A Mixed-level Co-simulation Method for System-level Design Space Exploration

Mark Thompson Andy D. Pimentel Simon Polstra Cagkan Erbas
Informatics Institute, University of Amsterdam
Email: {mthompsn,andy,spolstra,cagkan}@science.uva.nl

Abstract

The Sesame modeling and simulation framework aims at efficient system-level design space exploration of embedded multimedia systems. A primary objective of Sesame is the exploration at multiple levels of abstraction. As such, it targets gradual refinement of its (initially abstract) architecture performance models while maintaining architecture-independent application specifications. In this paper, we present a mixed-level co-simulation method, called trace calibration*, for incorporating external simulators into Sesame's abstract system-level performance models. We show that trace calibration only requires minor modification of the incorporated simulators and that performance overheads due to co-simulation are minimal. Also, we show that trace calibration transparantly supports distributed co-simulation, allowing for effectively reducing the system-level simulation slowdown due to the incorporation of lower-level simulators.*

1 Introduction

Early design space exploration is becoming a crucial ingredient of system-level design of advanced embedded systems. In recent years, a fair number of system-level simulation-based exploration environments have been proposed, like MetroPolis [6], MESH [5], various SystemC-based environments (e.g., [14]), and our own Sesame [13] framework. The Sesame modeling and simulation framework aims at efficient system-level design space exploration of embedded multimedia systems, allowing rapid performance evaluation of different architecture designs, application to architecture mappings, and hardware/software partitionings. Moreover, it does so at multiple levels of abstraction and for a wide range of multimedia applications. Key to this flexibility is the separation of application and architecture models, together with an explicit mapping step to map an application model onto an architecture model.

A primary objective of Sesame is the support for architectural exploration at multiple levels of abstraction. To this end, Sesame targets gradual refinement of its architecture performance models while maintaining high-level and architecture independent application models. This allows for re-using application models in the exploration cycle. After architectural exploration with Sesame's abstract architecture performance models and the design decisions that result from this, the next step would be to gradually incorporate more detailed simulators for specific architecture components into the system-level model. This means that some parts of the system-level architecture model remain operating at Sesame's high level of abstraction while other parts are simulated by (external) lower-level simulators, thereby yielding a mixed-level co-simulation.

In this paper, we present an efficient co-simulation technique, called *trace calibration*, for incorporating external simulators into Sesame's abstract system-level performance models. We will show that trace calibration only requires minor modification of the incorporated simulators and that performance overheads due to mixed-level co-simulation are minimal. Also, we will demonstrate that distributed co-simulation – which is straightforward and transparent in trace calibration – can effectively reduce the simulation slowdown due to the incorporated lower-level simulators.

In the next section, we briefly describe the Sesame modeling and simulation framework. Section 3 provides some background on the field of mixed-level co-simulation and explains our contribution to this field in the context of the Sesame framework. In Section 4, we describe the trace calibration technique. Section 5 presents a number of experiments in which co-simulation efficiency is measured when incorporating the SimpleScalar instruction-set simulator [4] into Sesame. Finally, Section 6 concludes the paper.

2 The Sesame environment

To facilitate flexible performance analysis of embedded (media) systems architectures, the Sesame modeling and simulation environment [13] uses separate application and architecture models. An application model describes the functional behavior of an application while the architecture model defines architecture resources and captures their performance constraints. After explicitly mapping an application model onto an architecture model, they are co-simulated via trace-driven simulation. This allows for evaluation of

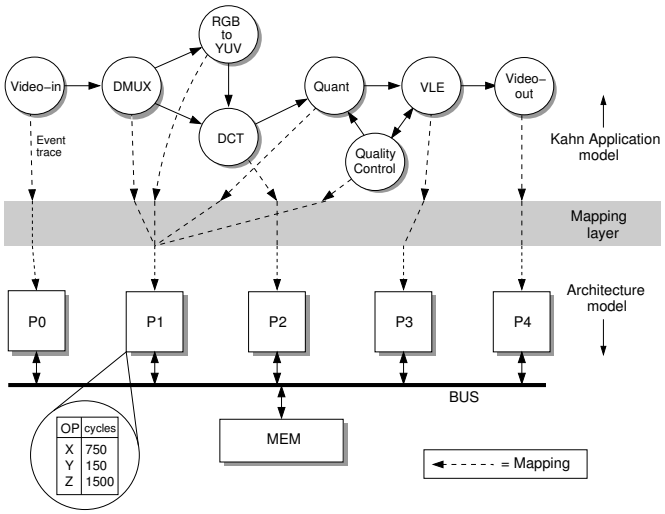

Figure 1. Modeling an Motion-JPEG application on an MP-SoC architecture.

the system performance of a particular application, mapping, and underlying architecture. Essential in this methodology is that an application model is independent from architectural specifics and assumptions on hardware/software partitioning. As a result, a single application model can be used to exercise different hardware/software partitionings and can be mapped onto a range of architecture models, possibly representing different architecture designs or modeling the same architecture design at various levels of abstraction. The layered infrastructure of Sesame is illustrated in Figure 1.

For application modeling, Sesame uses the Kahn Process Network (KPN) model of computation [9], which fits well to the multimedia application domain. In a KPN, parallel processes communicate with each other via unbounded FIFO channels, where reading from these channels is blocking and writing is non-blocking. The computational behavior of an application is captured by instrumenting the code of each Kahn process with annotations that describe the application's computational actions. The reading from and writing to Kahn channels represent the communication behavior of a process within the application model. By executing the Kahn model, each process records its actions in order to generate its own trace of *application events*, which is necessary for driving an architecture model. These application events typically are coarse grained, such as *Execute(DCT)* or *Read(channel_id,pixel-block)*.

An architecture model simulates the performance consequences of the computation and communication events generated by an application model. To model the timing consequences of application events, each architecture model component is parameterized with a table of operation latencies (illustrated for Processor 1 in Figure 1). The table entries could, for example, specify the latency of an *Execute(DCT)* event, or the latency of a memory access in the case of a memory component. The latency values are usually initialized using performance numbers from literature, and can be calibrated using measurements on available hardware or via lower-level simulations of architecture components.

To bind application tasks to resources in the architecture model, Sesame provides an intermediate *mapping layer*. This layer has three purposes [13]. First, it controls the mapping of Kahn processes (i.e. their event traces) onto architecture model components by dispatching application events to the correct architecture model component. The mapping also includes the mapping of Kahn channels onto communication resources in the architecture model. Second, the event dispatch mechanism in the mapping layer guarantees that no communication deadlocks occur in the case multiple application tasks are mapped onto a single architecture model component. In that case, the dispatch mechanism also provides various application event scheduling strategies. Finally, the mapping layer is capable of dynamically transforming application events into (lower-level) architecture events in order to facilitate flexible refinement of architecture models [13].

3 Background and related work

As mentioned before, a primary objective of Sesame is the support for gradual refinement of architecture performance models. It should, for example, be possible to refine only parts of an architecture model while leaving other parts at the higher abstraction level. The resulting *mixed-level simulations* enable more detailed performance evaluation of a specific architecture component within a system-level context. They therefore avoid the need for building a complete detailed architecture model during the early design stages. Moreover, mixed-level simulations do not suffer from deteriorated system evaluation efficiency caused by unnecessarily refined parts of the architecture model.

In prior work, we have studied mechanisms for refining Sesame's architecture models by incorporating dataflow actors in the mapping layer that allow for run-time transformation of application events into (more detailed) architecture events [13]. This event transformation technique enables architectural exploration at different levels of abstraction while maintaining high-level and architecture-independent application models. In this paper, the next step of model refinement will be addressed: the gradual incorporation of external lower-level simulation models into the Sesame environment. This yields mixed-level co-simulations consisting of abstract model components as well as lower-level simulators.

In the past decade, extensive work has been performed in the field of mixed-level HW/SW co-simulation, mainly from the viewpoint of co-verification. This has resulted in a multitude of academic and commercial co-simulation frameworks (e.g., [2, 1, 3, 8, 12, 11]). Such frameworks typically combine behavioral models, Instruction Set Simulators

(ISSs), bus-functional models or HDL models into a single co-simulation. These mixed-level co-simulations generally need to solve two important problems: i) making the co-simulation functionally correct by translating any differences in data and control granularity between simulation components, and ii) keeping the global timing correct by synchronizing the simulator components and overcoming differences in timing granularity. The functionality issue is usually resolved using wrappers, while global timing is typically controlled using either a parallel discrete-event simulation method [7] or a centralized simulation backbone. With respect to the latter, SystemC is nowadays a popular backbone for co-simulations [12, 11]. Synchronization between simulation components usually takes place using the finest timing granularity (i.e. lowest abstraction level) as the greatest common denominator between components. E.g., system-level co-simulations with cycle-accurate components are typically synchronized at cycle granularity, causing high performance overheads. Besides the performance overheads caused by wrappers and time synchronization, the IPC mechanisms often used for communication between the co-simulation components may also severely limit performance [10], especially when synchronizing at cycle granularity.

The next section presents a co-simulation technique, called trace calibration, that takes the opposite direction with respect to maintaining global timing. Instead of synchronizing simulation components at the finest timing granularity, it maintains correct global timing at the highest possible level of abstraction, being the level of Sesame's abstract architecture model components. As a result, the performance overhead caused by wrappers and time synchronizations is reduced to a minimum. Trace calibration shows some similarities with the recently proposed trace-driven co-simulation technique in [10]. However, the work of [10] operates at a lower abstraction level and is applied in a classical HW/SW co-simulation context. Consequently, its applicability to early design space exploration (e.g., rapidly assessing different HW/SW partitionings) is more limited. In Sesame, external simulators are gradually incorporated into an abstract application/architecture co-simulation, and this is done purely from the perspective of achieving effective system-level design space exploration. Eventually, the incorporation of more and more external simulators could lead to a classical HW/SW co-simulation. Moreover, in contrast to [10], we address distributed co-simulation to reduce the slowdowns caused by incorporating external simulators.

4 Trace calibration

As its name already suggests, the trace calibration technique accomplishes mixed-level co-simulation by means of dynamic calibration of the event traces that are generated by an application model. Rather than using fixed values in the latency tables of processing components in Sesame's architecture models (see Section 2), trace calibration dynamically computes – using lower-level simulators – the latency values of computational tasks. This is illustrated in Figure 2, where an instruction set simulator (ISS) is used for calibrating the trace from application process B. We note that Figure 2 focuses on the application model level, and only abstractly depicts the mapping and architecture model levels. Also, throughout the remainder of this paper, we take the example of incorporating an ISS into Sesame. However, other types of simulation models (like RTL models) can also be used with trace calibration.

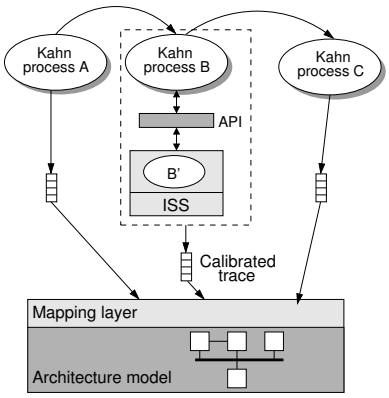

Figure 2. Incorporating an instruction set simulator using trace calibration.

In the example of Figure 2, the code from application process B is executed both in the Kahn application model and on the ISS (represented as B' in Figure 2). This allows us to keep the application model to a large extent unaltered, where B' runs as a "shadow process" of B to perform code fragment measurements. The two executions of B are synchronized by means of data exchanges using API calls, which are implemented with an underlying IPC mechanism. These data exchanges, which will be explained further on, only occur when the calibrated Kahn process performs communication. The simulator that performs the trace calibration (the ISS in Figure 2) measures the cycle counts between these (Kahn) communications. Subsequently, instead of generating fixed computational execution events, like *Execute(DCT)*, the Kahn application process generates *Execute(*Δ*)* events, where Δ equals to the actual cycle count taken by, for example, a DCT computation (or any other computation in between communications). Note that trace calibration is not restricted to the Kahn model of computation: it is equally applicable to other models of computation.

Compared to the original static operation-latency table method (see Section 2), a calibrated trace can more accurately represent the computational behavior of an architecture component. The system-level effects of this improved accuracy can subsequently be measured within Sesame's ar-

chitecture performance model which accounts for the global timing consequences of all (calibrated as well as non-calibrated) event traces generated by the application model. Efforts to quantify the improved accuracy, should however be performed with great care since their outcome heavily depends on the quality (i.e., accuracy) of the initial abstract performance models. This issue will be addressed briefly in Section 5. Moreover, trace calibration allows for gathering detailed execution statistics of the calibrated Kahn process(es). In the example of Figure 2, statistics on for example pipeline stalls and cache behavior for process B can be retrieved from the ISS. Also, Sesame's dataflow-based event refinement methodology [13] can still be applied to calibrated traces to, e.g., perform communication refinement, i.e., refine the *Read* and *Write* application events.

The increased accuracy comes however at the cost of higher execution times due to the inclusion of (slow) lower-level simulators. But, as will be demonstrated in the next section, these performance overheads due to wrappers (i.e., data and control exchange between simulation components) and time synchronizations are very small in our trace calibration technique. This is because time synchronization occurs at the highest level of abstraction, namely within Sesame's trace-driven architecture model, and data and control exchanges via our API only take place at (Kahn) communication points. In some occasions, it is even possible to eliminate almost all overheads related to trace calibration. For example, if no architectural exploration is performed on the architecture components that are simulated by the lower-level simulators, then the (calibrated and non-calibrated) traces could be generated once, storing them on disk, and can be re-used in the exploration process of the remaining parts of the system without rerunning the lower-level simulators.

To explain the details behind the trace calibration technique, consider Figure 3. This figure renders the dashed box from Figure 2 in more detail. The code in the Kahn application processes typically consists of alternating periods of communication and computation, as illustrated by the small code fragment for process B in Figure 3. In this fragment, some data is read from Kahn channel `c_in`, followed by some computational code (which may also be discarded, as will be explained later), after which the resulting data is written to Kahn channel `c_out`. The two boxes on the right of this code fragment indicate what the run-time system of the application model executes when it encounters the Kahn read and write communications. Note that these run-time system actions are automatic and transparent: the programmer does not need to add or change code. First, the run-time system queries the ISS via the API, using `API_get_cycles()`, to retrieve the current cycle count from the ISS. As will also be described later on, the ISS provides this cycle information by executing a matching `API_put_cycles()` call. The run-time system then generates an *Execute(*Δ*)* application event for the architecture model, where $\Delta = n_{cur} - n_{prev}$, i.e., Δ

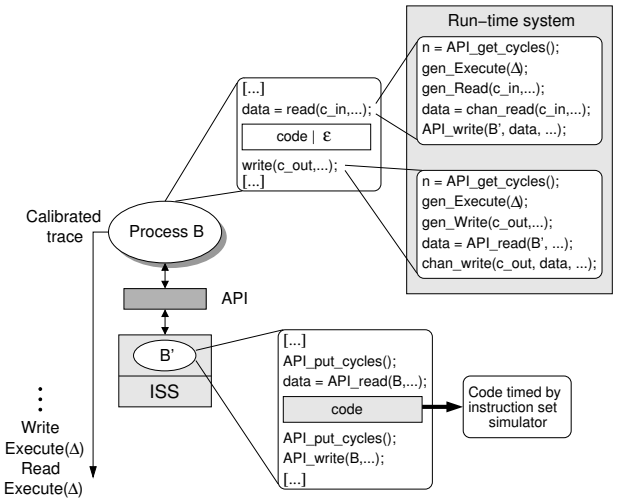

Figure 3. Interaction between application model and instruction set simulator.

equals to the time between the previous cycle query and the current one. Hence, the *Execute* event models the time that has past since the previous communication. Subsequently, a *Read* application event is generated for the architecture model. Hereafter, the actual read from Kahn channel `c_in` is performed. Finally, the data that has been read is copied, using `API_write`, to process B' running on the ISS.

Figure 3 also shows how the ISS side (process B') is handled. First, it sends the current cycle count of the ISS to the application model (`API_put_cycles`) to service the `API_get_cycles()` query from process B. Then, it reads the data that was sent by process B, i.e., the `API_read` from process B' matches up with the `API_write` from process B. After receiving the data, process B' executes the computational code shown in grey in Figure 3. This computational code is finished by a communication (a write to `c_out`), which again causes a cycle count query by the run-time system of the application model. The generated *Execute(*Δ*)* application event that follows, represents a detailed timing of the computational code on the ISS. Figure 3 also shows that process B' on the ISS first writes back the resulting data to process B in the application model before the latter forwards this data to Kahn channel `c_out`. This allows for discarding the computational code between the communications in process B in the application model. In that case, only process B' simulates computational functionality, while process B only communicates data with its neighboring application tasks. From the above, it should be clear that the `API_get_cycles` and `API_read` calls are blocking.

With trace calibration, it is relatively easy to incorporate any external low-level simulator into Sesame's system-level architecture models. Only three API functions need to be introduced in the low-level simulator or in the code that runs on it (in the case of an ISS): `API_put_cycles()`, `API_read()`, and `API_write()`.

Here, `API_put_cycles()` is the only function that needs a hook into the simulator to retrieve its cycle count. Most simulators provide such a hook, otherwise it can be created by a small modification to the simulator (as will be briefly explained in the next section). Using the API calls, the code to run on an ISS simulator (B') can be trivially derived from the application code (B). Therefore the total coding effort to enable trace calibration is small.

Moreover, for trace calibration, the execution of the lower-level simulators is location independent. That is, it is straightforward and completely transparent to place the lower-level simulators on different hosts, yielding *distributed co-simulation*. To this end, the implementation of the API between application process and lower-level simulator simply features different communication adaptors (e.g., shared memory or named pipes for local communication, and sockets for remote communication). As will be shown in the next section, the distributed co-simulation support considerably improves the scalability of the co-simulations in the case multiple lower-level simulators are incorporated.

As a side note, we need to mention that a source of inaccuracy, of which a similar situation is reported in [10], occurs when mapping multiple application tasks to a single (programmable) architecture component. In this situation, the traces from these application tasks would be calibrated by *different instances* of an ISS. The scheduling of the (calibrated) application events from the different traces is subsequently performed at Sesame's mapping layer. This approach has two major advantages. First, there is no need for running an OS-scheduler on an ISS since ISSs always execute a single task. Second, the ISS instances, representing a single processor in the architecture, can be executed in parallel on different hosts! However, since context-switching is not modeled by the ISSs, the simulated cache (performance) behavior in this situation may be inaccurate.

5 Experiments

To demonstrate that the performance overheads of trace calibration are low, we present a case study with a Motion-JPEG encoder application and a shared-memory MP-SoC architecture consisting of five processing elements. Sesame's application model, architecture model and mapping for this case study are shown in Figure 1. In this experiment, we have incorporated SimpleScalar's sim-outorder ISS [4] in Sesame's high-level model of the MP-SoC architecture. This required only one small extension of the ISS: a system-call that retrieves the cycle count, which is needed by `API_put_cycles()`. Small C macros implement the functions `API_read()`, `API_write()`, and `API_put_cycles()` and are compiled together with the application code executing on the ISS.

For the experiments, we have used a small cluster of unloaded Pentium M 1.7GHz (Debian) machines connected by 100 Mbit ethernet. In all simulation runs, we have simulated the encoding of 11 CIF frames with a resolution of 128x128 pixels. Table 1 shows the wall-clock times (averaged over several runs) for three different simulation configurations executed on a single host machine: a Sesame-only system-level simulation (without trace calibration), a trace-calibrated (mixed-level) co-simulation where the DCT application process is executed on the ISS (representing P2 in Figure 1), and a trace-calibrated co-simulation where both the DCT and VLE processes are executed on different ISSs (representing P2 and P3). For each simulation configuration, the number of simulated Kcycles/sec is also given. Table 1 clearly shows the performance drop when incorporating lower-level simulators in Sesame's architecture models. The results also show the efficiency of the Sesame-only simulation (8 Mcycles/sec when simulating a 5 core MP-SoC).

Table 1. Co-simulation performance.

	Sesame only	DCT on ISS	DCT+VLE on ISSs
Time (secs)	8.1	157.1	279.6
Kcycles/sec	8,000	414	233

To demonstrate that the performance decrease of our mixed-level co-simulations is almost entirely due to the lower-level simulators themselves and not due to co-simulation overheads, Figure 4 shows the execution time breakdowns of the two trace-calibrated co-simulation configurations from Table 1. The breakdowns clearly prove that the total execution times are totally dominated by the ISS execution times. The measured overheads are respectively 5% (DCT on ISS) and 1% (DCT+VLE on ISSs) of the total execution time. We suspect that the latter has lower overheads because of the scheduling of the two ISSs that allows for hiding some of the overheads caused by communications between the DCT/VLE processes and the ISSs.

By means of distributed co-simulation, the system-level simulation slowdown due to the incorporation of lower-level simulators can be effectively reduced. To illustrate this, Table 2 presents the performance effect of distributing the ISSs

Figure 4. Overhead for one or two ISSs.

Table 2. Distributed performance.

	DCT on remote ISS	DCT on local ISS VLE on remote ISS	DCT+VLE on remote ISSs
Time (secs)	144.2	160.6	150.2
Kcycles/sec	452	405	433

over different hosts. Here, we use the terms 'local' and 'remote' to indicate if an ISS is executed on respectively the same or a different host as Sesame's application and architecture models. The results show that distributing the lower-level simulators over multiple hosts is certainly beneficial. For example, by placing the ISSs for the DCT and VLE processes on two different hosts (see Table 2), a speedup of 1.86 is achieved in comparison to execution on a single machine (see Table 1). In this case, only 6.5% of the total execution time is due to overheads (including network overhead).

We also performed experiments in which less computational intensive application processes, like RGB-to-YUV and Quant, are trace-calibrated as well (these results are not shown in table form). A fully distributed co-simulation with ISSs for the DCT, VLE and RGB-to-YUV processes takes 158.3 seconds. Adding an ISS for the Quant process to the previous distributed co-simulation results in a wall-clock time of 165 seconds. These results indicate that, with distributed co-simulation, trace calibration scales well.

For comparison, [10] lists the performance of similar case studies using their own work, Seamless CVE and Synopsys System Studio. With respect to the latter two, our co-simulations are one to two orders of magnitude faster, while for our Sesame-only simulations this is even three orders. The reported performance of the state-of-the-art technique proposed in [10] approximates our co-simulation performance, but they have used the ARMulator ISS which is significantly faster than the SimpleScalar ISS we have used.

To get some indication of the accuracy gains of trace calibration, we have also performed several simple experiments that compare a fully trace-calibrated model to a partially uncalibrated model. In this case, the uncalibrated model components have to use estimated latency values for application events. Statically estimating the performance of code executed on a particular processor is a well-known problem as it may be hard to deduce the correct number and types of executed instructions due to data-dependent behavior and to determine the CPI for the processor. Our experiments show, e.g., that if for one processor (onto which the DCT is mapped) we know the right number and types of executed instructions but mispredict the CPI by only 0.07 (where the actual CPI is 0.43) to calculate the latencies for the application events, then the estimated performance for the *whole system* is off by 13%. If the misprediction is bigger or if more components are uncalibrated then the system-level accuracy is reduced even further.

6 Conclusions

In this paper, we have presented an efficient mixed-level co-simulation technique, called trace calibration. This technique has been prototyped within our Sesame modeling and simulation framework, which targets efficient system-level design space exploration of embedded multimedia systems. To evaluate trace calibration, we have used a Motion-JPEG case study in which we incorporated up to four external instruction-set simulators into Sesame's abstract performance models. These experiments show that trace calibration only requires minor modification of the incorporated simulators and that performance overheads due to co-simulation are very low. It was also demonstrated that distributed co-simulation – which is easy and transparent in trace calibration – allows for effectively reducing the slowdown due to the incorporation of lower-level simulators.

In the future, we intend to quantify the accuracy improvements due to the incorporation of lower-level simulators by using validation case studies against real system implementations running a wider range of multi-media applications.

References

[1] ConvergenSC, CoWare, http://www.coware.com/.

[2] Seamless, Mentor Graphics, http://www.mentor.com/.

[3] System Studio, Synopsys, http://www.synsopsys.com/.

[4] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35(2):59 – 67, Feb. 2002.

[5] A. Cassidy, J. Paul, and D. Thomas. Layered, multi-threaded, high-level performance design. In *Proc. of the Design, Automation and Test in Europe*, March 2003.

[6] F. Balarin et al. Metropolis: An integrated electronic system design environment. *Computer*, 36(4), April 2003.

[7] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, Oct. 1990.

[8] K. Hines and G. Borriello. Dynamic communication models in embedded system co-simulation. In *Proc. of the Design Automation Conference*, pages 395–400, June 1997.

[9] G. Kahn. The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 74*, 1974.

[10] D. Kim, Y. Yi, and S. Ha. Trace-driven hw/sw cosimulation using virtual synchronization technique. In *Proc. of the Design Automation Conference*, June 2005.

[11] L. Benini et al. SystemC cosimulation and emulation of multiprocessor SoC designs. *Computer*, 36(4):53–59, 2003.

[12] P. Gerin et al. Scalable and flexible cosimulation of SoC designs with heterogeneous multi-processor target architectures. In *Proc. of the Int. Conference on Asia South Pacific Design Automation*, pages 63–68, 2001.

[13] A. Pimentel, C. Erbas, and S. Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Trans. on Computers*, 55(2), 2006.

[14] T. Kogel et al. Virtual architecture mapping: A SystemC based methodology for architectural exploration of system-on-chip designs. In *Proc. of the workshop on Systems, Architectures, Modeling and Simulation*, pages 138–148, 2003.