System Level Design with Spade: an M-JPEG Case Study

Paul Lieverse[1] **Todor Stefanov**[2] **Pieter van der Wolf**[3] **Ed Deprettere**[2]

[1]Delft University of Technology
Delft, The Netherlands
lieverse@ieee.org

[2]Leiden Institute of Advanced Computer Science
Leiden, The Netherlands

[3]Philips Research Laboratories
Eindhoven, The Netherlands

Abstract

In this paper we present and evaluate the SPADE (System level Performance Analysis and Design space Exploration) methodology through an illustrative case study. SPADE is a method and tool for architecture exploration of heterogeneous signal processing systems. In this case study we start from an M-JPEG application and use SPADE to evaluate alternative multi-processor architectures for implementing this application. SPADE follows the Y-chart paradigm for system level design; application and architecture are modeled separately and mapped onto each other in an explicit design step. SPADE permits architectures to be modeled at an abstract level using a library of generic building blocks, thereby reducing the cost of model construction and simulation. The case study shows that SPADE supports efficient exploration of candidate architectures; models can be easily constructed, modified and simulated in order to quickly evaluate alternative system implementations.

Keywords

system level design, design space exploration, application modeling, architecture modeling, M-JPEG

1. Introduction

Modern signal processing systems are increasingly becoming multi-functional multi-standard systems. For example, digital televisions, set-top boxes, and mobile devices need to offer a variety of functions and must support different standards for transmission and coding of digital contents. In order to provide the required flexibility while satisfying performance requirements and constraints on cost and power consumption, such systems will be *heterogeneous systems*, i.e., systems composed of components in the range from fully programmable to fully dedicated. As technology progresses, future heterogeneous Systems-on-Chip will consist of tens of processor blocks connected by advanced communication structures and on-chip memory.

In this paper we present design technology to support the definition of heterogeneous systems architectures that satisfy the demands of a range of target applications. With the increasing system complexity it is becoming increasingly difficult to evaluate system level trade-offs using back-of-the-envelope calculations. However, building a cycle-accurate model of a complete system including the optimization and compilation of software components is a huge effort. Moreover, when building such a model a lot of details get fixed, leaving less room for optimizations. Building multiple models at this level in order to evaluate multiple candidate architectures is too costly in terms of manpower and simulation times. We therefore propose design technology that permits *system level trade-offs* to be studied at a more abstract level where design choices can be evaluated with less effort while having a high impact.

SPADE (System level Performance Analysis and Design space Exploration) [1] is a method and tool for *architecture exploration* of heterogeneous signal processing systems. The positioning of SPADE is illustrated in Figure 1. After initial back-of-the-envelope calculations, SPADE supports the construction of *abstract executable models* for the evaluation of alternative architectures. This helps to narrow the design space before proceeding to the level of cycle-accurate models.

In this paper we evaluate SPADE through a case study in which we map an M-JPEG application onto alternative multi-processor architectures. The objective of the case study is to validate the basic principles of the SPADE methodology and to verify the prototype tools and the associated library of architecture building blocks. The case study must prove that SPADE is indeed an effective tool for architecture exploration that permits alternative system implementations to be modeled and simulated efficiently.

In the next section we introduce the SPADE methodology. We discuss related work in Section 3. In Sections 4 through 7 we describe how the M-JPEG case study is modeled and executed with SPADE. This explains in more detail the SPADE concepts

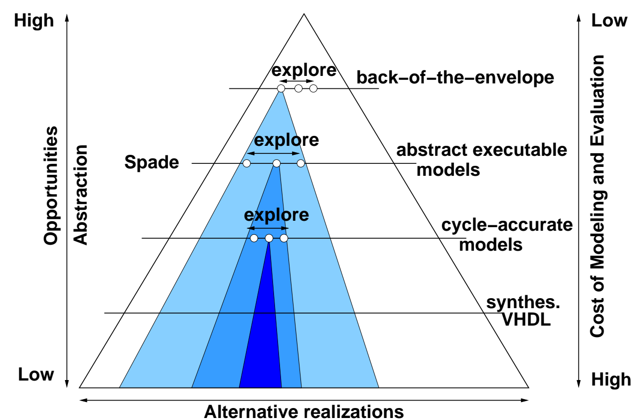

Figure 1. The design pyramid: the design space is iteratively narrowed by subsequent explorations at decreasing levels of abstraction

and how SPADE can be used in practice. In Section 8 we present a number of experiments and associated results. Finally, we draw conclusions based on our experiences with this case study.

2. The Spade Methodology

The SPADE methodology enables modeling and exploration of heterogeneous signal processing systems. SPADE is based on the *Y-chart* paradigm [2][3]. The Y-chart represents a general scheme for the design of heterogeneous systems. In the Y-chart a clear distinction is made between *applications* and *architectures*, which are related via an explicit *mapping* step. The Y-chart approach permits multiple target applications to be mapped one after another onto candidate architectures in order to evaluate their performance. The resulting performance numbers may inspire an architecture designer to improve the architecture. He may also decide to restructure the application(s) or to modify the mapping of the application(s).

The distinction between applications and architectures can be rephrased as the distinction between *workload* and *resources*. An application imposes a workload onto the resources defined by an architecture. A workload consists of *computation workload* and *communication workload*; resources can be *processing resources*, *communication resources*, and *memory resources*. The explicit mapping defines how the workload of an application is mapped onto the resources of an architecture. The architecture design process is concerned with the definition of an architecture that can best handle the workloads imposed by target applications.

SPADE provides techniques for modeling applications and architectures, as well as for capturing the mapping of application models onto architecture models. Application models can be used to verify the functional behavior of the application and to measure the workload. To obtain performance numbers we need to map such an application model onto an architecture model and evaluate the combined model. SPADE employs a *trace-driven simulation* technique to co-simulate an application model with an architecture model in order to evaluate the performance of the combined system. Trace-driven simulation techniques have been applied extensively for memory system simulation in the field of general-purpose processor design [4]. The workload of an application is captured in one or more *traces*. A trace contains symbols, called *trace entries*, that represent the computation and communication operations performed by an application upon processing a data set; data dependent behavior in the application is thus captured by these traces. The resources in an architecture accept these trace entries as the workload to be executed. The traces drive the simulation of an architecture model. The resources in an architecture model account time and report performance data for the computation and communication workload in the traces. As we go through the case study in Sections 4 through 7 we present SPADE in more detail. For more information on SPADE we also refer to [1].

3. Related Work

In the Polis [2] environment, an application is described as a network of Codesign Finite State Machines (CFSMs). This model is very well suited for reactive systems, but less suited for signal processing applications, which is the application domain targeted by SPADE.

VCC [5] has its roots in the Polis environment. It is targeted towards IP based design, and includes support for reactive, control dominated applications as well as for signal processing applications. It is also based on the Y-chart approach. VCC does not support hybrid modeling where abstract executable models can be mixed with cycle-accurate processor models for co-simulation. This is a key requirement for an IP based design environment in which abstract models can be refined incrementally by including off-the-shelf models for selected IP.

eArchitect [6] is another tool that follows the Y-chart approach. It utilizes event-based performance simulation in order to evaluate system performance. Processing components are modeled only with respect to throughput, response time, or latency, and not with respect to their actual behavior. This is similar to architecture modeling in SPADE. However, simulations in SPADE are fully functional simulations because of its application–architecture co-simulation technique, whereas the performance simulations of eArchitect are not fully functional.

Lahiri et al. [7] also use traces to capture the workload of applications. For performance analysis they use an analysis technique that manipulates the traces to derive the timing behavior of the system, whereas SPADE uses the traces to drive simulation of an architecture model. They have limited their work to bus-based communication architectures.

SystemC [8] is a C++ library and run-time environment for modeling systems both at the RT level and at more abstract levels. It has the advantage over HDLs that simulation of models is faster, and that the refinement from abstract models down to RT level models can be done in a single language and framework. Although SystemC aims at providing an extensive API for modeling at various levels of abstraction, this in itself does not yet provide a methodology for system level design.

In the Ptolemy project [9] heterogeneous modeling, simulation, and design of concurrent systems is studied, with a focus on embedded systems. Although Ptolemy offers some support for code generation, it does not offer a Y-chart approach to the definition and evaluation of communication architectures of heterogeneous systems.

4. Application

4.1 M-JPEG* Application

The application in the case study is a modified M-JPEG (Motion JPEG) encoder. We have chosen this application because it is not too complicated, but has enough features to illustrate the use and usefulness of SPADE. Like traditional M-JPEG encoders, the modified M-JPEG encoder compresses a sequence of video frames, applying JPEG [10] compression to each frame in the video sequence. M-JPEG is used for motion pictures compression like MPEG [11] but without interframe predictive coding. Our modified M-JPEG encoder, which we further refer to as M-JPEG*, differs from traditional M-JPEG encoders in three aspects.

- M-JPEG* supports only lossy encoding, whereas M-JPEG typically supports both lossy encoding and lossless encoding.

- M-JPEG* can operate on video data in both 4:2:2 YUV and RGB formats on a per-frame basis, whereas traditional M-JPEG typically uses only YUV format.

- M-JPEG* can process each incoming video frame with a different set of quantization and Huffman tables, depending on the output bit-rate and the accumulated statistics from previous video frames. Such dynamic change of the tables is typically not performed by traditional M-JPEG encoders.

The last two points imply that the behavior of M-JPEG* is dependent on the incoming video data. The M-JPEG* encoder application is depicted as a block diagram in Figure 2.

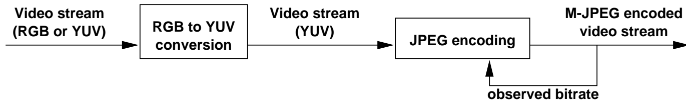

Figure 2. Block diagram of the M-JPEG* application.

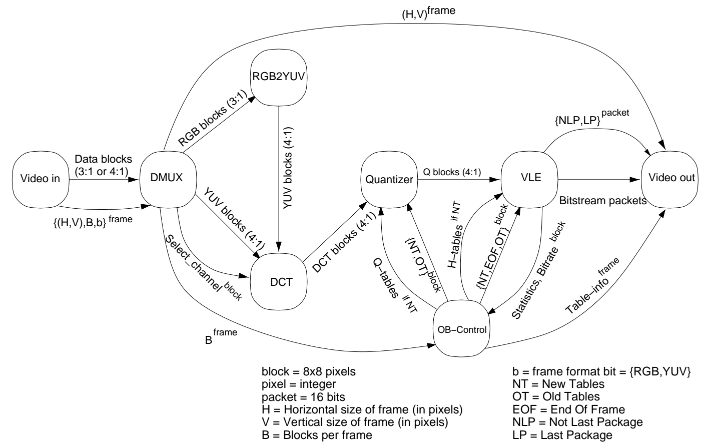

Figure 3. Structure of the M-JPEG* application model.

4.2 Application Modeling in Spade

As we are going to map the application onto a multiprocessor architecture, we have to expose task level parallelism and make communication explicit. In SPADE, we use the Kahn Process Networks [12] model of computation for application modeling. In the Kahn model, parallel *processes* communicate via unbounded FIFO *channels*. The Kahn model fits nicely with signal processing applications as it conveniently models *stream processing* and as it guarantees that no data is lost. Further, the execution of a Kahn Process Network is deterministic, meaning that for a given input always the same output is produced and the same workload is generated, irrespective of the execution schedule.

Application modeling in SPADE is done using YAPI [13]. YAPI is a simple API that can be used to structure C/C++ code as a Kahn Process Network. Upon execution of an application model, each process in the network produces a trace to capture the workload of that process. The following three API functions are provided[1].

- A `read` function. This function is used to read data from a channel via a process port. Furthermore, the function generates a *trace entry* in the trace of the process by which it is invoked, reporting on the execution of a read operation at the application level.

- A `write` function. This function is used to write data to a channel via a process port. It also generates a trace entry, reporting on the execution of a write operation.

- An `execute` function. This function performs no data processing, but is used as an annotation of computations performed by the process by which it is invoked. It generates a trace entry, reporting on processing activities at the application level. The execute function takes a *symbolic instruction* as an argument in order to distinguish between different processing activities. For example, such an instruction may correspond to a DCT operation on an eight by eight image block.

The trace entries generated by the `read` and `write` functions represent the *communication workload* of a process. The trace entries generated by the `execute` function represent the *computation workload* of a process.

[1]Note that the YAPI `select` function is not supported by SPADE.

4.3 M-JPEG* Application Model

For modeling the M-JPEG* application we started from a public domain JPEG codec implementation in C. First, we extracted the encoder part from the implementation. Then we modified it to match the M-JPEG* application. This involved the addition of an RGB to YUV conversion and of the implementation of the adaptation of the quantization and Huffman tables.

Next, we restructured this sequential implementation into a set of parallel communicating processes using YAPI. This restructuring involved, for example, removing global data structures, partitioning of the application, and insertion of calls to the YAPI functions `read` and `write`. The resulting Kahn Process Network has the structure shown in Figure 3.

The network is composed of eight processes. The Video_in, DCT, Quantizer, VLE (Variable Length Encoding), and Video_out processes together form the regular M-JPEG encoding algorithm. RGB2YUV is an additional process such that the application also accepts RGB frames as input data; the DMUX process is added to route the incoming data either directly to the DCT process or via the RGB2YUV process, depending on the incoming video format. The OB_Control process takes care of the quantization and Huffman table adaptation; it receives statistics from the VLE process and sends updated tables to both the Quantizer and the VLE processes.

Finally, we annotated the computations of each process using the YAPI `execute` function and symbolic instructions. For example, the VLE process has two `execute` calls; one with an instruction `op_VLE`, which represents all processing needed to perform the variable length encoding of an 8 by 8 block, and one with an instruction `op_MakeStatistics`, which represents the calculation of image statistics that are used in the adaptation of the quantization and Huffman tables.

4.4 Workload analysis

The M-JPEG* application model can be used for workload analysis. When it is executed, the YAPI functions `read`, `write`, and `execute` generate information on computation and communication workload of the application. For an input sequence of 8 RGB frames of size 720×576 pixels (PAL/SDTV), the workload numbers obtained are partly shown in Tables 1 and 2. Considering that all block data tokens are blocks of 8 by 8 pixels, with

Table 1. Computation workload analysis results.

Process	Instruction	number of invocations
RGB2YUV	op_RGB2YUV	51 840
DCT	op_DCT	103 680
VLE	op_VLE	103 680
	op_MakeStatistics	103 680
...	...	...

Table 2. Communication workload analysis results.

Channel			
from	to	data	number of tokens
Video_in	DMUX	block data	155 520
Video_in	DMUX	header	8
VLE	Video_out	bitstream	154 280
...	...	...	...

each pixel represented by either one or two bytes, and that the bitstream tokens are each one byte, we get an initial idea of the bandwidth needed to accommodate this amount of communication.

5. Architecture

5.1 Case Study Architecture

In the case study we are mapping the M-JPEG* application onto a heterogeneous multiprocessor architecture with centralized shared memory. The architecture is depicted in Figure 4. It consists of five processing components which are all connected to a bus and which are communicating via shared memory. For synchronization there is also some direct communication between the components; these links are not shown in this figure.

Two of the five processing components are DSPs. These DSPs are used for the computation intensive tasks, namely one DSP for the RGB to YUV conversion and the DCT transform, which we refer to as the *RGB2YUV/DCT* processor, and another DSP for the variable length encoding, which we refer to as the *VLEP*. A general purpose microprocessor (*mP*) is used for the less computation intensive quantization and for the adaptation of the quantization and Huffman tables. For the input and output processing we use two non-programmable components, which we refer to as the *VIP* (Video In Processor) and *VOP* (Video Out Processor), respectively. For the bus we initially take a 64 bit wide bus. The memory we assume to be SRAM for reasons of speed.

5.2 Architecture Modeling in Spade

In order to efficiently explore different architectures, it is required that architecture models can be easily constructed and modified. In SPADE, functional behavior is described at the application level. If this behavior is data dependent, the traces, which drive the operation of the architecture, also depend on the input data. Therefore, we can use architecture models that do not need to model the functional behavior, while maintaining correct data dependent behavior. Such architecture models can be constructed from *generic building blocks*. As the building blocks are generic, we can provide a library of such blocks. The generic building blocks need to model the different types of resources in an architecture, such as processing resources, communication resources, and memory resources. Defining an architecture then becomes as easy as instantiating building blocks from a library and interconnecting them.

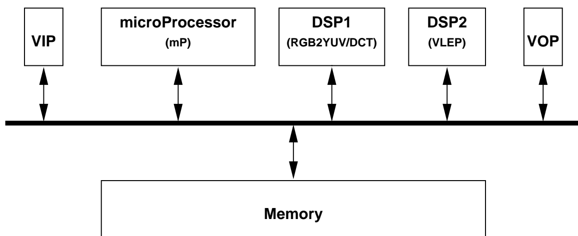

Figure 4. Abstract view of the architecture.

The processing resources in the architecture model take the traces generated by the application as an input. We have taken a modular approach to allow the construction of a great variety of processing resources from a small number of basic building blocks. A processing resource is built from the following two types of blocks.

- A *trace driven execution unit (TDEU)* which interprets trace entries. The entries are interpreted in the order in which they are put in the trace, thereby retaining the program order of the application processes. A TDEU has a configurable number of I/O ports. Communication via these I/O ports is based on a generic protocol.
- A number of *interfaces* which connect the I/O ports of a TDEU to a specific communication resource. An interface translates the generic protocol into a communication resource specific protocol, and may also include buffers to model input/output buffering present in processing resources. Currently, we have interfaces for point-to-point communication via a bus, and for communication via a bus and shared memory, both buffered and unbuffered. No interfaces are needed for communication via a FIFO or an unbuffered direct link; these communication blocks can be directly connected to the I/O ports of a TDEU.

The current library contains the TDEU and interface blocks described above, a generic bus block, including a first-come-first-serve arbiter, a FIFO block, an unbuffered direct link block, and a generic memory block. All blocks are parameterized. For each instantiated TDEU a list of *symbolic instructions* and their *latencies* has to be given. This list specifies which instructions from the traces can be executed by the processing resource and how many cycles each instruction takes when executed on this processing resource. These latencies can be obtained either from a lower level model of the processing resource, from estimation tools, or they can be estimated by an experienced designer; they reflect the processing power of the selected programmable or dedicated component. For instances of the FIFO and interface blocks, buffer sizes can be given. For a bus instance, the bus width, setup delay, and transfer delay can be specified.

An architecture is specified by means of a textual description using an architecture description language. In this description first the processors, buses, and FIFOs are defined. The user does not need to define the exact interfaces; these are inserted automatically when the architecture model is constructed. Only the parameters, such as, latencies, buffer sizes, and bus width need to be specified. The structure of the architecture is defined by describing for each FIFO and each bus which processor ports are connected to them. An example of an architecture description is given in Figure 5.

```
1    Architecture MJPEG_Arch;
     // 1 unit of size = 1 byte = 8 bits; 1 cycle = 10ns

     // Processor resources
5    Processor VIP {
         InPorts { }
         OutPorts {o1; o2;}
         Instructions {op_ElaborateFrame = 20;}
     }
10
     Processor RGB2YUVDCT {
         InPorts {i1; i2; i3; i4;}
         OutPorts {o1;o2;}
         Instructions {op_DCT = 1024; op_RGB2YUV = 192;}
15   }
             :
             :

     // Communication resources
20   Bus  B1 {width = 8; setup = 1; transfer = 2;}
     Fifo F3 {number = 4; size = 1;}
     Fifo F6 {number = 4; size = 64;}
             :
             :
25
     // Connections
     Structure {
         Bus  B1 {
             VIP.o1 {number = 1; size = 7;};
30           VIP.o2 {number = 1; size = 64;};
             RGB2YUVDCT.o1 {number = 1; size = 128;};
                         :
         }
         Fifo F3 {mP.o6  -> RGB2YUVDCT.i1;}
35       Fifo F6 {RGB2YUVDCT.o2 -> RGB2YUVDCT.i4;}
                    :
37   }
```

Figure 5. Fragment of the description of the architecture model.

5.3 Case Study Architecture Model

We modeled the architecture as described above using the library of architecture components provided by SPADE. For each processor a set of *latency values* had to be defined for the symbolic instructions used in the application processes mapped onto that processor. In order to determine realistic values for these latencies we assumed that both DSPs are Analog Devices ADSP-21160 [14] and that the general purpose microprocessor is a MIPS processor [15]. For the symbolic instructions of the processes mapped onto these three processors, low-level instruction models were constructed. Then we used the databooks of the processors to determine the latency values. For the symbolic instructions of the *VIP* and *VOP* we defined ranges of latency values to be explored.

The model was specified using the SPADE architecture description language. A fragment of the architecture description is shown in Figure 5. In the architecture model we defined one simulation cycle to be 10ns. We relate all times to this uniform time unit, even though different components may run at different clock speeds. All sizes in the architecture and the mapping description are expressed in bytes. The architecture description consists of three main parts. In the first part the processor resources are described. For example, in lines 5 to 9 of Figure 5 the *VIP* is defined; it has two output ports `o1` and `o2`, no input ports, and one symbolic instruction `op_ElaborateFrame` with a latency of 20 simulation cycles. In the second part the communication resources are specified. First the bus is specified (line 20), followed by a number of FIFO buffers which are used for synchronization. The last part of the description specifies the structure of the architecture. In this part the connections among the processor resources and communication resources are described. For example, in line 29 it is specified that output port `o1` of the *VIP* is connected to bus `B1` via a buffer with a total size of 7 bytes, and in line 34 it is specified that output port `o6` of the *mP* is connected to input port `i1` of the *RGB2YUV/DCT* processor via FIFO `F3`.

6. Mapping

6.1 M-JPEG* Mapping Specification

In Section 5.1 we already discussed the mapping of the processing workload of the M-JPEG* application onto the processing resources in the architecture. Also, we mentioned that most of the communication is done via shared memory. Only some synchronization and communication of status flags is mapped onto direct channels between the processors.

6.2 Mapping in Spade

With SPADE, mapping of an application model onto an architecture model is performed as follows.

- Each process is mapped onto a TDEU. This mapping can be many-to-one, in which case the trace entries of the processes need to be scheduled by the TDEU. SPADE provides a default round-robin scheduler and also provides an API for modeling and using user-defined schedulers.
- Each process port is mapped one-to-one onto an I/O port. This mapping also implicitly maps the channels onto a combination of communication resources and memory resources.

If it appears that the functionality of a single process needs to be distributed over more than one processing resource, then the designer first has to rewrite the application such that this process is partitioned into two or more processes.

Like the architecture, the mapping is specified by means of a textual description using a mapping description language.

6.3 M-JPEG* Mapping Description

Using the SPADE mapping description language we have specified the mapping that we described in Section 6.1. A fragment of the mapping description is shown in Figure 6; the mapping of the processes onto the processors is also illustrated in Figure 7. The first part of the description specifies the mapping of the processes and their ports onto the processor components and their ports. For instance, in lines 3 to 6 it is specified that process Video_in is mapped onto the *VIP* and that the ports `out_HeaderInfo` and `out_BlockData` of the Video_in process are mapped onto the ports `o1` and `o2` of the *VIP*, respectively. Next, for each application channel a *tokensize* is specified. If the channel is mapped onto a bus and the communication should take place via shared memory, then the number of places and the size of each place of the buffers in shared memory are specified. For example, in lines 28 to 31 it is specified that the tokens that are transfered via channel `DCT_Q_BlockData` have a size of 128 bytes, and that 4 buffers of 128 bytes are allocated in shared memory for this channel. The last part of the mapping description specifies the type of the schedulers. In lines 37 and 38 it is specified that for both the microprocessor and the *RGB2YUV/DCT* processor the default scheduler is selected.

7. Simulation and Performance Evaluation

As we already described in Section 2, SPADE employs a trace driven simulation technique to co-simulate an application model with an architecture model. The simulation of the application

```
1   Mapping MJPEG_Map (MJPEG_Appl, MJPEG_Arch);

    Video_in : VIP {
        out_HeaderInfo : o1;
5       out_BlockData  : o2;
    }

    DCT : RGB2YUVDCT {
        in_BlockData1 : i4;
10      in_BlockData2 : i3;
        in_BlockType  : i1;
        out_BlockData : o1;
    }

15  RGB2YUV : RGB2YUVDCT {
        in_BlockData  : i2;
        out_BlockData : o2;
    }
          :
20        :

    Channels {
        Video_DMUX_Header {
            tokensize = 7;
25          numbermembufs = 1; membufsize = 7;
        };
        RGB2YUV_DCT_BlockData {tokensize = 64;};
        DCT_Q_BlockData {
            tokensize = 128;
30          numbermembufs = 4; membufsize = 128;
        };
                :
                :
    }
35
    Schedulers {
        mP          : default { };
        RGB2YUVDCT  : default { };
39  }
```

Figure 6. Fragment of the description of the mapping shown in Figure 7

model is based on the Pamela [16] multi-threading environment, where each Kahn process is executed in a separate thread. The simulation of the architecture model is currently based on TSS (Tool for System Simulation), which is a Philips in-house architecture modeling and simulation framework.

In order to evaluate a system, the SPADE library blocks from which an architecture is built collect performance data during simulation. From this data *performance metrics*, such as throughput, frame rate, overall latency, and bus utilization, can be calculated. These metrics give an indication of the performance of the system. For example, for a video processing system, such as the M-JPEG* case study, we can collect the times at which a new frame is output; from those times we can calculate the frame rate of the system.

The following data is currently collected in the library blocks. Each TDEU keeps track of the number of cycles it was busy with computations, the number of cycles it was doing I/O, split out into reads and writes, the number of cycles it was waiting either for data or for room, and the number of cycles it had nothing to do at all, i.e., was idle. In addition, each input and output port counts the number of reads or writes that were performed, plus the number of cycles no room or data was available. Each bus counts the number of cycles it was in use and the amount of data transported.

8. Experiments and Results

In this section we present some of the experiments we have done and demonstrate that SPADE can be used effectively to evaluate the performance of alternative architectures.

In the experiments we look at the *throughput* at the output of the M-JPEG* system as the main performance metric. We measure this throughput in *frames per second*. The throughput depends on the parameters of the architecture, the size of the incoming frames and the format, YUV or RGB, of the frames. We

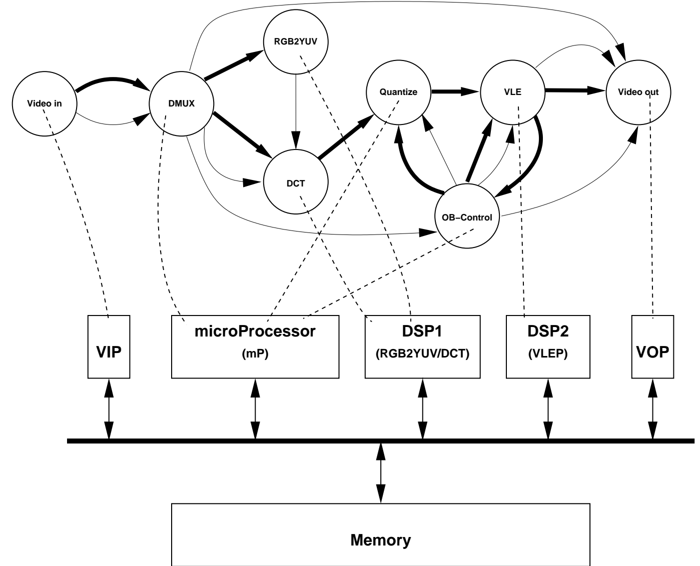

Figure 7. Mapping of the M-JPEG* application onto the proposed architecture.

study the effects of the following architecture parameters: the number of processing components, the latencies of these components, the speed of the bus, and the speed and size of shared memory. We have performed the experiments with sequences of RGB frames. For YUV frames the throughput will be at least as good as the throughput for RGB frames, as no RGB to YUV conversion needs to be performed.

The initial speed and width of the bus have been set to 100MHz and 64 bits, respectively. For the shared memory we selected an SRAM-type memory of size 64KB with read and write cycles of 10ns each.

We explore two different scenarios. The difference between the two scenarios is the required throughput. For Scenario 1 the required throughput is 25 CIF frames (352×288 pixels) per second. For Scenario 2 the required throughput is 25 PAL/SDTV frames (720×576 pixels) per second.

Initial simulations showed that for the application, architecture, and mapping as described in the previous sections the throughput is 27 CIF frames or 6.5 PAL/SDTV frames per second. The utilizations of the processing components are shown in Figure 8. These results are used as the starting point for the two exploration scenarios.

8.1 Scenario 1

In Scenario 1 the initial throughput of 27 CIF frames per second is already better than the required throughput of 25 CIF frames per second. Therefore we focus the exploration on improving the *performance–cost* ratio of the system. This exploration includes finding and removing redundant components and excess speed while still satisfying the required throughput. As the initial throughput is just above the required 25 frames per second, we can only improve the performance–cost ratio by a reduction in terms of cost, e.g., silicon area or power consumption.

The performance numbers in Figure 8 suggest that the given architecture has a poor load balance because the *mP* and the *VLEP* are not utilized very well. The DSP which we have chosen for

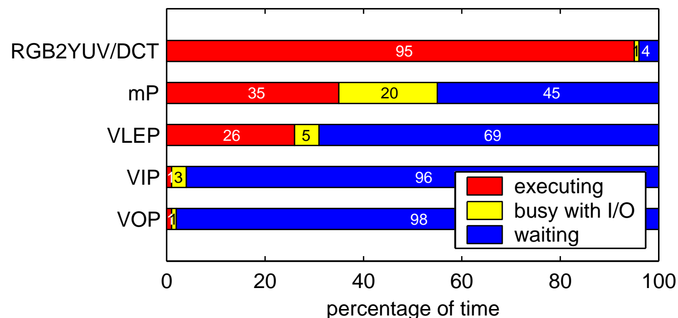

Figure 8. Utilizations of the processing components of the initial architecture. The bus utilization is 40%.

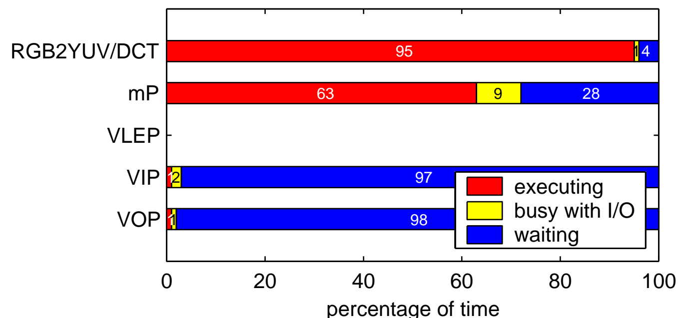

Figure 9. Utilizations of the processing components after removing the VLEP. The bus utilization is 12%.

the *VLEP* is too powerful for just the VLE process. That is why it is waiting most of the time (69%). The *mP* is executing and busy with I/O only 55% of the time. Taking these observations into account we conclude that we might not need a separate processor component for the run length encoding and Huffman encoding. We decide to remove the *VLEP* and to map the VLE process onto the *mP*. We assume that the Quantizer and VLE processes can be merged in such a way that there is no need to explicitly store intermediate results in shared memory; this could, for example, be accomplished by a fusion of the loops associated with the two processes.

We simulated this modified architecture in order to see how the performance has changed. Figure 9 presents the new performance numbers. The throughput is still 27 CIF frames per second. This means that removing the *VLEP* reduces the cost in terms of silicon area without a penalty on performance. The new simulation results show that the bus is utilized only 12% instead of 40% in the initial simulation. This reduction is a result of the assumption that the data from Quantizer to VLE does not need to go to shared memory any more. The low bus utilization means that the performance of the architecture is probably not very sensitive to a decrease of the speed of the bus and the shared memory. We observed through SPADE simulation that if we decrease the speed of the bus and the memory five times, then the throughput only drops to 26 CIF frames per second. The cost reduction we achieve is in terms of silicon area, mainly as we now can use DRAM instead of SRAM.

8.2 Scenario 2

In Scenario 2, the initial throughput of 6.5 SDTV frames per second is well below the required throughput of 25 SDTV frames

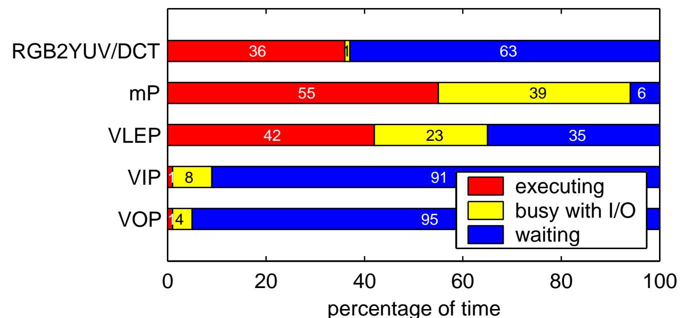

Figure 10. Utilizations of the processing components after decreasing the latency of the DCT transform operation on the RGB2YUV/DCT processor. The bus utilization is 62%.

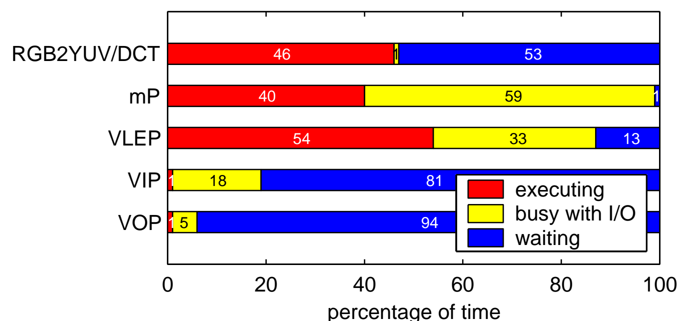

Figure 11. Utilizations of the processing components after decreasing the latency of the microprocessor. The bus utilization is 81%.

per second. The focus of this scenario is thus on improving the performance.

The SPADE performance numbers in Figure 8 show that the *RGB2YUV/DCT* processor is executing 95% of the time while the other components execute less than 36% of the time. This suggests that the throughput is very sensitive to the latency of the *RGB2YUV/DCT* processor. We decrease the latency of the DCT transform executed on this component five times; such a decrease can be obtained by designing a dedicated component, in which we exploit all parallelism present in the DCT algorithm. With this improved *RGB2YUV/DCT* processor the throughput becomes 10 SDTV frames per second. This still does not give us the required performance. The performance numbers obtained from this simulation are shown in Figure 10. The results show that the *RGB2YUV/DCT* processor is no longer the bottleneck. The results also show that we can expect an increase of the throughput if we decrease the latency of the microprocessor. We decreased the latency of the *mP* two times. For this case, we obtain a throughput of 14 SDTV frames per second. The performance numbers are given in Figure 11.

From this exploration we can see that the improvement of the processing components, i.e., the *RGB2YUV/DCT* processor and the microprocessor, results in a doubling of the throughput. However, it is still about a factor two below the required throughput of 25 SDTV frames per second. Decreasing the latencies of the components further is not an option, since such high performance components cannot be implemented. Also, we do not expect a significant speedup of the architecture, because Figure 10 and Figure 11 show a significant increase of the time the proces-

sor components spend on performing I/O operations. The latter means that the communication structure of the architecture becomes a bottleneck.

According to the results of the exploration we conclude that although our architecture consists of five fast processor components working in parallel, we cannot achieve the required real-time performance for SDTV frames.

9. Conclusions

We have presented a case study of an M-JPEG encoder application mapped onto a shared memory multi-processor architecture. We performed explorations of the initial application and architecture at an abstract level using SPADE. By doing these experiments we illustrated the use and usefulness of SPADE early in the process of designing heterogeneous signal processing architectures. It appears that SPADE gives a designer useful feedback on the performance of a system, which helps him in improving the system. This was illustrated by the scenarios in Section 8. More specifically we conclude that:

- The Y-chart approach helps in separating application and architecture concerns, which is crucial if multiple architectures are to be evaluated, possibly for multiple applications.
- The Kahn Process Networks model is effective for application modeling. Using a simple API, legacy C code can be transformed into a parallel model.
- Thanks to the trace-driven simulation technique of SPADE, non-functional architecture models can be used even for data dependent applications. As a consequence we can use a library of generic building blocks for architecture modeling. This turned out to be a key feature for efficient architecture modeling.
- SPADE permits architecture models to be parameterized, which helps in performing sensitivity analysis and tuning of architecture parameters.
- SPADE supports efficient exploration. The simulations done in the scenarios typically took a few minutes for an input sequence of several frames. Also, changes to the architecture and mapping could be easily made to the textual descriptions.
- SPADE currently lacks feedback on other metrics than performance metrics, such as silicon area and power dissipation.

An issue that will be subject of further research is the coupling to more detailed abstraction levels. Since SPADE uses the TSS cycle-driven simulation engine for architecture simulation, we think it is very well possible to enable co-simulation of abstract architecture models with detailed, cycle-accurate models, thereby providing a path from abstract level simulation and exploration to detailed level simulation in a single framework. We are also studying the use of SystemC for architecture modeling in the SPADE methodology.

Acknowledgments

This work was performed in part in the Artemis project (AES 5021), funded by STW under the Progress program.

References

[1] Paul Lieverse, Pieter van der Wolf, Ed Deprettere, and Kees Vissers, “A methodology for architecture exploration of heterogeneous signal processing systems,” in *Proc. 1999 Workshop on Signal Processing Systems (SiPS’99)*, Taipei, Taiwan, Oct. 20-22 1999, pp. 181–190.

[2] F. Balarin, E. Sentovich, M Chiodo, P. Giusto, H. Hsieh, B Tabbara, A. Jurecska, L. Lavagno, C. Passerone, K. Suzuki, and A. Sangiovanni-Vincentelli, *Hardware-Software Co-design of Embedded Systems – The POLIS approach*, Kluwer Academic Publishers, 1997.

[3] B. Kienhuis, E. Deprettere, K. Vissers, and P. van der Wolf, “An approach for quantitative analysis of application-specific dataflow architectures,” in *Proc. ASAP’97*, July 14-16 1997.

[4] Richard A. Uhlig and Trevor N. Mudge, “Trace-driven memory simulation: A survey,” *ACM Computing Surveys*, vol. 29, no. 2, pp. 128–170, June 1997.

[5] Grant Martin and Bill Salefski, “Methodology and technology for design of communications and multimedia products via system-level IP integration,” in *Proc. DATE’98 Designers’ Forum*, 1998, pp. 11–18.

[6] `http://innoveda.com/product/eArchitect`, Innoveda Inc.

[7] Kanishka Lahiri, Anand Raghunathan, and Sujit Dey, “Fast performance analysis of bus-based system-on-chip communication,” in *Proc. ICCAD’99*, San Jose, CA, Nov. 7–11 1999, pp. 566–572.

[8] Stan Liao, Steve Tjiang, and Rajesh Gupta, “An efficient implementation of reactivity for modeling hardware in the Scenic design environment,” in *Proc. DAC’97*, Anaheim, CA, June 9-13 1997, pp. 70–75.

[9] Edward A. Lee et al., “Overview of the Ptolemy project,” ERL Technical Report UCB/ERL M99/37, University of California, Berkeley, July 1999, `http://ptolemy.eecs.berkeley.edu`.

[10] W.B. Pennebacker and J.L. Mitchel, *JPEG Still Image Data Compression Standard*, Van Nostrand Reinhold, New York, 1993.

[11] W.B. Pennebacker, J.L. Mitchel, C.E. Fogg, and D.J. LeGall, *MPEG Video Compression Standard*, Chapman and Hall, 1996.

[12] Gilles Kahn, “The semantics of a simple language for parallel programming,” in *Proc. of the IFIP Congress 74*. 1974, North-Holland Publishing Co.

[13] 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. DAC’2000*, Los Angeles, CA, June 5-9 2000, pp. 402–405.

[14] Analog Devices Inc., *ADSP-21160 Preliminary Data Sheet*, Jan. 2000.

[15] `http://www.mips.com/products`, MIPS Technologies Inc.

[16] A.J.C. van Gemund, “Performance prediction of parallel processing systems: The PAMELA methodology,” in *Proc. 7th ACM Int. Conference on Supercomputing*, Tokyo, July 1993, pp. 318–327.