Timing Models for Fast Embedded Software Performance Analysis

21

Oliver Bringmann, Christoph Gerum, and Sebastian Ottlik

Abstract

In this chapter, we give an overview on timing models which provide an abstract representation of the timing behavior for a given software. These models can be driven by a functional simulation based on the simulated control flow. As the timing model itself can reach a level of accuracy that is comparable to a classic timing simulation of the represented software, these approaches enable a fast yet accurate software performance analysis. In this chapter, we focus on the generation and structure of various models but also provide a brief introduction into their integration with a functional simulation. The presented approaches are targeting software executing on current and future system-on-chips with a wide range of embedded processors – including Graphics Processing Units (GPUs).

Acronyms

ASIP	Application-Specific Instruction-set Processor
CFG	Control-Flow Graph
CPU	Central Processing Unit
DSP	Digital Signal Processor
GPU	Graphics Processing Unit
MLoC	Million Lines of Code

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

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

C. Gerum
Embedded Systems, University of Tübingen, Tübingen, Germany
e-mail: christoph.gerum@uni-tuebingen.de

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

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_22

RTL	Register Transfer Level
SoC	System-on-Chip
VIVU	Virtual Inlining and Virtual Unrolling
VLIW	Very Long Instruction Word
WCET	Worst-Case Execution Time

Contents

21.1 Introduction

The ever increasing demand for new and more advanced features in products including embedded systems is leading to an increased relevance and complexity of embedded software to be used to realize these features. In addition, the growing computational demand of embedded software can only be served by more and more complex hardware architectures. Furthermore, for many embedded systems, software performance or even strict adherence to timing requirements is a serious factor, in particular for safety-critical products. Therefore, it is essential to integrate efficient timing and performance analysis methods and tools into the development process.

Software timing simulations are one approach to software timing and performance analysis. In contrast to a direct evaluation of software on the target hardware, simulations offer many advantages; the most important are reproducibility and greatly enhanced observability. In contrast to static analysis, using simulation is more natural to a developer and does not suffer from the overly conservative approximations necessary to avoid state space explosion in static analysis.

Besides these decisive advantages, there are a number of factors that must be considered for simulation to achieve a high practical value: Firstly, a detailed simulation of the underlying hardware greatly impairs simulation performance, up to a degree where a simulation is essentially useless in many use cases. Therefore, an abstraction is necessary. Secondly, there is an inherent loss of simulation accuracy when raising the abstraction level. However, this accuracy loss needs to be kept within acceptable limits; otherwise, simulation results become meaningless. Thirdly, creating a complex simulation can take a considerable modeling effort. Consequently, when choosing a simulation approach, the time-to-model must be considered.

In this chapter, we discuss timing models that enable high-performance yet accurate timing simulation. Essentially, these models provide a target platform-specific model of the temporal behavior of the embedded software based on the internal control flow. As the control flow of software programs can be tracked very efficiently during fast functional simulation, these model can enable timing simulation with a very low overhead. Since many factors that influence software timing can be represented as a direct or an indirect function of the control flow, a very high simulation accuracy can be achieved. While these models can be generated automatically, additional knowledge of the target platform is required and needs to be modeled and generated.

A wide range of sources can be applied during model generation, ranging from observing code execution on prototyping hardware, where modeling effort is negligible, to complex analytical models used in abstract interpretation.

Use cases for these models can be mainly found in the domain of embedded software engineering and systems development. For example, they can be applied in early evaluations of nonfunctional properties or to select components in heterogeneous multiprocessor systems-on-a-chip. The need to regenerate the model when the target platform changes limits their use in microarchitecture design.

This chapter is organized as follows: In Sect. 21.2, we discuss software performance analysis in general but with a particular focus on simulation and a comparison of control-flow-driven timing models to other approaches. Afterward we give an overview of three different approaches to control-flow-driven timing models. These models differ in the necessary knowledge of the target platform and the modeling effort. In Sect. 21.3, we discuss an approach that allows target platform independent performance characterization during simulation. Target-specific timing estimates are then generated after the simulation using an analytical target model. In Sect. 21.4, we discuss an approach that employs a generic platform model, that only has to be parametrized for a given target platform. In Sect. 21.5, we discuss an approach that employs a specific platform model, that requires a detailed description of the target hardware. In Sect. 21.6, we focus on a case study on modeling GPU-based architectures as these kinds of architectures cannot be represented by related models. In Sect. 21.8, we compare the presented approaches to modeling the generation of software performance models. The main conclusions of this chapter are summarized in Sect. 21.9.

21.2 Background

21.2.1 Challenges in Performance Evaluation of Modern Embedded Systems

As the main challenges for performance modeling and evaluation, we see software complexity and code reuse, hardware complexity and heterogeneity, as well as speed of performance analysis, which are briefly discussed in the following.

21.2.1.1 Software Complexity and Code Reuse

Current embedded software is large and complex. For example, Boeings 787 is estimated to contain 6.5 Million Lines of Code (MLoC) for its avionics and on-board support systems, while current premium class automobiles even run more than 100 MLoC [8]. This high software complexity leads to an extensive reuse of software components across different products. Considering these complexities, performance modeling techniques need to be applicable to large software projects. In cases of code reuse in different target systems, it would be beneficial if the timing models could be reused if a software component is reused in a different product.

21.2.1.2 Hardware Complexity and Heterogeneity

Many embedded systems contain complex hardware architectures. The ARM processors from the Cortex-A series contain superscalar pipelines often with out-of-order execution. But not only complex general purpose Central Processing Units (CPUs) are used in current embedded systems, there are also a wide variety of specialized processors like Digital Signal Processors (DSPs), Application-Specific Instruction-set Processors (ASIPs), or Very Long Instruction Words (VLIWs) processors. Moreover, in recent years, different Graphics Processing Unit (GPU)-based architectures for embedded systems are offered [28] that combine standard embedded processor architectures with GPUs. These are suitable as accelerators for specific software tasks. Therefore, performance models should be able to represent the timing-relevant behavior of the components of these complex, heterogeneous systems.

21.2.1.3 Development Cycles and Modeling Effort

Embedded system designers face tight deadlines. This means that the development process needs to be parallelized and has to support timing analysis and timing error detection as early as possible. While hardware software codesign approaches allow to start software development before the developed hardware is available, performance models allow to expose performance relevant errors very early in the development of an embedded system [44].

To reach this goal, the effort for model generation should be as low as possible, and the generated models should be available very early in the development process.

21.2.1.4 Speed of Performance Analysis

When using timing models to evaluate the performance of embedded systems, the speed of performance analysis is always an important optimization goal. The application of high-level performance simulation increases the development productivity and acceptance as simulation can easily be integrated into the development process. There exist also some applications of performance models like *Software-in-the-Loop* simulation [47] where the performance simulation needs to be faster than the execution of the software on the target hardware.

21.2.2 Static Software Timing Analysis

Static software execution time analysis mostly uses a combination of abstract interpretation [11] and Programming (ILP) [48] to determine an estimate of the execution time of an embedded software without actually executing the software.

In static analysis, the modeling of timing behavior is only part of the analysis, while a reconstruction of program structure [26, 46], program values, and loop bounds is also an important aspect of the analysis. The usage of caches in embedded architectures further complicates the static timing analysis of embedded systems.

Static analysis has the advantage that, using properly designed analysis, certain properties of the timing estimation can be guaranteed. Especially a formally safe Worst-Case Execution Time (WCET) estimate is currently not possible using the other analysis techniques. On the other hand, static analysis is problematic as some architectures such as many-core processors and GPUs are currently not analyzable using a static analysis tool.

21.2.3 Simulation-Based Software Timing Analysis

In the following sections, we give an overview of methodologies that provide some notion of time while simulating the execution of software on a processor. In particular, we exclude trace-driven simulation [20] where only a prerecorded trace of instructions is replayed.

21.2.3.1 RTL Simulation

In principle, a simulation can be generated from the Register-Transfer Level (RTL) description of a system. While obtaining Register Transfer Level (RTL) code for a full system is not practical for most application developers, such models are sometimes available commercially. For example, ARM recently acquired Carbon Design System and plans to market models compiled from the RTL descriptions as ARM Cycle Models. However, even when accelerated using specialized compilers such as Verilator [19], these simulations have a very low performance and therefore are not a good choice for application performance analysis, unless exact results are absolutely required.

21.2.3.2 Fixed Throughput Simulation

Many commercial simulators, such as Imperas OVP [18] and ARM FastModels [1], as well as mainline QEMU, provide a simple timing simulation that is based on an user-specified instruction throughput. The main use case of these simulations is functional analysis, where the simplified timing simulation only serves to ensure a linear progression of time. As the throughput is assumed to be constant during a simulation but can vary significantly for nontrivial applications on real processors, these models are not appropriate for software performance analysis.

21.2.3.3 Microarchitectural Simulation

Microarchitectural simulation focuses on the interaction between hardware components of the system microarchitecture such as individual processor pipeline stages, functional units, or caches. Individual component models abstract implementation details and only aim at approximating timing characteristics. The simulation is usually cycle driven. For classical Systems-on-Chips (SoCs), well-known examples of this class of simulators are Simplescalar [2] and Gem5 [4]. GPGPUsim [3] applies this approach to GPUs.

While the low-abstraction level of these simulations suggest a near-exact simulation, this is usually not achieved. Reported simulation errors when modeling real processors [5, 16, 35] are, at best, equal to other approximate approaches, while performance is significantly lower [2, 10, 35]. The main benefit of these simulations is the ease of modifying low-level details (e.g., branch predictor policies). Therefore this approach is mainly useful for computer architecture research but not a good fit for software performance analysis.

21.2.3.4 Analytical Performance Estimation

Analytical performance models consist of a profiling phase and an estimation phase. In the profiling phase, performance metrics such as instruction count, cache miss rates, or the number of mispredicted branches are extracted during the execution of a program. During the estimation phase, the performance models then integrate the performance metrics to estimate program execution times using an analytic formula. Analytical models for performance estimations of GPU cores were presented in [22] and [31].

21.2.3.5 Phase-Based Performance Estimation

Another analytical approach is trying to reduce the simulation to phases of a program representative for the behavior of the whole program. Sampling-based analytical simulation models use a cycle accurate simulation but try to restrict the simulation to parts of a program. Very simple applications of this principle are just simulating the first n instructions of a program run and using the number of instruction per cycle obtained from this simulation to interpolate the timing behavior of the remainder of the program. A slight improvement can be reached when the representative phase from the middle of the execution trace [49].

An application of this simulation technique is SimPoint [37, 38]; it divides the program execution in phases of 100 million instructions. Phases are characterized

by the basic block vectors capturing the number of executions of each basic block during a phase. These basic block vectors are used to identify similar phases using a clustering algorithm. The performance of the programs is estimated from the results of a cycle accurate simulation of each phase closest to a cluster center. Other approaches combine a sampling-based model with a higher-level analytical performance model [43].

21.2.3.6 Interval-Based Simulation

Interval-based analytical performance models combine the profiling phase and the estimation phase in one run.

As shown in Fig. 21.1, these performance models divide the execution of a program in parts with a constant instruction throughput divided by stall events like branch mispredictions or cache misses. These models therefore allow a more accurate simulation of the interleaving in the processor pipeline compared to analytical models considering the whole program execution.

In simple interval-based models [12], the duration of an interval is approximated by the number of instructions in the interval divided by the dispatch width of the simulated processor pipeline and a miss penalty of the miss event at the end of the pipeline stage. Interval-based simulation is extended by Sniper [6] to improve the execution time estimation of an interval by incorporating information on the number of available functional units and allowing out of order execution of data cache miss events. GPUMech [17] is an interval-based GPU performance model. It uses interval-based simulation claims to simulate the timing behavior of GPU-based systems. Their methods are comparable to the ones used in Sniper for CPUs. This model has been applied to complex processor pipelines in [21].

21.2.3.7 Control Flow-Driven Simulation

Control flow-driven timing simulation is performed based on a priori estimations of the execution time for small portions of the program code, as shown in Fig. 21.2.

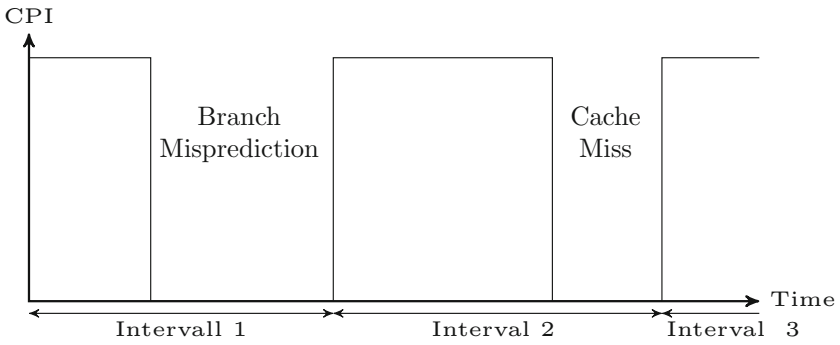


Fig. 21.1 Simulation by dividing time into intervals between miss events

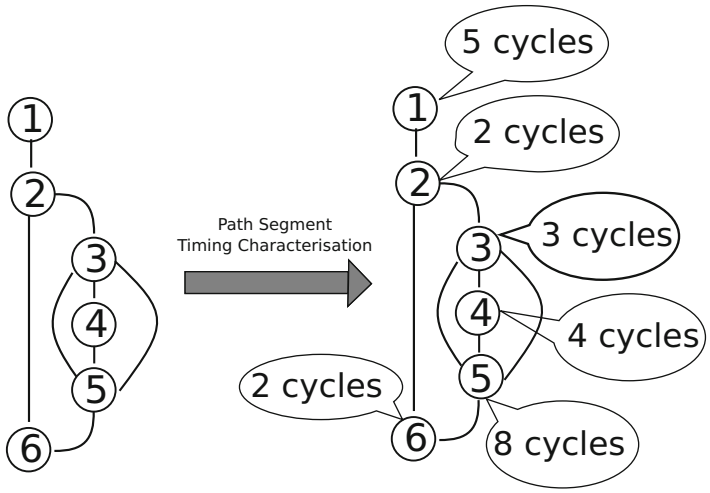


Fig. 21.2 Control flow-driven simulation uses static execution time estimates for small portions of programs

Most approaches use an a priori timing estimation to get the execution times for the basic block of the Control-Flow Graph (CFG).

For example, Tach, Tymiya, Kuwamura, and Ike [45] presented an approach where instruction block timings are first estimated assuming no cache misses or branch mispredictions. Timing is simulated by accumulating the block timings of all executed blocks and further penalties for cache misses and branch mispredictions.

Recent research [7, 13, 14, 29, 30, 32–34, 39, 41, 42] has demonstrated that simulation accuracy of this approach can be improved by differentiating different situations in which an instruction block is executed. In the most simple case [7, 13, 14], block timings are selected based on the preceding block. In contrast to a single block timing, this improves the consideration of instruction dependencies and instruction scheduling. Accuracy can be increased by considering more than one preceding block [33]. Besides an approach that implements this scheme, we also discuss a consideration of the preceding control flow using so-called Virtual Inlining and Virtual Unrolling (VIVU) context [29, 30, 42] in this chapter. These context enable a block timing granularity that allows an accurate simulation of complex applications on complex processor architectures without any online modeling of microarchitectural components, such as caches.

21.2.4 Summary

From the presented approaches especially control flow-driven simulation models combine many advantages. They clearly separate the functional simulation from the timing simulation. This allows a combination of these models with functional

simulation by source-level simulation (cf. ▸Chap. 17, "Parallel Simulation") or binary-level simulation (cf. ▸Chaps. 19, "Host-Compiled Simulation" and ▸20, "Precise Software Timing Simulation Considering Execution Contexts"). This separation also allows to choose different styles of timing models. In the following sections, we present three approaches to generate timing models for these kinds of systems at different abstraction levels.

21.3 Modeling Using Hardware-Independent Execution Cost Estimates

This approach to performance modeling of embedded systems tries to use the bare minimum of target specific information to produce a still meaningful result. While many approaches to generate performance models operate on the target-specific binary of a software, this approach aims to extract and quantify hardware-independent computational demand (HIC) from software source code and define a transition to gain hardware-specific execution costs (HSE). One main characteristic of our approach is that we extract computational demand for each software component from the source code. This has to be done only once. There is no need for target compilers or binary tools. We execute the application on the development platform to obtain data-dependent but hardware-independent execution characteristics of the application. If the developer changes components of the hardware platform or their configuration, only the transition to the HSE needs to be recalculated. The basic approach of the analysis is shown in Fig. 21.3.

The hardware independent computational demand is initially calculated on each basic block of the source-level control-flow graph of the original application. As

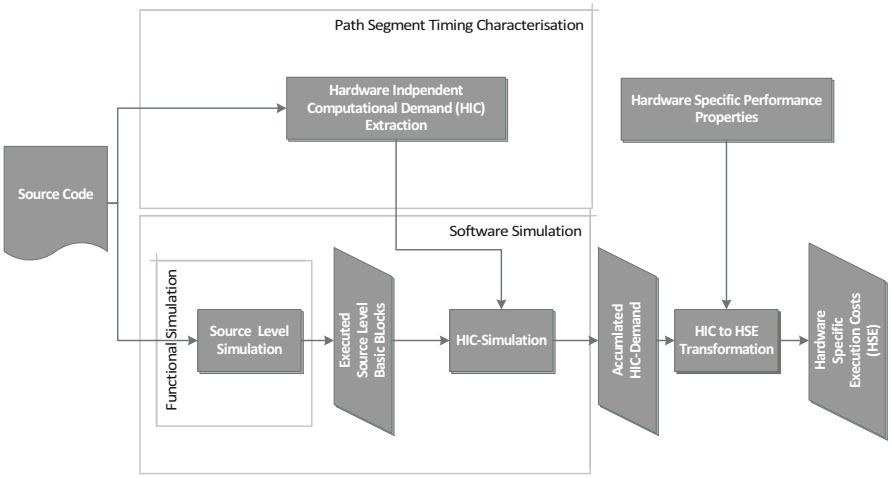
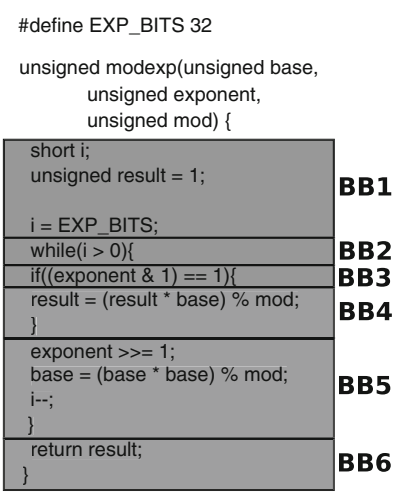


Fig. 21.3 Flow for the performance simulation with a hardware-independent computational demand simulation



computational demand		
BB1:		
= (int) 1	= (short) 1	
BB2:		
cond_branch 1	> (short) 1	
BB3:		
cond_branch 1	& (unsigned) 1	== (unsigned) 1
BB4:		
= (unsigned) 1	* (unsigned) 1	% (unsigned) 1
BB5:		
» 1	* (unsigned) 1	% (unsigned) 1
= (unsigned) 2	– (short) 1	
BB6:		
ind_branch 1		

Fig. 21.4 Calculating computational demand for a simple example

shown in Fig. 21.4, the approach extracts the computational demand by counting the operation type and the data types it operates on.

The HSE determination is based on abstract hardware specifications. The transition from HIC to HSE calculates the HSE for the considered pair of hardware and software. This estimation requires only an abstract hardware specification that can be extracted from the data sheets of the hardware platforms.

To calculate the hardware-specific execution costs, the target platform is specified with processor-specific and operation-specific attributes. The processor-specific attributes are:

1. The clock frequency at which the processor operates.
2. The branch predictor configuration, e.g., type and size of the target branch predictor
3. The used cache sizes and associativities and replacement strategies and cache miss penalties
4. A superscalar factor specifying the maximum number of instructions executed in parallel

In contrast to the general processor-specific attributes, the operation-specific attributes do not specify general behavior of the processor but give an execution cost attribute for the

1. The operation that is executed by this branch predictor.
2. The data type this operation operates on.
3. The number of cycles this operand needs for execution.
4. The number of parallel FUs that can execute this instruction type.

This means much less effort than implementing virtual models or using target compilers and binary tools for WCET analyzers. The transition only needs seconds which makes the approach much faster than the determination of HSE via virtual prototypes or ISS and can speed up the design process in complex software systems on heterogeneous hardware components before the initial mapping configuration is available. The HSE values can be used for an initial mapping determination approach. Experimental results show that the estimation is very fast and accurate enough to help designers making initial system configuration decisions.

21.4 Modeling Using Partial Architectural Knowledge

In this approach, we assume that the information available is comparable to the microarchitecture description given in architecture reference manuals of current microprocessors. This information is not sufficient to build a detailed cycle accurate simulator but can be used to derive analytic performance models of the microarchitecture. As generation of these kinds of models needs only partial knowledge of the modeled microarchitecture, we refer to them as partial microarchitectural models or microarchitecture-aware models.

The complete flow of the performance modeling and simulation using partial microarchitecture models is shown in Fig. 21.5. The method starts with the source code and the binary code of an application. We then extract the structure of the binary code CFG and do a structural matching of the source-level and binary-level control flow graphs and automatically annotated with function calls which reference the matched binary codes. Together with a generated path simulation code, this allows a simulation of binary-level paths through execution of the annotated source

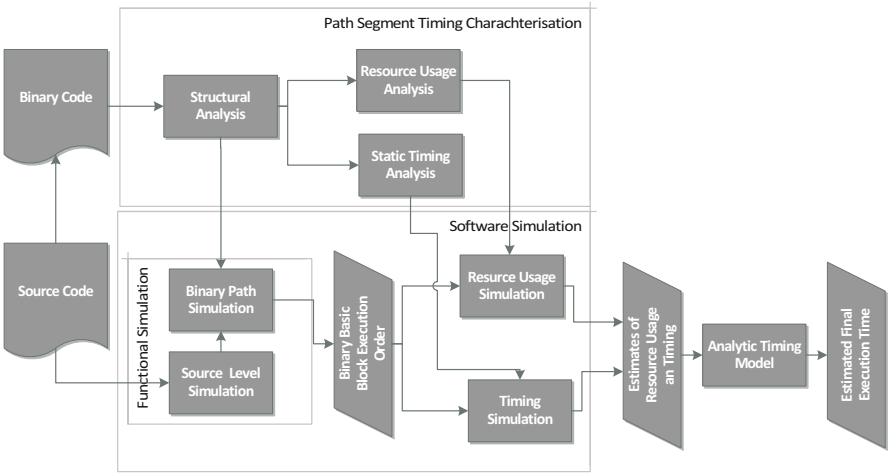


Fig. 21.5 Flow for the performance simulation with partial microarchitectural knowledge

code. The source to binary-level matching and path simulation code generation is not part of this chapter. Details of these steps are given in ▸ Chap. 17, "Parallel Simulation" and in [39–41].

The application-specific timing model is generated using an offline analysis of the binary-level control-flow graphs. It calculates for each basic block an estimated execution time depending on its predecessor. We additionally calculate a resource usage histogram for the instructions in each basic block. The results of the offline analysis are then added to the path simulation code and accumulate the per basic block matrix during execution of the instrumented source code. After the execution of the instrumented software, the accumulated performance metrics are used by a target-specific analytical performance model to produce a final estimate of the execution time.

Parts of the following subsections are based on our preceding publications [13, 14]. In these publications, we specifically focused on the application of this modeling style on GPUs. The following description is split in a part describing the modeling of GPU architectures (Sect. 21.6.1) and Sects. 21.4.1 and 21.4.2 describing the target agnostic parts of the model generation and simulation.

21.4.1 Static Timing Estimation Using Pipeline Execution Graphs

The static basic block timing analysis determines an optimistic execution time for each basic block in the binary-level control-flow graph. The effects of resource contention on the execution time can be incorporated by an analytical model after the timing simulation (Sect. 21.6.1). The analysis uses pipeline execution graphs [24] to model the timing behavior of each *instruction* on the pipeline. Our pipeline execution graph EG_B for a basic block is defined as

$$EG_B = (S_B, D_B, lat, use, res)$$

where the nodes in S_B represent each execution step for each instruction on the pipeline. $D_B \subset S_B \times S_B$ represents the dependence relation. It contains an edge for each dependence corresponding to the instructions in the basic block. In our model, an execution step might not directly correspond to a pipeline stage. The minimum latency between the start of an execution step and the start of a dependent execution step is given by the function $lat : D_B \rightarrow \mathbb{N}_0$. Latency is expressed in cycles.

To incorporate the dynamic resource usage into the timing model, we extend the pipeline execution graph model with an estimation of the resource usage. The resource usage function $use : V_B \rightarrow \mathbb{N}_0$ labels each step in the execution graph by the number of cycles the resources in this step are taken. The function $res : S_B \rightarrow \mathbb{N}_0$ maps each execution step to a unique identifier for the resource used in this step.

In Fig. 21.6, we show an example for a pipeline execution graph that is built for the analysis of an Bolero-3M-based microcontroller [36]. These microcontrollers are based on a PowerPC instruction-set architecture. Depending on the number of instructions in a PPC processor. The front end of the processor pipeline allows

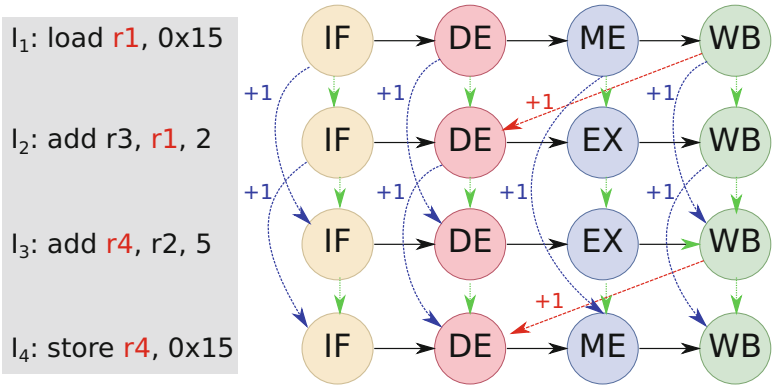


Fig. 21.6 Pipeline execution graph-based timing analysis

fetches of up to two 32-Bit instructions at once. This is indicated by the edges between the IF (instruction fetch) nodes of I_1 and I_3 and I_2 and I_4. So that every instruction is fetched one cycle after the instruction that is two positions ahead in the instruction stream. As the decode stage can handle two instructions as well, its representation is similar to the IF stage. In the third stage, the instructions are executed on multiple execution units. The instructions accessing memory are executed on a single memory unit (MEM), while there are two execution units for arithmetic and logic instructions available. This means there is a potential resource conflict between the MEM nodes of instruction I_1 and I_4. There is no potential resource conflict between I_1 and I_2 as the pipeline has two execution units for this instruction type available. The writeback to register file is handled for two instructions in parallel on this pipeline. The edges between the WB and DE nodes of the graph indicate a true data dependency. I1 loads data from memory that is used as an operand by the next instruction. In this case, the DE phase of instruction I_2 needs to wait for the cycle after the writeback of node instruction I_2. The same kind of dependency also exists between instruction I_3 and I_4.

Provided a pipeline execution graph, the timing analysis is done as a fixpoint iteration on the pipeline execution graph. The algorithm iterates over each state in the pipeline execution graph and calculates the earliest start times for each node by the maximum over the start times of the predecessors added with the corresponding latencies. The fixpoint iteration is finished when the start times for each state do not change anymore. To accelerate the convergence of the fixpoint iteration, we iterate over the states in topological sort order, but the result of the iteration is independent of the order in which the states are visited. The termination of the algorithm follows from the absence of cycles in the execution graph. For the static execution time analysis, we build the pipeline execution graph for the instructions of each basic block and calculate the fixpoint. It delivers a static timing analysis for the duration of the basic block by the maximum start time of each node in the execution graph. The analyzed execution time for a basic block v_i is called t_{v_i}. Building the execution

time of a kernel during simulation by summation over the basic block times t_i would overestimate the execution time of the kernel, because the execution of instructions from adjacent basic blocks can overlap. To incorporate this effect into the static timings, we also build pipeline execution graphs for the instructions from each pair of adjacent basic blocks. The analyzed execution time for each pair of basic blocks v_i, v_j is called $t_{(v_i,v_j)}$.

The results of this static analysis are then used for a simulative approximation of the timing behavior of a program.

21.4.2 Timing Annotation and Simulation

The timing information and resource information from static timing analysis is back annotated to the original source code, by inserting function calls to timing functions in the original source code and generating the corresponding timing function implementations. The algorithm for timing annotation is shown in Algorithm 1.

Algorithm 1 Algorithm for timing annotation

```
for v_S ∈ V_S do
  if ∃v_B ∈ V_B : map(v_B) = v_S then
    insert function call to timing simulation in v_S
    create function for timing simulation
    for paths p between a mapped block v'_B and v_B do
      Add code to function for:
      if last simulated block = v'_B then
        t_thread += Σ_{v_i ∈ p/v'_B} t_(v_{i-1},v_i) − t_{v_{i-1}}
        for resources r_i do
          u_thread(r_i) += Σ_{v_i ∈ p/v'_B} use(r_i, v_i)
        end for
      end if
    end for
  end if
end for
```

As the timing analysis has been done on the binary-level code and the simulation is based on annotations in the source code, the algorithm depends on a good mapping of binary basic blocks to source-level basic blocks. An example for a good method for mapping of source-level to binary-level control flow is given in ▸ Chap. 17, "Parallel Simulation". The algorithm first iterates over all mapped basic blocks in the source-level control-flow graph and inserts function calls that do the timing simulation according to the static analysis. The functions for timing simulation are generated in the inner loop of Algorithm 1. In this loop, the algorithm iterates over all binary-level paths between matched blocks and calculates the estimated execution time for this path by the sum over the pairwise execution times of the nodes in the path $t_{(v_{i-1},v_i)}$. As the summation would count the execution time for each start node twice, we subtract the execution time for each basic block $t_{v_{i-1}}$.

When doing simulations of GPU-based architectures, in addition to the execution times, we also annotate the threads resource usage for all resources in the pipeline. The results of an execution of the annotated source code on an OpenCL compatible device are an optimistic timing estimation of each thread and each accumulated resource usage of the thread for each resource in the pipeline. The final execution time of the kernel is estimated from these values using an analytical model. Details on an analytical model for GPUs are given in the following section.

21.5 Modeling Using Detailed Microarchitectural Knowledge

A complete, detailed description of the microarchitecture can be utilized to achieve an exact performance simulation. However, due to their extremely low performance, such simulations are, in practice, not a good choice for software performance analysis. Therefore, a simulation that achieves a high performance while maintaining an acceptable accuracy is preferable for this purpose, even if complete knowledge of the microarchitecture is available.

One concept for such a simulation is to shift most or all evaluations regarding the microarchitecture to an offline analysis that is executed before the simulation. The result of a single analysis can be reused by multiple simulations; the analysis cost is distributed among these simulations. This approach is therefore attractive for exploring software performance in different scenarios, such as over a wide range of input values. In this section, we give a brief overview of our simulation framework that realized this concept. More details can be found in ▸ Chap. 19, "Host-Compiled Simulation".

21.5.1 Framework Overview

An overview of our current framework is shown in Fig. 21.7. At first, the application binary code for the target processor is analyzed either statically or dynamically. In both cases, analysis results are stored in a software-specific timing model which is sometimes referred to as a timing database (TDB). A comparison of both analysis approaches is given in Sect. 21.5.2.

The timing model contains a description of the binary code control flow and timings for basic blocks of the program. Multiple timings are available for each block and differentiated by context. A context is an abstraction of the control flow leading to a block. The main advantages of using contexts is that the instructions that were executed before a block can be reflected in the context-dependent block timing. More details on this approach are given in Sect. 21.5.3.

A timing model can then be used in multiple simulations, where the timing simulation is driven by events indicating the execution of target basic blocks as defined by the control flow stored in the timing model. In a simulation of binary code execution (cf. ▸ Chap. 19, "Host-Compiled Simulation"), this can be achieved by instrumenting the first instruction of each basic block, whereas in a source-level

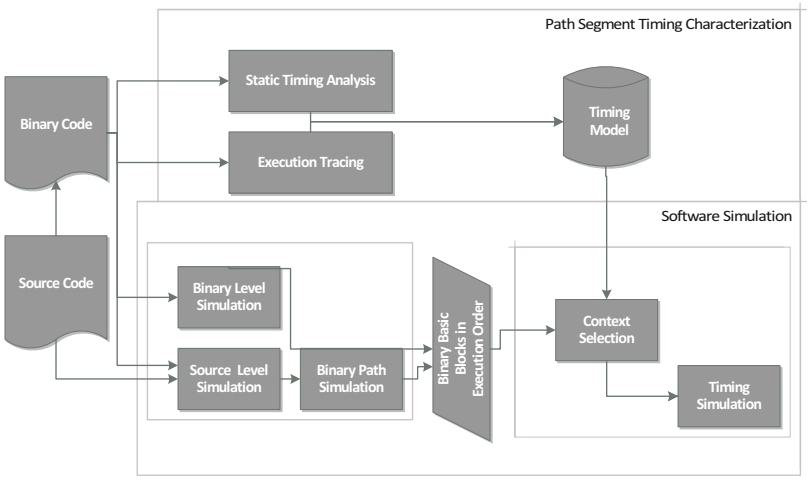


Fig. 21.7 Overview of the simulation framework

simulation (cf. ▸ Chap. 17, "Parallel Simulation", Section 21.4), a so-called path simulation is necessary, which simulates the target binary control flow based on the source-level control flow.

21.5.2 Static and Dynamic Analysis

The need for a precise microarchitectural model is the main drawback of detailed timing simulation. To limit the impact of this drawback, our framework is flexible in the kind of models that can be used by the offline analysis and supports both static and dynamic analyses. Thereby a model that was originally intended for a different purpose can be utilized by our simulation.

However, there are also various differences between static and dynamic analysis that lead to different advantages and disadvantages of one approach compared to the other. In principle, multiple timing models can be combined, which could also enable a hybrid analysis, but this topic is currently beyond the scope of our research.

In static analysis, the timing of the software is analyzed without executing the software. To avoid the halting problem, the actual states the software can get into have to be over-approximated. Firstly, this complicates the calculation of block timings. In practice, this currently restricts the application of static timing analysis to complex microarchitectures, in particular those including out-of-order execution. Secondly, it can lead to coverage deficits for programs containing asynchronous (e.g., interrupts) and indirect (e.g., function pointers) control flow changes. We developed a methodology to run multiple static analyses and combine their results to remove this issue.

In dynamic analysis, the timing of the software is analyzed by observing its execution. This avoids the issues of static analysis but makes it necessary to choose

program inputs that provide sufficient coverage. However, our experimental results demonstrate that selecting such inputs manually is feasible. Currently we observe software executions on target hardware, which removes the need for an additional model during the analysis.

21.5.3 Enhancing Accuracy by Considering Execution Contexts

On most modern embedded processors, the timing of an instruction sequence depends on the state of the microarchitecture before its execution, for example, due to instruction dependencies or cache contents. This state is heavily influenced by the already executed instructions. As aforementioned, this factor can be leveraged to improve simulation accuracy by obtaining multiple possible timings for each instruction block during the offline analysis and selecting an appropriate value for each execution of a block during the simulation.

More specifically, timings are differentiated by the preceding control flow. However, as the set of paths to block can in general be infinite and is likely excessively large for nontrivial programs, it is not possible to calculate a distinct timing for every path. Instead, the set of control flow paths is divided into a finite number of subsets, which are referred to as contexts, and block timings are differentiated by context.

In our framework, we apply the so-called VIVU contexts [26], which were originally developed for static analysis. Figure 21.8 shows a simple example for a VIVU contexts. This kind of context information captures the call stack and the loop iteration counts on the way from the start of the program to each basic block. The fact that this approach allows to distinguish the timing behavior of basic blocks depending on the call stack and the loop iteration count is reflected by its full name. Our experimental results demonstrate [30] that VIVU context coupled with a dynamic analysis by observing hardware executions enables a highly accurate timing simulation. This simulation is capable of simulating complex

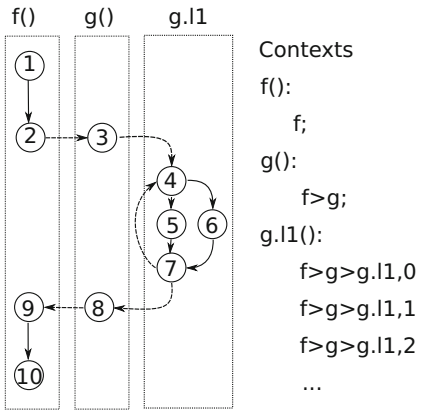


Fig. 21.8 An example for VIVU-context-based timing selection

software executing on complex processors at an error of typically less than 10% without any further modeling of microarchitectural elements including data caches.

21.6 Case Study: Modeling the Performance of a GPU-Based Microarchitecture

In this case study, we show the application of the timing model from Sect. 21.4 to a complex microarchitecture.

21.6.1 Applying the Simulation Approach to GPU Cores

To generate timing models for GPUs, the static analysis is run on the PTX assembly code and currently models the microarchitecture of a NVIDIA GTX 480 core. The analysis assumes that the pipeline is occupied by one warp exclusively.

As shown in Fig. 21.9, the static analysis models each instruction's execution on the given pipeline as a graph with five nodes. The first node (IF) corresponds to the front-end part of the pipeline up to the instruction buffer. The second part (IS) models the issue stage and the scoreboard. The latency of register accesses in the operand collector units is modeled by the node labeled OC. The fourth node (ALU) models the timing effects of the actual execution. This node is labeled according to the used execution unit (ALU, SFU, MEM). The last node models the writeback stage of the pipeline. This node is labeled WB.

Figure 21.10 shows an example of a pipeline execution graph for three add instructions on the pipeline of a GeForce GTX480 which is quite similar to the GPU core used in NVIDIAs embedded SoC Tegra K1 [25]. The latencies inherent to the

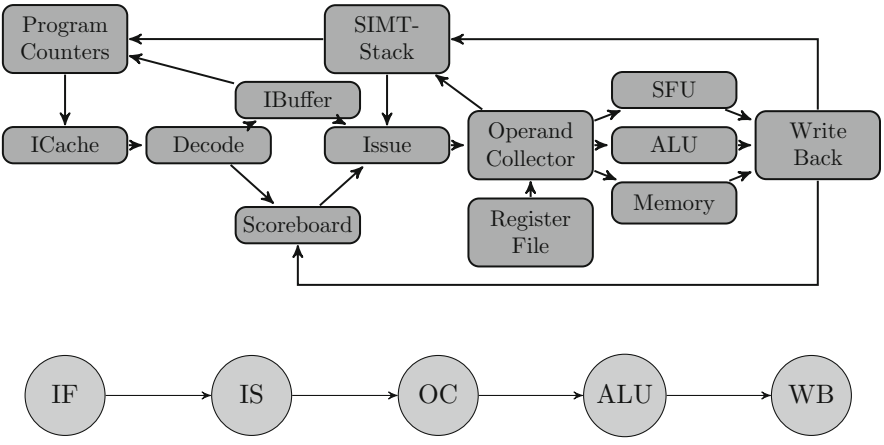


Fig. 21.9 GPU microarchitecture and the pipeline model

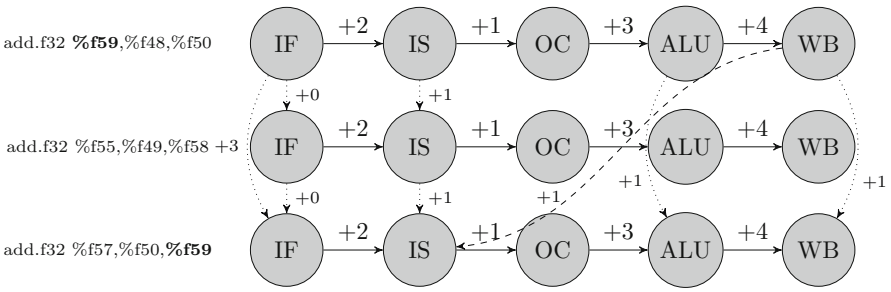


Fig. 21.10 Example of our pipeline analysis

pipelined execution of each instruction on the pipeline are shown by black edges. Instructions always take two cycles from instruction fetch to issue. The latency from issue to the operand collectors is one cycle for all instructions. The latency of the operand collectors depends on the number of registers read by the operation. The best case is always the number of registers read by the operation. The latency from execute to writeback can vary greatly depending on the instruction type. Load/store instructions show one cycle under the assumption of a cache hit and full coalescing of memory accesses, while double precision floating point division has a latency of 330 cycles. Resource dependencies between instructions are modeled by the blue edges in Fig. 21.10. As the front end fetches two adjoining instructions at a time in program order, each instruction is connected by an edge with latency zero in program order. To integrate the additional latency that every instruction is only fetched when the two entry instruction buffer is empty, we add additional edges between every second instruction. These edges have a latency of three cycles as this is the best-case instruction fetch latency. The issue stage issues one instruction a cycle in program order; this is modeled by an edge with latency one between each successive instruction. Resource dependencies in the execute stage are modeled in the same way. If there are multiple copies of a resource, the modeled pipeline has two ALUs, and the resource dependency edges skip instructions to account for the multiplicity of the resource. The same applies to the writeback step, as the pipeline can writeback two results at a time, there is only a dependency between every second node in WB. Data dependencies are always modeled by an edge with latency one from the writeback of the preceding instruction to the issue state of the depending instruction.

21.6.1.1 Analytical Timing Approximation for GPUs

Due to the inherent parallelism of GPU architectures, the cycle times and resource usages obtained from executing the annotated source code are specific to individual threads of the executed kernel, the levels of parallelism handled within the pipeline of a GPU core, simultaneous multi-threading, and warp-level parallelism.

Warps consist of the instructions of multiple threads from the same local work group. While the warp size may vary depending on the size of local work groups,

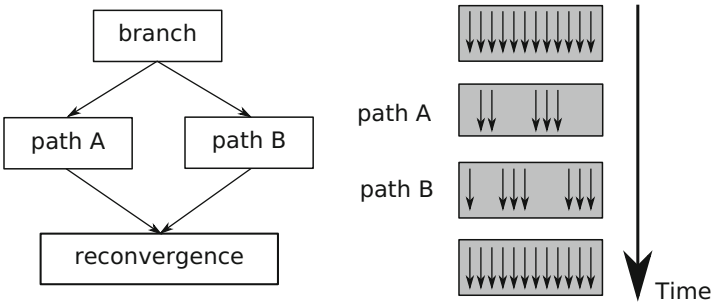


Fig. 21.11 An example of branch divergence

the preferred and maximum warp size on NVIDIA GPUs has been 32 threads for several generations. The threads of a warp always execute the same instruction in lockstep but are allowed to branch independently. As the instructions are allowed to branch independently, a so-called branch divergence can occur. Branch divergence is handled in hardware by executing each path sequentially and masking those operations which did not take the currently selected path. Figure 21.11 shows an example of branch divergence.

Apart from branch divergence, warps execute the instructions of several threads in lockstep. Each streaming multiprocessor handles multiple warps concurrently using fine-grained multi-threading.

The timing model uses the per thread execution times and resource usages as determined by the native execution of the annotated source code to calculate an estimate of the execution time of the whole kernel. The analytical model follows the hierarchy of parallelism of the multi-threaded execution. The algorithm starts with the execution time of the threads as determined by the source-level simulation. The execution time of a warp is then calculated from the execution time of the threads that form this warp. The execution times of all warps in a work group are combined to form the execution time of a work group. The execution time of the whole kernel is then estimated from the execution times of all work groups in the kernel. Timings up to this point do not consider the resource contention due to parallel execution of warps in the pipeline. This resource contention is taken into account by a final correction step.

The first step of the timing estimation calculates the execution time of each warp. This is done by first determining which threads form a common warp and then calculating the execution time of a warp t_{W_i} by taking the maximum execution time of all threads in the warp.

$$t_{W_i} = \max_{t_i \in W_i}(t_{t_i})$$

This calculation is motivated by the fact that all threads in a warp are executing the same instruction in lockstep. Due to branch divergence, it is still possible that the

simulated execution time of the threads in warp shows different execution times. The effects of branch divergence are approximated by taking the maximum execution time of all threads in the warp. This does not fully simulate all timing effects of branch divergence but accurately handles the most important case, of branch divergence at an *if* statement without an *else* or branch divergence at a loop exit condition. Real branch divergence in an if-else statement is not fully handled by our current model, but taking the maximum execution time still approximates the timing effects of this case. The execution times of a local work group WG_j is modeled by the end time of the last warp t_{last_j} in the work group. All warps in a local work group are started at the same time. So we can calculate the end time of the last warp by taking the maximum execution time of all warps in the local work group.

$$t_{last_j} = \max_{t_{w_i} \in WG_j} (t_{w_i})$$

Due to the limited bandwidth of the issue stage, the threads of a local work group, the i-th warp of a local work group issues at least $\lfloor i/2 \rfloor$ cycles after the first thread of the local work group. To incorporate this in our model, we add this to the finish time of the last warp. This leads us to the final equation for the execution time of the last warp

$$t_{last_j} = \max_{t_{w_i} \in WG_j} (t_{w_i} + \lfloor i/2 \rfloor)$$

The finish time of the last warp in each local work group is used to calculate the execution times of a kernel. The local work groups of a kernel can be executed in parallel, but due to constraints of a GPU's hardware, not all local work groups might be able to run in parallel. Given the number of parallel work groups n_{par}, we estimate the finish time of each work group. The first n_{par} work groups are started in parallel. The next work group is started when a work group finishes. This is expressed by the following equation:

$$t_{WG_j} = \begin{cases} t_{last_j} & : j < n_{par} \\ \min_{k \in \{j-n_{par},\ldots,j-1\}}(t_{WG_k}) + t_{last_j} & : j \geq n_{par} \end{cases}$$

The upper part of the equation calculates the finish times for the first n_{par} work groups, by using the finish time of the last warps. All further work groups are simulated by taking the minimum over the n_{par} predecessors and adding the finish time of the last warp in this work group. The optimistic execution time of the whole kernel is then the maximum finish time over all work groups:

$$t_{kernel\_opt} = \max_{WG_i}(t_{WG_i})$$

The kernel execution time so far does not consider any delays due to resource conflicts between multiple warps on the same pipeline. These resource conflicts

are modeled by the last step of our analytical model. The analytical model first calculates the resource usage $use_{W_j}(r_i)$ of a warp W_j as the maximum resource usage over all threads in the warp:

$$use_{W_j}(r_i) = \max_{t_k \in W_j} (use_{t_k}(r_i))$$

The resource usage is calculated for each resource r_i. We then calculate the resource usage of the whole kernel by summation over the resource usage of all warps in the kernel:

$$use_{kernel}(r_i) = \sum_{W_i \in Warps} use_{W_i}(r_i)$$

The optimistic execution time is then combined with the kernel's maximum resource usage to form the final execution time estimate. The most successful combination of resource usage and optimistic execution time we have found is the maximum of both values.

$$t_{kernel} = \max(t_{kernel\_opt}, \max_{r_i \in R} use_{kernel}(ri))$$

This model is surprising as it only takes resource contention into account when it is certain that there must be resource conflicts in the pipeline. But as GPUs are optimized for a high throughput, this model seems a reasonable choice. The accuracy of these models can be improved by doing a probabilistic approximation of the resource conflicts [14].

21.6.2 Results

We evaluated the performance model using several synthetic and real world benchmarks. All simulations were run on an Intel Core i7-4770 K CPU at 3.5 GHz with a NVIDIA GTX780 GPU with 12 streaming multiprocessors at 952 MHz. For the execution of the instrumented source code, we used either the CPU using Intel's implementation of OpenCL for CPUs or on the GPU using NVIDIA's implementation of OpenCL for GPUs. For comparison, we used the cycle accurate GPU simulator *gpgpu-sim*. Our configuration is based on the configuration for a NVIDIA GTX480 GPU as delivered by gpgpu-sim, but we reduced the model to more closely resemble an NVIDIA embedded gpu core. We also activated the so-called perfect memory mode of gpgpu-sim. This mode handles all memory accesses as cache hits. We choose to use a simulator for our evaluation as only limited information on the internal architecture of actual GPUs is available, and we needed to verify the results of the pipeline model ignoring influences from the memory subsystem.

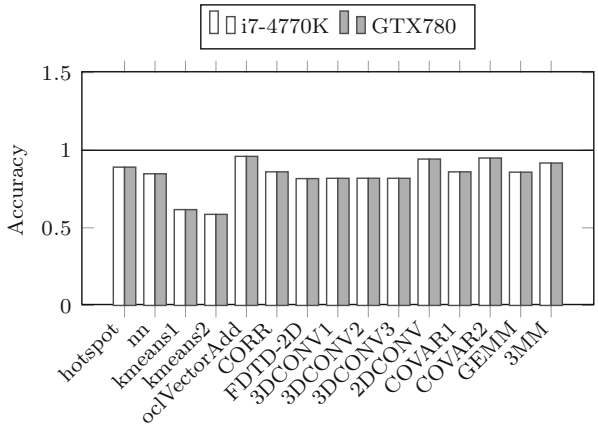


Fig. 21.12 Accuracy of the GPU performance simulation

The proposed simulation framework has been implemented as a shared library implementing the run-time API for translation and running of OpenCL kernels. For performance simulations, the library can be preloaded using the standard Unix `LD_PRELOAD` mechanism, so no changes to the host binaries used for simulation are needed. The kernels were translated to PTX code using *clang* as compiler [23]. All benchmarks were run with most compiler optimizations enabled (`-O3`), and we used the compiler switch to enable debug information in the compiler-generated assembly files (`-g`). We evaluated the performance model with benchmarks from the Rodina [9] and polybench-gpu [15] Benchmarks.

Figure 21.12 shows the execution times as estimated by our method divided by the execution times provided by `gpggpu-sim`. All our execution times underestimate the execution time as is expected by the best-case assumptions made by the performance modeling. For all but two kernels, the proposed simulation technique provides an accuracy of 80% or higher.

The accuracy results do not change between execution of the instrumented source code on a CPU or GPU as the performance simulation can be run on any OpenCL compatible device. The speedups in terms of the simulation time are shown in Fig. 21.13. The simulation time of our tool includes the time for data transfers from and to the OpenCL device, the execution of the kernel on the device, and the execution time of the analytical resource conflict model. As our goal is to support the simulation of long running application scenarios, the speedups do not include the time used for the static analysis of GPU kernels and binary to source matching. When instrumented source code is run on a CPU, speedups range between 140 for *oclVectorAdd* and 24061 for *kmeans2*. If the instrumented source code is run on a GPU, the speedups range between 477 for *oclVectorAdd* and 67750 for *COVAR*. The variation of execution speeds is partly explained by different basic block sizes in the applications. The other important factor considering speedups is the proportion of execution time of a thread to the number of threads. In our

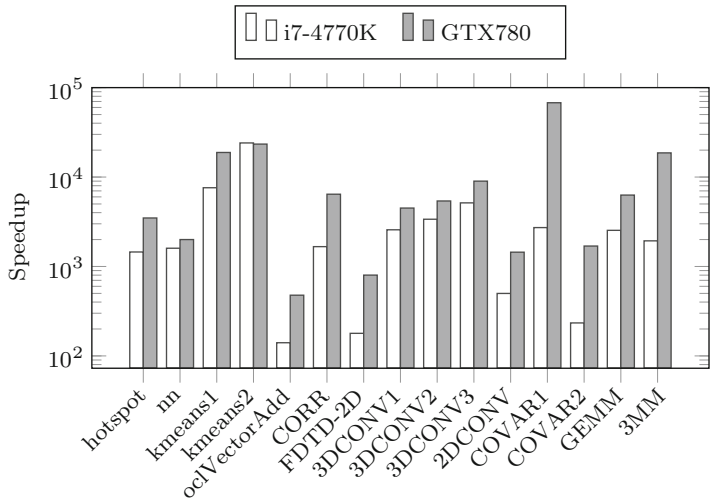


Fig. 21.13 Speedup of the simulation compared to the ISS *gpgpusim*

model, threads are simulated with the parallelism of the OpenCL device used for the simulations but our analytical performance model is run on the host without the use of parallelism. All benchmarks except *kmeans1* show an improvement of the simulation speed when the instrumented source code is executed on the GPU. The simulation performance of *kmeans1* degrades slightly as this benchmark does not fully utilize the available parallelism on the GPU.

21.7 Approaches to Include a Cache and Memory Simulation

The models so far assume that the latency of instructions can be statically determined. There are three basic approaches to integrate the simulation.

The simplest approach is an offline cache simulation. Prerequisite for an offline cache simulation is an approximation prior to the performance analysis. In an offline cache simulation, the cache miss rates of the application can be estimated using an external cache simulator. Either by executing the target binary code of the application on an instruction-set simulator or by executing the application on a host-compiled cache simulator like *cachegrind* [27]. Cache miss rates might even be provided as estimates by the developer using the performance model. The effect of the memory performance can then be incorporated into the model generation by increasing the latencies for each memory accessing instruction using an adoption of the standard formula to approximate the average memory access time:

$$\text{Average Memory Access Time} = \text{Cache Hit Time} + \text{Cache Miss Rate} \cdot \text{Miss Penalty}$$

This is the approach that is most compatible with the hardware-independent computational demand simulation, as the average memory access time can be used to dynamically determine the hardware-independent computational demand.

In contrast to an offline analysis that is run ahead of the simulation time, an online analysis that is run concurrently to the timing simulation might be carried out.

In the case of context-sensitive models based on hardware tracing, the memory subsystem often does not need to be modeled, as the timing impact of the memory subsystem is already part of the timed basic block traces, and a context-sensitive performance model reflects the timing impact well enough for many applications.

When higher accuracies or safe bounds are required. The timing relevant behavior of the memory subsystem can be analyzed by using a fully static analysis. In this case, the cache analysis is run in before the pipeline analysis and the results of the cache analysis are integrated with the pipeline analysis by changing the latencies of each individual instruction. If context-sensitive analyses are used, the pipeline analyses use individual latencies for each pair of instruction and context.

21.8 Discussion

The modeling techniques presented in this chapter represent models at different levels of abstraction and different trade-offs considering the challenges from Sect. 21.2. In this section, we briefly compare different modeling techniques addressing these challenges.

21.8.1 Comparison of Modeling Techniques

All of the considered modeling techniques have some specific drawbacks. For early platform component selection and task mapping onto different cores, it is probably the best solution to use hardware-independent execution cost estimates taken from Sect. 21.3 as these models require the least development effort and allow a good performance approximation for a wide variety of hardware components. However, the main drawback of these models are their quite low simulation accuracy.

The highest accuracy can be achieved by using context-sensitive performance models with detailed knowledge about the underlying microarchitecture (Sect. 21.5). These models often come up with an average simulation error below 1% and a maximum simulation error below 10%. The main drawback of this technique is the modeling effort and the need of a detailed microarchitectural model in terms of a cycle accurate simulator, a static WCET analysis tool, or a hardware implementation using a complex tracing unit. At least one of these approaches is generally available for standard embedded CPUs, but these are not generally available for application-specific processors or GPUs. If none of these approaches are available, the generation of a performance model would become necessary which requires an effort of several person months.

The technique using partial knowledge about the microarchitecture (Sect. 21.4 tries to balance the modeling effort and the model accuracy. Timing models can be developed in less than a month, and these models can represent most of the microarchitectural timing behavior of modern pipelined architectures. So these models are available very early on in the design process. This modeling style is also the only one that has been successfully applied to GPU-based microarchitectures, making it currently the correct choice for modeling of application-specific processors especially GPUs and GPU-based architectures.

21.9 Conclusions

In this chapter, we have discussed three different approaches to provide models for software performance analysis and compared their drawbacks. None of these techniques are able to fulfill all characteristics desired from a software performance model. Future research will need to allow an easier integration and seamless switching between different modeling techniques. Furthermore, an easy optimization of a performance model toward a timing characteristics of a desired target application has to be considered.

References

1. Arm fast models. http://www.arm.com/products/tools/models/fast-models/
2. Austin T, Larson E, Ernst D (2002) Simplescalar: an infrastructure for computer system modeling. Computer 35(2):59–67
3. Bakhoda A, Yuan GL, Fung WWL, Wong H, Aamodt TM (2009) Analyzing CUDA workloads using a detailed GPU simulator. IEEE, pp 163–174. doi:10.1109/ISPASS.2009.4919648
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. SIGARCH Comput Archit News 39(2):1–7. doi:10.1145/2024716.2024718
5. Butko A, Garibotti R, Ost L, Sassatelli G (2012) Accuracy evaluation of Gem5 simulator system. In: 2012 7th international workshop on reconfigurable communication-centric systems-on-Chip (ReCoSoC), pp 1–7. doi:10.1109/ReCoSoC.2012.6322869
6. Carlson TE, Heirman W, Eyerman S, Hur I, Eeckhout L (2014) An evaluation of high-level mechanistic core models. ACM Trans Archit Code Optim (TACO). doi:10.1145/2629677
7. Chakravarty S, Zhao Z, Gerstlauer A (2013) Automated, retargetable back-annotation for host compiled performance and power modeling. In: Proceedings of the ninth IEEE/ACM/IFIP international conference on hardware/software codesign and system synthesis (CODES+ISSS), Newport Beach
8. Charette RN (2009) This car runs on code. IEEE Spectr 46(3):3
9. Che S, Boyer M, Meng J, Tarjan D, Sheaffer JW, Lee SH, Skadron K (2009) Rodinia: a benchmark suite for heterogeneous computing. In: 2009 IEEE international symposium on workload characterization (IISWC), vol 2009. IEEE, pp 44–54. doi:10.1109/IISWC.2009.5306797
10. 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
11. 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 the 4th ACM SIGACT-SIGPLAN symposium principles of programming languages, New York

12. Eyerman S, Eeckhout L, Karkhanis T, Smith JE (2009) A mechanistic performance model for superscalar out-of-order processors. ACM Trans Comput Syst 27(2):3:1–3:37. doi:10.1145/1534909.1534910
13. Gerum C, Bringmann O, Rosenstiel W (2015) Source level performance simulation of GPU cores. In: Design automation and test Europe, Grenoble
14. Gerum C, Rosenstiel W, Bringmann O (2015) Improving accuracy of source level timing simulation for GPUs using a probabilistic resource model. In: International conference on embedded computer systems: architectures modeling and simulation (SAMOS), Samos
15. Grauer-Gray S, Xu L, Searles R, Ayalasomayajula S, Cavazos J (2012) Auto-tuning a high-level language targeted to GPU codes. In: 2012 innovative parallel computing (InPar). IEEE, pp 1–10. doi:10.1109/InPar.2012.6339595
16. Gutierrez A, Pusdesris J, Dreslinski RG, Mudge T, Sudanthi C, Emmons CD, Hayenga M, Paver N (2014) Sources of error in full-system simulation. In: 2014 IEEE international symposium on performance analysis of systems and software (ISPASS). IEEE, Piscataway, pp 13–22
17. Huang JC, Lee JH, Kim H, Lee HHS (2014) GPUMech: GPU performance modeling technique based on interval analysis. In: 47th annual IEEE/ACM international symposium on microarchitecture, pp 268–279. doi:10.1109/MICRO.2014.59
18. Imperas open virtual platforms. http://www.ovpworld.org/
19. Introduction to verilator. http://www.veripool.org/wiki/verilator
20. Isshiki T, Li D, Kunieda H, Isomura T, Satou K (2009) Trace-driven workload simulation method for multiprocessor system-on-chips. In: Proceedings of the 46th annual design automation conference, San Francisco
21. Karkhanis TS, Smith JE (2004) A first-order superscalar processor model. SIGARCH Comput Archit News 32(2):338–349. doi:10.1145/1028176.1006729
22. Lai J, Seznec A (2012) Break down GPU execution time with an analytical method. ACM Press, New York, pp 33–39. doi:10.1145/2162131.2162136
23. Lattner C, Adve V (2004) LLVM: a compilation framework for lifelong program analysis & transformation. In: Proceedings of the 2004 international symposium on code generation and optimization (CGO'04), Palo Alto
24. Li X, Roychoudhury A, Mitra T (2006) Modeling out-of-order processors for WCET analysis. Real-Time Syst 34(3):195–227. doi:10.1007/s11241-006-9205-5
25. Li A, Serban R, Negrut D (2014) An overview of NVIDIA Tegra K1 architecture. http://sbel.wisc.edu/documents/TR-2014-17.pdf
26. 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
27. Nethercote N, Seward J (2007) Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Proceedings of ACM SIGPLAN conference on programming language design and implementation (PLDI), Seattle
28. Nvidia: NVIDIA Tegra K1 A New Era in Mobile Computing, pp 1–26
29. 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)
30. 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), Hong Kong
31. Parakh AK, Balakrishnan M, Paul K (2012) Performance estimation of GPUs with cache. IEEE, pp 2384–2393. doi:10.1109/IPDPSW.2012.328
32. 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

33. 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), Hong Kong
34. 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), Tampere
35. 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), Abu Dhabi
36. Semiconductor F, Microelectronics S (2012) Bolero_3m microcontroller reference manual
37. Sherwood T, Perelman E, Calder B (2001) Basic block distribution analysis to find periodic behavior and simulation points in applications. In: Proceedings of the 2001 international conference on parallel architectures and compilation techniques. IEEE, Washington
38. Sherwood T, Perelman E, Hamerly G, Calder B (2002) Automatically characterizing large scale program behavior. ACM SIGARCH Comput Archit News 30(5):45–57, ACM
39. Stattelmann S (2013) Source-level performance estimation of compiler-optimized embedded software considering complex program transformations. Verlag Dr. Hut
40. Stattelmann S, Bringmann O, Rosenstiel W (2011) Dominator homomorphism based code matching for source-level simulation of embedded software. In: Proceedings of the seventh IEEE/ACM/IFIP international conference on hardware/software codesign and system synthesis, CODES+ISSS'11. ACM, New York, pp 305–314. doi:10.1145/2039370.2039417
41. Stattelmann S, Bringmann O, Rosenstiel W (2011) Fast and accurate source-level simulation of software timing considering complex code optimizations. In: Proceedings of the 48th design automation conference (DAC), San Diego
42. 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), Karlsruhe, pp 295–298
43. Van den Steen S, De Pestel S, Mechri M, Eyerman S, Carlson T, Black-Schaffer D, Hagersten E, Eeckhout L (2015) Micro-architecture independent analytical processor performance and power modeling. In: 2015 IEEE international symposium on performance analysis of systems and software (ISPASS), pp 32–41. doi:10.1109/ISPASS.2015.7095782
44. Teich J (2012) Hardware/software codesign: the past, the present, and predicting the future. Proc IEEE 100(Special Centennial Issue):1411–1430
45. 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), Dresden
46. Theiling H (2002) Control flow graphs for real-time systems analysis. Dissertation, Universität des Saarlandes
47. Werner S, Masing L, Lesniak F, Becker J (2015) Software-in-the-loop simulation of embedded control applications based on virtual platforms. In: 2015 25th international conference on field programmable logic and applications (FPL). IEEE, Piscataway, pp 1–8
48. Wilhelm R (2004) Why AI + ILP is good for WCET, but MC is not, nor ILP alone. In: Proceedings of the 5th international conference on verification, model checking, and abstract interpretation, VMCAI 2004, Venice, pp 309–322. doi:10.1007/978-3-540-24622-0_25
49. Yi JJ, Kodakara SV, Sendag R, Lilja DJ, Hawkins DM (2005) Characterizing and comparing prevailing simulation techniques. In: 11th international symposium on high-performance computer architecture (HPCA-11), San Francisco