

On Modeling Intra-task Parallelism in Task-level Parallel Embedded Systems

Andy D. Pimentel, Frank P. Terpstra, Simon Polstra and Joe E. Coffland

Computer Architecture and Parallel Systems group
Dept. of Computer Science, University of Amsterdam
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands

Abstract

The Sesame environment provides modeling and simulation methods and tools for the efficient design space exploration of heterogeneous embedded multimedia systems. It specifically targets the performance evaluation of embedded systems architectures in which task-level parallelism is available. In this paper, we present techniques that allow Sesame to model intra-task parallelism exploited at the architecture level. Moreover, we describe a case study using a QR decomposition application to validate our modeling concepts. To this end, we were able to compare the performance estimates of our abstract system models with the results of an actual FPGA implementation. The results are promising as they show good accuracy with minimal modeling effort.

Key Words: Embedded systems, modeling and simulation, system-level performance evaluation, intra-task parallelism, validation

I. Introduction

Modern embedded systems, like those for media and signal processing, often have a heterogeneous system architecture, consisting of components in the range from fully programmable processor cores to dedicated hardware components. Increasingly, these components are integrated as a system-on-chip exploiting task-level parallelism in applications. Due to the high degree of programmability that is usually provided by such embedded systems, they typically allow for targeting a whole range of applications with varying demands. All of the above characteristics greatly complicate the design of these embedded systems, making it more and more important to have good tools available for exploring different design choices at an early stage in the design.

In the context of the Artemis (ARchitectures and meThods for Embedded Media Systems) project [20], 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. This architecture workbench should allow for rapid performance 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.

In this paper, our focus is on a prototype modeling and simulation environment, called Sesame [19]. According to the Artemis modeling methodology [20], this environment uses separate application models and architecture models and an explicit mapping step to map an application model onto an architecture model. This mapping is realized by means of trace-driven co-simulation, where the execution of the application model generates application events that represent the application workload imposed on the architecture. Application models consist of communicating parallel processes, thereby expressing the task-level parallelism available in the applications. By mapping the event traces generated by different application model processes onto the various system architecture components, this task-level parallelism is exploited at the architecture level. In addition, the underlying architecture may also exploit intra-task parallelism inside a sin-

gle trace. This paper presents the newly added techniques Sesame applies to model architectures that exploit such intra-task parallelism. Moreover, using a case study with the QR decomposition algorithm as application, we demonstrate the effectiveness of our modeling methodology.

The remainder of this paper is organized as follows. Section II briefly describes related work in the area of modeling and simulation of complex embedded systems. Section III gives a general overview of the Sesame modeling and simulation environment, while in Section IV we present a more detailed description of Sesame’s synchronization layer. In Sections V and VI, we describe the methods applied to model intra-task parallelism and discuss their impact on Sesame’s synchronization and architecture model layers. Section VII presents some validation results we obtained from the case study with the QR decomposition application. Finally, Section VIII discusses several open issues and Section IX concludes the paper.

II. Related work

Various research groups are active in the field of modeling and simulating heterogeneous embedded systems, of which some are academic efforts (e.g., [6, 12, 10]) and others commercial [9] and industrial efforts (e.g., [5]). Many efforts in this field *co-simulate* the software parts, which are mapped onto a programmable processor, and the hardware components and their interactions together in one simulation. Because an explicit distinction is made between software and hardware simulation, it must be known which application components will be performed in software and which ones in hardware before a system model is built. This significantly complicates the performance evaluation of different hardware/software partitioning schemes since a new system model may be required for the assessment of each partitioning.

A number of exploration environments, such as VCC [1], Polis [4] and eArchitect [2], facilitate more flexible system-level design space exploration by providing support for mapping a behavioral application specification to an architecture specification. Within the Artemis project, however, we try to push the separation of modeling application behavior and modeling architectural constraints at the system level to even greater ex-

tents. To this end, we apply trace-driven co-simulation of application and architecture models. Like was shown in [19], this leads to efficient exploration of different design alternatives while also yielding a high degree of reusability. The work of [16] also uses 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 co-simulation stage is required in order to generate the traces.

Finally, the Archer project [23] shows a lot of similarities with our work. This is due to the fact that both our work and Archer are spin-offs from the Spade project [18]. A major difference is, however, that Archer follows an entirely different application-to-architecture mapping approach. Instead of using event-traces, it maps symbolic programs, which are derived from the application model, onto architecture resources.

III. The Sesame modeling and simulation environment

The Sesame modeling and simulation environment [19], which builds upon the ground-laying work of the Spade framework [18], facilitates the performance analysis of embedded systems architectures in a way that directly reflects the so-called Y-chart design approach [14]. In Y-chart based design, a 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).

In accordance to the Y-chart approach, Sesame recognizes separate application and architecture models within a 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. 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.

For application modeling, Sesame uses the Kahn Process Network (KPN) model of computation [13] in which parallel processes – implemented in a high level language – communicate 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 is 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. Initially, the application events typically are coarse grained, such as *execute(DCT)* or *read(pixel-block,channel_id)*, and they may be refined as the underlying architecture models are refined. We note that in the remainder of this paper, computational application events will be referred to as *execute events*.

To execute Kahn application models, and thereby generating the application events that represent the workload imposed on the architecture, Sesame features a process network execution engine supporting Kahn semantics. This execution engine runs the Kahn processes as separate threads using the Pthreads package. For now, there is a limitation that the Kahn processes need to be written in C++. In the near future, C and Java support will be added. The structure of the application models (i.e., which processes are used in the model and how they are connected to each other) is described in a language called YML (Y-chart Modeling Language)[8]. This is an XML-based language which is similar to Ptolemy’s MoML [17]

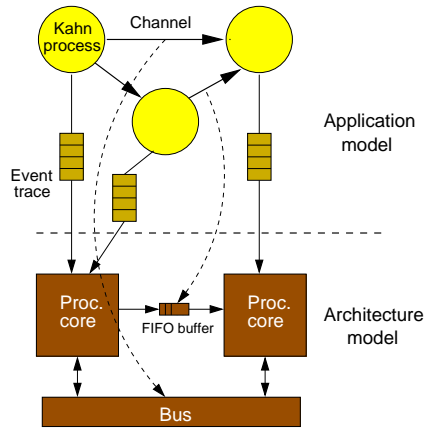


Figure 1. Mapping a Kahn application model onto an architecture model.

but is slightly less generic in the sense that YML only needs to support a few simulation domains. As a consequence, YML only supports a subset of MoML’s features. However, YML provides one additional feature in comparison to MoML as it contains built-in scripting support. This allows for loop-like constructs, mapping & connectivity functions, and so on, which facilitate the description of large and complex models.

The performance of an architecture can be evaluated by simulating the performance consequences of the incoming execute 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 therefore routed towards a specific component inside the architecture model by using a trace-event queue. This is illustrated in Figure 1. Since the application-model execution engine and the architecture simulator run as separate processes*, these trace-event queues are currently implemented via Unix named-pipes. Alternative implementations of the queues, such as using shared memory, are foreseen in the

*Running the application-model execution engine as a separate process also makes it easy to analyze the application model in isolation. This can be beneficial as it allows for investigation of the upper bounds of the performance and may lead to early recognition of bottlenecks within the application itself.

future. If two or more Kahn processes are mapped onto a single architecture component (e.g., when several application tasks are mapped onto a microprocessor), then the events from the different trace-event queues need to be scheduled. The next section explains how this is done.

An 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 template performance models for processing cores, communication media (like busses) and different types of memory. These template models can be freely extended and adapted. All 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 [19]. The structure of architecture models – specifying which building blocks are used from the library and the way they are connected – is also described in YML.

IV. The synchronization layer

When multiple Kahn application model processes are mapped onto a single architecture model component, the event traces need to be scheduled. For this purpose, Sesame provides an intermediate *synchronization layer*, which is illustrated in Figure 2. This layer guarantees deadlock-free scheduling of the application events and forms the application and architecture dependent structure that connects the architecture-independent application model with the application-independent architecture model. The synchronization layer, which can be automatically generated from the YML description of an application model, consists of virtual processor components and FIFO buffers for communication between the virtual processors. There is a one-to-one relationship between the Kahn processes in the application model and the virtual processors in the synchronization layer. This is also true for the Kahn channels and the FIFO channels in the synchronization layer, except for the fact that the buffers of the latter channels are limited in size. Their size is parameterized and dependent on

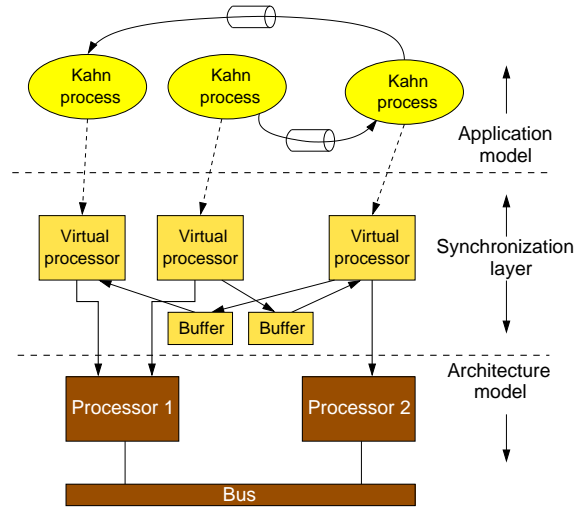


Figure 2. The three layers within Sesame: the application model layer, the architecture model layer, and the synchronization layer which interfaces between application and architecture models.

the modeled architecture. 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. Currently, this virtual processor to architectural processor mapping is specified in the YML description of the architecture model. We are working, however, towards an approach in which this mapping is specified in a separate YML mapping description.

As can be seen from Figure 2, multiple virtual processors can be mapped onto a single processor in the architecture model. In this scheme, execute events are directly dispatched by a virtual processor to the processor model. The latter subsequently schedules the events originating from different virtual processors according to some given policy (FCFS by default) and models their timing consequences. For communication events, however, the appropriate buffer at the synchronization layer is first consulted to check whether or not a communication is safe to take place so that no deadlock can occur. Only if it is found to be safe (i.e., for read

events the data should be available and for write events there should be room in the target buffer), then communication events may be dispatched to the processor component in the architecture model. As long as a communication event cannot be dispatched, the virtual processor blocks. This is possible because the synchronization layer is, like the architecture model, implemented in the Pearl simulation language and executes in the same simulation-time domain as the architecture model. As a consequence, the synchronization layer accounts for synchronization delays of communicating application processes mapped onto the underlying architecture, while the architecture model accounts for the computational latencies and the pure communication latencies (e.g., bus arbitration and transfer latencies). Each time a virtual processor dispatches an application event (either computation or communication) to a processor in the architecture model, it is blocked in simulated time until the event's simulation at the architecture level has finished.

The idea of concentrating synchronization behavior in a synchronization layer and separating it from (the latencies caused by) data transmission behavior is somewhat similar to the synchronization graph concept of [21]. However, our synchronization layer seems to be more flexible since it is dynamically scheduled and behaves like a "Kahn" process network in which the FIFO buffers are bounded. As a consequence of the dynamic scheduling of the synchronization layer and the architecture model (remember that they both are executed in the same discrete-event simulation domain), dynamics at the architecture level such as contention can easily be taken into account within the synchronization layer.

V. Modeling intra-task parallelism

Initially, Sesame only modeled the architecture's processing cores as black boxes which sequentially simulate the timing consequences of the incoming (linear) trace of application events. However, the architecture under investigation may also want to exploit intra-task parallelism which is present in a single event trace from a Kahn application process. For example, a processing element may have multiple communication units which allow for performing independent reads and writes in parallel, or

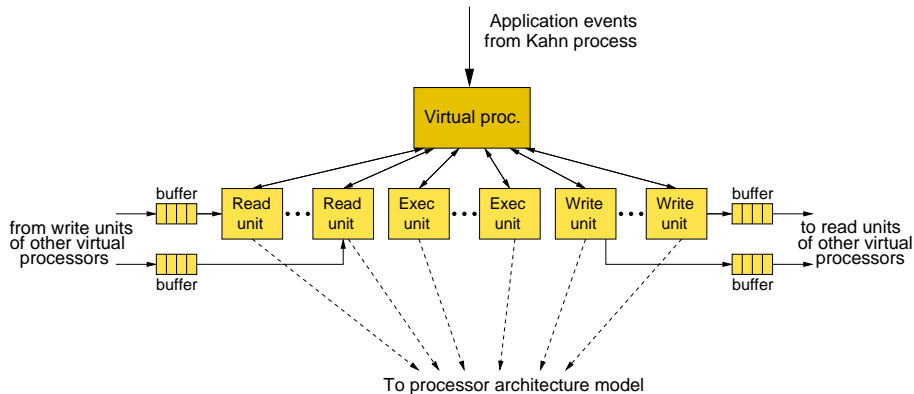


Figure 3. Refining a virtual processor in the synchronization layer.

it may have multiple execution units for concurrently processing independent computations. To support the modeling and simulation of such intra-task parallelism, we extended Sesame’s model library with component models that allow for refining the virtual processors in the synchronization layer and the processor components within the architecture models.

Figure 3 shows how a virtual processor in the synchronization layer, like the ones depicted in Figure 2, is refined. The virtual processor component now acts as a front-end to a range of (virtual) functional units. These functional units consist of read, write and execution units which can operate in parallel. The new virtual processor component has a symbolic-instruction window of parameterizable size in which it stores incoming application events and with which it analyzes them for parallel execution. According to the event type (execute event type, channel from/to which is read/written, etc.), the virtual processor dispatches incoming events to the appropriate functional unit. The number of entries in the symbolic-instruction window limits the number of outstanding (dispatched but not finished) events in the virtual processor. A window size of one implies sequential handling of the application events. In Figure 3, the arrows from the functional units back to the virtual processor refer to the acknowledgments the functional units transmit whenever the simulation of an event has finished.

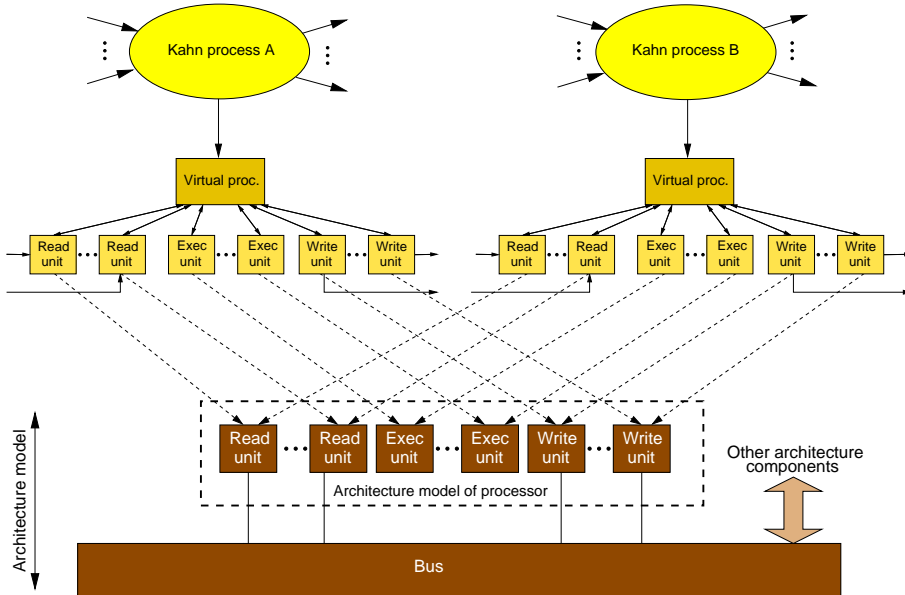


Figure 4. Mapping multiple refined virtual processors onto a refined processor architecture model.

The read and write units are connected via buffers¹ with other virtual processors, like discussed in Section IV, in order to establish the modeling of synchronizations between Kahn application processes in accordance to their mapping onto the underlying architecture. Hence, the read and write units do not dispatch a communication event to the architecture model unless it is safe to do so, i.e., the event cannot cause a deadlock. In addition, the execution and write units do not dispatch their incoming application events to the architecture model before all dependencies for these events are resolved. We will elaborate on this issue in the next section, which discusses the internal synchronizations within a refined virtual processor component.

Figure 4 illustrates how the refined virtual processors can be mapped onto a processor component in the architecture model which has been refined as well. The read units from the virtual processors that are mapped

¹Per read or write unit, there may be multiple buffers connected.

onto the same processor at the architecture level, are connected to the read units of the processor in the architecture model. Likewise, the virtual execution units are connected to the execution units of the processor architecture model, and so on. The functional units in the architecture model may again be black-box models which sequentially account for the timing consequences of the incoming application events dispatched by the synchronization layer. Alternatively, they may also be further refined. For example, a refined execution unit may model internally pipelined execution of execute events. Furthermore, in the example of Figure 4 all communication units in the architecture model are connected to a bus model. In reality, communication units within the architecture model may have different connections with each other (directly across a bus or via shared memory, point-to-point, etc.).

VI. Dataflow for functional unit synchronization

To properly model parallel execution of application events from a single event trace, the dependencies between the events should be taken into account. For example, an execution unit in the synchronization layer may only dispatch an execute event to the execution unit in the architecture model when the read events it depends on have been simulated and delivered the required input for the execute. Likewise, a write event may be dispatched to the architecture model when it is safe to do so and when the read/execute events it depends on have been simulated.

Consider the example in Figure 5(a) in which a virtual processor is shown for a processor architecture with a pipeline of two read units, one execution unit and two write units. In this example, the trace generating Kahn process reads/writes from/to two channels which are mapped onto separate read and write units. The execute events in this example are dependent on the two preceding read events, while the two write events are dependent on the preceding execute event. In Figure 5(b) the resulting pipeline parallelism is illustrated.

The synchronization between the functional units in order to resolve dependencies is done via buffered token channels. In Figure 5(a), for example, the read units have a token channel to the execution unit. A read unit

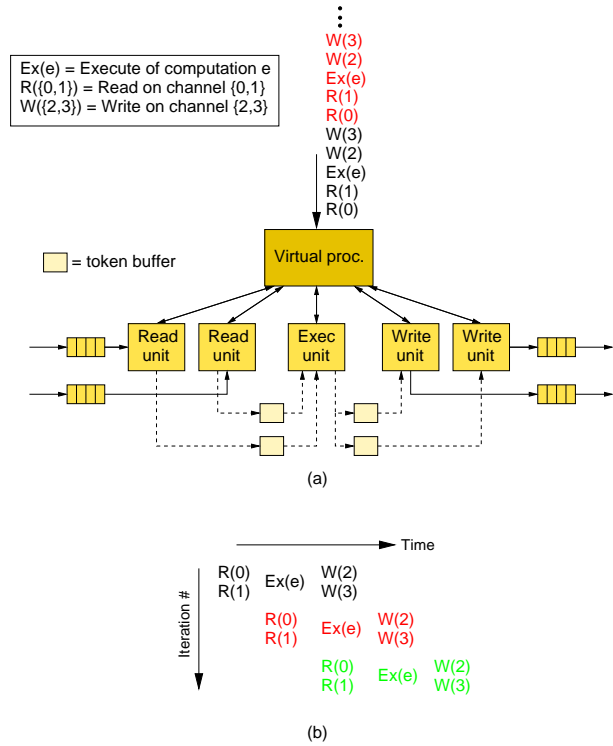


Figure 5. Dataflow-based synchronization to resolve dependencies between functional units in a virtual processor. The architecture shown in (a) exploits pipeline parallelism, which is illustrated in (b).

sends a token along its token channel whenever a read event finished, i.e., has been simulated at architecture level. The size of the token channel's buffer determines how far the read unit can run ahead, or in other words, the amount of internal buffering a read unit has. If the token channel's buffer is full, then the read unit stalls until the execution unit has removed one or more tokens from the channel's buffer. During such a stall, a read unit cannot handle new read events.

In our example, the execution unit reads the tokens generated by the read units. Associated with each execute event type, there are two *bitmaps*. The first one describes on which token channels the particular execute event is dependent, i.e., which read units produce data needed by the execute event.

The second bitmap describes which functional units are dependent on the execute event. So, it relates to output token channels.

The execution unit must have received a token from all of the required token channels, implying that dependencies have been resolved, before the execute event may be dispatched to the architecture model. Likewise, after an execute event has been simulated at the architecture level, the execution unit sends tokens along the required output token channels (as specified by the second bitmap). As a consequence, the write units, which are waiting for tokens from the execution unit, are enabled to dispatch dependent write events to the architecture model. To summarize, synchronizations due to dependencies between functional units in the synchronization layer are handled using the dataflow principle with token transmissions between the functional units. To be more specific, this dataflow mechanism adheres to integer-controlled dataflow [7]. Of course, the placement of token channels between functional units and their buffer sizes are freely adjustable. For the time being, however, we slightly restricted the choice of functional units as we currently assume that there can be only one execution unit per processor. In Section VIII, we come back to this issue and indicate how our modeling concepts may be extended to support multiple execution units per processor.

To give an impression of how the implemented models look like, Figure 6 shows the Pearl code for a read unit from the synchronization layer (the variable declarations have been omitted). As Pearl is an object-based language and architecture components are modeled by objects, the code shown in Figure 6 embodies the class of read unit objects.

In its main loop, the read unit object waits for (using the `block()` primitive) either one of two methods to be called: `sig_room` or `read`. The `sig_room` method is called whenever there is room for a new token in the token buffer that is associated with the read unit. Multiple calls to this method are queued by the Pearl runtime system. The `read` method is called when a read event needs to be processed by the unit. This method first checks if there is room in the token buffer by waiting until there is at least one call to the `sig_room` method queued up. It then synchronously (!) calls the `get` method in the input buffer object that is connected to the read unit. This means that the read unit will block in virtual time until it receives an acknowledgment from the input buffer object, signaling the

```

class v_read_unit

[...]
```

```

sig_room : ()
{ }

read : ()
{
    block(sig_room);           // block until there's room in token buffer
    input_buffer ! get();      // model fetching of data from input FIFO
    ex_unit !! sig_data(unit_id); // send token to execution unit
    virt_proc !! op_done();    // signal completion to virt. processor
}

{
    while (1) {
        block( read, sig_room ); // main loop
    }
}
}

```

Figure 6. Pearl code for a read unit object from the synchronization layer.

end of the data retrieval. Hereafter, the execution unit is signaled by means of an asynchronous method call (‘!!’) to inform it on the availability of the data, i.e., a token is sent. Finally, the virtual processor is signaled that the read unit is ready to receive a new read request. A more thorough explanation of the code is beyond the scope of this paper. Therefore, the interested reader is referred to [19] for a more detailed discussion of a Pearl code sample.

In our implementation, it is straightforward to change the policy defining when token buffers can be read from or written to. More specifically, a functional unit can wait until all of its required tokens are available before it retrieves the tokens from the buffers or it can retrieve a required token whenever it becomes available. In the latter case, the producer of the token may be unblocked earlier and thereby allowing it to proceed with processing new application events.

We note that the synchronizations between functional units are only performed in the synchronization layer and are not needed within the underlying architecture model. This is because once application events are dispatched from the synchronization layer to the architecture model, they are safe to simulate, i.e., they cannot cause deadlocks and their dependencies have been resolved. This scheme nicely fits our approach in which all synchronization overheads are accounted for in the synchronization layer.

VII. A case study: QR decomposition

To validate the previously presented concepts on how to model the exploitation of intra-task parallelism, we have performed a case study using a set of application model instances of the well-understood QR decomposition algorithm. These application models are the result of the Compaan work [15] done at Leiden University. The Compaan tool is able to automatically generate Kahn application models from nested loop programs written in Matlab, which in our case is the QR decomposition algorithm. In addition, it can perform code transformations such as loop unrolling to increase task-level parallelism inside applications [22].

The Kahn application models generated by the Compaan tool are suitable for a direct implementation in hardware on an FPGA. For this purpose, application models are translated into VHDL [11]. This gives us the unique opportunity to validate our abstract architecture models against an actual FPGA implementation. In the VHDL implementation of a Kahn application model, pre-defined node components are connected in a network. This is done according to the connections between the processes in the application model. The node components, which represent the functional behavior of the Kahn processes in the application model, are implemented in a pipelined fashion that is similar to the one shown in Figure 5. Conceptually, this means that each node component contains a number of read and write units and a single execution unit. So, besides exploiting task-level parallelism by the VHDL network of node components, each node component also exploits intra-task parallelism using its internally pipelined architecture.

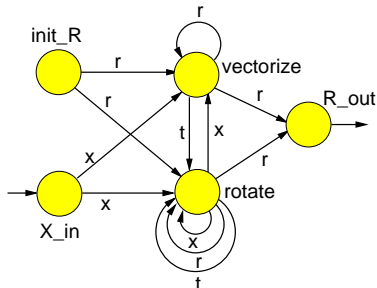
Regarding the QR application, we studied five different instances of its application model generated by Compaan. In each instance, the loops in the code have been unrolled a different number of times. This loop unrolling creates new Kahn processes, thereby increasing the task-level parallelism available in the application [22]. In Figure 7(a), the Matlab code for the QR decomposition algorithm – which is based on the iterative Givens Rotations method – is shown. Figure 7(b) depicts the Kahn application model Compaan generates for this Matlab code when loop unrolling is turned off. Note that the Kahn model does contain processes for input and output routines (e.g., `X_in`) which were omitted in Figure 7(a). Additional infor-


```

for k = 1:1:K,
  for j = 1:1:N,
    [r(j,i),t] = vectorize(r(j,i), x(k,i));
    for i = j+1:1:N,
      [r(j,i),x(k,i),t] = rotate(r(j,i), x(k,i), t);
    end
  end
end
end

```

(a)



(b)

Figure 7. In (a), the Matlab code for the QR decomposition is shown, while (b) depicts the Compaan-generated Kahn process network without loop unrolling.

mation on the Kahn application model of the QR decomposition algorithm can be found in [11]. For each of the application model instances, we described the structure of the application model in YAML to be able to run the model with Sesame’s application-model execution engine. As a side-note, it is worth mentioning that the generation of these YAML descriptions of the application model instances is performed fully automatically by means of a visitor tool.

Our Sesame architecture model, onto which the QR application model instances are mapped, is similar to the VHDL implementation of a Kahn application model in the sense that it also consists of processor components connected in a network with a topology identical to that of the application model. Each processor component is modeled with our refined (virtual) processor model (see Section V) and uses the pipelined architecture as shown in Figure 5(a). Between processor components in the architecture

model there are point-to-point FIFO channels.

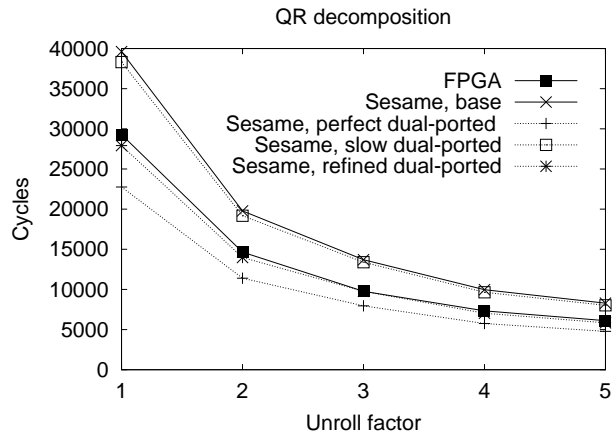
Recall that the structure of Sesame’s architecture models is described in YML. Because of YML’s built-in scripting support, this allowed us to construct a generic reusable template for the refined (virtual) processor model. The processor network in the architecture model is thus obtained by repetitively instantiating this template with possibly different parameters and linking these processor instances together according to the topology of the application model. This topology information is derived from our YML description of the Kahn application model.

A. Experiments

Our first experiments were performed using a Sesame synchronization layer and architecture model with the following characteristics. The size of the FIFO buffers is 256 elements, which guarantees deadlock-free execution of the studied application model instances [11]. The functional units of processor components as well as the FIFO buffers are modeled as black boxes. Read and write operations to the FIFO buffers take 3 cycles each as specified in [11], while all execute events² are handled in a single cycle. The latter reflects the performance of a fully-utilized internal execution pipeline with a single-cycle throughput. Moreover, the token channels between the functional units at the synchronization layer have single-entry buffers. This means that the read and execution units cannot produce more than one result before consumption, i.e., they have only limited internal buffering.

In Figure 8(a), the performance of the FPGA implementation (modeled in VHDL) of the five QR application instances – with loop unroll factors of one to five – is shown. The figure also shows the performance estimates of our black-box Sesame model for these application model instances. These results are referred to as the *base* model in Figure 8. As shown in Figure 8(b), the black-box model yields an average error of 36% and a worst-case error of 40% with respect to the performance results of the FPGA implementation. The Sesame (base) performance estimates show the correct trend behavior but are consistently more pessimistic than those for the

²In the QR application model, the execute events consist of vectorize and rotate operations.



(a)

Difference %	Base model	Dual-ported model		
		Perfect FIFO	Slow FIFO	Refined FIFO
Average	36	-21	32	-3.5
Worst case	40	-22	37	-4.7

(b)

Figure 8. Validation results of our Sesame models for the QR decomposition application against the results from an actual FPGA implementation. The graph in (a) shows the (estimated) performance for five application instances with different loop unroll factors. The table in (b) shows the differences (in %) between estimates from our models and the FPGA numbers.

FPGA.

According to [11], the FPGA buffer implementation is based around a dual-ported RAM, where our base model models single-ported buffers. This explains why the results of the base model are pessimistic. As a next step, we "opened up" the black-box FIFO model and adapted it to include dual-ported behavior. To this end, we modeled three variants of dual-ported

FIFO buffers. Two of these variants represent implementation extremes, while the third one reflects the performance behavior of the actual FPGA implementation. The results of these three dual-ported FIFO models are also shown in Figure 8. The curve labeled with *perfect dual-ported* shows the performance estimates when modeling the FIFO buffers as being perfectly dual-ported. The latter means that read and write operations on a buffer can be performed entirely in parallel, even when the buffer is empty. So, when receiving a read request in the empty buffer state, the read is blocked until a write request is coming in after which the incoming (written) data is immediately forwarded to the reading party. Consequently, both read and write latencies are entirely overlapped.

At the other extreme, the curve labeled with *slow dual-ported* in Figure 8 shows the Sesame performance estimates when modeling dual-ported FIFO buffers which are entirely sequential at the empty state. So, when receiving a read request in the empty buffer state, the read is blocked until a write has occurred and finished writing its data into the buffer (in our model, this takes 3 cycles).

Finally, the curve labeled with *refined dual-ported*, shows the Sesame results when incorporating more detailed knowledge on the actual FPGA buffer implementation into our model. Details on the FPGA implementation indicated that a monolithic 3-cycle read/write latency for the FIFO buffers does not reflect the actual behavior. In reality, the throughput at both sides of a FIFO buffer is 1 operation per 3 cycles, while the read latency turned out to be only 1 cycle. In our *refined dual-ported* model we have therefore split the 3-cycle delay into three 1-cycle delays and placed them at the appropriate places according to specification of the FPGA buffer implementation. This means that we refined the timing within our model while keeping its abstract structure intact.

Three important conclusions can be drawn from the results in Figure 8. First, the results reconfirm the modeling flexibility of Sesame. This is because we were able to model the three dual-ported buffer designs by changing less than ten lines in the code of the base model. Second, the results from the ‘perfect’ and ‘slow’ models – representing the two FIFO buffer implementation extremes – immediately indicate that the average accuracy of Sesame’s performance estimates must lie in the range of -21% and +32%. In fact, our ‘refined’ model demonstrates how close our perfor-

mance estimates can approximate reality since it yields an average error of only 3.5% and a worst case error of 4.7%. Knowing that Sesame targets performance evaluation in an early design stage and therefore models at a high level of abstraction, these accuracy numbers are very promising. Third, our results indicate that the studied hardware implementations of the QR decomposition application are highly sensitive to different FIFO buffer designs. Since the performance estimates of the ‘perfect’ buffer model show a speedup of 68% over the results of the ‘slow’ buffer model, the handling of the empty state in the FIFO buffer seems to be an important design issue.

Since Sesame targets performance evaluation in an early design stage, where the design space that needs to be explored typically is very large, the required modeling effort and the simulation speed of Sesame is worth noting. The architecture models in this case study, including the components in the synchronization layer, consist of less than 400 lines of Pearl code. It takes Sesame about 16 seconds on a 333MHz Sun Ultra 10 to perform the architecture simulation for all five application model instances in a batch.

VIII. Discussion

So far, we have assumed that in the set of functional units of a refined (virtual) processor there is only one execution unit. Processing cores, however, might have multiple execution units that can perform computations in parallel. We are currently investigating whether or not our dataflow approach is sufficient for dealing with dependencies between execution units. In any case, for such inter-execution dependencies we need to extend our dataflow scheme such that tokens are typed, like in the tagged-token model [3]. With the typed tokens, an execution unit can differentiate between the production of results from different execute event types. To support such typed tokens, the bitmaps need to be extended from single-bit values to multiple-bit values to be able to specify which token types are required for an application event.

Moreover, we currently use static bitmaps per execute event type. We found, however, that this causes problems when, for example, execute events of the same type require data from different read units in different

stages of the application model's execution. This can be solved by dynamically adding the bitmap information to the execute events in the traces.

We also intend to investigate whether (aspects from) the work from [23] can be integrated into Sesame since their mapping approach facilitates more easy exposure and specification of intra-task parallelism. This could make the use of explicit bitmaps for execute events entirely redundant.

IX. Conclusions

In this paper, we presented the techniques applied by the Sesame modeling and simulation environment to model intra-task parallelism exploited at the architecture level for task-parallel applications. To this end, our processor models are refined to the level of functional units which can operate in parallel and which are synchronized to resolve dependencies by means of a dataflow mechanism. Using a case study, in which we were able to compare our simulation results with the results from an actual FPGA implementation, we demonstrated that our modeling methodology is flexible and shows good accuracy.

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 the people from the Embedded Real-time Computing group at Leiden University, and in particular Ed Deprettere, Todor Stefanov, Bart Kienhuis, Vladimir Živković and Laurentiu Nicolae, for providing the application models for the QR case study and for their invaluable feedback on this work. In addition, we would like to thank Tim Hariss from QinetiQ Ltd (UK) for providing us with the details on the FPGA implementation of the QR application.

References

1. Cadence Design Systems, Inc., <http://www.cadence.com/>.
2. Innoveda Inc., <http://www.innoveda.com/>.
3. Arvind and K. P. Gostelow. The U-Interpreter. *IEEE Computer*, 15(2):42–49, Feb. 1982.
4. 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.
5. J.-Y. Brunel, E.A. de Kock, W.M. Kruijtzter, 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. J. T. Buck. Static scheduling and code generation from dynamic dataflow graphs with integer valued control streams. In *Proc. of the 28th Asilomar conference on Signals, Systems, and Computers*, Oct. 1994.
8. J. E. Coffland and A. D. Pimentel. A software framework for efficient system-level performance evaluation of embedded systems. To appear in the *Proc. of the ACM SAC conference, Embedded Systems track*, March 2003.
9. P. Dreike and J. McCoy. Co-simulating software and hardware in embedded systems. *Embedded Systems Programming*, 10(6), June 1997.
10. R.K. Gupta, C.N. Coelho Jr., and G. De Micheli. Synthesis and simulation of digital systems containing interacting hardware and software components. In *Proc. of the Design Automation Conference*, pages 225–230, June 1992.
11. T. Harriss, R. Walke, B. Kienhuis, and E.F. Deprettere. Compilation from Matlab to process networks realized in FPGA. In *Proc. of the 35th Asilomar conference on Signals, Systems, and Computers*, Nov. 2001.

12. K. Hines and G. Borriello. Dynamic communication models in embedded system co-simulation. In *Proc. of the Design Automation Conference*, pages 395–400, June 1997.
13. G. Kahn. The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 74*, 1974.
14. 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.
15. B. Kienhuis, E. Rijpkema, and E.F. Deprettere. Compaan: Deriving process networks from Matlab for embedded signal processing architectures. In *Proc. of the 8th International Workshop on Hardware/Software Codesign (CODES'2000)*, May 2000.
16. K. Lahiri, A. Raghunathan, and S. Dey. System-level performance analysis for designing on-chip communication architectures. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 20(6):768–783, June 2001.
17. E. A. Lee and S. Neuendorffer. MoML - a Modeling Markup Language in XML, version 0.4. Technical Report UCB/ERL M00/8, Electronics Research Lab, University of California, Berkeley, March 2000.
18. 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.
19. A. D. Pimentel, S. Polstra, F. Terpstra, A. W. van Halderen, J. E. Cofland, and L. O. Hertzberger. Towards efficient design space exploration of heterogeneous embedded media systems. In *Embedded Processor Design Challenges: Systems, Architectures, MOdeling, and Simulation*, pages 57–73. Springer, LNCS 2268, 2002.
20. 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.
21. S. Sriram and S. S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc., 2000.

22. T. Stefanov, B. Kienhuis, and E.F. Deprettere. Algorithmic transformation techniques for efficient exploration of alternative application instances. In *Proc. of the 10th Int. Symposium on Hardware/Software Codesign (CODES'02)*, pages 7–12, May 2002.
23. V. Živković, P. van der Wolf, E.F. Deprettere, and E.A. de Kock. Design space exploration of streaming multiprocessor architectures. To appear in the *Proc. of the IEEE Workshop on Signal Processing Systems (SIPS'02)*, Oct. 2002.