Host-Compiled Simulation

19

Daniel Mueller-Gritschneder and Andreas Gerstlauer

Abstract

Virtual Prototypes (VPs), also known as virtual platforms, have been now widely adopted by industry as platforms for early software development, HW/SW coverification, performance analysis, and architecture exploration. Yet, rising design complexity, the need to test an increasing amount of software functionality as well as the verification of timing properties pose a growing challenge in the application of VPs. New approaches overcome the accuracy-speed bottleneck of today's virtual prototyping methods. These next-generation VPs are centered around ultra-fast host-compiled software models. Accuracy is obtained by advanced methods, which reconstruct the execution times of the software and model the timing behavior of the operating system, target processor, and memory system. It is shown that simulation speed can further be increased by abstract TLM-based communication models. This support of ultra-fast and accurate HW/SW cosimulation will be a key enabler for successfully developing tomorrows Multi-Processor System-on-Chip (MPSoC) platforms.

Acronyms

API	Application Programming Interface
CFG	Control-Flow Graph
HAL	Hardware Abstraction Layer
HW	Hardware
IPC	Inter-Process Communication

D. Mueller-Gritschneder (✉)
Department of Electrical and Computer Engineering, Technical University of Munich, Munich, Germany
e-mail: daniel.mueller@tum.de

A. Gerstlauer
Department of Electrical and Computer Engineering, The University of Texas at Austin, Austin, TX, USA
e-mail: gerstl@ece.utexas.edu

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_18

IR	Intermediate Representation
ISA	Instruction-Set Architecture
ISS	Instruction-Set Simulator
MPSoC	Multi-Processor System-on-Chip
OS	Operating System
SLDL	System-Level Description Language
TD	Temporal Decoupling
TLM	Transaction-Level Model
VP	Virtual Prototype
WCET	Worst-Case Execution Time

Contents

19.1 Introduction . . . 594
19.1.1 Traditional Virtual Prototype Simulation . . . 595
19.1.2 Next-Generation Virtual Prototypes . . . 596
19.1.3 Temporal Decoupling . . . 597
19.2 Source-Level Software Simulation . . . 598
19.2.1 Binary to Source Mapping . . . 600
19.2.2 Memory Trace Reconstruction . . . 602
19.2.3 Block-Level Timing Characterization . . . 603
19.2.4 Back-Annotation . . . 604
19.3 Host-Compiled OS and Processor Modeling . . . 605
19.3.1 OS Modeling . . . 606
19.3.2 Processor Modeling . . . 608
19.3.3 Cache Modeling . . . 609
19.4 TLM Communication for Host-Compiled Simulation . . . 612
19.4.1 TD with No Conflict Handling . . . 612
19.4.2 TD with Conflict Handling at Transaction Boundaries . . . 613
19.4.3 TD with Conflict Handling at Quantum Boundaries . . . 614
19.4.4 Abstract TLM+ with Conflict Handling at SW Boundaries . . . 616
19.5 Summary and Conclusions . . . 617
References . . . 617

19.1 Introduction

Due to increased complexity of modern embedded and integrated systems, more and more design companies are adopting virtual prototyping methods. A Virtual Prototype (VP), also known as a virtual platform, is a computer model of a HW/SW system. In such a HW/SW system, *tasks* of an application are executed on one or more *target processors*, e.g., ARM cores. Tasks usually run on top of an Operating System (OS). The tasks can communicate and access memory and peripherals via communication fabrics, e.g., on-chip busses. Next to obtaining correct hardware with less iterations, VPs support early software development, performance analysis, HW/SW coverification, and architecture exploration.

Modern Multi-Processor System-on-Chip (MPSoC) platforms feature multiple hardware and software processors, where processors can each have multiple cores,

all communicating over an interconnection network, such as a hierarchy of busses. The large amount of functionality and timing properties that need to be validated for complex MPSoCs brings traditional VP approaches to their limits. New VPs are required, which raise the abstraction to significantly increase simulation speed. This is a challenging task, because high abstraction leads to a loss of timing information, penalizing simulation accuracy.

This gives rise to next-generation VPs, which are centered around host-compiled software models. Abstraction is applied at all layers of the system stack starting from the software level, including operating system and processor models, down to abstract communication models. Intelligent methods are applied to preserve simulation accuracy at ulta-high simulation speeds. These methods are independent of the used System-Level Description Language (SLDL). Yet, SystemC [1] has nowadays emerged as a quasi-standard for system-level modeling. Therefore, SystemC is used as the main SLDL to illustrate the modeling concepts throughout this chapter.

Next to ultra-high simulation speed, VPs based on host-compiled simulation also provide improved debug abilities. Both software and hardware events can be traced jointly and transparently as the simulation model describes both in one executable.

19.1.1 Traditional Virtual Prototype Simulation

The VP is simulated via an SLDL simulation kernel. The PC, which runs the simulation, is referred to as the *simulation host*. Naturally, it can have a different Instruction-Set Architecture (ISA) from the target processors. In discrete-event (compound adjective) simulation, the simulation kernel on the host advances the logical *simulation time*. To simulate concurrent behavior, simulation processes are sequentially executed based on *scheduling events* and the simulation time. Scheduling events suspend or resume simulation thread processes (*threads*) or activate method processes. Suspending and resuming the thread processes requires context switches, which can produce significant simulation overhead. This overhead reduces simulation speed, which measures how fast the simulation is performed in terms of the *physical time*.

Communication is usually modeled using abstract Transaction-Level Models (TLMs). TLMs center around memory-mapped communication but omit the detailed simulation of the bus protocol. In TLM, the bus interface is modeled by a TLM socket. A transaction is invoked by an *initiator* (master) module when calling a predefined transport function on its socket. The function is implemented at the *target* (slave) module. During simulation the initiator socket is bound to the target socket, and the respective transport function is called.

While TLMs have been successfully applied to model communication aspects, today's VPs usually model the computational part by emulating the software on Instruction-Set Simulators (ISSs), which are either inaccurate or slow. As such, the amount of functionality and timing properties that can be checked by traditional VPs remains limited by their simulation speed. The low speed results from the

high number of scheduling events created by the traditional computation and communication model.

19.1.2 Next-Generation Virtual Prototypes

Simulation speed can be increased by reducing the number of scheduling events. One possibility is to raise the level of abstraction and thus lower the level of detail in the simulation. Abstractions can increase simulation speed significantly but may decrease accuracy due to loss of timing information or missing synchronization events as discussed in Sect. 19.1.3.

Next-generation VPs are aimed at overcoming these challenges by using intelligent modeling approaches to preserve simulation accuracy. The abstraction is raised by source-level simulation of software as an advanced method for software performance analysis. Instead of emulating the software program with an ISS of the target processor at the binary level, the source code of the software is directly annotated with timing or other information. The annotated source code can be directly compiled and executed on the simulation host machine, which leads to a huge gain in simulation speed compared to ISS simulation. However, this requires access to the source code by the user, which might not be available especially for library functions. This is a limitation of such approaches, which can be partly overcome, e.g., by profiling library functions beforehand. Additionally, sophisticated methods are required to annotate potentially expensive and slow cosimulation models for any dynamic low-level target behavior, such as stack or cache behavior, that cannot be easily or accurately estimated through static analysis

Pure source-level simulation approaches focus on emulating stand-alone application behavior only. However, interferences among multiple tasks running on a processor as well as hardware/software interactions through interrupt handling chains and memory and cache hierarchies can have a large influence on overall software behavior. As such, OS and processor-level effects can contribute significantly to overall model accuracy, while also carrying a large simulation overhead in traditional solutions. So-called host-compiled simulation approaches therefore extend pure source-level models to encapsulate back-annotated application code with abstract, high-level, and lightweight models of OSs and processors. This is aimed at providing a complete, fast, and accurate simulation of source-level software running in its emulated execution environment.

Additionally, embedded and integrated systems are composed out of many communicating components as we move toward embedded multi-core processors. The simulation of communication events can quickly become the bottleneck in system simulation. Next-generation VPs tackle this challenge by providing abstract communication models. Special care must be taken in such abstract models to capture the effect of conflicts due to concurrent accesses on shared resources. Different methods are addressed, which can model the effect of arbitration, e.g., by retroactive correction of the timing behavior. This correction is usually performed by a central timing manager.

19.1.3 Temporal Decoupling

Overall, any discrete-event simulation of asynchronous interactions among concurrent system components will always come with a fundamental speed and accuracy trade-off. Concurrency is simulated by switching between execution of the simulation processes at scheduled simulation events. With increasing amount of such scheduling events, simulation speed drops due to the overhead caused by the involved context switches.

Naturally, simulation speed can be increased by reducing the number of scheduling events. This can be achieved by raising the level of abstraction in the simulation model. A less detailed simulation usually leads to fewer scheduling events. However, a simulation at coarser granularity also leads to timing information being potentially lost. Maintaining a coarser timing granularity results in a Temporal Decoupling (TD) of simulation processes. TD lets simulation processes run ahead of the logical simulation time up to a given *time quantum*, which increases the simulated timing granularity and decreases the number of scheduling events. The logical simulation time of the kernel is referred to as *global simulation time*. By contrast, components keep track of their time using a *local simulation time*, which is usually defined as an offset to the global simulation time.

For HW/SW systems, TD increases simulation speed but may decrease accuracy due to out-of-order accesses to or wrong prediction of conflicts on shared resources. Specifically, TD may decrease accuracy due to incoming events being captured too late, and also in terms of outgoing events being produced too early. This is illustrated for a scenario with two concurrent active threads in Fig. 19.1. Without TD, as shown in the upper half of the figure, Thread 2 writes the value of b to the shared target before it is read by Thread 1. Additionally, the shared target can arbitrate accesses. The writing process of b is ongoing when the read access is performed, which adds additional delay on the read access of Thread 1. In contrast, temporal decoupling leads to out-of-order accesses to b as shown in the lower half of the figure. Additionally, the conflicting access by Thread 2 cannot be predicted by the shared resource, and Thread 1 sees the write to b by Thread 2 only at a much later (local) time. Similar problems of events being recognized too late arise when modeling active threads that can be interrupted or preempted by external sources.

TD methods can be classified as optimistic or conservative. Optimistic approaches aggressively execute a model under temporal decoupling. Inaccuracies due to out-of-order execution are either tolerated or corrected for at a later point in simulation. Note that optimistic approaches do not guarantee an accurate order of events and interactions unless correction using a full rollback is possible in the simulator. As the name suggests, conservative approaches, by contrast, always maintain the correct order of events and interactions. They only apply temporal decoupling as long as it can be guaranteed that results do not depend on the order of the respective events. Both methods can benefit from intelligent compile-time or run-time usage of system knowledge. Accuracy of optimistic approaches can be improved by using system knowledge to perform retroactive timing corrections. Conservative approaches can increase their simulation speed by dynamically

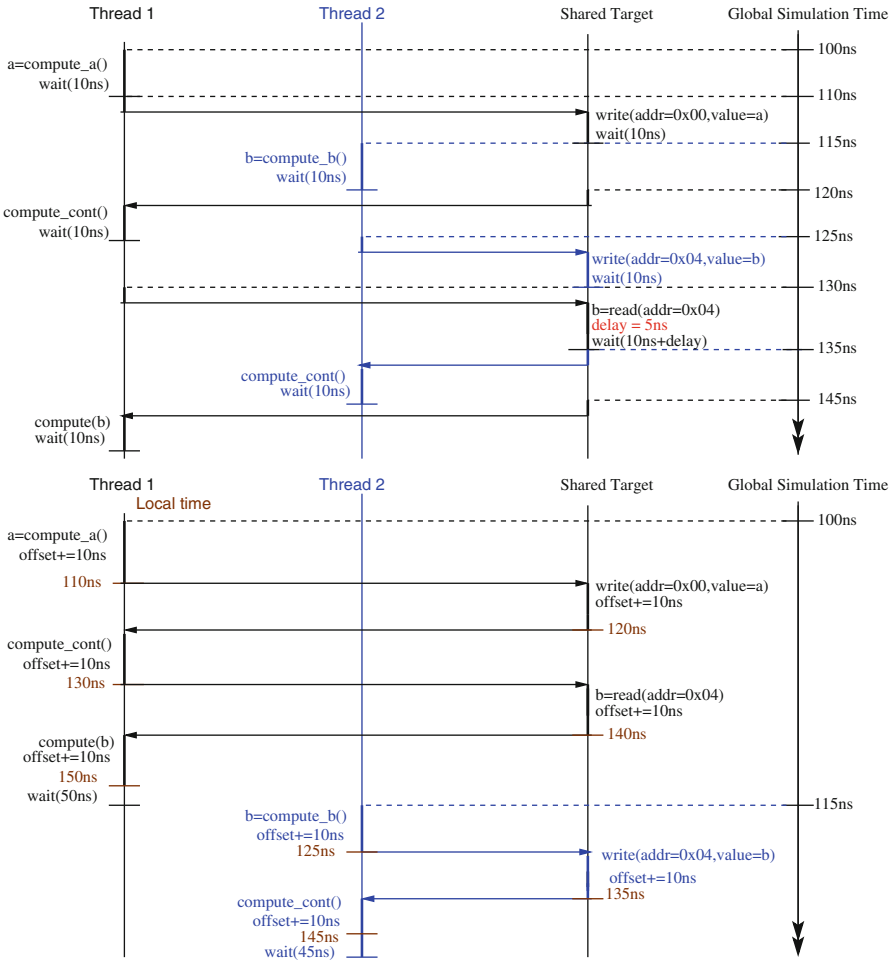

Fig. 19.1 Simulation without/with Temporal Decoupling (TD)

adjusting their local time quantum using system knowledge to perform prediction of possible future event interactions. In Sects. 19.2 and 19.4, we will show in detail how accuracy and speed can be improved by such intelligent TD modeling approaches.

19.2 Source-Level Software Simulation

Traditional virtual platforms simulate software execution at a detailed instruction level. This includes both a functional as well as, optionally, a timing model. Such low-level ISSs can be very accurate, especially when combined with a cycle-level microarchitecture model, but they also tend to be very slow, especially when cosimulating multiple processor or cores in a full-system context. Functional simulation

speed can be significantly improved by statically translating instructions (and caching translated results, see the ▶ Chap. 18, "Multiprocessor System-on-Chip Prototyping Using Dynamic Binary Translation") instead of dynamically interpreting them. Timing models can be accelerated by executing them in FPGAs or other dedicated hardware platforms [6]. Nevertheless, simulation speed, particularly when requiring accurate timing, remains a major concern.

Source-level approaches are aimed at improving the speed of both functional and timing simulations. Computation is modeled at the source or Intermediate Representation (IR) level, which allows a purely functional model to be natively compiled and executed on a host without having to emulate the functionality of a target ISA. For fast timing simulation, source-level methods employ a hybrid approach that combines the functional simulation with an abstract, statically derived timing model at much coarser program block granularity. This is similar to static Worst-Case Execution Time (WCET) estimation. However, a key challenge is enumeration of possible program paths when performing such static analysis at the whole program level. Source-level approaches avoid or simplify this problem by constructing a static timing model at finer block-level granularity and driving this model with block sequences of program paths encountered in the actual functional simulation. In practice, this is often done by simply back-annotating the timing model directly into the functional source or IR code. Nevertheless, models can be separated, and coarse-grain timing models can equally be combined with fast functional ISS models instead (see also ▶ Chaps. 20, "Precise Software Timing Simulation Considering Execution Contexts" and ▶ 21, "Timing Models for Fast Embedded Software Performance Analysis").

A remaining challenge is that due to pipeline, cache, and other effects, the timing of a program block is not statically fixed but generally depends on the dynamic machine state and hence previous program history. Traditional ISSs simulate detailed interactions with machine state for each encountered instruction. WCET approaches have to derive conservative bounds based on possible program history. By contrast, source-level approaches operate at an intermediate level. The functional simulation allows actual history and state-dependent effects to be accurately tracked. At the same time, only the relevant state is dynamically simulated while statically abstracting as many effects as possible. During static pre-characterization, blocks are analyzed on a target reference model under different possible conditions. Static timing numbers can then be selected based on the actual context encountered in simulation. This can provide accurate timing without replaying the same instruction sequence on the slow timing reference each time the block is executed. However, this is only possible as long as the possible state space affecting block timing is small. For dynamic structures with deep history, such as caches or branch predictors, detailed simulation models that dynamically adjust pre-characterized block timing can be included. Alternatively, a static average- or worst-case analysis can be applied. Ultimately, the amount of static analysis versus dynamic simulation overhead thereby determines the speed and accuracy trade-off in different source-level models.

A typical flow for generating a source-level timing simulation model is shown in Fig. 19.2. There are generally three stages in such a framework: (1) binary

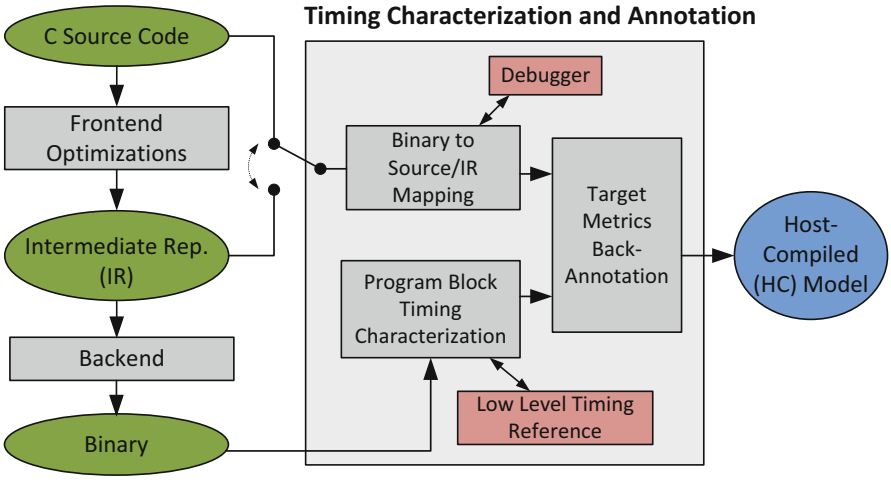

Fig. 19.2 Source-level timing characterization and back-annotation

to source/IR mapping, (2) program block timing characterization, and (3) target metrics back-annotation. During the mapping stage, a relationship between source-/IR-level code and binary need to be established. This usually includes an accurate Control-Flow Graph (CFG) mapping to provide insertion points of target profiling metrics. Additionally, memory accesses also need to be reconstructed in order to account for cache effects. For timing characterization, target metrics are extracted at the machine level at a certain granularity as a one-time effort. Finally, execution statistics are back-annotated into the source/IR model based on the previously determined relationship. The back-annotated source code can then be directly compiled and executed on the simulation host machine for fast and accurate software simulation.

19.2.1 Binary to Source Mapping

The first step in the timing back-annotation flow is to establish a matching between the CFGs of the target binary and source-level code, which is aimed at ultimately allowing target metrics to be annotated back into the source-level or IR code at correct insertion points.

The earliest works estimate and annotate target performance metrics purely based on source code analysis [4, 14]. Control flow and operation information are extracted directly at source level, which are further fed into abstract performance estimators to calculate the corresponding target metrics. These approaches avoid the mapping issues and provide a fast and approximate profiling strategy for early stage design space exploration. However, without consulting the corresponding target binary, such techniques cannot provide precise estimations compared to detailed characterization of binary blocks on a cycle-level reference model.

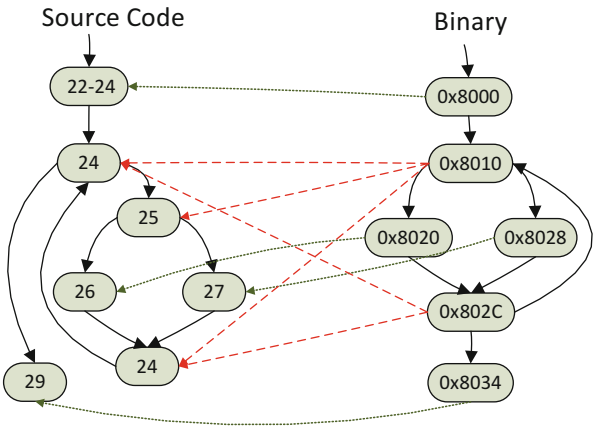

Fig. 19.3 Matching between source- and binary-level control flow

The challenge when working at the binary level is that, due to aggressive compiler optimization, CFGs can be significantly changed when source code is transformed into a binary implementation (Fig. 19.3). Thus, target metrics obtained for each binary block need to be annotated to a matching source code block such that numbers are guaranteed to be accounted for correctly when accumulated during subsequent source-level simulation. Early approaches relied on debug information alone to perform such a mapping [45]. However, debug information provided by standard compilers is often unreliable or ambiguous. For example, debug information often contains no or several source references for the same binary block. Each of these entries describes a potential relation between a binary and source-level basic blocks from which a unique mapping still needs to be determined, i.e., the debug information has to be subject to further analysis steps. These ambiguities and CFG mismatches are the main issues to be resolved to establish a valid and accurate mapping.

When targeting back-annotation at the source level, binary to source matching typically relies on complex structural analysis combined with the use of IR level and debug information to establish a mapping between the target binary and source code [24]. Structural analysis is performed on both source level and binary code to extract loop and control flow dependency characteristics. Along with debug information, these structural properties are then used as matching criteria to establish a tentative CFG mapping. In case of heavily optimized CFGs, advanced approaches annotate an additional IR level [43] or binary path simulation model [39, 41] to dynamically cosimulate, track, and reconstruct the actual binary execution flow and its corresponding accumulated timing. Working at source level benefits from better readability and convenience with respect to manual adjustments or analysis of the simulation model. However, it is usually hard to systematically handle the full range of compiler optimization for general CFG mapping.

Other works address these issues by performing back-annotation and simulation at the IR level. Working at the IR allows typical front-end compiler optimizations to be taken into account, where the IR provides a much closer representation of the final control flow. This simplifies the matching problem, i.e., improves accuracy with little to no penalty in execution speed. Early work [45] only used debug information to perform the mapping, which is more reliable when working at the IR level. Nevertheless, even IR and binary control flows do not always match cleanly due to aggressive compiler backend optimizations. In these cases, similar to advanced source-level approaches, a path tracking model that replicates the CFG of the target binary can be extracted during backend code generation for cosimulation with the IR [3]. However, this adds simulation overhead and requires detailed backend compiler and/or target ISA information to be available, making the approach dependent on the estimation target. Alternatively, a mapping can be established by a simplified structural analysis of both IR and binary CFGs. A general and fully retargetable approach is proposed in [5], where a synchronized depth-first traversal of both CFG is performed to identify legal matches based on a control flow representation using both loop and branch nesting levels. In addition, debug information is consulted when multiple equally likely matches are possible.

19.2.2 Memory Trace Reconstruction

Caches can have a large effect on overall performance estimation. At the same time, cache behavior is highly dynamic and strongly depends on the actual sequence of memory accesses made by the application, which cannot be fully determined statically during program block characterization. Some approaches employ an approximate solution by annotating a statistically calculated average or statically estimated worst-case delay for each memory access [14, 18]. Otherwise, memory accesses need to be reconstructed during the back-annotation stage with information from both binary and source code [21, 27, 44]. The source or IR code is thereby annotated with accurate information about the type and address of each memory access made by the binary code. Alternatively, approaches that already reconstruct a binary cosimulation model for path tracking purposes can equally annotate this model with memory access tracking code obtained from de-compiling the binary [42]. In either case, back-annotated memory accesses can then feed an abstract, dynamic cache simulation model that determines additional cache miss and memory delay penalties to be included during source-level timing simulation (see Sect. 19.3.3).

Memory access trace reconstruction can be generally decomposed into three categories: (1) accesses to static and global variables, (2) accesses to stack data, and (3) accesses to the heap. To reconstruct the addresses of static and global data accesses, their base addresses and, in case of non-scalars, their access offsets are required. Base address information of global data can easily be obtained from the symbol table of the target binary, while access offsets are extracted from analysis of IR or binary code. A key observation is that, with proper translation of primitive data types, access offsets in the IR are the same as in the target binary, while only base

addresses differ. Hence, IR-based approaches can directly obtain such information. By contrast, reconstructing accurate addresses at the source level requires falling back to IR or binary analysis.

Different from the global data, stack and heap accesses are more complicated to back-annotate. Their base addresses change dynamically depending on the local and global execution context. Tracking such memory accesses requires reconstructing the target stack/heap layout as well as the dynamic status of the stack pointer and heap manager during program execution. Thus, abstracted models for stack pointers and heap allocation are usually inserted into source or IR-level simulations to capture and track such information dynamically. Together with access offsets extracted from IR or source code, accurate target stack/heap accesses can then be reconstructed during simulation time.

19.2.3 Block-Level Timing Characterization

The core step in the back-annotation process is the characterization of block-specific target metrics. As mentioned above, accurate characterization is complicated by the fact that target metrics of a program block can be significantly affected by the dynamic machine state and previous program history. In general, the performance metrics for a code block are determined by its internal execution paths as well as the path history of code that has previously executed (Fig. 19.4).

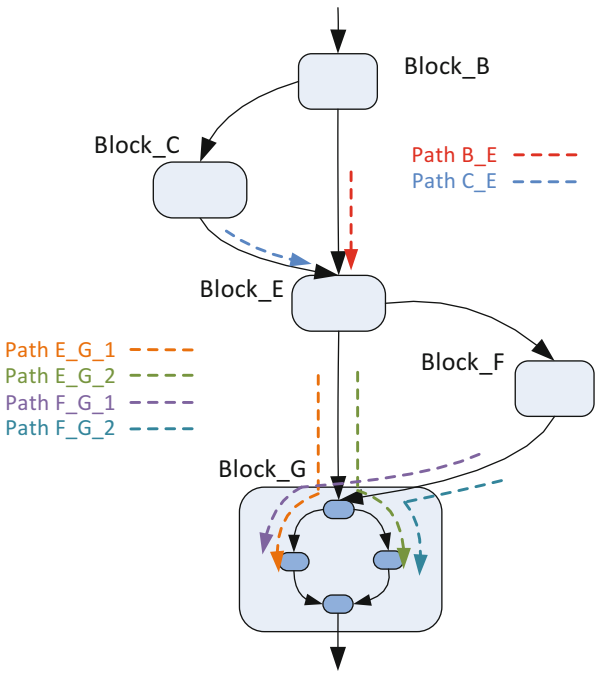

Fig. 19.4 Path dependency of block-level timing characterization

Overall speed and accuracy are determined by the granularity of code blocks at which characterization (and ultimately back-annotation are performed). Approaches that operate at a coarse function level [4] have to estimate average or worst-case behavior across all dynamic execution paths taken within each function body as determined, for example, by a previous profiling run. By contrast, solutions that annotate timing at the individual statement level [22] suffer from unnecessary characterization and simulation overhead. Most existing approaches instead work at a basic block level. Since there is only a single path through a basic block, the assumption is that its baseline timing can be accurately represented by a single, statically characterized number.

Machine state and execution history can, however, still significantly affect a block's timing. Such effects can be estimated by employing a WCET analysis for binary timing characterization, which provides an upper bound on the execution time for each individual basic block either alone or within its larger execution context [41]. In most cases, however, basic block timing is characterized through cycle-accurate simulation on a detailed microarchitecture reference model. Pipeline history effects are taken into account by characterizing each block in sequence with possible predecessors, such that variations in execution context are accurately accounted for. The characterization overhead thereby exponentially increases with the number of possible predecessors and depth of considered execution path history. Depending on the binary block length and machine pipeline depth, a maximum number of predecessors that can possibly affect dependencies until they are guaranteed to have percolated through the pipeline can be dynamically determined for each block and path [18]. Alternatively, blocks can simply be characterized through pairwise execution with all of their immediate predecessors, which has been shown to provide a good trade-off between accuracy and estimation complexity [5].

19.2.4 Back-Annotation

The final step in generating a source-level software simulation model is back-annotation of characterized timing metrics using previously obtained mapping information. Metrics gathered during the characterization step are usually recorded in a mapping table, which is then used for directing the annotation of target metrics into the IR or source code at correct insertion points.

For a single annotation unit, there are often multiple performance metrics accounting for path-dependent timing effects. In order to be able to pick up the correct set of metrics during simulation, the back-annotation process will usually insert extra data structures to record essential execution history, such as the dynamic predecessor of currently executing basic block. In this way, the back-annotated model can reconstruct the binary execution flow and properly accumulate block execution time along with the annotated points.

To account for dynamic execution characteristics that depend on complex history behavior, such as branch predictors [8] and caches (see Sect. 19.3.3), the source code is further augmented with calls to dynamic simulation models of such (micro-)

architectural structures. In case of caches, this includes annotating the code with previously reconstructed memory and cache access information. During simulation, delays can be adjusted according to the corresponding outcomes from such models. For this purpose, blocks are usually characterized assuming ideal conditions (such as always-correct branch prediction or perfect caches), where dynamically determined penalties are added during simulation. This approach is feasible for simpler in-order processors. By contrast, dynamic tracking of complex interactions between pipelines and other structures in out-of-order processors requires a significantly more involved characterization and back-annotation [26]. In all cases, the choice of annotating static estimates or dynamic simulation models, which incur additional, in some cases, significant simulation overhead, enables generation of different models with varying trade-off between simulation speed and accuracy. Furthermore, such source-level simulation approaches can be extended beyond timing to back-annotation of other performance, energy, reliability, power, and thermal (PERPT) metrics [5, 9, 17, 47].

19.3 Host-Compiled OS and Processor Modeling

As described previously, host-compiled simulators extend pure source-level approaches with fast yet accurate models of the complete software execution environment. Figure 19.5 shows a typical layered organization of a host-compiled simulation model [30, 35, 38]. Individual source-level application models that are annotated with timing and other metrics as described in Sect. 19.2 are converted

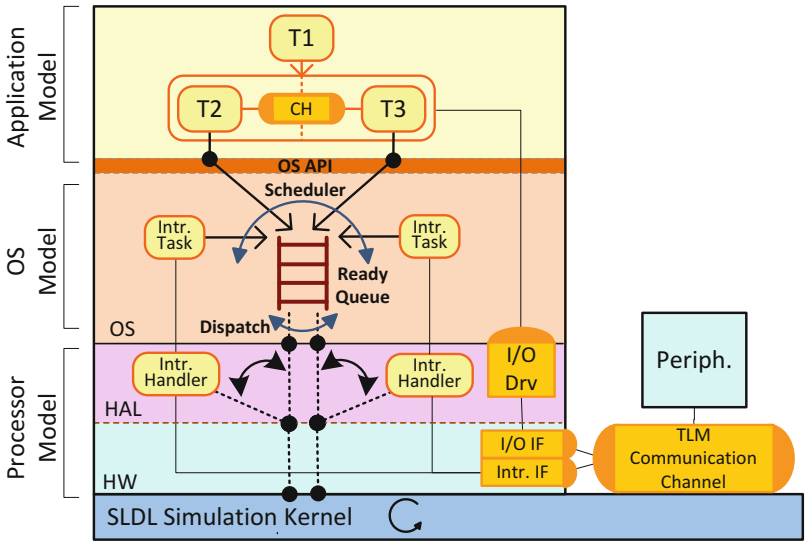

Fig. 19.5 Host-compiled simulation model

into tasks running on top of an abstract, canonical OS Application Programming Interface (API). Tasks are grouped and encapsulated according to a given partitioning to model the multi-threaded application mix running on each processor of an overall MPSoC. Within each processor, an OS model then provides an implementation of the OS API to manage tasks and replicate a specific single- or multi-core scheduling strategy. The OS model itself sits on top of models of the firmware and drivers forming a Hardware Abstraction Layer (HAL). An underlying Hardware (HW) layer in turn provides interfaces to external TLMs of the communication infrastructure (Sect. 19.4). Finally, the complete processor model is integrated and cosimulated with other system components on top of an SLDL. The SLDL simulation kernel thereby provides the basic concurrency and synchronization services for OS, processor, and system modeling.

19.3.1 OS Modeling

An OS model generally emulates scheduling and interleaving of multiple tasks on one or more cores [11, 12, 16, 23, 28, 31, 46]. It maintains and manages tasks in a set of internal queues similar to real operating systems. In contrast to porting and paravirtualizing a real OS to run on top of the modeled HAL, a lightweight OS model can provide an accurate emulation of real OS behavior with little to no overhead [35]. Tasks are modeled as parallel simulation threads on top of the underlying SLDL kernel. The OS model then provides a thin wrapper around basic SLDL event handling and time management primitives, where SLDL calls for advancing simulation time, event notification, and wakeup in the application model are replaced with calls to corresponding OS API methods. This allows the OS model to suspend, dispatch, and release tasks as necessary on every possible scheduling event, i.e., whenever there is a potential change in task states. An OS model will typically also provide a library of higher-level channels built around basic OS and SLDL primitives to emulate standard application-level Inter-Process Communication (IPC) mechanisms.

Figure 19.6 shows an example trace of two tasks T_0 and T_1 running on top of an OS model emulating a time slice-based round-robin scheduling policy on a single core [30]. Source-level execution times of tasks are modeled as calls to wait-for-time methods in the OS API. On each such call, the OS model will advance the simulated time in the underlying SLDL kernel but will also check whether the time slice is expired and switch tasks if this is the case. In order to simulate such a context switch, the OS model suspends and releases tasks on events associated with each task thread at the SLDL level. Overall, the OS model ensures that at any simulated time, only one task is active in the simulation. Note that this is different from scheduling performed in the SLDL kernel itself. Depending on available host resources, the SLDL kernel may serialize simulation threads in physical time. By contrast, the OS model serializes tasks in the simulated world, i.e., in logical time.

Within an isolated set of tasks on a core, this approach allows OS models to accurately replicate software timing behavior for arbitrary scheduling policies.

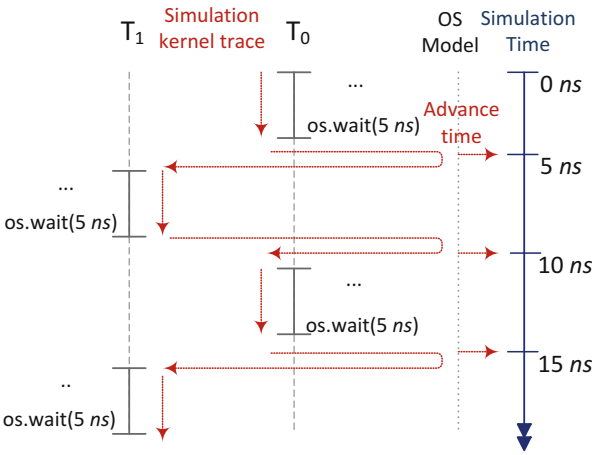

Fig. 19.6 Example of OS model trace

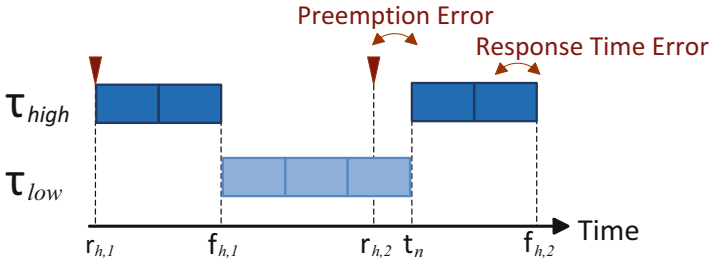

Fig. 19.7 Inherent preemption inaccuracies in discrete OS models

However, the discrete nature of such models introduces inherent inaccuracies in the presence of asynchronous scheduling events, such as task releases triggered by external interrupts or by events originating on other cores. Since the OS model advances (simulated) time only in discrete steps, it will not be able to react to such events immediately. Figure 19.7 shows an example of a low-priority task τ_{low} being preempted by a high-priority task τ_{high} triggered externally. In reality, the high-priority task is released at time $r_{h,2}$. In the simulation, however, the OS model is not able to perform the corresponding task switch until the next simulation step is reached at time t_n. This results in a corresponding preemption and response time error for both tasks (with τ_{low} potentially finishing too early).

As shown in the example of Fig. 19.7, the preemption error is generally upper bounded by the maximum timing granularity. By contrast, it can be shown that response time errors can potentially become much larger than the time steps themselves [33]. This is, for example, the case if τ_{low} in Fig. 19.7 finishes too early but should have been preempted and delayed by a very long running τ_{high}. This can be a serious problem for evaluation of real-time system guarantees. Adjusting

the timing granularity does not generally help to improve the maximum simulation error. Nevertheless, decreasing the granularity will reduce the likelihood of such large errors occurring, i.e., will improve average simulation accuracy.

At the same time, the timing granularity also influences simulation speed. A fine granularity allows the model to react quickly but increases the number of time steps, context switches and hence overhead in the simulator. Several approaches have been proposed to overcome this general trade-off and provide a fast coarse-grain simulation while maintaining high accuracy. Existing approaches are either optimistic [37] or conservative [32]. In optimistic solutions, a lower-priority task is speculatively simulated at maximum granularity assuming no preemption will occur. If a preemption occurs while the task is running, the higher-priority task is released concurrently at its correct time. In parallel, all disturbing influences are recorded and later used to correct the finish time of the low-priority task(s). Such an approach has also been used to model preemptive behavior in other contexts, such as in TLMs of busses with priority-based arbitration [36] (see also Sect. 19.4.2). Note that unless a full rollback is possible in the simulator, optimistic approaches cannot guarantee an accurate order of all task events and interactions, such as shared variable accesses. By contrast, in conservative approaches, at any scheduling event, the closest possible preemption point is predicted to select a maximum granularity not larger than that. If no prediction is possible, the model falls back onto a fine default granularity or a kernel mechanism that allows for coarse time advances with asynchronous interruptions by known external events. Conservative approaches, by their nature, always maintain the correct task order. In both optimistic and conservative approaches, the OS model will automatically, dynamically, and optimally accumulate or divide application-defined task delays to match the desired granularity. This allows the model to internally define a granularity that is independent from the granularity of the source-level timing annotations. Furthermore, both types of approaches are able to completely avoid preemption errors and associated issues with providing worst-case guarantees.

19.3.2 Processor Modeling

Host-compiled processor models extend OS models with accurate representations of drivers, interrupt handling chains, and integrated hardware components, such as caches and TLM bus interfaces [2, 10, 15, 38]. Specifically, accurate models of interrupt handling effects can contribute significantly to overall timing behavior and hence accuracy [35].

The software side of interrupt handling chains is typically modeled as special, high-priority interrupt handler tasks within the OS model [46]. On the hardware side, models of external generic interrupt controllers (GICs) interact with interrupt logic in the processor model's hardware layer. The OS model is notified to suspend the currently running task and switch to a handler whenever an interrupt for a specific core is detected. At that point, the handler becomes a regular OS task,

which can in turn notify and release other interrupt or user tasks. By back-annotating interrupt handlers and tasks with appropriate timing estimates, an accurate model of interrupt handling delays and their performance impact can be constructed.

An example trace for a complete host-compiled simulation of two task sets with three tasks each running on a dual-core platform is shown in Fig. 19.8 [30]. Task sets are mapped to run on separate cores, and the highest priority tasks are modeled as periodic. All interrupts are assigned to Core_0. The trace shows a conservative OS model using dynamic prediction of preemptions. The model is in a fine-grain fallback mode whenever there is a higher-priority task or handler waiting for an unpredictable external event. In all other cases, the model switches to a predictive mode using accumulation of delays. Note that high-priority interrupt handlers and tasks are only considered for determining the mode if any schedulable tasks is waiting for the interrupt. This allows the model to remain in predictive mode for the majority of time. Handlers and tasks themselves can experience large errors during those times. However, under the assumption that they are generally short and given that no regular task can be waiting, accuracy losses will be small.

When applied to simulation of multi-threaded, software-only Posix task sets on a single-, dual-, and quad-core ARM-Linux platform, results show that host-compiled OS and processor models can achieve average simulation speeds of 3,500 MIPS with less than 0.5% error in task response times [35]. When integrating processor models into a SystemC-based virtual platform of a complete audio/video MPSoC, more than 99% accuracy in frame delays is maintained. For some cases, up to 50% of the simulated delays and hence accuracy is attributed to accurately modeling the Linux interrupt handling overhead. Simulation speeds, however, drop to 1,400 MIPS. This is due to the additional overhead for cosimulation of HW/SW interactions through the communication infrastructure. Methods for improving performance of such communication models will be discussed in Sect. 19.4.

19.3.3 Cache Modeling

Next to external communication and synchronization interfaces, a host-compiled processor simulator will generally incorporate timing models for other dynamic aspects of the hardware architecture. Specifically, timing effects of caches and memory hierarchies are hard to capture accurately as part of a static source-level back-annotation. Hit/miss rates and associated delay penalties depend heavily on the execution history and the specific task interactions seen by the processor. To accurately model such dynamic effects, a behavioral cache simulation can be included [25, 27, 42, 44].

As described in Sect. 19.2, the source level can be annotated to re-create accurate memory access traces during simulation. Such task-by-task traces can in turn drive an abstract cache model that tracks history and hit/miss behavior for each access. Resulting penalties can then be used to dynamically update source-level timing annotations. Note that cache models only need to track the cache state in terms of line occupancy. The data itself is natively handled within the simulation host.

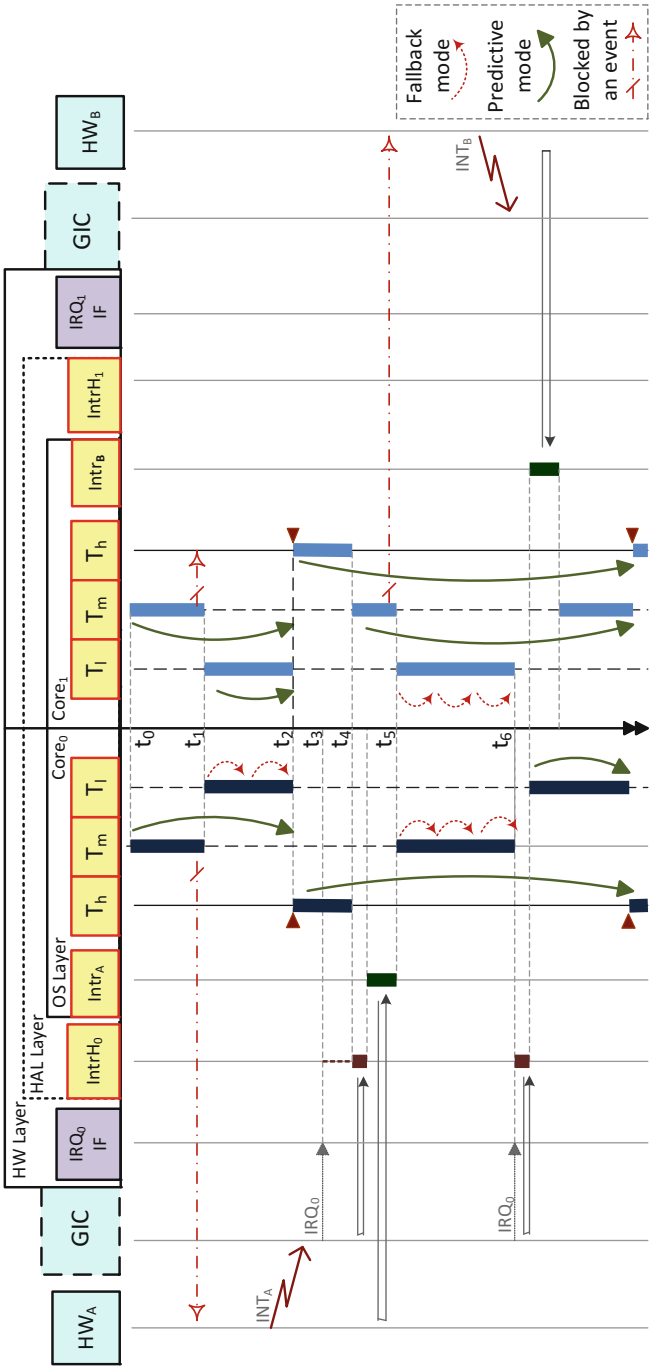

Fig. 19.8 Host-compiled simulation trace

When combined with an OS model, such an approach allows for accurate modeling of cache pollution and interference among different tasks. A particular challenge emerges, however, when multiple cores can interfere through a shared cache. A cache model can accurately track shared state, including coherency effects across multiple cache levels, as long as individual core models issue cache accesses in the correct global order. As mentioned above (Sect. 19.3.2), this is generally not the case in a coarse-grain, temporally decoupled simulation. Cores may produce outgoing events ahead of each other, and, as a result, multiple cores may commit their accesses to the cache globally out-of-order. At the same time, from a speed perspective, it is not feasible to decrease granularity to a point where each memory access is synchronized to the correct global time.

Several solutions have been proposed to tackle this issue and provide a fast yet accurate multi-core out-of-order cache (MOOC) simulation in the presence of temporal decoupling [34, 40]. The general approach is to first collect individual accesses from each core including accurate local time stamps. Later, once a certain threshold is reached, accesses are reordered and committed to the cache in their globally correct sequence. Figure 19.9 illustrates this concept [34]. In this approach, both cores first send accesses to a core-specific list maintained in the cache model. After each time advance, cores notify the cache to synchronize and commit all accesses collected up to the current time. It is thereby guaranteed that all other cores have advanced and produced events up to at least the same time.

An added complication are task preemptions [30]. Since cores and tasks can run ahead of time, a task may generate accesses that would otherwise not have

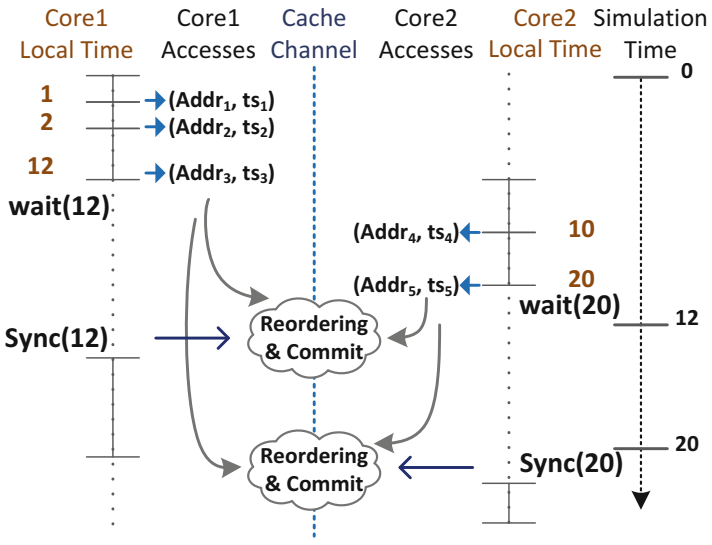

Fig. 19.9 Multi-core out-of-order cache simulation trace

been issued until after a possible preemption is completed. This requires access reordering to be tightly integrated with the OS model. By maintaining task-specific access lists in the OS model instead of the cache, the OS can adjust remaining time stamps by the duration of the preemption whenever such a preemption occurs. Overall, such an approach can maintain 100% accuracy of cache accesses at the speed of a fully decoupled simulation.

In other approaches, the cache model is moved outside of the processor to become part of the TLM backplane itself [40]. In this case, the cache is accessed via regular bus transactions, and all of the reordering is relegated to a so-called quantum giver within a temporally decoupled TLM simulation (see Sect. 19.4.3). Note that this still requires OS model support to generate accurate transaction time stamps in the presence of preemptions. Similar reordering techniques can then also be applied to other shared resources, such as busses, as will be shown in the following sections.

19.4 TLM Communication for Host-Compiled Simulation

Embedded and integrated systems are comprised of many communicating components as we move toward embedded multi-core processors. Fast simulation requires advanced communication models at transaction level. Usually, scheduling events of the simulation kernel are closely coupled to the communication events. For high-speed virtual prototyping, usually the blocking transaction style, called loosely-timed TLM in SystemC [1], is applied. Blocking transactions can be synchronized to the global simulation time at each accessed module. As many communication resources such as busses or target (slave) modules are shared between initiator (masters) modules, accurate models additionally schedule an arbitration event at each arbitration cycle.

Novel works have shown that these requirements can be usually relaxed to improve simulation speed. These works either raise the abstraction of the communication, e.g., a single simulated block transaction represents a set of bus transactions performed by the HW/SW system, or apply TD. TD implies that initiators perform accesses, which are located in the future with respect to the current global simulation time. This leads to several challenges, as discussed in Sect. 19.1.3. An overview of selected communication models is given in the following.

19.4.1 TD with No Conflict Handling

The TLM-2.0 standard offers the Quantum Keeper. The TLM-2.0 Quantum Keeper provides a global upper bound to the local time offset. The Quantum Keeper is easily applicable to realize temporal decoupling. It offers no standard way to handle data dependencies or resource conflicts. Shared variables have to be protected by additional synchronization methods. Figure 19.10a shows a message diagram for an example. We assume blocking TLM communication style indicated by the

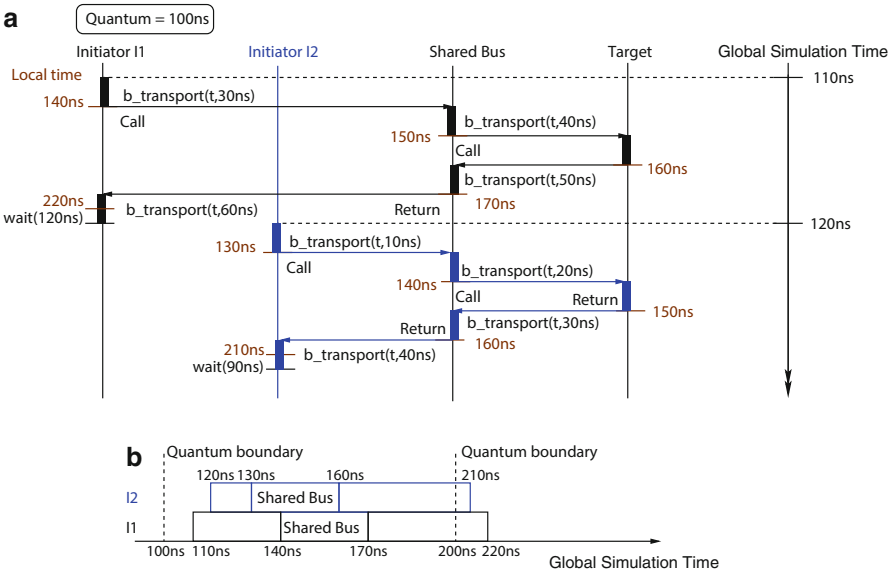

Fig. 19.10 Temporal decoupling with Quantum Keeper

b_transport function. The b_transport function propagates as its argument of the offset between the local simulation time and the global simulation time. Both initiators are only executed once per quantum until their local time exceeds the next quantum boundary. The communication is out-of-order. Transaction of initiator I1 to the shared bus starts at 140 ns, yet it is executed before the transaction of I2 starting at 130 ns. Additionally, the transactions concurently access the shared bus as shown in Fig. 19.10b, which would lead to arbitration. Yet, I1 can finish its transaction without conflict delay because the transaction of I2 was not yet known at the shared bus. Simulation speed is highest, but communication timing is optimistic, and out-of-order accesses may lead to incorrect simulation results. Thus, several methods for using system knowledge for improving accuracy in TLM communication using TD were proposed, which are presented in the following.

19.4.2 TD with Conflict Handling at Transaction Boundaries

In [29, 36], the additional delay due to resource conflicts are resolved at the transaction boundaries. At the start of a transaction, the communication state is inspected in [36]. If a higher-priority transaction is ongoing, the end time of the considered transaction is computed accordingly. Yet, still an optimistic end time is computed at the beginning of a transaction because future conflicting transactions are not considered. When the end time of the transaction is reached, additional delay due to other conflicting transaction is retroactively added. In the case of [36], another

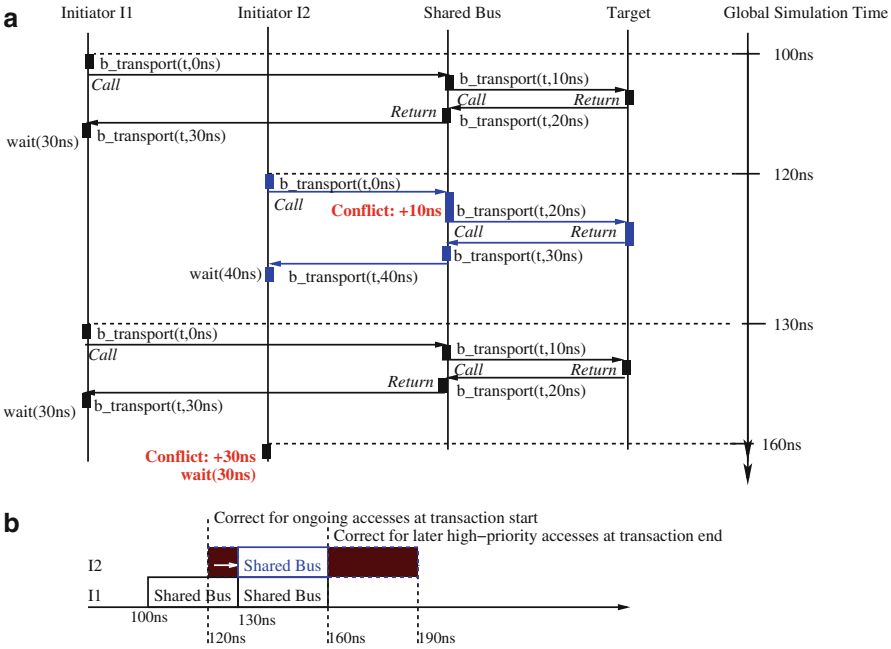

Fig. 19.11 Handling conflicts at transaction boundaries

wait is issued to account for the additional delay, but intelligent event re-notification could also be applied. The method needs minimally one context switch (single call to wait()) per transaction. An example is shown in Fig. 19.11 with I1 having higher priority on the shared bus. The transaction of I2 is delayed during execution due to the conflict with the first transaction of I1. As I1 issues another transaction, the delay for I2 is adapted with another call to wait to consider the second conflict. In [29], a similar approach is presented that handles conflicts at the transaction boundaries. It additionally combines several atomic transactions into block transactions. If these block transactions get preempted, the transactions are split to assure that the order of data accesses is preserved.

19.4.3 TD with Conflict Handling at Quantum Boundaries

With the Quantum Giver [40], each initiator can issue multiple transactions until its individual local quantum is exceeded during the so-called simulation phase. All transactions in one quantum are executed instantaneously but use time stamping to record their start times. After the quantum is reached, the initiator informs a central timing manager, the so-called Quantum Giver, and waits for an end event. During the scheduling phase, the Quantum Giver retroactively orders all transactions according to their time stamps. It computes the delays due resource

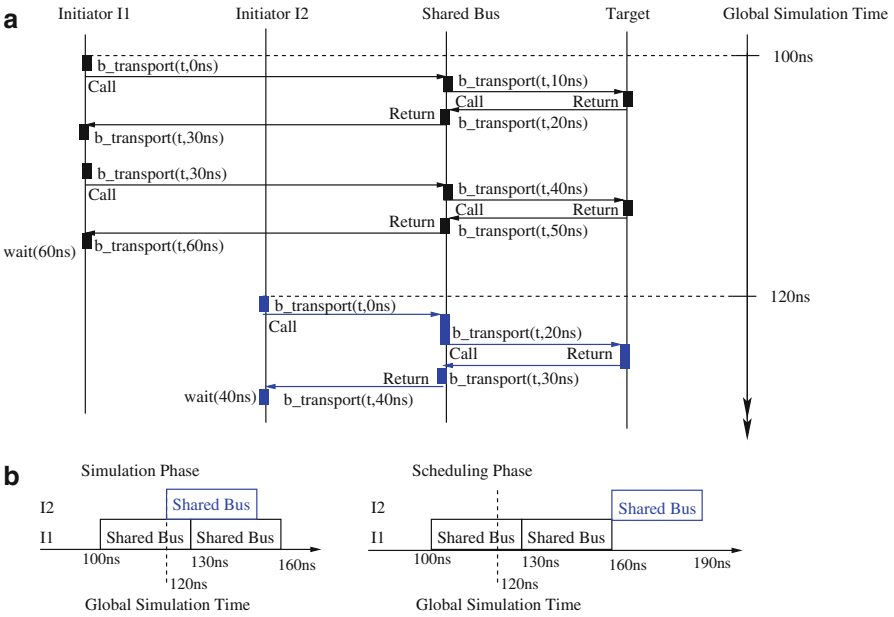

Fig. 19.12 Handling conflicts with Quantum Giver

conflicts and resolves all dependencies. According to the conflicts, the end event of each initiator is notified at the correct time. Finally, the quantum of each initiator is adjusted for the next simulation phase. The concept is illustrated in Fig. 19.12. During the simulation phase, transaction on the shared bus still overlap. The resulting delays are computed in the scheduling phase by traversing the list of transaction ordered by their starting time. The method also considers that conflicts on different shared resources might have impact on each other. This method targets fast simulation with temporal decoupling. Only a single context switch is required in each quantum, which may include several transactions. Yet, the transactions are executed immediately; thus, out-of-order accesses to shared variables must be avoided with additional synchronization guards. In [13], Advanced Temporal Decoupling (ATD) is presented. It applies TD but is a conservative approach that preserves access order. The initiators may advance their local time until they meet an inbound data dependency, e.g., a read on a shared variable. All write transactions performed on shared data are buffered by an additional communication layer. After all initiators have completed execution, the transactions are ordered, and the write transactions are completed according to their start time together with pending read transactions. This preserves the correct access order. So-called Temporal Decoupled Semaphores handle resource conflicts and compute arbitration delays. The ATD communication model is implemented in a transparent TLM (TTLM) library, which hides the implementation details from the model developer.

19.4.4 Abstract TLM+ with Conflict Handling at SW Boundaries

TLM+ is a SW-centric communication model for processors, which can only be applied for host-compiled SW simulation [7]. It not only applies TD but raises the abstraction of communication. Usually a driver function does not transfer a single data item but a range of control values together with a possible block of data. Execution of a driver function involves a complete set of bus transactions from the processor. This set of bus transactions is abstracted into a single TLM+ block transaction. The HW/SW interface is adapted in the host-compiled simulation. The software could, thus, not simulate on an ISS. Conflicts at shared resources are handled by a central timing manager, the so-called resource model. In order to give good estimates on the delay due to conflicts, the resource model requires to save a communication profile of the original driver function [19]. This profile allows to extract a demand for communication resources. Usually, a driver function would not block a shared bus completely. Accesses of different cores accessing other modules would interleave, as driver functions are executed concurrently. Yet both cores may suffer from additional delays due to arbitration conflicts. Analytic demand-availability estimators inside the resource model can be used to estimate these delays [20].

The scheduling is conducted by the resource model at the transaction boundaries. These boundaries then correspond to the entry and exit to the respective software driver function. The concept is illustrated in Fig. 19.13. Initiator I1 first executes a block transaction trigged by the execution of a driver function. At a later point, I2 starts its block transaction. I1 has higher priority; therefore, I2 is scheduled to take longer as it has not the full resource availability on the shared bus. When the block transaction of I1 finishes, I2 is not subject to further high-priority traffic blocking its bus accesses. Its end time is rescheduled to an earlier end time. This is done by event re-notification, which leads to a single call to wait() for each block transaction.

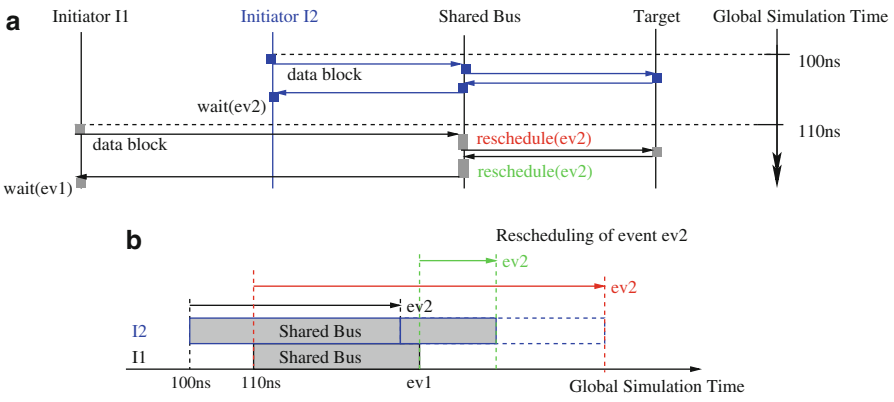

Fig. 19.13 Handling conflicts in TLM+

TLM+ targets very high abstraction and faster simulation compared to the other TLM communication methods. Yet, it does not execute the original SW because the driver functions are replaced by abstract TLM+ counterparts.

19.5 Summary and Conclusions

With time to market shrinking day by day, developing fast and accurate models is no more a luxury or good-to-have-methodology. It is essential for companies to invest in making software models for meeting their time to market. However, the fastest model is not a good model if it does not accurately match or predict the final design reality. Therefore, new methods are required that enable efficient but accurate simulation of HW/SW systems. Next-generation virtual prototypes based on host-compiled software simulation can provide such ultra-fast yet highly accurate modeling solutions.

Acknowledgments The authors acknowledge Oliver Bringmann, Wolfgang Müller, and Zhuoran Zhao for their contributions in Sects. 19.2 and 19.3.

References

1. 1666–2011 – IEEE standard for standard SystemC language reference manual (2012)
2. Bouchhima A, Bacivarov I, Youssef W, Bonaciu M, Jerraya A (2005) Using abstract CPU subsystem simulation model for high level HW/SW architecture exploration. In: Proceedings of the Asia and South Pacific design automation conference (ASPDAC)
3. Bouchhima A, Gerin P, Petrot F (2009) Automatic instrumentation of embedded software for high level hardware/software co-simulation. In: Proceedings of the Asia and South Pacific design automation conference (ASPDAC)
4. Cai L, Gerstlauer A, Gajski D (2004) Retargetable profiling for rapid, early system-level design space exploration. In: Proceedings of the 41st annual conference on design automation. ACM, San Diego, pp 281–286. doi:10.1145/996566.996651. http://portal.acm.org/citation.cfm?id=996566.996651
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. Chiou D, Sunwoo D, Kim J, Patil NA, Reinhart W, Johnson DE, Keefe J, Angepat H (2007) FPGA-accelerated simulation technologies (FAST): fast, full-system, cycle-accurate simulators. In: Proceedings of the international symposium on microarchitecture (MICRO)
7. Ecker W, Esen V, Schwencker R, Steininger T, Velten M (2010) TLM+ modeling of embedded hw/sw systems. In: Design, automation test in Europe conference exhibition (DATE), pp 75–80
8. Faravelon A, Fournel N, Petrot F (2015) Fast and accurate branch predictor simulation. In: Proceedings of the design automation and test in Europe conference. ACM, pp 317–320
9. Gandhi D, Gerstlauer A, John L (2014) FastSpot: host-compiled thermal estimation for early design space exploration. In: Proceedings of the international symposium on quality electronic design (ISQED)
10. Gerin P, Shen H, Chureau A, Bouchhima A, Jerraya A (2007) Flexible and executable hardware/software interface modeling for multiprocessor SoC design using SystemC. In: Proceedings of the Asia and South Pacific design automation conference (ASPDAC)

11. Gerstlauer A, Yu H, Gajski D (2003) RTOS modeling for system level design. In: Proceedings of the design, automation and test in Europe (DATE) conference
12. He Z, Mok A, Peng C (2005) Timed RTOS modeling for embedded system design. In: Proceedings of the real time and embedded technology and applications symposium (RTAS)
13. Hufnagel S (2014) Towards the efficient creation of accurate and high-performance virtual prototypes. Ph.D. thesis. https://kluedo.ub.uni-kl.de/frontdoor/index/index/docId/3892
14. Hwang Y, Abdi S, Gajski D (2008) Cycle-approximate retargetable performance estimation at the transaction level. In: Proceedings of the design, automation and test in Europe (DATE) conference
15. Kempf T, Dorper M, Leupers R, Ascheid G, Meyr H, Kogel T, Vanthournout B (2005) A modular simulation framework for spatial and temporal task mapping onto multi-processor SoC platforms. In: Proceedings of the design, automation and test in Europe (DATE) conference
16. Le Moigne R, Pasquier O, Calvez JP (2004) A generic RTOS model for real-time systems simulation with SystemC. In: Proceedings of the design, automation and test in Europe (DATE) conference
17. Lee CM, Chen CK, Tsay RS (2013) A basic-block power annotation approach for fast and accurate embedded software power estimation. In: Proceedings of the international conference on very large scale integration (VLSI-SoC)
18. Lin KL, Lo CK, Tsay RS (2010) Source-level timing annotation for fast and accurate TLM computation model generation. In: Proceedings of the Asia and South Pacific design automation conference (ASPDAC)
19. Lu K, Muller-Gritschneder D, Schlichtmann U (2012) Accurately timed transaction level models for virtual prototyping at high abstraction level. In: Design, automation test in Europe conference exhibition (DATE), pp 135–140
20. Lu K, Muller-Gritschneder D, Schlichtmann U (2013) Analytical timing estimation for temporally decoupled tlms considering resource conflicts. In: Design, automation test in Europe conference exhibition (DATE), pp 1161–1166
21. Lu K, Muller-Gritschneder D, Schlichtmann U (2013) Memory access reconstruction based on memory allocation mechanism for source-level simulation of embedded software. In: Proceedings of the Asia and South Pacific design automation conference (ASP-DAC)
22. Meyerowitz T, Sangiovanni-Vincentelli A, Sauermann M, Langen D (2008) Source-level timing annotation and simulation for a heterogeneous multiprocessor. In: Proceedings of the design, automation and test in Europe (DATE) conference
23. Miramond B, Huck E, Verdier F, Benkhelifa MEA, Granado B, Aichouch M, Prevotet JC, Chillet D, Pillement S, Lefebvre T, Oliva Y (2009) OveRSoC: a framework for the exploration of RTOS for RSoC platforms. Int J Reconfig Comput 2009(450607):1–18
24. Mueller-Gritschneder D, Lu K, Schlichtmann U (2011) Control-flow-driven source level timing annotation for embedded software models on transaction level. In: EUROMICRO conference on digital system design (DSD)
25. Pedram A, Craven D, Gerstlauer A (2009) Modeling cache effects at the transaction level. In: Proceedings of the international embedded systems symposium (IESS)
26. 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)
27. Posadas H, Díaz L, Villar E (2011) Fast data-cache modeling for native co-simulation. In: Proceeding of the Asia and South Pacific design automation conference (ASPDAC)
28. Posadas H, Damez JA, Villar E, Blasco F, Escuder F (2005) RTOS modeling in SystemC for real-time embedded SW simulation: a POSIX model. Des Autom Embed Syst 10(4):209–227
29. Radetzki M, Khaligh R (2008) Accuracy-adaptive simulation of transaction level models. In: Design, automation and test in Europe, DATE'08, pp 788–791
30. Razaghi P (2014) Dynamic time management for improved accuracy and speed in host-compiled multi-core platform models. Ph.D. thesis, The University of Texas at Austin

31. Razaghi P, Gerstlauer A (2011) Host-compiled multicore RTOS simulator for embedded real-time software development. In: Proceedings of the design, automation test in Europe (DATE) conference
32. Razaghi P, Gerstlauer A (2012) Automatic timing granularity adjustment for host-compiled software simulation. In: Proceedings of the Asia and South Pacific design automation conference (ASPDAC)
33. Razaghi P, Gerstlauer A (2012) Predictive OS modeling for host-compiled simulation of periodic real-time task sets. IEEE Embed Syst Lett (ESL) 4(1):5–8
34. Razaghi P, Gerstlauer A (2013) Multi-core cache hierarchy modeling for host-compiled performance simulation. In: Proceedings of the electronic system level synthesis conference (ESLsyn)
35. Razaghi P, Gerstlauer A (2014) Host-compiled multi-core system simulation for early real-time performance evaluation. ACM Trans Embed Comput Syst (TECS) 13(5s). http://dl.acm.org/citation.cfm?id=2660459.2678020
36. Schirner G, Dömer R (2007) Result oriented modeling a novel technique for fast and accurate TLM. IEEE Trans Comput Aided Des Integr Circuits Syst (TCAD) 26(9):1688–1699
37. Schirner G, Dömer R (2008) Introducing preemptive scheduling in abstract RTOS models using result oriented modeling. In: Proceedings of the design, automation and test in Europe (DATE) conference
38. Schirner G, Gerstlauer A, Dömer R (2010) Fast and accurate processor models for efficient MPSoC design. ACM Trans Des Autom Electron Syst (TODAES) 15(2):10:1–10:26
39. 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
40. Stattelmann S, Bringmann O, Rosenstiel W (2011) Fast and accurate resource conflict simulation for performance analysis of multi-core systems. In: Design, automation test in Europe conference exhibition (DATE), 2011
41. 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)
42. Stattelmann S, Gebhard G, Cullmann C, Bringmann O, Rosenstiel W (2012) Hybrid source-level simulation of data caches using abstract cache models. In: Proceedings of the design, automation test in Europe (DATE) conference
43. Wang Z, Henkel J (2012) Accurate source-level simulation of embedded software with respect to compiler optimizations. In: Proceedings of the design, automation test in Europe (DATE) conference
44. Wang Z, Henkel J (2013) Fast and accurate cache modeling in source-level simulation of embedded software. In: Design, automation test in Europe conference exhibition (DATE), pp 587–592. doi:10.7873/DATE.2013.129
45. Wang Z, Herkersdorf A (2009) An efficient approach for system-level timing simulation of compiler-optimized embedded software. In: Proceedings of the design automation conference (DAC)
46. Zabel H, Müller W, Gerstlauer A (2009) Accurate RTOS modeling and analysis with SystemC. In: Ecker W, Müller W, Dömer R (eds) Hardware-dependent software: principles and practice. Springer, Berlin
47. Zhao Z, Gerstlauer A, John LK (2017) Source-level performance, energy, reliability, power and thermal (PERPT) simulation. IEEE Trans Comput Aided Des Integr Circuits Syst (TCAD) 36(2):299–312