A Trace Transformation Technique for Communication Refinement

Paul Lieverse
Dept. of Information Technology and Systems
Delft University of Technology
The Netherlands
p.lieverse@its.tudelft.nl

Pieter van der Wolf
Philips Research
Eindhoven, The Netherlands

Ed Deprettere
Leiden Institute of Advanced Computer Science
Leiden University
The Netherlands

ABSTRACT

Models of computation like Kahn and dataflow process networks provide convenient means for modeling signal processing applications. This is partly due to the abstract primitives that these models offer for communication between concurrent processes. However, when mapping an application model onto an architecture, these primitives need to be mapped onto architecture level communication primitives. We present a trace transformation technique that supports a system architect in performing this communication refinement. We discuss the implementation of this technique in a tool for architecture exploration named SPADE and present examples.

1. INTRODUCTION

In the design of embedded signal processing systems, such as digital televisions, set-top boxes, and mobile devices, a key design step is the definition of a hardware/software architecture that correctly implements the required behavior of the system. In a structured design methodology first the functional behavior of the system is specified and validated using executable functional models. We refer to such models as *application models*. Subsequently, a (heterogeneous) *architecture* is proposed onto which the functional behavior is to be *mapped*. In this stage a designer needs support for architectural exploration and for validation of proposed architectures given the specified functional behavior.

In this paper we focus on the mapping and exploration stage in the design of embedded signal processing systems. In particular, we study the problems associated with the mapping of primitives used for expressing communication behavior at the application level onto primitives used to implement the communication in architectures. In the next section we discuss the application–architecture mapping in more detail and explain a number of fundamental problems associated with this mapping task. We have solved these problems in the context of SPADE, a method and tool for the modeling and exploration of embedded signal processing systems [9]. We introduce the SPADE methodology in Section 3. Our solution is based on trace transformation techniques, which we present in Section 4. We present an example in Section 5 and discuss related work in Section 6. We present conclusions in Section 7.

2. PROBLEM DEFINITION

Different formalisms can be used for application modeling, each having their strengths and weaknesses depending on the targeted application domain and the kind of validation that needs to be performed. Examples are FSM (finite state machine) models and dataflow models. Such formalisms are often referred to as *models of computation*. Models of computation well suited to the domain of signal processing are Kahn process networks [6] and dataflow process networks [8].

In the Kahn model, concurrent *processes* communicate via unbounded FIFO *channels*. Each process performs sequential computation on its private state space. The computation actions of a process are interleaved with communication actions that read data from input channels and write data to output channels. The Kahn model fits nicely with signal processing applications as it conveniently models *stream processing* and as it guarantees that no data is lost in communication. Kahn process networks are *deterministic*, i.e., the stream of data that travels along each channel is determined by the input data; it does not depend on the order in which the processes are executed. As a result, application programmers can easily combine processes into process networks. Dataflow process networks are a special case of Kahn process networks.

The Kahn and dataflow process network models permit applications to be modeled relatively independent of a specific target architecture. This enables *reuse* of application models and permits companies to build libraries of reusable *functional IP*. In particular, the primitives used for communication between processes abstract from implementation aspects that need to be addressed later in the design trajectory. For example, a *read* operation blocks until data is available and then copies data from the FIFO buffer into the private state space of the process. Similarly, a *write* operation copies data from the private state space into a FIFO. The application programmer does not have to worry about such issues as synchronization with other processes, physical locations of buffers, sharing of interconnect or memory resources, etc. Besides promoting reuse, this abstraction also helps application programmers to efficiently perform the task at hand: functional modeling. However, when a system architect departs from an application model to explore candidate architectures, he needs to take into account how the application level primitives are implemented at the architecture level.

We illustrate this mapping problem by means of the following example. Consider the application model shown in Figure 1. Now suppose that this application is mapped onto an architecture consisting of a CPU and a dedicated coprocessor, both connected to a bus and shared memory, as depicted in Figure 2. Furthermore, suppose that the coprocessor has a local memory. The producer is mapped onto the CPU; the consumer onto the coprocessor. The producer does its computation on its private state space which is allocated in shared memory. The write call of the producer copies the results of the computation from this private state space into a

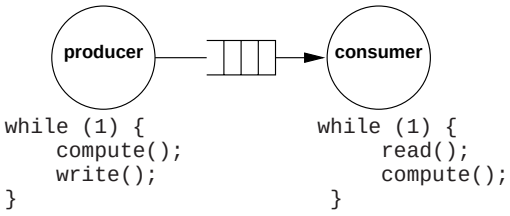


Figure 1: Producer-consumer application.

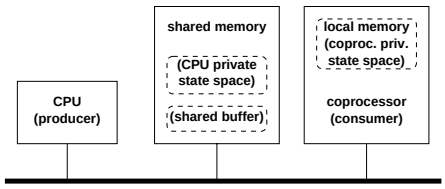


Figure 2: Shared memory architecture for the producer–consumer application.

shared buffer, which is also allocated in shared memory. The consumer then can read this data from the shared buffer into its private state space, located in the local memory of the coprocessor, and do the computation on the data in its private state space.

A different implementation would be to let the producer claim room in the shared buffer before starting with the computation, so that resulting data can be written to the shared buffer directly when it becomes available during the computation. Figure 3 shows this behavior. Note that we now have separated *data transfer* (store-

```
while (1) {
    check_room();
    compute();
    store_data();
    signal_data();
}
```

```
while (1) {
    check_data();
    load_data();
    signal_room();
    compute();
}
```

Figure 3: Behavior of producer-consumer application with a different mapping onto the architecture of Figure 2.

data/load-data) and *synchronization* (check-room/check-data and signal-data/signal-room) and made both these aspects of communication explicit. The *read* and *write* operations implicitly combine these aspects. Also note that the change from the behavior of the producer of Figure 1 to that of Figure 3 is more than a simple substitution of the write operation. For the producer this implementation avoids the copy operation from its private state space into the shared buffer, thereby avoiding the extra CPU cycles, bus traffic, and buffer space.

For an implementation in which we do not have a local memory in the coprocessor, the consumer should also operate directly on the data in the shared buffer. This behavior is shown in Figure 4. It correctly models that the data in the shared buffer must remain available until the computation has completed: `signal_room` after `compute`. In a final implementation the transfer of data to the coprocessor and the computation may be interleaved at a finer grain. This is approximated by the coarse grain `load_data` and `compute`. The synchronization behavior, however, is modeled correctly thereby allowing architecture exploration with reasonable accuracy.

For reasons of efficiency and reuse, application developers should not model their applications at the level of detail shown in Figure 4. They should produce a more architecture independent specification using read and write operations as shown in Figure 1. Architectural exploration tools must then support a system architect in the evaluation of candidate architectures starting from such abstract models.

```
while (1) {
    check_room();
    compute();
    store_data();
    signal_data();
}
```

```
while (1) {
    check_data();
    load_data();
    compute();
    signal_room();
}
```

Figure 4: Behavior of producer-consumer application when mapped onto an alternative architecture.

Specifically, such tools must allow the system architect to model the selected communication architecture and how the application level communication operations are mapped onto the architecture level communication operations. Performance analysis tools must then provide accurate feedback on the performance of application-architecture-mapping combinations.

3. SPADE

SPADE (System level Performance Analysis and Design space Exploration) [9] is a method and tool for architecture exploration of heterogeneous signal processing systems. It is based on the Y-chart [1][7] paradigm. Following the Y-chart, a clear distinction is made between *applications* and *architectures*, which are related via an explicit *mapping* step. SPADE provides techniques for modeling applications and architectures, as well as for capturing the mapping of application models onto architecture models. For application modeling the Kahn process network model is used [4]. For architecture modeling a library of architecture building blocks is provided, of which the timing behavior can be simulated using a cycle-based simulator. SPADE employs a *trace-driven simulation* technique to co-simulate an application model with an architecture model in order to evaluate the performance of the combined system. The workload of an application is captured in *traces*; each process in the application generates a single trace. A trace contains symbols, called *trace operations*, that represent the computation and communication operations that are performed by an application when it is executed. For example, the producer behavior given in Figure 1 would result in a trace consisting of an infinite sequence of `compute` and `write` operations. Data dependent behavior, which is the result of control structures in the application code, is captured by the traces. The resources in an architecture accept trace operations as the workload to be executed. The traces drive the computation and communication activities in the architecture, for which the architecture model accounts time and reports performance data. This is illustrated in Figure 5.

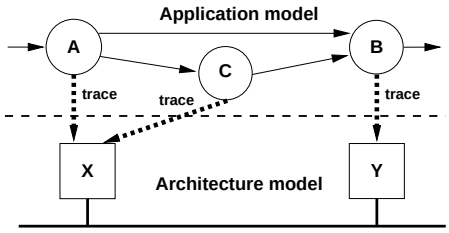


Figure 5: Trace-driven simulation: the execution of the architecture model is driven by traces from the execution of the application model.

The traces that are generated by an application model consist of three types of application level trace operations: read R, write W, and execute E. The read R and write W operations represent the use of Kahn communication primitives. The execute E operation represents computation done in a process (the `compute` function used above).

The architecture model accepts seven types of architecture level operations: check-data *cd*, load-data *ld*, signal-room *sd*, check-room *cr*, store-data *st*, signal-data *sd*, and execute *E*. The execute *E*

operation represents the computation done; during simulation it results in a computation delay specified by the system architect. The load-data *ld* and store-data *st* operations represent the actual transfer of data in the architecture. The other primitives are synchronization primitives. The check-data *cd* and check-room *cr* operations stall a process until data or room is available. The signal-room *sr* and signal-data *sd* operations are their counterparts; they signal the availability of room or data. These seven primitives allow the implementation of different communication schemes, as was illustrated in Section 2.

Since we have a clear separation of the application models and the architecture models, and are using trace-driven simulation, we can easily insert a layer in between the application model and the architecture model that transforms the traces generated by the application model into traces that are accepted by the architecture model. This transformation takes place at the dashed line in the middle of Figure 5. For more details on SPADE we refer the reader to [9].

4. TRACE TRANSFORMATION

In this section we explain the concept of *trace transformations*. These transformations take as input a trace generated by an application process. Such trace consists of the three application level operations. As output a trace is generated that can be accepted by an architecture model and which contains the architecture level operations. So, a trace transformation provides the mapping of application level communication primitives onto architecture level communication primitives. In SPADE this transformation needs to be done at runtime because of the co-simulation technique employed.

Both the input trace and the output trace are linear, i.e., the trace operations are totally ordered. For the input trace this is due to the sequentiality of each application process. For the output trace this linearity is a property we enforce because the architecture models we are using can accept only linear traces.

Transformation of individual operations in traces is straightforward. In the remainder we denote a transformation using '$\stackrel{\Theta}{\Longrightarrow}$'.

$$\mathsf{R} \stackrel{\Theta}{\Longrightarrow} \mathit{cd} \rightarrow \mathit{ld} \rightarrow \mathit{sr} \quad \text{(TR 1)}$$

$$\mathsf{W} \stackrel{\Theta}{\Longrightarrow} \mathit{cr} \rightarrow \mathit{st} \rightarrow \mathit{sd} \quad \text{(TR 2)}$$

$$\mathsf{E} \stackrel{\Theta}{\Longrightarrow} \mathit{E} \quad \text{(TR 3)}$$

So, a read operation R is substituted by a sequence of a check-data operation *cd*, a load-data operation *ld*, and a signal-room operation *sr*. Similarly, a write operation W is substituted by a sequence of a check-room operation *cr*, a store-data operation *st*, and a signal-data operation *sd*. An execute operation E is simply transformed into an architecture level execute operation *E*.

For a linear trace, i.e., a sequence of individual trace operations, we could simply apply the transformations for the individual operations.

$$\mathsf{R} \rightarrow \mathsf{E} \rightarrow \mathsf{W} \stackrel{\Theta}{\Longrightarrow} \mathit{cd} \rightarrow \mathit{ld} \rightarrow \mathit{sr} \rightarrow \mathit{E} \rightarrow \mathit{cr} \rightarrow \mathit{st} \rightarrow \mathit{sd}$$

However, as we already observed in Section 2, Figure 3, we may need to change the ordering of the operations dependent on the proposed architecture and mapping. This could for example result in the following transformation.

$$\mathsf{R} \rightarrow \mathsf{E} \rightarrow \mathsf{W} \stackrel{\Theta'}{\Longrightarrow} \mathit{cd} \rightarrow \mathit{cr} \rightarrow \mathit{ld} \rightarrow \mathit{E} \rightarrow \mathit{st} \rightarrow \mathit{sr} \rightarrow \mathit{sd}$$

In this case, the computation operates on data in external buffers, such as the shared buffer in shared memory in Section 2. This is modeled by issuing a check-room before, and a signal-room after the execute operation. Likewise, we can come up with many more transformations, each having their own ordering of operations.

The expansion of the individual operations as given in (TR 1), (TR 2), and (TR 3) is independent of the architecture. Also, there are a number of dependencies between operations that always should be retained, independent of the architecture. Therefore, we split the trace transformation into two steps. The first step is independent of the architecture on which the trace is to be executed. In this step the individual operations are expanded and dependencies are added between the resulting architecture level operations. We call this step *trace expansion*. The result of this step is a *partially ordered* intermediate trace. Since we support only architecture models that accept linear traces, the intermediate trace is linearized in a second step. We call this step *trace linearization*. This linearization is highly dependent on the architecture on which the trace is to be executed. Future architecture models may be able to accept partially ordered traces, thereby making the linearization step no longer needed. In Sections 4.2 and 4.3 we describe the two steps. First we give some definitions.

4.1 Definitions

Definition 1. A *trace* $T = (O, \mathcal{R})$ consists of a set of trace operations $O = \{o_i\}$, with i being a unique index, and a transitively closed ordering relation $\mathcal{R} \subseteq O^2$. The trace operations are *partially ordered* by the relation $\mathcal{R}$.

A trace is thus a partially ordered set (poset). We say that operation o_i *precedes* operation o_j if and only if $(o_i, o_j) \in \mathcal{R}$. A trace with a total ordering is called a *linear trace*.

Definition 2. An *application trace* T^{APT} is a linear trace that consists of application level operations (read, write, execute).

Definition 3. An *architecture trace* T^{ART} is a linear trace that consists of architecture level operations (check-data, load-data, signal-room, check-room, store-data, signal-data, execute).

We usually only consider the *base* of a trace.

Definition 4. For $x, y \in O$ of a poset $(O, \mathcal{R})$ we say that y *covers* x in $\mathcal{R}$ if and only if $(x, y) \in \mathcal{R}$ and there is no element $z \in O$ such that both (x, z) and $(z, y) \in \mathcal{R}$.

Definition 5. A *base* $\mathcal{B}(\mathcal{R})$ of an ordering relation $\mathcal{R} \subseteq O^2$ is the set of all pairs $(x, y) \in \mathcal{R}$ for which y *covers* x in $\mathcal{R}$.

Definition 6. The *base* $B(T)$ of a trace $T = (O, \mathcal{R})$ is defined as the ordered pair $(O, \mathcal{B}(\mathcal{R}))$.

The base of a trace thus only contains the direct predecessor and successor relations. We can obtain T from $B(T)$ by taking the transitive closure. It is easily shown that $\mathcal{B}(\mathcal{R})$ is unique.

Definition 7. A *trace transformation* $\Theta(T)$ is a mapping of one trace into another. A transformation may change both the set of operations O and the relation $\mathcal{R}$, or only the relation $\mathcal{R}$.

4.2 Trace Expansion

The first step in the transformation from an application trace T^{APT} into an architecture trace T^{ART} is the *expansion* Θ_e of the application operations into the more detailed architecture operations. As we already discussed, this expansion is more complex than just replacing the application operations with a sequence of architecture operations. Such a straightforward substitution would not allow for moving the check-room and signal-room operations, as was shown in Figures 3 and 4. Instead, in addition we do a transformation of the dependencies such that we get a partially ordered intermediate trace T^{IT}.

To define the expansion, we use the following two guidelines.

- We retain the ordering that is present in the application trace. This means that we take a pessimistic assumption on the dependencies among the trace operations.
- We only want to give *limited mobility* to operations, i.e., we want to make sure that each operation has a predecessor and a successor such that the number of options where to put the operation during linearization is limited.

We can use a set of transformation rules that describe the transformation. The first three rules are the simple expansions of the application operations into the architecture operations, which were given above in (TR 1), (TR 2), and (TR 3).

For the transformations of the dependencies we define a rule for each pair of operation types. These dependencies and their transformation are indicated in the following rules by the bold printed arrows.

$$\mathsf{E}_i \rightarrow \mathsf{E}_j \quad \overset{\Theta_e}{\Longrightarrow} \quad E_i \rightarrow E_j \tag{TR 4}$$

$$\mathsf{E} \rightarrow \mathsf{R} \quad \overset{\Theta_e}{\Longrightarrow} \quad E \rightarrow cd \rightarrow ld \rightarrow sr \tag{TR 5}$$

$$E \rightarrow \mathsf{W} \quad \overset{\Theta_e}{\Longrightarrow} \quad \begin{array}{l} E \rightarrow st \rightarrow sd \\ \quad\quad\; cr \nearrow \end{array} \tag{TR 6}$$

Here we see that the check-room operation has no dependency with the preceding execute operation. This means that in the linearization we can put the check-room either before or after this execute. The same holds for the signal-room operation in the next rule.

$$\mathsf{R} \rightarrow E \quad \overset{\Theta_e}{\Longrightarrow} \quad \begin{array}{l} cd \rightarrow ld \rightarrow E \\ \quad\quad\quad\;\; sr \searrow \end{array} \tag{TR 7}$$

$$\mathsf{R}_i \rightarrow \mathsf{R}_j \quad \overset{\Theta_e}{\Longrightarrow} \quad \begin{array}{l} cd_i \rightarrow cd_j \searrow \\ \quad\quad ld_i \rightarrow ld_j \searrow \\ \quad\quad\quad\quad sr_i \rightarrow sr_j \end{array} \tag{TR 8}$$

The strict ordering of the operations in (TR 8) is due to the assumption we made that we retain all dependencies. So even if R_i and R_j are reads from different ports which could have been performed in parallel, we do not exploit this possible parallelism.

$$\mathsf{R} \rightarrow \mathsf{W} \quad \overset{\Theta_e}{\Longrightarrow} \quad \begin{array}{l} cd \rightarrow ld \rightarrow st \rightarrow sd \\ \quad\quad\;\; cr \nearrow \quad sr \nearrow \end{array} \tag{TR 9}$$

In (TR 9) we have added the dependencies (cd, cr) and (sr, sd). These dependencies limit the mobility of the check-room and signal-room operations; without these dependencies we could move all check-room operations to the beginning of the trace, and all signal-room operations to the end of the trace.

$$\mathsf{W} \rightarrow \mathsf{E} \quad \overset{\Theta_e}{\Longrightarrow} \quad cr \rightarrow st \rightarrow sd \rightarrow E \tag{TR 10}$$

$$\mathsf{W} \rightarrow \mathsf{R} \quad \overset{\Theta_e}{\Longrightarrow} \quad cr \rightarrow st \rightarrow sd \rightarrow cd \rightarrow ld \rightarrow sr \tag{TR 11}$$

$$\mathsf{W}_i \rightarrow \mathsf{W}_j \quad \overset{\Theta_e}{\Longrightarrow} \quad \begin{array}{l} \quad\quad\quad\quad sd_i \rightarrow sd_j \\ \quad\quad\quad st_i \nearrow \rightarrow st_j \nearrow \\ cr_i \nearrow \rightarrow cr_j \nearrow \end{array} \tag{TR 12}$$

These twelve rules can be applied to a base of an application trace. However, that would not satisfy our second guideline of limited mobility. For example, if we have a base of a trace

$$\mathsf{R} \rightarrow \mathsf{E} \rightarrow \mathsf{W} \tag{1}$$

and we use the expansion rules above, we get

$$\begin{array}{l} cd \rightarrow ld \rightarrow E \rightarrow st \rightarrow sd \\ \quad\quad\;\; cr \nearrow \quad sr \nearrow \end{array} \quad .$$

Unlike the result of (TR 9), the check-room and signal-room operations are now no longer limited in their mobility. What we want to get is

$$\begin{array}{l} cd \rightarrow ld \rightarrow E \rightarrow st \rightarrow sd \\ \quad\quad\;\; cr \nearrow \quad sr \nearrow \end{array} \quad . \tag{2}$$

The extra constraints (cd, cr) and (sr, sd) are exactly what rule (TR 9) gives us if we would transform the dependency (R, W). This dependency is present in the application trace, but not in the base of this trace. Therefore, we introduce the *extended base* of a trace.

Definition 8. For $x, y \in O$ of a trace $T = (O, \mathcal{R})$ we say that y *extendedly covers* x if and only if $(x, y) \in \mathcal{R}$ and either

- there is no element $z \in O$ such that both (x, z) *and* $(z, y) \in \mathcal{R}$; or
- x is a read operation or y is a write operation, or both, and there is a sequence $(q_1, q_2, \ldots, q_n)$, $q_1 = x$, $q_n = y$, $(q_k, q_{k+1}) \in \mathcal{R}$ for all $1 \leq k < n$, such that operations $q_i, 1 < i < n$ are all execute operations and q_{k+1} covers q_k.

Definition 9. The *extended base* $B_e(T)$ of a trace $T = (O, \mathcal{R})$ is defined as the pair $(O, \mathcal{B}_e(\mathcal{R}))$. Here, $\mathcal{B}_e(\mathcal{R})$ is the set of all ordered pairs $(x, y) \in \mathcal{R}$ for which y *extendedly covers* x.

For the trace expansion transformation we now apply rules (TR 1) through (TR 12) on the extended base of an application trace. For example, the extended base of the trace of which the base was given in (1) is

$$\mathsf{R} \nearrow \rightarrow \mathsf{E} \searrow \rightarrow \mathsf{W} \quad .$$

The expansion of this extended base using the twelve transformation rules was already given in (2).

4.3 Trace Linearization

The second step in the two step transformation is *trace linearization*. This step takes the partially ordered intermediate trace T^{IT} as an input, and linearizes the ordering of the operations to obtain an architecture trace T^{ART}.

Unlike trace expansion, trace linearization is highly dependent on the architecture. This is the step where we can distinguish between different implementations at the architecture level of the application level communication primitives.

So, how are we going to use these architecture characteristics to find a linearization of the intermediate trace that matches the architecture? From the transformation rules we can see that the only freedom is in the placement of the signal-room and check-room operations. However, although we enforced limited mobility of these operations, the number of options to choose from can be vast for a long trace. Consider for example the following intermediate trace.

$$\begin{array}{l} \quad\quad\quad\quad st_i \rightarrow sd_i \rightarrow E \rightarrow st_j \rightarrow sd_j \\ cr_i \nearrow \longrightarrow\longrightarrow\longrightarrow\longrightarrow cr_j \nearrow \end{array}$$

We can obtain four different architecture traces from this intermediate trace by linearization.

$$cr_i \rightarrow st_i \rightarrow sd_i \rightarrow E \rightarrow \mathbf{cr}_j \rightarrow st_j \rightarrow sd_j \tag{3a}$$

$$cr_i \rightarrow st_i \rightarrow sd_i \rightarrow \mathbf{cr}_j \rightarrow E \rightarrow st_j \rightarrow sd_j \tag{3b}$$

$$cr_i \rightarrow st_i \rightarrow \mathbf{cr}_j \rightarrow sd_i \rightarrow E \rightarrow st_j \rightarrow sd_j \tag{3c}$$

$$cr_i \rightarrow \mathbf{cr}_j \rightarrow st_i \rightarrow sd_i \rightarrow E \rightarrow st_j \rightarrow sd_j \tag{3d}$$

Each of these linearizations could be a valid representation of the behavior at the architecture level. Given an architecture, a system architect should be able to specify any of these linearizations. Ideally, these specifications should have some clear relation to the architecture, such that they can be easily derived from it, either automatically or manually by the system architect. For now the specifications are derived by hand, using constraints such as 'as soon as possible but after operation X' or 'as late as possible but before operation Y'.

5. EXAMPLE

In this section we demonstrate the use of trace transformations. We show the effect of having or not having local memory inside a processor, and how the trace transformation technique easily deals with these different architectures.

Consider a part of an application as shown in Figure 6. Process A

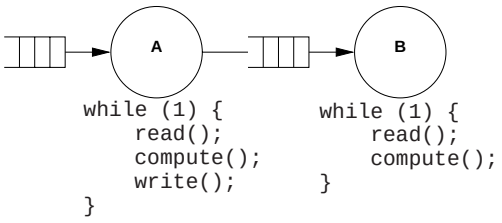


Figure 6: Part of application model.

reads data from its input FIFO, does some computation, and writes data to its output FIFO. Process B reads data from this FIFO and does some computation. The application traces of these processes look as follows.

$$\mathsf{R} \to \mathsf{E} \to \mathsf{W} \to \mathsf{R} \to \mathsf{E} \to \mathsf{W} \to \ldots \qquad (T_A^{\text{APT}})$$

$$\mathsf{R} \to \mathsf{E} \to \mathsf{R} \to \mathsf{E} \to \ldots \qquad (T_B^{\text{APT}})$$

After the first step of the transformation, the trace expansion, we get the following intermediate traces.

$$cd \to ld \to E \to st \to sd \to cd \to ld \to E \to st \to sd \to \ldots \quad (T_A^{\text{IT}})$$
$$cr \quad sr \qquad\qquad cr \quad sr$$

$$cd \to ld \to E \to cd \to ld \to E \to \ldots \quad (T_B^{\text{IT}})$$
$$sr \qquad\qquad\qquad sr \to \ldots$$

Now suppose we have an architecture as depicted in Figure 7. Processor X reads its input data from a shared buffer in shared

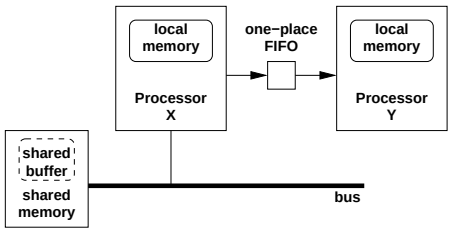


Figure 7: Architecture for application of Figure 6.

memory. The communication between processor X and processor Y is mapped onto a one-place FIFO buffer. Both processors have a local memory. Both can copy their input data to this local memory before starting a computation. Processor X can also store the results of a computation there before copying it to the FIFO buffer. This means that we get the following specification for the linearization.

- Processor X:
  - signal-room *sr* as soon as possible
  - check-room *cr* as late as possible
- Processor Y:
  - signal-room *sr* as soon as possible

This results in the following architecture traces.

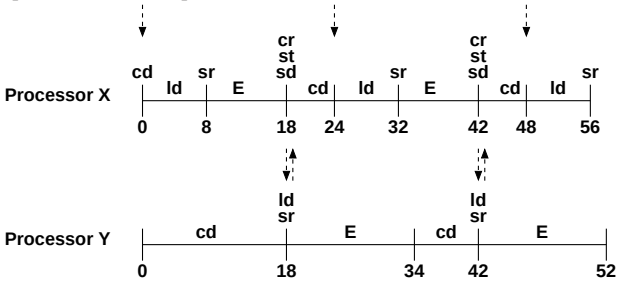
$$cd \to ld \to sr \to E \to cr \to st \to sd \to$$
$$cd \to ld \to sr \to E \to cr \to st \to sd \to \ldots \quad (T_A^{\text{ART}})$$

$$cd \to ld \to sr \to E \to cd \to ld \to sr \to E \to \ldots \quad (T_B^{\text{ART}})$$

The total transformation is thus no more than a straightforward substitution of the read R and write W by their expansions.

When we look at the timing behavior of this implementation we get the following timelines. They are based on the following assumptions.

- The input data for processor X is available at cycles 0, 24, 48, etc.
- The execution time of the computation of processor X is 10 cycles; the execution time of the computation of processor Y is 16 cycles.
- The transfer of a single token over the bus takes 8 cycles.
- The copying of data between the local memory and the one-place FIFO takes place in zero time.



These timelines should be read as follows. At cycle 0 processor X checks for data in its input buffer in shared memory. Then it starts a load-data to fetch this data into its local memory; this takes 8 cycles for the bus transfer. Immediately a signal-room is given, and then the computation is started which takes 10 cycles. At cycle 18 a check-room on the one-place FIFO is done, data is copied into this FIFO, and a signal-data is given. Then again a check-data is started, which now stalls the process until new input data is available in cycle 24. The dashed arrows at the top indicate the arrival of input data for processor X; the dashed arrows in between the two timelines indicate the signal-data and signal-room operations on the one place FIFO.

We conclude from these timelines that the throughput of the system is limited by the rate at which new data for processor X is available; processor X has to wait upon each check-data before it can continue.

Now consider an architecture in which both processors do not have a local memory. This means that processor X directly operates on the data in shared memory, and stores its results in the one-place FIFO, and that processor Y operates on the data in the one-place FIFO. The linearization for this architecture is specified as follows.

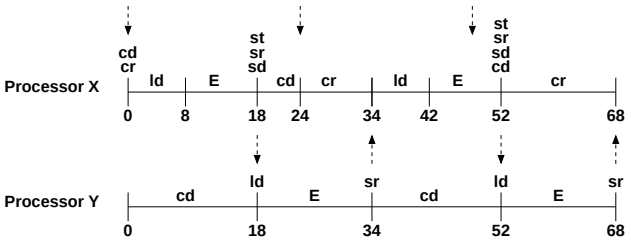
- Processor X:
  - signal-room *sr* as late as possible
  - check-room *cr* as soon as possible
- Processor Y:
  - signal-room sr_i as late as possible, but before the next check-data cd_{i+1}

This results in the following architecture traces.

$$cd \to cr \to ld \to E \to st \to sr \to sd \to$$
$$cd \to cr \to ld \to E \to st \to sr \to sd \to \ldots \quad (T_A^{\text{ART}\prime})$$

$$cd \to ld \to E \to sr \to cd \to ld \to E \to sr \to \ldots \quad (T_B^{\text{ART}\prime})$$

Again, we can construct timelines for both processors.



We observe that the throughput is no longer limited by the rate at which the input data arrives. Because both processors need access to the one-place FIFO during their computations, these are now fully sequentialized. Processor X has to wait for the computation of processor Y before it can continue with its next computation. As a result, throughput has dropped from one computation every 24 cycles to one computation every 34 cycles.

6. RELATED WORK

In concurrency theory and process algebras the notions of traces and partial orders are well known. For example, in [12] a model is used for modeling concurrency that is closely related to our trace model using partial orders. The *labeled partial order* defined in [12] matches Definition 1. In [13] is discussed how *action refinement* can be defined on event structures, which is another model resembling our traces. Action refinement is closely related to trace expansion; main difference is that they allow only straightforward substitution of a single action or operation by a (partially ordered) set of actions. This would limit the resulting trace to orderings as in (3a), whereas our trace transformation can also result in orderings such as in (3b), (3c), and (3d). A more powerful notion of action refinement is introduced in [5] which is more like our trace transformation, in the sense that in addition to the substitution of a single action there is also a transformation of the dependencies.

In the ESPRIT/OMI COSY project applications are specified using the same model as in SPADE. They have also defined a refinement of the application level communication primitives read and write into more detailed implementation primitives [2]. However, this refinement does not allow for reordering of operations. This means that they can only implement a behavior in which data is copied from the private state space of a producer into a shared buffer, and then from this shared buffer into the private state space of a consumer.

The C-Heap framework [11] provides low-level communication primitives to an application programmer. These primitives are very much like the architecture level primitives we defined. A transformation from more abstract primitives such as read and write is thus not needed. However, the application models are more tightly coupled to a specific architecture and implementation. This reduces reuse of functional IP and puts an extra burden on the application programmer.

The handshake expansion and reshuffling used in the design of asynchronous circuits [10][3] have resemblance with our transformation technique, but are based on static analysis whereas we use a runtime transformation technique. In addition, we try to fit the transformation to a proposed architecture, whereas they try to optimize the transformation and synthesize the architecture from the obtained behavior.

7. CONCLUSIONS AND FUTURE WORK

We presented a new trace transformation technique for communication refinement in architecture exploration. This technique supports the system architect in mapping application level communication primitives to architecture level primitives. We illustrated how the technique can perform different mappings for different architectures. Our examples demonstrate that an architecture exploration tool that supports this technique can be used to evaluate alternative architectures starting from abstract application models.

We are currently extending the presented techniques to also support grain refinements in the mapping step. This will support system architects in exploring architectures in which synchronization and data transfer are performed at finer grains than the communication specified in the application model, or in which synchronization is performed at a different grain than the data transfers. We plan to further automate the communication refinement by automatically inferring the refinement directives from the definition of the architecture under consideration.

8. REFERENCES

[1] 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.

[2] 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 (CODES'99)*, pages 142–146, Rome, Italy, May 3–5 1999.

[3] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Automatic handshake expansion and reshuffling using concurrency reduction. In *Proc. 19th Int. Conf. on Application and Theory of Petri Nets*, pages 86–110, Lisbon, Portugal, June 1998.

[4] E. de Kock, G. Essink, W. Smits, P. van der Wolf, J.-Y. Brunel, W. Kruijtzer, P. Lieverse, and K. Vissers. YAPI: Application modeling for signal processing systems. In *Proc. 37th Design Automation Conference (DAC'2000)*, pages 402–405, Los Angeles, CA, June 5–9 2000.

[5] W. Janssen, M. Poel, and J. Zwiers. Action systems and action refinement in the development of parallel systems; an algebraic approach. In J. Baeten and J. Groote, editors, *Proc. 2nd Int. Conf. on Concurrency Theory (CONCUR'91)*, LNCS 527, pages 298–316, Amsterdam, The Netherlands, Aug. 1991.

[6] G. Kahn. The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 74*. North-Holland Publishing Co., 1974.

[7] B. Kienhuis, E. Deprettere, K. Vissers, and P. van der Wolf. An approach for quantitative analysis of application-specific dataflow architectures. In *Proc. IEEE Int. Conf. on Application-Specific Systems, Architectures, and Processors (ASAP'97)*, July 14–16 1997.

[8] E. Lee and T. Parks. Dataflow process networks. *Proc. of the IEEE*, 83(5):773–801, May 1995.

[9] P. Lieverse, P. van der Wolf, E. Deprettere, and K. Vissers. A methodology for architecture exploration of heterogeneous signal processing systems. In *Proc. 1999 IEEE Workshop on Signal Processing Systems (SiPS'99)*, pages 181–190, Taipei, Taiwan, Oct. 20–22 1999.

[10] A. J. Martin. Synthesis of asynchronous VLSI circuits. In J. Staunstrup, editor, *Formal Methods for VLSI Design; IFIP WG 10.5 Lecture Notes*, chapter 6, pages 237–283. North-Holland, 1990.

[11] A. Nieuwland and P. Lippens. A heterogeneous HW-SW architecture for hand-held multi-media terminals. In *Proc. 1998 IEEE Workshop on Signal Processing Systems (SiPS'98)*, pages 113–122, Cambridge, MA, Oct. 8–10 1998.

[12] V. Pratt. Modeling concurrency with partial orders. *Int. Journal of Parallel Programming*, 15(1):33–71, Feb. 1986.

[13] R. van Glabbeek and U. Goltz. Equivalences and refinement. In I. Guessarian, editor, *Semantics of Systems of Concurrent Processes, Proc. LITP Spring School on Theoretical Computer Science*, LNCS 469, pages 309–333, La Roche Posay, France, Apr. 23–27 1990. Springer.