Rapid Evaluation of Instantiations of Embedded Systems Architectures: a Case Study

Frank Terpstra Simon Polstra Andy Pimentel Bob Hertzberger

Department of Computer Science
University of Amsterdam
Kruislaan 403, 1098 SJ Amsterdam
{ftrpstra,spolstra,andy,bob}@science.uva.nl

Abstract—**Modern signal processing and multimedia embedded systems increasingly have heterogeneous system architectures. In these systems, programmable processors provide flexibility to support multiple applications, while dedicated hardware blocks provide high performance for time-critical application tasks. The heterogeneity of these embedded systems and the varying demands of their growing number of target applications greatly complicate the system design. For this reason, the design of these embedded systems should be supported by tools which facilitate efficient design space exploration.**

As part of the Artemis project, we are developing the Sesame modeling and simulation environment which addresses the design space exploration of heterogeneous embedded systems architectures. Sesame aims at supporting the rapid evaluation of instantiations of embedded systems architectures at multiple levels of abstraction. In this paper, we present an overview of the current status of Sesame by using an illustrative case study of a modified M-JPEG application.

Keywords—**Media and signal processing, system architecture design, design space exploration, computer architecture modeling and simulation**

I. Introduction

Modern embedded systems, such as those used for multi-media and signal processing, are becoming increasingly complex. These systems need to support multiple applications and different standards for communication and coding of digital content. They also have to be flexible so that they can easily support new standards and applications in the future. In the case of mobile devices this also has to be achieved with only limited power consumption. This means that modern embedded systems have programmable microprocessors which provide flexibility allowing multiple applications to run on one system. In addition, they also have dedicated hardware to save power, or to give time critical tasks the high performance they demand. These systems are called *heterogeneous* because they use both dedicated and programmable hardware. Within the scope of the Artemis project [1], we aim at reducing the design time of heterogeneous embedded media systems. One of the goals of the Artemis project is to create a modeling and simulation environment for efficient design space exploration of architectures for these embedded systems. The requirements for efficient design space exploration are twofold. Firstly, the ability to quickly evaluate the performance impact of the following design issues for a wide range of (media) applications:

- new architecture designs
- application to architecture mappings, i.e., which application tasks are mapped onto which architecture components
- different hardware/software partitionings, i.e., which application tasks are implemented by software and which ones are implemented by hardware

Secondly, design space exploration should allow for simulation at multiple levels of abstraction. This enables a stepwise refinement approach. Abstract simulation models are used for the efficient exploration of large design spaces early on in the design process, while more detailed models are used for more focused exploration in the later stages.

Within the Artemis project there are two prototype modeling and simulation frameworks: Spade (System-level Performance Analysis and Design space Exploration) [2] and Sesame (Simulation of Embedded System Architectures for Multi-level Exploration) [3]. In [4], the authors demonstrated, by means of the Spade environment, that the modeling and simulation methodology of Artemis facilitates efficient evaluation of different application to architecture mappings and hardware/software partitionings. In this paper, we use the Sesame environment to show the capability of quickly evaluating different architecture designs. To this end, we studied a modified M-JPEG application for which we evaluated multi-processor architectures with different communication interconnects. This case study was restricted to a single (high) level of abstraction, as the support for multiple levels of abstraction in Sesame is still planned as future work.

The paper is organized as follows. First we briefly describe the Sesame modeling and simulation environment and how it achieves a high degree of flexibility. Section III gives a detailed description of our M-JPEG case study. In Section IV we describe our modifications to the M-JPEG target architecture. Finally, we will discuss the results and give our conclusions.

II. Sesame

The Sesame modeling and simulation environment is an architecture exploration tool with which architectural design choices can be explored in an efficient and flexible manner. It should be used alongside a design tool, as a research vehicle to rapidly investigate the design choices and for the designer to gain insight into architectures.

Sesame does not perform traditional hardware/software co-simulation [5] in which the software and hardware parts of an embedded system are simulated together in one simulation (illustrated in Figure 1a). The reason for not using such co-simulation is that it forces the designer to make decisions about software/hardware partitioning before the simulation models are build. Evidently, this significantly hampers the evaluation of different hardware/software partitionings, since a new system model is required for the assessment of every partitioning. Instead, Sesame enforces a separate *application model* and *architecture model* (shown in Figure 1b). The application model describes the functional behavior of an application in an architecture independent manner. It generates traces of events that represent the application workload imposed on the architecture. These traces of application events drive the simulation of the underlying architecture model. Subsequently, the architecture model accounts for the architectural performance constraints and simulates the performance consequences of the events generated by the application model. By varying the latency associated with an application event, one can simulate hardware or software execution of that event. This allows us to rapidly experiment with different hardware/software partitionings. Essential in this approach is that an application model is as independent from an architecture model as possible so that it can easily be co-simulated with alternative architecture models, either modeling different architectures or the same architecture at different abstraction levels.

Because we separate application and architecture models the traces generated by the application model have to be explicitly mapped onto the components of the architecture model. This mapping can be changed easily allowing for evaluation of different application to architecture mappings (see also section III-B).

Application and architecture modeling will be briefly discussed in the following sections. For a complete description we refer to [6].

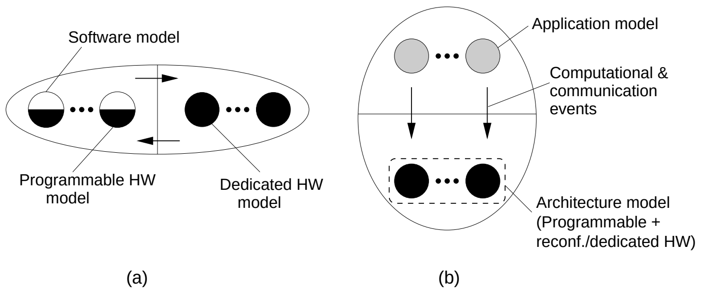
Software model
Programmable HW model
Dedicated HW model
(a)
Application model
Computational & communication events
Architecture model (Programmable + reconf./dedicated HW)
(b)

Fig. 1. Traditional co-simulation (A). The Artemis methodology: separate application and architecture models (B).

A. Application modeling

Application modeling uses the Kahn Process Network [7] model of computation. This model consists of parallel processes communicating via unbounded FIFO channels. In this model reading from channels is done in a blocking manner while writing is non-blocking. The reason for using Kahn Process Networks is that they fit nicely with signal processing and multi-media applications and that they are deterministic. The latter guarantees that application behavior is architecture independent. This is necessary for separating the architecture and application models.

To model applications we start from sequential programs which are restructured into a Kahn Process Network. The tool used to do this is called Y-Chart Application Programmers Interface [8] or YAPI. The restructuring is performed with the three functions `read`, `write` and `execute`, provided by YAPI. The `read` and `write` functions are used for communication in the Kahn Process Network. As a side-effect they generate trace events, which represent the communication workload of a Kahn process. The `execute` function is used as an annotation only. This annotation generates a trace event describing the computational activity of a Kahn process. These trace events may be coarse grain, such as “ execute a Discrete Cosine Transform (DCT)”.

B. Architecture modeling

The Sesame architecture models are implemented in the discrete-event simulation language Pearl [9]. This is a relatively small but powerful object-based language which has specifically been designed with the purpose of (abstract) computer architecture modeling in mind. As a consequence, Pearl has shown to be extremely suitable for easily and quickly building new or extending existing architecture models [10], while also yielding simulation speeds of up to an order of magnitude faster than those obtained by more general-purpose simulation languages. This greatly improves the scope of the design space that can be ex-

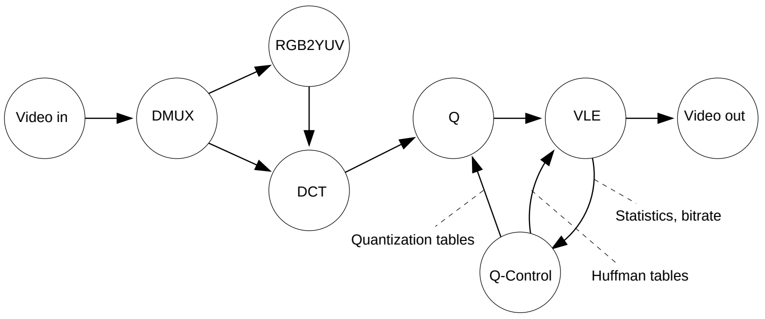

Fig. 2. The M-JPEG* application model with Kahn processes (circles) and communication channels (arrows).

plored in a reasonable amount of time.

Pearl provides the user with a default set of post-mortem simulation statistics, such as utilization of architecture components and critical path analysis between components, and there is support for user-instrumentation to collect statistics. In addition, Pearl allows for easily constructing a run-time visualization of the architecture simulation.

Currently, Sesame only provides a small set of Pearl models of architecture components such as processing cores, communication media (like busses) and different types of memory. All models are at the so-called black box level. For the processing core models this means that they can simulate timing behavior of both programmable and dedicated hardware components by simply assigning a parameterizable latency to the application events from the Kahn application model. When simulating a programmable processor, application events are processed using higher latencies, while for dedicated hardware lower latencies are used.

III. The M-JPEG* case study

A. The application

To demonstrate the flexibility of modeling in Sesame we applied the current version of our modeling and simulation environment to a modified M-JPEG encoding application, referred to as M-JPEG* [11]. The M-JPEG* application slightly differs from traditional M-JPEG as it can operate on video data in both YUV and RGB formats on a per-frame basis. In addition, it includes dynamic quality control by means of on-the-fly generation of quantization and Huffman tables. The application model of M-JPEG* is shown in Figure 2. In this figure, the circles refer to the Kahn processes and the arrows to the Kahn (FIFO) communication channels.

The data received in the `Video-in` Kahn process, which is either in RGB or YUV format, is sent to the `DMUX` in blocks of 8×8 pixels. The `DMUX` first determines the format and then forwards data from RGB frames to the `RGB2YUV` converter process, while YUV data is sent directly to the `DCT` Kahn process. Once the data has been transformed by the `DCT` process the blocks are quantized by the `Q` Kahn process using s set of quantization tables. The next step, performed by the `VLE` process, is the variable length encoding of the quantized DCT coefficients followed by entropy encoding, such as Huffman encoding. Finally, the resulting bitstream is sent to the `Video-out` process.

Huffman encoding[1] requires tables which are either static (specified by the user) or dynamic, i.e., specifically computed on a per-frame basis for the video stream. In M-JPEG*, the tables for Huffman encoding and those required for quantization are generated for each frame in the video stream. The quality control process (`Q-Control`) computes the tables from information gathered about the previous video frame. For this purpose, image statistics and obtained compression bitrate are transmitted by the `VLE` to the `Q-Control` Kahn process. When the calculations by the `Q-Control` process are finished, updated tables are sent to both the `Q` and `VLE` Kahn processes. A detailed description of the M-JPEG* application and its model can be found in [11].

B. The base architecture and mapping

The base M-JPEG* target architecture has five processing components connected via a common bus to a shared memory. In Figure 3, this architecture is shown together with the mapping of the M-JPEG* application onto it. Of the five processing components in the architecture, one is a general purpose microprocessor, two are DSPs and two are non-programmable components. The non-programmable components are used for input and output processing and are referred to as respectively the VIP (Video In Processor) and VOP (Video Out Processor). The two DSPs are used for computationally intensive tasks. One of them is used for RGB to YUV conversion and the DCT transform. We

[1]In M-JPEG*, we assume that Huffman encoding is the default entropy encoding scheme.

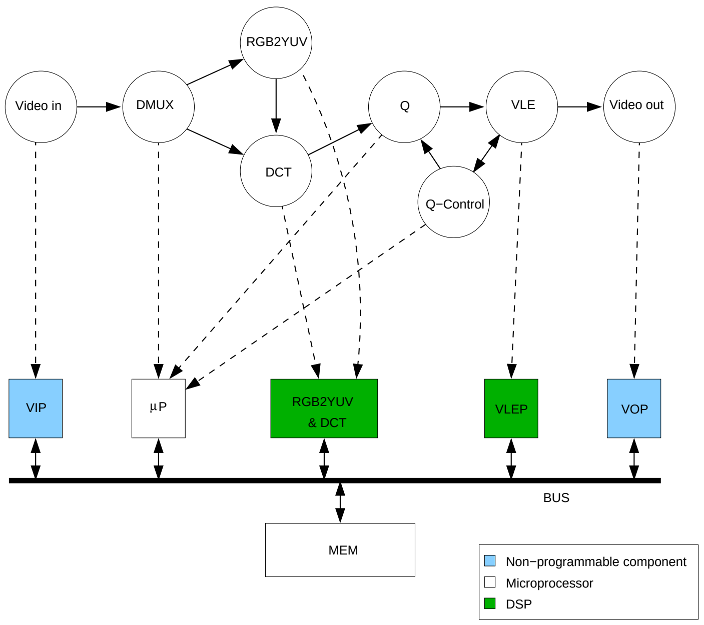

Fig. 3. M-JPEG*'s application to architecture mapping

```
class bus

mem   : memory
setup : integer

load : (nbytes:integer, address:integer)->void
{
   blockt( setup );
   mem ! load(nbytes, address);
   reply();
}

// [ store method is omitted ]

{
  while(true) {
    block(load, store);
  };
}
```

Fig. 4. Pearl code for the common bus object.

refer to this component as the RGB2YUV & DCT component. The other DSP is used for variable length encoding and is referred to as the VLEP. For the memory we assume DRAM, while the bus is assumed to be 64 bits wide. Communication between components is performed through buffers in shared memory.

In the remainder of this section, we demonstrate how Sesame and, in particular, its Pearl simulation language, allows for easily building abstract architecture models. To this end, Figure 4 shows the Pearl code of the bus model for the M-JPEG* target architecture. As Pearl is an object-based language and architecture components are modeled by objects, the code shown in Figure 4 embodies the class of bus objects.

A bus object has two object variables, `mem` and `setup`. These variables are initialized at the beginning of the simulation, and more specifically, at the instantiation of a bus object. The `mem` variable references the memory object that is connected to the bus and the setup time of a connection on the bus is specified by `setup`. A bus object has two methods: `load` and `store`. The `store` method is not shown here since it is identical to the `load` method. To explain how the `load` method works we first need to give some background on the `blockt()` function. Pearl is equipped with a virtual clock that holds the current simulation time. When an object wants to wait for an interval in simulated time it uses the `blockt()` function. In our example, the bus object uses the `blockt()` function to wait for `setup` time units in order to account for the connection setup latency. The statement "`mem ! load(nbytes, address)`" calls the `load` method of the memory object `mem` by sending it a synchronous message. Since it is synchronous the bus has to wait until the memory has explicitly returned a reply message. The latter is done by the `reply()` primitive. This synchronous message passing may also cause the virtual clock to advance in time, e.g., when the memory object accounts for the time it takes to retrieve the requested data.

In Figure 4, the `reply()` returns control to one of the processor objects (which are connected to the bus object) that has called the `load` method. At the bottom is the main loop of the object which does nothing until either the `load` or `store` method is called (by one of the processor objects). We note that we do not explicitly model bus arbitration, but instead we use Pearl's internal scheduling: it applies a FCFS strategy for incoming method calls at an object. As the piece of code in Figure 4 shows, Pearl objects are simple and compact, and therefore allow for easy construction of the architecture models.

In the Pearl simulation language, the instantiation of objects and the specification of the connections between objects are done using a so-called "topology file". In Sesame, this file is also used for specifying the mapping of the incoming application traces from the Kahn model to the components in the architecture model. Figure 5 shows the topology file for the M-JPEG* base architecture and mapping as shown in Figure 3. The first column of the topology file contains the names of the objects that need to be instantiated, while after the colon the object class is specified. Together with this class-name, a number of parameters are specified. The different classes and their parameters are explained below.

- The program class is actually not part of the architecture

```
commonbus() {
  vidin   : program(6,2,[header,buf1],vip)
  rgbyuv  : program(4,2,[buf2,buf3],rgbdct)
  dct     : program(1,4,[buf3,xx,type,buf4],rgbdct)
  dmux    : program(2,7,[header,buf1,fsize,buf2,xx,type,numof],mp)
  quant   : program(3,4,[buf4,qtable,qcmd,buf5],mp)
  control : program(0,7,[numof,stats,qtable,qcmd,hcmd,htable,info],mp)
  vle     : program(5,6,[buf5,hcmd,htable,stats,flag,stream],vlep)
  vidout  : program(7,4,[fsize,flag,info,stream],vop)
  vip     : processor(bus,10,[0,0,20,0,0,0,0,0,0,0])
  rgbdct  : processor(bus,10,[0,200,0,0,0,0,0,192,0,0])
  mp      : processor(bus,10,[180,0,0,0,154,1,23,0,2,154])
  vlep    : processor(bus,10,[0,0,0,0,154,0,0,0,0,154])
  vop     : processor(bus,10,[0,0,0,20,0,0,0,0,0,0])
  xx      : buffer(0,   64, 1)
  fsize   : buffer(0,    4, 4)
  header  : buffer(1,    7, 1)
  type    : buffer(0,    1, 4)
  numof   : buffer(0,    2, 4)
  info    : buffer(1,  672, 2)
  buf1    : buffer(1,   64, 16)
  buf2    : buffer(1,   64, 2)
  buf3    : buffer(0,   64, 4)
  buf4    : buffer(1,  128, 4)
  buf5    : buffer(1,  128, 4)
  stream  : buffer(1,    1, 4)
  flag    : buffer(0,    1, 4)
  qtable  : buffer(1,  128, 2)
  qcmd    : buffer(0,    1, 150)
  hcmd    : buffer(0,    1, 150)
  htable  : buffer(1, 1536, 2)
  stats   : buffer(1,  514, 1)
  bus     : bus(mem, 1, 8)
  mem     : memory(10,8)
}
```

Fig. 5. Topology definition for the M-JPEG* simulation: this shows how Pearl objects are instantiated and connected.

model itself. Instead, a program object reads in an application trace from a Kahn process. It then dispatches the events to a component in the architecture model. So, it acts as an interface between the application model and the architecture model. The program class has four parameters of which the first one is an identifier used for identifying the event trace queue to read from. The second one gives the number of FIFO buffers connected to a program object, after which these FIFO buffers are given in an array. The purpose of these two parameters will be elaborated upon in the next section. The last parameter defines on which processor a program will run: this is the application to architecture mapping.

- The processor class has three parameters. The first one describes to which memory interconnect the processor is connected. The second parameter gives the size of the instruction set, being the different application events for which the timing behavior needs to be modeled. This is followed by the latencies of each of these instructions. The value of these latencies can be obtained by a low-level simulator, performance estimation tools or by estimations of an experienced designer. By adapting these latencies, one can easily change the speed of a processor.
- The buffer class has three parameters. The first one specifies whether communication is performed over the interconnect or internally. When a buffer connects two programs which are on the same processor, communication is assumed to be performed internally. When the two programs are on different processors, communication is performed through shared memory, resulting in bus traffic. The second parameter of the buffer class specifies the size of the tokens in the buffer while the third parameter specifies the maximum number of these tokens that can be in the buffer at one time.
- The bus class has three parameters. The first one specifies the memory it is connected to. The second one defines the time for setting up a connection and the third one specifies the bus width in bytes.
- The memory class takes two parameters. The first specifies the delay for reading or writing one word and the second specifies the width in bytes of the memory interconnect it is attached to.

As one can see the topology file allows for easy configuration of a Pearl simulation. It is simply a question of changing a few numbers to change the application architecture mapping or to change the characteristics of a pro-

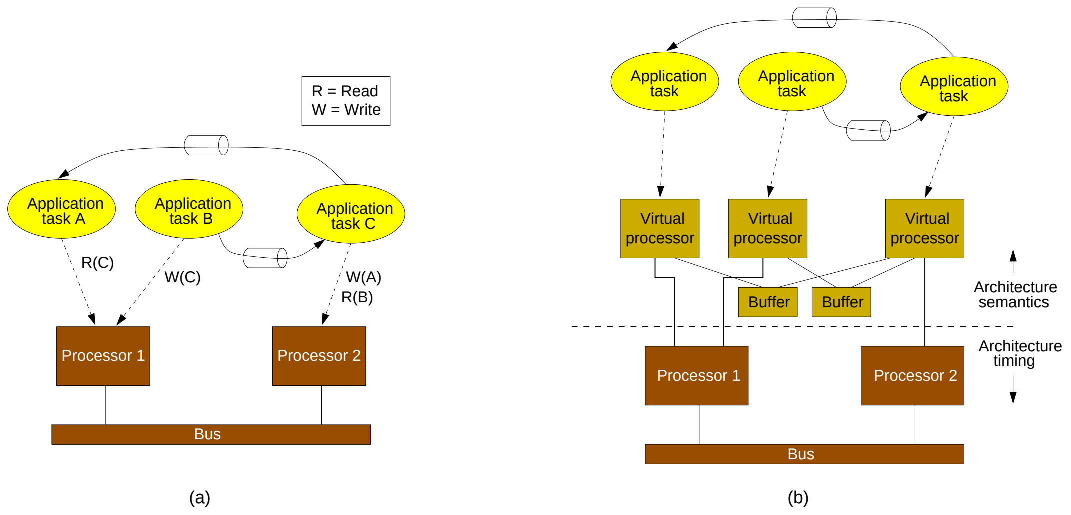

Fig. 6. (A) shows a potential deadlock situation due to scheduling of communication events. Our solution to this problem, using virtual processors, is illustrated in (B).

cessor. For example, replacing a DSP for a dedicated hardware component in our M-JPEG* base architecture model can simply be achieved by reducing the instruction latencies of the processor object in question.

C. Scheduling of communication events

As we already described, multiple Kahn processes of the application model can be mapped onto a single architecture component. In this case, the incoming event traces need to be scheduled. Scheduling of communication events is, however, not straightforward as it may cause deadlocks. Such a situation is illustrated in Figure 6a. In this example, Kahn process A reads data from Kahn process C, Kahn process B writes data for process C and Kahn process C first reads the data from B after which it writes the data for A. Since Kahn processes A and B are mapped onto a single processor, their read and write events need to be scheduled. Assume that the read event from Kahn process A is dispatched first to the processor 1. Processor 2 receives the read event from Kahn process C. In this case, a deadlock occurs since both dispatched read events cannot be carried out as there are no matching write events. This results in the processors to block.

In Figure 6b, our solution to the above problem is depicted. In the architecture model, we distinguish a semantic level and a timing level. The semantic level consists of virtual processor components (in the topology file of Figure 5 referred to as program objects) and FIFO buffers for communication between the virtual processors. These buffers have a one-to-one relationship with the FIFO channels in the Kahn application model. As can be seen from Figure 6b, multiple virtual processors can be mapped onto a single model of an actual processor. In this scheme, computation events are directly forwarded by a virtual processor to the actual processor model which subsequently models the timing consequences of these events. However, for communication events, the appropriate buffer at the semantic level is first consulted to check whether or not a communication is safe to take place. If it is found to be safe (e.g., data is available when performing a read event), then communication events may be forwarded to the actual processor model.

IV. Design space exploration

In this section we illustrate that Sesame and its Pearl simulation language facilitate efficient evaluation of different candidate architectures. For this purpose, we performed an experiment in which we modeled, simulated and briefly studied two alternative communication structures for the M-JPEG* architecture: a crossbar and an Omega network. To avoid confusion the original M-JPEG* architecture will be referred to as the common bus architecture. In our experiments, the input video stream consists of images captured in a resolution of 128×128 pixels with RGB color encoding. Figure 7 shows the simulation results in terms of measured memory throughputs for all three candidate architectures (using a common bus, crossbar or Omega network). In the following subsections, the results for each of the communication structures are explained in more detail.

A. Common bus

In Figure 8, a description is given of the activities of the various architecture components during simulation of the common bus architecture. For each component, a bar shows the breakdown of the time each component spends

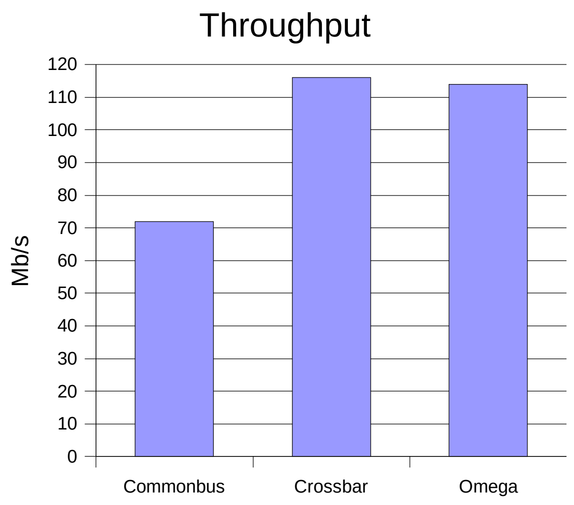

Fig. 7. Throughput in MBytes per second

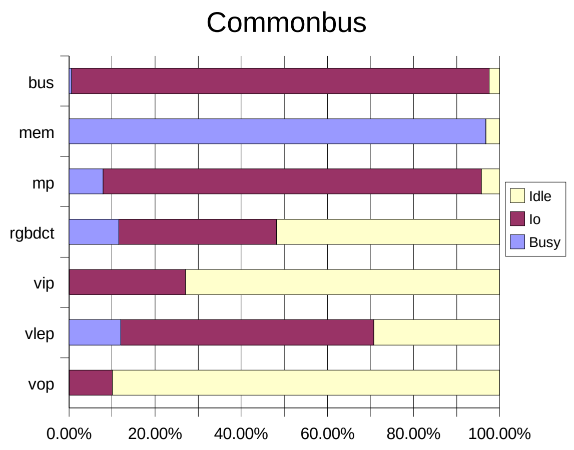

Fig. 8. Results for the common bus showing busy/io/idle statistics for all architecture components.

on I/O, being busy and being idle. As Figure 8 shows, the common bus architecture has a high memory utilization while the various processors have low utilization and spend a lot of time doing I/O. As Figure 7 shows, memory throughput for the common bus is only 70 megabytes per second. With the input stream of images in 128×128 resolution, a framerate of 76 frames per second is obtained. While this is more than enough for real time operation, this is a very low resolution. Such performance is roughly equivalent to only 3 frames per second in full resolution PAL television (720×576). The common bus interface to the memory is clearly a bottleneck and therefore a candidate for further exploration. In [4], similar conclusions about the M-JPEG* architecture were drawn from experiments using the Spade architecture modeling and simulation environment.

B. Crossbar

A crossbar is a well-known architecture for applications in which a high data throughput is needed. Figure 9 shows an architecture of a crossbar. It directly connects every processor with every memory bank with just a single switch latency for each connection. Besides being a fast architecture it is also expensive because it uses p^2 switches, p being the number of processors.

To reduce the communication bottleneck of our M-JPEG* architecture, we implemented a Pearl simulation model for the architecture shown in Figure 9. We replaced the common bus model in M-JPEG* with this crossbar model. The memory in this architecture is distributed over five banks. Therefore, a mapping of buffers to memories was defined. This mapping is performed in the topology file (see Figure 5) and is, like the application to architecture mapping, easy to configure. As the results in Figure 7 show there is a substantial gain in throughput compared with the common bus. When we look at the architecture component statistics in Figure 10 we see that all the components spent more time doing work and less time waiting for I/O. Since the memory load is now divided over five memories, memory busy percentage is at about 20% for most memories. Note that memory 5 is still busy for more than 80% of the time. This will be explored in Section IV-D.

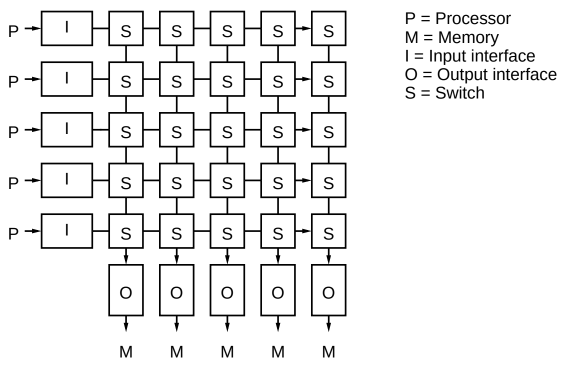

Fig. 9. The crossbar memory interconnect used in our experiments.

C. Omega Network

As an alternative to the crossbar we also implemented a model of an Omega network. The main difference is that the crossbar is a single-stage network whereas the Omega network is a multi-stage network. This means that the Omega network does not provide a direct connection between a processor and the memory. Figure 11 shows the three-stage Omega network that we used in our experiment. Messages are routed through several stages before they reach their destination. In general, there are $2 \log n$

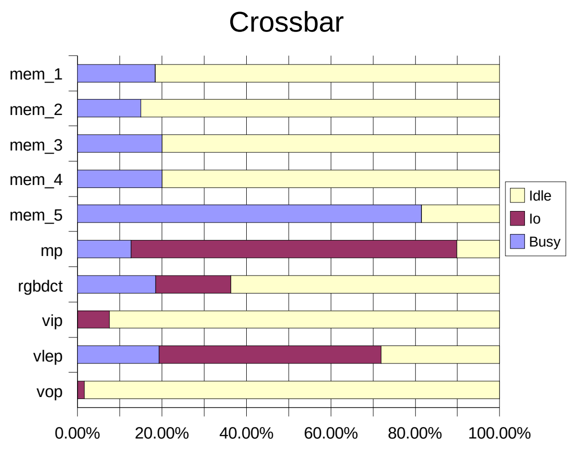

Fig. 10. Results for the crossbar showing busy/io/idle statistics for all architecture components.

stages with $n/2$ switches per stage where n is the number of inputs. The routing of the messages through the stages works as follows: The outputs are represented by binary numbers. These numbers are regarded as a string of routing information for the messages. This means that each message has a bit string attached that is processed in a bitwise fashion. At stage i the i-highest bit determines in which direction the message is routed. This is repeated for each stage so that at the final stage the destination address of the message matches the output.

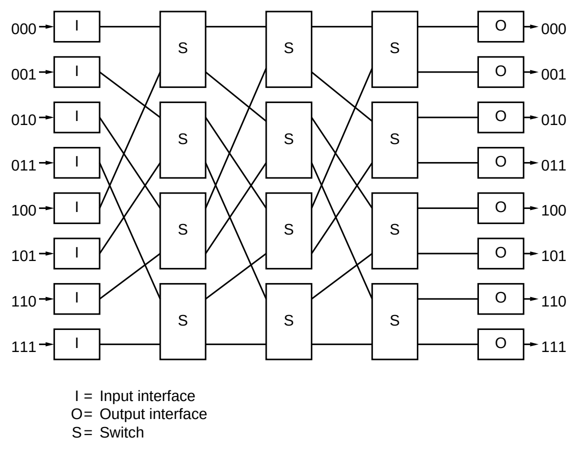

Fig. 11. The three stage Omega network used in our experiments. The stages of the network are connected in a perfect shuffle pattern.

The advantage of an Omega network is that it is cheaper to implement because it has less switches than a crossbar: it uses $n \log n/2$ switches, whereas a crossbar uses n^2 switches. The disadvantages are that the setup of a connection costs more because more switches are involved, and that it is a blocking network. The latter means that it is not always possible to connect an idle input to an idle output. The results in Figure 7 show that the Omega network is less than 1% slower than a crossbar. The detailed statistics in Figure 12 show that the microprocessor spends more time waiting for I/O compared to the crossbar. This leads to a slightly lower utilization.

The performance of the Omega network is almost the same as the performance of the crossbar. Therefore, when considering both cost and performance, the Omega network seems to be the better choice for replacing the common bus.

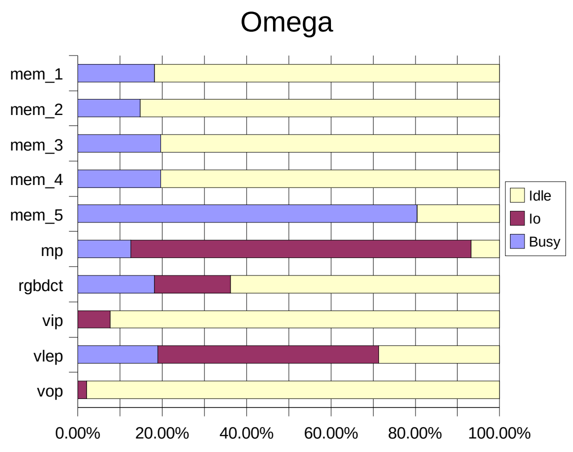

Fig. 12. Results for the Omega Network showing busy/io/idle statistics for all architecture components.

D. *Application bottlenecks*

Until now we focussed mainly on the architecture modeling side of the case study. The results of these architecture simulations indicate that, with the applied architecture and mapping, the performance is highly communication bound. However, not all the bottlenecks found in the simulation are caused by architectural constraints. While experimenting with the buffer to memory mapping for the Omega and crossbar architectures it emerged that one buffer takes 53% of memory bandwidth. This buffer contains the statistics needed for the (re)calculation of the Huffman and quantization tables as explained in [11]. For every block of image data in the M-JPEG* application, these statistics are sent from the `VLE` process to the `Q-control` process. Moreover, the calculation of the Huffman and quantization tables themselves also causes a large performance penalty that shows dramatically in run-time visualization of the simulation. During this calculation by the microprocessor, which is performed for each frame, all other components come to a halt. Only after the computation of the tables has been finished, all components start with processing again.

V. Discussion and conclusions

The main goal of this work was to study the flexibility of modeling in Sesame, which aims at the efficient design space exploration of embedded media systems architectures. More specifically, we performed a case study using a modified M-JPEG encoding application for which we investigated the efficiency of modeling and simulating several different candidate architectures. This was done by exploring three different alternatives for the memory interconnect, namely a common bus, crossbar and Omega network.

Due to the simplicity and expressive power of Sesame's Pearl simulation language, modeling and simulating the three candidate architectures was performed in only a matter of days. Pearl is an object-based language, which means that we could exploit features such as "class subtyping" to easily exchange the models for the different communication/memory architectures. Making these models a sub-type of a generic interconnect type, the models could be replaced in a plug-and-play manner.

Not only can models be constructed quickly in Pearl, but the actual simulation also takes little time. The simulation of M-JPEG* mapped onto the crossbar-based architecture takes just under 7 seconds. This was done on a Sun Ultra 5 Sparcstation with a video input stream of 16 frames of 128×128 pixels with RGB encoding.

Although model development in Sesame is already quick, we plan to make it even faster and easier by developing a library of template models for common architecture components. Evidently, such a library-based modeling approach greatly simplifies the reuse of architecture model components

References

[1] A. D. Pimentel, P. van der Wolf, E. F. Deprettere, L. O. Hertzberger, J. T. J. van Eijndhoven, and S. Vassiliadis. The Artemis architecture workbench. In *Proceedings of the Progress workshop on Embedded Systems*, pages 53–62, Oct. 2000.

[2] P. Lieverse, P. van der Wolf, E. F. Deprettere, and K. A. Vissers. A methodology for architecture exploration of heterogeneous signal processing systems. In *Proc. of the Workshop on Signal Processing Systems*, pages 181–190, Oct. 1999.

[3] A. W. van Halderen, S. Polstra, A. D. Pimentel, and L.O. Hertzberger. Sesame: Simulation of embedded system architectures for multi-level exploration. In *Proc. of the conference of the Advanced School for Computing and Imaging (ASCI)*, pages 99–106, May 2001.

[4] P. Lieverse, T. Stefanov, P. van der Wolf, and E.F. Deprettere. System Level Design with Spade: an M-JPEG case study. In *Proc. of the Int. Conference on Computer Aided Design*, November 2001.

[5] J. Rowson. Hardware/software co-simulation. In *Proc. of the Design Automation Conference*, pages 439–440, 1994.

[6] A.D. Pimentel, P. Lieverse, P. van der Wolf, L.O. Hertzberger, and E.F. Deprettere. Exploring embedded-systems architectures with artemis. *IEEE Computer*, 34(11), Nov. 2001.

[7] G. Kahn. The semantics of a simple lamguage for parallel programming. In *Proceedings of the IFIP Congress 74*, 1974.

[8] E. A. de Kock, G. Essink, W. J. M. Smits, P. van der Wolf, J. Y. Brunel, W. M. Kruijtzer, P. Lieverse, and K. A. Vissers. Yapi: Application modeling for signal processing systems. In *Proc. of the Design Automation Conference*, pages 402–405, June 2000.

[9] H. L. Muller. *Simulating computer architectures*. PhD thesis, Dept. of Computer Science, Univ. of Amsterdam, Feb. 1993.

[10] A. D. Pimentel. *A Computer Architecture Workbench*. PhD thesis, Dept. of Computer Science, Univ. of Amsterdam, Dec. 1998.

[11] Andy D. Pimentel, Berry A.W. van Halderen, Paul Lieverse, Todor P. Stefanov, and Ed F. Deprettere. The startemis case study. Technical report, University of Amsterdam, 2000.