Signature-based calibration of analytical system-level performance models

Jaddoe, S.; Pimentel, A.D.

DOI
10.1007/978-3-540-70550-5_30
Publication date
2008
Document Version
Author accepted manuscript
Published in
Embedded Computer Systems: Architectures, Modeling, and Simulation

Link to publication

Citation for published version (APA):
Jaddoe, S., & Pimentel, A. D. (2008). Signature-based calibration of analytical system-level performance models. In M. Bereković, N. Dimopoulos, & S. Wong (Eds.), *Embedded Computer Systems: Architectures, Modeling, and Simulation: 8th International Workshop, SAMOS 2008, Samos, Greece, July 21-24, 2008 : proceedings* (pp. 268-278). (Lecture Notes in Computer Science; Vol. 5114). Springer. https://doi.org/10.1007/978-3-540-70550-5_30

Signature-Based Calibration of Analytical System-Level Performance Models

Stanley Jaddoe and Andy D. Pimentel

Computer Systems Architecture group
Informatics Institute, University of Amsterdam, The Netherlands
{vjaddoe,andy}@science.uva.nl

Abstract. The Sesame system-level simulation framework targets efficient design space exploration of embedded multimedia systems. Even despite Sesame's efficiency, it would fail to explore large parts of the design space simply because system-level simulation is too slow for this. Therefore, Sesame uses analytical performance models to provide steering to the system-level simulation, guiding it toward promising system architectures and thus pruning the design space. In this paper, we present a mechanism to calibrate these analytical models with the aim to deliver trustworthy estimates. Moreover, we also present some initial evaluation results with respect to the accuracy of our calibration mechanism using a case study with a Motion-JPEG encoder.

1 Introduction

The increasing complexity of modern embedded systems, which are more and more based on (heterogeneous) MultiProcessor-SoC (MP-SoC) architectures, has led to the emergence of system-level design. 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 early stages of design to perform, for example, architectural Design Space Exploration (DSE). Such early DSE is of eminent importance as early design choices heavily influence the success or failure of the final product.

With our Sesame modeling and simulation framework [1,2], we target 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. 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.

Although Sesame's system-level simulation allows for efficiently evaluating different application/architecture combinations, it would fail to explore large parts – let alone the entire span – of the design space. This is because system-level simulation is simply too slow for comprehensively exploring the design space, which is at its largest during the early stages of design. For this reason, Sesame uses analytical models [3,4] to provide steering to the system-level simulation, guiding it toward promising system architectures and therefore allowing for *pruning* the design space. These analytical models, which include models for performance, power and cost estimation, are used for quickly searching the design space by means of multi-objective optimization using evolutionary algorithms. So far, this analytical modeling stage lacked a systematic method for deriving the model parameters that specify application requirements and architecture capabilities. Clearly, the accuracy of these analytical models is highly dependent on the correct determination of these parameters.

In this paper, we focus on the performance estimation part of our analytical models (i.e. the power and cost models are not addressed) and present a technique based on execution profiles, referred to as *signatures*, that allows for deriving the application and architecture specific parameters in these analytical performance models. Using a preliminary experiment with a Motion-JPEG encoder application and an MP-SoC architecture, we also show initial results of the accuracy of our approach by comparing the estimations of our signature-based analytical model with those from simulation.

The remainder of the paper is organized as follows. In the next section, we introduce the basic analytical system model [3,4] for which we want to derive the model parameters. Section 3 describes how we determine application specific model parameters via a profiling mechanism based on signatures. Section 4 describes how architecture specific parameters are derived using a comparable mechanism. In Section 5, we put together the pieces of the puzzle presented in Sections 3 and 4 to actually construct signature-based analytical performance models. Section 6 presents initial results of the evaluation of the accuracy of our approach using an experiment with a Motion-JPEG encoder application. Section 7 describes related work, and Section 8 concludes the paper.

2 Basic Analytical System Model

In the Sesame framework, applications are modeled using the Kahn Process Network (KPN) [5] model of computation in which parallel processes communicate with each other via unbounded FIFO channels. By executing the application model, each Kahn process records its actions in order to generate its own *trace of application events* which is necessary for driving an architecture model. There are three types of application events, divided in two groups: *execute* events for computational behavior and *read* and *write* events for communication behavior.

The architecture models in Sesame simulate the performance consequences of the computation and communication events generated by an application model. Architecture models are constructed from building blocks provided by a library containing template models for processing cores, and various types of memories and interconnects.

Since Sesame makes a distinction between application and architecture models, it needs an explicit mapping step to relate these models for co-simulation. In this step, the designer decides for each application process and FIFO channel a destination architecture model component to simulate its workload. This is an important step in the design process, since the final success of the design can be highly dependent on these mapping choices. To decide on an optimum mapping, many instances need to be considered (and thus simulated). In realistic cases, in which the underlying architecture is also varied during the process of design space exploration, simulation of all points in the design space is infeasible. Therefore, analytical models are needed to prune the design space, steering the designer towards a small set of promising design points which then can be simulated. The remainder of this section elaborates on the basic analytical performance model [3,4] we use in Sesame for design space pruning, after which the subsequent sections present our signature-based mechanism to 'calibrate' this analytical model.

The application models in Sesame are represented by a graph $KPN = (V_K, E_K)$ where the set V_K and E_K refer to the Kahn processes and the directed FIFO channels between these processes, respectively. For each process $a \in V_K$, we define $B_a \subseteq E_K$ to be the set of FIFO channels connected to process a, $B_a = \{b_{a1}, \ldots, b_{an}\}$. For each Kahn process, we define a computation requirement, shown with α_a, representing the computational workload imposed by that Kahn process onto a particular component in the architecture model. The communication requirement of a Kahn process is not defined explicitly, rather it is derived from the channels attached to it. We have chosen this type of definition for the following reason: if the Kahn process and one of its channels are mapped onto the same architecture component, the communication overhead experienced by the Kahn process due to that specific channel is simply neglected. For the communication workload imposed by the Kahn process, only those channels that are mapped onto different architecture components are taken into account. So our model neglects internal communications and only considers external communications. Formally, we denote the communication requirement of the channel b with β_b. To include memory latencies into our model, we require that mapping a channel onto a specific memory asks computation tasks from the memory. To express this, we define the computational requirement of the channel b from the memory as α_b. Here, it is ensured that the parameters β_b and α_b are only taken into account when the channel b is mapped onto an external memory. The actual determination of the above model parameters, which is the contribution of this paper, will be addressed in the next section.

Similarly to the application model, the architecture model is also represented by a graph $ARC = (V_A, E_A)$ where the sets V_A and E_A denote the architecture components and the connections between the architecture components, respectively. In our model, the set of architecture components consists of two disjoint subsets: the set of processors (P) and the set of memories (M), $V_A = P \cup M$ and $P \cap M = \emptyset$. For each processor $p \in P$, the set $M_p = \{m_{p1}, \ldots, m_{pj}\}$ represents the memories which are reachable from the processor p. We define processing capabilities for both the processors and the memories as c_p and c_m, respectively. These parameters need to be set such that they reflect processing capabilities for processors, and memory access latencies for memories. The determination of these parameters will be addressed in Section 4.

The above model needs to adhere to a number of constraints, such as that each Kahn process has to be mapped to a processor, each channel has to be mapped to a processor (in case of local communication) or memory, and so on. For a formal description of these constraints, we refer to [3,4].

3 Application Requirements

As indicated in the previous section, we need to determine the model parameters for application requirements (α_a, α_b and β_b) and architecture capabilities (c_p and c_m). To this end, we present an approach based on execution profiles of application events, referred to as *signatures*, to determine these model parameters. In the remainder of this section, we focus on the derivation of the model parameters – via these signatures – for application requirements. As will become clear, our approach strictly adheres to the separation of concerns concept [6], separating application (requirements) from architecture (capabilities) signatures.

A signature of a Kahn process represents its computational requirements. These process signatures describe the computational complexity at a high level of abstraction using an Abstract Instruction Set (AIS). Currently, our AIS consists of the small set of abstract instruction types as shown in Table 1(a)[1]. To construct a signature, the real machine instructions that embody the computation, derived from an Instruction Set Simulator (ISS), are first mapped onto the AIS, after which a compact execution profile is made. This means that the resulting signature is a vector containing the instruction counts of the different AIS instructions. The first column in Table 1(a) shows the signature (vector) index that each AIS instruction type corresponds to.

To illustrate the process of determining the process signatures, consider Table 1(b) which shows an example event trace of Kahn process k_1. When deriving the signature of process k_1, only the *execute* events in its event trace are considered. Each *execute* event comes with an identifier of an operation, to indicate which operation was executed. The signature of k_1 is the sum of the signatures of the operations executed by k_1. In the example of Table 1(b), operations op_1 and op_2 have signatures that describe the computational requirements of these operations. Now, assume that an ISS generates the trace of (in this case, ARM) instructions as shown in the first column of Table 1(c) for op_1. The next step is to classify these instructions (is it a basic integer instruction, or a memory operation, or a branch instruction, etc.). In other words, the assembly instructions have to be mapped to the AIS instructions defined for our signatures. The result of this classification is shown in the second column of Table 1(c). Then, a signature for op_1 can be generated based on the counts of the AIS opcodes. For op_1, this gives

$$op_1.\text{signature} = [3,15,1,0,3,9,0,0] \tag{1}$$

with the AIS counts ranked according to the first column of Table 1(a). Using the same method, a signature for op_2 can be generated. Assume that its signature is:

$$op_2.\text{signature} = [8,17,8,0,2,29,2,0] \tag{2}$$

Then, using these signatures we can answer the original question, that is, calculate the signature of process k_1 (i.e., α_{k_1}). According to the event trace of process k_1, op_1 was executed two times, op_2 one time. Thus,

$$k_1.\text{signature} = 2op_1.\text{signature} + op_2.\text{signature} = [14,47,10,0,8,47,2,0] \tag{3}$$

An important thing to note is that in practice, if an operation is executed more than once, the derived signatures for each execution of the operation may not be equal (due to data dependencies, or pseudo-random behaviour of the operation). In that case, the operation's signature becomes the average signature of all executions of that operation.

[1] In this paper, we focus on programmable cores as processor targets, but the AIS also consists of a special "co-processor" instruction that can be used for modeling dedicated HW blocks.

Table 1. Table (a) shows the currently defined AIS instructions with their index in the vector-based process signatures. Table (b) lists the event trace of process k_1, and Table (c) shows an execution trace of op_1 as obtained by an ARM ISS (left column) and the corresponding AIS instructions (right column).

Signature index	AIS opcode	Description
1	`AIS_BMEM`	Block memory transfers
2	`AIS_MEM`	Memory transfers
3	`AIS_BRANCH`	Branches
4	`AIS_COPROC`	Co-proc. instructions
5	`AIS_IMUL`	Int. multiplications
6	`AIS_ISIMPLE`	Simple Int. arithmetic
7	`AIS_OS`	Software interrupts
8	`AIS_UNKNOWN`	Non-mappable instruction

(a)

`read`	f_2
`execute`	op_1
`write`	f_1
`read`	f_2
`execute`	op_2
`write`	f_1
`execute`	op_1
`write`	f_1
`write`	f_1

(b)

ARM instruction	AIS opcode
bl 0x81c4;	`AIS_BRANCH`
mov ip, **sp**;	`AIS_ISIMPLE`
stmdb sp, fp, ip, lr, pc;!	`AIS_BMEM`
sub fp, **ip**, #4;	`AIS_ISIMPLE`
sub sp, **sp**, #12;	`AIS_ISIMPLE`
ldr r2, [**fp**, #−16];	`AIS_MEM`
ldr r3, [**fp**, #−20];	`AIS_MEM`
add r2, r2, r3;	`AIS_ISIMPLE`
ldr r3, [**fp**, #−24];	`AIS_MEM`
rsb r3, r3, r2;	`AIS_ISIMPLE`
str r3, [**fp**, #−24];	`AIS_MEM`
ldr r2, [**fp**, #−16];	`AIS_MEM`
ldr r3, [**fp**, #−20];	`AIS_MEM`
add r2, r2, r3;	`AIS_ISIMPLE`
ldr r3, [**fp**, #−24];	`AIS_MEM`
mul r3, r2, r3;	`AIS_IMUL`
str r3, [**fp**, #−16];	`AIS_MEM`
ldr r2, [**fp**, #−20];	`AIS_MEM`
ldr r3, [**fp**, #−16];	`AIS_MEM`
mul r3, r2, r3;	`AIS_IMUL`
str r3, [**fp**, #−24];	`AIS_MEM`
ldr r2, [**fp**, #−16];	`AIS_MEM`
ldr r3, [**fp**, #−24];	`AIS_MEM`
add r2, r2, r3;	`AIS_ISIMPLE`
ldr r3, [**fp**, #−20];	`AIS_MEM`
mul r3, r2, r3;	`AIS_IMUL`
str r3, [**fp**, #−16];	`AIS_MEM`
sub sp, **fp**, #12;	`AIS_ISIMPLE`
ldmia sp, {**fp**, **sp**, **pc**};	`AIS_BMEM`
mov ip, **sp**;	`AIS_ISIMPLE`
stmdb sp, fp, ip, lr, pc;!	`AIS_BMEM`

(c)

A signature of a FIFO channel describes the load induced by the channel on memory components (i.e., α_b and β_b from Section 2). This communication requirement of a FIFO channel depends on the size of the token (in bytes) sent via the channel, and the total number of tokens sent. In our application models, the size of the tokens sent via a FIFO channel is fixed. The number of tokens sent via a FIFO channel can be extracted from the Kahn process' event trace. Each *write*-event in an event trace contains data about to which communication port the token was sent. So, the signature of a FIFO channel f is a two-element vector containing the number of tokens sent via the channel and the size of each token:

$$f.\text{signature} = [n_{tokens}, n_{size}] \tag{4}$$

For example, assume the event trace of process k_1 in Table 1(b) and a token size for channel f_1 of $n_{size} = 12$ bytes. Since process k_1 writes four times a token of 12 bytes to f_1 (see Table 1(b)), the signature of f_1 thus becomes:

$$f_1.\text{signature} = [4, 12] \tag{5}$$

4 Architectural Capabilities

Previously, the computational and communication *requirements* of an application have been defined. In this section, the computational and communication *capabilities* of processors and memories will be defined. These capabilities will also be encoded as (vector-based) signatures.

If a Kahn process k_1 is mapped onto a processor p_1, then the number of cycles p_1 is busy processing k_1 (denoted as $\mathcal{T}(p_1)$) can be calculated as a function of the signatures of k_1 (the computational requirements) and p_1 (the processor capabilities):

$$\mathcal{T}(p_1) = f(k_1.\text{signature}, p_1.\text{signature}) \tag{6}$$

The aim is to find or define both p_1.signature and the function f in (6). With these, we can calculate the number of cycles a processor is busy processing the *execute* events emitted by Kahn processes mapped onto the processor.

Using an ISS, we can measure how many cycles a certain operation takes when executed on a specific processor (like an ARM). If this is repeated for many operations, a *training set* can be built. Using this training set, the computational capabilities of a processor (i.e., its signature) can be derived by, for example, linear regression, or techniques used in the field of machine learning.

Using the example from the previous section, a (very small) training set can be made. This training set consists of the signatures of op_1 and op_2 and the associated cycle counts. Let us assume that executing op_1 took 185 cycles, and that op_2 took 369 cycles when executed on an ARM processor. Since a training set consists of a list of vectors (operation signatures), and a list of cycle counts, this problem can be solved using the least-squares method. For example, let SM be the matrix with the signatures of operations op_1 and op_2 as rows, p_1.signature be the weight vector we want to calculate for processor p_1, and c be the vector with cycle counts for each row in SM. Then, $SM \cdot p_1.\text{signature} = c$ is solved using the least squares method.

$$\begin{pmatrix} 3 & 15 & 1 & 0 & 3 & 9 & 0 & 0 \\ 8 & 17 & 8 & 0 & 2 & 29 & 2 & 0 \end{pmatrix} \cdot p_1.\text{signature} = \begin{pmatrix} 185 \\ 369 \end{pmatrix} \tag{7}$$

The signature of p_1 is the the vector consisting of weights for each AIS instruction. The unit of the elements in the vector is ‘cycles per instruction’. Note that these weights can be adapted in order to perform high-level architectural design space exploration for the given processor (e.g., make multiplications more/less expensive, etc.).

$$p_1.\text{signature} = [2.19, 7.11, 1.62, 0.0, 1.19, 7.4, 0.33, 0.0] \tag{8}$$

Given an operation signature s that is not included in the training set, the estimated number of cycles on p_1 for that signature is simply the inner product of s and p_1.signature.

The signature (and thus the communication capability) of a memory component (i.e., c_m) is a two-element vector $[r_{read}, r_{write}]$ that only consists of the (average) read and write latencies. So far, in contrast to processor signatures, we have not developed any methods to get reliable memory signatures. Instead, a designer may use values from memory data sheets to create a memory signature.

5 Analytical Performance Estimation

In the previous sections, portions of a (signature-based) analytical performance model were presented. In this section, these portions will be forged together to get an analytical performance model for an architecture.

```
$\mathcal{T}^c(p) \leftarrow 0$
**foreach** $k \in X_p$ **do**
    **foreach** $f \in \mathit{FIFOChannels}_{k,\mathrm{ext}}$ **do**
        $b \leftarrow f.\mathrm{signature}[n_{tokens}] \cdot f.\mathrm{signature}[n_{size}]$
        $m \leftarrow \mathcal{M}(f)$
        **if** *f is an incoming channel of k* **then**
            $\mathcal{T}^c(p) \leftarrow \mathcal{T}^c(p) + b/m.\mathrm{signature}[r_{read}]$
        **end**
        **if** *f is an outgoing channel of k* **then**
            $\mathcal{T}^c(p) \leftarrow \mathcal{T}^c(p) + b/m.\mathrm{signature}[r_{write}]$
        **end**
    **end**
**end**
```

Algorithm 1. Calculation of $\mathcal{T}^c(p)$

First, some definitions have to be made. The set X_p is the set of processes that are mapped onto processor p. A similar definition applies to X_m, the set of channels mapped onto memory m. $\mathcal{M}(f)$ denotes the memory onto which channel f is mapped and $\mathit{FIFOChannels}_{k,\mathrm{ext}}$ is the set of channels of process k that are mapped onto an external memory.

The time $\mathcal{T}^e(p)$ a processor p is spending on executing operations is the inner product of the sum of the signatures of all processes mapped on p, with the signature of p.

$$\mathcal{T}^e(p) = \left\langle \left(\sum_{k \in X_p} k.\mathrm{signature} \right), p.\mathrm{signature} \right\rangle \tag{9}$$

The time $\mathcal{T}^c(p)$ the processor is communicating depends on the number of bytes sent and received via FIFO channels that are mapped on an external memory. This quantity can be calculated by Algorithm 1.

The total time processor p is busy processing *read*, *write*, and *execute* events is

$$\mathcal{T}(p) = \mathcal{T}^e(p) + \mathcal{T}^c(p) \tag{10}$$

The number of cycles $\mathcal{T}(m)$ a memory m is busy sending or receiving data is calculated in Algorithm 2, in a similar way as $\mathcal{T}^c(p)$.

The maximum processing time of an architecture with a certain mapping depends on the architecture component with the largest processing time. Therefore, we need to solve

$$\min\max \left(\max_{p \in P} \mathcal{T}(p), \max_{m \in M} \mathcal{T}(m) \right) \tag{11}$$

6 Experimental Results

In this section, mapping exploration results of the signature-based analytic method will be compared to simulation results using a Motion-JPEG (M-JPEG) encoder application. The target MP-SoC architecture we used in this experiment consists of four ARM processors with local memory and a crossbar interconnect. The design space we considered for this experiment consists of all possible mappings of the M-JPEG tasks (i.e. processes) on the processors in the MP-SoC platform.

```
$b \leftarrow 0$
foreach $f \in X_m$ do
    $b \leftarrow b + f.\text{signature}[n_{tokens}] \cdot f.\text{signature}[n_{size}]$
end
$\mathcal{T}(m) \leftarrow b/m.\text{signature}[r_{read}] + b/m.\text{signature}[r_{write}]$
```

Algorithm 2. Calculation of $\mathcal{T}(m)$

Before the M-JPEG application model was mapped on the architecture model, the application was compiled using an ARM C++ compiler, and executed within the SimIt-ARM instruction set simulator environment [7]. The generated ARM instruction traces were used to create the application and architecture signatures. These signatures were subsequently used for determining the parameters in our analytical performance model, as was previously explained. Note that this process is only a one-time effort.

Since the design space in our experiment is relatively limited (consisting of 4096 different mappings), it was possible to evaluate all of these mappings, both analytically as well as by simulation using our Sesame framework. The analytical and simulation results are shown in Figure 1. Note that only the first fifty mappings are depicted due to space limitations (to avoid cluttering in the graph). Each mapping gets a certain index. The order of the mappings in Figure 1 is more or less arbitrary. Mappings with successive indices are not necessarily related to each other. In this experiment, we measured an average relative error of our analytical model compared to simulation of only 0.1%, with a standard deviation of 0.2. From this, it can be concluded that the performance estimates of our analytic method are promising since they show small errors with respect to the simulation-based estimates.

It should be noted however that this is only a preliminary evaluation, using some simplified assumptions and circumstances: we obtained the signatures by training with the application itself, and the application used in this case study is still a fairly static, pipeline-based application of which the workload is well suited for prediction. Also, the application does not cause any contention on the interconnect. In an additional experiment, we artificially generated excessive network contention for the M-JPEG application. As a result, the error increased to an average of 14% with a standard deviation of 26. But since in this case the analytical estimates were optimistic and still showed the correct performance trends, we believe that these results are still very promising in the scope of high-level design space pruning (the pruning does not throw away possible good candidate mappings). We also stress that the evaluation time of our analytical performance models is several orders of magnitude smaller as compared to Sesame's system-level simulations.

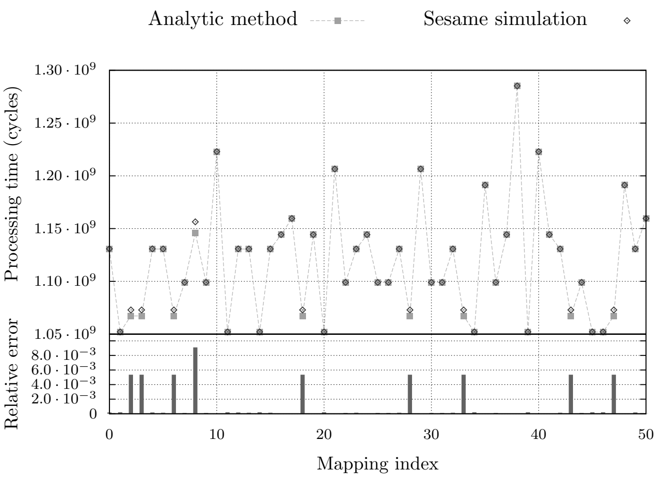

Fig. 1. Comparison between simulation and analytical methods of M-JPEG mappings on a crossbar-based multiprocessor architecture

7 Related Work

Much work has been performed in the area of software performance estimation [8], including methods that use profiling information, typically gathered at the instruction level. For example, in [9] a static software performance estimation technique is presented which uses profiling at the instruction level and which includes the modeling of pipeline hazards in the timing model. In [10], a source-based estimation technique is proposed using the concept of ”virtual instructions”. These are similar (albeit a bit more low level) to our AIS instructions, but which are directly generated by a compiler framework. Software performance is then calculated based on the accumulation of the performance estimates of these virtual instructions. The idea of convolving application and machine signatures, where the signatures contain coarse-grained system-level information, has also been applied in the domain of performance prediction for high-performance computer systems [11]. In [12], a workload modeling approach based on execution profiles is discussed for statistical micro-architectural simulation. Because they address micro-architectural simulation, their profiles include much more details (such as pipeline and cache behavior), while we address the system level at a higher level of abstraction. In [13], the authors suggest to derive a linear model from a small set of simulations. This method tries to model the performance of a processor at a mesoscopic level. For example, cache behaviour and pipeline characteristics are taken into account. The significance of all cache and pipeline related parameters is determined by simulation-based linear regression models. This may be comparable with the ‘weight’ vector discussed in Section 4. Another interesting approach is presented in [14], in which the CPI for in-order architectures is predicted using a Monte Carlo based model.

8 Conclusions

In this paper, we presented a technique for calibrating our analytical performance models used for system-level design space pruning. More specifically, we introduced the concept of application and architecture signatures, which can be related with each other to obtain performance estimates. Using a preliminary case study with a Motion-JPEG encoder application, we showed that our signature-based analytical performance model shows promising results with respect to accuracy. But since this application still is relatively static in its behavior, we need to extend our experiments in the future to also include more dynamic applications. Moreover, we need to further study the (off-line) generation of training sets for deriving processor signatures, as well as to investigate extending our signatures to better capture micro-architectural behavior.

References

1. Pimentel, A.D., Erbas, C., Polstra, S.: A systematic approach to exploring embedded system architectures at multiple abstraction levels. IEEE Trans. on Computers 55, 99–112 (2006)
2. Erbas, C., Pimentel, A.D., Thompson, M., Polstra, S.: A framework for system-level modeling and simulation of embedded systems architectures. EURASIP Journal on Embedded Systems (2007) doi:10.1155/2007/82123
3. Erbas, C., Cerav-Erbas, S., Pimentel, A.D.: A multiobjective optimization model for exploring multiprocessor mappings of process networks. In: Proc. of the int. conference on Hardware/Software Codesign & System Synthesis (CODES+ISSS), pp. 182–187 (2003)
4. Erbas, C., Cerav-Erbas, S., Pimentel, A.D.: Multiobjective optimization and evolutionary algorithms for the application mapping problem in multiprocessor system-on-chip design. IEEE Trans. on Evolutionary Computation 10, 358–374 (2006)
5. Kahn, G.: The semantics of a simple language for parallel programming. Information Processing 74, 471–475 (1974)
6. Keutzer, K., Malik, S., Newton, A., Rabaey, J., Sangiovanni-Vincentelli, A.: System level design: Orthogonalization of concerns and platform-based design. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems 19 (2000)
7. Qin, W., Malik, S.: Flexible and formal modeling of microprocessors with application to retargetable simulation. In: Design, Automation and Test in Europe (DATE) Conference, pp. 556–561 (2003)
8. Bammi, J.R., Harcoun, E., Kruijtzer, W., Lavagno, L., Lazarescu, M.: Software performance estimation strategies in a system level design tool. In: International Conference on Hardware Software Codesign (CODES), pp. 82–87 (2000)
9. Beltrame, G., Brandolese, C., Fornaciari, W., Salice, F., Sciuto, D., Trianni, V.: An assembly-level execution-time model for pipelined architectures. In: Proc. of Int. Conference on Computer Aided Design (ICCAD), pp. 195–200 (2001)
10. Giusto, P., Martin, G., Harcourt, E.: Reliable estimation of execution time of embedded software. In: Proc. of the Design, Automation, and Test in Europe (DATE) Conference, pp. 580–588 (2001)
11. Snavely, A., Carrington, L., Wolter, N.: Modeling application performance by convolving machine signatures with application profiles. In: Proc. of the IEEE Workshop on Workload Characterization, pp. 149–156 (2001)

12. Eeckhout, L., Nussbaum, S., Smith, J., De Bosschere, K.: Statistical simulation: adding efficiency to the computer designer's toolbox. IEEE Micro 23, 26–38 (2003)
13. Joseph, P., Vaswani, K., Thazhuthaveetil, M.: Construction and Use of Linear Regression Models for Processor Performance Analysis. In: Proc. of the Int. Symposium on High-Performance Computer Architecture, pp. 99–108 (2006)
14. Srinivasan, R., Cook, J., Lubeck, O.: Performance Modeling Using Monte Carlo Simulation. IEEE Computer Architecture Letters 5 (2006)