Towards Efficient Design Space Exploration of Heterogeneous Embedded Media Systems

A.D. Pimentel, S. Polstra, F. Terpstra,
A.W. van Halderen, J.E. Coffland, and L.O. Hertzberger

Dept. of Computer Science, University of Amsterdam
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
{andy,spolstra,ftrpstra,berry,jcofflan,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.
As part of the Artemis project, we are developing a modeling and simulation environment which aims at efficient design space exploration of heterogeneous embedded systems architectures. In this paper, we present an overview of the modeling and simulation methodology used in Artemis. Moreover, using a case study in which we have applied an initial version of our prototype modeling and simulation environment to an M-JPEG encoding application, we illustrate the ease with which alternative candidate architectures can be modeled and evaluated.

1 Introduction

Modern embedded systems, like those for media and signal processing, increasingly need to be multifunctional and must support multiple standards. A high degree of *programmability*, which can be provided by applying microprocessor technology as well as reconfigurable hardware, is key to the development of such advanced embedded systems. However, performance requirements and constraints on cost and power consumption still require substantial parts of these systems to be implemented in dedicated hardware blocks. As a result, modern embedded systems often have a *heterogeneous system architecture*, i.e., they consist of components in the range from fully programmable processor cores to dedicated hardware components for the time-critical application tasks. Increasingly, such heterogeneous systems are integrated on a single chip. This yields heterogeneous multi-processor systems-on-a-chip (*SoC*s) that exploit task-level parallelism in applications.

For these modern embedded systems, it becomes more and more important to have good tools available for exploring different design choices at an early stage in the design. This is because the heterogeneity of the embedded systems and the varying demands of their target applications greatly complicate the system design, which already affects the very first design decisions. Common simulation practice for the design space exploration of heterogeneous embedded systems architectures is unable to cope with this increase in complexity and is especially becoming unsuited for the early design stages. Designers typically use only relatively detailed, often cycle-accurate, simulators for design space exploration of embedded systems architectures. For complex embedded systems, the effort required to build such detailed simulators can be high, making it impractical to use those simulators in early design stages. Moreover, their low simulation speeds significantly hamper the architectural exploration.

In the scope of the Artemis (ARchitectures and meThods for Embedded MedIa Systems) project [17], we are developing an architecture workbench which provides modeling and simulation methods and tools for the efficient design space exploration of heterogeneous embedded multimedia systems [16]. This architecture workbench should allow for rapid evaluation of different architecture designs, application to architecture mappings, and hardware/software partitionings and it should do so at multiple levels of abstraction *and* for a wide range of multimedia applications. By allowing simulation at multiple abstraction levels, the speed, required modeling effort, and attainable accuracy of the architecture simulations can be controlled. This enables a stepwise refinement approach in which abstract simulation models are used to efficiently explore the large design space in the early design stages, while in a later stage more detailed models can be applied for focused architectural exploration.

Another important requirement for our architecture design space exploration environment is that it should be open to reuse of intellectual property, thereby allowing for reducing the time-to-market of products. For example, simulation models of architecture components, such as microprocessors, busses and memories, must be reusable with relative ease. This calls for a high degree of modularity when building system architecture models and, as we show later on, a clear separation between specifying application behavior and architectural performance constraints.

In this paper, we present an overview of the modeling and simulation methodology used in Artemis and the open research problems it addresses. Using a case study with an M-JPEG encoding application we illustrate the ease with which different architectural design choices can be evaluated at a high level of abstraction. To this end, we have used an initial version of our prototype modeling and simulation environment, called Sesame, to evaluate three alternative multi-processor target architectures with different memory interconnects.

The next section describes how Artemis relates to other efforts in the field of simulation of embedded systems architectures. In Section 3, we describe the modeling and simulation methodology applied in Artemis. In Section 4, the Sesame modeling and simulation environment is described. Section 5 presents the case study with an M-JPEG application and Section 6 concludes the paper.

2 The limitations of traditional co-simulation

System architecture modeling and simulation of heterogeneous systems is a relatively new research field which has received a lot of attention in recent years. The key concept in most efforts in this field is *co-simulation*. Like its name already suggests, co-simulation implies that the software parts (which will be mapped onto a programmable processor) and the hardware components and their interactions are simulated together in one simulation [18]. Traditional co-simulation frameworks (e.g., Seamless VCE [11], Virtual CPU [2], and the work of [7, 4]) often combine two simulators, one for simulating the programmable components running the software and one for the dedicated hardware. For software simulation, instruction-level processor simulators, host code execution or bus-functional processor models [18] are typically used. To simulate the hardware components, HDLs such as VHDL are a popular choice.

A major drawback of such co-simulation is its inflexibility: because an explicit distinction is made between software and hardware simulation, it must already be known which application components will be performed in software and which ones in hardware before the system model is built. This significantly complicates the performance evaluation of different hardware/software partitionings since a whole new system model may be required for the assessment of each partitioning. For this reason, the co-simulation stage is often preceded by a stage in which the application is studied in isolation by means of a functional (behavioral) software model written in a high level language. This typically results in rough estimations of the application's performance requirements, which are subsequently used as guidance for the hardware/software partitioning. In that case, the co-simulation stage is mainly used as verification of the chosen hardware/software partitioning and not as a design space exploration vehicle.

A number of exploration environments, such as VCC [1], Polis [3] and eArchitect [2], facilitate more flexible system-level design space exploration by providing support for mapping a behavioral application specification to an architecture specification. However, in contrast to these efforts, Artemis pushes the separation of modeling application behavior and modeling architectural constraints at the system level to its extremes. As will be shown in the next section, such separation leads to efficient exploration of different design alternatives while also yielding a high degree of reusability.

3 Modeling and simulation methodology

We strongly believe that for the design of programmable embedded systems a clear distinction should be made between *applications* and *architectures*, and that an explicit *mapping* step must be supported. This permits multiple target applications to be mapped one after the other onto candidate architectures for evaluation of their performance. This approach is referred to as the *Y-chart* of system design [10, 3]. Typically, the designer studies the target applications, makes some initial calculations, and proposes an architecture. The performance of this architecture is then quantitatively evaluated and compared against alternative architectures. For such performance analysis, each application is mapped onto the architecture under investigation and the performance of each application-architecture combination is evaluated. Subsequently, the resulting performance numbers may inspire the designer to improve the architecture, restructure the application(s) or modify the mapping of the application(s).

The Artemis modeling and simulation environment facilitates the performance analysis of embedded systems architectures in a way that directly reflects the Y-chart design approach: Separate application and architecture models are recognized for system simulation. An *application model* describes the functional behavior of an application, including both computation and communication behavior. The *architecture model* defines architecture resources and captures their performance constraints. Essential in this modeling methodology is that an application model is independent from architectural specifics, assumptions on hardware/software partitioning, and timing characteristics. As a result, a single application model can be used to exercise different hardware/software partitionings and can be mapped onto a range of architecture models, possibly representing different system architectures or simply modeling the same system architecture at various levels of abstraction. This clearly demonstrates the strength of decoupling application models and architecture models: it enables the *reuse* of *both* types of models. After mapping, an application model is co-simulated with an architecture model allowing for evaluation of the system performance of a particular application, mapping, and underlying architecture.

3.1 Trace-driven co-simulation

To co-simulate application models and architecture models, an interface between the two must be provided, including a specification of the mapping. For this purpose, we apply trace-driven simulation. In our approach, the application model is structured as a network of *concurrent communicating processes*, thereby expressing the inherent task-level parallelism available in the application and making communication explicit. Each process, when executed, produces a trace of events which represents the application workload imposed on the architecture by that particular process. Thus, the trace events refer to the computation and communication operations performed by an application process. These operations may be coarse grain, such as "*compute a Discrete Cosine Transform (DCT)*".

Since application models represent the functional behavior of applications, the traces correctly reflect data dependent behavior. Consequently, the architecture models, which are driven by the application traces, do not need to represent functional behavior but only need to account for the performance consequences of the application events.

3.2 Application modeling

For modeling of applications, we use the Kahn Process Network (KPN) model of computation [9]. To obtain a Kahn application model, a sequential application (written in a high-level language) is restructured into a program consisting of parallel processes communicating with each other via unbounded FIFO channels. In the Kahn paradigm, reading from channels is done in a blocking manner, while writing is non-blocking.

The computational behavior of an application can be captured by instrumenting the code of each Kahn process with *annotations* which describe the application's computational actions. The reading from or writing to Kahn channels represents the communication behavior of a process within the application model. By executing the Kahn model, each process records its actions in order to generate a trace of application events, which is necessary for driving an architecture model.

In the field of application modeling, a lot of research has been done on models of computation (e.g., [6]). We decided to use KPNs for application modeling because they fit nicely with the multimedia application domain and are deterministic. The latter means that the same application input always results in the same application output, i.e., the application behavior is architecture independent. This automatically guarantees the validity of event traces when the application and architecture simulators are executed independently of each other [8]. However, because of the semantics of KPNs which disallow, for example, the modeling of interrupts, we are currently not able to model applications with time dependent behavior.

A beneficial side effect of using a separate application model is that it also makes it possible to analyze the computational/communication needs and the potential performance constraints of an application in isolation from any architecture. This can be a benefit as it allows for investigation of the upper bounds of the performance and may lead to early recognition of bottlenecks within the application itself.

3.3 Architecture modeling

A model of an architecture is based on components representing (co)processors, memories, buffers, busses, and so on. A performance evaluation of an architecture can be achieved by simulating the performance consequences of the incoming computation and communication events from an application model. This requires an explicit mapping of the processes and channels of a Kahn application model onto the components of the architecture model. The generated trace of application events from a specific Kahn process is routed towards a specific component inside the architecture model by using a trace-event queue. The Kahn process dispatches its application events to this queue while the designated component in the architecture model consumes them. This is illustrated in Figure 1. Mapping the FIFO channels between Kahn processes (shown by the dashed arrows) defines which communication medium at the architecture level is used for the data exchanges. In Figure 1, one application channel stays unmapped since both its application tasks are mapped onto the same processing component. Mapping the trace-event queues from multiple Kahn processes onto a single architecture component occurs when, for example, several application tasks are executed on a microprocessor. In this case, the events from the different queues need to be scheduled.

We reiterate that the underlying architecture model solely accounts for architectural (performance) constraints and therefore does not need to model functional behavior. This is possible because the functional behavior is already captured in the application model, which subsequently drives the architecture simulation. An architecture model is constructed from generic building blocks provided by a library. This library contains performance models for processing cores, communication media (like busses) and different types of memory. Evidently, such a library-based modeling approach greatly simplifies the reuse of architecture model components.

At a high level of abstraction, the model of a processing core is a *black box* which can model timing behavior of a programmable processor, a reconfigurable component or a dedicated hardware unit. Modeling such a variety of architectural implementations is accomplished by the fact that the architecture simulator assigns parameterizable latencies to the incoming application events. For example, to model software execution of an application event, a relatively high latency can be assigned to the event. Likewise, to model the application event being executed by dedicated or reconfigurable hardware one only needs to tag the event with a lower latency. So, by simply varying the latencies for computational application events, different hardware/software partitionings can

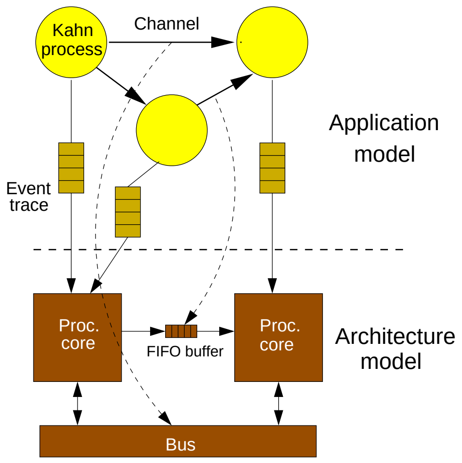


Fig. 1. Mapping a Kahn application model onto an architecture model.

rapidly be evaluated at a high level of abstraction. The latencies themselves can be obtained either from a lower level model of an architecture component, from performance estimation tools, or they can be estimated by an experienced designer.

In this approach, the communication events from the application model are used for modeling the performance consequences of data transfers and synchronizations at the architecture level. These events cause the appropriate communication component within the architecture model (onto which the communicating Kahn channel is mapped) to account for the latencies associated with the data transfers. Unlike in the application model where all FIFO channels are unbounded, writes at the architecture level may also be blocking dependent on the availability of resources (e.g., buffer space).

As design decisions, such as hardware/software partitioning, are made, components of the architecture model may be refined. This implies that the architecture model starts to reflect the characteristics of a particular implementation (e.g., dedicated versus programmable hardware). To facilitate the process of model refinement, the architecture model library should include models of common architecture components at several levels of abstraction. For example, there may be multiple instances of a microprocessor model such as a black box model, a model which accounts for the performance consequences of the processor's memory hierarchy (e.g., translation lookaside buffers and caches), and a model which accounts for the performance impact of both its memory hierarchy and datapath (e.g., pipelining and instruction-level parallelism). Moreover, to support architecture model refinement, events from the application model should also be refined to match the level of detail present in the architecture model. Providing flexible support for such event refinement is still largely an open problem [13, 5].

The process of model refinement may continue to the level at which detailed simulators for certain architecture components, e.g., instruction-level simulators or Register Transfer Level (RTL) simulators, are embedded into the overall system architecture simulation. For instance, consider the example in which it is decided that a certain application task will be implemented in software. In that case, instead of using an abstract architecture model of a processor core onto which the Kahn process in question is mapped, a detailed instruction-level simulator can be used which emulates the actual code of the application task. The process of embedding more detailed simulators can be continued such that more and more functionality is gradually incorporated into an architecture model. In the end, the architecture model can then be used as a starting point for more traditional hardware/software co-simulation composed of instruction-level simulators and RTL simulators.

4 The Sesame modeling and simulation environment

For the development of the Artemis architecture modeling and simulation environment, we currently are developing and experimenting with two prototype simulation frameworks: *Spade* (System-level Performance Analysis and Design space Exploration) [14] and *Sesame* (Simulation of Embedded System Architectures for Multi-level Exploration) [20]. Both frameworks act as technology drivers in the sense that they allow for testing and evaluating new simulation models and simulation methods to gain insight into their suitability for the Artemis modeling and simulation environment. Only those simulation models and simulation methods that have proven to be effective will be incorporated in Artemis. In this paper, we limit our discussion to Sesame only.

The Sesame framework aims at studying the potentials of simulation at multiple levels of abstraction and the concepts needed to refine simulation models across different abstraction levels in a smooth manner. For example, refinement of one component in an architecture model should not lead to a completely new implementation of the entire model. This means that the modeling concepts being studied should also include support for refining only parts of an architecture model, thus creating a *mixed-level simulation model*. The resulting mixed-level simulations allow for more detailed evaluation of a specific architecture component within the context of the behavior of the whole system. They therefore avoid the need for building a complete detailed architecture model during the early design stages. Moreover, mixed-level simulations do not suffer from deteriorated system evaluation efficiency caused by unnecessarily refined parts of the architecture model.

Sesame currently only provides a library of black box architecture models. In the near future, the library will be extended with models for architecture components at several levels of abstraction in order to facilitate the performance evaluation of architectures from the black box level towards the level of cycle-accurate models. This library will eventually be supplemented with techniques and tools to assist the modeler in gradually refining the models and performing mixed-level simulations. Currently, these issues are still largely open research problems.

The architecture models in Sesame are implemented using a small but powerful discrete-event simulation language, called Pearl, which provides easy construction of the models and fast simulation [15]. Evidently, these characteristics greatly improve the scope of the design space that can be explored in a reasonable amount of time. The architecture library components in Sesame are not meant to be fixed building blocks with pre-defined interfaces but merely template models which can be freely extended and adapted. With this approach, a high degree of flexibility is achieved (which can

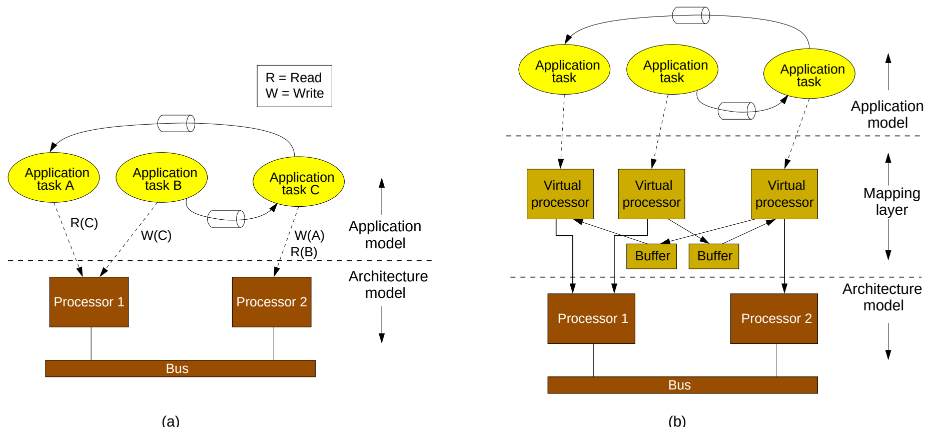


Fig. 2. Figure (a) shows a potential deadlock situation due to scheduling of communication events. The Sesame solution, using virtual processors, is illustrated in Figure (b).

be helpful when refining models) at the cost of a slightly increased effort required to build architecture models. This effort will however still be relatively small due to the modeling ease provided by Pearl.

As we have described, multiple Kahn processes of the application model can be mapped onto a single processing component in the architecture model. 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 2(a). 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 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. As a result, the processors block.

In Figure 2(b), Sesame's solution to the above problem is depicted. Between the application and architecture layers, we distinguish an extra mapping layer. This mapping layer, which is implemented in the Pearl language and which can be automatically generated from an application model, consists of virtual processor components and FIFO buffers for communication between the virtual processors. A virtual processor reads in an application trace from a Kahn process and dispatches the events to a processing component in the architecture model. The mapping of a virtual processor onto a processing component in the architecture model is parameterized and thus freely adjustable. The FIFO buffers in the mapping layer have a one-to-one relationship with the FIFO channels in the Kahn application model but they are limited in size. Their size is parameterized and dependent on the modeled architecture.

As can be seen from Figure 2(b), 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 processor model. The latter subsequently schedules the events in a FCFS fashion and models their timing consequences. However, for communication events, the appropriate buffer at the mapping layer is first consulted to check whether or not a communication is safe to take place. Only 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.

5 The M-JPEG* case study

To demonstrate the flexibility of modeling in Sesame we have applied its current version to a modified M-JPEG encoding application, referred to as M-JPEG*. This application has already been studied in the scope of the Spade environment [12, 19], which demonstrated that the modeling and simulation methodology of Artemis facilitates efficient evaluation of different application to architecture mappings and hardware/software partitionings. In this section, we use the Sesame environment to show the capability to quickly evaluate different architecture designs.

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 3.

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. 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.

In M-JPEG*, the tables for Huffman encoding[1] 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.

5.1 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 (assumed to be a MIPS R4000), two are DSPs (assuming Analog Devices' ADSP-21160s) and two are

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

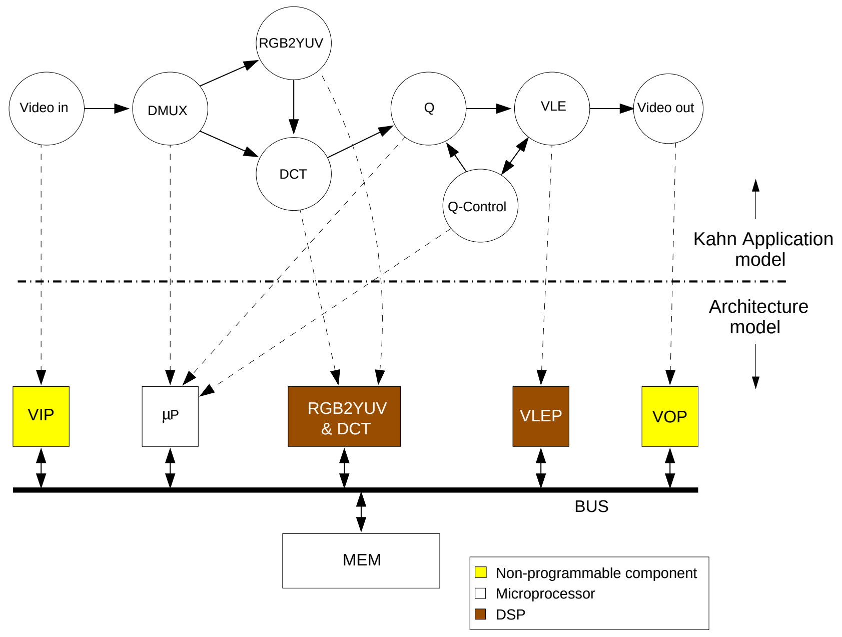


Fig. 3. M-JPEG*'s application to architecture mapping

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 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. A detailed description of the M-JPEG* application, its application model, and the base architecture model can be found in [19].

To demonstrate the ease of modeling in Pearl (Sesame's simulation language), Figure 4 shows the Pearl code of the bus model for the M-JPEG* target architecture. This bus model simulates transactions at the granularity of message transfers of abstract data types. Extending this model to account for 64-bit transactions is trivial. 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, while 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

```
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.

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. In our example, the synchronous message passing also causes the virtual clock to advance in time, because the memory object accounts for the time it takes to retrieve the requested data before replying to the bus. After having received a reply from the memory object, the bus itself executes a `reply()` to return control to one of the processor objects (which are connected to the bus object) that has called the `load` method. At the bottom of Figure 4 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).

In the bus model of the M-JPEG* case study, we have not explicitly modeled bus arbitration. Instead, we use Pearl's internal scheduling, which applies a FCFS strategy to incoming method calls for the bus object. We note, however, that an arbiter component which implements other strategies than FCFS can be added to the model with relative ease.

In the Pearl 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. For the purpose of brevity, we left out a number buffer specifications. 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.

```
commonbus() {
  vidin   : virt_proc(6,2,[header,buf1],vip)
  rgbyuv  : virt_proc(4,2,[buf2,buf3],rgbdct)
  dct     : virt_proc(1,4,[buf3,xx,type,buf4],rgbdct)
  dmux    : virt_proc(2,7,[header,buf1,fsize,buf2,xx,type,numof],mp)
  quant   : virt_proc(3,4,[buf4,qtable,qcmd,buf5],mp)
  control : virt_proc(0,7,[numof,stats,qtable,qcmd,hcmd,htable,info],mp)
  vle     : virt_proc(5,6,[buf5,hcmd,htable,stats,flag,stream],vlep)
  vidout  : virt_proc(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])
  header  : buffer(1,    7, 1)
  info    : buffer(1,  672, 2)
  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)
  mem     : memory(10,8)
}
```

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

- The `virt_proc` class implements the virtual processor components as described in Section 4. This 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 `virt_proc` object, after which these FIFO buffers are specified in an array. The last parameter defines to which actual processor a virtual processor is linked: this is the application to architecture mapping.
- The `processor` class has three parameters. The first one describes to which memory interconnect it 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. 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 virtual processors which are mapped onto the same (actual) processor, communication is assumed to be performed internally. When the two virtual processors are mapped 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 two parameters. The first one specifies the memory it is connected to and the second one defines the time for setting up a connection.
- 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.

Obviously, the topology file allows for easy configuration of a Pearl simulation. It is simply a matter of changing a few numbers to change the application architecture mapping or to change the characteristics of a processor. For example, replacing a DSP for a dedicated hardware component in our M-JPEG* base architecture model can be achieved by reducing the instruction latencies of the processor object in question.

5.2 Design space exploration

To illustrate that Sesame and its Pearl simulation language facilitate efficient evaluation of different candidate architectures, we have performed an experiment [20] 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. For the architecture, we have assumed conservative timings: The bus-arbitration overhead when a request (at the level of abstract data types) is granted access to the bus equals to 10ns, while it takes 100ns to read/write a 64-bit word from/to DRAM. The instruction latencies for the microprocessor and DSP components were estimated using technical documentation. Figure 6(a) shows the simulation results in terms of the measured number of frames per second for all three candidate architectures (using a common bus, crossbar or Omega network). Below, the results for each of the communication structures are explained in more detail.

In Figure 6(b), 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 on I/O, being busy and being idle. As Figure 6(b) shows, the common bus architecture has a high memory

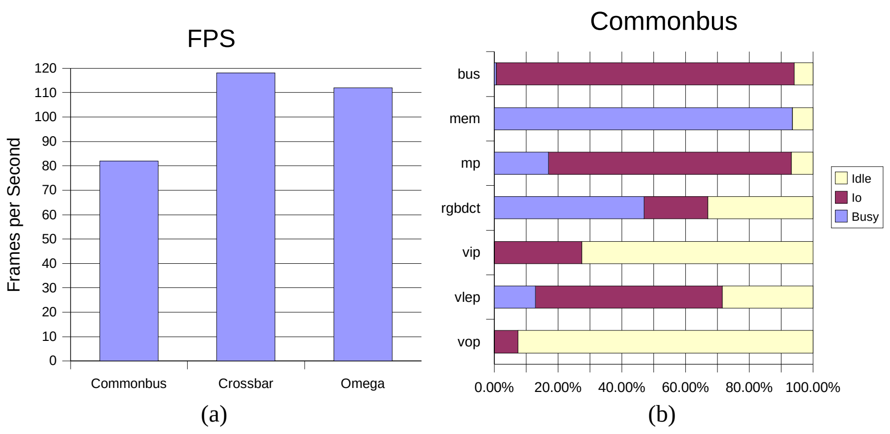


Fig. 6. Fig. (a) shows the measured frames per second for all three interconnects. Fig. (b) depicts a breakdown for the common bus showing busy/io/idle statistics for all architecture components.

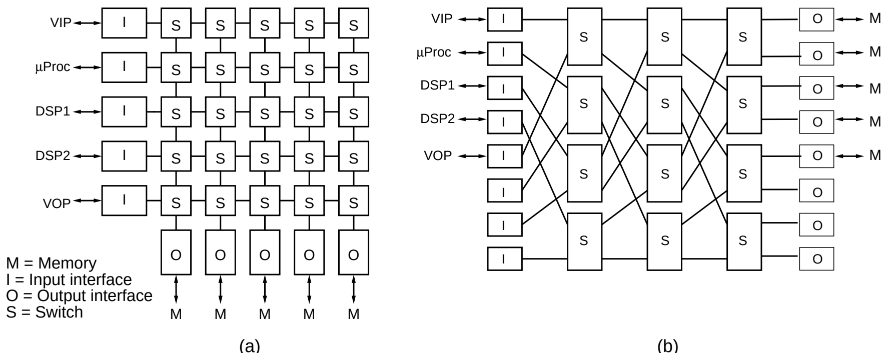


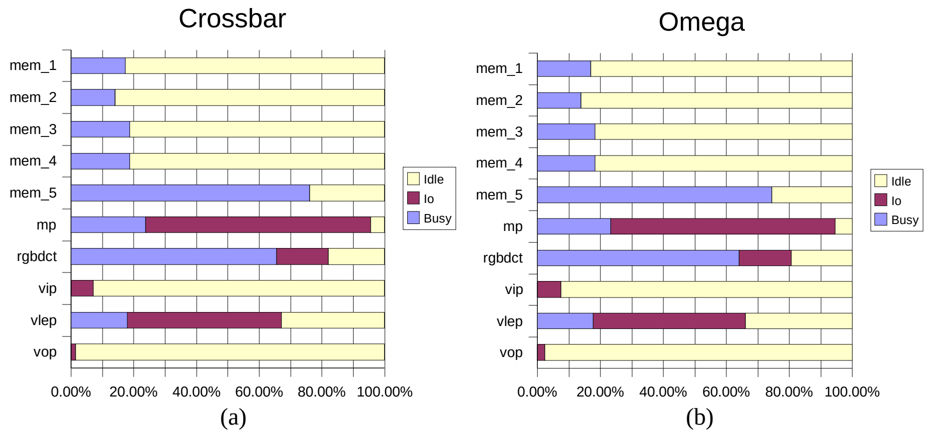
Fig. 7. The crossbar (a) and Omega (b) memory interconnects used in our experiments.

utilization while the various processors have low utilization and spend a lot of time doing I/O. Figure 6(a) shows that the common bus architecture obtains a framerate of 82 frames per second. While this is more than enough for real time operation, this is for a 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. Similar conclusions about the M-JPEG* architecture were drawn from experiments using the Spade environment [12].

To reduce the communication bottleneck of our M-JPEG* architecture, we have implemented a Pearl simulation model of a 5×5 crossbar switch, shown in Figure 7(a), and 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 is defined in the topology file. This mapping is, like the application to architecture mapping, easy to configure. In our crossbar model, the delay to set up a connection is identical to the bus-arbitration delay in the common bus model (10ns).

As the results in Figure 6(a) show, there is a substantial gain in frames per second compared with the common bus. When we look at the architecture component statistics in Figure 8(a) we see that all the components spend more time doing work and less time waiting for I/O. Since the memory load is now divided over five memories, the memory utilization is at about 20% for most memories. Note that memory 5 is still busy for almost 80% of the time. The reason for this is that one buffer takes 53% of memory bandwidth. This buffer contains the statistics needed for the (re)calculation of the Huffman and quantization tables. For every block of image data in the M-JPEG* application, these statistics are sent from the `VLE` process to the `Q-control` process.

As an alternative to the crossbar we have also modeled the Omega network as shown in Figure 7(b). 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, and thus requires routing. The Omega network is generally cheaper to implement than a crossbar because it has less switches, but the setup of a connection costs more (we account for a setup



(a) (b)

Fig. 8. Results for the crossbar (a) and Omega network (b) showing busy/io/idle statistics for all architecture components.

latency of 10ns per hop) and connections may be blocking. The latter means that it is not always possible to connect an idle input to an idle output.

The results in Figure 6(a) show that the Omega network is about 5% slower than a crossbar. Detailed statistics show that the processing cores spend a little more time waiting for I/O compared to the crossbar, leading to a slightly lower utilization. This is, however, hardly noticeable in Figure 8(b). So, when considering both cost and performance, the Omega network might be the better choice for replacing the common bus. Nevertheless, the simulation results indicate that, with the applied (multi-processor) architecture and mapping, the performance is highly communication bound. Therefore, mapping more application tasks to a single processing component (thereby reducing communication) or reducing the memory latency will certainly lead to improvements, which has already been demonstrated in [12, 19].

5.3 A note on modeling and simulation efficiency

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. This includes the construction of the crossbar and Omega network models, which had to be implemented from scratch, as well as the realization of a run-time visualization of the architecture simulations. Pearl is an object-based language, which means that we could exploit features such as "class sub-typing" to easily exchange the models for the different communication/memory architecture components. Making these models a sub-type of a generic interconnect type, the models could be replaced in a plug-and-play manner.

Models are not only constructed quickly in Pearl, but the actual simulation is also fast. For example, the simulation of M-JPEG* mapped onto the crossbar-based architecture takes just under 7 seconds. This was done on a 270Mhz Sun Ultra 5 Sparcstation with a video input stream of 16 frames of 128×128 pixels with RGB encoding.

6 Conclusions

In this paper, we have described a modeling and simulation methodology that allows for the efficient architectural exploration of heterogeneous embedded media systems. The presented methods and techniques are currently being realized in the Sesame modeling and simulation environment. Using an initial version of Sesame and an M-JPEG encoding application, we have illustrated the ease and swiftness with which the performance of different candidate architectures can be evaluated. More specifically, we have explored three shared-memory multi-processor target architectures, each with a different memory interconnect (common bus, crossbar and Omega network).

Research on Sesame will be continued along the lines described in this paper, with an emphasis on techniques for model refinement. In particular, the support for mixed-level simulation introduces many new research problems that need to be addressed. In addition, we intend to perform more case studies with industrially relevant applications to further demonstrate the effectiveness of the methods and tools we are developing.

Acknowledgments

This research is supported by PROGRESS, the embedded systems research program of the Dutch organization for Scientific Research NWO, the Dutch Ministry of Economic Affairs and the Technology Foundation STW. We thank Paul Lieverse, Todor Stefanov, Pieter van der Wolf and Ed Deprettere for their invaluable contributions to this work. In addition, we acknowledge Bart Kienhuis and Kees Vissers for their contribution to the modeling methodology.

References

1. Cadence Design Systems, Inc., http://www.cadence.com/.
2. Innoveda Inc., http://www.innoveda.com/.
3. 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.
4. M. Bauer and W. Ecker. Hardware/software co-simulation in a VHDL-based test bench approach. In *Proc. of the Design Automation Conference*, 1997.
5. J.-Y. Brunel, E.A. de Kock, W.M. Kruijtzer, H.J.H.N. Kenter, and W.J.M. Smits. Communication refinement in video systems on chip. In *Proc. 7th Int. Workshop on Hardware/Software Codesign*, pages 142–146, May 1999.
6. J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *Int. Journal of Computer Simulation*, 4:155–182, Apr. 1994.
7. S.L. Coumeri and D.E. Thomas. A simulation environment for hardware-software codesign. In *Proceedings of the Int. Conference on Computer Design*, pages 58–63, Oct. 1995.

8. M. Dubois, F.A. Briggs, I. Patil, and M. Balakrishnan. Trace-driven simulations of parallel and distributed algorithms in multiprocessors. In *Proc. of the Int. Conference in Parallel Processing*, pages 909–915, Aug. 1986.
9. G. Kahn. The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 74*, 1974.
10. B. Kienhuis, E.F. Deprettere, K.A. Vissers, and P. van der Wolf. An approach for quantitative analysis of application-specific dataflow architectures. In *Proc. of the Int. Conf. on Application-specific Systems, Architectures and Processors*, July 1997.
11. R. Klein and S. Leef. New technology links hardware and software simulators. Electronic Engineering Times, June 1996. http://www.mentorg.com/seamless/.
12. 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.
13. P. Lieverse, P. van der Wolf, and E.F. Deprettere. A trace transformation technique for communication refinement. In *Proc. of the 9th Int. Symposium on Hardware/Software Codesign*, pages 134–139, Apr. 2001.
14. P. Lieverse, P. van der Wolf, E.F. Deprettere, and K.A. Vissers. A methodology for architecture exploration of heterogeneous signal processing systems. *Journal of VLSI Signal Processing for Signal, Image and Video Technology*, 29(3):197–207, November 2001. Special issue on SiPS'99.
15. H.L. Muller. *Simulating computer architectures*. PhD thesis, Dept. of Computer Science, Univ. of Amsterdam, Feb. 1993.
16. 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):57–63, Nov. 2001.
17. 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 *Proc. of the Progress workshop on Embedded Systems*, pages 53–62, Oct. 2000.
18. J. Rowson. Hardware/software co-simulation. In *Proc. of the Design Automation Conference*, pages 439–440, 1994.
19. T. Stefanov, P. Lieverse, E.F. Deprettere, and P. van der Wolf. Y-chart based system level performance analysis: an M-JPEG case study. In *Proc. of the Progress workshop on Embedded Systems*, pages 113–124, Oct. 2000.
20. F. Terpstra, S. Polstra, A.D. Pimentel, and L.O. Hertzberger. Rapid evaluation of instantiations of embedded systems architectures: A case study. In *Proc. of the Progress workshop on Embedded Systems*, pages 251–260, Oct. 2001.