Journal of Signal Processing Systems 50, 99–114, 2008

DOI: 10.1007/s11265-007-0085-2

Calibration of Abstract Performance Models for System-Level Design Space Exploration

ANDY D. PIMENTEL, MARK THOMPSON, SIMON POLSTRA AND CAGKAN ERBAS
Computer Systems Architecture Group, Informatics Institute, University of Amsterdam, Kruislaan 403, 1098 SJ, Amsterdam, The Netherlands

Received: 24 January 2007; Revised: 27 March 2007; Accepted: 8 May 2007

Abstract. High-level performance modeling and simulation have become a key ingredient of system-level design as they facilitate early architectural design space exploration. An important precondition for such high-level modeling and simulation methods is that they should yield trustworthy performance estimations. This requires validation (if possible) and calibration of the simulation models, which are two aspects that have not yet been widely addressed in the system-level community. This article presents a number of mechanisms for both calibrating isolated model components as well as a system-level performance model as a whole. We discuss these model calibration mechanisms in the context of our Sesame system-level simulation framework. Two illustrative case studies will also be presented to indicate the merits of model calibration.

Keywords: system-level modeling and simulation, performance analysis, model calibration

1. Introduction

The increasing complexity of modern embedded systems has led to the emergence of system-level design [21]. A key ingredient of system-level design is the notion of high-level modeling and simulation in which the models allow for capturing the behavior of system components and their interactions at a high level of abstraction. As these high-level models minimize the modeling effort and are optimized for execution speed, they can be applied at the very early design stages to perform, for example, architectural design space exploration. Such early design space exploration is of eminent importance as early design choices heavily influence the success or failure of the final product.

A fair number of promising system-level simulation-based exploration environments have been proposed in recent years, such as Metropolis [6], MESH [13], Milan [27], Sesame [14, 31], and various SystemC-based environments like GRACE++ [24]. These environments typically facilitate efficient and flexible performance evaluation of embedded systems architectures. However, in the system-level performance modeling domain, two important and closely related aspects, namely *model validation* and *model calibration* have received relatively little attention. Figure 1 depicts a conceptual view of these two aspects in relation to a simulation (performance) model. Model validation refers to the testing of the extent to which the model's performance estimates reflect the actual behavior. Model calibration entails the fine-tuning of the model parameters such that the model's performance predictions more accurately reflect the actual behavior. Both model validation and calibration require (detailed) reference information about the system under study and its performance behavior, which may originate from datasheets (or other forms of detailed documentation), low(er)-level simulators, or

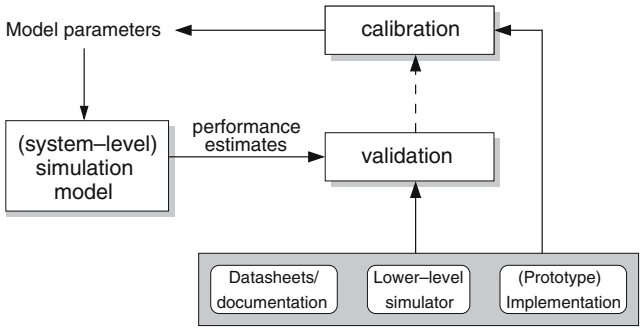

Figure 1. Validation and calibration of architectural simulation models.

actual (prototype) implementations of the system. In addition, model calibration may also use validation results if available. Since the sources of (detailed) reference information usually are not abundant during the early design stages, validation and calibration of system-level performance models remains an open and challenging problem.

In this article, we address the calibration of system-level performance models. More specifically, we discuss model calibration in the context of our Sesame simulation framework [14, 31]. Sesame aims at efficient system-level performance analysis and design space exploration of embedded multimedia systems. It allows for rapid 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 and signal processing applications.

We will present an approach for providing support for calibrating the model components in Sesame's system-level architecture models. To this end, we use ISS-based mechanisms for calibrating programmable model components and an automated component synthesis approach to calibrate dedicated model components. In addition, we will also discuss a strategy for validating and calibrating a constellation of calibrated Sesame model components, forming a system-level model. Using a Motion-JPEG (M-JPEG) encoder application, we will illustrate two cases of model calibration: one case focusing on calibration at the model component level and one case performing calibration at the system level.

The remainder of this article is organized as follows. In the next section, we briefly introduce the Sesame system-level simulation framework. Section 3 provides an overview of the mechanisms used for calibrating Sesame's architecture model components. Section 4 discusses a strategy for validating and calibrating Sesame's system-level models as a whole. In Section 5, we present several experiments with a M-JPEG encoder illustrating model calibration, and also provide some validation results. Section 6 describes related work. Finally, Section 7 concludes the article and discusses future work.

2. The Sesame Environment

The Sesame modeling and simulation environment [14, 31] facilitates performance analysis of embedded (media) systems architectures according to the Y-chart design approach [6, 22]. This means that Sesame recognizes separate application and architecture models, where an application model describes the functional behavior of an application and 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 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, assumptions on hardware/software partitioning, and timing characteristics. 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 shown in Fig. 2.

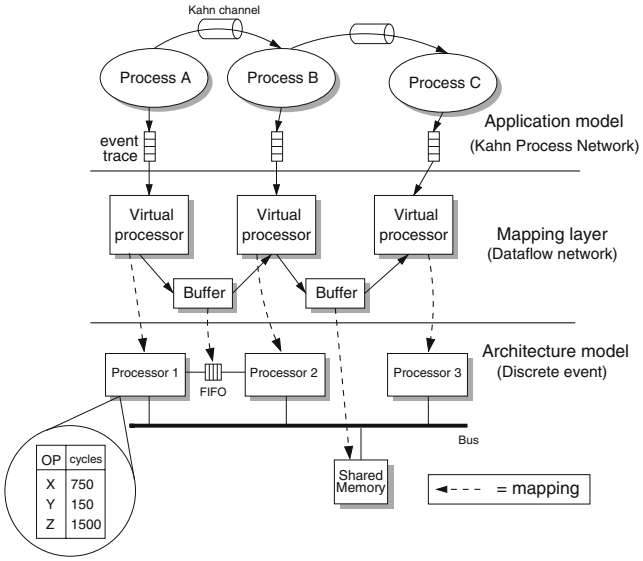

Figure 2. Sesame's application model layer, architecture model layer, and mapping layer which interfaces between application and architecture models.

For application modeling, Sesame uses the Kahn Process Network (KPN) model of computation [20, 25]. This implies that applications are structured as a network of *concurrent communicating processes*[1], thereby expressing the inherent task-level parallelism available in the application and making communication explicit. The Kahn application models, which are implemented in C++, are either derived by hand (e.g. from existing sequential code) or are generated by tools such as Compaan [34, 37] – which will be discussed later on – or KPNgen [39].

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 computational and communication 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. It solely accounts for architectural (performance) constraints and does not need to model functional behavior. This is possible because the functional behavior is already captured in the application model, which subsequently drives the architecture simulation. 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 Fig. 2). 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. This trace-driven simulation approach allows to, for example, quickly assess different HW/SW partitionings by simply experimenting with the latency parameters of processing components in the architecture model (i.e., a low latency for a computational action refers to a HW implementation while a high latency mimics a SW implementation). Sesame's architecture models are implemented in SystemC or Pearl [14]. The latter is a small but powerful discrete-event simulation language providing easy construction of high-level architecture models and fast simulation.

To interface between application and architecture models, Sesame features an intermediate *mapping layer*. This layer, which is automatically generated, consists of virtual processor components and FIFO buffers for communication between the virtual processors. There is a one-to-one relationship between, on one hand, the Kahn processes and channels in the application model and, on the other hand, the virtual processors and buffers in the

mapping layer. But unlike the Kahn channels, the buffers in the mapping layer are limited in size, and their size is dependent on the modeled architecture. The mapping layer has three purposes [31]. 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 buffers in the mapping layer 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 (i.e., multiple event traces) 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 [30, 31]. For a more detailed overview of Sesame, we refer the reader to [31].

3. Model Calibration

Calibration (and validation) of Sesame's system-level architecture models plays a continuous and ever-returning role. The following questions should be asked by the designer repeatedly:

1. "Are the values specified in the operation latency tables of Sesame's architecture model components reflecting realistic performance behavior?", and
2. "Is the constellation of model components – constituting the system-level model – adequately reflecting the actual system behavior?"

Where in Pimentel et al. [32] we only focused on the calibration of the model components' latency tables, i.e. addressing the first question, this article will also address the second question. In the remainder of this section, however, we limit our discussion to the mechanisms we use for calibrating the performance parameters (i.e., the operation latency tables) of *separate model components* in Sesame's system-level architecture models. These mechanisms can be classified into those used for the calibration of programmable model components and those to calibrate dedicated model components. Calibration of communication or memory model components will not be addressed in this article. In the subsequent section, we will elaborate on our strategy for validating and calibrating our system-level models as a whole.

3.1. *Calibration of Programmable Model Components*

To calibrate a Sesame architecture model component such that it adequately mimics the performance of a programmable processor (e.g., a general purpose processor core, DSP, etc.), one could of course use documented performance behavior of the processor or real performance measurements on the processor, if these are available. In addition, Sesame also provides explicit support for calibrating programmable model components. More specifically, Sesame allows for both *off-line* and *on-line* calibration of model components using an Instruction Set Simulator (ISS). Both types of calibration are illustrated in Fig. 3, which is based on a more abstract representation of Fig. 2. Currently, Sesame only supports the SimpleScalar ISS[2] [4] for calibration purposes, but other ISSs could also be added with relative ease. As we will discuss below, on-line and off-line calibration provide different trade-offs between accuracy and simulation performance.

In off-line model calibration, the ISS is used to statically (i.e., before system-level simulation) calibrate the values in an operation latency table according to code-fragment performance measurements on the ISS. To explain this in more detail, consider Fig. 3a. In this example, we assume that model component *p2* – onto which application process B is mapped (see Fig. 2) – needs to be calibrated using the ISS. This means that the code of Kahn application process B is cross-compiled for the ISS (indicated by B′ in Fig. 3a). The cross-compiled code is further instrumented such that it measures the performance of the code fragments that relate to the computational application events generated by the application process. For example, if process B can generate an *execute(DCT)* event, then the performance of the code in process B that is responsible for the DCT calculation is measured. To this end, we instrument the code at assembly level (currently done manually) to indicate where to start and stop the timing of code fragments. In the case of

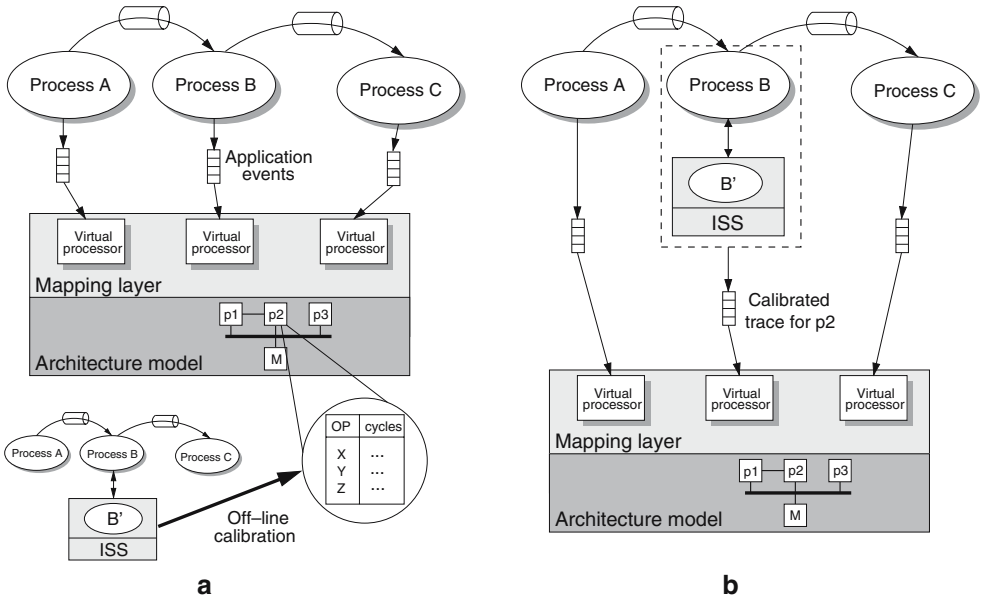

Figure 3. Calibration of programmable model components. **a** Off-line calibration, and **b** on-line calibration (also referred to as trace calibration).

the SimpleScalar ISS, we use its annote instruction field for this purpose.

To perform the actual code fragment timings for application process B, the code of this process is executed both in the Kahn application model and on the ISS (the cross-compiled B′). 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 – implemented with an underlying IPC mechanism – which are needed to provide B′ (on the ISS) with the correct application input-data. These data exchanges – which will be illustrated and explained using Fig. 4 and Table 1 later on – only occur when the Kahn application process taking part in the calibration (process B in our example) performs communication. For example, when Kahn process B reads data from its input channel, it forwards the data to process B′ on the ISS, i.e., process B′ reads and writes its data from/to process B instead of a Kahn channel. During execution, the ISS keeps track of the code fragment timings, which are *average timings* over multiple invocations of a code fragment. The resulting average timings are then used for (manually) calibrating the latency values of the architecture model component in question (in this case *p2*). Off-line calibration is a relatively efficient mechanism since it is performed before system-level simulation and basically is a one-time effort. However, if there is high variability in the demands of computations (i.e., data-dependant computations) then the measured average timings may be of moderate accuracy.

In on-line model calibration, which is illustrated in Fig. 3b, the ISS is incorporated into the system-level simulation to dynamically "calibrate an application event trace" destined for the architecture model. This technique, which we also refer to as *trace calibration*, essentially yields a mixed-level co-simulation of high-level Sesame architecture model components and one or more low(er)-level ISSs [36]. Rather than using fixed values in the latency tables of model components, on-line calibration dynamically computes the *exact latencies* of computational tasks using the ISS. In the example of Fig. 3b, the code from application process B is again executed both in the Kahn application model and on the ISS, like is done in off-line calibration. The ISS measures the cycle count of any computational task *in between the Kahn communications* in process B. Subsequently, instead of generating symbolic computational execution events, like *Execute(DCT)*, application process B now generates $Execute(\Delta)$ events, where Δ equals to the actual measured number of cycles taken

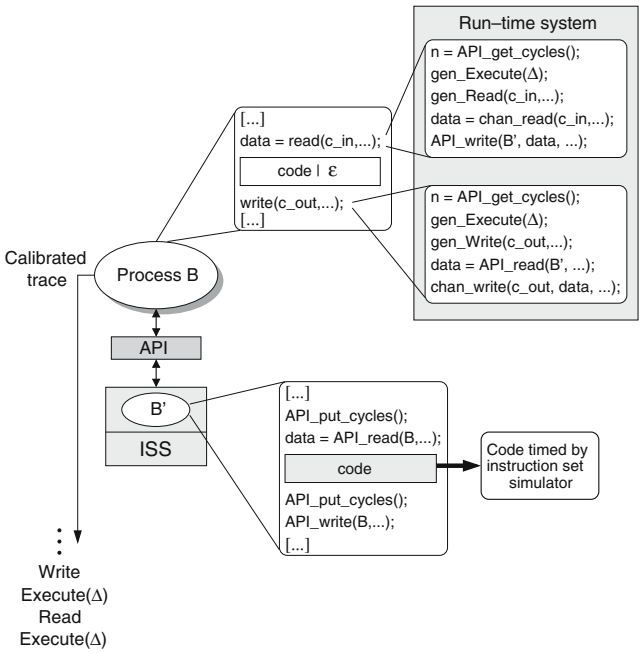

Figure 4. On-line model calibration: interaction between application model and instruction set simulator.

by, for example, a DCT computation (or any other computation in between communications). Clearly, on-line calibration trades off simulation performance for increased accuracy when compared to off-line calibration.

To provide a more detailed description of on-line model calibration with its synchronizations and data exchanges, consider Fig. 4. This figure renders the dashed box from Fig. 3b 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 Fig. 4. 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 the code fragment in Fig. 4 indicate what the run-time system of the application model executes when it encounters the read and write Kahn communications. Note that these run-time system actions are automatic and transparent: the programmer does not need to add or change code. For the case of an application read, Table 1 also shows the sequence of run-time system operations together with the information exchanges they cause. 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 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

Table 1. Run-time system operations for *read* from channel `c_in`.

Run-time system operation process B	Exchanged information	Information exchange partner
1. `n` = `API_get_cycles()`	Cycle count	From ISS (via API)
2. `gen_Execute`(Δ)	*Execute*(Δ) event	To architecture model
3. `gen_Read`(`c_in`, ..)	*Read* event	To architecture model
4. `data` = `chan_read`(..)	Data	From application process
5. `API_write`(B′, `data`, ..)	Data	To ISS (via API)

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′ that is running on the ISS.

Figure 4 also shows how the ISS side (process B′) handles the read communication. First, it sends the current cycle count of the ISS to the application . model (`API_put_cycles`) in order 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′ can execute the computational code shown in grey in Fig. 4. 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 4 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 makes it possible to discard 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 in the application model only communicates data with its neighboring application tasks. From the above, it should be clear that the `API_get_cycles` and `API_read` calls have blocking semantics.

3.2. *Calibration of Dedicated Model Components*

To calibrate model components that mimic the performance behavior of a dedicated implementation of a certain task, Sesame exploits a tool-set that has been developed at Leiden University. This tool-set, consisting of the Compaan [34, 37] and Laura [34, 40] tools, is capable of transforming a sequential application specification into a parallel application specification (a KPN to be more specific), and subsequently allows for producing synthesizable very high-level design language (VHDL) code that implements the application specified by the KPN for a specific field programmable gate-array (FPGA) platform.

Figure 5 illustrates how the Compaan and Laura tools can be applied for the purpose of model calibration. Let us assume that model component *p3* is a dedicated implementation of application process C. To calibrate this model component, the (sequential) code from application process C is first converted into a parallel KPN using the Compaan tool. By means of automated source-level transformations, Compaan is able to produce different input–output equivalent KPNs [33], in which for example the degree of parallelism can be varied. Since the different KPNs lead to different hardware implementations in the end, the transformations provided by Compaan are a mechanism to control the synthesis process. Using the Laura tool, a Compaan-generated KPN can subsequently be synthesized to VHDL code which can then be mapped (using regular commercial tools) onto an FPGA (-based) platform. As will be shown in Section 5,

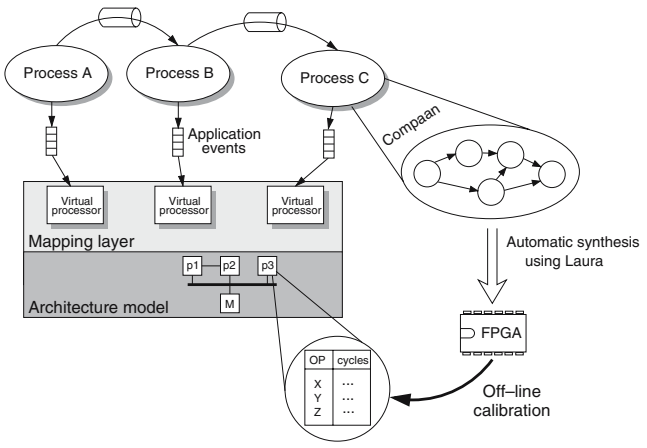

Figure 5. Off-line calibration of dedicated model components.

one of the reconfigurable platforms that Laura can use as a mapping target is the Molen platform [38].

This automated synthesis trajectory for specific application tasks can be traversed in the order of minutes. Actually, the place & route onto the FPGA platform is currently the limiting stage in the trajectory. Evidently, such synthesis results can be used to calibrate the performance parameters of Sesame's model components that represent dedicated hardware blocks. In Section 5, this will be illustrated using a case study with a M-JPEG encoder application.

4. System-Level Validation and Calibration

Given the assumption that the platform architecture under study is composed from a library of pre-determined and pre-verified IP components, we are also able to validate and calibrate our system-level models as a whole, i.e., the constellation of calibrated model components. To this end, we use the ESPAM design flow [29], which like Compaan and Laura is also developed at Leiden University. Figure 6 illustrates how this system-level validation and calibration using ESPAM is realized.

A pivotal element in this approach is the XML-based system-level specification, specifying the deployed platform architecture, the Kahn application(s), and the mapping of the application(s) onto the platform architecture. Here, the platform architecture specification consists of a system-level netlist description specifying which components from the IP library are used and how they are connected. Currently, the IP components that are available in the library include a variety of programmable processors (PowerPCs and MicroBlazes), dedicated hardware blocks (like a DCT), memories (random access as well as FIFO buffers), and interconnects (point-to-points links, shared bus, and crossbar switch). Using a generic interface component, these aforementioned components can readily be glued together to form a large variety of different platform instances.

On one hand, the system-level specification is used as input to Sesame in order to perform system-level performance analysis using abstract models of the IP components (as described in Section 2). On the other hand, the system-level specification can also act as input to ESPAM, together with RTL versions of the components from the IP library, to automatically generate synthesizable VHDL that implements the

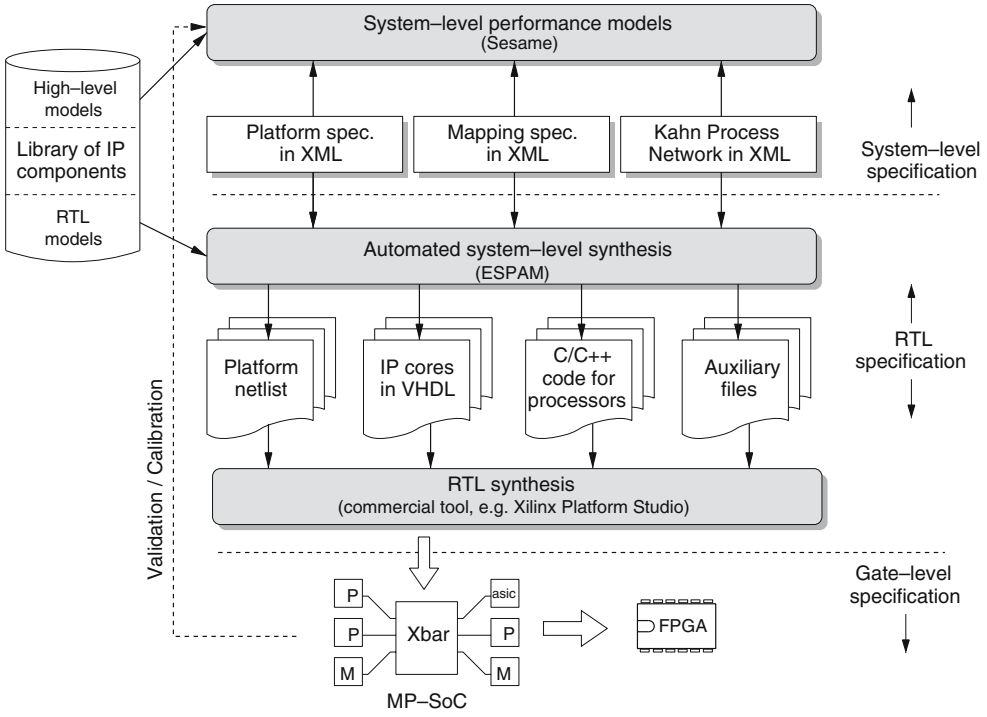

Figure 6. System-level model validation and calibration using the ESPAM design flow.

candidate platform architecture. In addition, ESPAM also generates the C/C++ code for those application processes that are mapped onto programmable cores. Using commercial synthesis tools and compilers, this implementation can readily be mapped onto an FPGA for prototyping. For a relatively complex multi-processor system-on-chip platform, ESPAM's system-level synthesis design flow (including the physical implementation) can be traversed in only a matter of hours [29]. Clearly, the results from such rapid system-level synthesis and prototyping allows for calibrating and validating Sesame's system-level models. This will also be illustrated in the next section.

5. Experiments

First, we present an experiment that illustrates how model (component) calibration can be performed using the Compaan/Laura tool-set (as described in Section 3.2). To this end, we modeled a M-JPEG encoder application and selected the DCT task from this application to be used for model calibration. In other words, the DCT is assumed to be implemented as a dedicated hardware block, and Sesame's model component accounting for the DCT's performance behavior needs to be calibrated accordingly. This implies that the DCT task is taken "all the way down" to a hardware implementation to gain more insight in its low-level performance aspects. To do so, the following steps were taken[3], which are integrally shown in Fig. 7. The DCT was first isolated from the M-JPEG code and used as input to the Compaan tool. Subsequently, Compaan generated a KPN application specification for the DCT task. This DCT KPN is internally specified at pixel level but has in- and output tasks that operate at the level of pixel blocks because the original M-JPEG application specification also operates at this pixel-block level.

Using the Laura tool, the KPN for the DCT task was converted into a VHDL implementation, in which for example the 2D-DCT component is implemented as a 92-stage pipelined IP block. This implementation can subsequently be mapped onto an FPGA platform. In this example, the DCT implementation was mapped onto the Molen reconfigurable platform [38]. The Molen platform connects a programmable processor (depicted as Core Processor in Fig. 7) with a reconfigurable processor which is based on FPGA technology. It uses microcode to incorporate architectural support for the reconfigurable processor (i.e., to control the reconfiguration and execution). By mapping the Laura-generated DCT implementation on Molen's reconfigurable processor and mapping the remainder of the M-JPEG code onto Molen's core processor, we can study the hardware DCT implementation, for the purpose of model calibration, in the context of the M-JPEG application.

For the Sesame system-level simulation part of the experiment, we decided to model the Molen reconfigurable platform architecture itself. This gives us the opportunity to actually validate our performance estimations against the real numbers from the implementation. The resulting system-level Molen model contains two processing components (Molen's core and reconfigurable processors) which are bi-directionally connected using two uni-directional FIFO buffers. Like in the real Laura→ Molen mapping, we mapped the DCT Kahn process from our M-JPEG application model onto the reconfigurable processor in the architecture model, whereas the remaining Kahn processes were mapped onto the core processor component.

The reconfigurable processor component in our architecture model was also refined – using our dataflow-based architecture model refinement methodology as discussed in [30, 31] – such that it models the pixel-level pipelined DCT from the Compaan/Laura implementation. Here, we used low-level information – such as pipeline depth of the Preshift and 2D-DCT units, latencies for reading/writing a pixel from/to a buffer and so on – from the Compaan/Laura implementation to calibrate the reconfigurable processor component in our system-level model. The core processor component in the architecture model was not refined, implying that it operates (i.e., models timing consequences) at the same (pixel-block) level as the application events it receives from the application model. The performance parameters of this model component have been calibrated using several simple (off-line) timing experiments performed on Molen's core processor. Here, we would like to note that the resulting system-level architecture model is mixed-level since the reconfigurable processor component is modeled at a lower level of abstraction (i.e., it has been refined to account for the pipelined DCT implementation) than the core processor component.

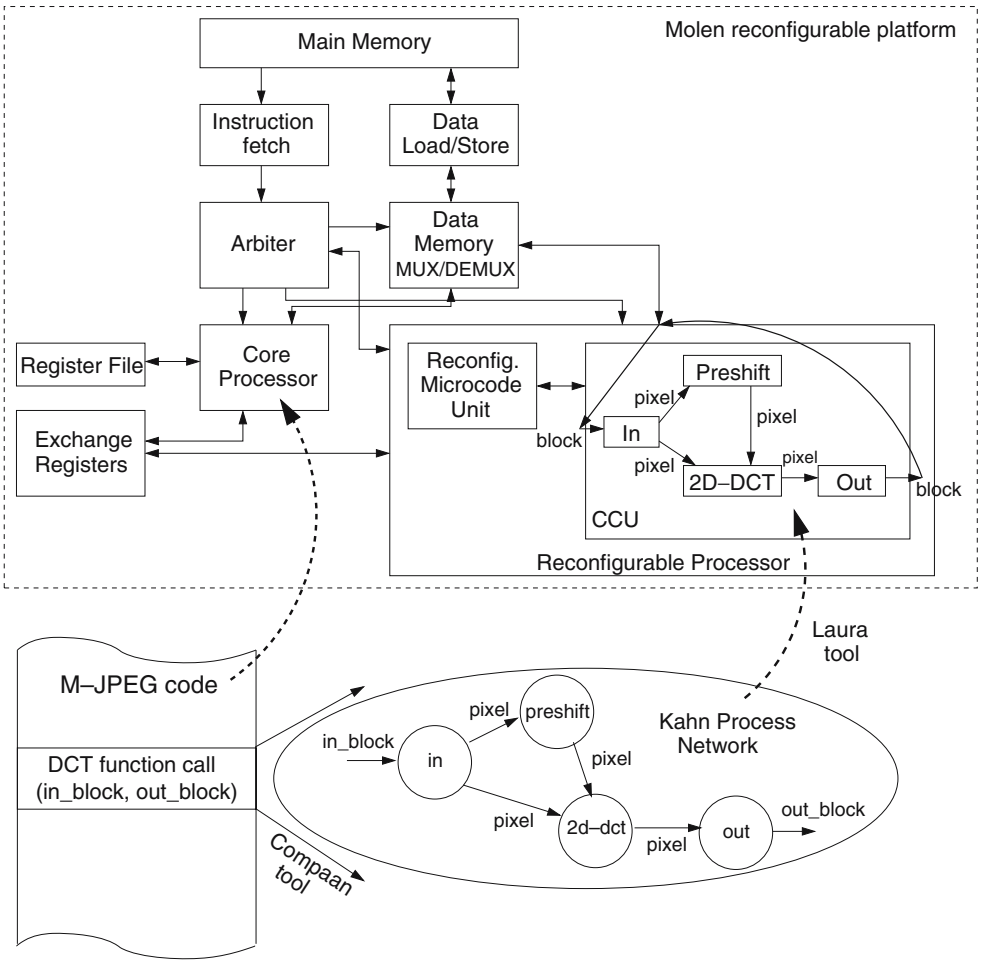

Figure 7. Model calibration for the DCT task using the Compaan/Laura tool-set and the Molen reconfigurable platform.

To check whether or not the resulting model, which was calibrated at model component level, produces accurate performance estimates at the system level, we compared the performance of the M-JPEG encoder application executed on the real Molen platform with the results from our system-level performance model. Table 2 shows the validation results for a sequence of sample input frames. The results from Table 2 include both the cases in which all application tasks are performed in software (i.e., they are mapped onto Molen's core processor) and in which the DCT task is mapped onto Molen's reconfigurable processor. Here, we would like to stress that we did not perform any tuning of our system-level model with Molen's M-JPEG execution results (i.e., we did not perform multiple validation↔calibration iterations, see Fig. 1). The results from Table 2 clearly indicate that Sesame's system-level performance estimations are, with the help of model calibration, quite accurate.

In a second experiment, we again used the M-JPEG application. We now mapped the application onto a cross-bar based multi-processor platform with up to 4 processors (either MicroBlaze or PowerPC) and distributed memory. This platform architecture is fully composed of IP components from the library supported by ESPAM (see Section 4). The processor, memory and interconnect components in Sesame's architecture model were therefore directly taken from the high-level model IP component library, as shown in Fig. 6. Only the performance parameters specific to the selected platform architecture needed to be specified, such as the latencies for computational actions, the latencies for setting up and communicating over the crossbar, and so on. We determined the values of these performance param-

Table 2. Validation results of the M-JPEG experiment.

	Real Molen (cycles)	Sesame simulation (cycles)	Error (%)
Full SW implementation	84581250	85024000	0.5
DCT mapped onto reconf. Processor	39369970	40107869	1.9

eters by a combination of ISS-based calibration (for the computational latencies of the MicroBlaze and PowerPC processors) as well as from low(er)-level information from the RTL versions of the IP components themselves.

Using the above platform model, we performed a design space exploration experiment by considering three degrees of freedom: the number (1–4) and type (MicroBlaze or PowerPC) of processors in the platform, and the mapping of application tasks to processors. The network configuration (crossbar switch) as well as the buffer/memory organization and sizes remained unaltered. Because of Sesame's efficiency, we were able to exhaustively explore the resulting design space – consisting of 10,148 design points – using system-level simulation, where the M-JPEG application was executed on 8 consecutive 128×128 resolution frames for each design point. This design space sweep took 86 min on a mid-range laptop. Figure 8a shows for each platform instance the estimated performance of the best application mapping found during exploration[4]. These particular platform instances and application mappings have also been synthesized and prototyped using ESPAM. The performance results of these implementations are shown in Fig. 8c. Table 3a provides the errors in percentages between the simulation and prototyping results. Clearly, the simulation results from Fig. 8a do not reflect the same behavior as the actual results from the implementations. This is also shown by the significant errors in Table 3a.

To address these discrepancies, we first looked at further calibrating the separate model components. For example, we used real performance measurements from the MicroBlaze and PowerPC cores instead of ISS-based measurements to calibrate processor model components. However, these additional model component calibrations did not improve the situation. Therefore, we shifted our focus to system-level aspects of our performance model and, to this end, used ESPAM for calibration purposes (as discussed in Section 4). Here, we found that the problem was caused by the mechanism for scheduling application tasks on shared resources in the multi-processor architecture. Where in our simulations a dynamic scheduling policy was used (the default in Sesame), the real implementation applies a static approach in which application tasks are 'merged' according to a static schedule determined at compile time. As a result, the mappings found by means of simulation with dynamic scheduling (see Fig. 8a) perform poorly when implemented with ESPAM because in these cases the static scheduling causes some undesired sequentializations of the parallel application. After adapting Sesame's application-event scheduler at the mapping layer to better reflect the actual static scheduling behavior[5], Sesame's performance estimates for the same set of application mappings now show the correct performance trends (see Fig. 8b), with an average error of 11.7% and worst-case error of 19% (as shown in Table 3b). The remaining inaccuracies in terms of absolute cycle numbers are mainly caused by the modeling of the PowerPC processors. This is because these processors are connected to the crossbar using a bus that is also used for access to the processor's local data and instruction memory. Since we do not explicitly model (contention on) this bus, our abstract PowerPC performance model is too optimistic.

6. Related Work

Model calibration is a well-known and widely-used technique in many modeling and simulation domains. In the computer engineering domain, the calibration of performance models is mostly applied in cycle-accurate modeling of system components like processor simulators, e.g., Black and Shen [9] and Moudgill et al. [28]. So far, the calibration of high-level performance models that aim at (early) system-level design space exploration has not been widely addressed yet. The work in Mathur and Prasanna [26] proposes a so-called vertical simulation approach that shows similarities with our

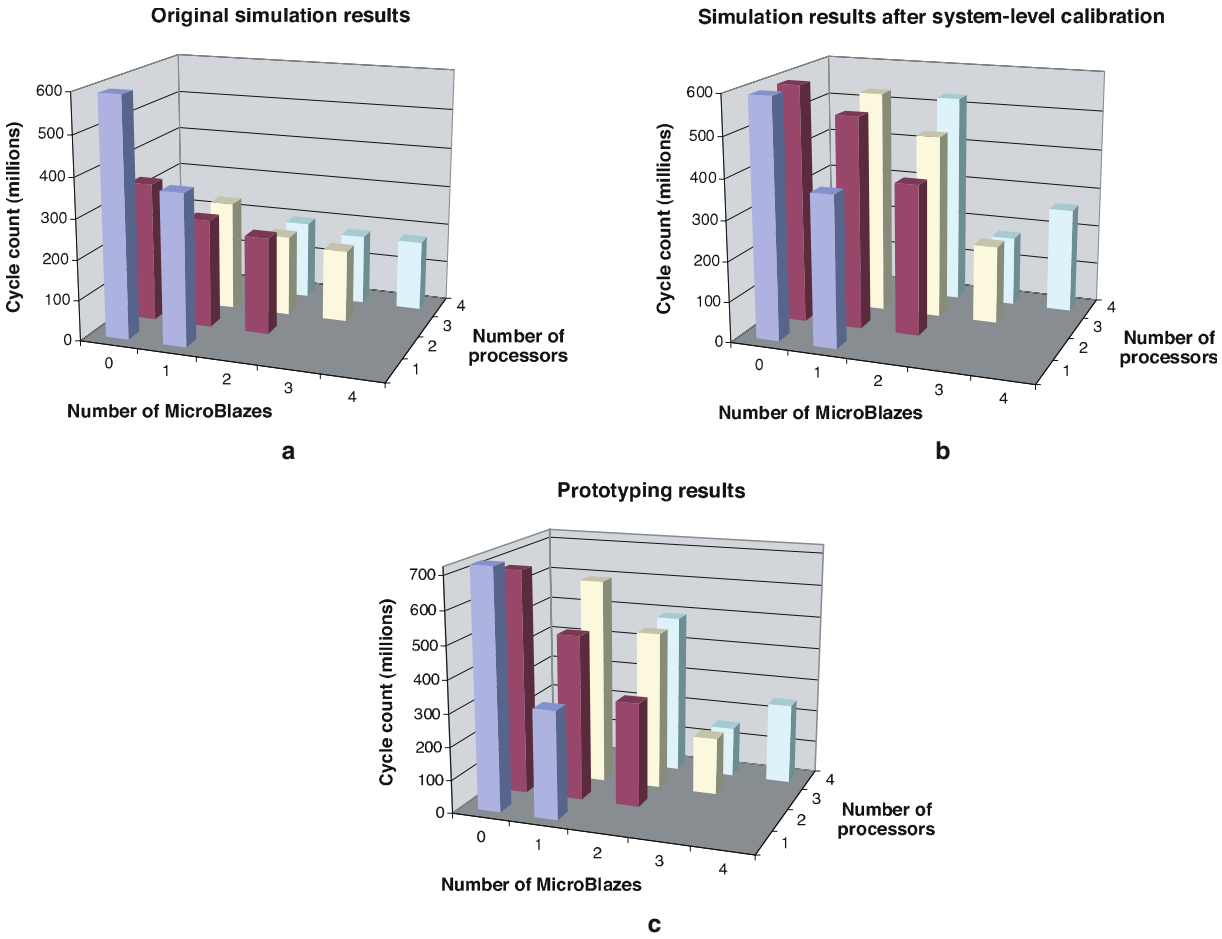

Figure 8. System-level model calibration: **a** Original simulation results, **b** simulation results after system-level calibration, and **c** prototype results from ESPAM implementations.

calibration approach. It is unclear, however, whether or not vertical simulation has ever been realized. In Brunel et al. [10], a high-level communication model is discussed which is calibrated using a cycle-true simulator.

The *back annotation* technique is closely related to model calibration. In back annotation, performance latencies measured by a low-level simulator are back annotated in a higher-level model. For example, an un-timed behavioral model could be back annotated such that it tracks timing information for a specific implementation. So, rather than calibrating a fixed set of performance model parameters, back annotation *adds* architecture-specific timing behavior (usually by means of code instrumentation) to a higher-level model. Back annotation is a widely-used technique for (high-level) performance modeling of software [17]. In the context of system-level modeling, various research efforts (e.g., [5, 11, 12]) also refer to back annotation as a technique for adding more detailed timing information to higher-level models in the case lower-level models are available. But these efforts generally do not provide much insight of how back annotation is applied during the early stages of design where lower-level models typically are not abundant. In a way, our calibration methods can be considered as a form of back annotating the latency tables in Sesame's architecture models using results from ISS simulation and/or automated component synthesis.

Table 3. Errors (in %) between original simulation and prototyping results (a) and between calibrated simulation and prototyping results (b).

	Number of MicroBlazes				
# Proc.	0	1	2	3	4
a					
1	18.2	15.2			
2	49.2	46.2	23.8		
3		56.5	58.2	3.7	
4			60.1	14.7	28.3
Average error=34%					
b					
1	18.2	15.2			
2	12.9	5.5	19.0		
3		11.3	4.5	12.6	
4			7.7	14.7	7.5
Average error=11.7%					

Related to our on-line calibration technique, much 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., [1, 2, 3, 5, 8, 16, 19]). Such frameworks typically combine behavioral models, ISSs, bus-functional models or HDL models into a single co-simulation. These mixed-level co-simulations generally need to solve two important problems: (1) making the co-simulation functionally correct by translating any differences in data and control granularity between simulation components, and (2) 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 [15] or a centralized simulation backbone using e.g. SystemC [8, 16]. Synchronization between simulation components usually takes place with 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 [23], especially when synchronizing at cycle granularity.

In the mixed-level co-simulation that results from our on-line (trace-)calibration technique, we take 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 shown in Thompson et al. [36], the performance overhead caused by wrappers and time synchronizations is in that case reduced to a minimum. Our on-line calibration technique shows some similarities with the trace-driven co-simulation technique in Kim et al. [23]. However, the latter operates at a lower abstraction level and is applied in a classical HW/SW co-simulation context.

As mentioned in our introduction, there are various related architectural exploration environments (e.g., [7, 13, 24, 27, 35]) that, like Sesame, also facilitate flexible system-level performance evaluation by providing support for mapping a behavioral application specification to an architecture specification. In Gries [18], an excellent survey is presented of various methods, tools and environments for early design space exploration. Compared to most related efforts, Sesame tries to push the separation of modeling application behavior and modeling architectural constraints at the system level to even greater extents. Doing so, it aims at optimizing the potentials for model re-use during the exploration cycle.

7. Conclusions

High-level performance modeling and simulation has become a key component in system-level design. Although many promising system-level modeling and simulation frameworks have been proposed, the aspects of model validation and calibration have not yet been widely addressed in this domain. This article presented the mechanisms currently available to our Sesame simulation framework for the calibration of its system-level performance models. These mechanisms can be classified into ISS-based calibration for calibrating programmable model components, and synthesis-based calibration (exploiting an external synthesis tool-flow) for calibrating dedicated model components as well as system-level models

as a whole. To show the merits of model calibration, we also presented two illustrative case studies with a M-JPEG encoder application. These studies indicate that model calibration is a crucial ingredient of system-level performance analysis. Currently, we are performing additional case studies to further evaluate our model calibration mechanisms. Also, we intend to incorporate more types of lower-level models for model calibration in Sesame. These also include, for example, low(er)-level models for the calibration of Sesame's high-level interconnection network models.

Acknowledgements

The Compaan/Laura and ESPAM tool-flows have been developed at Leiden University. Main contributors of these tools-flows are Alexandru Turjan, Bart Kienhuis, Edwin Rijpkema, Todor Stefanov, Claudiu Zissulescu, Hristo Nikolov and Ed Deprettere. The Molen platform has been designed and developed at Delft University of Technology. The Molen contributors we would like to mention here, are Stephan Wong, Georgi Kuzmanov, Georgi Gaydadjiev and Stamatis Vassiliadis. We would like to give special credits to Todor Stefanov, Hristo Nikolov, and Georgi Kuzmanov for their work on mapping the M-JPEG application onto the Molen and ESPAM-based platform architectures for the purpose of model calibration. This research is supported by PROGRESS, the embedded systems research program of the Dutch organization for Scientific Research NWO, the Dutch Ministry of Economic Affairs and the Technology Foundation STW.

Notes

1. In the Kahn paradigm, processes communicate via unbounded FIFO channels. Reading from these channels is done in a blocking manner, while writing is non-blocking.
2. We use SimpleScalar's detailed micro-architectural `sim-outorder` simulator.
3. The actual mapping of M-JPEG onto the Molen reconfigurable platform, using the Compaan and Laura tool-set, was done by colleagues of ours at Leiden University and Delft University of Technology. See the Credits section.
4. Note that results are only depicted for platforms with up to two PowerPCs. This is because the ESPAM-based synthesis trajectory is restricted to these platform due to the specific Xilinx Virtex-II-Pro FPGA chip it currently uses for prototyping.
5. This is only a matter of plugging in a different "policy model component."

References

1. CoWare, "ConvergenSC," http://www.coware.com/, 2007.
2. Mentor Graphics, "Seamless," http://www.mentor.com/, 2007.
3. Synopsys, "System Studio," http://www.synopsys.com/, 2007.
4. T. Austin, E. Larson, and D. Ernst, "Simplescalar: An Infrastructure for Computer System Modeling," *IEEE Comput.*, vol. 35, no. 2, 2002, pp. 59–67.
5. A. Baghdadi, N.-E. Zergainoh, W. Cesario, T. Roudier, and A.A. Jerraya, "Design Space Exploration for Hardware/Software Codesign of Multiprocessor Systems," in *IEEE International Workshop on Rapid System Prototyping*, 2000, pp. 8–13.
6. F. Balarin, E. Sentovich, M. Chiodo, P. Giusto, H. Hsieh, B. Tabbara, A. Jurecska, L. Lavagno, C. Passerone, K. Suzuki, and A. Sangiovanni-Vincentelli, *Hardware-Software Co-Design of Embedded Systems-The POLIS Approach*, Kluwer, 1997.
7. F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli, "Metropolis: An Integrated Electronic System Design Environment," *IEEE Comput.*, vol. 36, no. 4, 2003, pp. 45–52.
8. L. Benini, D. Bertozzi, D. Bruni, N. Drago, F. Fummi, and M. Poncino, "SystemC Cosimulation and Emulation of Multiprocessor SoC Designs," *IEEE Comput.*, vol. 36, no. 4, 2003, pp. 53–59.
9. B. Black and J.P. Shen, "Calibration of Microprocessor Performance Models," *IEEE Comput.*, vol. 31, no. 5, 1998, pp. 59–65.
10. J.-Y. Brunel, W.M. Kruijtzer, H.J. Kenter, F. Pétrot, L. Pasquier, E.A. de Kock, and W.J.M. Smits, "COSY Communication IP's," in *Proc. ACM/IEEE Design Automation Conference*, 2000, pp. 406–409.
11. L. Cai, and D. Gajski, "Transaction Level Modeling: An Overview," in *Proc. Int. Conference on HW/SW Codesign and System Synthesis (CODES-ISSS)*, 2003, pp. 19–24.
12. J. Calvez, D. Heller, and O. Pasquier, "Uninterpreted Co-Simulation for Performance Evaluation of hw/sw Systems," in *Proc. Int. Workshop on Hardware-Software Co-Design*, 1996, pp. 132–139.
13. A. Cassidy, J. Paul, and D. Thomas, "Layered, Multi-Threaded, High-Level Performance Design," in *Proc. Design, Automation and Test in Europe (DATE)*, 2003.
14. J.E. Coffland, and A.D. Pimentel, "A Software Framework for Efficient System-Level Performance Evaluation of Embedded Systems," in *Proc. ACM Symp. on Applied Computing (SAC)*, 2003, pp. 666–671, http://sesamesim.sourceforge.net.
15. R.M. Fujimoto, "Parallel Discrete Event Simulation," *Commun. ACM*, vol. 33, no. 10, 1990, pp. 30–53.
16. P. Gerin, S. Yoo, G. Nicolescu, and A.A. Jerraya, "Scalable and Flexible Cosimulation of SoC Designs with Heterogeneous Multi-Processor Target Architectures," in *Proc. Int. Conference on Asia South Pacific Design Automation*, 2001, pp. 63–68.
17. P. Giusto, G. Martin, and E. Harcourt, "Reliable Estimation of Execution Time of Embedded Software," in *Proc. Conference on Design, Automation and Test in Europe (DATE)*, 2001, pp. 580–589.
18. M. Gries, "Methods for Evaluating and Covering the Design Space During Early Design Development," *Integr. VSLI J.*, vol. 38, no. 2, 2004, pp. 131–183.
19. K. Hines and G. Borriello, "Dynamic Communication Models in Embedded System Co-Simulation," in *Proc. Design Automation Conference*, 1997, pp. 395–400.

20. G. Kahn, "The Semantics of a Simple Language for Parallel Programming," in *Proc. IFIP Congress 74*, 1974.
21. K. Keutzer, S. Malik, A. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli, "System Level Design: Orthogonalization of Concerns and Platform-Based Design," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 19, no. 12, 2000.
22. B. Kienhuis, E.F. Deprettere, K.A. Vissers, and P. van der Wolf, "An Approach for Quantitative Analysis of Application-Specific Dataflow Architectures," in *Proc. IEEE Int. Conference on Application-Specific Systems, Architectures and Processors*, 1997.
23. D. Kim, Y. Yi, and S. Ha, "Trace-Driven hw/sw Cosimulation Using Virtual Synchronization Technique," in *Proc. Design Automation Conference*, 2005.
24. T. Kogel, A. Wieferink, R. Leupers, G. Ascheid, H. Meyr, D. Bussaglia, and M. Ariyamparambath, "Virtual Architecture Mapping: A SystemC Based Methodology for Architectural Exploration of System-on-Chip Designs," in *Proc. Int. workshop on Systems, Architectures, Modeling and Simulation (SAMOS)*, 2003, pp. 138–148.
25. E.A. Lee and T.M. Parks, "Dataflow Process Networks," *Proc. IEEE*, vol. 83, no. 5, 1995, pp. 773–801.
26. V. Mathur and V. Prasanna, "A Hierarchical Simulation Framework for Application Development on System-on-Chip Architectures," in *IEEE Intl. ASIC/SOC Conference*, 2001.
27. S. Mohanty and V.K. Prasanna, "Rapid System-Level Performance Evaluation and Optimization for Application Mapping onto SoC Architectures," in *Proc. IEEE International ASIC/SOC Conference*, 2002.
28. M. Moudgill, J.-D. Wellman, and J.H. Moreno, "Environment for PowerPC Microarchitecture Exploration," *IEEE MICRO*, vol. 19, no. 3, 1999, pp. 15–25.
29. H. Nikolov, T. Stefanov, and E. Deprettere, "Multi-Processor System Design with ESPAM," in *Proc. Int. Conf. on HW/SW Codesign and System Synthesis (CODES-ISSS' 06)*, 2006, pp. 211–216.
30. A.D. Pimentel and C. Erbas, "An IDF-Based Trace Transformation Method for Communication Refinement," in *Proc. ACM/IEEE Design Automation Conference*, 2003, pp. 402–407.
31. A.D. Pimentel, C. Erbas, and S. Polstra, "A Systematic Approach to Exploring Embedded System Architectures at Multiple Abstraction Levels," *IEEE Trans. Comput.*, vol. 55, no. 2, 2006, pp. 99–112.
32. A.D. Pimentel, M. Thompson, S. Polstra, and C. Erbas, "On the Calibration of Abstract Performance Models for System-Level Design Space Exploration," in *Proc. Int. Conference on Embedded Computer Systems: Architectures, MOdeling, and Simulation (IC-SAMOS 2006)*, 2006, pp. 71–77.
33. T. Stefanov, B. Kienhuis, and E.F. Deprettere, "Algorithmic Transformation Techniques for Efficient Exploration of Alternative Application Instances," in *Proc. Int. Symposium on Hardware/Software Codesign (CODES)*, 2002, pp. 7–12.
34. T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E.F. Deprettere, "System Design Using Kahn Process Networks: The Compaan/Laura Approach," in *Proc. Int. Conference on Design, Automation and Test in Europe (DATE)*, 2004, pp. 340–345.
35. T. Kangas et al., "UML-Based Multi-Processor SoC Design Framework," *ACM Trans. Embed. Comput. Syst.*, vol. 5, no. 2, 2006, pp. 281–320.
36. M. Thompson, A.D. Pimentel, S. Polstra, and C. Erbas, "A Mixed-Level Co-Simulation Method for System-Level Design Space Exploration," in *Proc. IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia'06)*, 2006, pp. 27–32.
37. A. Turjan, B. Kienhuis, and E.F. Deprettere, "Translating Affine Nested Loop Programs to Process Networks," in *Proc. Int. Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, 2004.
38. S. Vassiliadis, S. Wong, G.N. Gaydadjiev, K. Bertels, G. Kuzmanov, and E.M. Panainte, "The Molen Polymorphic Processor," *IEEE Trans. Comput.*, vol. 53, no. 11, 2004, pp. 1363–1375.
39. S. Verdoolaege, H. Nikolov, and T. Stefanov, "Improved Derivation of Process Networks," in *Proc. Int. Workshop on Optimization for DSP and Embedded Systems*, 2006.
40. C. Zissulescu, T. Stefanov, B. Kienhuis, and E.F. Deprettere, "LAURA: Leiden Architecture Research and Exploration Tool," in *Proc. 13th Int. Conference on Field Programmable Logic and Applications (FPL)*, 2003.

Andy D. Pimentel received the M.Sc. and Ph.D. degrees in Computer Science from the University of Amsterdam, where he currently is an Assistant Professor in the Department of Computer Science. He is a member of the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC). His research interests include computer architecture, computer architecture modeling and simulation, system-level design, design space exploration, performance analysis, embedded systems, and parallel computing. He is a senior member of the IEEE and a member of the IEEE Computer Society.

Mark Thompson received the M.Sc. degree in Computer Science from the University of Amsterdam. Currently, he is a Ph.D. student in the Systems Architecture Group at the University

of Amsterdam. His research interests include methods and tools for high-level simulation, modeling and performance analysis of heterogeneous embedded systems and design space exploration.

Simon Polstra received his M.Sc. degree in Computer Science from the University of Amsterdam. Currently, he is a member of the Computer Systems Architecture group of the University of Amsterdam. He has previously been active as an Engineer at the National Aerospace Lab in the Netherlands. His research interests include computer architecture and abstract system modeling.

Cagkan Erbas received the B.Sc. degree in Electrical Engineering from Middle East Technical University, the M.Sc. degree in Computer Engineering from Ege University, and a Ph.D. degree in Computer Science from the University of Amsterdam. He is currently a Postdoc in Computer Science at the University of Amsterdam. His research interests include embedded systems, hardware/software codesign, multiobjective search algorithms and meta-heuristics. He is a student member of the IEEE.