An IDF-based Trace Transformation Method for Communication Refinement

Andy D. Pimentel Cagkan Erbas
Department of Computer Science, University of Amsterdam
Kruislaan 403, 1098 SJ, Amsterdam, The Netherlands
{andy,cagkan}@science.uva.nl

ABSTRACT

In the Artemis project [13], design space exploration of embedded systems is provided by modeling application behavior and architectural performance constraints separately. Mapping an application model onto an architecture model is performed using trace-driven co-simulation, where event traces generated by an application model drive the underlying architecture model. The abstract communication events from the application model may, however, not match the architecture-level communication primitives. This paper presents a trace transformation method, which is based on integer-controlled data-flow models, to perform communication refinement of application-level events. We discuss the proposed method in the context of our prototype modeling and simulation environment. Moreover, using several examples and a case study, we demonstrate that our method allows for efficient exploration of different communication behaviors at architecture level without affecting the application model.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Modeling Techniques

General Terms

Performance, design

Keywords

Design space exploration, communication refinement

1. INTRODUCTION

Modern embedded systems, like those for media and signal processing, increasingly have a heterogeneous system architecture consisting of components in the range from fully programmable processor cores to dedicated hardware components. These systems often provide a high degree of programmability as they need to target a range of applications with varying demands. Such characteristics greatly complicate the system design, making it more and more important to have good tools available for exploring different design choices at an early stage.

DAC 2003, June 2–6, 2003, Anaheim, California, USA.

In the context of the Artemis project [13], 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 workbench should allow for rapid performance evaluation of different architecture designs, application to architecture mappings, and hardware/software partitionings. In addition, it should do so at multiple levels of abstraction *and* for a wide range of multimedia applications. Key to this flexibility is that separate application and architecture models are used together with an explicit mapping step to map an application model onto an architecture model. This mapping is realized by means of trace-driven co-simulation of the application and architecture models, where the execution of an application model generates application events that represent the application workload imposed on the architecture.

Refinement of architecture models in Artemis requires that the application events driving the architectural simulator should also be refined in order to match the architectural detail. Such refinement should be supported in a way that allows for a smooth transition between abstraction levels, without the need for re-implementing (parts of) the application model. In this paper, we propose a novel method for refining the modeling of communication behavior. This method, which partly builds upon the work of [10], is based on Integer-controlled Data-Flow (IDF) models [5]. Using examples, we show that our method allows for effectively refining synchronization points as well as the granularity of data transfers.

The proposed refinement method currently is prototyped in our Sesame modeling and simulation framework [14, 6], which is being developed in the scope of the Artemis project. Therefore, the next section will first give an introduction to Sesame before starting the discussion on communication refinement. The latter is done in Section 3. We present our IDF-based communication refinement method in Section 4 and its application to a small case study in Section 5. Section 6 describes related work. Section 7 concludes the paper with a discussion on open issues and research challenges.

2. THE SESAME ENVIRONMENT

The Sesame modeling and simulation environment [14, 6] facilitates the performance analysis of embedded systems architectures according to the Y-chart design approach [9, 2]. This means that 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. After explicitly mapping an application model onto an architecture model, they are co-simulated via trace-driven simulation. This allows for evaluation of the system performance of a particular appli-

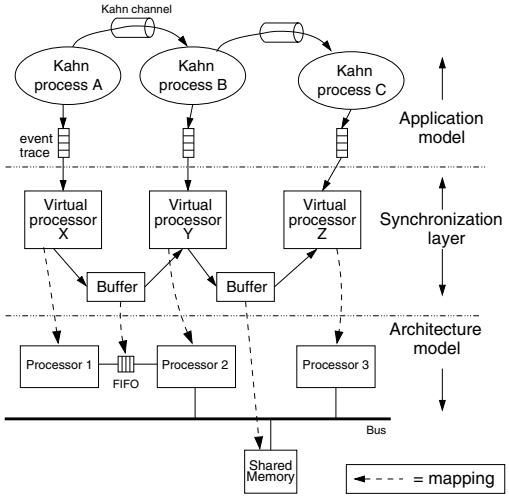

Figure 1: Sesame's application model layer, architecture model layer, and synchronization layer which interfaces between application and architecture models.

cation, mapping, and underlying architecture. The layered infrastructure of Sesame is shown in Figure 1.

For application modeling, Sesame uses the Kahn Process Network (KPN) model of computation [8] 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 its own trace of application events, which is necessary for driving an architecture model. These application events typically are coarse grained, such as *execute(DCT)* or *read(pixel-block,channel_id)*.

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 in the application model, which subsequently drives the architecture simulation. An architecture model is constructed from generic building blocks provided by a library, which contains template performance models for processing cores, communication media (like busses) and various types of memory. 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 [14].

To map Kahn processes (i.e., their event traces) from an application model onto architecture model components and to support the scheduling of application events from different event traces when multiple Kahn processes are mapped onto a single architecture component (e.g., a programmable processor), Sesame provides an intermediate *synchronization layer*. This layer 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 the modeled architecture. A virtual processor reads in an application trace from a Kahn process via a trace event queue 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 freely adjustable. In addition, the buffers from the synchronization layer are also mapped onto the architecture model. In Figure 1, for example, one buffer is placed in shared memory[1] while the other buffer is mapped onto a point-to-point FIFO channel between processors 1 and 2.

The mechanism with which application events are dispatched from a virtual processor to an architecture model component guarantees deadlock-free scheduling of the application events from different event traces [14]. In this mechanism, computation events are directly dispatched by a virtual processor to the architecture component onto which it is mapped. The latter schedules incoming events that originate from different event queues according to a given policy (FCFS by default) and subsequently models their timing consequences. For communication events, however, a virtual processor first consults the appropriate buffer at the synchronization layer 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 component in the architecture model, it is blocked in simulated time until the event's simulation at the architecture level has finished.

3. COMMUNICATION REFINEMENT

The traces of application events from our Kahn processes usually consist of *R(ead)*, *W(rite)* and *E(execute)* events. Consider, for example, Figure 2 which shows the application model of Figure 1 in more detail. The three application processes A, B and C form a pipeline through which data blocks are sent. Let us con-

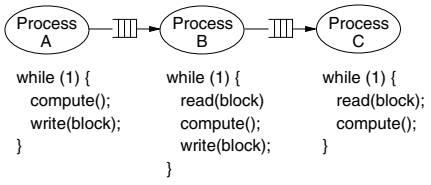

Figure 2: Three application processes in a pipeline.

centrate on Kahn process B of the application model. This process generates an event trace with recurring $R \rightarrow E \rightarrow W$ event sequences, where '$\rightarrow$' denotes 'followed by' and the E event represents the

[1]The architecture model accounts for the modeling of bus activity (arbitration, transfers, etc.) when accessing this buffer.

`compute` in Figure 2. The synchronization layer and architecture model simulate these events atomically (i.e., the simulation of an application event cannot be interrupted by another application event from the same trace) and in strict trace-order. According to Figure 1, Kahn process B is mapped onto processor 2 at the architecture level. Now, suppose that this processor has no local memory and directly operates on its input and output buffers. This means, for example, that processor 2 may only perform an *R* event when the room in its output buffer in shared memory is sufficient to store the results of the computation (as it cannot temporarily store the results locally). Using coarse-grained *R* and *W* events that are simulated atomically such a requirement cannot be modeled.

To allow for modeling communication behavior such as sketched above, the underlying architecture model needs to have more refined communication events. Also, there should be a transformation mechanism that translates the *R* and *W* events from the application model into the finer grained architecture-level events. Such transformations should be possible without affecting the application model, allowing for flexible exploration of different communication behaviors at architecture level. Hence, a natural solution is to place the transformation mechanism in between the application and architecture models, that is, in Sesame's synchronization layer.

For the architecture-level events, we use the events that were proposed for the Spade framework [10]: *CD* (Check Data*), *Ld* (Load data†), *SR* (Signal Room*), *E*, *CR* (Check Room*), *St* (Store data†) and *SD* (Signal Data*). Here, the events marked with * refer to synchronizations while those marked with † refer to data transmissions. A direct translation of *R* and *W* events into the architecture-level events is easily made. When using the notation style of [10] where $\stackrel{\Theta}{\Longrightarrow}$ denotes an event transformation, then:

$$\mathrm{R} \stackrel{\Theta}{\Longrightarrow} \mathrm{CD} \rightarrow \mathrm{Ld} \rightarrow \mathrm{SR} \quad (1)$$

So, an *R* event is functionally equivalent with a *CD* followed by an *Ld* and then an *SR*. Similarly, for *W* events:

$$\mathrm{W} \stackrel{\Theta}{\Longrightarrow} \mathrm{CR} \rightarrow \mathrm{St} \rightarrow \mathrm{SD} \quad (2)$$

The *E* application events remain *E* events at the architecture level:

$$\mathrm{E} \stackrel{\Theta}{\Longrightarrow} \mathrm{E} \quad (3)$$

Implicitly, the separation between synchronizations and data transfers is already realized in Sesame because of the recognition of a synchronization layer and an architecture model layer (see Section 2). However, now we make this separation explicit in order to enable further transformations on the refined events, namely the transformation of synchronization points and the refinement of the granularity of data transfers. For example, the *Ld* and *St* events may be transformed into new *Ld*/*St* events that operate on finer data granularities[2]. Or the relative position of synchronization events in the trace may be changed in order to change the points of synchronization. In addition, multiple synchronization events for the same buffer may be combined into a single one to reduce the number of synchronizations. Conforming Sesame's layered infrastructure, the synchronization events are simulated in the synchronization layer, while the timing consequences of the data-transferring events (*Ld* and *St*) are simulated by the architecture model.

Returning to Kahn process B in Figure 2, its trace with the recurring $R \rightarrow E \rightarrow W$ event sequence can directly be refined using transformations (1), (2) and (3):

$$\mathrm{R} \rightarrow \mathrm{E} \rightarrow \mathrm{W} \stackrel{\Theta}{\Longrightarrow} \mathrm{CD} \rightarrow \mathrm{Ld} \rightarrow \mathrm{SR} \rightarrow \mathrm{E} \rightarrow \mathrm{CR} \rightarrow \mathrm{St} \rightarrow \mathrm{SD} \quad (4)$$

[2]As we will see later on, such a transformation may also require coarse-grained *E* events to be partitioned into smaller *E* tasks to better match the refined data transfers.

However, if we again assume that processor 2 – onto which process B is mapped (see Figure 1) – lacks a local memory, then the availability of its output buffer must be checked first before fetching the data from the input FIFO. To this end, a *CR* event for the output buffer in shared memory needs to be scheduled in front of the *Ld* event. In addition, the FIFO buffer from which processor 2 reads, must be available until the processor has finished operating on it (i.e., after writing the results to the output buffer). The following transformation models the above behavior:

$$\mathrm{R} \rightarrow \mathrm{E} \rightarrow \mathrm{W} \stackrel{\Theta}{\Longrightarrow} \mathrm{CD} \rightarrow \mathrm{CR} \rightarrow \mathrm{Ld} \rightarrow \mathrm{E} \rightarrow \mathrm{St} \rightarrow \mathrm{SD} \rightarrow \mathrm{SR} \quad (5)$$

4. IDF-BASED TRACE TRANSFORMATION

To realize the communication refinements such as discussed in the previous section, we propose a method that is based on Integer-controlled Data-Flow (IDF) models [5]. Refining the communications in the incoming event traces is done by refining the virtual processor components in Sesame's synchronization layer themselves. To this end, an IDF model describes the internal behavior of a virtual processor. In this approach, the incoming event traces from the application model specify *when* and *with whom* a virtual processor communicates, while the internal IDF model within a virtual processor specifies *how* the communication takes place. The IDF models are implemented in the Pearl simulation language and are, like the original synchronization layer, executed in the same simulation-time domain as the architecture model.

In the remainder of this section, we will use two example refinements to describe our IDF-based refinement method. The first example illustrates the basic concepts, whereas the second one shows a more complex refinement in which all our IDF-model building blocks are used.

Let us reconsider the example presented in the previous section, in which Kahn process B (see Figure 2) is mapped onto processor 2 (see Figure 1) that lacks a local memory. Because of this mapping, the $R \rightarrow E \rightarrow W$ event sequences generated by process B should be refined according to transformation (5). Figure 3 shows the IDF model that realizes this refinement[3]. The names of the actors in the IDF model represent their functionality. Some actors are represented by boxes rather than circles because they are building blocks which are, as will be explained, composed of multiple sub-actors. If no explicit number is specified at a channel of an actor, then it is assumed that a single token on that particular channel is consumed/produced. For the sake of discussion, we differentiate between three types of token channels, although all of these channels are functionally equivalent. The *intra-event dependency channels* are the token channels within a single branch of the root switch. They specify the dependencies within the refinement of an application event. For example, in Figure 3, the intra-event dependency channels specify that a *CD* is followed by an *Ld* which again is followed by an *SR*. Opposed to intra-event dependency channels are *inter-event dependency channels* which are token channels between different intra-event branches. They specify the dependencies between refinements of different application events, such as that an *E* must be preceded by an *Ld*. The third channel type (the dashed arrows in Figure 3) will be discussed later on.

At the top of Figure 3, the application trace comes into the IDF model. These trace events are handled as typed (i.e., integer-valued) tokens by the root switch. Dependent on the event type (*R, W* or *E*), a token is transmitted along one of the three branches. Assuming the first event is an *R*, then the *CD*-actor receives a token. To fire,

[3]In Figure 3, only the relevant IDF parts of virtual processors X and Z are shown.

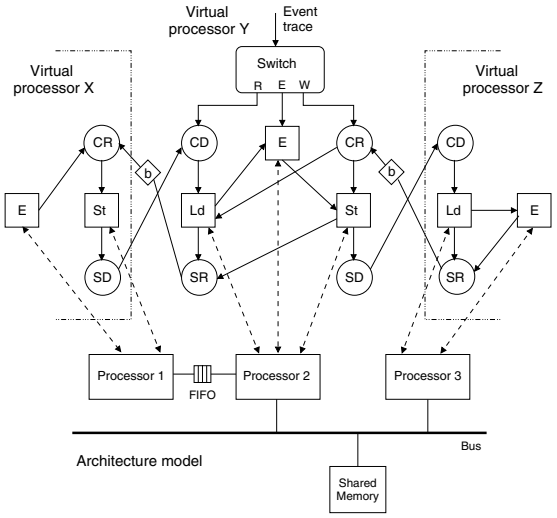

Figure 3: Refining events for processor 2 without local memory.

the actor needs an additional token from an *SD*-actor of the IDF model of virtual processor X. Likewise, *CR*-actors require a token from an *SR*-actor of a remote IDF model. Initially, they can have b tokens on this inter-event dependency channel – called a *delay* and denoted by a $\diamond$ – to model a FIFO buffer with a size of b elements.

Let us return to the *CD*-actor in Figure 3. The firing of this actor simply entails the generation of a token on its output channel. The destination of this token is the *Ld*-actor, which also needs a token on its inter-event dependency channel from the *CR*-actor. This *CR*-actor fires when a *W* trace event was encountered and there is room in the output buffer (i.e., a token from an SR-actor of virtual processor Z is available). The firing of the *Ld*-actor (when enabled by the *CR*-actor) is special as it actually involves two stages of subactors. The same is true for *St* and *E* actors. The decomposition of these actors is depicted in Figure 4, where the X refers to the actor name (either *Ld*, *St* or *E*).

In the first stage of an *Ld*-actor, embodied by the *Ld-init* subactor, a token is sent to the architecture model (see also the dashed arrow in Figure 3). This token is typed as being an *Ld*-token. The performance consequences of this *Ld* event are simulated within the architecture model, after which the architecture simulator sends an acknowledgment token back to the IDF model. The *Ld-exit* stage of the *Ld*-actor, which in the case of Figure 3 produces an intra-event token for the *SR*-actor and an inter-event token for the *E*-actor, only fires when the acknowledgment token from the architecture model is received. Because our IDF models and the underlying architecture model are executed within the same Pearl simulation (and thus are sharing the virtual clock), the token transmissions to/from the

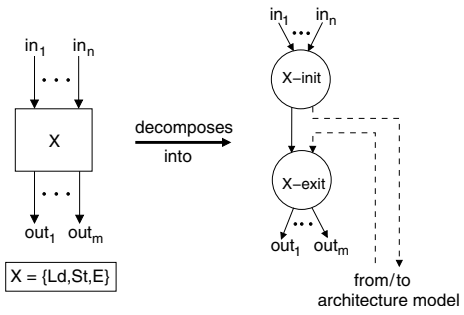

Figure 4: Decomposing *Ld*, *St* and *E* actors.

architecture model yield a *timed IDF model*. For example, the time delay of the *Ld* actor within the IDF model is dependent on the *Ld*-event's simulation in the underlying architecture model. The *St* and *E* actors in the IDF model operate in an identical fashion.

We note that in Figure 3 the *SR* and *SD* event actors are fired in parallel. It depends on the scheduler which of these two actions is performed first. If no timing is involved in these actions (which we assume in this paper), then the order of execution is of no importance. If, however, the order is essential, then the designer can force a scheduling by adding an inter-event dependency channel between the *SR* and *SD* actors. Moreover, if the execution of synchronization events involves a latency, then these events can be modeled using the actor type of Figure 4 where there is a token exchange with the architecture model. The remainder of the actor firings in Figure 3 should be self explanatory.

To further demonstrate the capabilities of our refinement method, we present a slightly more complicated example. Suppose that processor 2 (onto which application process B is mapped, see Figures 1 and 2) is an architecture block that operates at the granularity of lines rather than blocks and that it can only locally store a single line. This means that the processor repetitively reads a line of data, computes on it and outputs a line again. Assume that three lines equal to two blocks and that processor 2 synchronizes its accesses to input and output buffers at this granularity of three lines (= 2 blocks). Processors 1 and 3 still operate on blocks and also synchronize at the granularity of a single block. This means, for example, that after processor 2 has processed and outputted three lines, processor 3 can subsequently read and process two blocks. We note that for such inter-processor communication with mixed grain-sizes, the point-to-point FIFO channel between processors 1 and 2 has been removed and replaced by a buffer in shared memory.

Given the above, processor 2 yields the following behavior for application process B. When three lines of data are available in the input buffer in shared memory and there is room for three lines in the output buffer, then processor 2 starts processing the three lines one after each other. This behavior can be described by the following trace transformation:

$$\begin{aligned} \mathrm{R} \rightarrow \mathrm{E} \rightarrow \mathrm{W} \rightarrow \mathrm{R} \rightarrow \mathrm{E} \rightarrow \mathrm{W} &\stackrel{\Theta}{\Longrightarrow} \\ \mathrm{CD} \rightarrow \mathrm{CR} \rightarrow \mathrm{Ld(line)} \rightarrow \mathrm{E(line)} \rightarrow \mathrm{St(line)} \rightarrow \mathrm{Ld(line)} &\rightarrow \\ \mathrm{E(line)} \rightarrow \mathrm{St(line)} \rightarrow \mathrm{Ld(line)} \rightarrow \mathrm{E(line)} \rightarrow \mathrm{St(line)} \rightarrow \mathrm{SR} \rightarrow \mathrm{SD} & \qquad (6) \end{aligned}$$

So, two subsequent $R{\rightarrow}E{\rightarrow}W$ event sequences from application process B (operating at block granularity and processing two blocks) are needed for the transformation into architecture-level events (processing three lines). Note that the two *E* events from the application model are transformed into three architectural *E* events that operate on lines rather than blocks. In other words, this example also involves refining the grain-size of computational events.

Figure 5 shows an IDF model for the required refinement. The intra-event refinement paths of the *R* and *W* events each contain a *switch* and *select* actor. These actors operate in a cyclo-static fashion, i.e., they alternately read from (select) and write to (switch) input/output channels 0 and 1. By doing so, the *CD* and *CR* actors only fire on the occurrence of even *R*/*W* events. The IDF model also features up/down samplers (*n to m*-actors) that increase/reduce the number of tokens. In Figure 5, the connection to the IDF model of virtual processor Z is also depicted to illustrate the synchronizations between virtual processor Y (operating on lines) and virtual processor Z (operating on blocks). Firing the *SR*-actor in virtual processor Z twice (after reading two blocks), generates one token on the input channel of the *CR*-actor in virtual processor Y, implying that the next three lines can be processed.

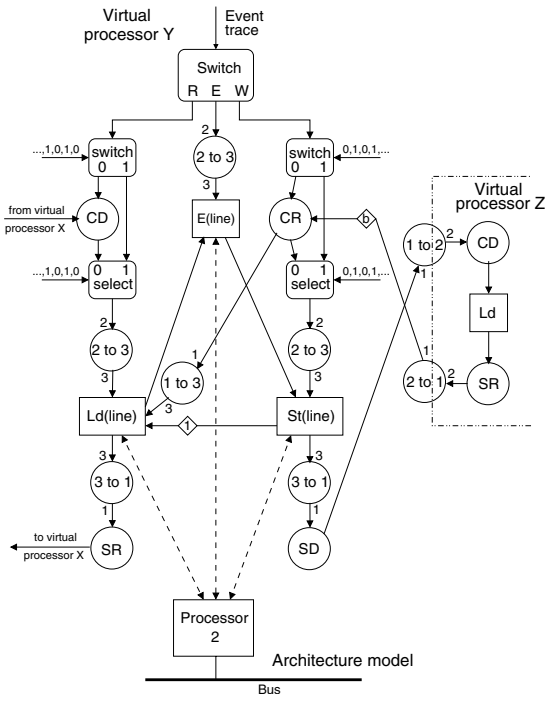

Figure 5: Rescheduling/reducing the points of synchronization and refining the data granularity.

After firing the *CD* and *CR* actors and assuming that two *R* events were encountered, the *Ld(line)*-actor has three tokens on its intra-event dependency channel and can – due to the single-token delay on the intra-event dependency channel from the *St(line)*-actor – be fired for the first time. This firing subsequently enables the *E(line)*-actor to fire once, which in its turn enables the *St(line)*-actor to fire. The latter produces a token for the *Ld(line)*-actor, allowing it to fire for the second time. This circular actor chain is fired three times in a row, after which the *SR* and *SD* actors are fired.

As a final remark, we would like to note that using one or two compound building blocks, the complexity of our IDF models (e.g., the number of up/down samplers) can be reduced significantly.

5. CASE STUDY

To demonstrate the effective usage of the trace transformation method defined in the previous section, we performed a simple case study in Sesame. We have taken the application model given in Figure 2 and evaluated two candidate architectures which are based on the architecture shown in Figure 1. The two architectures are identical except that processor 2 has no local memory in the first architecture, whereas it does have a local memory in the second one. The refinement IDF for the first candidate architecture is already given in Figure 3, while the IDF for the second architecture is shown in Figure 6.

We implemented the Kahn process network of Figure 2 using Sesame's application modeling framework [6]. To this end, we described the topology of the application model in the Y-Chart Modeling Language (YML) which is an XML based model description language specifically developed for describing simulation models in Sesame. The topology of the architecture models, implemented in Pearl, is also described using YML. The reader is referred to [6] for a complete discussion of Sesame's software framework.

The IDF models (see Figures 3 and 6) of the refined virtual processor mapped onto processor 2 have different dependency chan-

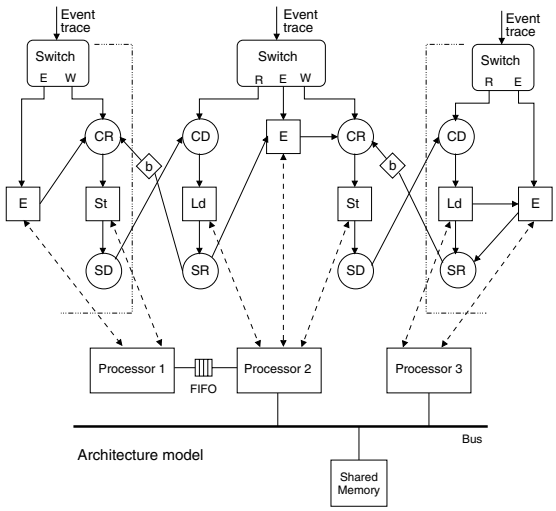

Figure 6: Refining events for processor 2 with local memory.

nels resulting in different refined architecture traces. To give an example, the *CR* in Figure 3 is scheduled before *Ld* and *E*, while in Figure 6 it is scheduled after both *Ld* and *E* events. This is the result of the fact that processor 2 in the architecture of Figure 3 cannot store data locally.

Figure 7 shows the performance estimation results for both architectures. We have performed 10 simulation runs each with different computational workloads associated with E_2 (mapped onto processor 2) and E_3 (mapped onto processor 3) events. As expected, using a processor with a local memory improves total performance, depending on the workload imposed by E_2 and E_3 events. All latencies and processing times in this sample simulation are in abstract time-units. This is because we are illustrating the concept of our trace refinement methodology rather than evaluating a real-life architecture. In Figure 7, we also observe that, when processor 2 has a local memory, the performance is almost only dependent on E_3 if E_2 is less than or equal to E_3. Without a local memory, the performance also depends on E_2. The reason for this behavior follows from the fact that the processor with local memory is less dependent on the execution of the other processors in the pipeline, so the total execution time is dominated by the slowest processor, in this case processor 3. Without local memory, processors in the pipeline are more tightly coupled. We notice this as an increase in total execution time, even when the faster processor, in this case processor 2, is slowed down by increasing E_2.

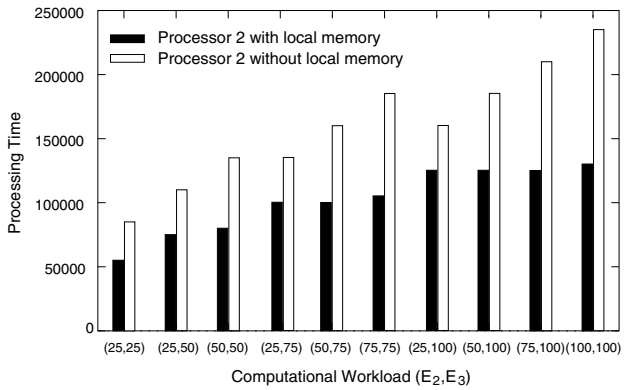

Figure 7: Simulation results for two candidate architectures.

6. RELATED WORK

There are a number of exploration environments, such as VCC [1] and Polis [2], that facilitate flexible system-level design space exploration by providing support for mapping a behavioral application specification to an architecture specification. Within Sesame, which builds upon the ground-laying work of Spade [11], we try to push the separation of modeling application behavior and modeling architectural constraints at the system level to even greater extents. To this end, we apply trace-driven co-simulation of application and architecture models. Like was shown in [14], this leads to efficient exploration of different alternatives.

A fair amount of research has been performed in the field of communication refinement. In [10], a refinement method is proposed in which an application trace is dynamically rewritten into an intermediate partially-ordered trace of architecture-level events. To this end, a number of architecture-independent rewrite rules need to be applied. Then, in a second stage, this intermediate trace is linearized using knowledge on the underlying architecture. However, this work lacks a mechanism – like our IDF models – that actually realizes the refinements. [3] also defines a refinement of the application-level communication primitives into more detailed implementation primitives. However, this refinement does not allow for reordering of operations. A similar approach is taken by VCC [1]. In the framework of [12], low-level communication primitives – similar to our architecture-level primitives – are provided to the application (model) programmer. The drawback of such an approach is that it makes application models architecture dependent. This complicates the reuse of application models and puts extra burden on the application programmer.

7. DISCUSSION

We presented a trace transformation technique that is based on integer-controlled data-flow models. It allows for refining abstract communication events generated by an application model into architecture-level communication primitives. Using examples and a case study, we illustrated that the refinement method enables the performance evaluation of different communication behaviors at the architecture level while the application model remains unaffected.

The refinement examples we have shown are still relatively simple. In reality, the IDF models may become more complex. For example, distinct refinements may be needed for communications on different channels. Or communication patterns for a fixed set of channels may change during application execution. Such behavior calls for combining multiple IDF sub-models (one for each required refinement) in a hierarchy of switches that selects the appropriate refinement. Further study is needed to investigate how this affects our refinement method.

Another important issue is the analyzability of our IDF models. In [4], it is shown that dynamic data-flow models, such as boolean and integer-controlled data-flow models, are hard to analyze statically since many of their analysis problems are undecidable. In our IDF models, the incoming event trace at a root switch acts as a control token stream of which the contents are, in general, hard to predict. However, in many of our target applications, the Kahn processes communicate via fixed, typically recurring, communication patterns. The refinement schemes for each of these communication patterns – when considered in isolation – can be regarded as *cyclo-static* data-flow models [7] rather than IDF models. In that case (like in the examples presented in this paper) we can reason about our models since the pattern of application events entering the IDF model is known a priori, making it possible to switch tokens in a cyclo-static manner. Currently, we are exploring the theoretical aspects of our class of IDF models and experimenting with more realistic case studies.

8. 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 Joseph Coffland, Simon Polstra and Ed Deprettere for their valuable feedback on this work.

9. REFERENCES

[1] Cadence Design Systems, Inc., http://www.cadence.com/.

[2] F. Balarin et al. *Hardware-Software Co-design of Embedded Systems – The POLIS approach*. Kluwer Academic, 1997.

[3] J.-Y. Brunel, E. de Kock, W. Kruijtzer, H. Kenter, and W. Smits. Communication refinement in video systems on chip. In *Proc. 7th Int. Workshop on Hardware/Software Codesign*, pages 142–146, May 1999.

[4] J. T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory using the Token Flow Model*. PhD thesis, Dept. of Electrical Engineering and Computer Sciences, University of California, Berkeley, 1993.

[5] 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.

[6] J. E. Coffland and A. D. Pimentel. A software framework for efficient system-level performance evaluation of embedded systems. In *Proc. of the ACM SAC*, March 2003.

[7] M. Engels, G. Bilsen, R. Lauwereins, and J. Peperstraete. Cyclo-static data flow: Model and implementation. In *Proc. 28th Asilomar Conf. on Signals, Systems, and Computers*, pages 503–507, 1994.

[8] G. Kahn. The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 74*, 1974.

[9] 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.

[10] P. Lieverse, P. van der Wolf, and E. F. Deprettere. A trace transformation technique for communication refinement. In *Proc. of the 9th Int. Symposium on Hardware/Software Codesign*, pages 134–139, Apr. 2001.

[11] 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, Nov. 2001. Special issue on SiPS'99.

[12] A. Nieuwland and P. Lippens. A heterogeneous HW-SW architecture for hand-held multi-media terminals. In *Proc. IEEE Workshop on Signal Processing Systems*, pages 113–122, Oct. 1998.

[13] 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.

[14] A. D. Pimentel et al. Towards efficient design space exploration of heterogeneous embedded media systems. In *Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation*, pages 57–73. Springer tutorial series, LNCS 2268, 2002.