The Next Generation of Virtual Prototyping: Ultra-fast Yet Accurate Simulation of HW/SW Systems

Oliver Bringmann[1], Wolfgang Ecker[2], Andreas Gerstlauer[3], Ajay Goyal[2],
Daniel Mueller-Gritschneder[4], Prasanth Sasidharan[2], Simranjit Singh[2]
[1]University of Tuebingen, Germany [2]Infineon Technologies
[3]University of Texas at Austin, USA, [4]Technische Universitaet Muenchen, Germany
oliver.bringmann@uni-tuebingen.de, gerstl@ece.utexas.edu, ajay.goyal@infineon.com, daniel.mueller@tum.de

***Abstract*—Virtual Prototypes (VPs) have been now widely adopted by industry as platforms for early SW development, HW/SW co-verification, 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 and efficient hardware peripheral models. Additionally, an industrial flow for efficient model development is outlined. This support of ultra-fast and accurate HW/SW co-simulation will be a key enabler for successfully developing tomorrow's multi-processor system-on-chip (MPSoC) platforms.**

I. Introduction

Due to increased complexity of modern embedded and integrated systems, more and more design companies are adapting virtual prototyping methods. Next to obtaining correct hardware with less iterations, virtual prototypes (VPs) support early SW development, performance analysis, HW/SW co-verification 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 and peripheral models. Intelligent methods are applied to preserve simulation accuracy at ulta-high simulation speeds. These methods are independent of the used System-level Design 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 paper.

In this paper, we present an overview of state-of-the-art approaches for ultra-fast yet highly-accurate next-generation VPs. The paper is structured as follows: Basic concepts of virtual prototyping for HW/SW systems are discussed in Sec. II and the major differences between traditional and next-gerneration VPs are outlined. Source-level software simulation methods are discussed in Sec. III. OS and processor models are presented in Sec. IV. Communication models are shown in Sec. V and peripheral models in Sec. VI. Finally, an industrial model development flow is shown in Sec. VII. Sec. VIII concludes.

II. Virtual Prototypes (VPs)

In 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 buses. In the context of this work, a VP is a computer model of such a HW/SW system.

A. Traditional VP Simulation

The VP is simulated via a 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 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. This 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 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 pre-defined 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.

B. Next-Generation VPs

Simulation speed can be increased by reducing the number of scheduling events. One possibility is to raise the layer of abstraction. Less detailed simulation usually leads to less scheduling events. This includes methods for so-called Temporal Decoupling (TD). TD lets simulation processes run ahead of the logical simulation time to increase the simulated timing granularity and decrease the number of scheduling events. The logical simulation time of the kernel is referred to as *global simulation time*. In contrast, the components can keep track of their time using a *local simulation time*, which is usually defined as an offset to the global simulation time. TD and other abstractions can increases simulation speed significantly, but

may decrease accuracy due to loss of timing information or missing synchronization events.

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. This information is obtained by establishing an accurate relation between the source code of a program and its machine code. By comparing the structure of both program representations, timing information of the software can be extracted at the machine level and included in the source code for performance analysis. 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.

Pure source-level simulation approaches focus on emulating standalone 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 compromised of many communicating components as we move towards 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 retro-active correction of the timing behavior. This correction is performed by a central timing manager, a so-called resource model. Finally, peripherals with cycle-dependent behavior such as timers or analog digital converters (ADCs) need to be simulated efficiently. This requires a design methodology which enables designers to achieve cycle-accuracy without any significant loss in simulation speed. A Centralized Clock Control Mechanism is presented, which is employed for Infineon's peripheral models. Finally, for deployment of these next-generation VPs, fast development and reuse needs to be enabled. To provide a proper development cycle, code generation and verification methodology is needed for providing the maximum benefit to industry. An example of such a modeling flow is given based on Infineon's internal meta-modeling framework.

III. Ultra-Fast Source-Level Timing Simulation High Accuracy Needs Exact Code Matching

This section discusses a solution to overcome the limitations of timing simulation based on direct source-level timing annotations. The approach exploits an automated mapping between the source level and binary level control flow. Based on the resulting structural matching, low-level timing properties obtained through a static analysis of the target machine code are annotated to the source code of the program. The binary-level control flow is reconstructed, allowing a dynamic selection of timing annotations. This is achieved by adding markers to the source code of the instrumented program which interacts with special path simulation code generated from the binary-level control flow graph of the machine code. The path simulation

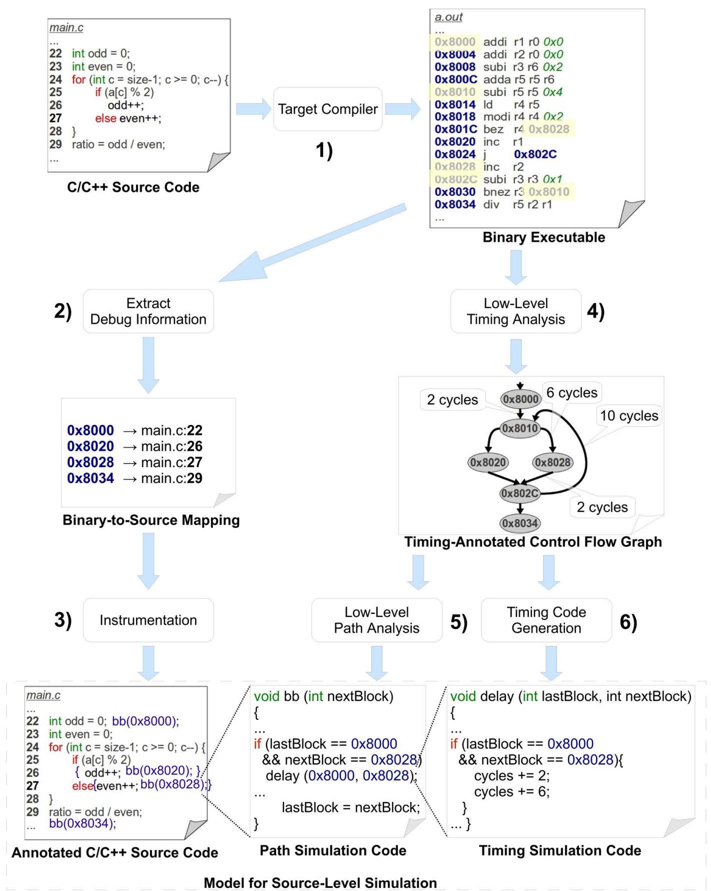

Fig. 1. Proposed Timing Instrumentation Work Flow

code represents the structure of the binary program and allows the dynamic correction of structural differences between the source code and the target machine instructions as well as the incorporation of context-sensitive timing annotations.

The dynamic simulation of binary-level control flow during host-compiled execution provides an efficient technique for timing simulation which accurately covers compiler optimizations. Dynamic path selection helps to include path-dependent timing estimates for sequences of basic blocks with respct to the cache and pipeline behavior. This is also very useful for simulating loops, e.g. to correct the number of simulated iterations in order to compensate for unrolling performed by the compiler. Furthermore, it allows handling other compiler optimization like function inlining more elegantly.

The complete work flow is depicted in Figure 1. After a program has been cross-compiled for the target architecture using a standard compiler (Figure 1, step 1), the compiler-generated debug information is used to relate the source code and the binary code (Figure 1, step 2). Instead of using this information to relate source-level and binary-level basic blocks for a direct annotation of low-level properties, the proposed method only uses this information for a tentative estimation of which source code portions correspond to the binary-level basic blocks. For every binary-level basic block, an equivalent source code position is determined from this data. During instrumentation (Figure 1, step 3), markers for the binary-level basic blocks are added to the source code at this position.

The execution time of the basic blocks in the binary code is obtained using a low-level timing analysis which models pipeline effects and static branch penalties of the target processor (Figure 1,

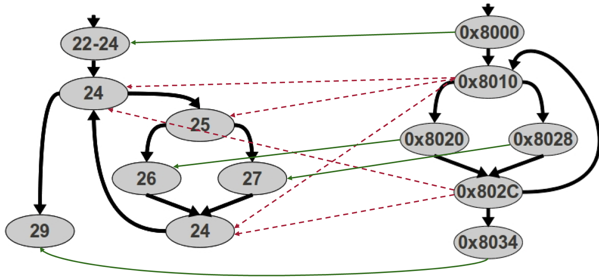

Fig. 2. Matching Between Source- and Binary-Level Control Flow

step 4). The result of this analysis is a timing-annotated control flow graph of the binary executable. The edges in this graph, which describe the transition between basic blocks during an actual execution of the program, are labeled with the execution time required by the respective sequence of machine instructions. Based on the binary-level CFG, the program control flow on the target architecture is analyzed to create *path simulation code* which models the target-specific behavior of the program (Figure 1, step 5). The generation of timing simulation code (Figure 1, step 6) is based on a path-dependent timing estimation to cover different execution times of a basic block including the transition to the following basic block with respect to the architectural state determined by the previously executed program path.

Compiling the instrumented source code and the path simulation code for the simulation host yields a model of the program which determines its execution time on the target processor. Using the markers that were added to the original source code during instrumentation, the path simulation code can approximate the path taken through the binary executable. This reconstruction of binary-level control flow is executed in parallel to the functionality of the original source code during simulation on the simulation host and allows the dynamic selection of timing annotations.

A. Matching Source-Level to Binary-Level Control Flow

The proposed method for relating source code and optimized binary code tries to solve the matching based on the so-called dominance relation between basic blocks. A basic block a in a control flow graph dominates another basic block b, if every path from the entry node of the graph to b goes through a. The dominance relation is applied to the source-level and binary-level CFG and helps to find the best mapping between source-level and binary-level basic blocks. The actual matching algorithm proceeds in the following steps [2]:

- Create the source-level and binary-level CFG and use it to calculate the dominator relation. Details about this step will be omitted.
- Read the compiler-generated line information and determine the set of potential references for every binary-level basic block. Additionally, the compiler-generated debug information can also be moved and interpolated between basic blocks to improve the quality of the line information.
- To select a source-level reference for every binary-level basic block, a matching is constructed using the set of potential references and the dominance relation.

On the example from Figure 1 the algorithm constructs the matching from Figure 2. As the presented method relies on compiler-generated debug information to relate source-level basic blocks and binary-level basic blocks, this relation is crucial for its accuracy. If optimizations are used, the debug information provided by standard compilers is often not accurate enough for this purpose. Therefore the mapping between source code and binary code provided by the debug information is subject to further analysis steps. Debug information often contains several references to source code lines for one basic block. Each of these entries describes a potential relation between a binary-level basic block and a source-level basic block. From these potential relations, an accurate mapping between binary-level and source-level basic blocks is determined by selecting at most one source-level equivalent for every binary-level basic block. This is done in such a way that the order of execution between basic blocks in the source-level and binary-level control flow graph is preserved. Hence if one binary-level basic block is always executed before a second one, the same relation holds for their respective source-level entries in the mapping. A more detailed description of this technique can be found in [3].

B. Source-Level Simulation of the Timed Binary Control Flow

The reconstruction of binary control flow is performed based on markers which are added to the source code during instrumentation. These markers describe the *potential* binary-level equivalents of a source-level basic block. The instrumentation code of a marker is a simple function call which gets the unique identifier of the respective binary-level basic block as parameter (cf. Figure 1). The arguments of these function calls are determined using compiler-generated debug information. The marker function `bb` contains the path simulation code and is responsible for performing the path reconstruction. In effect, the path simulation code determines which binary-level basic blocks would be executed during an actual execution of the program on the target processor. It also maintains information about the execution context of the simulation, meaning which basic blocks have already been executed (cf. Figure 1).

As the path reconstruction is based on the binary-level CFG, only feasible paths through the binary program are simulated. For instance in the example from Figure 1, the path reconstruction code would decide to simulate the basic block at `0x8010` for the first iteration of the loop, but not for further loop iterations. The transitions between markers correspond to paths between basic blocks in the binary-level CFG. According to the reconstructed path, the correct timing estimates for the respective execution of target binary code is determined. By simulating the transition between basic blocks, the path simulation can also consider structural differences between source code and binary code. So not every marker in the source code always results in the simulation of the respective binary-level basic blocks. Instead, the path simulation code can accumulate markers, for instance to model loop unrolling, or completely skip them if they do not match an actual path through the binary-level control flow graph. The latter case can occur if a marker was incorrectly added to the source code as a result of incorrect debug information generated by the compiler [4].

As can be seen in Figure 1, the `delay` function, which calculates the time consumed by a basic block transition, can consider arbitrary architectural details to derive precise timing estimates. Since the transition between basic blocks is modeled accurately, the effects of pipelining and static branch prediction in the target processor can be easily considered when determining the execution time of a basic block. This also holds for the effects of caches and dynamic branch prediction, but to consider these effects, additional dynamic cache and branch prediction models have to be integrated into the simulation. However, once the simulation executes blocks in an order not considered during analysis the situation is far more complex. For example, interrupts are not considered during analysis, as this would essentially require edges between every block and the start and return nodes of interrupt handlers thereby significantly increasing analysis complexity. Consequentially, once the execution of an interrupt handler is simulated, no appropriate edge is contained in the analysis results. Furthermore, the CFG is reconstructed starting from a specific function and only changes in control flow to known

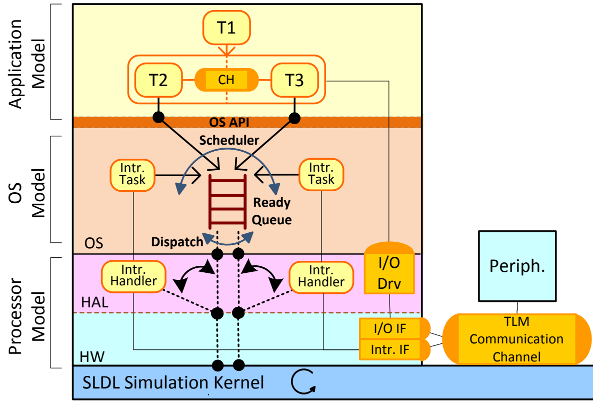

Fig. 3. Host-compiled simulation model.

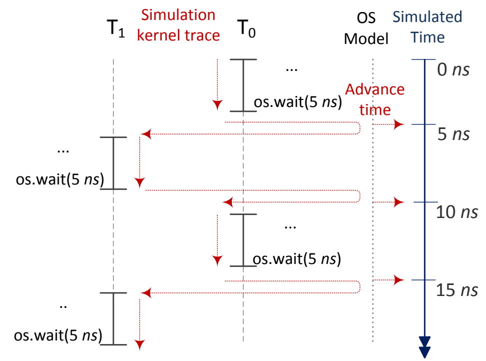

Fig. 4. Example of OS model trace.

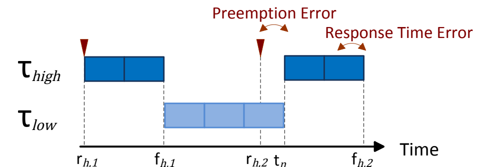

Fig. 5. Inherent preemption inaccuracies in discrete OS models.

targets of branch instructions reachable from this starting point are considered. Thus, the code for an interrupt handler will typically not be contained in the analysis results unless it is used as starting point, in which case other functions are likely not included. This limitation can be eliminated by picking a suitable one when no predetermined choice is available, thereby enabling a context-sensitive simulation including asynchronous events such as interrupts [5].

IV. 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. Fig. 3 shows a typical layered organization of a host-compiled simulation model [6], [7], [8]. Individual source-level application models that are annotated with timing and other metrics as described in Section III are converted into task running on top of an abstract, canonical OS 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 (Section V) and peripherals (Section VI). Finally, the complete processor model is integrated and co-simulated 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.

A. OS Modeling

An OS model generally emulates scheduling and interleaving of multiple tasks on one or more cores [9], [10], [11], [12], [13], [14], [15]. It maintains and manages tasks in a set of internal queues similar to real operating systems. 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.

Fig. 4 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 [6]. 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. 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. Fig. 5 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. 5, 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 [16]. This is, for example, the case if τ_{low} in Fig. 5 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 granu-

larity 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 tradeoff and provide a fast coarse-grain simulation while maintaining high accuracy. Existing approaches can be broadly categorized as either optimistic [17] or conservative [18]. 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 buses with priority-based arbitration [19] (see also Section V-B).

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. Note that unless a full rollback is possible in the simulator, optimistic approaches can not guarantee an accurate order of task events and interactions, such as shared variable accesses. As the name suggests, conservative approaches, by their nature, always maintain the correct task order. In both 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.

Overall, any coarse-grain, discrete-event modeling of asynchronous interactions among concurrent cores, processors or other system components will always come with a fundamental speed and accuracy tradeoff. A coarse simulation of a component decreases accuracy not only due to incoming events being captured too late, but also in terms of outgoing events being produced too early. Approaches mentioned above are aimed at improving or avoiding the speed-accuracy tradeoff in reactions to incoming events. Outgoing events can be reproduced accurately by simply advancing simulation time to its correct point before producing each external event. However, this reduces granularity and increases overhead. By contrast, maintaining a coarse granularity temporally decouples simulation processes and lets individual components run ahead of the global simulation time. This can in turn result in a wrong ordering of events being observed among components, e.g. in terms of accesses to shared resources such as caches. Various approaches for improving speed and accuracy tradeoffs under such scenarios will be discussed in later sections.

B. 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 [8], [20], [21], [22]. Specifically, accurate models of interrupt handling effects can contribute significantly to overall timing behavior and hence accuracy [7].

The software side of interrupt handling chains is typically modeled as special, high-priority interrupt handler tasks within the OS model [10]. 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

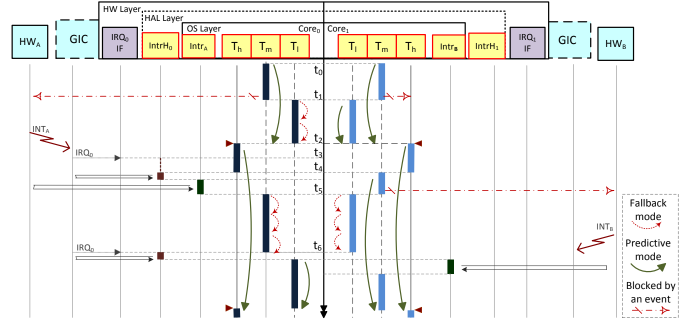

Fig. 6. Host-compiled simulation trace.

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 task each running on a dual-core platform is shown in Fig. 6 [6]. 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 [7]. 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 co-simulation of HW/SW interactions through the communication infrastructure. Methods for improving performance of such communication models will be discussed in Section V.

C. 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 [23], [24], [25], [26], [27].

As described in Section III, 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

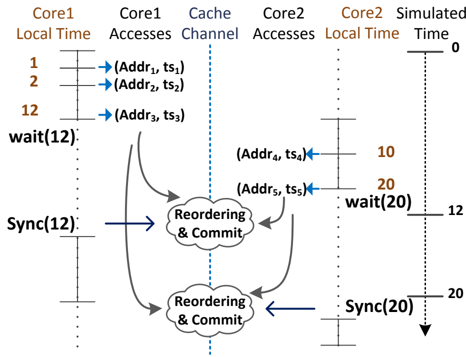

Fig. 7. Multi-core out-of-order cache simulation trace.

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.

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 accurate 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 (Section IV-B), 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 [28], [3]. 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. Fig. 7 illustrates this concept [28]. 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 [6]. Since cores and tasks can run ahead of time, a task may generate accesses that would otherwise not have been issued until after a possible preemption is completed. This requires access re-ordering 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 [3]. 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 Section V-C). Note that this still requires OS model support to generate accurate transaction time stamps in the presence of preemptions. Similar re-ordering techniques can then also be applied to other shared resources, such as buses, as will be shown in the following sections.

V. Abstract TLM Models for Accurate and Fast SoC Simulation

Embedded and integrated systems are compromised of many communicating components as we move towards 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. An initiator (master) module should be synchronized to the global simulation time before it starts executing its transactions. As many communication resources such as buses or target (slave) modules are shared between initiators, accurate models may additionally require to schedule an arbitration event at each arbitration cycle. Novel works have shown that this requirement 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 Temporal Decoupling (TD).

TD and block transaction imply that initiators perform accesses, which are located in the future with respect to the current global simulation time. This leads to several challenges, which may penalize simulation accuracy: Firstly, an initiator may generate transactions early with respect to global simulation time, which may change the order of accesses to shared variables. Secondly, shared resources are unaware of future conflicting transactions and cannot compute arbitration delays for TD transactions. Additionally, the initiator module is also unaware of resource conflicts, which may have blocked execution due to incoming data dependencies. Any computation afterwards may not rely on the local time offset, as this value does not reflect the additional delay time due to resource conflicts. This may cause a chain of dependencies, which limit simulation accuracy. Different methods addresses these challenges successfully and enable accurate and fast simulation. A selection of advanced communication models is briefly outlined in the following:

A. *Optimistic Simulation*

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 optimistic temporal decoupling. In optimistic TD, we ignore any possible data-dependencies or resource conflicts. Shared variables have to be protected by additional synchronization methods. An example can be seen from Fig. 8. The transaction of initiator I1 start at 140 ns, yet it is executed before the transaction of I2 due to the local time offset of 30ns. Additionally, the transactions to the target may overlap on the shared bus, which would require arbitration. Yet, I1 can finish its transaction without delay because the transaction of I2 was not yet simulated. Thus, timing is very optimistic.

B. *Conflict Handling at Transaction Boundaries*

In [19][29], 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 [19]. If as higher priority transaction is on-going, 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 retro-actively added. In the case of [19], another wait is issued to account for the additional delay but intelligent event re-notification could also be applied. The method has as minimum only a single wait call per transaction. An example is shown in Fig. 9 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

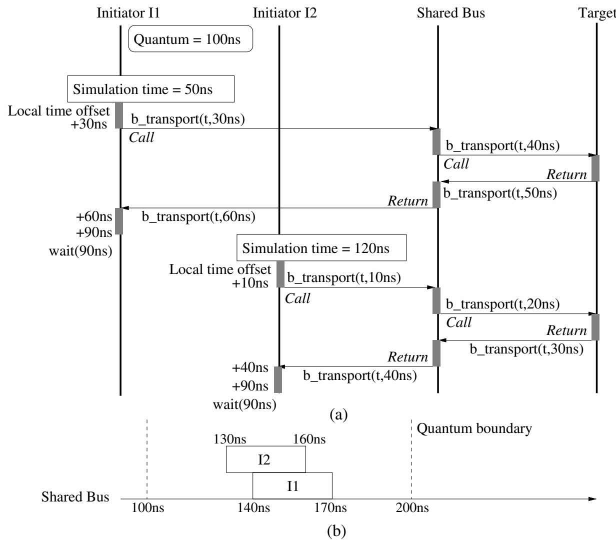

Fig. 8. Optimistic TD with Quantum Keeper

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 pre-empted, the transactions is split to assure that the order of data accesses is preserved.

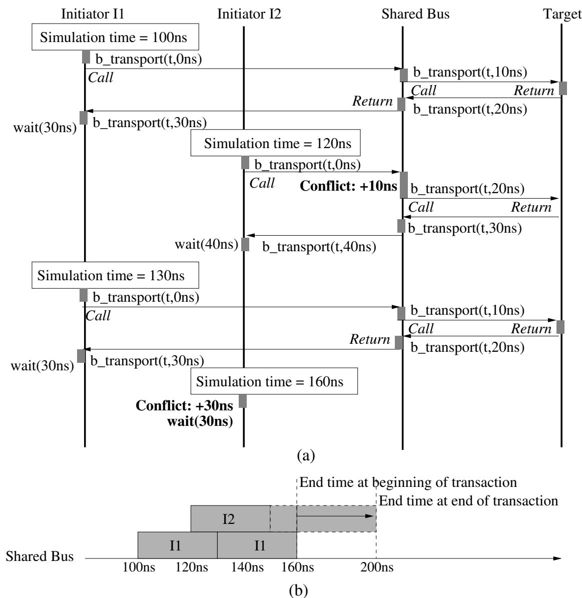

Fig. 9. Handling conflicts at transaction boundaries

C. *Conflict Handling at Quantum Boundaries*

With the Quantum Giver [3], 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 retro-actively orders all transactions according to their time stamps. It computes the delays due resource 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. 8. 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 effect 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 [30], Advanced Temporal Decoupling (ATD) is presented. It targets TD but also cycle-accuracy and preservation of 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 order of transactions. 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.

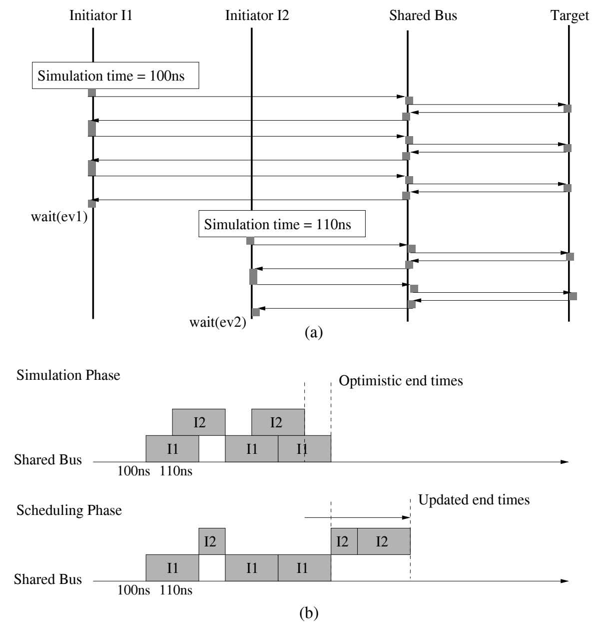

Fig. 10. Handling conflicts with Quantum Giver

D. *TLM+: Conflict Handling at Software Boundaries*

TLM+ is a SW-centric communication model targeting host-compiled SW simulation [31]. 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 accordingly. 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 profile of the original driver function [32]. This profile allows to extract a demand for communication resources. Usually, a driver function would not

block a shared bus completely such two TLM+ block transactions can interleave. Analytical demand-availability estimators inside the resource model can be used to estimate the delay due to resource conflicts [33].

The scheduling is conducted by the resource model at the transaction boundaries. These boundaries then correspond to the entry and exit to the respective driver function. The concept is illustrated in Fig. 11. Initiator I1 first executes a block transaction. At a later point, I2 starts another block transaction. Yet, because I1 has higher priority, I2 is scheduled to take longer as it has not the full availability of the shared bus. When the block transaction of I1 finishes, I2 has no further conflicts on the bus. Its end time gets re-scheduled to an earlier time. This is done by event re-notification, which leads to a single call to wait for each block transaction. TLM+ targets higher abstraction and faster performance compared to the other methods. It also does not execute the original SW as the driver functions are abstracted away.

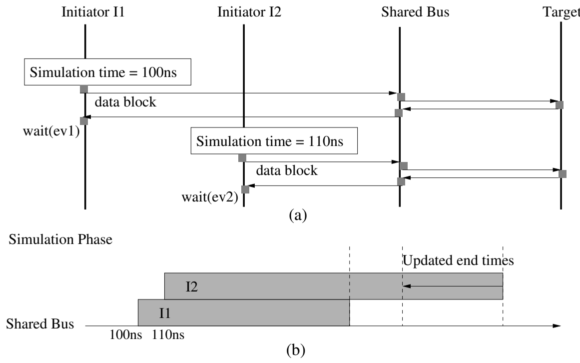

Fig. 11. Handling conflicts in TLM+

VI. Peripherals: Centralized Clock Control Mechanism for Cycle-Accuracy with Speed

With the increasing interest in the architecture exploration and performance analysis, there is an increase in demand of cycle-accurate SystemC models along with simulation performance comparable to that of loosely-timed models [1]. This requires not only CPU, Bus and Memory Models to be designed efficiently but also the way peripherals like ADC (Analog Digital Converter), interacts with them. It requires change in the modelling techniques to achieve cycle-accuracy with the desired performance.

In this section, we talk about a design methodology which enables designers to achieve cycle-accuracy without any significant loss in simulation performance.

A. Centralized Clock Control Mechanism

Accuracy and speed of the model can be achieved by functional abstraction, efficient modelling of time related behaviour using centralized clock control mechanisms, reduced number of processes and process activations in the design.

With the conventional method of using toggling clock to model timing behaviour, cycle-accuracy can be achieved but it is not feasible to achieve the desired simulation performance.

An alternative is to use clock period to model timing behaviour. In this approach, the clock period is used to predict time at which the required clock edge would occur rather than waiting for the clock edge. All the processes in the design predict when are to be triggered again based on the clock time period and schedule the triggering accordingly in the form of event notification and wait statements. The advantage of this methodology is that it is easier to achieve better simulation performance for the software models. However, the models loose accuracy with changes in the clock frequency as It is difficult to synchronize already scheduled processes in the design when period of the clock changes. For complex designs, it becomes difficult to debug and to adapt for cycle-accuracy. The simulation performance also degrades as the number of processes increase in a design due to increased context switching and the advantage of this approach is lost [34].

Our new design methodology refines this approach by providing a generic clock control unit (CCU) which acts as a single source of clock-information in a design. It recommends a design to be modularized where the sub-modules implement clock-dependent operations in call-backs registered with CCU instead of processes waiting on events or time outs. It reduces the overall number of processes in the design and the associated scheduling logic which results in better simulation performance. It further reduces the model complexity and makes it easier to adapt models for cycle-accuracy. The CCU maintains the operations synchronized and handles changes in the clock-period. It assures cycle-accuracy in all cases.

As shown in Fig. 12, the CCU becomes part of the design as a sub-module. It is connected to the modules input clock and to all other sub-modules, also called clock-clients (CCt), which require the clock to operate via registered call-backs. The CCU supports call-backs on both, rising and falling edges of clock as triggering edge. It works on a request-call mechanism where a CCt can request CCU to provide a call-back after a given number of clock cycles or clock ticks. The CCts can be categorized as active and passive. An active CCt is the one that requests the CCU for a call-back i.e. it works directly on clock edges. A passive CCt does not request the CCU for a call-back but implement call-back to perform operations on change in input from another CCt or external input.

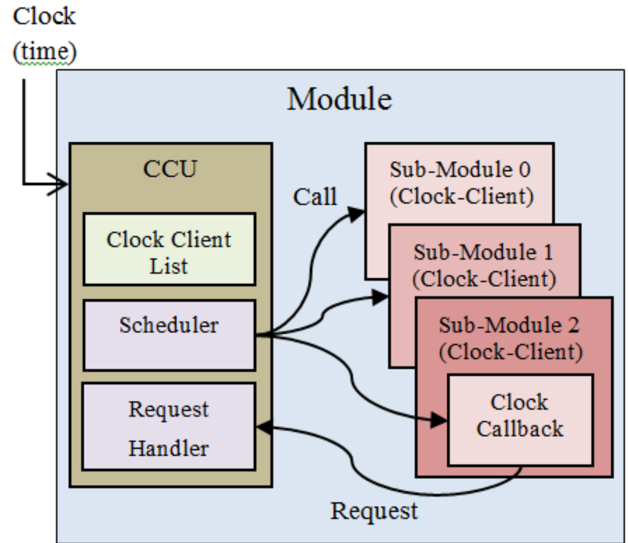

Fig. 12. Modeling with CCU

On receiving call-back requests, the CCU selects the minimum of the number of cycles requested by various clock-clients and schedules call-backs using clock period. After the elapse of requested clock-cycles, the call-backs are invoked on all the clock clients, both active and passive. For active CCts, the call-backs are executed only if the elapsed number of cycles matches the requested cycles; else the required number of cycles are re-adjusted and requested back to the CCU. The call-backs are invoked sequentially in the order they were registered which omits non-deterministic behaviour. In a scenario where two CCts are registered with different CCUs and need to exchange data, race condition may happen if call-backs are invoked

simultaneously on both the CCts. In such case, the predefined channel sc signal can be used to avoid race conditions.

This approach was used to re-develop an ADC and a Timer module. The results are discussed below:

ADC: In comparison to old model with traditional modeling approach, the number of SystemC process has been reduced from thirteen to five (including CCU). The reduced event handling and context switching resulted in 77% improvement in simulation speed. It further increased the cycle accuracy of the design.

Timer: It is a relatively much simpler design than the ADC. The model with traditional approach had fewer processes to implement the functionality but couldnt handle the in-accuracies due to clock frequency changes. With the new approach, it resulted in an acceptable 10% decrease in the simulation speed in comparison with old approach but is a more accurate model.

VII. Infineon's Method for Better Development Productivity and Verification

We need a proper development cycle, code generation and verification methodology to ensure quick development of the models and best use giving the maximum benefit to the company.

A. *Common Infineon Library*

At Infineon, a central modelling library based on SystemC is used to develop various models. The library provides classes for modelling modules, registers, attributes, multiple clock and reset domains as well as interface implementation. It supports register callbacks to enable user implementation on register read-write operations. It supports TLM-2.0 standard interfaces and provides ready to use master and slave interface implementation, thus reduces the overall development time and brings consistency in modelling.

B. *Code Generation*

Along with other techniques to improve the productivity and accuracy in building software models, code generation is one of the most important mechanisms to ensure quality and consistency. The single source concept where everything is derived from the same specification and concept becomes important with more complex designs which involve more number of people and with increasing number of different abstract views.

The biggest challenge at system level is that all companies/groups have their own modelling and coding styles. So a common code generation mechanism developed by an external vendor doesnt fit for all. Hence an adoptable in-house code generation methodology becomes very important and need of the time.

In Infineon, we use meta-modelling and code generation methodology which takes data at higher level of abstraction and generates the desired output. Automatic generation ensures that we get consistent code quality and faster development of models at all levels.

In meta-modelling flow, the target of generation is a so called view, which may be e.g. SystemC code or a schematic description. The view is generated by a so called generator, which is often a template engine [35]. This template engine renders a so called template, a mix of target code, substitutions, and generation pragmas.

In order to generate register view in SystemC style, often from a specification - the template engine has to retrieve the required data appropriately. This data is stored in a structured way in a so called model.

The structure of the model is defined in a so called meta-model as shown in Fig. 13. Here meta means above and meta-model means a model above or more abstract than a model [36]. A meta-model is also called a model of a model or in other words, a model is an instance of a meta-model.

The data of the model is read from a specification, parsed from any other document, imported from a description formulated in a so called domain specific language, or entered through a generated GUI.

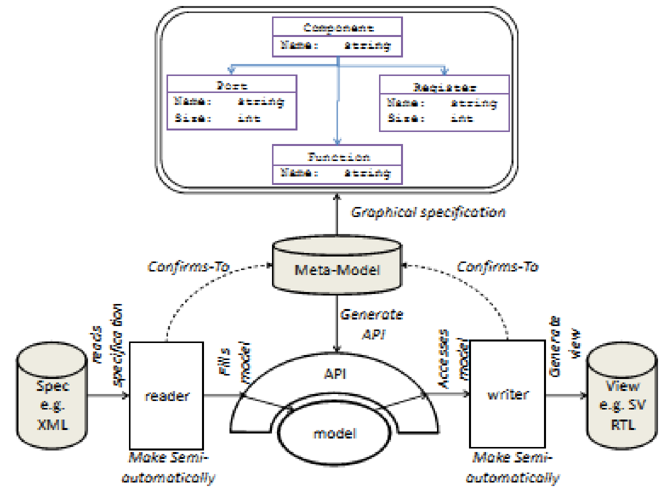

Fig. 13. Flow diagram of a meta-modeling environment

C. *Verification of SystemC Models*

For a SystemC model, functional accuracy and in case of architectural exploration and performance analysis, cycle accuracy is very important for developing software that will run successfully on the real chip. The SystemC modelling is done based on the IP specification, which could be interpreted differently by a designer and could result in a totally different implementation on RTL. Such inaccuracies may cause the developed software to be re-worked when it is run on the real chip. At IFX, we have a co-simulation methodology to verify the SystemC model which is re-using existing verification environment (VE) of the corresponding RTL. In this approach, the RTL model is replaced with its SystemC model in the VE. It allows the SystemC model to be tested with the same stimulus as used for RTL and compare its output against the same reference model in the VE.

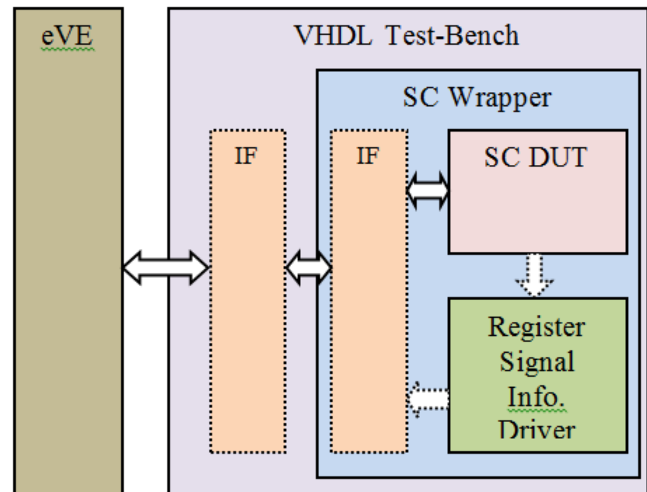

Fig. 14. SystemC model as DUT in Specman Verification Environment (eVE)

In Fig. 14, a SystemC model as DUT in a specman e verification environment (eVE) is depicted. The required wrapper to connect SystemC model as DUT is generated from the source files and the VHDL test-bench. The VE refers to the internal signals of the RTL DUT to perform certain checks which are not valid for the SystemC DUT. Hence it requires an additional block Register

Signal Information Driver to drive these signals to the VE. This approach reduces the verification effort and provides a platform to compare RTL against the SystemC model. With this co-simulation, the functional as well as temporal accuracy between the RTL and SystemC models can be established.

VIII. Conclusion

With time to market shrinking day by day, developing fast and accurate models are 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 platforms based on host-compiled software simulation can provide such a ultra-fast yet highly-accurate modeling solutions.

Acknowledgments

The authors would like to thank Andy D. Pimentel (University of Amsterdam) for his support of the special session as session chair.

References

[1] "1666-2011 - IEEE standard for standard SystemC language reference manual," 2012.

[2] S. Stattelmann, O. Bringmann, and W. Rosenstiel, "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*, 2011.

[3] ——, "Fast and accurate resource conflict simulation for performance analysis of multi-core systems," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2011.

[4] ——, "Fast and accurate source-level simulation of software timing considering complex code optimizations," in *ACM/IEEE Design Automation Conference (DAC)*, 2011.

[5] S. Ottlik, S. Stattelmann, A. Viehl, W. Rosenstiel, and O. Bringmann, "Context-sensitive timing simulation of binary embedded software," in *Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, ser. CASES '14, 2014.

[6] P. Razaghi, "Dynamic time management for improved accuracy and speed in host-compiled multi-core platform models," Ph.D. dissertation, The University of Texas at Austin, 2014.

[7] P. Razaghi and A. Gerstlauer, "Host-compiled multi-core system simulation for early real-time performance evaluation," *ACM Transactions on Embedded Computing Systems (TECS)*, accepted for publication 2014.

[8] G. Schirner, A. Gerstlauer, and R. Dömer, "Fast and accurate processor models for efficient MPSoC design," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 15, no. 2, pp. 10:1–10:26, March 2010.

[9] P. Razaghi and A. Gerstlauer, "Host-compiled multicore RTOS simulator for embedded real-time software development," in *Proceedings of the Design, Automation Test in Europe (DATE) Conference*, March 2011.

[10] H. Zabel, W. Müller, and A. Gerstlauer, "Accurate RTOS modeling and analysis with SystemC," in *Hardware-dependent Software: Principles and Practice*, W. Ecker, W. Müller, and R. Dömer, Eds. Springer, 2009.

[11] B. Miramond, E. Huck, F. Verdier, M. E. A. Benkhelifa, B. Granado, M. Aichouch, J.-C. Prvotet, D. Chillet, S. Pillement, T. Lefebvre, and Y. Oliva, "OveRSoC : a framework for the exploration of RTOS for RSoC platforms," *International Journal on Reconfigurable Computing*, vol. 2009, no. 450607, pp. 1–18, December 2009.

[12] H. Posadas, J. damez, E. Villar, F. Blasco, and F. Escuder, "RTOS modeling in SystemC for real-time embedded SW simulation: A POSIX model," *Design Automation for Embedded Systems*, vol. 10, no. 4, pp. 209–227, December 2005.

[13] Z. He, A. Mok, and C. Peng, "Timed RTOS modeling for embedded system design," in *Proceedings of the Real Time and Embedded Technology and Applications Symposium (RTAS)*, March 2005.

[14] R. Le Moigne, O. Pasquier, and J.-P. Calvez, "A generic RTOS model for real-time systems simulation with SystemC," in *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, February 2004.

[15] A. Gerstlauer, H. Yu, and D. Gajski, "RTOS modeling for system level design," in *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, March 2003.

[16] P. Razaghi and A. Gerstlauer, "Predictive OS modeling for host-compiled simulation of periodic real-time task sets," *IEEE Embedded Systems Letters (ESL)*, vol. 4, no. 1, pp. 5–8, March 2012.

[17] G. Schirner and R. Dömer, "Introducing preemptive scheduling in abstract RTOS models using result oriented modeling," in *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, March 2008.

[18] P. Razaghi and A. Gerstlauer, "Automatic timing granularity adjustment for host-compiled software simulation," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, January 2012.

[19] G. Schirner and R. Dömer, "Result oriented modeling a novel technique for fast and accurate TLM," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 26, no. 9, pp. 1688–1699, September 2007.

[20] P. Gerin, H. Shen, A. Chureau, A. Bouchhima, and A. Jerraya, "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)*, January 2007.

[21] T. Kempf, M. Dorper, R. Leupers, G. Ascheid, H. Meyr, T. Kogel, and B. Vanthournout, "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*, March 2005.

[22] A. Bouchhima, I. Bacivarov, W. Youssef, M. Bonaciu, and A. Jerraya, "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)*, January 2005.

[23] S. Stattelmann, G. Gebhard, C. Cullmann, O. Bringmann, and W. Rosenstiel, "Hybrid source-level simulation of data caches using abstract cache models," in *proceedings of the Design, Automation Test in Europe (DATE) Conference*, March 2012.

[24] H. Posadas, L. Diaz, and E. Villar, "Fast data-cache modeling for native co-simulation," in *Proceeding of the Asia and South Pacific Design Automation Conference (ASPDAC)*, January 2011.

[25] A. Pedram, D. Craven, and A. Gerstlauer, "Modeling Cache Effects at the Transaction Level," in *Proceedings of the International Embedded Systems Symposium (IESS)*, September 2009.

[26] J. Schnerr, O. Bringmann, A. Viehl, and W. Rosenstiel, "High-performance timing simulation of embedded software." in *ACM/IEEE Design Automation Conference (DAC)*, 2008, pp. 290–295.

[27] J. Schnerr, O. Bringmann, M. Krause, A. Viehl, and W. Rosenstiel, "Systemc-based performance analysis of embedded systems," in *Model-Based Design for Embedded Systems*, P. J. M. Gabriela Nicolescu, Ed. CRC Press, 2009. [Online]. Available: ://dx.doi.org/10.1007/978-1-4614-1427-8_11

[28] P. Razaghi and A. Gerstlauer, "Multi-core cache hierarchy modeling for host-compiled performance simulation," in *Proceedings of the Electronic System Level Synthesis Conference (ESLsyn)*, May 2013.

[29] M. Radetzki and R. Khaligh, "Accuracy-adaptive simulation of transaction level models," in *Design, Automation and Test in Europe, 2008. DATE '08*, March 2008, pp. 788–791.

[30] S. Hufnagel, "Towards the efficient creation of accurate and high-performance virtual prototypes," Ph.D. dissertation, Technical University of Kaiserslautern, 2014. [Online]. Available: https://kluedo.ub.uni-kl.de/frontdoor/index/index/docId/3892

[31] W. Ecker, V. Esen, R. Schwencker, T. Steininger, and M. Velten, "Tlm+ modeling of embedded hw/sw systems," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2010, pp. 75–80.

[32] K. Lu, D. Muller-Gritschneder, and U. Schlichtmann, "Accurately timed transaction level models for virtual prototyping at high abstraction level," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, March 2012, pp. 135–140.

[33] ——, "Analytical timing estimation for temporally decoupled tlms considering resource conflicts," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, March 2013, pp. 1161–1166.

[34] J. D. D. C. Black, *SystemC: From the Ground Up*, 2nd ed. Springer, 2009.

[35] "Mako templates for python," http://www.makotemplates.org/.

[36] M. M. Brambilla, J. Cabot, *Model-Driven Software Engineering in Practice*. Morgan&Claypool Publishers, 2012.