Fast and Accurate Resource Conflict Simulation for Performance Analysis of Multi-Core Systems

Stefan Stattelmann†, Oliver Bringmann†
†FZI Forschungszentrum Informatik
Haid-und-Neu-Str. 10–14
D-76131 Karlsruhe, Germany
{stattelmann, bringmann}@fzi.de

Wolfgang Rosenstiel†‡
‡University of Tuebingen
Sand 13
D-72076 Tuebingen, Germany
rosenstiel@informatik.uni-tuebingen.de

***Abstract*—This work presents a SystemC-based simulation approach for fast performance analysis of parallel software components, using source code annotated with low-level timing properties. In contrast to other source-level approaches for performance analysis, timing attributes obtained from binary code can be annotated even if compiler optimizations are used without requiring changes in the compiler. To consider concurrent accesses to shared resources like caches accurately during a source-level simulation, an extension of the SystemC TLM-2.0 standard for reducing the necessary synchronization overhead is proposed as well. This enables the simulation of low-level timing effects without performing a full-fledged instruction set simulation and at speeds close to pure native execution.**

***Index Terms*—System analysis and design; Timing; Modeling; Software performance;**

I. Introduction

An increasing portion of functionality in current embedded systems is implemented in software. For systems which have to fulfill timing constraints, estimating software performance at an early design stage is often crucial. If the violation of timing requirements is detected too late, it can entail a costly redesign of the entire system. Yet estimating the performance of software-intensive systems is a complex task, as it is influenced by the actual hardware on which the software is executed and the final binary code to which it gets compiled. Especially in multi-core systems, the interaction of software components due to concurrent accesses to shared resources does have a significant effect. Hence, to estimate performance precisely, considering a single software component in isolation is insufficient. Instead, the effect of all other system components must be considered as well.

There are static and dynamic methods for analyzing software execution times. Static approaches do not require an actual execution of the analyzed system or program; instead, performance properties are derived from an analytical model. Most static timing analyses determine execution times through a sequence of isolated, subsequent steps [1]: Firstly, the execution time of smaller program parts is analyzed on machine code level using an abstract processor model. This information is then used to determine the maximal execution time of a single task [2]. Based on the characteristics of the complete system design (e.g the scheduling policy of the operating system), the performance of the entire system can be evaluated [3]. As all analysis steps are executed independently, the interference between different components must be approximated conservatively. Hence static methods can often only determine properties of the worst-case performance of a given software component. To provide accurate results, the use of shared resources like caches must be restricted [4]. This limitation prohibits the application of static timing analysis for common multi-core architectures.

Dynamic methods for timing analysis rely on observing a system while it is running. Measurements using real hardware or register transfer level (RTL) models are hardly feasible in early system design phases. Measuring timing properties this way is very time consuming as all details of the design must be observed or simulated. Instead, an instruction set simulator (ISS) can be used to estimate the performance of binary code. Using a modeling language like SystemC [5], multiple instances of an ISS can be employed in parallel to analyze the behavior of a multi-core system [6]. More recently, the annotated source code of software components [7] is used for this purpose using native execution instead of an ISS interpreting binary code. While this approach allows a faster simulation of software execution, it also creates additional challenges, like the consideration of compiler optimizations and the synchronization of accesses to shared resources.

This paper presents a fast and accurate approach for system level performance analysis of software components in multi-core systems. To precisely estimate the execution times without using an ISS, software components are annotated with timing properties from a low-level execution time analysis to precisely estimate their performance. To model accesses to shared caches precisely without sacrificing simulation speed, an extension of the SystemC TLM-2.0 standard is described which reduces the synchronization overhead necessary to model shared resources. Combining these two approaches allows simulating low-level effects of software execution without an ISS and at speeds close to pure native execution.

II. SystemC TLM-2.0

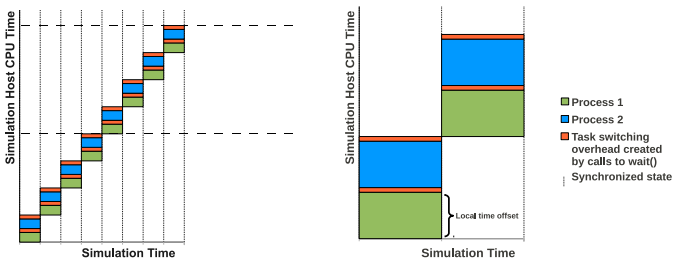
For the purpose of architecture exploration and pre-silicon software development, *virtual prototypes* based on SystemC using transaction level modeling (TLM) are in widespread use. SystemC uses discrete event simulation to model arbitrary hardware or software components using SystemC processes written in C++. The simulation of the processes is synchronized by the SystemC scheduler which keeps track of global simulation time and the time-ordered sequence of events from all processes. SystemC uses a cooperative scheduling mechanism which requires all processes to yield control in order to allow the scheduling of other simulation processes. To increase the simulation speed of SystemC-based virtual prototypes and to ease the exchange of these models, the SystemC TLM-2.0 standard [8] has been introduced. TLM-2.0 defines two *coding styles* to model timing behavior at different abstraction levels: the *approximately-timed* coding style (TLM-AT) based on lock-step simulation and the *loosely-timed* coding style (TLM-LT) using temporally decoupled simulation to increase simulation performance.

978-3-9810801-7-9/DATE11/©2011 EDAA

TLM-AT assumes that all simulated processes can be annotated with specific delay values. In order to run in lock-step with the SystemC simulation kernel, these delay values are used in calls to the `wait` function at certain fixed synchronization points. As a result of these calls, another process gets scheduled by the simulation kernel which introduces a lot of task switching overhead. Fig. 1 shows the negative effect of synchronization on simulation performance. If `wait` is called too often by the simulation processes, a large amount of CPU time on the simulation host is spent for task switching. To reduce the task switching overhead of repeated synchronization, the TLM-LT coding style does not require that all simulated processes always synchronize with the simulation kernel explicitly. Instead, TLM-LT allows processes to run ahead of the global simulation time. To allow this so called *temporal decoupling*, each process keeps track of its local time offset with respect to the global simulation time. Due to the cooperative scheduling mechanism employed in the SystemC simulation kernel, a maximal value for the local offset, the *global time quantum*, is defined for all simulated processes. As soon as the local time offset of a temporally decoupled process reaches the global time quantum, it must synchronize its local time with the global simulation time by calling `wait`.

For simplifying a loosely timed simulation with temporal decoupling, the TLM-2.0 standard implementation provides the so called *Quantum Keeper* class. It can be used to automatically synchronize a process as soon as the local time offset passes its maximal value by replacing calls to `wait` with calls to the respective method of the Quantum Keeper which simply sums up the local offset. The Quantum Keeper only calls `wait` if the time offset reaches the threshold or synchronization is explicitly demanded. Nonetheless, temporal decoupling and the quantum keeper are solely methods to allow a fast simulation despite the cooperative scheduling mechanism of SystemC. They cannot replace a synchronization mechanism to resolve data dependencies between processes or accesses to shared resources.

The lack of a synchronization mechanism leaves it to processes what should be done if a dependency between the execution of temporally decoupled processes is detected. In effect this means that it either cannot be guaranteed that data exchanged between processes is consistent or all involved processes have to synchronize their local time. If, for instance, shared memory is used, processes should synchronize before every memory access. Otherwise, they might read old data or newly written data can be overwritten by an earlier write from another process which is scheduled afterwards. In the worst case, excessive synchronization will degrade the TLM-LT simulation performance to the level of TLM-AT. On the other hand, exchanging inconsistent data between processes might not be an option if the functional or non-functional properties of the simulated software components depend on the order in which transactions are executed. If for instance an instruction or data cache is simulated, the order of accesses can determine whether an access is cache hit or cache miss. While data dependencies in the executed code usually require a synchronization for the functional part of the simulation, this does not hold for non-functional properties like an increased execution time because of a cache miss or due to task preemption. If the simulation of software components is carried out at the source level using instrumented versions of the original source code [7], the simulation of functional and non-functional properties can be clearly separated. In this case, synchronizing at each cache access for



(a) Standard SystemC Simulation (b) Temporal Decoupling

Fig. 1: Synchronization Frequency

example incurs an unacceptable performance penalty, but the accesses must be synchronized in order to detect conflicting accesses of concurrent software tasks. Similar problems occur if the execution of software components is controlled using a simulated OS scheduler or if tasks can be suspended by interrupts at arbitrary points in time: the non-functional property to be evaluated (i.e. execution time and deadline violations) does not necessarily depend on the functional simulation, yet a precise synchronization of competing tasks is necessary to get precise results.

III. Access Synchronization of Shared Resources

This section describes a synchronization mechanism for TLM-LT which reduces the synchronization overhead necessary to model shared resources. In order to speed up simulation, transactions are completed without enforcing synchronization, but a central instance keeps track of all transactions which are potentially influenced by other processes which have not yet synchronized their local time. After all processes have reached the maximal local time offset or an explicit synchronization point, conflicting transactions are arbitrated and execution times can be revised with a posteriori knowledge. These retroactive adjustments are carried out in a way which is completely transparent to the involved processes. In effect, this approach achieves a precise synchronization of transactions despite the use of temporal decoupling and without any task switching overhead in the SystemC kernel.

A. The Quantum Giver for TLM-2.0

To allow a precise synchronization of conflicting transactions in a temporally decoupled simulation, the *Quantum Giver* synchronization approach is presented. By adding an additional layer of abstraction, it is possible to hide the synchronization mechanisms from the involved SystemC processes and hence reduce the number of context switches during simulation. The Quantum Keeper defined in the SystemC TLM-2.0 standard implementation hides the abstraction of temporal decoupling by simply providing interfaces to "consume" simulation time to the processes using it. Calling `wait` and hence synchronization with the other SystemC processes occurs implicitly if the local time offset reaches the global time quantum.

The proposed Quantum Giver generalizes the concept of the Quantum Keeper by providing means to consider the effect of conflicting transactions during the synchronization process. In effect, the synchronization mechanism proceeds in rounds which contain several phases. In the first phase, the *simulation phase*, processes are simulated using temporal decoupling. All transactions issued by an initiator are completed immediately, meaning without synchronizing with other processes. After all initiators have reached the maximal local time offset or a point in time where synchronization was explicitly demanded by one of the

processes, the *synchronization phase* is executed. In this phase, all target components order the transactions they have received in the simulation phase and detect any changes in the previously predicted time for transactions due to conflicts. These changes are then broadcasted to all other target components, possibly triggering further changes. This procedure is performed until all components have reached a stable state. If the local time offset of an initiator component has changed because of interfering transactions detected in the synchronization phase, this is not signaled to the component itself, but to the Quantum Giver. The Quantum Giver is responsible for adjusting the local time offset of the component for the next round in the *scheduling phase*. This means that an initiator can get a smaller time slice in the next round if one of its transactions was delayed, or a bigger slice if a transaction was accelerated. Compared to TLM-LT, the proposed method requires no additional context switches to perform these adjustments, i.e. there is only one context switch per process in each round.

B. Synchronization Protocol

One round of the synchronization mechanism is depicted in Fig. 2, which will be explained in the following. One prerequisite not explicitly stated is that all initiators and all target components register with the Quantum Giver during elaboration of the SystemC virtual prototype.

1) After SystemC elaboration, the synchronization mechanism starts with its first simulation phase. In this phase, all processes are scheduled using the standard SystemC scheduler. This means that the order in which processes are executed is arbitrary. To control the order of transactions without explicit synchronization using the SystemC kernel, the target components execute an incoming transaction under the optimistic assumption that no other transaction will interfere with its outcome. To allow an eventual correction, transactions are also stored in an internal list. After the local time offset of an initiator has reached the global quantum, it notifies the Quantum Giver using a simple method call (i.e. without overhead in the simulation kernel) and yields control to another process by waiting for a SystemC event which will be generated by the Quantum Giver. If the Quantum Giver has received this notification from all initiator components, the simulation phase has been completed and the synchronization phase can be started.
2) To resolve conflicting accesses and inter-component effects, the Quantum Giver orders all targets to sort their internal transaction list according to the time stamps of the transactions in the synchronization phase. If overlapping transactions are detected, the target component is responsible for arbitrating this conflict. As a consequence of this arbitration, the time it takes to complete a transaction can be different from the time estimated during the simulation phase. To revise incorrect predictions, a target component which detects a change in the duration of a transaction must notify all other targets about this change. This information is then used to update the time stamps of all transactions from the same initiator. In order to avoid degrading simulation performance, the respective changes are accumulated over the complete simulation phase and only their effect on the next round of the protocol are communicated among components. As the synchronization phase is entered as soon as the last initiator component has depleted its time slice using standard method calls and without task switches in the simulation kernel, it consumes no simulation time. If no conflicts are detected, the target components will only sort the requests they have received and execute them in the correct order, so the induced overhead depends mainly on the data structure used for storing and sorting transactions.
3) When all transactions have been processed successfully, the Quantum Giver enters the scheduling phase. Depending on the actual time required for the transactions of an initiator during the simulation phase, the Quantum Giver creates SystemC events to wake up the respective process. This event will only be created if the process has not already depleted its time slice for the next round. So if an initiator consumes more execution time from the next round than the global quantum, its wake up event will not be scheduled. Hence no additional task switches are created.

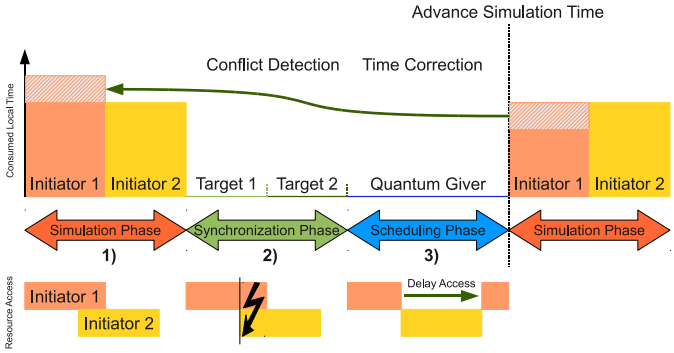


Fig. 2: Phases of Synchronization Protocol

The timing correction performed by the Quantum Giver is illustrated in Fig. 2: during the simulation phase, the simulation process of *Initiator 1* is scheduled before the process of *Initiator 2*; the resource access of *Initiator 1* is simulated before those of *Initiator 2* and both initiator processes use up the same amount of simulation time. As *Initiator 1* is assumed to have a lower priority than *Initiator 2*, the access of *Initiator 1* would have been delayed by the target components in a lock-step simulation. In a temporally decoupled simulation, this is not always possible as concurrent accesses with higher priority could be simulated after the current process has consumed its complete time slice. Using the Quantum Giver approach, conflicts from concurrent accesses are detected during the synchronization phase and the transactions of the respective process are delayed. In the given example, this leads to an increase of the local time for *Initiator 1*, since the transactions require more time than originally predicted in the simulation phase. As simulation time from the next round has been used due to this wrong prediction, *Initiator 1* must use less simulation time than *Initiator 2* in the next round of the synchronization protocol. The Quantum Giver performs the required correction by adjusting the local simulation time for the simulation process of *Initiator 1* and by delaying the activation event of the respective process. These adjustments are performed transparently for the initiator process and there is no additional overhead as the processes simply wait for the event scheduled by the Quantum Giver. For this reason, existing simulation code can be very easily modified to use the Quantum Giver approach. All it requires is replacing calls to `wait` with calls to the respective method of the Quantum Giver.

IV. Source-Level Simulation of Machine Code

In order to use the presented synchronization approach, software components must be augmented with information about their timing behavior. To estimate the timing behavior of a software component using native execution on a simulation host, a precise mapping between its source code and the compiled binary code for the target architecture is necessary. Using this mapping, properties of the target machine code can be annotated to the source code before it is used in the simulation model. If an optimizing compiler is used, relating binary code and source-level statements can be very complicated, as the compiler might change the program structure significantly. The compiler-generated debug information, which is used to relate binary code and source code for source-level debugging, cannot be trusted if optimizations are enabled, as the generated information might be incomplete, ambiguous or incorrect. To overcome this problem, this section describes a method for analyzing debug information to match binary code and source code at points that are important for program control flow. Combining this mapping with analyses of binary control flow and low-level execution times allows source-level performance estimation despite compiler optimizations. The complete analysis flow is depicted in Fig. 3.

A. Relating Source Code and Binary Control Flow

Save an exhaustive analysis of program semantics, compiler-generated debug information is the only way to relate source code and machine instructions of a program without modifying the compiler used for creating the machine code. If compiler optimizations are used, the line information stored in the debug information might not be accurate. This means that the order of executed machine instructions is different from the order of the source code statements from which the instructions were created according to line information. To allow a precise source-level simulation of binary code execution, these inconsistencies in the compiler-generated debug information must be eliminated.

A precise relation between source code and binary code can be established based on control flow information from both levels. A method for achieving this is sketched in the analysis steps 1–5 of Fig. 3. 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 (Fig. 3, step 3). 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 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 determined by step 5 in Fig. 3. Based on the reconstructed relation between basic blocks in the binary code and source code lines, instrumentation code can be added to the source for a source-level simulation of binary attributes (Fig. 3, step 8).

B. Simulating Binary Control Flow

Relating the binary-level control flow graph and the respective source-level control flow graph without ambiguity is not always possible statically, as there is not always a unique source code position for every basic block in the binary code. This can happen for example due to function inlining, as inlining creates

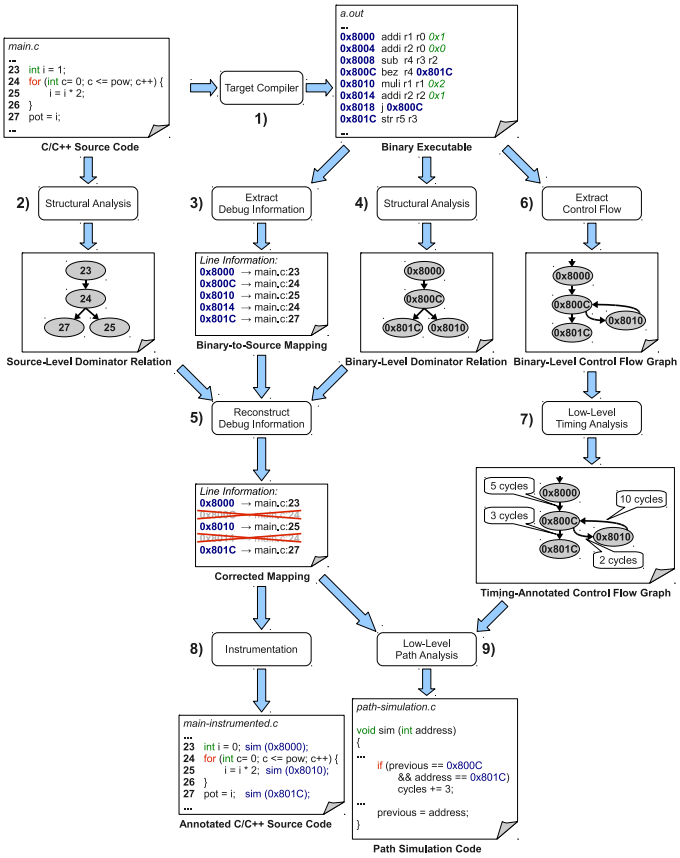


Fig. 3: Analysis and Instrumentation Work Flow

several copies of identical basic blocks all referencing the same source location. Yet, if the predecessors of the respective basic blocks are known, it is usually clear which instance in the binary code would be executed during an actual execution.

Dynamic information can be used to resolve the remaining ambiguity in the mapping between source code and binary code which is used for the annotation of timing properties. Based on the corrected line information (Fig. 3, step 5), the binary-level control flow is analyzed to create code for performing a dynamic reconstruction of executed basic blocks (Fig. 3, step 9). For source code locations which are referenced by multiple binary basic blocks, the generated code decides which basic block would be executed during an execution of the actual target binary. Based on previously executed basic blocks, the remaining ambiguity can be resolved and the correct annotations for every basic block can be applied during native simulation of the instrumented source code.

Performing this dynamic path reconstruction also makes the approach more robust against incomplete line information. If some binary-level basic blocks cannot be mapped to a source code line, the control flow can still be reconstructed precisely using the information from surrounding basic blocks. Dynamically reconstructing binary control flow has the additional advantage that the transition between basic blocks can be annotated with an execution time, not just the basic blocks themselves.

Hence the effects of the processor pipeline, i.e. the performance gain from an overlapping execution of basic blocks or the performance penalty of pipeline flushes induced by branch instructions, can already be considered during the generation of the annotated source code. Thus dynamic effects can be simulated without the need for additional computations at runtime. To determine the execution time of machine instructions,

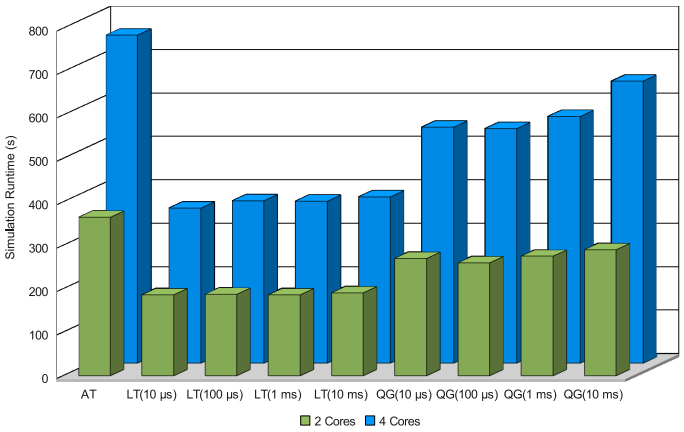


Fig. 4: Evaluation of Simulation Performance

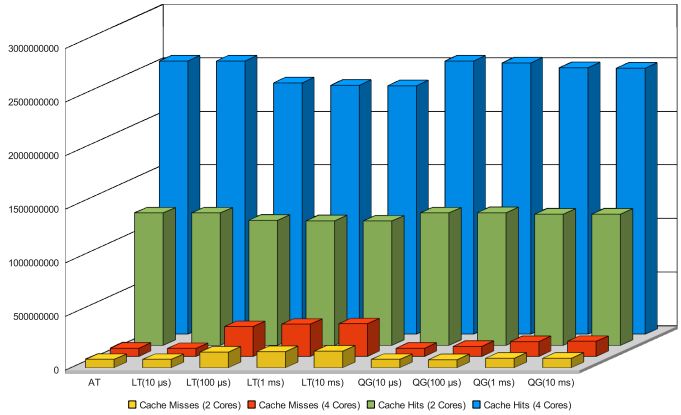


Fig. 5: Cache Simulation Accuracy

the commercial tool AbsInt aiT [9] was integrated into the presented analysis flow to produce a binary-level control flow graph annotated with execution times (Fig. 3, step 7). This approach allows the annotation of arbitrary properties to the original source since all machine instructions from the original binary can be accessed in the graph. For the experimental evaluation, instruction addresses were extracted to perform a simulation of the instruction cache. The approach can easily be extended for other non-functional properties e.g. power analysis using instruction-dependent or instruction-sequence-dependent power models [10].

V. Preliminary Experimental Results

The presented synchronization method was evaluated using a synthetic example platform and parts of a traffic sign recognition system. The simulated multi-core systems consisted of two and four ARM7TDMI cores. The cores were sharing 1kB of 2-way set associative instruction cache using a line size of 16 bytes and a PLRU replacement policy. All cores were running an instance of a circle recognition algorithm. Timing annotations were added to the source code of the program using the work flow described in the previous section. Additionally, the memory accesses for instruction fetches were extracted from the binary code and annotated as well. Instances of the annotated software components were combined with a cache model to form the model of the simulated platform. The cache model implemented various methods to synchronize the order of the memory requests created by the software components to determine whether an access is a cache hit or a cache miss.

To determine the synchronization overhead for cache simulation, three synchronization approaches were tested:

- No synchronization, meaning each access to the instruction cache was executed directly during TLM-LT simulation.
- Optimistic synchronization using the Quantum Giver approach.
- Explicit synchronization using calls to `wait` before each cache access which corresponds to TLM-AT.

For the test runs using the TLM-LT and Quantum Giver methods, the global time quantum was varied from 10 μs to 10 ms. The outcome of the performance evaluation in Fig. 4 depicts the simulation runtime in seconds. Results using explicit synchronization are labeled AT, those for temporal decoupled simulation without synchronization are labeled LT and the results for the Quantum Giver approach are labeled QG. The labels of the latter two also include the value of the global quantum used for simulation in round brackets.

For an unsynchronized TLM-LT simulation, the number of predicted cache misses doubles for each additional simulated processor core. The reason for this is that in the evaluated system, all cores access the same memory regions as they execute the same program code. During temporally decoupled simulation, the first scheduled thread simulating one of the cores loads data into the cache and also evicts it again before yielding control. The thread simulating the next core will then do the same, even if it accesses the same data as the first thread. Hence the simulation does not respect the effects which would occur in the real system. Similar effects would occur if all cores were executing different programs.

The estimates of the synchronized simulations report fewer cache misses as they represent the interaction of the cores correctly. If one instance of the algorithm changes the cache content and an instance running on another core accesses the same code region shortly afterwards, the simulation correctly detects a cache hit. The estimates provided by the Quantum Giver synchronization approach are very close to the estimates reported by lock-step simulation as long as the global quantum is not too large. Despite the additional overhead for ordering the transactions, the simulation performance is improved by about 25%. The optimistic synchronization approach is not completely accurate, as not all cache accesses are strictly ordered. For performance reasons, the delays created by conflicting transactions are accumulated for one complete round of the synchronization protocol. Hence if the values chosen for the global quantum are too large, the simulation results become less accurate. Nevertheless, it is more accurate than simulation without synchronization and considerably faster than lock-step simulation. The overhead of the optimistic synchronization approach can potentially be improved, as the implementation of the conflict resolution algorithm is not optimized yet. For example, cache accesses are stored in a simple list data structure which is traversed during conflict resolution.

Although the system design used for the experiments was not very complex, the results are already very encouraging. In larger systems and particularly for shared resources which are accessed by several other components, the advantage of the optimistic synchronisation approach can be even bigger as a larger number of components usually means more task switching overhead during lock-step simulation. Since the Quantum Giver synchronization approach allows a precise resolution of resource access conflicts, without the need to perform a lock-step simulation, the performance of larger systems can be analyzed accurately.

VI. Related Work

Several methods for dynamic performance evaluation using source code, enriched with timing annotations obtained from binary code, can be found in the literature. Earlier approaches do not support compiler optimizations [11]. More recently, the use of compiler optimizations is supported through the use of modified compiler tool chains [12][13]. A dynamic correction of statically determined execution times using a non-functional cache simulation was originally proposed by Schnerr et al. in [7]. Very similar approaches were presented by Lin et al. in [14] and Castillo et al. in [15], while the latter also describes a cache simulation specifically developed for non-functional simulation which increases simulation performance. Transaction level models of software components are used in [16] to simulate memory accesses, but the presented method requires synchronization before an access is performed. However, none of these publications provide solutions for an efficient temporally decoupled simulation of shared caches.

The concept of *Result-Oriented Modeling* [17] for buses in transaction level models shares many similarities with the presented Quantum Giver approach. It also relies on optimistic estimation of transaction duration and retroactive adjustments in case of an incorrect prediction. As the focus of the technique is to make conflict resolution more accurate, the adjustments can induce additional context switches and hence have a negative effect on simulation performance. This is not the case for the Quantum Giver approach. Result-oriented modeling has also been successfully used to simulate preemptive scheduling in real-time operating systems [18].

Reducing the number of simulated transactions by modeling sequences of similar transactions as a single transaction is used in [19] to increase simulation performance of TLM-2.0 models. Conflict resolution occurs through a central control instance called *Resource Model*, which is very similar to the Quantum Giver. The temporal order of transactions is not resolved precisely, which is however necessary to perform an accurate cache simulation.

VII. Limitations and Further Improvements

While the presented synchronization approach can improve simulation performance for some applications, it is not equally well-suited for all abstraction levels. In order to perform transactions optimistically, it is assumed that data accessed by the transaction is provided through the native execution on the simulation host. This assumption often holds for the simulation of non-functional properties, for example if a cache simulation is only performed to determine cache hits and misses. On the other hand, if the actual values stored in the cache are of interest for the simulation, e.g. since an instruction set simulator is used, explicit synchronization remains mandatory to avoid read-before-write conflicts.

VIII. Conclusion

This work presented a framework for fast and accurate performance analysis of software components using SystemC TLM-2.0. The Quantum Giver synchronization approach has been introduced to precisely model execution time influencing resource access conflicts of concurrent software tasks. Simulation of software timing properties is achieved by annotating low-level properties to the source code of a task before compiling it for the simulation host. These annotations are obtained from binaries for the simulated target architecture and can include many non-functional properties like timing, memory accesses or power consumption.

Experimental results have shown that the presented synchronization approach is almost as accurate as lock-step simulation, but significantly faster. Component interaction must be modeled precisely to obtain accurate performance estimates in multi-core systems, so the presented method is particularly useful for software performance analysis in these systems. Nonetheless, it can also be used for access synchronization at arbitrary shared resources like buses, memory hierarchies, peripherals and software scheduling during system-level performance analysis of complex embedded systems.

Acknowledgment

This work has been partially supported by the BMBF project SANITAS under grant 01M3088C and by the ITEA/BMBF project VERDE under grant 01|S09012A.

References

[1] E. Wandeler, L. Thiele, M. Verhoef, and P. Lieverse, "System Architecture Evaluation Using Modular Performance Analysis: a Case Study," *Int. J. Softw. Tools Technol. Transf.*, vol. 8, no. 6, pp. 649–667, 2006.

[2] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The Worst-Case Execution-Time Problem — Overview of the Methods and Survey of Tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, pp. 1–53, 2008.

[3] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst, "System Level Performance Analysis - the SymTA/S Approach," *Computers and Digital Techniques, IEE Proceedings*, vol. 152, no. 2, pp. 148–166, 2005.

[4] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand, "Memory Hierarchies, Pipelines, and Buses for Future Architectures in Time-Critical Embedded Systems," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 28, no. 7, pp. 966 –978, 2009.

[5] "Open SystemC Initiative (OSCI)," http://www.systemc.org.

[6] L. Benini, D. Bertozzi, D. Bruni, N. Drago, F. Fummi, and M. Poncino, "SystemC Cosimulation and Emulation of Multiprocessor SoC Designs," *Computer*, vol. 36, no. 4, pp. 53 – 59, 2003.

[7] J. Schnerr, O. Bringmann, A. Viehl, and W. Rosenstiel, "High-Performance Timing Simulation of Embedded Software," *45th Design Automation Conference (DAC 2008)*.

[8] "Open SystemC Initiative TLM-2.0 Language Reference Manual," http://www.systemc.org/downloads/.

[9] "AbsInt aiT WCET Analyzer," http://www.absint.com/ait/.

[10] B. Sander, J. Schnerr, and O. Bringmann, "ESL Power Analysis of Embedded Processors for Temperature and Reliability Estimations," in *CODES+ISSS '09: Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*.

[11] T. Meyerowitz, A. Sangiovanni-Vincentelli, M. Sauermann, and D. Langen, "Source-Level Timing Annotation and Simulation for a Heterogeneous Multiprocessor," *DATE '08: Proceedings of the conference on Design, automation and test in Europe*.

[12] A. Bouchhima, P. Gerin, and F. Petrot, "Automatic Instrumentation of Embedded Software for High Level Hardware/Software Co-Simulation," *14th Asia and South Pacific Design Automation Conference (ASP-DAC 2009)*.

[13] Z. Wang and A. Herkersdorf, "An Efficient Approach for System-Level Timing Simulation of Compiler-Optimized Embedded Software," *46th Design Automation Conference (DAC 2009)*.

[14] K.-L. Lin, C.-K. Lo, and R.-S. Tsay, "Source-Level Timing Annotation for Fast and Accurate TLM Computation Model Generation," *15th Asia and South Pacific Design Automation Conference (ASP-DAC 2010)*.

[15] J. Castillo, H. Posadas, E. Villar, and M. Martinez, "Fast Instruction Cache Modeling for Approximate Timed HW/SW Co-Simulation," in *GLSVLSI '10: Proceedings of the 20th Great lakes symposium on VLSI*.

[16] E. Cheung, H. Hsieh, and F. Balarin, "Memory Subsystem Simulation in Software TLM/T Models," *14th Asia and South Pacific Design Automation Conference (ASP-DAC 2009)*.

[17] G. Schirner and R. Dömer, "Result-Oriented Modeling - A Novel Technique for Fast and Accurate TLM," *IEEE Transactions on CAD of Integrated Circuits and Systems*, 2007.

[18] G. Schirner and R. Dömer, "Introducing Preemptive Scheduling in Abstract RTOS Models using Result Oriented Modeling," in *Design, Automation and Test in Europe (DATE 2008)*.

[19] W. Ecker, V. Esen, R. Schwencker, T. Steininger, and M. Velten, "TLM+ Modeling of Embedded HW/SW Systems," in *Design, Automation and Test in Europe (DATE 2010)*.