On the 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
Email: {andy,mthompsn,spolstra,cagkan}@science.uva.nl

***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 paper presents an initial attempt to provide support for calibrating various model components of a system-level performance model. We discuss these model calibration mechanisms in the context of our Sesame system-level simulation framework. An illustrative case study will also be presented to indicate the merits of model calibration.**

I. INTRODUCTION

The increasing complexity of modern embedded systems has led to the emergence of system-level design [1]. 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 (Metro)Polis [2], [3], MESH [4], Milan [5], Sesame [6], [7], and various SystemC-based environments (e.g., [8]). 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 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 paper, we address the calibration of system-level performance models. More specifically, we discuss model calibration in the context of our Sesame simulation framework [6], [7]. 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 initial attempt to provide 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. We will also illustrate a case of model component calibration using a Motion-JPEG encoder application.

The remainder of this paper is organized as follows. In the next section, we briefly introduce the Sesame system-level simulation framework. Section III provides an overview of the mechanisms used for calibrating Sesame's architecture model

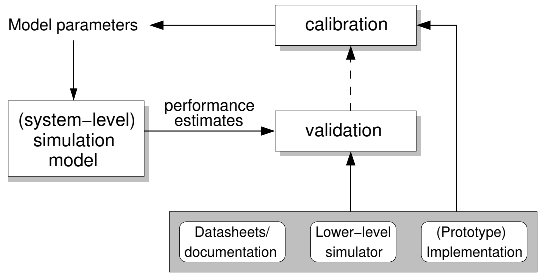
Model parameters
calibration
performance estimates
(system−level) simulation model
validation
Datasheets/ documentation
Lower−level simulator
(Prototype) Implementation

Fig. 1. Validation and calibration of architectural simulation models.

components. In Section IV, we present an experiment with a Motion-JPEG encoder illustrating model calibration, and also provide some validation results. Section V discusses related work. Finally, Section VI concludes the paper and discusses future work.

II. The Sesame environment

The Sesame modeling and simulation environment [6], [7] facilitates performance analysis of embedded (media) systems architectures according to the Y-chart design approach [9], [2]. 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 Figure 2.

For application modeling, Sesame uses the Kahn Process Network (KPN) model of computation [10]. This implies that applications are structured as a network of *concurrent communicating*[1] *processes*, 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 a tool called Compaan [11], [12] (which will be discussed later on).

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

[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.

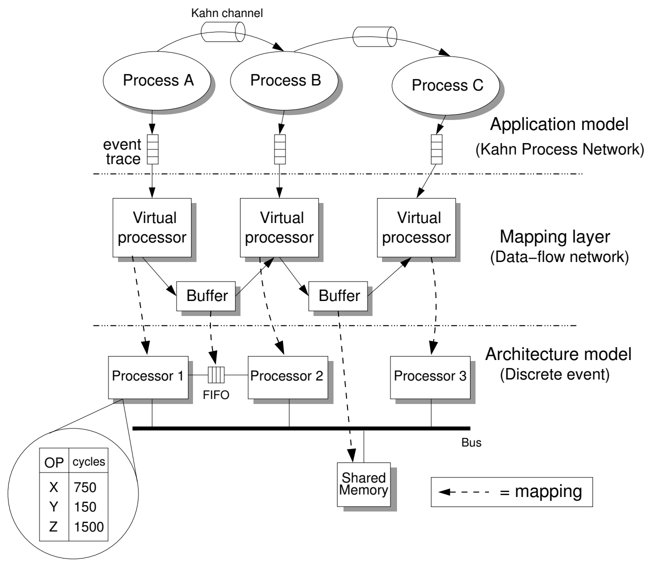

Fig. 2. Sesame's application model layer, architecture model layer, and mapping layer which interfaces between application and architecture models.

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 Figure 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 [6]. 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 [7]. 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

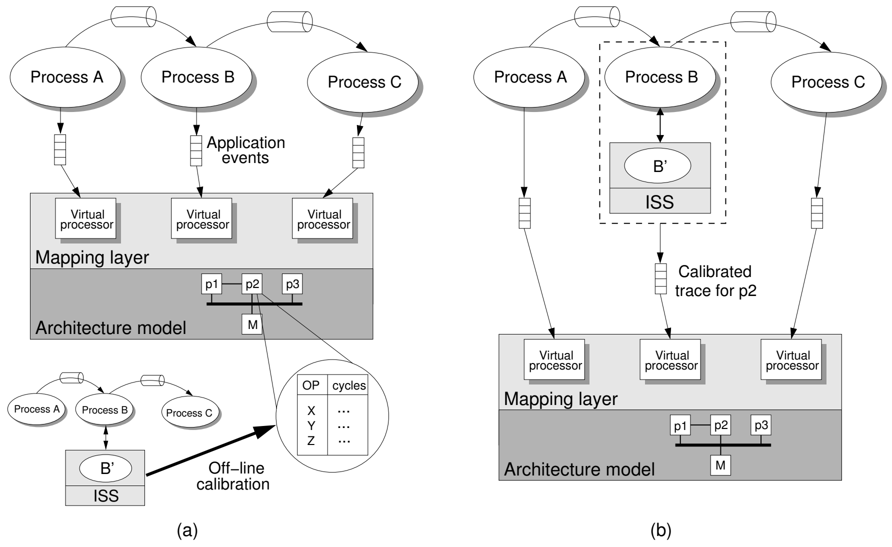

Fig. 3. Calibration of programmable model components. (a) Off-line calibration, and (b) on-line calibration (also referred to as trace calibration).

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 [13], [7]. For a more detailed overview of Sesame, we refer the reader to [7].

III. 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:

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

Calibration of a system-level architecture model as a whole, i.e. answering the second question, is often not feasible since a more detailed reference system model/implementation is usually not available during the early stages of design. Therefore, we focus on the first question in this paper. Or, in other words, in the remainder of this section we will address a number of mechanisms 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 paper.

A. 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 Figure 3, which is based on a more abstract representation of Figure 2. Currently, Sesame only supports the SimpleScalar ISS[2] [14] for calibration purposes, but other ISSs could also be added with relative ease.

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 Figure 3(a). In this example, we assume that model component *p2* – onto which application process B is mapped (see Figure 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 Figure 3(a)). 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 the

[2]We use SimpleScalar's detailed micro-architectural `sim-outorder` simulator.

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 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[3]. 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*).

In on-line model calibration, which is illustrated in Figure 3(b), 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 [15]. Rather than using fixed values in the latency tables of model components, on-line calibration dynamically computes the latencies of computational tasks using the ISS. In the example of Figure 3(b), 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 fixed computational execution events, like *Execute(DCT)*, application process B now generates *Execute(Δ)* events, where Δ equals to the actual measured number of cycles taken by, for example, a DCT computation (or any other computation in between communications). More details about the co-simulation technique behind on-line calibration, which also allows for easy and transparent distributed simulation of the different simulators that take part in the co-simulation, can be found in [15].

B. 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 [11], [12] and Laura [16], [12] 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 VHDL code that implements the application specified by the KPN for a specific FPGA platform.

[3]Actually, the mechanism allows to entirely discard the computational functionality in Kahn process B as this is also simulated by process B' on the ISS. In that case, Kahn process B only performs the data exchanges.

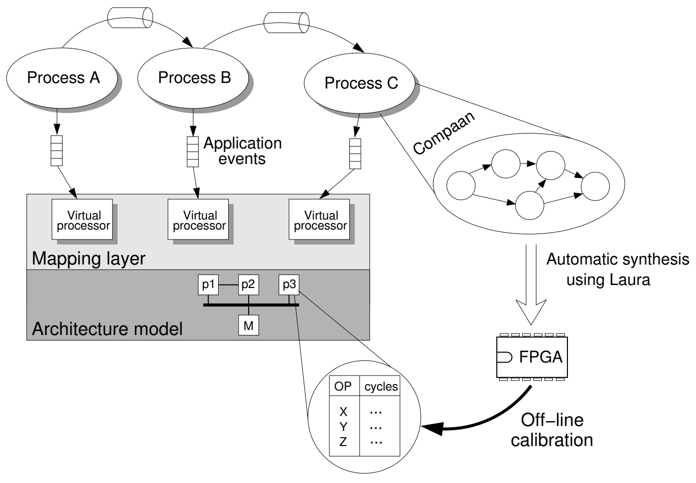

Fig. 4. Off-line calibration of dedicated model components.

Figure 4 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 [17], 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 the next section, one of the reconfigurable platforms that Laura uses as a mapping target is the Molen platform [18].

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 the next section, this will be illustrated using a case study with a Motion-JPEG encoder application.

IV. Experiments

In this section, we present an experiment that illustrates how model calibration can be performed using the Compaan/Laura tool-set [11], [12], [16]. To this end, we modeled a Motion-JPEG (M-JPEG) encoder application and selected the DCT task from this application to be used for model calibration.

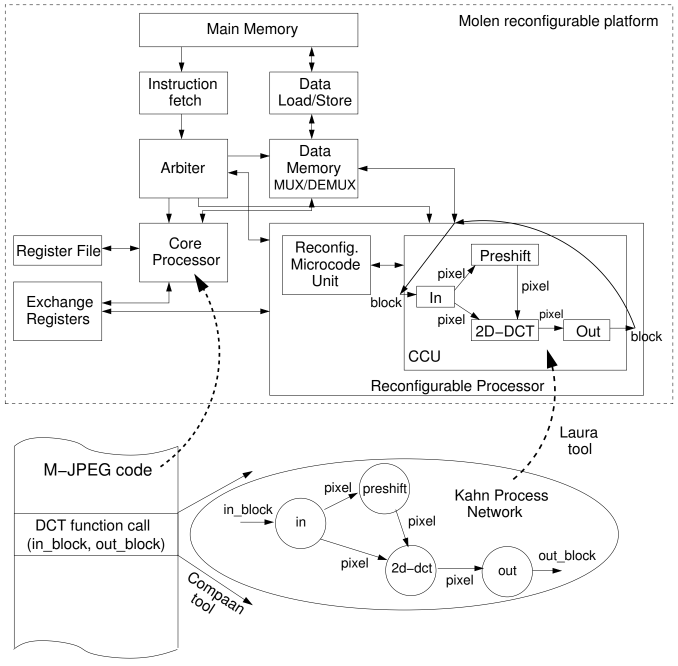

Fig. 5. Model calibration for the DCT task using the Compaan/Laura tool-set and the Molen reconfigurable platform.

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[4], which are integrally shown in Figure 5. 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 [18]. The Molen platform connects a programmable processor (depicted as Core Processor in Figure 5) 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.

[4]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.

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 [13], [7] – 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/Molen 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 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.

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 I shows the validation results for a sequence of sample input frames. The results from Table I 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

TABLE I
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

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 Figure 1). The results from Table I clearly indicate that Sesame's system-level performance estimations are, with the help of model calibration, quite accurate.

V. 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., [19], [20]). 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 [5] proposes a so-called vertical simulation approach that shows some similarities with our calibration approach. It is unclear, however, whether or not vertical simulation has ever been realized. In [21], 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 [22]. In the context of system-level modeling, various research efforts (e.g., [23], [24], [25]) 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 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.

VI. 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 paper presented the mechanisms currently available in 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. To show the merits of model calibration, we also presented an illustrative case study with a Motion-JPEG encoder application. Currently, we are performing additional case studies, which include ISS-based calibration, to 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.

Credits

The Compaan and Laura tool-set has been developed at Leiden University. Main contributors of this tools-set are Alexandru Turjan, Bart Kienhuis, Edwin Rijpkema, Todor Stefanov, Claudiu Zissulescu, 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 and Georgi Kuzmanov for their work on mapping the M-JPEG application onto Molen 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.

References

[1] K. Keutzer, S. Malik, A. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli, "System level design: Orthogonalization of concerns and platform-based design," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 12, Dec. 2000.

[2] 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 Academic Publishers, 1997.

[3] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli, "Metropolis: An integrated electronic system design environment," *IEEE Computer*, vol. 36, no. 4, April 2003.

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

[5] V. Mathur and V. Prasanna, "A hierarchical simulation framework for application development on system-on-chip architectures," in *IEEE Intl. ASIC/SOC Conference*, Sept. 2001.

[6] J. E. Coffland and A. D. Pimentel, "A software framework for efficient system-level performance evaluation of embedded systems," in *Proc. of the ACM Symp. on Applied Computing (SAC)*, March 2003, pp. 666–671, `http://sesamesim.sourceforge.net/`.

[7] A. D. Pimentel, C. Erbas, and S. Polstra, "A systematic approach to exploring embedded system architectures at multiple abstraction levels," *IEEE Transactions on Computers*, vol. 55, no. 2, pp. 99–112, 2006.

[8] 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. of the Int. workshop on Systems, Architectures, Modeling and Simulation (SAMOS)*, 2003, pp. 138–148.

[9] 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. of the IEEE Int. Conference on Application-specific Systems, Architectures and Processors*, July 1997.

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

[11] A. Turjan, B. Kienhuis, and E. F. Deprettere, “Translating affine nested loop programs to process networks,” in *Proc. of the Int. Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, Sept. 2004.

[12] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. F. Deprettere, “System design using Kahn process networks: The Compaan/Laura approach,” in *Proc. of the Int. Conference on Design, Automation and Test in Europe (DATE)*, Feb. 2004, pp. 340–345.

[13] A. D. Pimentel and C. Erbas, “An IDF-based trace transformation method for communication refinement,” in *Proc. of the ACM/IEEE Design Automation Conference*, June 2003, pp. 402–407.

[14] T. Austin, E. Larson, and D. Ernst, “Simplescalar: An infrastructure for computer system modeling,” *IEEE Computer*, vol. 35, no. 2, pp. 59 – 67, Feb. 2002.

[15] A. D. Pimentel, M. Thompson, S. Polstra, and C. Erbas, “Trace calibration: A mixed-level co-simulation technique for system-level design space exploration,” Unpublished, submitted for publication, see also `http://www.science.uva.nl/~andy/trace_cal.pdf`.

[16] 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)*, Sept. 2003.

[17] T. Stefanov, B. Kienhuis, and E. F. Deprettere, “Algorithmic transformation techniques for efficient exploration of alternative application instances,” in *Proc. of the Int. Symposium on Hardware/Software Codesign (CODES)*, May 2002, pp. 7–12.

[18] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, “The Molen polymorphic processor,” *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1363–1375, Nov. 2004.

[19] B. Black and J. P. Shen, “Calibration of microprocessor performance models,” *IEEE Computer*, vol. 31, no. 5, pp. 59–65, 1998.

[20] M. Moudgill, J.-D. Wellman, and J. H. Moreno, “Environment for PowerPC microarchitecture exploration,” *IEEE Micro*, vol. 19, no. 3, pp. 15–25, 1999.

[21] 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. of the ACM/IEEE Design Automation Conference*, 2000, pp. 406–409.

[22] P. Giusto, G. Martin, and E. Harcourt, “Reliable estimation of execution time of embedded software,” in *Proc. of the conference on Design, Automation and Test in Europe (DATE)*, 2001, pp. 580–589.

[23] J. Calvez, D. Heller, and O. Pasquier, “Uninterpreted co-simulation for performance evaluation of hw/sw systems,” in *Proc. of the Int. Workshop on Hardware-Software Co-Design*, 1996, pp. 132–139.

[24] 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.

[25] L. Cai and D. Gajski, “Transaction level modeling: An overview,” in *Proc. of the Int. Conference on HW/SW Codesign and System Synthesis (CODES-ISSS)*, Oct. 2003, pp. 19–24.