Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2007, Article ID 82123, 11 pages
doi:10.1155/2007/82123

Research Article

A Framework for System-Level Modeling and Simulation of Embedded Systems Architectures

Cagkan Erbas, Andy D. Pimentel, Mark Thompson, and Simon Polstra

Computer Systems Architecture Group, Informatics Institute, Faculty of Science, University of Amsterdam, Kruislaan 403, SJ Amsterdam, The Netherlands

Received 31 May 2006; Revised 7 December 2006; Accepted 18 June 2007

Recommended by Antonio Nunez

The high complexity of modern embedded systems impels designers of such systems to model and simulate system components and their interactions in the early design stages. It is therefore essential to develop good tools for exploring a wide range of design choices at these early stages, where the design space is very large. This paper provides an overview of our system-level modeling and simulation environment, Sesame, which aims at efficient design space exploration of embedded multimedia system architectures. Taking Sesame as a basis, we discuss many important key concepts in early systems evaluation, such as Y-chart-based systems modeling, design space pruning and exploration, trace-driven cosimulation, and model calibration.

1. INTRODUCTION

The ever increasing complexity of modern embedded systems has led to the emergence of system-level design [1]. High-level modeling and simulation, which allows for capturing the behavior of system components and their interactions at a high level of abstraction, plays a key role in system-level design. Because high-level models usually require less modeling effort and execute faster, they are especially well suited for the early design stages, where the design space is very large. Early exploration of the design space is critical, because early design choices have eminent effect on the success of the final product.

The traditional practice for embedded systems performance evaluation often combines two types of simulators, one for simulating the programmable components running the software and one for the dedicated hardware part. For simulating the software part, instruction-level or cycle-accurate simulators are commonly used. The hardware parts are usually simulated using hardware RTL descriptions realized in VHDL or Verilog. However, using such a hardware/software cosimulation environment during the early design stages has major drawbacks: (i) it requires too much effort to build them, (ii) they are often too slow for exhaustive explorations, and (iii) they are inflexible in evaluating different hardware/software partitionings. Because an explicit distinction is made between hardware and software simulation, a complete new system model might be required for the assessment of each hardware/software partitioning. To overcome these shortcomings, a number of high-level modeling and simulation environments have been proposed [2–5]. These recent environments break off from low-level system specifications, and define separate high-level specifications for behavior (what the system should do) and architecture (how it does it).

This paper provides an overview of the high-level modeling and simulation methods as employed in embedded systems design, focusing on our Sesame framework in particular. The Sesame environment primarily focuses on the multimedia application domain to efficiently prune and explore the design space of target platform architectures. Section 2 introduces the conceptual view of Sesame by discussing several design issues regarding the modeling and simulation techniques employed within the framework. Section 3 summarizes the design space pruning stage which is performed before cosimulation in Sesame. Section 4 discusses the cosimulation framework itself from a software design and implementation point of view. Section 5 addresses the calibration of system-level simulation models. In Section 6, we report experimental results achieved using the Sesame framework. Section 7 discusses related work. Finally, Section 8 concludes the paper.

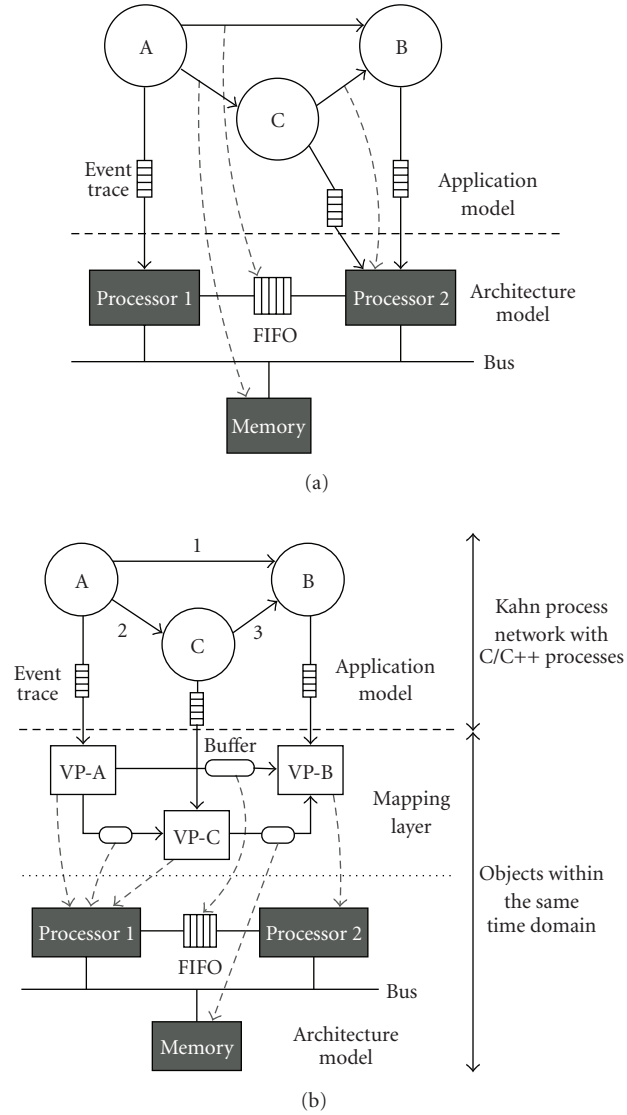


Figure 1: (a) Mapping an application model onto an architecture model. An event-trace queue dispatches application events from a Kahn process towards the architecture model component onto which it is mapped. (b) Sesame's three-layered structure: application model layer, architecture model layer, and the mapping layer which is an interface between application and architecture models.

2. THE SESAME APPROACH

The Sesame modeling and simulation environment facilitates performance analysis of embedded media systems architectures according to the Y-chart design principle [6, 7]. This means that Sesame decouples application form architecture by recognizing two distinct models for them. According to the Y-chart approach, an application model—derived from a target application domain—describes the functional behavior of an application in an architecture-independent manner. The application model is often used to study a target application and obtain rough estimations of its performance needs, for example, to identify computationally expensive tasks. This model correctly expresses the functional behavior, but is free from architectural issues, such as timing characteristics, resource utilization, or bandwidth constraints. Next, a platform architecture model—defined with the application domain in mind—defines architecture resources and captures their performance constraints. Finally, an explicit *mapping step* maps an application model onto an architecture model for cosimulation, after which the system performance can be evaluated quantitatively. This is depicted in Figure 1(a). The performance results may inspire the system designer to improve the architecture, modify the application, or change the projected mapping. Hence, the Y-chart modeling methodology relies on independent application and architecture models in order to promote their reuse to the greatest conceivable extent.

For application modeling, Sesame uses the Kahn process network (KPN) [8] model of computation in which parallel processes—implemented in a high-level language—communicate with each other via unbounded FIFO channels. Hence, the KPN model unveils the inherent task-level parallelism available in the application and makes the communication explicit. Furthermore, the code of each Kahn process is instrumented with annotations describing the application's computational actions, which allows to capture the computational behavior of an application. The reading from and writing to FIFO channels represent the communication behavior of a process within the application model. When the Kahn model is executed, each process records its computational and communication actions, and thus generates a trace of *application events*. These application events represent the application tasks to be performed and are necessary for driving an architecture model. Application events are generally coarse grained, such as *read(channel_id, pixel_block)* or *execute(DCT)*.

Parallelizing applications. The KPN applications of Sesame are obtained by *automatically* converting a sequential specification (C/C++) using the KPNgen tool [9]. This conversion is fast and correct by construction. As input KPNgen accepts sequential applications specified as static affine nested loop programs, onto which as a first step it applies a number of source-level transformations to adjust the amount of parallelism in the final KPN, the C/C++ code is transformed into single assigment code (SAC), which resembles the dependence graph (DG) of the original nested loop program. Hereafter, the SAC is converted to a polyhedral reduced dependency graph (PRDG) data structure, being a compact representation of a DG in terms of polyhedra. In the final step, a PRDG is converted into a KPN by associating a KPN process with each node in the PRDG. The parallel Kahn processes communicate with each other according to the data dependencies given in the DG. Further information on KPN generation can be found in [9, 10].

An architecture model simulates the performance consequences of the computation and communication events generated by an application model. It solely accounts for architectural (performance) constraints and does not need to model functional behavior. This is possible because the functional behavior is already captured by the application model, which drives the architecture simulation. The timing consequences of application events are simulated by

parameterizing each architecture model component with a table of operation latencies. The table entries could include, for example, the latency of an *execute(DCT)* event, or the latency of a memory access in the case of a memory component. This trace-driven cosimulation of application and architecture models allows to, for example, quickly evaluate different hardware/software partitionings by just altering the latency parameters of architecture model components (i.e., a low latency refers to a hardware implementation (computation) or on-chip memory access (communication), while a high latency models a software implementation or accessing an off-chip memory). With respect to communication, issues such as synchronization and contention on the shared resources are also captured in the architectural modeling.

To realize trace-driven cosimulation of application and architecture models, Sesame has an intermediate mapping layer. This layer consists of virtual processor components, which are the representation of application processes at the architecture level, and FIFO buffers for communication between the virtual processors. As shown in Figure 1(b), there is a one-to-one relationship between the Kahn processes and channels in the application model and the virtual processors and buffers in the mapping layer. The only difference is that the buffers in the mapping layer are limited in size, and their size depends on the modeled architecture. The mapping layer, in fact, has three functions [2]. First, it controls the mapping of Kahn processes (i.e., their event traces) onto architecture model components by dispatching application events to the correct architecture model component. Second, it makes sure that no communication deadlocks occur when multiple Kahn processes are mapped onto a single architecture model component. In this case, the dispatch mechanism also provides various strategies for application event scheduling. Finally, the mapping layer is capable of dynamically transforming application events into lower-level architecture events in order to realize flexible refinement of architecture models [2, 11].

The output of system simulations in Sesame provides the designer with performance estimates of the system(s) under study together with statistical information such as utilization of architecture model components (idle/busy times), the degree of contention in a system, profiling information (time spent in different executions), critical path analysis, and average bandwidth between architecture components. These high-level simulations allow for early evaluation of different design choices. Moreover, they can also be useful for identifying trends in the systems' behavior, and help reveal design flaws/bottlenecks early in the design cycle.

Despite of being an effective and efficient performance evaluation technique, high-level simulation would still fail to explore large parts of the design space. This is because each system simulation only evaluates a single design point in the maximal design space of the early design stages. Thus, it is extremely important that some direction is provided to the designer as a guidance toward promising system architectures. Analytical methods may be of great help here, as they can be utilized to identify a small set of promising candidates. The designer then can focus only on this small set, for which simulation models can be constructed at multiple levels of abstraction. The process of trimming down an exponential design space to some finite set is called *design space pruning*. In the next section, we briefly discuss how Sesame prunes the design space by making use of analytical modeling and multiobjective evolutionary algorithms [12].

3. DESIGN SPACE PRUNING

As already mentioned in the previous section, Sesame supports separate application and architecture models within its exploration framework. This separation implies an explicit mapping step for cosimulation of the two models. Since the enumeration of all possible mappings grows exponentially, a designer usually needs a subset of best candidate mappings for further evaluation in terms of cosimulation. Therefore, in summary, the mapping problem in Sesame is the optimal mapping of an application model onto a (platform) architecture model. The problem formulation in Sesame takes three objectives into account [12]: maximum processing time in the system, total power consumption of the system, and the cost of the architecture. This section aims at giving an overview of the formulation of the mapping problem which allows us to quickly search for promising candidate system architectures with respect to the above three objectives.

Application modeling

The application models in Sesame are process networks which can be represented by a graph $\mathrm{AP} = (V_K, E_K)$, where the sets V_K and E_K refer to the nodes (i.e., processes) and the directed channels between these nodes, respectively. For each node in the application model, a computation requirement (workload imposed by the node onto a particular component in the architecture model), and an allele set (the processors that it can be mapped onto) are defined. For each channel in the application model, a communication requirement is defined only if that channel is mapped onto an external memory element. Hence, we neglect internal communications (within the same processor) and only consider external (interprocessor) communications.

Architecture modeling

The architecture models in Sesame can also be represented by a graph $\mathrm{AR} = (V_A, E_A)$, where the sets V_A and E_A denote the architecture components and the connections between them, respectively. For each processor in an architecture model, we define the parameters processing capacity, power consumption during execution, and a fixed cost.

Having defined more abstract mathematical models for Sesame's application and architecture model components, we have the following optimization problem.

Definition 1 (MMPN problem [12, 13]). Multiprocessor mappings of process networks (MMPN) problem is

$$\begin{aligned} &\min \mathbf{f}(\mathbf{x}) = \left(f_1(\mathbf{x}), f_2(\mathbf{x}), f_3(\mathbf{x})\right) \\ &\text{subject to } g_i(\mathbf{x}), \quad i \in \{1, \ldots, n\},\ \mathbf{x} \in X_f, \end{aligned} \tag{1}$$

where f_1 is the maximum processing time, f_2 is the total power consumption, f_3 is the total cost of the system.

The functions g_i are the constraints, and $\mathbf{x} \in X_f$ are the decision variables. These variables represent decisions like which processes are mapped onto which processors, or which processors are used in a particular architecture instance. The constraints of the problem make sure that the decision variables are valid, that is, X_f is the feasible set. For example, all processes need to be mapped onto a processor from their allele sets; or if two communicating processes are mapped onto the same processor, the channel(s) between them must also be mapped onto the same processor, and so on. The optimization goal is to identify a set of solutions which are superior to all other solutions when all three objective functions are minimized.

Here, we have provided an overview of the MMPN problem. The exact mathematical modeling and formulation can be found in [12].

3.1. Multiobjective optimization

To solve the above multiobjective integer optimization problem, we use the (improved) strength Pareto evolutionary algorithm (SPEA2) [14] that finds a set of approximated Pareto-optimal mapping solutions, that is, solutions that are *not dominated* in terms of quality (performance, power, and cost) by any other solution in the feasible set. To this end, SPEA2 maintains an external set to preserve the nondominated solutions encountered so far besides the original population. Each mapping solution is represented by an individual encoding, that is, a chromosome in which the genes encode the values of parameters. SPEA2 uses the concept of dominance to assign fitness values to individuals. It does so by taking into account how many individuals a solution dominates and is dominated by. Distinct fitness assignment schemes are defined for the population and the external set to always ensure that better fitness values are assigned to individuals in the external set. Additionally, SPEA2 performs *clustering* to limit the number of individuals in the external set (without losing the boundary solutions) while also maintaining diversity among them. For selection, it uses binary tournament with replacement. Finally, only the external nondominated set takes part in selection. In our SPEA2 implementation, we have also introduced a repair mechanism [12] to handle infeasible solutions. The repair takes place before the individuals enter evaluation to make sure that only valid individuals are evaluated.

In [12], we have shown that an SPEA2 implementation to heuristically solve the multiobjective optimization problem can provide the designer with good insight on the quality of candidate system architectures. This knowledge can subsequently be used to select an initial (platform) architecture to start the system-level simulation phase, or to guide a designer in finding for example alternative architectures when system-level simulation indicates that the architecture under investigation does not fulfill the requirements. Next, we continue discussing implementation details regarding Sesame's system-level simulation framework.

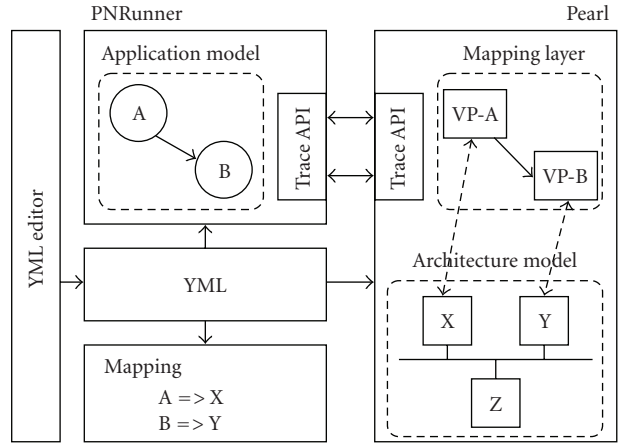


Figure 2: Sesame software overview. Sesames model description language YML is used to describe the application model, the architecture model, and the mapping which relates the two models for cosimulation.

4. THE COSIMULATION ENVIRONMENT

All three layers in Sesame (see Figure 1(b)) are composed of components which should be instantiated and connected using some form of object creation and initialization mechanism. An overview of the Sesame software framework is given in Figure 2, where we use YML (Y-chart modeling language) to describe the application model, the architecture model, and the mapping which relates the two models for cosimulation. YML, which is an XML-based language, describes simulation models as directed graphs. The core elements of YML are `network`, `node`, `port`, `link`, and `property`. YML files containing only these elements are called flat YML. There are two additional elements `set` and `script` which were added to equip YML with scripting support to simplify the description of complicated models, for example, a complex interconnect with a large number of nodes. We now briefly describe these YML elements.

(i) `network:` network elements contain graphs of nodes and links, and may also contain subnetworks which create hierarchy in the model description. A network element requires a `name` and optionally a `class` attribute. Names must be unique in a network for they are used as identifiers.

(ii) `node:` node elements represent building blocks (or components) of a simulation model. Kahn processes in an application model or components in an architecture model are represented by nodes in their respective YML description files. Node elements also require a `name` and usually a `class` attribute which are used by the simulators to identify the node type. For example, in Figure 3(a), the class attribute of node A specifies that it is a C++ (application) process.

(iii) `port:` port elements add connection points to nodes and networks. They require `name` and `dir` attributes. The `dir` attribute defines the direction of the port and may have values *in* or *out*. Port names must also be unique in a node or network.

```
<network name="ProcessNetwork" class="KPN">
  <property name="library" value="libPN.so"/>
  <node name="A" class="CPP_Process">
    <port name="port0" dir="in"/>
    <port name="port1" dir="out"/>
  </node>
  <node name="B" class="CPP_Process">
    <port name="port0" dir="in"/>
    <port name="port1" dir="out"/>
  </node>
  <node name="C" class="CPP_Process">
    <port name="port0" dir="in"/>
    <port name="port1" dir="out"/>
  </node>
  <link innode="B" inport="port1"
    outnode="A" outport="port0"/>
  <link innode="A" inport="port1"
    outnode="C" outport="port0"/>
  <link innode="C" inport="port1"
    outnode="B" outport="port0"/>
</network>
```

(a) YML description of process network in Figure 1

```
<set init="$i = 0" cond="$i &lt; 10" loop="$i++">
  <script>
   $nodename="processor$i"
  <script/>
  <node name="$nodename" class="pearl_object">
    <port name="port0" dir="in"/>
    <port name="port1" dir="out"/>
  </node>
</set>
```

(b) An example illustrating the usage of set and script elements

```
<mapping side="source" name="application">
  <mapping side="dest" name="architecture">
    <map source="A" dest="X">
      <port source="portA" dest="portBus"/>
    </map>
    <map source="B" dest="Y">
      <port source="portB" dest="portBus"/>
    </map>
    <instruction source="op_A" dest="op_A"/>
    <instruction source="op_B" dest="op_B"/>
  </mapping>
</mapping>
```

(c) The YML for the mapping in Figure 2

FIGURE 3: Structure and mapping descriptions via YML files.

(iv) `link`: link elements connect ports. They require `innode`, `inport`, `outnode`, and `outport` attributes. The `innode` and `outnode` attributes denote the names of nodes (or subnetworks) to be connected. Ports used for the connection are specified by `inport` and `outport`.

(v) `property`: property elements provide additional information for YML objects. Certain simulators may require certain information on parameter values. For example, Sesame 's architecture simulator needs to read an array of execution latencies for each processor component in order

to associate timing values to incoming application events. In Figure 3(a), the *ProcessNetwork* element has a *library* property which specifies the name of the shared library where the object code belonging to *ProcessNetwork*, for example, object codes of its node elements *A*, *B*, and *C* reside. Property elements require `name` and `value` attributes.

(vi) `script:` the script element supports Perl as a scripting language for YML. The text encapsulated by the script element is processed by the Perl interpreter in the order it appears in the YML file. The script element has no attributes. The namings in `name`, `class`, and `value` attributes that begin with a "$" are evaluated as global Perl variables within the current context of the Perl interpreter. Therefore, users should take good care to avoid name conflicts. The script element is usually used together with the set element in order to create complex network structures. Figure 3(b) gives such an example, which will be explained below.

(vii) `set:` the set element provides a for-loop like structure to define YML structures which simplifies complex network descriptions. It requires three attributes `init`, `cond`, and `loop`. YML interprets the values of these attributes as a script element. The `init` is evaluated once at the beginning of set element processing, `cond` is evaluated at the beginning of every iteration and is considered as a boolean. The processing of a set element stops when its `cond` is false or 0. The `loop` attribute is evaluated at the end of each iteration. Figure 3(b) provides a simple example in which the `set` element is used to generate ten processor components.

The YML description of the process network in Figure 1(a) is shown in Figure 3. The process network defined has three C++ processes, each associated with input and output ports, which are connected through the link elements and embedded in *ProcessNetwork*. In addition to structural descriptions, YML is also used to specify mapping descriptions, that is, relating application tasks to architecture model components.

(i) `mapping:` mapping elements identify application and architecture simulators for mapping. An example is given with the following `map` element.

(ii) `map:` map elements map application nodes (model components) onto architecture nodes. The node mapping in Figure 2, that is mapping processes A and B onto processors X and Y, is given in Figure 3(c) where *source* (*dest*) refers to the application (architecture) side.

(iii) `port:` port elements relate application ports to architecture ports. When an application node is mapped onto an architecture node, the connection points (or ports) also need to be mapped to specify which communication medium should be used in the architecture model simulator.

(iv) `instruction:` instruction elements specify computation and communication events generated by the application simulator and consumed by the architecture simulator. In short, they map application event names onto architecture event names.

Sesame 's application simulator is called *PNRunner* , or process network runner. PNRunner implements the semantics of Kahn process networks and supports the well-known YAPI interface [15]. It reads a YML application description file and executes the application model described there. The object code of each process is fetched from a shared library as specified in the YML description, for example, "libPN.so" in Figure 3. PNRunner currently supports C++ processes, while any language for which a process loader class is written could be used. This is because PNRunner relies on the loader classes for process executions. Besides, from the perspective of PNRunner , data communicated through the channels is typed as "blocks of bytes." Interpretation of data types is done by processes and process loaders. As already shown in Figure 3, the class attribute of a node informs PNRunner which process loader it should use. To pass arguments to the process constructors or to the processes themselves, the property *arg* has been added to YML. Process classes are loaded through generated stub code. In Figure 4, we present an example application process, which is an IDCT process from an H.263 decoder application. It is derived from the parent class *Process* which provides a common interface. Following YAPI, ports are template classes to set the type of data exchanged.

As can be seen in Figure 2, PNRunner also provides a trace API to drive an architecture simulator. Using this API, PNRunner can send application events to the architecture simulator where their performance consequences are simulated. While reading data from or writing data to ports, PNRunner generates a communication event as a side effect. Hence, communication events are automatically generated. Computation events, however, must be signaled explicitly by the processes. This is achieved by annotating the process code with *execute(char*)* statements. In the main function of the IDCT process in Figure 4, we show a typical example. This process first reads a block of data from port *blockInP*, performs an IDCT operation on the data, and writes output data to port *blockOutP*. The *read* and *write* functions, as a side effect, automatically generate the communication events. However, we have added the function call *execute("IDCT")* to record that an IDCT operation is performed. The string passed to the *execute* function represents the type of the execution event and needs to match to the operations defined in the YML file.

Sesame 's architecture models are implemented in the Pearl discrete event simulation language [16], or in SCPEx [17], which is a variant of Pearl implemented on top of SystemC. Pearl is a small but powerful object-based language which provides easy construction of abstract architecture models and fast simulation. It has a C-like syntax with a few additional primitives for simulation purposes. A Pearl program is a collection of concurrent objects which communicate with each other through message passing. Each object has its own data space which cannot be directly accessed by other objects. The objects send messages to other objects to communicate, for example, to request some data or operation. The called object may then perform the request, and if expected, may also reply to the calling object.

The Pearl programming paradigm (as well as that of SCPEx) differs from the popular SystemC language in a number of important aspects. Pearl, implementing the message-passing mechanism, abstracts away the concept of ports and

```
class Idct: public Process {
  InPort<Block> blockInP;
  OutPort<Block> blockOutP;
  // private member function
  void idct (short* block);

  public:
    Idct(const class Id& n, In<Block>& blockinF,
     Out<Block>& blockOutF);
    const char* type() const {return "Idct";}
    void main();
};

// constructor
Idct::Idct(const class Id& n, In<Block>& blockInF,
  Out<Block>& blockOutF)
    : Process(n), blockInP(id("blockInP"), blockInF),
     blockOutP(id("blockOutP"), blockOutF)
{ }
// main member function
void Idct::main() {
  Block tmpblock;

  while(true) {
    read(blockInP, tmpblock);
    idct(tmpblock.data);
    execute("IDCT");
    write(blockOutP, tmpblock);
  }
}
```

Figure 4: C++ code for the IDCT process taken from an H.263 decoder process network application. The process reads a block of data from its input port, performs an IDCT operation on the data, and writes the transformed data to its output port.

explicit channels connecting ports as employed in SystemC. Buffering of messages in the object message queues is also handled implicitly by the Pearl run-time system, whereas in SystemC one has to implement explicit buffering. Additionally, Pearl's message-passing primitives lucidly incorporate interobject synchronization, while separate event notifications are needed in SystemC. As a consequence of these abstractions, Pearl is, with respect to SystemC, less prone to programming errors [17].

Figure 5 shows a piece of Pearl code implementing a high-level processor component. Pearl objects communicate via synchronous or asynchronous messages. The `load` method of the processor object in Figure 5 communicates with the memory object synchronously via the message call:

```
mem ! load (nbytes, address);
```

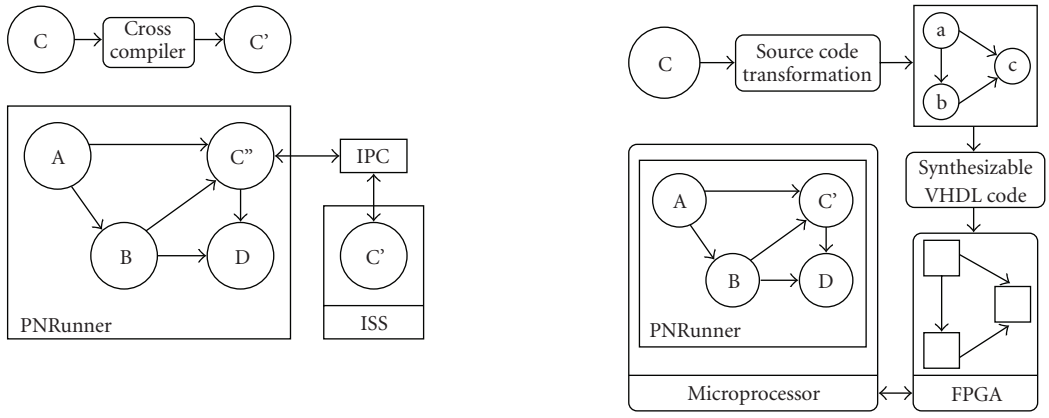
An object sending a synchronous message blocks until the receiver replies with the `reply()` primitive. Asynchronous messages, however, do not cause the sending object to block; the object continues execution with the next instruction. Pearl objects have message queues where all received messages are collected. Objects can wait for messages to arrive using `block()` with the method names as parameter or *any* to refer to all methods. To wait for a certain interval in simulation time, the `blockt(interval)` primitive is used. In Figure 5, for example, the `compute` method models an execution latency with the `blockt` using the array of operation latencies provided by the YML description. So, dependent on the type of the incoming computation event, a certain latency is modeled. At the end of simulation, the Pearl runtime system outputs a post-mortem analysis of the simulation results. For this purpose, it keeps track of some statistical information such as utilization of objects (idle/busy times), contention (busy objects with pending messages), profiling (time spent in object methods), critical path analysis, and average bandwidth between objects.

5. CALIBRATING SYSTEM-LEVEL MODELS

As was explained, an architecture model component in Sesame associates latency values to the incoming application events that comprise the computation and communication operations to be simulated. This is accomplished by parameterizing each architecture model component with a table of operation latencies. Therefore, regarding the accuracy of system-level performance evaluation, it is important that these latencies correctly reflect the speed of their corresponding architecture components. We now briefly discuss two techniques (one for software and another one for hardware implementations) which are deployed in Sesame to attain latencies with good accuracy.

```
class processor

mem : memory
nopers : integer   // needed for array size
opers_t = [nopers] integer   // type definition
opers : opers_t   // array of operation latencies

simtime : integer   // local variable

compute : (operindx:integer) -> void {
  simtime = opers[operindx];   // simulation time
  blockt(simtime);   // simulate the operation
  reply();
}

load : (nbytes:integer, address:integer) -> void {
  mem ! load(nbytes, address);   // memory call
  reply();
}

// store method omitted

{
  while(true) {
    block(any);
  }
}
```

Figure 5: Pearl implementation of a generic high-level processor.



(a) Solution for software implementations

(b) Solution for hardware implementations

Figure 6: Obtaining low-level numbers for model calibration.

The first technique can be used to calibrate the latencies of programmable components in the architecture model, such as microprocessors, DSPs, application specific instruction processors (ASIPs), and so on. The calibration technique, as depicted in Figure 6(a), requires that the designer has access to the C/C++ cross compiler and a low-level (ISS/RTL) simulator of the target processor. In the figure, we have chosen to calibrate the latency value(s) of (Kahn) process C which is mapped to some kind of processor for which we have a cross compiler and an instruction set simulator (ISS). First, we take process C, and substitute its Kahn communication for UNIX IPC-based communication (i.e., to realize the interprocess communication between the two simulators: PNRunner and the ISS), and generate binary code using the cross compiler. The code of process C in PNRunner is also modified (now called process C"). Process C" now simply forwards its input data to the ISS, blocks until it receives processed data from the ISS, and then writes received data to its output Kahn channels. Hence, process C" leaves all computations to the ISS, which additionally records the number of cycles taken for the computations while performing them. Once this mixed-level simulation is finished, recordings of the ISS can be analyzed statistically, for example, the arithmetic means of the measured code fragments can be taken as the latency for the corresponding architecture component in the system-level architecture model. This scheme can also be easily extended to an application/architecture mixed-level cosimulation using a recently proposed technique called *trace calibration* [18].

Table 1: Simulation and validation results.

Case study	Simulation efficiency	Accuracy
Motion-JPEG [2] (nonrefined)	700 000 cycles/s on 2.8 GHz Pentium 4	—
Motion-JPEG [2] (refined)	250 000 cycles/s on 2.8 GHz Pentium 4	—
QR Algorithm [21]	5000 cycles/s on 333 MHz Sun Ultra 10	3.5% (best) 36% (worst)
Motion-JPEG [22] (refined)	1 350 000 cycles/s on 2.8 GHz Pentium 4	0.5% (best) 1.9% (worst)

The second calibration technique makes use of reconfigurable computing with field programmable gate arrays (FPGAs). Figure 6(b) illustrates this calibration technique for hardware components. This time it is assumed that the process C is to be implemented in hardware. First, the application programmer takes the source code of process C and performs source code transformations on it, which unveils the parallelism within the process C. These transformations, starting from a single process, create a functionally equivalent (Kahn) process network with processes at finer granularities. The abstraction level of the processes is lowered such that a one-to-one mapping of the process network to an FPGA platform becomes possible. There are already some prototype environments which can accomplish these steps for certain applications. For example, the Compaan tool [19] can automatically perform process network transformations while the Laura [20] tool can generate VHDL code from a process network specification. This VHDL code can then be synthesized and mapped onto an FPGA using commercial synthesis tools. By mapping process C onto an FPGA and executing the remaining processes of the original process network on a microprocessor (e.g., an FPGA board connected to a computer using a PCI bus, or a processor core embedded into the FPGA), statistics on the hardware implementation of process C can be collected to calibrate the corresponding system-level hardware component.

6. EXPERIMENTS

In Table 1, we present some numbers of interest from our earlier experiments with the Sesame framework. The first two rows correspond to two system-level simulations, where we have subsequently mapped a Motion-JPEG encoder onto an MP-SoC platform architecture [2]. In both simulations, we have encoded 11 picture frames each with a resolution of 352×288 pixels and used nonrefined (black-box) processor components except the DCT processor. The only difference in two simulations is that the DCT processor is nonrefined in the first simulation, while a refined pipelined model is used on the second case. These simulation results reveal that system-level simulation can be very fast, simulating the entire multiprocessor system within a range of hundreds of thousands to a few millions of cycles/s, even in the case of model refinements. The last two rows of Table 1 are on the accuracy of system-level simulation based on some earlier validation

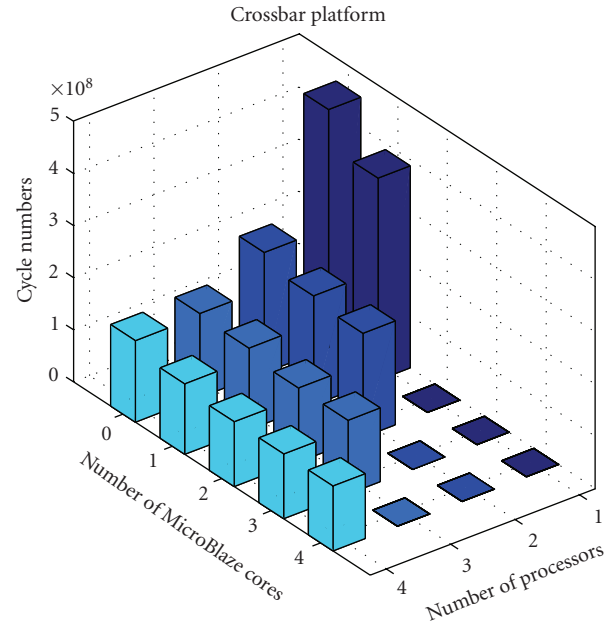


Figure 7: Performance results of the best mappings obtained by exhaustive search.

experiments. These results have been obtained by calibrating Sesame using techniques from Section 5 and comparing the results with real implementations on an FPGA. The results suggest that well-calibrated system-level models can be very accurate. We should further note that the architecture models in QR and M-JPEG experiments are only composed of around 400 and 600 lines of Pearl code, respectively.

Figure 7 shows the results from an experiment in which we have mapped a restructured version of the aforementioned M-JPEG encoder—containing six application processes—onto an MP-SoC platform architecture. This architecture consists of up to four processor cores connected by a crossbar switch. The processor cores can be of the type MicroBlaze or PowerPC. This is due to the fact that we are currently using a Virtex II Pro FPGA platform to validate our simulation results against a real system prototype. Thanks to Sesame's fast architecture simulator, we were able to determine the performance consequences of all points in a part of the design space by exhaustively simulating every single point. This means that we have varied the number of processors from one to four, the type of processors from MicroBlaze to PowerPC, and the mappings of the six application processes onto these different instances of the platform architecture. All of this yields 10 148 experiments which in total took 86 minutes using the Sesame system-level simulation framework. In Figure 7, we have plotted the performance of the design points with the best mappings of the application onto the fourteen different instances of the platform architecture. We observe that the estimated execution time of the system ranges from 124, 287, 479 cycles for the fastest implementation to 457, 546, 152 cycles for the slowest to process an input of 8 consecutive frames of 128×128 pixels in YUV format. For bigger systems where it is infeasible to explore every point

in the design space, as explained in Section 3, Sesame relies on the outcome of a design space pruning stage, which precedes the system-level simulation stage and provides input to the this stage by identifying a set of high-potential design points that may yield good performance.

7. RELATED WORK

There are a number of architectural exploration environments, such as (Metro)Polis [4, 6], Mescal [23], MESH [5], Milan [24], and various SystemC-based environments like in [25], that facilitate flexible system-level performance evaluation by providing support for mapping a behavioral application specification to an architecture specification. For example, in MESH [5], a high-level simulation technique based on frequency interleaving is used to map logical events (referring to application functionality) to physical events (referring to hardware resources). In [26], an excellent survey is presented of various methods, tools, and environments for early design space exploration. In comparison to most related efforts, Sesame tries to push the separation of modeling application behavior and modeling architectural constraints at the system level to even greater extents. This is achieved by architecture-independent application models, application-independent architecture models, and a mapping step that relates these models for trace-driven cosimulation.

In [27] Lahiri et al. also use a trace-driven approach, but this is done to extract communication behavior for studying on-chip communication architectures. Rather than using the traces as input to an architecture simulator, their traces are analyzed statically. In addition, a traditional hardware/software cosimulation stage is required in order to generate the traces. Archer [28] shows similarities with the Sesame framework due to the fact that both Sesame and Archer stem from the earlier Spade project [29]. A major difference is, however, that Archer follows a different application-to-architecture mapping approach. Instead of using event traces, it maps the so-called symbolic programs, which are derived from the application model, onto architecture model resources. Moreover, unlike Sesame, Archer does not include support for rapidly pruning the design space.

8. DISCUSSION

This paper provided an overview of our system-level modeling and simulation environment—Sesame. Taking Sesame as a basis, we have discussed many important key concepts such as Y-chart-based systems modeling, design space pruning and exploration, trace-driven cosimulation, model calibration and so on. Future work on Sesame will include (i) extending application and architecture model libraries further with components operating at multiple levels of abstraction, (ii) improving its accuracy with techniques such as trace calibration [18], (iii) performing further validation case studies to test proposed accuracy improvements, and (iv) applying Sesame to other application domains.

What is more, the calibration of timing parameters of the system-level models by getting feedback from (or coupling with) low-level simulators or from FPGA prototype implementations can also be extended to calibrate power numbers. For example, instead of coupling Sesame with simplescalar to measure timing values for software components, one could as well couple Sesame with a low-level power simulator such as Wattch [30] or Simplepower [31] to obtain power numbers. The same is true for the hardware components. Once an FPGA prototype implementation is built, it can be used for power measurement during execution.

REFERENCES

[1] K. Keutzer, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli, "System-level design: orthogonalization of concerns and platform-based design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 12, pp. 1523–1543, 2000.

[2] A. D. Pimentel, C. Erbas, and S. Polstra, "A systematic approach to exploring embedded system architectures at multiple abstraction levels," *IEEE Transactions on Computers*, vol. 55, no. 2, pp. 99–112, 2006.

[3] A. Bakshi, V. Prasanna, and A. Ledeczi, "Milan: a model based integrated simulation framework for design of embedded systems," in *Proceedings of the Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES '01)*, pp. 82–87, Snowbird, Utah, USA, June 2001.

[4] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli, "Metropolis: an integrated electronic system design environment," *Computer*, vol. 36, no. 4, pp. 45–52, 2003.

[5] A. Cassidy, J. Paul, and D. Thomas, "Layered, multi-threaded, high-level performance design," in *Proceedings of the International Conference on Design, Automation and Test in Europe (DATE '03)*, pp. 954–959, Munich, Germany, March 2003.

[6] F. Balarin, P. D. Giusto, A. Jurecska, et al., *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach*, Kluwer Academic, Boston, Mass, USA, 1997.

[7] B. Kienhuis, E. Deprettere, K. Vissers, and P. van der Wolf, "An approach for quantitative analysis of application-specific dataflow architectures," in *Proceedings of IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP '97)*, pp. 338–349, Zurich, Switzerland, July 1997.

[8] G. Kahn, "The semantics of a simple language for parallel programming," in *Proceedings of the IFIP Congress on Information Processing*, pp. 471–475, Stockholm, Sweden, August 1974.

[9] S. Verdoolaege, H. Nikolov, and T. Stefanov, "Improved derivation of process networks," in *Proceedings of the 4th International Workshop on Optimization for DSP and Embedded Systems (ODES '06)*, New York, NY, USA, March 2006.

[10] T. Stefanov, B. Kienhuis, and E. Deprettere, "Algorithmic transformation techniques for efficient exploration of alternative application instances," in *Proceedings of the 10th International Symposium on Hardware/Software Codesign (CODES '02)*, pp. 7–12, Estes Park, Colo, USA, May 2002.

[11] C. Erbas and A. D. Pimentel, "Utilizing synthesis methods in accurate system-level exploration of heterogeneous embedded systems," in *Proceedings of IEEE Workshop on Signal Processing Systems (SIPS '03)*, pp. 310–315, Seoul, Korea, August 2003.

[12] C. Erbas, S. Cerav-Erbas, and A. D. Pimentel, "Multiobjective optimization and evolutionary algorithms for the application mapping problem in multiprocessor system-on-chip design," *IEEE Transactions on Evolutionary Computation*, vol. 10, no. 3, pp. 358–374, 2006.

[13] C. Erbas, S. Cerav-Erbas, and A. D. Pimentel, "A multiobjective optimization model for exploring multiprocessor mappings of process networks," in *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pp. 182–187, Newport Beach, Calif, USA, October 2003.

[14] E. Zitzler, M. Laumanns, and L. Thiele, "SPEA2: improving the strength pareto evolutionary algorithm for multiobjective optimization," in *Evolutionary Methods for Design, Optimisation and Control with Application to Industrial Problems*, K. Giannakoglou, D. Tsahalis, J. Periaux, K. D. Papailiou, and T. Fogarty, Eds., pp. 95–100, International Center for Numerical Methods in Engineering, Barcelona, Spain, 2002.

[15] E. A. de Kock, G. Essink, W. Smits, et al., "YAPI: application modeling for signal processing systems," in *Proceedings of the 37th Design Automation Conference (DAC '00)*, pp. 402–405, Los Angeles, Calif, USA, June 2000.

[16] J. E. Coffland and A. D. Pimentel, "A software framework for efficient system-level performance evaluation of embedded systems," in *Proceedings of the ACM Symposium on Applied Computing*, pp. 666–671, Melbourne, Fla, USA, March 2003.

[17] M. Thompson and A. D. Pimentel, "A high-level programming paradigm for systemC," in *Proceedings of the 4th International Workshops on Systems, Architectures, Modeling, and Simulation (SAMOS '04)*, vol. 3133 of *Lecture Notes in Computer Science*, pp. 530–539, Springer, Samos, Greece, July 2004.

[18] M. Thompson, A. D. Pimentel, S. Polstra, and C. Erbas, "A mixed-level co-simulation method for system-level design space exploration," in *Proceedings of the IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia*, pp. 27–32, Seoul, Korea, October 2006.

[19] B. Kienhuis, E. Rijpkema, and E. Deprettere, "Compaan: deriving process networks from Matlab for embedded signal processing architectures," in *Proceedings of the 18th International Workshop Hardware/Software Codesign (CODES '00)*, pp. 13–17, San Diego, Calif, USA, May 2000.

[20] C. Zissulescu, T. Stefanov, B. Kienhuis, and E. Deprettere, "Laura: leiden architecture research and exploration tool," in *Proceedings of the 13th International Conference on Field-Programmable Logic and Applications (FPL '03)*, P. Cheung, G. Constantinides, and J. de Sousa, Eds., vol. 2778 of *Lecture Notes in Computer Science*, pp. 911–920, Springer, Lisbon, Portugal, September 2003.

[21] A. D. Pimentel, F. Terpstra, S. Polstra, and J. E. Coffland, "On the modeling of intra-task parallelism in task-level parallel embedded systems," in *Domain-Specific Processors: Systems, Architectures, Modeling, and Simulation*, S. Bhattacharyya, E. Deprettere, and J. Teich, Eds., pp. 85–105, Springer, Berlin, Germany, 2003.

[22] A. D. Pimentel, "The artemis workbench for system-level performance evaluation of embedded systems," *International Journal of Embedded Systems*, vol. 1, no. 7, 2005.

[23] A. Mihal, C. Kulkarni, C. Sauer, et al., "Developing architectural platforms: a disciplined approach," *IEEE Design and Test of Computers*, vol. 19, no. 6, pp. 6–16, 2002.

[24] S. Mohanty and V. K. Prasanna, "Rapid system-level performance evaluation and optimization for application mapping onto SoC architectures," in *Proceedings of the 15th Annual IEEE International ASIC/SOC Conference*, pp. 160–167, Rochester, NY, USA, September 2002.

[25] T. Kogel, A. Wieferin, R. Leupers, et al., "Virtual architecture mapping: a systemC based methodology for architectural exploration of system-on-chip designs," in *Proceedings of the 3rd International Workshop on Computer Systems: Architectures, Modeling, and Simulation (SAMOS '03)*, pp. 138–148, Samos, Greece, July 2003.

[26] M. Gries, "Methods for evaluating and covering the design space during early design development," *Integration, the VLSI Journal*, vol. 38, no. 2, pp. 131–183, 2004.

[27] K. Lahiri, A. Raghunathan, and S. Dey, "System-level performance analysis for designing on-chip communication architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 6, pp. 768–783, 2001.

[28] V. Zivkovic, E. Deprettere, P. van der Wolf, and E. de Kock, "Fast and accurate multiprocessor architecture exploration with symbolic programs," in *Proceedings of the International Conference on Design, Automation and Test in Europe (DATE '03)*, pp. 656–661, Munich, Germany, March 2003.

[29] P. Lieverse, P. van der Wolf, E. Deprettere, and K. Vissers, "A methodology for architecture exploration of heterogeneous signal processing systems," *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 29, no. 3, pp. 197–207, 2001.

[30] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: a framework for architectural-level power analysis and optimizations," in *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA '00)*, pp. 83–94, Vancouver, BC, Canada, June 2000.

[31] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin, "The design and use of simplepower: a cycle-accurate energy estimation tool," in *Proceedings of the 37th Design Automation Conference (DAC '00)*, pp. 340–345, Los Angeles, Calif, USA, June 2000.