the instruction block containing the decrement, thus, depends on which block is executed beforehand: the block preceding the loop (i.e., containing the load) or the loop block (i.e., itself). This issue can significantly impair simulation accuracy and becomes even more significant for the deeper and wider pipelines found in high-performance processors.

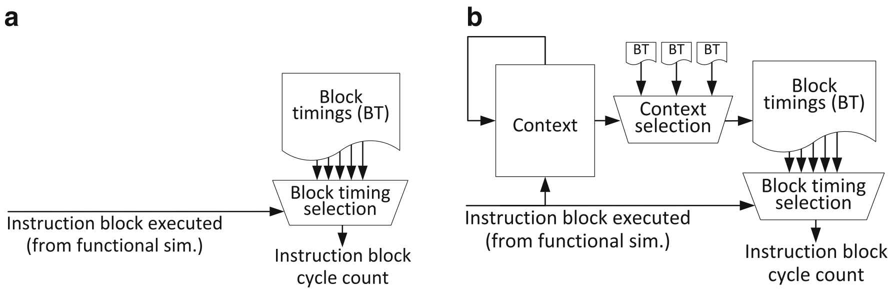

a
Block timings (BT)
Block timing selection
Instruction block executed (from functional sim.)
Instruction block cycle count
b
BT
BT
BT
Context
Context selection
Block timings (BT)
Block timing selection
Instruction block executed (from functional sim.)
Instruction block cycle count

Fig. 20.2 Quintessential difference between context-insensitive and context-sensitive timing simulation. (**a**) Context-insensitive simulation. (**b**) Context-sensitive simulation

Context-sensitive simulation aims to improve simulation accuracy by enabling a consideration of the influence of preceding instructions on the timing of an instruction block. Multiple values are provided for each block, one of which is selected for a particular execution of a block based on the current context. The context essentially abstracts the precedingly executed instructions. This difference between context-sensitive and context-insensitive block-level timing simulation is illustrated in Fig. 20.2. A context-insensitive simulation selects the timing purely based on the currently executing instruction block, whereas a context-sensitive simulation can additionally consider the execution history by means of the current context.

The use of context-sensitive block timings can provide further advantages, because they can accurately reflect the influence of caches and branch prediction without an explicit model: Firstly, while online cache and branch prediction models are much faster than an online model of the full processor, they can still lead to a significant overhead [13, 26]. Secondly, a fixed penalty for cache misses and branch mis-predictions essentially corresponds to assuming the pipeline is halted in these situations, whereas, for example, in a typical super-scalar processor only instruction that depend on the result of the instruction that caused a cache miss will be stalled.

The main drawback of context-sensitive simulation is the increased complexity of calculating block cycle counts. All published approaches, therefore, require an analysis step to derive these values before an actual simulation can be performed. However, such an PSTC, that is an ahead-of-time partial characterization of timing properties of execution path segments, is also necessary for other simulation approaches (cf. ▶ Chap. 21, "Timing Models for Fast Embedded Software Performance Analysis"). For context-sensitive simulations, the PSTC needs to calculate possible execution paths through the software and characterize segments of these paths regarding their timing properties. PSTC results are then utilized to simulate the actual timing of the simulated path during the simulation. The PSTC can be static or dynamic. A static PSTC considers multiple (or even all) possible executions of the software in a single analysis. A dynamic PSTC considers only a single execution of the software at a time.

20.2.2 Control-Flow Graphs

The control flow of a program can be formally expressed by a Control-Flow Graph (CFG). A CFG is a directed graph where nodes represent Basic Blocks (BBs) and edges represent possible control flow between them. Basic blocks can only be entered at the beginning and left at the end by control-flow changes. They usually represent particular code sequences, but sometimes additional empty nodes that do not represent actual code are also included. Here, we are only concerned with CFGs of binary code without such empty nodes. However, CFGs are also used on other abstraction levels such as source code.

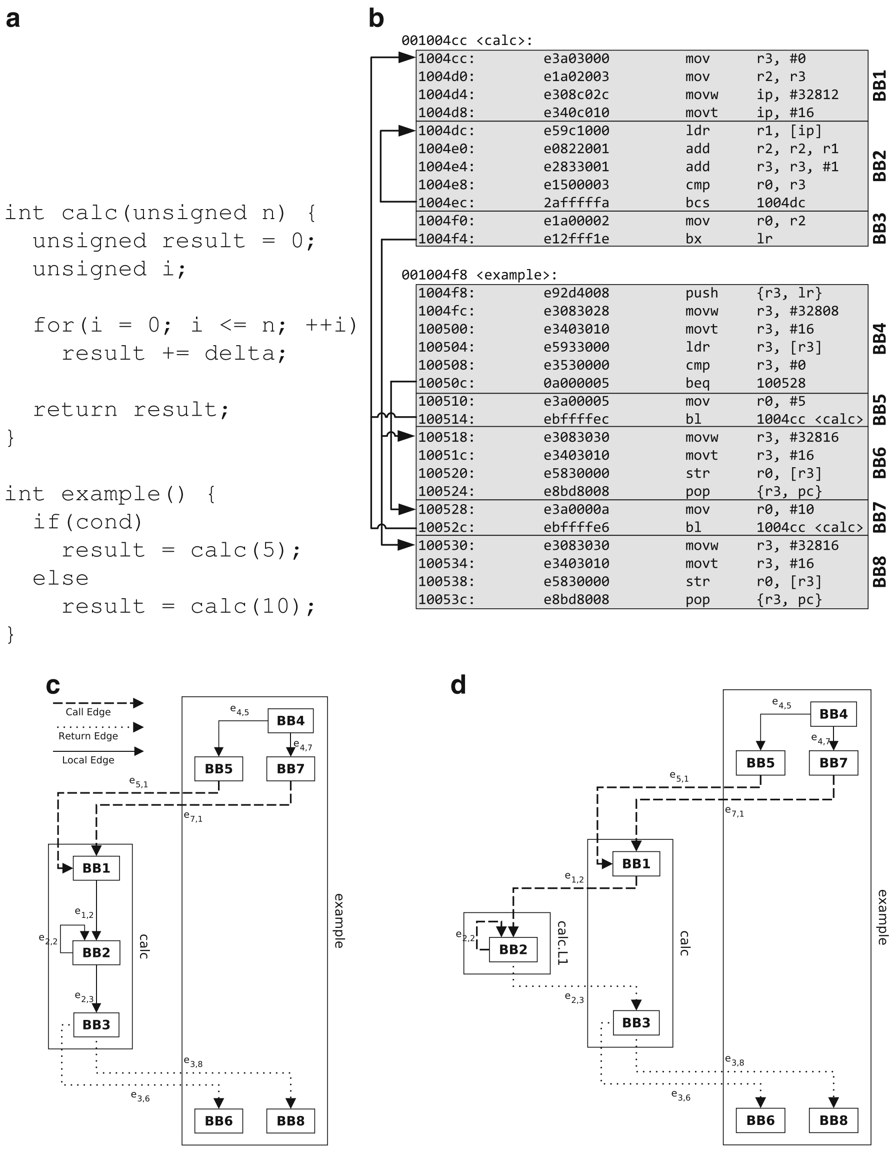
In a binary-level CFG, basic blocks cover sequences of processor instructions, as shown in Fig. 20.3b. Branch instructions establish the control flow and thus basic block boundaries. In a basic block, branches can only occur as the last instruction. Furthermore, instructions that can be executed after a branch (i.e., branch targets and instructions directly after a branch) can only occur as the first instruction.

For a context-sensitive timing simulation, basic knowledge of the CFG for the simulated software is required and usually reconstructed from binary code. For some approaches, a simple subdivision into blocks and edges is sufficient. However, more complex approaches operate on an Interprocedural Control-Flow Graph (ICFG), where blocks are grouped into routines, and edges carry additional semantic information to express various types of control flow (e.g., calls and returns). Additionally, loops can be represented as independent, recursive routines, and loop-back edges are represented as recursive routine calls. These concepts are illustrated in Fig. 20.3 for the source code shown in Fig. 20.3a. Typically, the function itself would be represented in a ICFG as shown in Fig. 20.3c, whereas with loop extraction a ICFG as shown in Fig. 20.3d is used. In such an ICFG, the term *routine* may also refer to extracted loops. The transformation of loop to recursive routines is only performed for so-called natural loops. In practice, this is not a significant restriction, as this condition is fulfilled for typical code of reasonable quality [14].

20.2.3 Context Mappings

The set of instruction sequences that can be executed before a particular block of instructions can be expressed by the set of ICFG paths that lead to it. In the presence of loops and recursions, this set can be infinite. Hence, it is infeasible to provide a separate timing value for each path leading to a particular block of instructions. Therefore context-sensitive timings are only provided for a finite number of subsets of the set of control-flow paths. These subsets are referred to as *contexts*. A function that maps a control-flow path to a context is referred to as a *context mapping*. This function must be chosen such that the set of contexts is finite.

This consideration and the concept of contexts is well known in the domain of static program analysis, where alternative context concepts are also known (e.g., parameter-value-based contexts). Here, we restrict our discussion to the two mappings that have been applied to timing simulation in the literature: The n-block



a

```
int calc(unsigned n) {
  unsigned result = 0;
  unsigned i;

  for(i = 0; i <= n; ++i)
    result += delta;

  return result;
}

int example() {
  if(cond)
    result = calc(5);
  else
    result = calc(10);
}
```

b

```
001004cc <calc>:
1004cc:     e3a03000    mov     r3, #0                BB1
1004d0:     e1a02003    mov     r2, r3
1004d4:     e308c02c    movw    ip, #32812
1004d8:     e340c010    movt    ip, #16
1004dc:     e59c1000    ldr     r1, [ip]              BB2
1004e0:     e0822001    add     r2, r2, r1
1004e4:     e2833001    add     r3, r3, #1
1004e8:     e1500003    cmp     r0, r3
1004ec:     2afffffa    bcs     1004dc
1004f0:     e1a00002    mov     r0, r2                BB3
1004f4:     e12fff1e    bx      lr

001004f8 <example>:
1004f8:     e92d4008    push    {r3, lr}              BB4
1004fc:     e3083028    movw    r3, #32808
100500:     e3403010    movt    r3, #16
100504:     e5933000    ldr     r3, [r3]
100508:     e3530000    cmp     r3, #0
10050c:     0a000005    beq     100528
100510:     e3a00005    mov     r0, #5                BB5
100514:     ebffffec    bl      1004cc <calc>
100518:     e3083030    movw    r3, #32816            BB6
10051c:     e3403010    movt    r3, #16
100520:     e5830000    str     r0, [r3]
100524:     e8bd8008    pop     {r3, pc}
100528:     e3a0000a    mov     r0, #10               BB7
10052c:     ebffffe6    bl      1004cc <calc>
100530:     e3083030    movw    r3, #32816            BB8
100534:     e3403010    movt    r3, #16
100538:     e5830000    str     r0, [r3]
10053c:     e8bd8008    pop     {r3, pc}
```

Fig. 20.3 Example of control-flow representations. (**a**) Source code. (**b**) Target code subdivided into basic blocks. (**c**) Plain ICFG. (**d**) ICFG with extracted loop

mapping and the Virtual Inlining and Virtual Unrolling (VIVU) mapping. A simplified comparison of the two mappings is shown in Fig. 20.4. The example shows a control-flow path through the example CFG from Fig. 20.3 to the second iteration of the loop. We will use this example in the following to explain both mappings.

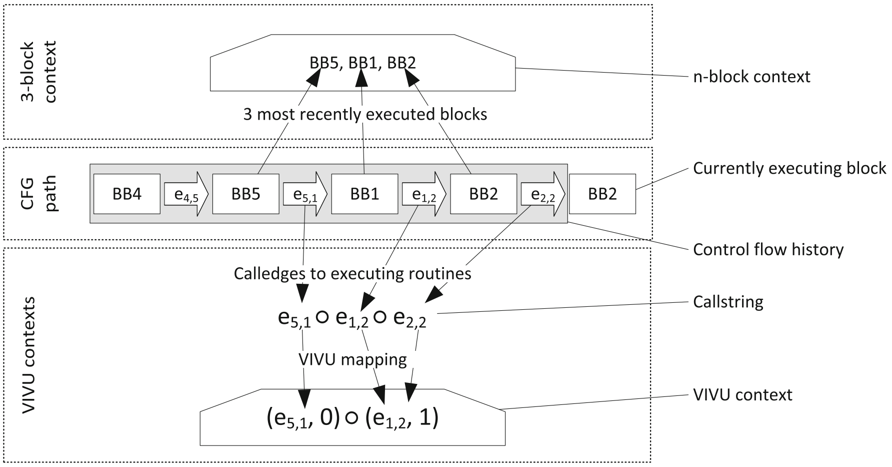


Fig. 20.4 Example of context mappings (cf. Fig. 20.3)

As VIVU only considers call edges, it only differentiates contexts by a subset of the preceding blocks. Therefore, VIVU can consider much longer histories, but at a smaller granularity. This has the advantage that the timing influence of states that are maintained much longer, such as cache contents, can be reflected accurately without explicit modeling. In practice this means that a simulation using VIVU does not necessarily require an online cache model, whereas it is necessary when using an n-block mapping. However, this advantage comes at the cost of being less sensitive to more local influences such as instruction dependencies. We would expect a combination of VIVU and an n-block mapping to provide the overall best results, but currently find such an extension unnecessary based on the very high simulation accuracy provided by the VIVU mapping in our experiments.

20.2.3.1 n-Block Mapping

For the n-block mapping, the control-flow path to the currently executing basic block is expressed as a sequence of executed blocks. In an n-block mapping, this path is simply truncated to the last n elements to form a context. In more practical terms, contexts are differentiated by the n most recently executed basic blocks.

This mapping has two main advantages: Firstly, it is relatively simple and therefore straightforward to apply. Secondly, it always includes all most recently executed instructions to differentiate contexts and can therefore provide a high accuracy for local timing influences, such as instruction dependencies.

The main drawback of this mapping is that only small values of n can be chosen, as otherwise an excessive number of contexts has to be considered. This limits the ability of this mapping to accurately reflect global timing influences such as caches. For example, the path from Fig. 20.4 would be extended by BB2 for each further loop iteration. Even for the largest published $n = 16$ [20], only the first 16 iterations of the loop could be differentiated, as afterward the context would always be a sequence of 16 instances of BB2.

20.2.3.2 VIVU Mapping

The VIVU mapping [14] is an approach to enhancing the accuracy of interprocedural program analysis with a special consideration of loops. It has been successfully applied in static Worst-Case Execution Time (WCET) analysis of binary code [9]. Our presentation of the VIVU mapping is based on the notation used by Theiling in his dissertation [27]. A sequence of a followed by b is denoted as $a \circ b$. The empty sequence is denoted as ε.

For VIVU the control-flow path to the currently executing basic block is expressed as a sequence of edges. Compared to the n-block mapping, VIVU reduces the accuracy for local timing influences, but can accurately reflect global influences. VIVU is usually applied to an ICFG, where loops are represented as recursive routines (cf. Sect. 20.2.2). Thereby, besides function calls, a call string also reflects loop iterations counts. As shown in Fig. 20.4, VIVU does not consider the whole path, but only edges that represent calls to routines that are currently executing (i.e., have not returned yet). These edges are referred to as *call edges*, and the sequence of call edges is referred to as *call string*. For example, after the loop is left, the call string would be $e_{5,1}$, as $e_{1,2}$ and $e_{2,2}$ represent calls to the loop.

VIVU maps a call string to a sequence of so-called context links. A *context link* is a pair (e, x) of a call edge e and a recursion count of x. A sequence of context links is referred to as a *VIVU context*. The call string is mapped to a VIVU context by iteratively extending the empty context ε by each call edge in the call string using the *VIVU context connector* $\oplus_n^k$ and starting with the first (i.e., oldest) call edge. The context ε is empty in terms of a context as a sequence of context links. In terms of a context as a set of control-flow paths it is the exact opposite, that is the context for all control-flow paths. The call string $e_{5,1} \circ e_{1,2} \circ e_{2,2}$ in Fig. 20.4 is mapped to $((\varepsilon \oplus_n^k e_{5,1}) \oplus_n^k e_{1,2}) \oplus_n^k e_{2,2})$. The context connector $\oplus_n^k$ applies one of two rules. It can be adapted by the parameters n and k to influence the granularity of the contexts. A particular VIVU mapping is denoted as VIVU(n,k). n is an upper limit for the recursion count, while k limits the number of context links in a context. Both k and n can be unbounded, which is, respectively, denoted as $n = \infty$ or $k = \infty$. Their exact functionality is discussed below.

Which of the two rules is applied depends on whether the connected edge (right-hand side) has the same destination as any edge in a context link in the to-be-extended context (left-hand side). An edge where this condition is true w.r.t. the to-be-extended context is referred to as recursive.

If the edge is not recursive, the context is extended by a context link with that edge and a recursion count of 0. For example, $(\varepsilon \oplus_n^k e_{5,1}) \oplus_n^k e_{1,2} = (e_{5,1}, 0) \oplus_n^k e_{1,2} = (e_{5,1}, 0) \circ (e_{1,2}, 0)$. If the length of the new context would exceed k, it is shortened to the last k elements. With $k = 1$, for example, $(e_{5,1}, 0) \oplus_n^k e_{1,2} = (e_{1,2}, 0)$. The context link $(e_{5,1}, 0)$ is dropped to meet the length restriction. For $k = 0$, all call strings are mapped to ε.

If the edge is recursive, the context is truncated to end with the context link with the same destination, and its recursion count is incremented by 1. For example, $((e_{5,1}, 0) \circ (e_{1,2}, 0)) \oplus_n^k e_{2,2} = (e_{5,1}, 0) \circ (e_{1,2}, 1)$, as $e_{1,2}$ and $e_{2,2}$ have the same destination. The recursion count is further bounded to the value of n. Theiling [27] denotes the recursion count in this case as $\top$, we do not make this distinction. For

$n = 1$, $((e_{5,1}, 0) \circ (e_{1,2}, 1)) \oplus_n^k e_{2,2} = (e_{5,1}, 0) \circ (e_{1,2}, 1)$, because the recursion count is limited to 1. As a more complex example, assume the ICFG in Fig. 20.3d contained an additional edge $e_{2,1}$, because the function `calc` calls itself recursively from within the loop. A typical context would be calculated as $((e_{5,1}, 0) \circ (e_{1,2}, 1)) \oplus_n^k e_{2,1} = (e_{5,1}, 1)$. The context link $(e_{1,2}, 1)$ is dropped because the context is truncated to end with the context link containing $e_{5,1}$, as it has the same destination as $e_{2,1}$.

In some cases the interaction between rules and parameters can become complex. For $k = 1$, $((e_{5,1}, 0) \oplus_n^1 e_{1,2}) \oplus_n^1 e_{2,1} = (e_{1,2}, 0) \oplus_n^1 e_{2,1} = (e_{2,1}, 0)$. Because $(e_{5,1}, 0)$ is dropped in the first application of the context connector, $e_{2,1}$ is not considered recursive anymore in the second application of the context connector.

To ensure a finite set of contexts when $n = \infty$, an edge-specific limit $n_{\max}(e)$ is required and applied in the same way as n. In a static PSTC, this value can be derived automatically in some cases and has to be specified by a user otherwise. In a dynamic PSTC it can formally be assumed to be equal to the number of traversals of an edge that in a particular analysis and thus ignored in practice.

20.2.4 Related Work

Here we provide a brief introduction to closely related work, a more comprehensive overview can be found in our previous publications [16, 17, 25], which also served as a foundation of this chapter and ▶ Chap. 21, "Timing Models for Fast Embedded Software Performance Analysis".

Chakravarty, Zhao, and Gerstlauer [5] presented a SLS with a 1-block mapping. A similar, but more general, concept for a BLS for an n-block mapping with $n \leq 16$ has been presented by Plyaskin, Wild, and Herkersdorf [19–21]. Stattelmann [23] applied VIVU contexts in a SLS. In our framework we also apply VIVU.

The main difference between the use of VIVU in our simulation framework and Stattelmann's simulation is that his approach only uses the mapping implicitly by essentially tracing the path through a so-called expanded supergraph [14]. The expanded supergraph is an ICFG where a node is copied for each context that node can be executed in. While this approach simplifies an efficient implementation, the inherent loss of the meaning of a context restricts the handling of contexts. Most importantly, control flow that was not considered during the preceding static PSTC is not handled in Stattelmann's simulation.

20.2.5 Challenges in Context-Sensitive Simulation

While context-sensitive simulation has many advantages to offer, the increased complexity in the required understanding of a program also poses several challenges that we discuss in the following.

20.2.5.1 Incomplete Data

At its core a context-sensitive simulation requires knowledge of the program control flow during the PSTC. It can be obtained by static or dynamic techniques. In both cases discovering the full control flow is nontrivial: Dynamic analysis techniques are restricted to the control flow of the observed executions, which in turn depends on the used input. While static techniques do not suffer from this issue, it is complicated by computed and indirect branches (e.g., function pointers) and asynchronous changes of control flow (e.g., interrupts). In practice achieving a sufficiently high coverage is nontrivial in a single dynamic or static PSTC.

Three types of coverage issues can be differentiated:

Code	If the PSTC fails to discover parts of the code the respective nodes are missing from the reconstructed control flow graph. For example, for a program with an interrupt handler, a static PSTC will not discover the handler code, as typically no explicit control flow to the handler is contained in a program. In a dynamic PSTC on the other hand, the handler could be missing if the interrupt is not raised during the observation.
Control-Flow	A PSTC can also fail to discover edges between nodes. For example, when it fails to discover all possible call sites for a function pointer.
Context	A dynamic PSTC is unlikely to execute a program in all contexts that can occur during a simulation. However, this issue can also occur in static PSTC as a consequence of limited code or control-flow coverage.

20.2.5.2 Context Granularity

Even for small benchmarks, the large number of possible control-flow paths can lead to an excessive number of contexts. An increased number of contexts typically reduces the simulation performance, due to the increased memory footprint of the timing data and the increased complexity of context selection. On the other hand, with only few contexts, a simulation will provide a reduced accuracy.

The number of contexts is therefore a trade-off between simulation accuracy and performance. Furthermore, different contexts can be identical in terms of timing, as contexts only indirectly reflect the state of the microarchitecture. Most context mappings can be parametrized to influence the granularity of contexts. Furthermore it is possible to automatically remove unnecessary contexts, without reducing simulation accuracy.

20.2.5.3 Execution Variations

Context-sensitive timing simulations achieve a good trade-off between simulation accuracy and performance by preestimating many influences on software timing. A drawback of this approach is that variations between the simulation and the PSTC are not trivial to handle. Variations in the following areas can be considered:

Input	If dynamic PSTC is used, it is desirable that the simulation is not restricted to the executions that were observed during the analysis – in particular for changes to program input including hardware stimulations such as interrupts. This can firstly lead to the aforementioned coverage issues, but can cause subtle changes in timing behavior.

Hardware	As large, or all, parts of the hardware components that influence software timing are considered before the simulation, different simulation cannot reflect different hardware configurations without a reexecution of the PSTC.
Code	If the analyzed code changes, the analysis results become stale. Current context-sensitive approaches require a full reexecution of the PSTC in this case. In the future more advanced analyses may allow an adaption of the exiting timing data.

20.3 Context-Sensitive Simulation Framework

An overview of our context-sensitive timing simulation framework is shown in Fig. 20.5. As first step, context-sensitive timings of instruction blocks in a given binary code have to be obtained by a PSTC based on a detailed microarchitectural model. Our framework is very flexible in what kind of analysis technique and model can be used. Currently, we support static analysis using abstract models intended for WCET analysis by abstract interpretation and dynamic analysis using actual hardware implementations of a processor. Additionally, we plan to support further models, such as Field-Programmable Gate Array (FPGA)-based processor prototypes and detailed models in Gem5 [4], in the future. Therefore, even though our framework requires a detailed model, it is retargetable with a low effort if a model is available. Supported targets of our framework so far include the ARM7TDMI, LEON3, ARM Cortex-M3, ARM Cortex-A9, and ARM Cortex-A15 processors.

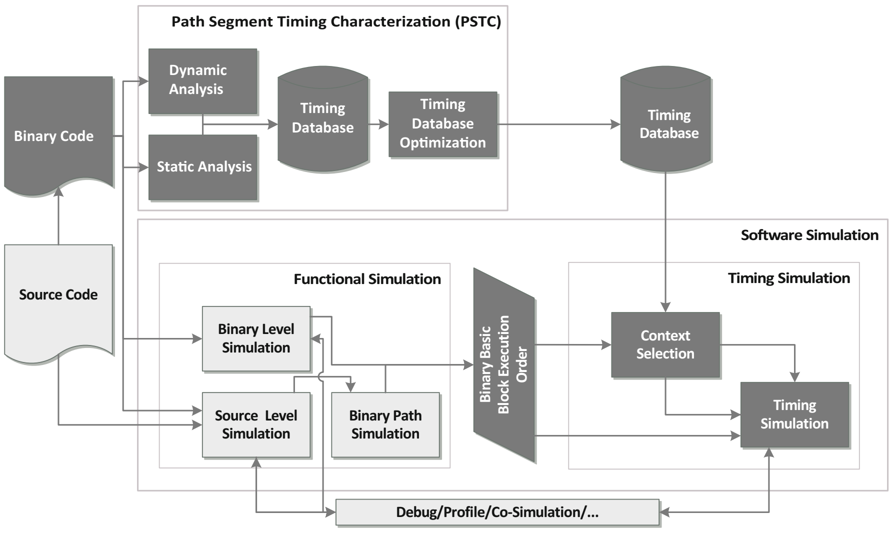


Fig. 20.5 Simulation framework overview (*grey areas*: beyond article scope)

The PSTC produces an initial TDB. The TDB includes an ICFG of the analyzed binary code and context-sensitive timing data for instruction blocks. We make use of VIVU [14] (cf. Sect. 20.2.3.2) as context mapping. This initial TDB could in principle already be used in a simulation. However, it is usually improved by applying further optimizations that lead to both, improved simulation performance and accuracy. Furthermore, it is possible to merge multiple TDBs.

After a TDB has been generated, it can be reused in multiple simulations. As a functional simulator, we support BLS based on QEMU [3] and are currently working toward the integration of an experimental SLS tool. Both produce a trace of the binary basic blocks executed in the simulation to drive the timing simulation.

Based on the basic block execution order, contexts are selected from the TDB. The current context is used to choose a timing value for a simulated block execution by which to advance simulation time. The combined functional and timing simulation for the software is then usually combined with a hardware simulation in order to provide a cosimulation. However, other use cases such as a direct profiling or debugging of the simulated software are also supported. Currently, the framework only targets single core processors. Multi- and many-core systems can be modeled in a cosimulation using multiple instances of the software simulation. A feedback between the timing and functional simulation is also possible. For example, when interrupts influence the control flow in the functional simulation and are triggered by a simulated peripheral based on the simulated timing.

20.3.1 Timing Database Contents

The TDB consists of two main parts: an ICFG and context-dependent cycle counts for instruction blocks.

20.3.1.1 Control Flow

In the TDB the control flow of the target binary code is expressed by an ICFG with loops extracted, as outlined in Sect. 20.2.2 and shown in Fig. 20.3d. To support a tracking of the current context during the simulation, edges can be marked as a return or call edge. While a call edge always represents a single call, a return edge can represent multiple returns. For example, if the program shown in Fig. 20.3 contained a return within the loop, the corresponding edges from BB2 to BB6/BB8 would represent two returns. In our current implementation, the number of returns always corresponds to the number of routines left by the return, including all directly recursive calls of these routines. More details on the handling of returns are given in Sect. 20.3.3.1.

20.3.1.2 Context-Sensitive Timings

Context-sensitive timings are differentiated by the outgoing edges of a block and an associated context. More formally, the stored timings can be thought of as a function timing(e, c) that returns the timing of block b_x, when it is left via an edge $e = (b_x, b_y)$ to execute b_y in context c. An example is shown in Table 20.1 and explained in Sect. 20.3.1.3.

Table 20.1 Example of context-sensitive timings for a loop (cf. Fig. 20.3)

Context	Timings		Valid for call string	
	$e_{2,2}$	$e_{2,3}$	$e_{5,1} \circ e_{1,2}$	$e_{7,1} \circ e_{1,2} \circ e_{2,2}$
ε	5	7	Yes	Yes
$(e_{1,2}, 1)$	3	5	No	Yes, highest precedence
$(e_{1,2}, 0)$	6	6	Yes	Yes
$(e_{7,1}, 0) \circ (e_{1,2}, 0)$	5	6	No	Yes
$(e_{5,1}, 0) \circ (e_{1,2}, 0)$	3	8	Yes, highest precedence	No

In classic use of VIVU contexts for static WCET analysis, the set of known control-flow paths from an analysis starting point (e.g., the `main` function) is partitioned into contexts. Essentially, there is exactly one corresponding context for each path, which can be calculated by applying a particular VIVU mapping. However, this approach is not practical for our framework, as it should provide a best-effort approximation for paths that were not considered during TDB generation due to the limitations of the used analysis technique (cf. Sect. 20.2.5).

For example, a TDB generated from timings observed in an execution of the program in Fig. 20.3 where `calc` is called via BB5, there may not be a context for calls via BB7. This would prevent the simulation of an execution where `calc` is called via BB7 instead of BB5. In this simple example, a reasonable alternative would be a context that does not differentiate between the two call edges to `calc`. In the general case, a flexible relationship between paths and contexts must be considered to achieve the goals of our framework.

Arbitrary contexts may be combined in the TDB, and a context is considered valid for a control-flow path, if the path could be mapped to the context for arbitrary VIVU parameters n and k and an arbitrary analysis starting point. In other words, a VIVU context in the TDB can be thought of as a condition that must be fulfilled by an arbitrary suffix of the simulated path to make the respective context-sensitive timings a valid choice. The drawback of this approach is that it becomes necessary to define which of the valid contexts should be selected during simulation.

20.3.1.3 Context Precedence

As outlined in the preceding section, a TDB may – and typically does – contain multiple contexts that are valid for a path or more specifically the respective call string. One of these contexts has to be selected during the simulation such that the simulated timing is as accurate as possible. In our framework this process is based on two assumptions: First, in terms of the set of control-flow paths represented by a context, a context that represents a smaller set is more likely to be accurate because variations in the timing of fewer paths are less likely. Second, a context that represents paths that share recent control flow is likely more accurate, as it is more likely that recent control flow (i.e., more recently executed instructions) still have an impact on the timing of the currently executing block. For example, if two precedingly executed blocks caused memory to be loaded into the cache, the memory that was loaded by the more recently executed block is more likely to still reside in the cache.

An example of context-sensitive timings and the precedence of various contexts is shown in Table 20.1 for the loop from the example program in Fig. 20.3. As can be seen each context provides timings for the outgoing edges of the blocks in the routine (BB2). As every call string may be mapped to ε, it is a valid choice for every call string, but also takes the lowest precedence. Thus it can serve as a fallback if no other context is available. For the call string $e_{5,1} \circ e_{1,2}$, the context $(e_{5,1}, 0) \circ (e_{1,2}, 0)$ is the preferred choice, as it is more specific than $(e_{1,2}, 0)$ or ε. For the call string $e_{7,1} \circ e_{1,2} \circ e_{2,2}$, the preferred context is $(e_{1,2}, 1)$. While $(e_{7,1}, 0) \circ (e_{1,2}, 0)$ can be considered equal in terms of the first assumption, it represents less recent control flow and thus has a lower precedence. Roughly it can be said that the choice between $(e_{7,1}, 0) \circ (e_{1,2}, 0)$ and $(e_{1,2}, 1)$ is a choice between a context that reflects $e_{7,1}$ and one that reflects $e_{2,2}$. Since $e_{2,2}$ ultimately represents more recently executed instructions, $(e_{1,2}, 1)$ is the preferred context.

Since a context may represent an infinite number of paths, some of which may not have been discovered during TDB generation, using these assumptions directly to establish context precedence is not a good solution. Furthermore the efficiency of the context lookup has also been considered. The context selection scheme of the simulation framework is outlined in Sect. 20.3.3.3.

20.3.2 Timing Database Generation

The TDB can be generated using either a static or a dynamic PSTC of the application binary code for the target platform. Compared to the static PSTC, the main advantage of the dynamic PSTC-based generation is that it can provide highly accurate results even for complex system architectures. However, this requires appropriate program stimuli during the dynamic PSTC, which is not necessary for the static PSTC.

20.3.2.1 Dynamic PSTC

The dynamic PSTC-based TDB generation requires a timed trace of a program execution as an input. The trace must allow a full reconstruction of the program control flow (i.e., the exact sequence of executed instructions), whereas timing data is only required for every execution of a basic block. In practice, such traces can be obtained from on-chip tracing units available for many embedded processors. For example, they are supported by ARM CoreSight [2] and the Nexus 5001 standard [15]. Note that such tracing units typically only provide time stamps for executed branch instructions. More specifically, ARM style traces provide a time stamp for take and non-taken branches, whereas Nexus style traces only provide time stamps for taken branches. Therefore, some time stamps may cover multiple basic blocks. In this case the timing of individual block executions is interpolated.

To construct a TDB, two passes are performed over a trace. During the first pass, an ICFG is extracted. The extracted graph is analyzed to identify loops and subsequently transformed to the form described in Sect. 20.2.2. During the second pass, the graph is traversed based on the trace, and the observed timings are stored in the TDB for the current context.

If the VIVU mapping is limited (i.e., $n \neq \infty$ or $k \neq \infty$) or the traced execution contains indirect recursions, it is possible that multiple observations are stored in the same context. In this case an average is stored in the TDB. Additionally, we store the number of observations underlying a timing value as this information is required in the context generalization optimization. This optimization is applied after the TDB generation to enable an accurate simulation if the simulated control flow differs from the traced control flow. A description is given in Sect. 20.3.2.3.

20.3.2.2 Static PSTC

For the static PSTC, we adopt well-known techniques from the domain of static WCET analysis. More specifically, we employ a static control-flow analysis to extract the ICFG, on which block timings are estimated using abstract interpretation [7]. In our current implementation, these steps are performed by Absint aiT [1], which is a commercial tool for WCET analysis, from which we read intermediate results. The intermediate results are translated to a TDB, and we typically execute multiple analyses and merge the resulting TDBs, as a single analysis may not provide sufficient coverage. This is, for example, the case for typical programs with interrupt handlers: As the analysis only discovers explicit control-flow changes, an analysis of the main function does not cover interrupt handlers and vice versa. The TDB generator could, for example, execute two distinct analyses in aiT, one starting at the main function and another at the interrupt handler, and merge their results in a single TDB.

As aforementioned, the control-flow representation and context mapping used in the TDB are based on research into WCET analysis. More specifically, they are based on their use in aiT and earlier work [23, 24] by our groups that essentially directly made use of the same intermediate results. However, for the TDB we make two small, but significant changes: First, aiT only considers discovered control-flow paths for each context, whereas the TDB assumes a context is valid for any control-flow path that can be mapped to it. As a result, when simulating control flow beyond the control flow considered during the analysis, the worst-case timing guarantees made by the analysis cannot be maintained. We expect that changing the static analysis to enable a generic worst-case timing simulation is feasible, but this topic is beyond the scope of our current research activities. Second, there is a difference between the ICFGs used by the static analysis and the ICFGs used in simulation that we describe in the following.

The ICFGs used during the static analysis include special empty nodes to avoid the complexity of handling various special cases. For example, calls are represented by a call and a return node that follow/precede the basic block a call is made from/returns to. As these nodes contain no instructions, their execution cannot be registered during a simulation, and we therefore do not include them in the TDB. During the translation of analysis results to a TDB, we map path between nonempty blocks that contain only empty blocks to a single edge. A resulting edge can therefore cross call and/or multiple returns. This translation is not sensible for arbitrary graphs, but in our experience the analysis only produces graphs where this translation can be handled with a reasonable effort.

20.3.2.3 Optimization

Various optimizations can be applied to the TDB that improve simulation performance and/or accuracy. Performance improvements can be achieved by removing unnecessary contexts. Thereby the overall memory footprint of a simulation is reduced and the context lookup algorithm used in the simulation, which we describe later in Sect. 20.3.3.3, can terminate early. Accuracy improvements on the other hand are achieved by synthesizing new contexts and/or adding new data to existing contexts. These new contexts may then be used to simulate the timing of paths that were not considered during the PSTC and thus may not be represented by the context produced by the PSTC. This step is especially critical for the dynamic PSTC, as only a single execution is considered. In practice it enables a simulation of the timing of a program for multiple different inputs (and thus potentially different paths) based on the timing observed for one input. Typically, the resulting TDB is smaller after all optimizations have been applied. All optimizations are performed on a per-routine basis. They are constructed such that timing data remains unchanged for the executions considered during TDB generation.

Removing Unnecessary Contexts

Contexts can be removed from a TDB under some conditions. For example, assume a TDB contained only the contexts $(e_{5,1}, 0) \circ (e_{1,2}, 0)$ and $(e_{5,1}, 0) \circ (e_{1,2}, 1)$ for the loop from the example in Fig. 20.3. As the contexts are not differentiated by the context link $(e_{5,1}, 0)$, the contexts $(e_{1,2}, 0)$ and $(e_{1,2}, 1)$ can be used, respectively. This is possible because these shorter contexts will be selected for each path the original contexts would be selected for, and as no other contexts exists, these were the only paths considered during analysis. Furthermore, if the context-sensitive data for both contexts is identical, $(e_{1,2}, 1)$ can be removed based on the same rationale. As $(e_{1,2}, 0)$ is the only remaining context, it can simply be replaced by ε.

In general, these optimizations are expressed by two rules that define which contexts can be considered for removal without considering the associated data and separate consideration if the data of two contexts is mergeable. The data for two contexts is considered mergeable, if cycle counts are identical for all edges that are present in both sets of data. In more practical terms, two contexts are mergeable if cycle count lookup that was successful in either context before the merge returns the same result after the merge. Note that a context without any associated data may be merged with any other context. The first rule is referred to as shorten call strings. It defines that a context $(e_1, r_1) \circ (e_2, r_2) \circ \cdots \circ (e_n, r_n)$ may be removed and merged with $(e_2, r_2) \circ \cdots \circ (e_n, r_n)$, if no other context $(e_x, r_x) \circ (e_2, r_2) \circ \cdots \circ (e_n, r_n)$ with $e_x \neq e_1 \wedge r_x \neq e_1$ exists. The second rule is referred to as merge leaf iterations. It defines that a context $(e_x, r_x) \circ \ldots$ may be removed and merged with $(e_x, r_y) \circ \ldots$, if $r_y < r_x$ and r_y is the largest value where this holds. As can be seen from the example in the beginning of this section, each rule can create a situation where the other rule can be applied. They are therefore applied in turn until a fixed point is reached.

Context Generalization

The prime motivation for the context generalization optimization is the incompleteness of a TDB created from a single program execution using the dynamic PSTC-based TDB generation. As example assume a TDB for the example in Fig. 20.3 was generated from the timings observed for an execution where `calc` is called from BB5. Such a TDB could only contain the contexts $(e_{5,1}, 0) \circ (e_{1,2}, 0)$ and $(e_{5,1}, 0) \circ (e_{1,2}, 1)$. If `calc` is called via BB7 in a simulation, a call string for the second iteration of the loop would be $e_{7,1} \circ e_{1,2} \circ e_{2,2}$. No context could be selected for this call string. A good alternative would be to use the data from context $(e_{5,1}, 0) \circ (e_{1,2}, 1)$, that is, use the timing of the second iteration when the call was from BB5 to simulate the timing of the second iteration when the call was from BB7. This could, for example, accurately reflect that timing under consideration that the instruction cache was already filled with the loop instructions in the first iteration.

For a general understanding of this optimization, it is helpful to consider contexts as sets of control-flow paths (i.e., the set of paths that could be mapped to the context). As a first step consider the valid contexts of a call string that was not considered during the initial TDB generation. Each of these contexts represents a set of control-flow paths that includes the paths represented by the call string. For some of these potential contexts, there exist other contexts in the TDB that represent a subset of a potential context, a condition that is always at least true for ε. For the example mentioned in the preceding paragraph, a valid context for the call string $e_{7,1} \circ e_{1,2} \circ e_{2,2}$ would be $(e_{1,2}, 1)$, of which the existing context $(e_{5,1}, 0) \circ (e_{1,2}, 1)$ is a subset. The best context to generate would be the one with the highest precedence as discussed in Sect. 20.3.1.3.

However, generating these contexts is not possible before the simulation, as not all call strings may be known, and could be very inefficient during simulation. Instead the context generalization synthesizes more general versions of the existing contexts. For each context $(e_1, r_1) \circ (e_2, r_2) \circ \cdots \circ (e_n, r_n)$, new contexts $(e_2, r_2) \circ \cdots \circ (e_n, r_n)$, $(e_3, r_3) \circ \cdots \circ (e_n, r_n)$ and so on until ε are generated. Each new context is associated with average timings of the underlying original contexts. The averages are weighted by the number of underlying observations, if a dynamic PSTC was used for the initial TDB generation.

An additional issue is also handled by this optimization: The timing for some edges may only have been observed for some of the original contexts. To ensure that all context contain a timing for all edges of a routine, missing values in original and synthesized contexts are filled in from the most specific context that contains a value for that edge and is a generalized version of the context with the missing data.

20.3.3 Simulation

To make use of the context-dependent timing, appropriate contexts have to be selected during a simulation. A naive approach to look up a context from the TDB during simulation would be to maintain the current call string during the simulation, and on each change find the valid context with the highest precedence in the TDB by applying various VIVU(n, k) mappings. However, this approach is unlikely to achieve a high simulation performance.

To enable a robust, flexible, and efficient context lookup, under consideration of the discussion of context precedence outline in Sect. 20.3.1.3, we developed the following approach: During simulation, similarly to a call string, we maintain a so-called *intermediate string*. Contexts in the TDB are structured in a per-routine *context tree*. They can be looked up by the *dynamic context selection* algorithm.

During simulation the execution of target binary code basic blocks must be registered as an event. In BLS, we perform this by instrumenting the first instruction of every basic block. In SLS, this can be achieved by so-called path simulation code [24], which reconstructs the execution path through the target binary code based on the source code execution path (cf. ▶ Chap. 19, "Host-Compiled Simulation"). Time is advanced for each executed block by the cycle count stored in the TDB for the taken outgoing edge in the current context.

20.3.3.1 Intermediate String

An intermediate string is similar and in simple cases identical to a VIVU context for $n = \infty$ and $k = \infty$. More specifically, it is a sequence of context links (i.e., a pair of a call edge and a recursion count) with an unbounded length and unbounded recursion counters. On each traversal of a call edge, the intermediate string is updated. Usually, a new context link with a recursion count of 0 is appended to the intermediate string. If an edge is a direct recursion (i.e., has the same destination as the last context link element) and is marked as a non-returning call (e.g., it is a loop back edge), no element is appended and the recursion counter of the last element is increased instead. This process is similar to the VIVU context connector, but prevents a loss of information that may be required for a context lookup or a future update of the intermediate string. In contrast to maintaining a call string, it has the advantage that no additional memory is required for loop iterations, which are the most common case of recursive calls.

An edge can only represent a single call, as every natural loop has a distinct header (control-flow structures that could intuitively be considered two distinct loops with a shared header are formally considered a single natural loop) and, in practice, functions always have a few prologue instructions. However, an edge can represent multiple returns, for example, when a loop contains a function return. In the TDB an edge can therefore be a call edge (or not) and additionally represent an arbitrary number of returns, which shall be performed before the call. The number of returns is not the number of call edges that should be removed from a call string, as this number may be statically unknown (e.g., in the aforementioned case of a return from within a loop). Instead it is defined as the number of context links that shall be removed from the intermediate string.

20.3.3.2 VIVU Context Tree

In the TDB contexts for a routine are structured as a tree, where each node corresponds to a particular VIVU context but may be empty/unused. The children of a node represent more specific versions of contexts of their parent. The root of the tree represents the context ε and therefore is a valid context for any control-flow path. Each step from a node to a direct child is associated with a context link, where the context of the child is given by prepending the context link to the context of the parent. The full context of a node is therefore the sequence of context links for the path from the node to the root (the path from the root to the node is the reversed context). An example for the context tree is given in the next section.

20.3.3.3 Dynamic Context Selection

The dynamic context selection algorithm is used to select a particular context for an intermediate string from a context tree. Essentially, the context tree is ordered such that during a descend into the tree, each step to a child node leads to a more specific context for the intermediate string. As the path from the root to a node is the revered context for that node, the intermediate string is also processed in reverse. Usually the child to descend into is selected based on the next context link in the reversed intermediate string until no further valid child is available. An exception are recursions over multiple call edges and returnable direct recursions, where multiple children may be a valid choice and a scoring scheme is used to select a particular context. In the following, we first describe the lookup algorithm without consideration for this special case and provide a separate description afterward.

The lookup starts at the root node, which corresponds to the context ε, which is a valid choice for any intermediate string. For descending into child nodes, the intermediate string is considered in reverse order. A child node is considered a candidate for descending, if the edge of the context link for the step to the child node and the edge for the currently considered context link of the intermediate string are identical. Often (e.g., for loops) multiple candidates with different recursion counts exist. In this case the candidate with the highest recursion count is selected, that is, less than or equal to the recursion count of the currently considered context link of the intermediate string. This process is repeated until no further candidate for descending can be found. The deepest encountered node with valid timing data is selected as current context, usually this is the last node the lookup descended to. In the worst case, when no descend from the root is possible, the context ε is selected if it contains valid timing data. In our current implementation, we ensure this is always the case by an optimization of the TDB (cf. Sect. 20.3.2.3).

An example for the context tree and dynamic context selection is shown in Fig. 20.6. Both intermediate strings are evaluated in reveres. For $(e_{5,1}, 0) \circ (e_{1,2}, 0)$ all intermediate strings are evaluated, and the context links to the child nodes and from the intermediate string are identical. For $(e_{5,1}, 0) \circ (e_{1,2}, 2)$ on the other hand, the child node for $(e_{1,2}, 1)$ is selected, as there is no child node with a higher recursion count. Furthermore, the intermediate string is not evaluated further, as there are no child nodes to descend into.

To handle recursions over multiple call edges and returnable direct recursions, the algorithm checks whether other context links of the (unreversed) intermediate string preceding the context link has an identical destination. If this is the case, it preforms additional descends for the other elements and a recursion count that is the sum of both recursion counts plus one (to account for the increment that would have been performed by the VIVU mapping but was not performed for the intermediate

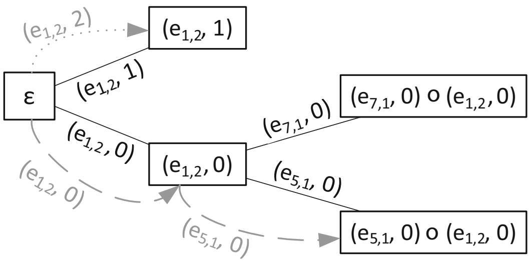


Fig. 20.6 Example of context tree (*black*) for the example contexts from Fig. 20.1 and context selection (*gray*) for intermediate strings $(e_{5,1}, 0) \circ (e_{1,2}, 0)$ (*dashed*) and $(e_{5,1}, 0) \circ (e_{1,2}, 2)$ (*dotted*)

string). The depth is not a sufficient criterion to select the most specific context in this case. Therefore, each encountered node is scored based on the sum of all recursion counts of all context links that were evaluated during the descend to this node plus the number of evaluated context links. The node with the highest score is selected and if multiple nodes have the same score the first encountered node is preferred.

20.3.3.4 Fallback Strategies

It is unlikely that a TDB provides perfect coverage of the simulated control flow for various reasons: For example, changes in control flow due to interrupts (e.g., a timer interrupt driving a preemptive scheduler) are not expressed in an ICFG. Another reason can be imperfect matchings in a SLS, which can lead to blocks missing from the path simulation code. The simulation framework includes various fallback mechanisms to handle such cases. First, the TDB provides a fallback cycle count for each block, which we currently calculate by averaging all cycle counts for the outgoing edges of a block. This value can be used if a block execution order is simulated for which no corresponding edge exists in the ICFG stored in the TDB. Second, if such an execution order is simulated the intermediate string may be incorrect, because the missing edge may have been a call or return. This is handled by removing elements from the intermediate string until the edge of the last context link is an incoming edge of the currently executing routine. In the worst case the intermediate string is cleared. For example, a program with preemptive scheduling is roughly simulated as follows: The program is simulated normally as outlined above. When control flow is diverted to the timer interrupt handler, the fallback value for the currently executing block is used to advance simulation time and the intermediate string is cleared. During the subsequent execution of the scheduler, the intermediate string is rebuild providing progressively more accurate timings. Once the interrupt handler returns to a different or the previously executing thread, a similar sequence is repeated.

20.4 Experimental Results

In this section we present experimental results for our timing simulation framework. First, we establish a baseline regarding simulation accuracy based on results for small benchmarks. Afterward, we discuss case studies for more complex applications. Due to space restrictions, we do not provide an in-depth description of the experiments here, more details can be found in our preceding publications [16, 17].

20.4.1 Benchmarks

As benchmarks we use several programs from the Mälardalen WCET benchmark [10] collection. Each of these benchmarks focuses on specific control-flow structures, for example, typical computations such as matrix multiplications or implementation patterns such as state machines. This has the drawback that the benchmark collection does not provide representative workloads. However, it allows an evaluation of the relationship between program structure and simulation properties.

Some benchmarks were excluded for various reasons: Several benchmarks had very short execution times or were even completely optimized by the compiler. Furthermore we had to adapt benchmarks to allow a variation of inputs for our experiments using a dynamic PSTC. For these experiments we selected benchmark where such an adaptation was feasible with a reasonable effort. This adaptation was primarily necessary, as our framework can simulate the timing exactly without input variations.

We performed experiments with these benchmarks for TDB generation using static PSTC for an ARM Cortex-M3 system and using dynamic PSTC for an ARM Cortex-A9 system. The experiments using a static PSTC were evaluated regarding both simulation accuracy and performance, whereas the experiments using a dynamic PSTC were only evaluated regarding simulation accuracy.

20.4.1.1 Simulation Accuracy

Figure 20.7 shows the deviation of simulation results from hardware measurements for the Mälardalen benchmarks using four different TDBs and two different hardware configurations. The TDBs inf and opt represent the intended use of our simulation framework: a TDB that was generated using a VIVU(∞,∞) mapping in the static PSTC and, in the case of opt, further optimization of the TDB. Noctx represents a simulation that does not itself make use of contexts but relies on the results of a static PSTC that does use contexts. Whereas the zero case represents a simulation where contexts are used in neither.

Many programs can be simulated accurately by any TDB, but only the context-sensitive simulations provide accurate simulation results in all cases. The difference is more pronounced in the configuration with two wait states. Here, the increased latency of the flash the benchmarks are executed from makes the timing behavior more complex, which can only be reflected accurately by the context-sensitive simulation.

Table 20.2 shows minimum and maximum simulation errors when using a dynamic PSTC to generate a TDB. As the main issue with a dynamic analysis-based TDB generation is a variation of program inputs, we adapted a subset of the Mälardalen benchmarks to enable parameter variations. For this evaluation we chose a wide range of values for each parameter (e.g., size parameters in a range

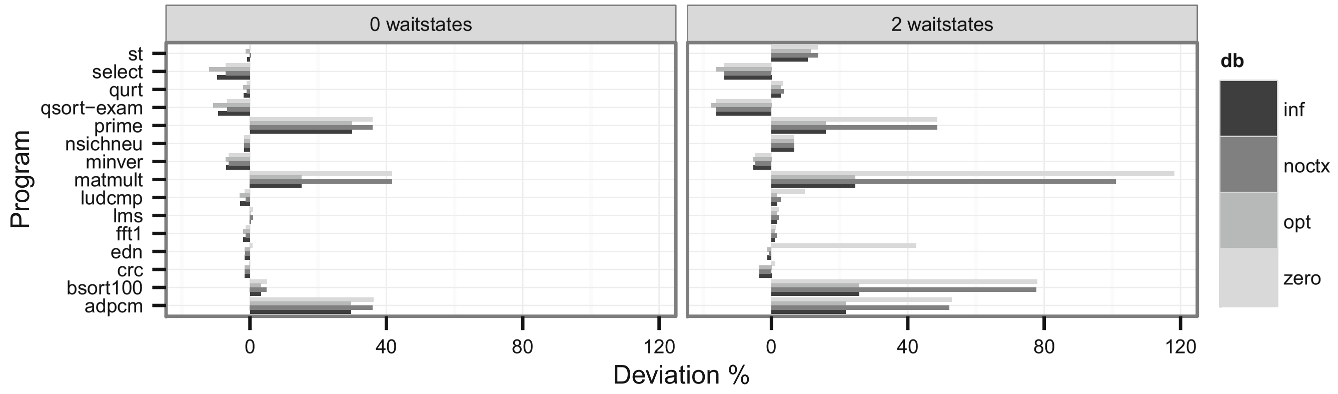


Fig. 20.7 Deviation of simulation results from hardware measurements for Cortex-M3 when using a static PSTC

Table 20.2 Simulation error when using a dynamic PSTC

Program	Param.	Avg./max. error %					
		All[a]		Single[b]		Pair[c]	
matmult	Seed	0.0	0.0	0.0	0.0	0.0	0.0
	Matrix size	12.5	28.8	8.2	16.2	4.1	12.0
crc	Input size	2.4	14.2	0.8	3.9	0.4	1.2
	Input values	0.2	0.5	0.1	0.4	0.0	0.2
adpcm	Input amp.	0.9	1.8	0.7	1.7	0.0	0.1
	Input freq.	1.0	3.1	0.2	0.6	0.1	0.4
	Num. samples	0.6	1.5	0.4	0.7	0.0	0.0
bsort100	Seed	0.0	0.0	0.0	0.0	0.0	0.0
	Array size	15.2	84.3	0.7	2.1	0.2	0.8
qsort-exam	Seed	4.2	7.0	2.8	4.0	2.8	4.0
	Array size	8.2	21.9	3.4	6.5	2.4	6.1
prime	Tested prime	11.1	29.0	6.3	12.8	2.3	5.3
select	Seed	0.7	1.8	0.5	0.9	0.3	0.6
	Array size	8.0	42.7	2.2	9.2	0.6	1.3
All	None	0.0	0.0	0.0	0.0	0.0	0.0

[a]All: All TDBs
[b]Single: on average most accurate TDB
[c]Pair: Most accurate pair of TDBs when choosing the more accurate one of the pair per program

such that the working set size exceeds the L1 cache size for some values but not for others), traced executions for each parameter value and generated a TDB from every trace. Afterward, each TDB was used to simulate executions for all parameters. We calculate mean and maximum simulation errors from these simulation for multiple scenarios of selecting a TDB for a simulation. In all we include all TDBs, which is similar to a user choosing tracing inputs arbitrarily. In single we selected the single, on average best TDB, which represents the best result a user could achieve when he deliberately selects tracing inputs. The pair selection strategy is similar, but additionally considers that a user may generate multiple TDBs – two in this case – to further improve simulation accuracy.

The control flow in most benchmarks does not change significantly when the working set size is identical in TDB generation and simulation. In these cases even using arbitrary inputs during tracing leads to a high simulation accuracy. An extreme exception to these rules is the `prime` benchmark, as it makes heavy use of a low-level math function to perform modulo operations. This function essentially contains a large switch statement, if the TDB is generated for shorter executions the switch statement is not covered sufficiently. In this case and also variations of the working set size, a more careful selection of tracing input is required. In most cases using two TDBs provides a further accuracy improvements.

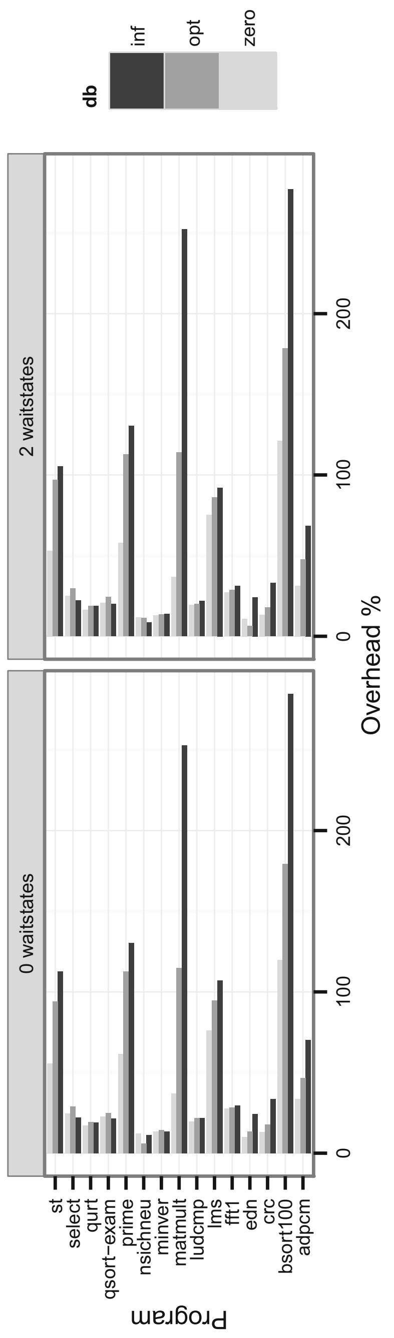


Fig. 20.8 Simulation overhead

20.4.1.2 Simulation Performance

Figure 20.8 shows the overhead of the timing simulation for the Mälardalen benchmarks in the experiments for the Cortex-M3 processor using a static PSTC. The overhead is independent of the used hardware configuration, as during the simulation the only difference between the two cases is the timing values stored in the TDB. While not shown in the figure, this has also the consequence that there is no significant difference in simulation performance for a TDB generated using a dynamic or static PSTC. A performance gain of over factor 10 can be achieved utilizing SLS and an automaton based encoding of the timing data [18].

20.4.2 Case Studies

Benchmarks are usually significantly smaller and less complex than real applications, but our simulation becomes more complex with application size and complexity. We therefore put a focus on using real code in the evaluations of our simulation framework. Here we present a few selected case studies.

20.4.2.1 eCos

Our eCos case study explores the capabilities of the simulation to exceed the limitations of a static PSTC used during TDB generation. For this purpose we ported the embedded operating system eCos [8] to our reference hardware that was also used in the experiments for the Mälardalen benchmarks using static PSTC. We configured eCos with support for preemptive multi-threading and crafted an application that executed two concurrent instances of the matmult benchmark.

The preemption creates a feedback between the temporal and the functional behavior of the program, as changes in software performance lead to additional thread switches. This factor complicates the static PSTC used to generate a TDB. In particular an accurate simulation would be impossible without dynamic context selection and the fusion of results from multiple analyses.

When using an optimized TDB, our simulation can approximate the hardware execution with an error of less than 0.25%, whereas a simulation and analysis without context lead to an error of over 70%. This large error is not only the result of the decreased accuracy of the block-level timings but also the resulting change in the functional simulation due to additional thread switches. This demonstrates the importance of an accurate low-level timing simulation in the presence of feedback between the functional and the timing behavior of an application.

While the difference in the functional behavior prevents an exact calculation of the timing simulation overhead, the difference in simulation performance is comparable to the results for the Mälardalen benchmarks: The context-sensitive simulation with an optimized TDB achieves about 22 Million Instructions Per Second (MIPS), while a simulation without timing and thus a sequential execution of both threads achieves 39 MIPS.

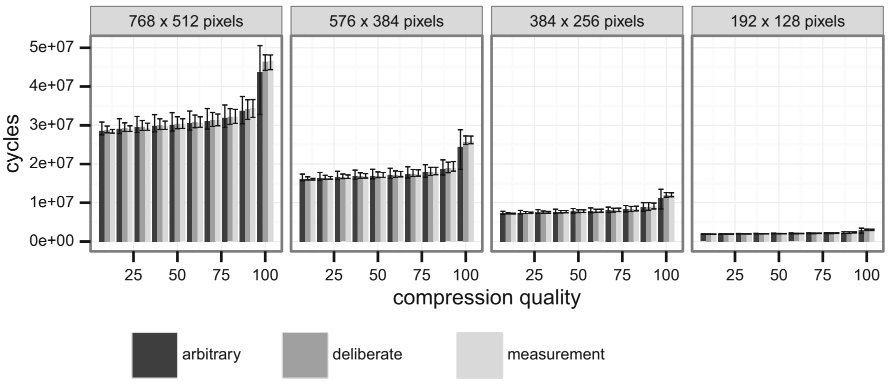


Fig. 20.9 Min/mean/max number of cycles for execution of image compression as simulated using arbitrarily and deliberately selected tracing inputs compared to hardware measurements

20.4.2.2 Image Compression

As an example for highly optimized software, we investigated the simulation accuracy for the libjpeg-turbo [12] image compression library on a Cortex-A9, which makes use of Single Instruction, Multiple Data (SIMD) instruction set extensions. We created an application that compresses images from a collection by Kodak [11] at various compression quality settings. To further vary the image size, we rescaled some of the images to obtain four groups of constant size with five images each. TDBs were generated using a dynamic PSTC and explored the relationship between tracing input and simulation accuracy.

If an arbitrary input is used in tracing, the simulation error is 2.07% on average and at most 30.63%. If more care is taken by making sure tracing and simulation image size are identical and the special case of a compression quality of 100 is handled separately, the error can be reduced to 1.18% on average and at most 11.66%. Figure 20.9 shows a comparison of simulation results with hardware measurements. As can be seen, the simulation accurately characterizes the application timing, but for the case of arbitrary tracing inputs suggests a wider variation.

20.4.2.3 Advanced Driver Assistance

As an example for a complex, modern embedded application we experimented with a video based circular traffic sign recognition on the same platform used in the image compression application. The application processes each image in two stages. First, in the segmentation stage circles are detected. In the subsequent classification stage these circles are classified as a particular traffic signs or no sign. For our experiments we varied two segmentation parameters that influence the number of circles that go into the classification stage and must be tuned in practice to achieve a reasonable trade-off between application performance and recognition accuracy. Furthermore we used several input images.

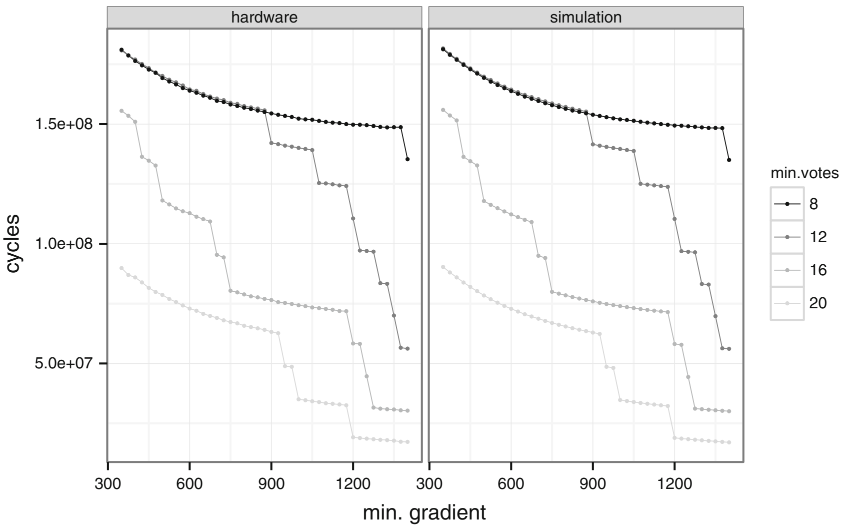


Fig. 20.10 Subset of simulation results for advanced driver assistance application compared to hardware measurements

If an arbitrary input is used in tracing, the simulation error is 0.98% on average and at most 7.55%. If more care is taken by making sure tracing and simulation images are identical or at least very similar, the error can be reduced to 0.27% on average and at most 2.06%. Figure 20.10 shows a comparison of simulation results with hardware measurements. As can be seen, the simulation slightly idealizes the application timing, which is more erratic on hardware.

20.5 Discussion

Context-sensitive software timing simulation offers many advantages over other approaches to software timing simulation. However, the need for a PSTC of the software code and the complexity of the control-flow models as well as context mappings lead to a number of limitations. In this section we list advantages and limitations of context-sensitive simulation at the current state of the art.

20.5.1 Advantages

While the simulation at its core is still driven by events at a higher level of abstraction than individual processor cycles, the use of contexts allows an accurate approximation of the flow of instructions through the processor pipeline from a timing perspective. As a consequence, context-sensitive timing simulation can offer an extremely high accuracy, in particular, when used with a dynamic PSTC that can also consider application behavior for typical workloads.

Furthermore, if VIVU is used as context mapping, contexts can also approximate cache effects. This makes online cache models unnecessary, which otherwise can significantly reduce simulation performance. Furthermore, both caches and the pipeline can be considered simultaneously during the PSTC, which enables an accurate consideration of their interactions that is not possible in a context-insensitive simulation. This applies to instruction caches in particular, as their impact on application timing is mainly influenced by the control flow. However, our experimental results demonstrate that it is also possible to accurately model the timing influence of data and unified caches if a dynamic PSTC is used. It remains to be seen how far these advantages can be maintained in multi- and many-core systems. For example, if coherent caches are used the timing of instructions executed on one core can be influenced by those executed on another core.

Another interesting advantage is offered by our simulation framework if a dynamic PSTC by hardware tracing is used. As the timing information is extracted from hardware, a software model of the hardware is not needed during the PSTC. Such a model is not only hard to construct, but the necessary details of the microarchitecture are not always publicly available.

20.5.2 Limitations

The main limitation of context-sensitive simulation is the need to re-execute the PSTC when the software code or the system configuration (to the extent considered in the PSTC) changes. This complicates the application of context-sensitive simulation in Design Space Exploration (DSE) and for self modifying code.

For systems with self-modifying code, for example, those using just-in-time compilation (e.g., Java in Android) or dynamic code relocation (e.g., shared libraries under Linux) context-sensitive simulation is currently not possible. However, for DSE, Plyaskin, Wild, and Herkersdorf [21] demonstrated that limited variations in the hardware architecture can still be considered in a context-sensitive simulation.

20.6 Conclusions

In this chapter we discussed context-sensitive timing simulation for embedded software. This concept improves simulation accuracy by considering the control flow that leads to the execution of an instruction sequence to more accurately approximate the timing of that particular execution. Furthermore, it permits novel approaches, such as highly accurate simulations of application performance over a wide range of inputs based on hardware measurements for only few inputs. Moreover, as could be observed in our experimental results, context-sensitive simulation can remove the need for slow online models of caches and branch prediction.

These advantages come at the drawback of requiring a PSTC of the application code before a simulation. This analysis induces an overhead which is only recouped if multiple and/or longer running simulations are performed based on the results of one analysis. Furthermore the complexity of dynamic and static PSTC complicates a practical application.

Even though these drawbacks appear significant, we expect an increase in the practical relevance of context-sensitive simulations in future hardware/software cosimulations. We base this assessment on three factors: First, future products such as autonomous vehicles will require highly complex and performance-demanding software which must fulfill high safety standards. Cosimulations using context-sensitive software timing simulation can provide a valuable tool to developers of these products. Second, we expect that future research will simplify the application of the approach, for example, by simplifying the PSTC by combining static and dynamic analysis techniques. Third, other future and existing simulation techniques using block-level timings can exploit contexts to improve simulation accuracy, even if they only apply simple mappings that only consider the preceding block.

References

1. AbsInt Angewandte Informatik GmbH (2016) aiT: worst-case execution time analyzers. http://www.absint.com/ait
2. ARM: CoreSight: V1.0 architecture specification
3. Bellard F (2005) QEMU, a fast and portable dynamic translator. In: Proceedings of the USENIX annual technical conference (ATEC)
4. Binkert N, Beckmann B, Black G, Reinhardt SK, Saidi A, Basu A, Hestness J, Hower DR, Krishna T, Sardashti S, Sen R, Sewell K, Shoaib M, Vaish N, Hill MD, Wood DA (2011) The GEM5 simulator. ACM SIGARCH Comput Archit News 39(2):1–7
5. Chakravarty S, Zhao Z, Gerstlauer A (2013) Automated, retargetable back-annotation for host compiled performance and power modeling. In: Proceedings of the international conference on hardware/software codesign and system synthesis (CODES+ISSS)
6. Chiang MC, Yeh TC, Tseng GF (2011) A QEMU and SystemC-based cycle-accurate ISS for performance estimation on SoC development. IEEE Trans Comput Aided Des Integr Circuits Syst 30(4):593–606
7. Cousot P, Cousot R (1977) Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of 4th ACM SIGACT-SIGPLAN symposium principles of programming languages
8. eCos. http://ecos.sourceware.org
9. Ferdinand C, Heckmann R, Langenbach M, Martin F, Schmidt M, Theiling H, Thesing S, Wilhelm R (2001) Reliable and precise WCET determination for a real-life processor. In: Embedded software. Springer, Berlin/Heidelberg, pp 469–485
10. Gustafsson J, Betts A, Ermedahl A, Lisper B (2010) The Mälardalen WCET benchmarks – past, present and future. In: Lisper B (ed) WCET2010. OCG, Brussels, pp 137–147
11. Kodak test images. http://www.cipr.rpi.edu/resource/stills/kodak.html
12. libjpeg-turbo. http://libjpeg-turbo.virtualgl.org/
13. Lu K, Müller-Gritschneder D, Schlichtmann U (2013) Fast cache simulation for host-compiled simulation of embedded software. In: Design, automation & test in Europe, pp 637–642. doi:10.7873/DATE.2013.139

14. Martin F, Alt M, Wilhelm R, Ferdinand C (1998) Analysis of loops. In: Compiler construction. Lecture notes in computer science, vol 1383. Springer, Berlin/Heidelberg, pp 80–94. doi:10.1007/BFb0026424
15. Nexus 5001 Forum (2012) Nexus 5001 Forum Standard. IEEE-ISTO 5001-2012
16. Ottlik S, Stattelmann S, Viehl A, Rosenstiel W, Bringmann O (2014) Context-sensitive timing simulation of binary embedded software. In: Proceedings of the 2014 international conference on compilers, architecture and synthesis for embedded systems, CASES'14
17. Ottlik S, Borrmann JM, Asbach S, Viehl A, Rosenstiel W, Bringmann O (2016) Trace-based context-sensitive timing simulation considering execution path variations. In: 21st Asia and South Pacific design automation conference (ASP-DAC)
18. Ottlik S, Gerum C, Viehl A, Rosenstiel W, Bringmann O (2017) Context-Sensitive Timing Automata for Fast Source Level Simulation, In: Proceedings of Design, Automation & Test in Europe (DATE), 2017
19. Plyaskin R, Herkersdorf A (2010) A method for accurate high-level performance evaluation of MPSoC architectures using fine-grained generated traces. In: Architecture of computing systems – ARCS 2010. Lecture notes in computer science, vol 5974. Springer, Berlin/Heidelberg, pp 199–210
20. Plyaskin R, Herkersdorf A (2011) Context-aware compiled simulation of out-of-order processor behavior based on atomic traces. In: 2011 IEEE/IFIP 19th international conference on VLSI and system-on-chip (VLSI-SoC)
21. Plyaskin R, Wild T, Herkersdorf A (2012) System-level software performance simulation considering out-of-order processor execution. In: 2012 international symposium on system on chip (SoC)
22. Rosa F, Ost L, Reis R, Sassatelli G (2013) Instruction-driven timing CPU model for efficient embedded software development using OVP. In: 2013 IEEE 20th international conference on electronics, circuits, and systems (ICECS)
23. Stattelmann S (2013) Source-level performance estimation of compiler-optimized embedded software considering complex program transformations. Verlag Dr. Hut, München
24. Stattelmann S, Bringmann O, Rosenstiel W (2011) Fast and accurate source-level simulation of software timing considering complex code optimizations. In: 2011 48th ACM/EDAC/IEEE design automation conference (DAC)
25. Stattelmann S, Ottlik S, Viehl A, Bringmann O, Rosenstiel W (2012) Combining instruction set simulation and WCET analysis for embedded software performance estimation. In: 2012 7th IEEE international symposium on industrial embedded system (SIES), pp 295–298
26. Thach D, Tamiya Y, Kuwamura S, Ike A (2012) Fast cycle estimation methodology for instruction-level emulator. In: 2012 design, automation & test in Europe conference & Exhibition (DATE)
27. Theiling H (2002) Control flow graphs for real-time systems analysis. Dissertation, Universität des Saarlandes