In Proc. of the 3rd Int. Workshop on Systems, Architectures, Modeling, and Simulation, LNCS, Samos, Greece, July 2003.

IDF Models for Trace Transformations: A Case Study in Computational Refinement

Cagkan Erbas, Simon Polstra, and Andy D. Pimentel

Dept. of Computer Science, University of Amsterdam
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
{cagkan,spolstra,andy}@science.uva.nl

Abstract. The Sesame environment provides methods and tools for efficient design space exploration of heterogeneous embedded systems. It uses separate application and architecture models. The application model is explicitly mapped onto the architecture model and they are simulated together, using trace driven co-simulation. Since the abstraction level of the application model may not match the abstraction level of the architecture model, techniques are needed to refine the traces if necessary. In [13], we introduced integer-controlled dataflow (IDF) models to perform trace transformations for communication refinement. This paper uses these trace transformation methods to refine computational events. A simple case study, consisting of a 2D-IDCT application model mapped onto different architecture models, is used to show the capabilities of these IDF modeling techniques.

1 Introduction

Modern embedded systems, like those for media and signal processing, usually 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 even more important to have good tools available for exploring different design choices at an early stage.

In the context of the Artemis project [14], we are developing the Sesame [6] modeling and simulation framework which provides modeling and simulation methods and tools for the efficient design space exploration of heterogeneous embedded multimedia systems. This framework 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.

As a designer gradually refines architecture models in Sesame, the abstraction level of application models does not match the abstraction level of the refined architecture models anymore. At the same time, implementing a new refined application model for every abstraction level at the architecture layer puts too much burden on the application programmer. Due to this overhead, we do not want to hamper the re-usability of our application models and want to be able to use them with different architecture models. Therefore, new techniques are needed within the Sesame environment which will support the gradual refinement of architecture models, without causing any limitation or hindrance on the re-usability of our application models. For this purpose, in [13] we have proposed a new method to refine application traces within the simulation environment. This method combines the utilization of process and dataflow networks within a single simulation environment to perform communication refinement.

In this paper, we show how computational and communicational grain-size refinements can be performed using dataflow networks. These dataflow networks also enable us to model intra-task level parallelism at the architecture layer. To demonstrate this, we make use of a two dimensional inverse discrete cosine transform (2D-IDCT) case study. With this case study, we demonstrate how we accomplish refinement at the architecture layer without changing the application model. The rest of the paper is organized as follows: the next section gives a short introduction of the Sesame environment. Section 3 presents trace transformations and the dataflow actors within Sesame. In Section 4, we demonstrate how we accomplish computational refinement with a simple case study. Section 5 discusses the related work. Finally, Section 6 concludes the paper.

2 The Sesame Environment

The Sesame environment recognizes separate application and architecture models within a system simulation. An application model describes the functional behavior (i.e. computation and communication behavior) of an application. 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 application, mapping and underlying architecture. The layered structure 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 FIFO channels. The workload of an application is captured by instrumenting the code of each Kahn process with annotations. 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. There are three types of application events: *read* and *write* for communication events, *execute* for computation events. These

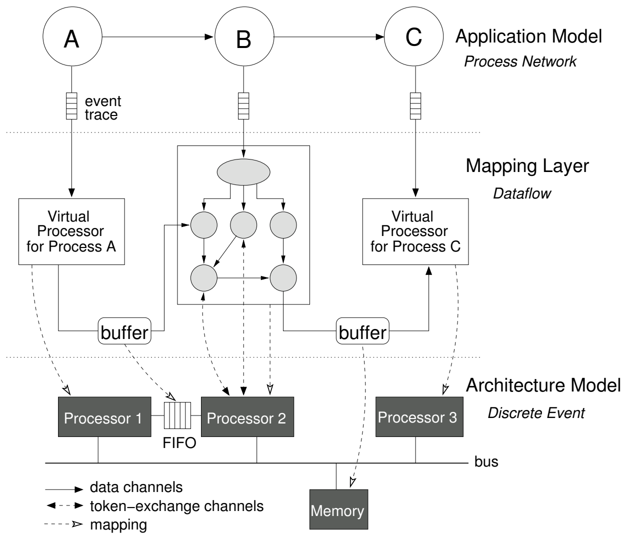


Fig. 1. The three layers within Sesame: the application model layer, the architecture model layer, and the mapping layer which interfaces between the former two.

events are typically coarse-grained like *read(pixel-block,channel-id)*, *write(frame-header,channel-id)*, and *execute(DCT)*.

An architecture model simulates the performance consequences of the computation and communication events generated by an application model. It solely accounts for performance constraints and does not model functional behavior, since the latter is already captured in the application model. 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 in Pearl [6] which is a discrete-event simulation language.

To map Kahn processes from an application model onto architecture model components, Sesame provides an intermediate *mapping layer*. This layer consists of virtual processors communicating via FIFO buffers. There is a one-to-one relationship between the Kahn processes in the application model and the virtual processors in the mapping layer. 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. When multiple Kahn processes are mapped onto a single architecture component, the event traces need to be scheduled. For computation events, a given policy (FCFS by default) is used. In the case of communication events, the appropriate buffer at the mapping layer is first checked whether or not a communication event is safe to occur so that no deadlock is introduced. Only when it is found to be safe (i.e., available data for read events and enough space for write events in the target buffer), communication events are dispatched to processor components in the architecture model. If a communication event cannot be dispatched, the virtual processor blocks. This means that the mapping layer accounts for synchronization latencies, while the architecture layer accounts for pure communication latencies. This is possible since both the mapping and the architecture layers are implemented in Pearl.

With the introduction of gradual refinement of architecture model components, the virtual processors are also refined with dataflow networks. The latter allows us to do simulation at multiple levels of abstraction, without modifying the application model. In Figure 1, we express this fact by refining the virtual processor for the process B with a fictive dataflow network. In the next section, we provide more insight on this refinement approach by explaining relations between the trace transformations for refinement and dataflow actors at the mapping layer.

3 Trace Transformations in Sesame

Refining architecture model components requires that the application events driving them should also be refined to match the architectural detail. Since we want smooth transition between different abstraction levels, we do not want to re-implement (parts of) the application models for each abstraction level. The salient way to accomplish this in Sesame is to refine the virtual processors with dataflow actors. This way, the coarse-grained application events (in the application trace) can be refined up to the desired abstraction level at the mapping layer, and subsequently be used to drive the architecture model components.

In Sesame, SDF [9] actors are utilized for trace transformations. Such a trace transformation refines application-level operations (or events) into finer-grained architecture-level operations. IDF [5] actors are subsequently used to model repetitions and branching conditions which may be present in the application code. However, as in [13], they may also be utilized within static transformations to achieve less complicated (in terms of the number of actors and channels) dataflow graphs. To give an example, the following trace transformations refine *read* (R) and *write* (W) operations such that the synchronizations are separated from real data transfers as follows [11]:

$$R \overset{\Theta_{ref}}{\Longrightarrow} cd \rightarrow ld \rightarrow sr, \tag{1}$$

$$W \overset{\Theta_{ref}}{\Longrightarrow} cr \rightarrow st \rightarrow sd \ . \tag{2}$$

Here refined architecture-level operations *check-data*, *load-data*, *signal-room*, *check-room*, *store-data*, *signal-data* are abbreviated as *cd*, *ld*, *sr*, *cr*, *st*, *sd*, respectively. The arrows between these indicate the ordering relations. The purpose of such refinements is that it, for instance, allows for moving synchronizations when a *pattern* of operations is transformed [13]. For example, in the following transformation we early check room to store data, which may be useful for a processor that cannot store data locally,

$$R \rightarrow E \rightarrow W \overset{\Theta_{ref}}{\Longrightarrow} \mathrm{cd} \rightarrow \mathrm{cr} \rightarrow ld \rightarrow E \rightarrow st \rightarrow sr \rightarrow sd \ . \tag{3}$$

In Figure 2, we give two dataflow graphs implementing this transformation. If the SDF actor T in Figure 2(a) is invoked initially, and if the rest of the SDF actors are also invoked in the order in which they appear on the right hand side of the equation (3), this simply constructs a *valid* schedule for the SDF graph in Figure 2(a). In all cases where the outcome of the trace transformation is a total ordering, a valid schedule is guaranteed to exist and it is constructed as explained. In general, a trace with a total ordering is called a *linear trace.* At this point we should recall that the schedules which are both *admissible* and *periodic* are valid [3]. It is easy to verify that the constructed schedule for this graph is both admissible and periodic. We see that all the actors are immediately fireable when they are invoked in the order they appear in the schedule. This makes the schedule admissible. It is also observed that when all the actors are fired, the graph returns its original state, i.e. the number of tokens on the channels remains the same, which means that the schedule is also periodic. To see this, one may alternatively write the balance equations. The same is true for the graph in Figure 2(b), in which a similar valid schedule can be easily constructed by replacing the single appearance of SDF actor T with three consecutive occurrences of the IDF switch actor.

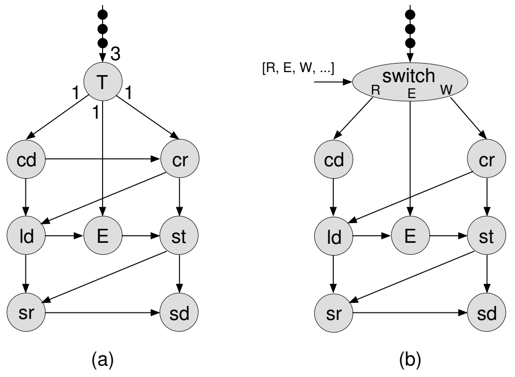


Fig. 2. (a) An SDF graph for a trace transformation in Sesame (b) An IDF graph implementing the same transformation.

In Sesame, all dataflow graphs to implement linear trace transformations are constructed in this manner, so we always end up with valid schedules. We also make use of multiphase and/or multistage IDF actors, like REPEAT-BEGIN, REPEAT-END, which were first proposed in [5], to further simplify our dataflow graphs. Although IDF graphs have analyzibility problems, in Sesame we try to overcome this problem by constructing a static schedule. In most of the cases, this is possible because we may run the application model beforehand and obtain trace files representing the workload of the Kahn processes. However, one can also co-simulate the application and architecture models and schedule the IDF actors dynamically. But in this case, one cannot guarantee whether the execution is completed in finite-time or whether it is performed within bounded memory [4].

In this paper, we are interested in refining grain sizes of both computation and communication events and subsequently model parallel execution (intra-task parallelism) of these events at the architecture level. More specifically, as our Kahn application models often operate on blocks of data, we look at the following transformations,

$$R \overset{\Theta}{\Longrightarrow} R(l) \rightarrow \ldots \rightarrow R(l), \tag{4}$$

$$E \overset{\Theta}{\Longrightarrow} E(l) \rightarrow \ldots \rightarrow E(l), \tag{5}$$

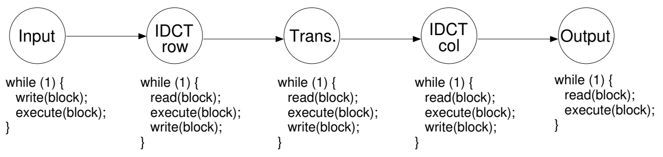
$$W \overset{\Theta}{\Longrightarrow} W(l) \rightarrow \ldots \rightarrow W(l), \tag{6}$$

$$E(l) \overset{\Theta}{\Longrightarrow} e_1 \rightarrow \ldots \rightarrow e_n \ \ . \tag{7}$$

In the first three transformations, read, execute and write operations at the block level are refined to multiple (e.g., 1 block = 8 lines) corresponding operations at the line level. We represent line level operations with an 'l' in parenthesis. The last transformation further refines execute operations at line level to model multiple pipeline execute stages inside a single processor. In (7), refinement for a processor with n-stage pipeline execution is given.

4 Case Study

In Figure 3, the application model for a 2D-IDCT case study is given. All the Kahn processes in the application model operate at block level, i.e. they read/write and execute operations on blocks of data. The Input process writes blocks of data for the IDCT-row process which in turn reads blocks of data, executes IDCT, and writes blocks of data. The Transpose process simply performs a matrix transpose to prepare data for the IDCT-col process.



```
while (1) {
  write(block);
  execute(block);
}
```

```
while (1) {
  read(block);
  execute(block);
  write(block);
}
```

```
while (1) {
  read(block);
  execute(block);
  write(block);
}
```

```
while (1) {
  read(block);
  execute(block);
  write(block);
}
```

```
while (1) {
  read(block);
  execute(block);
}
```

Fig. 3. Kahn process network for the 2D-IDCT case study.

We investigate two alternative architecture models, given in Figure 4, for this application model. Both architecture models have the same four processing elements, PE1 to PE4. The mapping of Kahn processes onto the processing elements is identical in both cases, they differ in how these communicate. In the first architecture model, they are connected via dedicated buffers while in the second architecture a shared memory is used. Each processing element is able to perform read, execute and write operations in parallel, so it can perform its task in a pipelined fashion. Input and Output processes are mapped onto PE1 and PE4, IDCT-row and IDCT-col are mapped onto PE2 and PE3, respectively. The Transpose process is not mapped onto anything, since its functionality is simply implemented as follows: the processing element on which the IDCT-row process is mapped simply writes rows of data to the memory while the processing element on which the second IDCT process is mapped reads columns of data from the memory. We should note that this type of implementation of the matrix transpose forces those processing elements, operating at line level (as we will explain later on in this section), to be synchronized at block level. This is because the second processing element cannot start processing lines until the first one is finished with the last line of data.

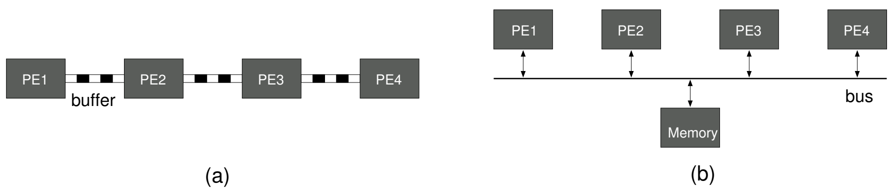


Fig. 4. Two different target architectures.

In both architectures, we modeled PE1 and PE4 to operate at block level. We first modeled PE2 and PE3, in the target architectures operating at line level, at a more abstract block level and then later refined them to operate at line level. For this reason, in the latter case, the application events from the IDCT processes need to be refined. The pattern to be refined for the IDCT processes is $R \to E \to W$. For simplicity if we assume 1 block = 2 lines then,

$$\begin{tikzcd}
R \to E \to W \overset{\Theta}{\Longrightarrow} & R(l) \arrow[r] \arrow[dr] & E(l) \arrow[r] \arrow[dr] & W(l) \arrow[dr] & \\
& & R(l) \arrow[r] & E(l) \arrow[r] & W(l)\ .
\end{tikzcd} \tag{8}$$

If we define that PE2 and PE3 are processing elements with 2-stage pipeline execution units, which creates an execution pipeline inside the previously mentioned task pipeline, then from (7) with $n = 2$ we obtain,

$$\begin{tikzcd}
R \to E \to W \overset{\Theta}{\Longrightarrow} & R(l) \arrow[r] \arrow[dr] & e_1 \arrow[r] \arrow[dr] & e_2 \arrow[r] \arrow[dr] & W(l) \arrow[dr] & \\
& & R(l) \arrow[r] & e_1 \arrow[r] & e_2 \arrow[r] & W(l)\ .
\end{tikzcd} \tag{9}$$

Table 1. Parameters for simulation.

	non-refined, shared	refined, shared	non-refined, fifo	refined, fifo
pipeline size	–	3/8	–	3/8
PE2, PE3 exec. lat.	300	13/5	300	13/5
PE2, PE3 data size	64	8	64	8
PE1, PE4 data size	64	64	64	64
fifo lat.	–	–	1 . . . 60	1 . . . 60
memory lat.	1 . . . 60	1 . . . 60	–	–
memory width	8	8	–	–
bus setup lat.	1	1	–	–
bus width	8	8	–	–

In Table 1, we give the simulation parameters. We have performed four simulations represented by the four columns in the table. The terms *non-refined* and *refined* indicate whether the processing elements PE2 and PE3 operate at block level or at line level. The terms *fifo* and *shared* refer to the architectures in Figures 4(a) and 4(b), respectively. Execution latency is measured in cycles, and data size in bytes. Memory and FIFO latencies are given in cycles/line, where 1 line is 8 bytes and 8 lines make up a block. We note that in these experiments the ratios between the parameters are more important than the actual values being used. We assume that executing a 1D-IDCT takes 300 cycles per block on a non-refined execution unit, so in a 3-stage pipelined execution unit operating on lines, the amount of work is divided by the number of stages, and by the number of lines in a block. So the execution latency of one stage in the 3-stage pipeline is 13 cycles and that of the 8-stage is 5 cycles.

In Figure 5, we give the performance graph obtained when we map the 2D-IDCT application onto the architectures in Figure 4. In all experiments, we have processed 500 blocks of input data. In the first experiment, the processing elements PE2 and PE3 operate at block level and no refinement is performed. This gives us performance results for single and double buffer implementations, i.e. where the double buffer is a 2-entry buffer so that the producer can write to it and the consumer can read from it, simultaneously. We change a single buffer to a double buffer model, simply by adding an extra initial token at the mapping layer. In the second experiment, we refined the processing elements PE2 and PE3 in the architecture model, and explored four alternative cases. For these two processing elements, we have used a 3-stage and an 8-stage execution pipeline.

For the buffers, we have again experimented with single and double buffer implementations. When we compare the single and double buffer performance of the non-refined models, we observe that it is the same until point A. After that point, as the communication latency increases, the single buffer model becomes communication bounded. The performance of the double buffer model is affected by the increased communication latency at point B, when the time to transfer a block of data becomes equal to the time it takes to perform an IDCT on a block of data. When we compare the refined models with the non-refined models, we observe that once the communication becomes a bottleneck (point A for single buffer and point B for double buffer), the advantage of having a pipelined exe-

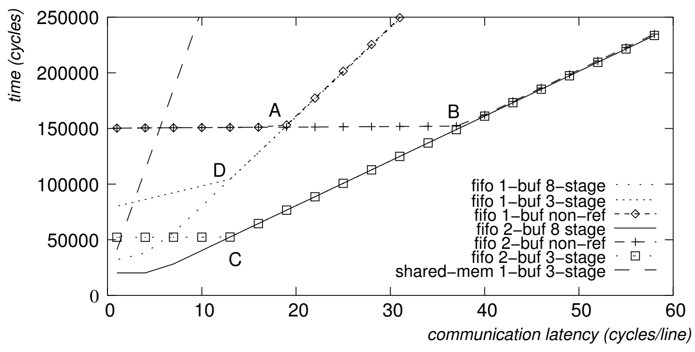


Fig. 5. Performance results for the FIFO architecture.

cution unit disappears. When the models become communication bounded, the non-refined and refined models predict the same performance numbers. We note that a similar situation occurs at points C and D, when increased communication latencies negate the effect of having a longer pipeline. Finally, when we compare these results with the results of the shared memory architecture, we observe that in the latter, the performance is very quickly bounded by the communication, because increasing the communication latency causes contention on the shared bus. This makes the effect of pipelined execution very limited. For this reason, we only present the graph for the refined case with the 3-stage pipeline.

5 Related Work

Within the context of embedded systems and hardware/software codesign, as the search for models of computation continues [10], many system-level design and simulation environments [1], [14] together with system modeling languages [2], [7] have been developed. In parallel to the exploration environments that facilitate the idea of separate functionality and architecture, in Sesame we try to push this separation to even greater extents. This is achieved by an architecture-independent application model, an application-independent architecture model and a mapping step which relates these models for co-simulation. Besides, within the same simulation environment, we use multiple models of computation. It is chosen specifically in accordance with the task to be achieved. As already shown in this paper, we use process networks for application specification, dataflow networks for certain tasks at the mapping layer (e.g., trace transformations) and a discrete-event simulator to model processing components for fast simulations.

Both the Spade [12] and Archer [15] environments show a lot of similarities with the Sesame environment in the sense that they share the same philosophy by recognizing separate application and architecture models. However, each of these environments uses its own architecture simulator and follows a different mapping strategy for co-simulation.

6 Conclusion

In this paper, we showed how computational and communicational grain-size refinement can be performed using the IDF models which was first proposed in [13] for communication refinement. Performing a simple case study, we illustrated how easily we could model task-level parallelism and intra-task parallelism at the architecture layer. Additionally, using similar dataflow graphs, we could also model pipeline execution stages inside a single processor. While doing this, we kept the application model unaffected. Currently, we are testing our new methodology on more realistic real-life media applications. We are especially interested in investigating its efficiency in terms of modeling and simulation time when more complex applications and transformations are considered.

References

1. F. Balarin et al. Metropolis: An integrated electronic system design environment. *IEEE Computer*, Apr. 2003.
2. L. Benini et al. SystemC cosimulation and emulation of multiprocessor SoC designs. *IEEE Computer*, Apr. 2003.
3. S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, 1996.
4. J. T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory using the Token Flow Model*. PhD thesis, Dept. of EECS, UC 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 Symposium on Applied Computing*, Mar. 2003.
7. D. Gajski et al. *The SpecC Language*. Kluwer Academic Publishers, 1997.
8. G. Kahn. The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress*, 1974.
9. E. A. Lee and D. G. Messerschmitt. Synchronous Data Flow. *Proc. of the IEEE*, Sep. 1987.
10. E. A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Trans. on CAD*, Dec. 1998.
11. P. Lieverse et al. A trace transformation technique for communication refinement. In *Proc. of the IEEE/ACM CODES*, Apr. 2001.
12. P. Lieverse et al. A methodology for architecture exploration of heterogeneous signal processing systems. In *Proc. of the IEEE Workshop SiPS*, Oct. 1999.
13. A. D. Pimentel and C. Erbas. An IDF-based trace transformation method for communication refinement. In *Proc. of the ACM/IEEE DAC*, June 2003.
14. A. D. Pimentel et al. Exploring embedded-systems architectures with Artemis. *IEEE Computer*, Nov. 2001.
15. V. Živković et al. Design space exploration of streaming multiprocessor architectures. In *Proc. of the IEEE Workshop SiPS*, Oct. 2002.