Tools and techniques for efficient system-level design space exploration
Thompson, M.

Link to publication

Citation for published version (APA):
Thompson, M. (2012). Tools and techniques for efficient system-level design space exploration

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 4

Model calibration

4.1 Introduction

In the previous chapters we have seen how models are defined in Sesame. The behavior of the model as a whole is –of course– defined by all its component parts as well as by the interconnections of the components. The behavior of a model component is in turn specified by its simulation code and its interactions with other components. Additionally, the behavior may depend on application input data (such as video input streams), initialization of random seeds, the initialization of its variables, etc. Sesame helps the model designer with all modeling aspects, for example by offering separation of concerns, reusable model components, an easy-to-use simulation language, deterministic model execution, a graphical model editor, etc. In this way the designer can quickly and easily create functionally correct models for many different systems and applications. These models can then be quickly and easily evaluated thanks to good model run time performance and automatically generated evaluation reports.

However, functional correctness does not by itself mean that the model displays the correct non-functional behavior as well. Non-functional properties that are important to a system designer include system performance, power consumption, production cost as well as physical characteristics such as die size, thermal hot-spots, etc. For example, in Sesame, we can define a video encoding platform by trivially mapping the application onto an architecture model for which no realistic properties (eg. processor timings, clock speed, memory sizes) have been defined. In this case the (application) model will be functionally correct since it produces a correctly encoded video stream, but performance behavior of the architecture model will be unrealistic. Therefore, such a model is likely unusable to predict non-functional behavior of any realistic system. We say that a model is behaviorally correct if it is able to represent (with a certain level of accuracy) those non-functional properties that are of interest to the designer.

To determine the behavioral correctness of a model, one has to consider (among other things) the following. Firstly there is the domain of (non-functional) behavior that is important for the designer for example: performance, power, cost, physical characteristics, etc.
CHAPTER 4. MODEL CALIBRATION

Secondly, the designer may be interested only in the behavior of part of the system, such as network behavior or contention of processing or memory resources. Sesame supports this by offering mixed-level modeling where parts of the system that are of particular interest may be defined at a more detailed level of abstraction than the rest of the model. Choosing the right level of abstraction is the third main consideration that may influence behavioral correctness. The latter is particularly important, since it is well known that the level of detail used by a simulation model can have major impact on the run-time performance of the model.

With the above considerations in mind, the challenge of system-level design can be summarized as follows: to create a functionally and behaviorally correct model that accurately and quickly approximates the important parts of the system under evaluation.

When a system designer constructs a system-level model, then this model rarely displays the correct non-functional behavior from the beginning. Analogous to the traditional method of software debugging to achieve functional correctness of application software, there exists a need to tune the non-functional behavior of (parts of) the model in order to achieve behavioral correctness. This process is called model calibration, and it is one of the most difficult problems involved in creating a good system-level model (for a conceptual overview see Figure 4.1).

In this chapter we report on three methods to perform calibration of system-level simulation models, focusing on the performance objective. In Chapter 8 we will show an extensive case study using one of these techniques and there the simulation results will be validated against prototype system implementations in the context of the Daedalus design flow.

4.2 Model calibration

In Chapter 3 the Sesame modeling methodology was discussed in detail. The workload that is modeled by the application model is forwarded to the architecture model by means of event traces. The architecture model consumes these events and models their performance behavior as a delay (or latency) on the relevant model components. This may be done by simply associating a latency to that event, but the event may also trigger complex interactions between components. Using special primitives in the simulation language, it is easy to describe the (possibly complex) interaction between different components in the model. Moreover, the propagation of latencies as a consequence of component interaction is automatically performed by the simulation language’s runtime system. However, determining the correct latency value at any place in the model can pose quite a challenge depending

Figure 4.1: Modeling overview
4.2. MODEL CALIBRATION

on the type of model, the type of component and even the particular interest and requirements of the designer. In general, correct latency values can be obtained from many sources: component specification and documentation, instruction set simulators, measurements on prototypes, and even rough estimations may be sufficient in some cases. It may be necessary to recalibrate the latency values during the different stages of development of the system model as more details become known and more design decisions are taken. In the end, only validation of the model (e.g. by comparing the behavior with a (partially) implemented prototype) can provide certainty about the quality of the model. Therefore, calibration does not supersede, but rather supplements model validation, as a method to make the non-functional behavior of the model match the behavior of the actual system. Figure 4.2 schematically details the modeling process and the calibration and validation techniques.

![Figure 4.2: The calibration and validation process](image)

For some types of components latencies are relatively easily obtained because their behavior is known and predictable. Take for example an on-chip network component, such as a bus or crossbar: a performance model could assume some latency for setting up a new connection and subsequently add a certain latency per amount of communicated data. The latencies for the setup time and transfer rate are part of a component’s specification. Note once more that these latencies do not need to include any additional latency caused by contention of components (such as a shared bus), since this is will be automatically captured by the simulation runtime system (e.g. using Pearl’s synchronous method calls: Section 3.2.2).

Another possibility is that the designer is making a model that contains components which are yet to be developed. For example, the designer needs a component to process a certain task X, but still does not know whether X should be implemented in software or hardware. Creating a dedicated hardware component represents a significant development cost, but may speed-up the overall system performance. Since the designer will not be able to obtain accurate latencies for X, he could estimate a ball-park value based on his experience. Using the estimated value he can the use the system-level model to support such design choices in the very early design stages.

Latencies can be much more difficult to estimate for some other types of components.
E.g., programmable processor components associate a latency to any application execution events. In our initial, high-level system models such an execution event typically represents a large chunk of code. The exact latency to be associated with that event may vary according to data dependencies, but in our simplest and most abstract models, the latency for an execution event of a certain type is set to an average value. In this case, the calibration process is focused on finding usable average values.

In general, the design of MPSoC embedded systems heavily depends on the reuse of existing components and therefore it is often the case that data sheets, tools, and prototypes are available for at least parts of the system. Since all system-level models need calibration it is crucial to have generic and feasible methods to calibrate those models by finding sufficiently accurate latencies in a structural and easy way. In the next section we will present the off-line model calibration method, which does just that, by measuring these values using an Instruction Set Simulator (ISS). We present also the on-line trace calibration method, which is a form of mixed-level co-simulation which has the potential to provide more accurate latency values even when latencies are data-dependent.

A possible disadvantage of the off-line and on-line calibration methods is that the calibration is performed for a specific application. Therefore, in the second part of this chapter a method will be presented that attempts to associate a "signature" to an architecture component that describes the architecture component’s capabilities in an application-independent way. Using this signature, the relevant latencies can be derived automatically within the component. We will present the results from a small DSE case study.

### 4.3 Off-line model calibration

In Figure 4.3 a code fragment is shown from the basic processor model component available in Sesame. The processor component accepts read, write and execute events from the virtual layer and models processing time of an event as being busy or blocked for a certain number of cycles (line 19). We focus here on the modeling of execution (EX) events and in particular those that are mapped onto programmable processor cores. In this very simple model, a table is used to associate a latency to a particular type of event (line 18). For non-programmable processor cores that are dedicated to process certain fixed tasks, these latencies can often simply be taken from their respective documentation. However, for general purpose programmable cores the cycle count for certain events may vary depending for example on input data.

Here, we integrate a lower-level simulator in order to statically (i.e., before system-level simulation) calibrate the values in an operation latency table according to code-fragment performance measurements on the ISS. To explain this in more detail, consider Fig. 4.4. In this example, we assume that model component p2 onto which application process B is mapped needs to be calibrated using the ISS. This means that the code of Kahn application process B is crosscompiled for the ISS (indicated by B’ in Fig. 4.4a). We note that Figure 4.4a focuses on the application model level, and only abstractly depicts the mapping and architecture model levels. Also, throughout the remainder of this chapter, we take the
example of incorporating an ISS into Sesame. However, other types of simulation models (like RTL models) can also be used with trace calibration.

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 SimpleScalar ISS, we use its annotate 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 by an API using an underlying IPC mechanism needed to provide B’ (on the ISS) with the correct application input data. These data exchanges, which will be discussed in detail in the next section, only occur when the Kahn application process taking part in the calibration (process B in our example) performs communication. For example, when Kahn process B reads data from its input channel, it forwards the data to process B’ on the ISS, i.e., process B’ reads and writes its data from/to process B instead of a Kahn channel.

During execution, the ISS keeps track of the code fragment timings. From these measurements, timing averages are then used for calibrating the latency values of the architecture model component in question: i.e., the average value is used in the latency table of a processor model component. Off-line calibration is a relatively efficient mechanism since it is performed before system-level simulation and basically is a one-time effort. However, if there is high variability in the demands of computations (i.e., data-dependent computations) then the measured average timings may be of moderate accuracy. In this case it may be beneficial to perform calibration of trace events in a dynamic way as shown in the next section.

```c
1 class processor
2 // latency table initialized in yml:
3 operations_t = [] integer
4 operations : operations_t
5
6 read: (...) -> void {
7     ...
8 }
9
10 write: (...) -> void {
11     ...
12 }
13
14 execute: operand: integer
15     -> void {
16         latency : integer =
17         operations[operand];
18         block(latency);
19         reply();
20     }
21 }
22 }
23 while (true) {
24     block(any);
25 }
26
Figure 4.3: Extract of simulation code for a processor component
```
4.4 On-line trace calibration

The *on-line* trace calibration technique accomplishes mixed-level co-simulation by means of dynamic calibration of the event traces that are generated by an application model. Rather than using fixed values in the latency tables of processing components in Sesame’s architecture models (see Section 2), trace calibration dynamically computes – using lower-level simulators – the latency values of computational tasks. Subsequently, instead of generating fixed computational execution events, like `EXECUTE(T)` events, the Kahn application process generates `EXECUTE(D)` events, where D equals to the actual cycle count taken by, for example, a DCT computation (or any other computation in between communications). This is illustrated in Figure 4.4(b), where an instruction set simulator (ISS) is used for calibrating the trace from application process B.

![Diagram](image-url)

Figure 4.4: Incorporating an instruction set simulator using *off-line* model calibration (a) and *on-line* trace calibration (b)

Compared to the original static operation-latency table method, a calibrated trace can more accurately represent the computational behavior of an architecture component. For example, in Figure 4.4(b), statistics on pipeline stalls and cache behavior for process B can be retrieved from the ISS. Also, Sesame’s dataflow-based event refinement methodology [79] can still be applied to calibrated traces to, e.g., perform communication refinement, i.e., refine the `Read` and `Write` application events. The system-level effects of this improved accuracy can subsequently be measured within Sesame’s architecture performance model which accounts for the global timing consequences of all (calibrated as well as non-calibrated) event traces generated by the application model. Efforts to quantify the improved accuracy,
4.4. ON-LINE TRACE CALIBRATION

should however be performed with great care since their outcome heavily depends on the quality (i.e., accuracy) of the initial abstract performance models.

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

To explain the details behind the trace calibration technique, consider Figure 4.5. This figure renders the dashed box from Figure 4.4(b) in more detail. The code in the Kahn application processes typically consists of alternating periods of communication and computation, as illustrated by the small code fragment for process B in Figure 4.5. In this fragment, some data is read from Kahn channel c_in, followed by some computational code (which may also be discarded, as will be explained later), after which the resulting data is written to Kahn channel c_out. The two boxes on the right of this code fragment indicate what the run-time system of the application model executes when it encounters the Kahn read and write communications. Note that these run-time system actions are automatic and transparent: the programmer does not need to add or change code. First, the run-time system queries the ISS via the API, using API_get_cycles(), to retrieve the current cycle count from the ISS. As will also be described later on, the ISS provides this cycle information by executing a matching API_put_cycles() call. The run-time system then generates an Execute($\Delta$) application event for the architecture model, where $\Delta = n_{cur} - n_{prev}$, i.e., $\Delta$ equals to the time between the previous cycle query and the current one. Hence, the Execute event models the time that has past since the previous communication. Subsequently, a Read application event is generated for the architecture model. Hereafter, the actual read from Kahn channel c_in is performed. Finally, the data that has been read is copied, using API_write, to process B' running on the ISS.

Figure 4.5 also shows how the ISS side (process B') is handled. First, it sends the current cycle count of the ISS to the application model (API_put_cycles) to service the API_get_cycles() query from process B. Then, it reads the data that was sent by process B, i.e., the API_read from process B' matches up with the API_write from process B. After receiving the data, process B' executes the computational code shown in grey in Figure 4.5. This computational code is finished by a communication (a write to c_out), which again causes a cycle count query by the run-time system of the application model. The generated Execute($\Delta$) application event that follows, represents a detailed timing of the computational code on the ISS. Figure 4.5 also shows that process B' on the ISS first writes...
Figure 4.5: Interaction between application model and instruction set simulator.

back the resulting data to process B in the application model before the latter forwards this data to Kahn channel c_out. This allows for discarding the computational code between the communications in process B in the application model. In that case, only process B’ simulates computational functionality, while process B only communicates data with its neighboring application tasks. From the above, it should be clear that the API_get_cycles and API_read calls are blocking.

With trace calibration, it is relatively easy to incorporate any external low-level simulator into Sesame’s system-level architecture models. Only three API functions need to be introduced in the low-level simulator or in the code that runs on it (in the case of an ISS): API_put_cycles(), API_read(), and API_write(). Here, API_put_cycles() is the only function that needs a hook into the simulator to retrieve its cycle count. Most simulators provide such a hook, otherwise it can be created by a small modification to the simulator (as will be briefly explained in the next section). Using the API calls, the code to run on an ISS simulator (B’) can be trivially derived from the application code (B). Therefore, the total coding effort to enable trace calibration is small.

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

As a side note, we need to mention that a source of inaccuracy, of which a similar situation is reported in [55], occurs when mapping multiple application tasks to a single (programmable) architecture component. In this situation, the traces from these application...
tasks would be calibrated by different instances of an ISS. The scheduling of the (calibrated) application events from the different traces is subsequently performed at Sesame’s mapping layer. This approach has two major advantages. First, there is no need for running an OS-scheduler on an ISS since ISSs always execute a single task. Second, the ISS instances, representing a single processor in the architecture, can be executed in parallel on different hosts. However, since context-switching is not modeled by the ISSs, the simulated cache (performance) behavior in this situation may be inaccurate.

4.5 Trace calibration experiments

Figure 4.6: Modeling a Motion-JPEG application on an MP-SoC architecture.

To demonstrate that the performance overheads of trace calibration are low, we present a case study with a Motion-JPEG encoder application and a shared-memory MP-SoC architecture consisting of five processing elements. Sesame’s application model, architecture model and mapping for this case study are shown in Figure 4.6. In this experiment, we have incorporated SimpleScalar’s sim-outorder ISS [5] in Sesame’s high-level model of the MP-SoC architecture. This required only one small extension of the ISS: a system-call that retrieves the cycle count, which is needed by API_put_cycles(). Small C macros implement the functions API_read(), API_write(), and API_put_cycles() and are compiled together with the application code executing on the ISS.

For the experiments, we have used a small cluster of unloaded Pentium M 1.7GHz (Debian) machines connected by 100 Mbit ethernet. In all simulation runs, we have simulated the encoding of 11 CIF frames with a resolution of 128x128 pixels. Table 4.1 shows the wall-clock times (averaged over several runs) for three different simulation configurations executed on a single host machine: a Sesame-only system-level simulation (without trace calibration), a trace-calibrated (mixed-level) co-simulation where the DCT applica-
Table 4.1: Co-simulation performance.

<table>
<thead>
<tr>
<th></th>
<th>Sesame only</th>
<th>DCT on ISS</th>
<th>DCT+VLE on ISS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (secs)</td>
<td>8.1</td>
<td>157.1</td>
<td>279.6</td>
</tr>
<tr>
<td>Kcycles/sec</td>
<td>8,000</td>
<td>414</td>
<td>233</td>
</tr>
</tbody>
</table>

The simulation process is executed on the ISS (representing P2 in Figure 4.6), and a trace-calibrated co-simulation where both the DCT and VLE processes are executed on different ISSs (representing P2 and P3). For each simulation configuration, the number of simulated Kcycles/sec is also given. Table 4.1 clearly shows the performance drop when incorporating lower-level simulators in Sesame’s architecture models. The results also show the efficiency of the Sesame-only simulation (8 Mcycles/sec when simulating a 5 core MP-SoC).

To demonstrate that the performance decrease of our mixed-level co-simulations is almost entirely due to the lower-level simulators themselves and not due to co-simulation overheads, Figure 4.7 shows the execution time breakdowns of the two trace-calibrated co-simulation configurations from Table 4.1. The breakdowns clearly prove that the total execution times are totally dominated by the ISS execution times. The measured overheads are respectively 5% (DCT on ISS) and 1% (DCT+VLE on ISSs) of the total execution time. We suspect that the latter has lower overheads because of the scheduling of the two ISSs that allows for hiding some of the overheads caused by communications between the DCT/VLE processes and the ISSs.

By means of distributed co-simulation, the system-level simulation slowdown due to the incorporation of lower-level simulators can be effectively reduced. To illustrate this, Table 4.2 presents the performance effect of distributing the ISSs over different hosts. Here, we use the terms ‘local’ and ‘remote’ to indicate if an ISS is executed on respectively the same or a different host as Sesame’s application and architecture models. The results show that distributing the lower-level simulators over multiple hosts is certainly beneficial. For
example, by placing the ISSs for the DCT and VLE processes on two different hosts (see Table 4.2), a speedup of 1.86 is achieved in comparison to execution on a single machine (see Table 4.1). In this case, only 6.5% of the total execution time is due to overheads (including network overhead).

<table>
<thead>
<tr>
<th></th>
<th>DCT on remote ISS</th>
<th>DCT on local ISS</th>
<th>DCT+VLE on remote ISSs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (secs)</td>
<td>144.2</td>
<td>160.6</td>
<td>150.2</td>
</tr>
<tr>
<td>Kcycles/sec</td>
<td>452</td>
<td>405</td>
<td>433</td>
</tr>
</tbody>
</table>

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

For comparison, [55] lists the performance of similar case studies using their own work, Seamless CVE and Synopsys System Studio. With respect to the latter two, our co-simulations are one to two orders of magnitude faster, while for our Sesame-only simulations this is even three orders. The reported performance of the state-of-the-art technique proposed in [55] approximates our co-simulation performance, but they have used the ARMulator ISS which is significantly faster than the SimpleScalar ISS we have used.

To get some indication of the accuracy gains of trace calibration, we have also performed several simple experiments that compare a fully trace-calibrated model to a partially uncalibrated model. In this case, the uncalibrated model components have to use estimated latency values for application events. Statically estimating the performance of code executed on a particular processor is a well-known problem as it may be hard to deduce the correct number and types of executed instructions due to data-dependent behavior and to determine the CPI for the processor. Our experiments show, e.g., that if for one processor (onto which the DCT is mapped) we know the right number and types of executed instructions but mispredict the CPI by only 0.07 (where the actual CPI is 0.43) to calculate the latencies for the application events, then the estimated performance for the whole system is off by 13%. If the misprediction is bigger or if more components are uncalibrated then the system-level accuracy is reduced even further. For a more extensive validation case study of our calibrated models compared to an implemented system, we refer to [80]. Moreover, in Chapter 8, another validation case study will be shown.
4.6 Signature based model calibration

As we mentioned before, the off-line calibration method in Section 4.3 has a possible disadvantage that calibration is dependent on the application, requiring re-iteration of the calibration process to obtain new values for the latency table for each application. In the case of on-line calibration these values could be obtained automatically, provided that the KPN code is sufficiently platform independent to cross-compile it without problems for the ISS simulator. In both cases the (calibrated) event traces can be stored when the application workload has been fixed. The stored traces can subsequently be reused for further architectural design space exploration, resulting in a model performance improvement for the on-line calibrated model. This way, any additional effort and model execution time can be considered only a minor manual effort in the early stages of design for a system that runs a small number of applications. However, this effort may become a problem for a system targeting wider range of applications, since the calibration overhead would increase. In those situations it is desirable to be able to describe the behavioral (performance) properties of system components in an application-independent way.

In the following we use a general decomposition of the problem of determining performance values (eg. for use in a system-level model) in three sub-problems:

1. determine workload requirement
2. determine component capacity
3. compute performance estimate from (1) and (2)

More specifically, we can say that for a Kahn process $k_1$ and processor $p_1$, the performance $T$ is defined by a function $f$:

$$ T_{p_1} = f(k_1, p_1) \quad (4.1) $$

The performance of a certain architectural component naturally depends on both its computational capacity and the computational requirement of the workload that is mapped onto it. The aim is then to define a function $f$ that calculates the performance of $p_1$ as a number of cycles taking as input some workload requirement of $k_1$ and some capacity of $p_1$. We will refer to $k_1$ and $p_1$ respectively as the application signature and the architecture signature.

In this section we will describe the signature-based calibration method as proposed in [46] and show a simple case study to demonstrate the usefulness of this approach. Furthermore, we show that the accuracy of the signature-based method provides good relative (though not absolute) performance numbers.

4.6.1 Application requirements

To characterize application requirements a description is proposed that captures a Kahn process’s computational requirement at a high abstraction level. Towards this end we define an execution profile as a vector of instruction counts for the individual execution events generated by a process. In order to keep this so-called application signature architecture
4.6. SIGNATURE BASED MODEL CALIBRATION

<table>
<thead>
<tr>
<th>read</th>
<th>f_2</th>
<th>Signature index</th>
<th>AIS opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>execute</td>
<td>op_1</td>
<td>1</td>
<td>AIS_BRANCH</td>
<td>Block memory transfers</td>
</tr>
<tr>
<td>write</td>
<td>f_1</td>
<td>2</td>
<td>AIS_MEM</td>
<td>Memory transfers</td>
</tr>
<tr>
<td>read</td>
<td>f_2</td>
<td>3</td>
<td>AIS_BRANCH</td>
<td>Braches</td>
</tr>
<tr>
<td>execute</td>
<td>op_2</td>
<td>4</td>
<td>AIS_CORE</td>
<td>Co-proc. instructions</td>
</tr>
<tr>
<td>write</td>
<td>f_1</td>
<td>5</td>
<td>AIS_MEM</td>
<td>Int. multiplications</td>
</tr>
<tr>
<td>execute</td>
<td>op_1</td>
<td>6</td>
<td>AIS исполни</td>
<td>Simple Int. arithmetic</td>
</tr>
<tr>
<td>write</td>
<td>f_1</td>
<td>7</td>
<td>AIS_OPS</td>
<td>Software interrupts</td>
</tr>
<tr>
<td>write</td>
<td>f_1</td>
<td>8</td>
<td>AIS_UNKNOWN</td>
<td>Non-mappable instruction</td>
</tr>
</tbody>
</table>

Figure 4.8: Table (a) lists the event trace of process \( k_1 \). Table (b) shows the currently defined AIS instructions with their index in the vector-based process signatures and Table (c) shows an execution trace of \( op_1 \) as obtained by an ARM ISS (left column) and the corresponding AIS classification (right column).
independent, we use an abstract instruction set (AIS), see Figure 4.8b. For each type of processor that uses the signature based calibration method a mapping has to be provided that maps its particular instructions to one of the AIS categories. Next we use an instruction set simulator to generate an execution profile for those parts of the code that are represented by Sesame’s execute events. As an example Figure 4.8a shows a Kahn process event trace containing two \( op_1 \) and one \( op_2 \) events. Note that here we discard the description for communication events (read and write) for simplicity, more details can be found in [47]. In the source code we annotate the beginning and end of each of the execute events in a very similar way as with the trace calibration technique. This enables the ISS to know the start and end points to count instructions according to our AIS categories. For the example execution trace in Figure 4.8a; one of the two \( op_1 \) executions has the detailed execution profile as listed in Figure 4.8c. Measurements for each \( op_1 \) results in the following derived signatures:

\[
\begin{align*}
\text{\( op_1 \) signature} & = [7, 17, 8, 0, 2, 31, 2, 0] \\
\text{\( op_2 \) signature} & = [3, 15, 1, 0, 3, 9, 0, 0] \\
\text{\( op_1 \) signature} & = [8, 15, 8, 0, 3, 29, 2, 0]
\end{align*}
\]

Note that for the purpose of the example we only consider two execution events, but in general we would aim to measure as many signatures as possible. Also note that when an \( op_1 \) is measured repeatedly, its signature may consist of different AIS instruction counts because of data-dependencies or pseudo-random behavior.

### 4.6.2 Processor capacity and performance estimation

Next we will show how to derive the signature \( p_1 \) that describes the computational capacity of a processor. Using this signature we will then be able to estimate the number of cycles required for any workload by defining \( f \) from Equation 4.1.

\[
T_{p_1} = f(k_1 \text{.signature}, p_1 \text{.signature})
\]  

While the ISS is executing (and counting instructions of the application signatures), we also register the increase of simulated clock cycles within the ISS. In this way we not only obtain the instruction count for each \( op_x \), but also the measured latencies for each \( op_x \). If there was an exact linear relationship between a signature and its execution performance, then the following would hold:

\[
\begin{pmatrix}
\text{\( op_1 \) signature} \\
\text{\( op_2 \) signature} \\
\text{\( op_1 \) signature}
\end{pmatrix} \cdot \text{\( p_1 \) signature} = \begin{pmatrix} T_{op_1} \\ T_{op_2} \\ T_{op_1} \end{pmatrix}
\]  

Of course it is not to be expected that such an exact linear relationship exists, but instead we approximate a solution \( p_1 \) signature for the matrix equation using the least squares method. The vector of application signatures and its measured timings will serve as our benchmark.
4.7. SIGNATURE-BASED CALIBRATION EXPERIMENTS

(training set) for the approximation. The performance estimation of the processor signature in the previous example would for example result in:

$$\begin{pmatrix} 7 & 17 & 8 & 0 & 2 & 31 & 2 & 0 \\ 3 & 15 & 1 & 0 & 3 & 9 & 0 & 0 \\ 8 & 15 & 8 & 0 & 3 & 29 & 2 & 0 \end{pmatrix} \cdot p_1.\text{signature} = \begin{pmatrix} 185 \\ 369 \\ 196 \end{pmatrix}$$ (4.4)

The processor’s signature can be considered vector of 8 weight-factors, where each value represents the average estimated contribution of each AIS instruction to the total processing time. Now we can define the function $f$ from Equation (4.2) as the inner product of both signatures. This allows us to estimate a cycle count for an arbitrary operation $op_x$ that was not in our original training set:

$$T_{op_x} = p_1.\text{signature}^T \cdot op_x.\text{signature}$$ (4.5)

In general, the training set should be made as large as possible, containing $op_x$ measurements from a range of different applications in order to provide a good approximation of the processor signature for a particular processor. Also note that the training process may measure the same operation signature $op_x$ with differently measured cycle counts, due to the previously mentioned data-dependencies or micro-architectural effects on pipelining and caching. All these effects will therefore be included in the approximation of the processor signature.

4.7 Signature-based calibration experiments

In this section, we present an experiment using a Motion-JPEG (M-JPEG) encoder application in which the possibilities of mapping application tasks to architecture components are explored. We compare the results of an off-line calibrated model with a model using signature-based calibration. The signature-based performance models are calibrated using the Mediabench benchmark suite [61] as an external training set. Finally we discuss the impact of the choice for the training set on the exploration results.

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 4.9. The target MP-SoC consists of four ARM processors with 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) onto the processors in the MP-SoC platform. Without consideration for any duplicate mappings caused by the symmetry in the target platform, the size of the design space can be calculated as: $4^6 = 4096$ (6 tasks onto 4 processors).

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 [82]. The generated ARM instruction traces were used to create the application and architecture signatures as described in the previous section. Note
that determining the application and architecture signatures is one-time (pre-exploration) effort, and that the resulting execution latencies can be used to explore the full range of different mapping options. To train our performance model, we first constructed the (application) operation signatures for each separate Mediabench program\(^1\) using the approach as discussed in Section 4.6.1. Since the execution time (in terms of simulated cycles) of the \texttt{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. Figure 4.10 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 4.10, Figure 4.11 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 4.10 and 4.11 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 4.12 shows a comparison of the DSE results of the Mediabench-trained ("Full Mediabench" in the graph) and the reference simulation model. The reference model uses “exact” latencies for the various computational events as directly obtained by ISS measurements (no latencies that were obtained using our signature-regression model). 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 4.3). But the trends between the graphs of the two models still is highly similar. This is especially true

\(^1\)The \texttt{mpeg2enc} and \texttt{mpeg2dec} benchmarks were excluded due to execution problems on the Simult-ARM simulator.

---

![Figure 4.9: Mapping an M-JPEG application to a crossbar-based MP-SoC architecture.](image-url)
for the better-performing mapping instances (lower mapping indices). This again implies that both models find exactly the same optimal mappings and that the model is quite good at determining the relative performance of the mappings (which is an important property for early DSE).

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 4.12 and Table 4.3. The results of this new performance model are slightly better (average error of 24.9%, see Table 4.3) 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 4.12 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.

<table>
<thead>
<tr>
<th></th>
<th>Full Mediabench</th>
<th>Outliers removed</th>
<th>Large programs</th>
<th>Small programs</th>
<th>DCT-similar programs</th>
<th>MJPEG-self trained</th>
</tr>
</thead>
<tbody>
<tr>
<td>Av. error</td>
<td>29.6%</td>
<td>24.9%</td>
<td>18.6%</td>
<td>79.8%</td>
<td>7.0%</td>
<td>9.2</td>
</tr>
<tr>
<td>Std.dev.</td>
<td>3.8</td>
<td>4.4</td>
<td>5.1</td>
<td>2.1</td>
<td>4.5</td>
<td>4.5</td>
</tr>
</tbody>
</table>

Table 4.3: Average error and standard deviation of the various trained models as compared to the reference model.

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 a model that is trained the MJPEG application itself (last column of 4.3).

### 4.7.1 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., [13, 72]. 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 [67] proposes a so-called vertical simulation approach that shows similarities with our calibration approach. It is unclear, however, whether or not vertical simulation has ever been realized. In [14], 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 [38]. In the context of system-level modeling, various research efforts (e.g., [7, 16, 15]) also refer to back annotation as a technique for adding more detailed timing information to higher-level models in the case lower-level models are available. But these efforts generally do not provide much insight of how back annotation is applied during the early stages of design where lower-level models typically are not abundant. In a way, our calibration methods can be considered as a form of back annotating the latency tables in Sesame’s architecture models using results from ISS simulation and/or automated component synthesis.

Related to our on-line calibration technique, much work has been performed in the field of mixed-level HW/SW co-simulation, mainly from the viewpoint of co-verification. This has resulted in a multitude of academic and commercial co-simulation frameworks (e.g., [2, 1, 3, 42, 32, 10]). Such frameworks typically combine behavioral models, ISSs, bus-functional models or HDL models into a single co-simulation. These mixed-level co-simulations generally need to solve two important problems: i) making the co-simulation functionally correct by translating any differences in data and control granularity between simulation components, and ii) keeping the global timing correct by synchronizing the simulator components and overcoming differences in timing granularity. The functionality issue is usually resolved using wrappers, while global timing is typically controlled using either a parallel discrete-event simulation method [31] or a centralized simulation backbone using e.g. SystemC [32, 10]. Synchronization between simulation components usually takes place with the finest timing granularity (i.e. lowest abstraction level) as the greatest common denominator between components. E.g., system-level co-simulations with cycle-accurate components are typically synchronized at cycle granularity, causing high performance overheads. Besides the performance overheads caused by wrappers and time synchronization, the IPC mechanisms often used for communication between the co-simulation components may also severely limit performance [55], especially when synchronizing at cycle granularity.

In the mixed-level co-simulation that results from our on-line (trace-)calibration technique, we take the opposite direction with respect to maintaining global timing. Instead of
4.8. CONCLUSION

synchronizing simulation components at the finest timing granularity, it maintains correct
global timing at the highest possible level of abstraction, being the level of Sesame’s ab-
stract architecture model components. As shown in [105], the performance overhead caused
by wrappers and time synchronizations is in that case reduced to a minimum. Our on-line
calibration technique shows some similarities with the trace-driven co-simulation technique
in [55]. However, the latter operates at a lower abstraction level and is applied in a classical
HW/SW co-simulation context.

In particular with respect to the signature-based calibration technique, we note that much
work has been performed in the area of software performance estimation [8], including meth-
ods 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 [38], a source-based estimation technique is proposed using the concept of "vir-
tual 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 instruc-
tions. 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 [92].

As In [24], a workload modeling approach based on execution profiles is discussed for
statistical micro-architectural simulation. Because they address micro-architectural simula-
tion, 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 [50], the authors suggest to
derive a linear model from a small set of simulations. This method tries to model the per-
formance of a processor at a mesoscopic level. For example, cache behavior 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 com-
parable with the ‘weight’ vector discussed in Section 4.6.2. Another interesting approach
is presented in [93], in which the CPI for in-order architectures is predicted using a Monte
Carlo based model. The Milan framework [70] deploys a design pruning approach using
symbolic (instead of analytic) analysis methods to reduce the design space that needs to be
explored with simulation.

4.8 Conclusion

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

A disadvantage of the trace calibration method is that it has to be performed anew for each application. The signature-based calibration technique is aimed at finding a characteristic “signature” that summarizes in an application independent way the computational capacity of a processor component. Experiments showed that although signature-based performance models accurately represent relative performance behavior, achieving correct absolute performance behavior is more difficult. The latter requires very careful selection of benchmarks to train the model. This is, however, not a major objection, since in the early stages of system design, determining relative performance of different design options is often more important than absolute performance.

In summary, both trace-calibration (on-line and off-line) and signature based calibration techniques can be effectively used to create system-level performance models with relative ease. Choosing the right technique for a given design problem depends on the required speed-accuracy trade-off, the availability of suitable lower-level simulators and the ability to create good training sets. Encouraged by the results of the on-line trace-calibration method, we have started work on a co-simulation technique where event trace measurements are performed on actual hardware, instead of an ISS. The technique is completely analogous to the on-line trace calibration method, but now the API functions transfer data and cycle measurements (Section 4.3) directly to a process running on an FPGA prototype platform. An advantage of this “hardware-in-the-loop” co-simulation is that the measured execution latencies are 100% accurate, whereas an ISS-simulator often is still an approximation of the actual target processor. However, our initial results showed that synchronization overhead is larger than in the case of on-line trace-calibration with ISS, due to the communication mechanism with the stand-alone FPGA board. Nevertheless, we will report on this method in future work, since it is likely to provide yet another valuable option for the designer with respect to the speed-performance trade-off.
Figure 4.10: Histogram with the various AIS opcode counts of the Mediabench training set.

Figure 4.11: Histogram with the various AIS opcode counts of M-JPEG.
Figure 4.12: Comparing the DSE results from our Mediabench-trained models