Signature-based Calibration of Analytical Performance Models for System-level Design Space Exploration

Stanley Jaddoe, Mark Thompson, and Andy D. Pimentel

Computer Systems Architecture group
Informatics Institute, University of Amsterdam, The Netherlands
{m.thompson,a.d.pimentel}@uva.nl

Abstract. The Sesame system-level simulation framework targets efficient design space exploration of embedded multimedia systems. Even despite Sesame's high efficiency, it would still 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 based on execution profiles, referred to as *signatures*, to calibrate these analytical models with the aim to deliver trustworthy estimates. Moreover, we also present a number of experiments in which we evaluate the accuracy of our signature-based performance models using a case study with a Motion-JPEG encoder and the Mediabench benchmark suite for performing off-line calibration of the models.

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, 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 simulations allow for evaluating different application/architecture combinations in a highly efficient fashion, it would typically fail to explore large parts – let alone the entire span – of the design space. This is because system-level simulation is still 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, which extends [5], 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 an experiment with a Motion-JPEG encoder application and an ARM-based target MP-SoC architecture, we validate the accuracy of our approach by comparing the estimations of our signature-based analytical model with those from simulation. Moreover, we also present an experiment in which we perform so-called *off-line training* of our signature-based performance model using the Mediabench benchmark suite [6], after which this externally calibrated model is again applied to the Motion-JPEG case study.

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 validation results of our approach using an experiment with a Motion-JPEG encoder application. Moreover, it also presents results from an experiment in which we study off-line training of our signature-based performance models using the Mediabench benchmark suite. 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) [7] model of computation. The use of KPNs is motivated by the fact that this model of computation nicely fits the targeted multimedia-processing application domain and is deterministic. The latter implies that the same application input always results in the same application output, irrespective of the scheduling of the KPN processes. This provides complete scheduling freedom when, as will be discussed later on, mapping KPN processes onto MP-SoC architecture models for quantitative performance analysis and design space exploration. In a KPN, 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

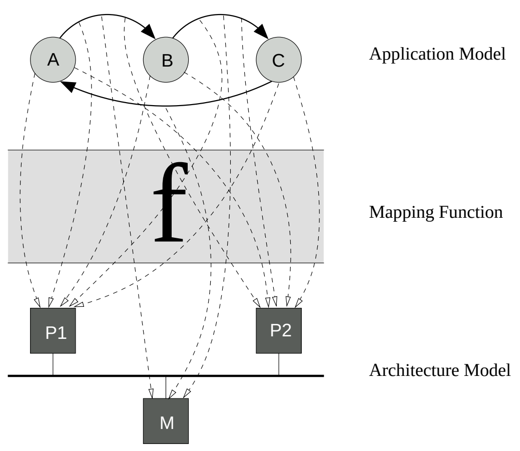

Fig. 1. An example mapping problem.

application events, divided in two groups: *execute* events for computational behavior and *read* and *write* events for communication behavior.

The architecture models in Sesame are cycle-approximate TLM models and 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. Here, Sesame provides support for modeling a variety of scheduling policies in case multiple application processes are mapped onto a single architectural processing element. Mapping applications onto the underlying architectural resources is an important step in the design process, since the final success of the design can be highly dependent on these mapping choices. In Figure 1, we illustrate this mapping step on a very simple example. In this example, the application model consists of three Kahn processes and FIFO channels. The architecture model contains two processors and one shared memory. To decide on an optimum mapping, many instances need to be considered (and thus simulated). In realistic cases, in which the underlying architecture can also be 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 provides an outline of 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 sets 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 (i.e., B_a). 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 (processor) 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 a memory component 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 application model parameters 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. The next section will address the derivation of the model parameters for architecture capabilities, which is done using the signature mechanism as well. As will become clear, our approach strictly adheres to the concept of ‘separation

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`

ARM instruction	AIS opcode
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)

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).

of concerns' [8], 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.

The high level of abstraction of the AIS makes it architecture independent and, as will become clear later on, makes the signatures relatively small and easy to handle.

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

Nevertheless, the AIS could always be refined when needed (see also Section 6). Given the fact that our AIS only consists of a few instruction opcodes, many different real machine instructions will thus map onto the same AIS instruction. For example, all (single-element) load and store instructions will map onto `AIS_MEM`, integer multiplications and divisions onto `AIS_MUL`, and basic integer operations such as additions, subtractions and logical operations onto `AIS_ISIMPLE`. The `AIS_UNKNOWN` opcode is used when a machine instruction cannot be mapped onto any of the other AIS instruction types. Our experiments have demonstrated, however, that the influence of this `AIS_UNKNOWN` class of AIS instructions is negligible.

The computational (i.e., *execute*) events in the application event traces from a KPN application model together with the separate signatures of each of the associated computational operations determine the signature for each Kahn process. To illustrate this, 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 sequence 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 different AIS opcodes. For op_1, this gives

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

with the AIS opcode 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.

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 one particular FIFO channel is always fixed (although the token size between channels can vary). 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 $\vec{S}$ 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 $\vec{c}$ be the vector with cycle counts for each row in $\vec{S}$. Then, $\vec{S} \cdot p_1.\text{signature} = \vec{c}$ 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 vector consisting of weights for each AIS opcode. 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. As will be elaborated in Section 6, this allows us to perform (off-line) training of our signature-based performance models. To derive processor signatures in this case, we use operation signatures from a training set of selected benchmark applications which is representative for our multimedia application domain. Hereafter, the computational performance of operation signatures from any (multimedia) application can be estimated using the above inner product.

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. Evidently, our current memory model fully abstracts away the underlying memory architecture. Also, in contrast to processor signatures, we have not yet 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 are forged together to get an analytical performance model for an MP-SoC architecture.

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 $\mathit{FIFOChannels}_{k,\text{ext}}$ is the set of channels of process k that are mapped onto an external memory. Thus, $\mathit{FIFOChannels}_{k,\text{ext}} \subseteq B_k$ (see Section 2).

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

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

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. This algorithm should be self-explanatory as it simply calculates $\mathcal{T}^c(p)$ by accumulating all access latencies of the memories used by processor p.

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 $\mathcal{T}^c(p)$. Here, for each byte stored in the memory we accumulate both its read and write latencies.

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

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

```
$b \leftarrow 0$
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 $\mathcal{T}(m)$.

The processing time of an architecture with a certain mapping depends on the architecture component with the largest processing time. Therefore, optimizing for performance (i.e., minimizing processing time) during design space exploration thus leads to solving

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

6 Experimental results

In this section, we present a validation experiment using a Motion-JPEG (M-JPEG) encoder application in which mapping exploration results from our signature-based analytic performance model are compared to simulation results. Furthermore, we also discuss an experiment in which we perform off-line training of the signature-based performance models using the Mediabench benchmark suite [6] as an external training set.

6.1 Initial validation

To validate our signature-based analytic performance model, we studied the mapping of a Motion-JPEG (M-JPEG) encoder application onto an MP-SoC architecture. This is illustrated in Figure 2. The target MP-SoC consists of four ARM processors with

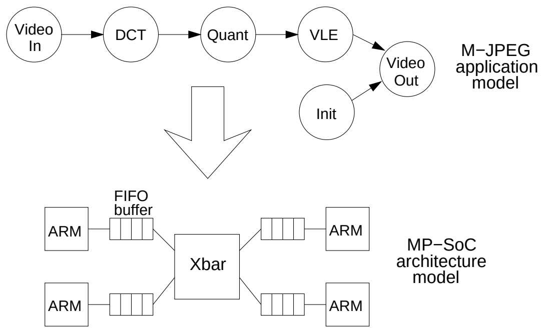

Fig. 2. Mapping an M-JPEG application to a crossbar-based MP-SoC architecture.

local memory, FIFO buffers for streaming data, 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.

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 [9]. 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 a one-time effort only.

Since the design space in our experiment is limited (consisting of 4096 mappings[2]), it was possible to quickly evaluate all of these mappings, both analytically as well as by simulation using our Sesame framework. In a first experiment, we applied the concept of signatures both in our analytical model *and* in the simulation model to which we compare. With respect to the latter, the processor components in our simulation model use the processor signatures to dynamically calculate computational latencies of incoming computational events (which are described using operation signatures) by means of the inner product as explained in Section 4. So, in this experiment we compare static (i.e. analytical) versus dynamic (i.e., simulative) performance estimation, both using signature-based models.

The analytical and simulation results are shown in Figure 3. Note that only the first fifty mappings are depicted due to space limitations (to avoid cluttering in the graph). Each mapping instance gets a certain index. The order of the mappings in Figure 3 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. Evidently, these small errors with respect to the simulation-based estimates are promising results.

Although the above validation is a good sanity check, comparing against a simulation model that uses the same signature-based performance approximation provides

[2] The number of unique mappings is even considerably less since the target platform is symmetrical.

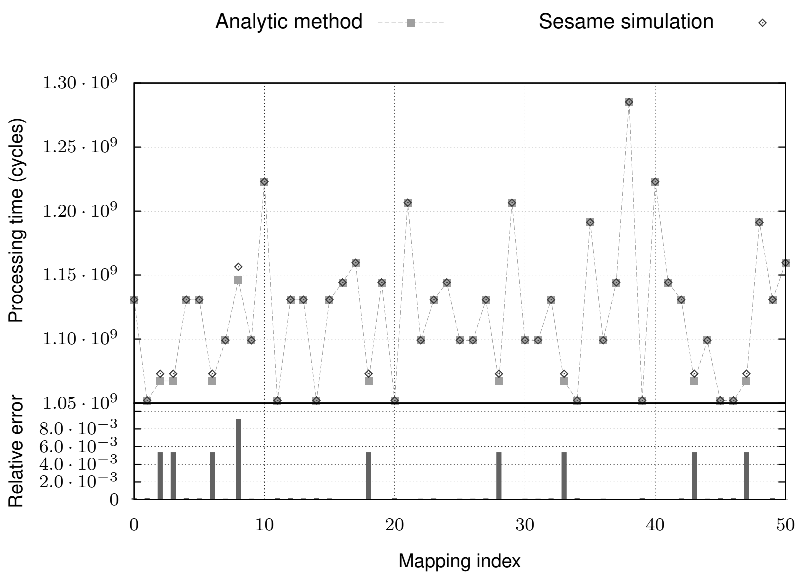

Fig. 3. Comparison between signature-based simulation and analytical models for M-JPEG mapping DSE on a crossbar-based multiprocessor architecture.

good preconditions for obtaining small errors. Therefore, in a next experiment, we compare the results from our signature-based analytical model to the results from a more accurate simulation model that uses 'exact' latencies for the various computational events as directly obtained by ISS measurements (so, no latencies that are obtained using linear regression). In the remainder of this paper, we refer to this simulation model as our *reference model*. Figure 4 shows a scatter plot of the DSE results for both the signature-based analytical model and the reference model. The graph only shows the performance results of *unique* mapping instances. Here, we have sorted all mapping instances based on the performance order of the mappings from the reference model.

The results from Figure 4 indicate that the estimates of our signature-based analytical model are fairly accurate, especially for the better-performing mappings (i.e. lower mapping indices in Figure 4). The overall average error is 9.2% with a standard deviation of 5.8. For the best 100 mappings, the average error only is 4.5% with a standard deviation of 2.1. But most importantly, our signature-based analytical model finds exactly the same optimal mappings as the reference model. Needless to say, this precision of finding the correct optimal mappings is of eminent importance to the process of design space pruning, helping the designer to decide which design points need to be studied in more detail using simulative methods. Another observation that can be made is that for the less optimal mappings (i.e., higher mapping indices), the error as well as the trend behavior deteriorate. This can be explained by the fact that in these

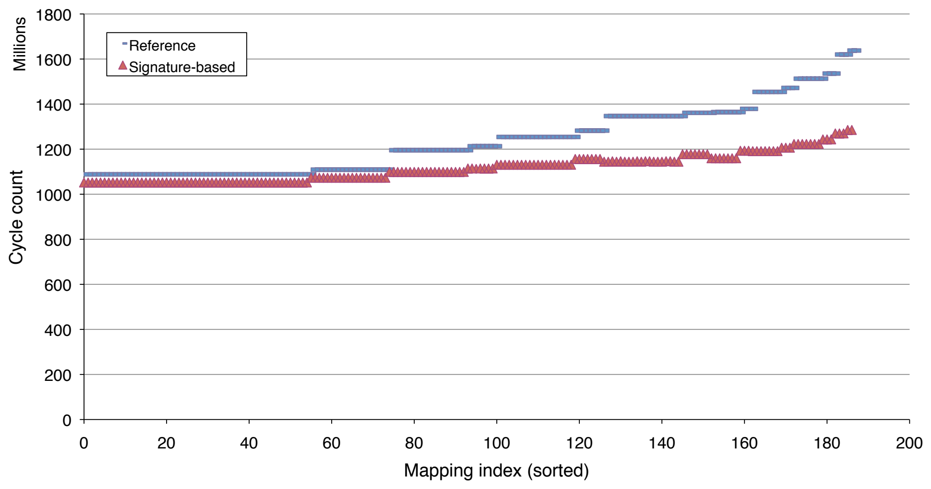

Fig. 4. Comparing the DSE results from our signature-based model against those from an ISS-calibrated reference simulation model.

less optimal mappings less concurrency is exploited, and therefore any errors in the signature-based latency calculations are more easily accumulated.

Besides accuracy, the efficiency of the performance evaluation is of great importance since the proposed performance models are targeted towards pruning large design spaces. Here, we would like to stress that the evaluation time of our signature-based analytical performance models is several orders of magnitude smaller as compared to Sesame's system-level simulations. This should allow us to study much larger design spaces, from which promising candidate designs can be selected that can then be further studied using system-level simulation.

6.2 Off-line training of the performance model

In the previous section, we used the same application (M-JPEG) for both training of the signature-based analytical performance model (i.e., obtaining processor signatures for ARM processors) and the performance estimation itself. Since this may be less realistic, we also present several results from an experiment in which we perform off-line training of the signature-based performance model using the Mediabench benchmark suite [6], after which we again apply this externally calibrated performance model to the M-JPEG case study.

To train our performance model, we first constructed the (application) operation signatures for each separate Mediabench program[3] using the approach as discussed in

[3] The `pgp`, `ghostscript`, and `sphere` benchmarks were excluded due to execution problems on the SimIt-ARM simulator.

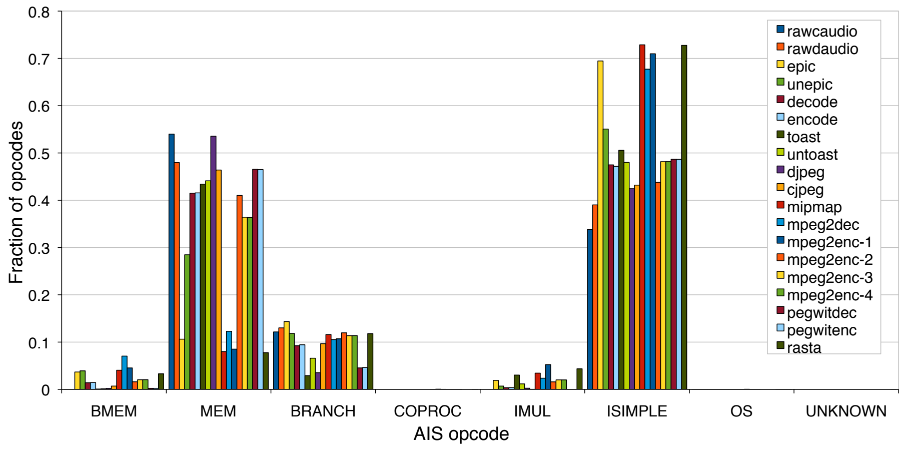

Fig. 5. Histogram with the various AIS opcode counts of the Mediabench training set.

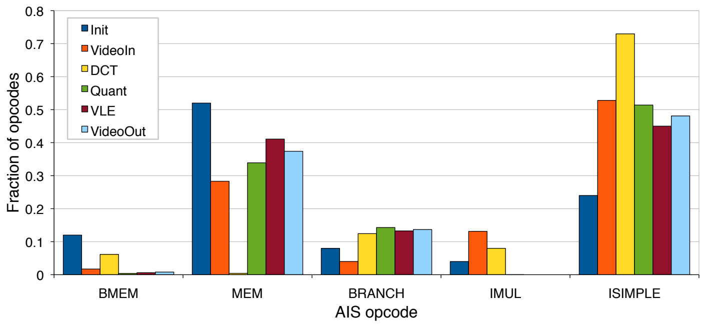

Fig. 6. Histogram with the various AIS opcode counts of M-JPEG.

Section 3. Since the execution time (in terms of simulated cycles) of the `mpeg2enc` program is quite high, we split up the execution of this program into four chunks, and generated a separate operation signature for each of these chunks (each representing different execution phases of the program). Figure 5 shows the histogram of the resulting AIS opcode counts for each Mediabench program. This graph clearly shows that most programs are dominated by `AIS_ISIMPLE` instructions, followed by `AIS_MEM` and `AIS_BRANCH` instructions respectively.

Similar to Figure 5, Figure 6 shows the AIS opcode histogram for the application processes in our target M-JPEG application. Here, we excluded the AIS opcodes with a zero or insignificant contribution. At first sight, the trends in both Figures 5 and 6

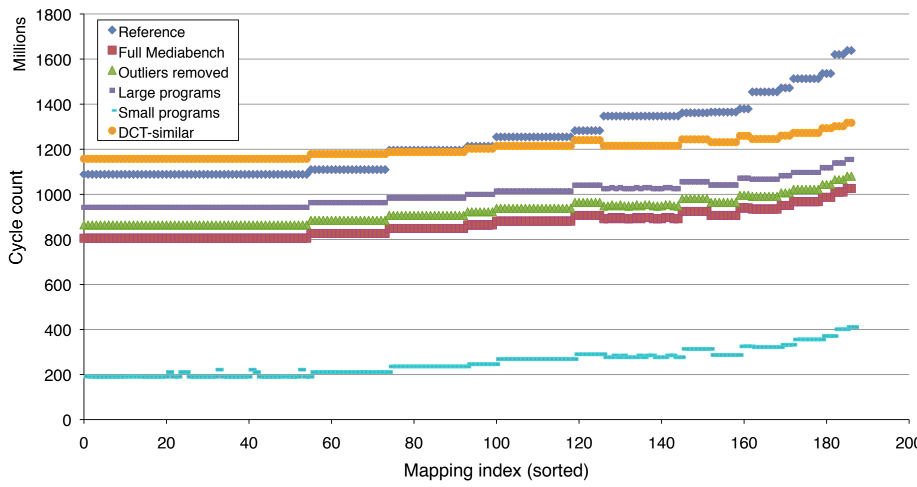

Fig. 7. Comparing the DSE results from our Mediabench-trained models against those from the reference model from Section 6.1.

are similar, which thus appears to be confirming that Mediabench is a representative training set for M-JPEG.

As a next step, we determined the processor signatures for our performance model using the Mediabench operation signatures. Using this Mediabench-trained performance model, we again performed the DSE experiment with the M-JPEG application. Figure 7 shows a comparison of the DSE results of the Mediabench-trained (”Full Mediabench” in the graph) and the reference simulation model from the previous section. Again, the graph only shows the performance results of unique mappings, and all mapping instances are sorted based on the performance order of the mappings from the reference model. Comparing the Mediabench-trained model (”Full Mediabench”) to the reference model, it is clear that there is a significant absolute error between the results of these models (an average error of 29.6%, see Table 2). But, the trends between the graphs of the two models still is highly similar. This is especially true for the better-performing mapping instances (lower mapping indices). This again implies that both models find exactly the same optimal mappings.

To study the sensitivity of the selected benchmark programs that are used for training, we performed a number of experiments in which we clustered the Mediabench programs according to some measure, after which we trained our performance model with the programs from such a cluster only. As a first experiment, we applied our performance model that was trained with all Mediabench programs to the operation signatures from the Mediabench programs themselves. Then, we clustered only those programs that show a good fit with the performance model (i.e., removing the outliers). Training the performance model again with this cluster, gives the results that are tagged with ”Outliers removed” in Figure 7 and Table 2. The results of this new performance model

	Full Mediabench	Outliers removed	Large programs	Small programs	DCT-similar programs
Av. error	29.6%	24.9%	18.6%	79.8%	7.0%
Std.dev.	3.8	4.4	5.1	2.1	4.5

Table 2. Average error and standard deviation of the various trained models as compared to the reference model.

are slightly better (average error of 24.9%, see Table 2) than the model that was trained with all Mediabench programs.

We selected the program size as the second means to cluster our Mediabench training set. Programs with more than 500 million executed instructions are clustered as ”Large programs”, while the remaining programs are clustered as ”Small programs”. Figure 7 again shows the DSE results when training our performance model with one of these clusters. The cluster with large programs again shows an accuracy improvement, lowering the average error to 18.6%. Clearly, the cluster with small programs only yields poor results, both in terms of average error (79.8%) and trend behavior. The latter can even be seen at the lower mapping indices where some optimal mappings (according to the reference model) are not considered optimal according to the model trained with small programs only.

Since the DCT process in the M-JPEG encoder is dominant in terms of computational intensity, our final clustering is based on similarity with the DCT process (in terms of AIS opcode distribution). We again trained our model with these DCT-similar programs. The DSE results of this cluster show again considerable improvement with an average error of only 7%, which is even slightly better than the result from the M-JPEG-trained model from the previous section. This can be explained by the fact that M-JPEG also contains processes that are dissimilar to the DCT process and which have a negative influence on the training of the model.

6.3 Discussion

The results from the previous section show that Mediabench can be a suitable training set for our signature-based performance models, provided that a proper cluster of Mediabench programs is selected for training. The ”Big programs” cluster already yields decent results whereas the ”DCT-similar” cluster shows that fairly accurate results are possible with off-line training. Noteworthy, however, is the fact that almost all training sets / clusters, except for the cluster with small Mediabench programs, yield the same optimal design points. This indicates that, at least for the studied M-JPEG application, signature-based performance modeling is a relatively robust technique for quickly exploring and pruning the vast mapping design space. However, more study is needed in several directions. For example, we need to investigate additional types of clustering of Mediabench, where e.g. the granularity of the code sequences used for generating operation signatures also plays a role. That is, so far we have used entire Mediabench programs (except for `mpeg2enc`) to generate operation signatures. But, like we did for `mpeg2enc`, we can also study the splitting of other large programs in smaller chunks (representing different execution phases of these programs) to have more control on the size of the training set, which could e.g. open up new possibilities for clustering.

Moreover, and naturally following from the above clustering discussion, we also need to investigate the refinement of our AIS. Since most machine instructions map onto only three AIS opcodes (`AIS_ISIMPLE`, `AIS_MEM`, and `AIS_BRANCH`), these opcodes can be refined to improve the accuracy of our analytical models. To further increase accuracy of our signature-based models, we should also study the extension of our signatures to better capture micro-architectural behavior (such as cache behavior), but still at a high level of abstraction. Of course, such extensions may again affect the possibilities for clustering of the training set (e.g., clustering on different cache behavior).

7 Related Work

Much work has been performed in the area of software performance estimation [10], including methods that use profiling information, typically gathered at the instruction level. For example, in [11] 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 [12], 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 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 [13].

In [14], a workload modeling approach based on execution profiles is discussed for statistical micro-architectural simulation. Because the authors address simulation at the micro-architectural level, their profiles include much more details (such as pipeline and cache behavior), while we address system-level modeling at a higher level of abstraction. In [15], 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 [16], in which the CPI for in-order architectures is predicted using a Monte Carlo based model. The Milan framework [17] deploys a design pruning approach using symbolic (instead of analytic) analysis methods to reduce the design space that needs to be explored with simulation.

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 case study with a Motion-JPEG encoder application, we showed that our signature-based analytical performance model shows promising results with respect to accuracy. Moreover, we presented a number of experiments in which we performed so-called off-line training of our signature-based performance model using the Mediabench benchmark suite. These experiments indicate that such off-line training is a promising mechanism for obtaining trustworthy estimation models in the scope of early design space pruning.

Since the Motion-JPEG application used in our study still is relatively static in its behavior, and thus fairly well-suited for prediction, we need to extend our experiments in the future to also include more dynamic applications. Moreover, we need to further study the off-line training of our models, also in relationship to possible refinement and/or extension of our signatures to better capture both application as well as microarchitectural 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** (2006) 99–112
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. Proc. of the int. conference on Hardware/-Software Codesign & System Synthesis (CODES+ISSS) (2003) 182–187
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** (2006) 358–374
5. Jaddoe, S., Pimentel, A.D.: Signature-based calibration of analytical system-level performance models. In: Proc. of the Int. Symposium on Systems, Architectures, MOdeling and Simulation (SAMOS '08). (2008) 268–278
6. Lee, C., Potkonjak, M., Mangione-Smith, W.H.: Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In: Proc. of the ACM/IEEE international symposium on Microarchitecture (Micro). (1997) 330–335
7. Kahn, G.: The semantics of a simple language for parallel programming. Information Processing **74** (1974) 471–475
8. 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)
9. Qin, W., Malik, S.: Flexible and formal modeling of microprocessors with application to retargetable simulation. Design, Automation and Test in Europe (DATE) Conference (2003) 556–561
10. 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). (2000) 82–87
11. 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). (2001) 195–200

12. 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. (2001) 580–588
13. 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. (2001) 149–156
14. Eeckhout, L., Nussbaum, S., Smith, J., De Bosschere, K.: Statistical simulation: adding efficiency to the computer designer's toolbox. IEEE Micro **23** (2003) 26–38
15. 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. (2006) 99–108
16. Srinivasan, R., Cook, J., Lubeck, O.: Performance Modeling Using Monte Carlo Simulation. IEEE Computer Architecture Letters **5** (2006)
17. Mohanty, S., Prasanna, V.K.: Rapid system-level performance evaluation and optimization for application mapping onto SoC architectures. In: Proc. of the IEEE Int. ASIC/SOC Conference. (2002)