Improved Derivation of Process Networks

Sven Verdoolaege Hristo Nikolov Nikolov Todor Plamenov Stefanov
Leiden Institute of Advanced Computer Science, Leiden, The Netherlands
{sverdool,nikolov,stefanov}@liacs.nl

Abstract— **Current emerging embedded System-on-Chip platforms are increasingly becoming multiprocessor architectures. System designers experience significant difficulties in programming these platforms. The applications are typically specified as sequential programs that do not reveal the available parallelism in an application, thereby hindering the efficient mapping of an application onto a parallel multiprocessor platform. In this paper we present our compiler techniques that facilitate the migration from a sequential application specification to a *parallel* application specification using the Process Network model of computation. Our work is inspired by a previous research project called Compaan. With our techniques we address optimization issues such as the generation of Process Networks with simplified topology and communication without sacrificing the Process Networks performance. Moreover, we describe a technique for compile-time memory requirement estimation which we consider as an important contribution of this paper. We demonstrate the usefulness of our techniques on several examples.**

I. Introduction and Motivation

The complexity of embedded multimedia and signal processing applications has reached a point where the performance requirements of these applications can no longer be supported by embedded system platforms based on a single processor. Therefore, modern embedded Systems-on-Chip platforms have to be multiprocessor architectures. In the recent years a lot of attention has been paid to building such multiprocessor platforms. Fortunately, advances in chip technology facilitate this activity. However, less attention has been paid to compiler techniques for efficient programming of multiprocessor platforms, i.e., the efficient mapping of applications onto these platforms is becoming a key issue. Today, system designers experience significant difficulties in programming multiprocessor platforms because the way an application is specified by an application developer does not match the way multiprocessor platforms operate. The applications are typically specified as sequential programs using imperative programming languages such as C/C++ or Matlab. Specifying an application as a sequential program is relatively easy and convenient for application developers, but the sequential nature of such specification does not reveal the available parallelism in an application. This fact makes the efficient mapping of an application onto a parallel multiprocessor platform very difficult. By contrast, if an application is specified using a parallel model of computation (MoC) then the mapping can be done in a systematic and transparent way using a disciplined approach [17], but specifying an application using a parallel MoC is difficult, not well understood by application developers, and a time consuming and error prone process. That is why application developers still prefer to specify an application as a sequential program, which is well understood, even though such a specification is not suitable for mapping an application onto a parallel multiprocessor platform.

This gap between between a sequential program and a parallel model of computation motivates us to research and develop compiler techniques that facilitate the migration from a sequential application specification to a parallel application specification. These compiler techniques depend on the parallel model of computation used for parallel application specification. Although many parallel models of computation exist [14], [15], in this paper we consider the Process Network model of computation [12] because its operational semantics are simple, yet general enough, to conveniently specify *stream-oriented* data processing that fits nicely with the application domain we are interested in—multimedia and signal processing applications. Moreover, for this application domain, many researchers [6]–[8], [11], [16], [18], [19], [21], [24] have already indicated that Process Networks are very suitable for systematic and efficient mapping onto multiprocessor platforms.

In this paper we present our compiler techniques for deriving Process Network specifications for applications specified as static affine nested loop programs (SANLPs), thereby bridging the gap mentioned above in a particular way. SANLPs are important in Scientific, Matrix Computation and Multimedia and Adaptive Signal Processing applications. Our work is inspired by previous research on Compaan [13], [20], [23]. The techniques presented in this paper can be seen as a significant improvement of the techniques developed in the Compaan project in the following sense. The Compaan project has identified the fundamental problems that have to be solved in order to derive Process Networks systematically and automatically and has proposed and implemented basic solutions to these problems. However, many optimization issues that improve the quality of the derived Process Networks have not been fully addressed in Compaan. The techniques presented in this paper try to address optimization issues in four main aspects:

Given an application specified as a SANLP,

1) *Derive (if possible) Process Networks (PN) with fewer communication channels between different processes compared to Compaan derived PNs without sacrificing the PN performance.*
2) *Derive (if possible) Process Networks (PN) with fewer processes compared to Compaan derived PNs without sacrificing the PN performance.*
3) *Replace (if possible) reordering communication channels with simple FIFO channels without sacrificing the*

PN performance.

4) *Determine the size of the communication FIFO channels at compile time*. The problem of deriving efficient FIFO sizes has not been addressed by Compaan. Our techniques for computing FIFO sizes constitute a starting point to overcome this problem.

The rest of this paper is organized as follows. In Section II, we first introduce some concepts that we will need throughout this paper. We explain how to derive and optimize Process Networks in Section III and how to compute FIFO sizes in Section IV. Detailed examples are given in Section V, with a further comparison to Compaan-generated networks in Section VI. After a comparison to other related work in Section VII, we conclude in Section VIII.

II. Preliminaries

In this section, we introduce the process network model and parametric integer programming, our main analysis tool.

A. The Process Network Model

As the names suggests, a process network consists of a set of *processes*, also called *nodes*, that communicate with each other through *channels*. Each process has a fixed internal schedule, but there is no (a priori) global schedule that dictates the relative order of execution of the different processes. Rather, the relative execution order is solely determined by the channels through which the processes communicate. In the special case of a Kahn Process Network (KPN), the communication channels are unbounded FIFOs that support a blocking read. That is, a process that reads data from a channel will block until data is available and this is the only way in which a process influences the execution of another process.

In practice, FIFOs are not unbounded and so our process network model also supports blocking writes. It is important then to ensure the FIFOs are large enough to avoid deadlocks. Note that determining suitable channel sizes may not be possible in general, but it is possible for process networks derived from SANLPs as defined in Section III-A. We also allow data to be written to a channel in an order that is different from the order in which the data is read. Such channels are called *reordering channels* and could be implemented using a piece of addressable memory. Note that allowing reordering channels does not extend the expressive power of the model, since the process reading data from the channel could equally well be changed to read the data in the order in which it is sent and to store it in an internal memory block.

B. Parametric Integer Programming

We will be dealing in this paper with sets of vectors of integers defined by linear inequalities, $S = \{\, \mathbf{i} \in \mathbb{Z}^n \mid A\mathbf{i} + \mathbf{c} \geq \mathbf{0} \,\}$, with $A \in \mathbb{Z}^{m \times n}$ and $\mathbf{c} \in \mathbb{Z}^m$. We will call such sets simply *integer sets*. The elements of the vectors in these sets could, for instance, refer to the iterators of a loop nest from a sequential program, with the linear inequalities corresponding to the lower and upper bounds of the loops. We will further assign an ordering to these integer vectors that corresponds to the order in which the iterations of the loop nest are executed. This order is called the *lexicographical order* and will be denoted by $\prec$. A vector $\mathbf{a} \in \mathbb{Z}^n$ is said to be lexicographically (strictly) smaller than $\mathbf{b} \in \mathbb{Z}^n$ if for the first position i in which $\mathbf{a}$ and $\mathbf{b}$ differ, we have $a_i < b_i$, or, equivalently,

$$\mathbf{a} \prec \mathbf{b} \equiv \bigvee_{i=1}^{n} \left(a_i < b_i \wedge \bigwedge_{j=1}^{i-1} a_j = b_j \right). \tag{1}$$

In some integer sets, the variables appear in two (or more) groups, say $\mathbf{i} \in \mathbb{Z}^{n_1}$ and $\mathbf{j} \in \mathbb{Z}^{n_2}$. Such sets can be seen as subsets of the Cartesian product $\mathbb{Z}^{n_1} \times \mathbb{Z}^{n_2}$ and will be called *integer relations*. The constraints defining these sets or relations may also involve additional, *existentially quantified* variables $\boldsymbol{\alpha} \in \mathbb{Z}^{n'}$ as well as some *parameters* $\mathbf{p} \in \mathbb{Z}^{n''}$, i.e., $S = \{\, \mathbf{i} \in \mathbb{Z}^n \mid \exists \boldsymbol{\alpha} \in \mathbb{Z}^{n'} : A\mathbf{i} + B\boldsymbol{\alpha} + C\mathbf{p} + \mathbf{c} \geq \mathbf{0} \,\}$, with $A \in \mathbb{Z}^{m \times n}$, $B \in \mathbb{Z}^{m \times n'}$, $C \in \mathbb{Z}^{m \times n''}$ and $\mathbf{c} \in \mathbb{Z}^m$.

Parametric integer programming [9] is a technique for computing the lexicographically smallest element of a parametric integer set. The result is a subdivision of the parameter space with for each cell of this subdivision a description of the corresponding unique minimal element as an affine combination of the parameters. This result can be described as a union of parametric integer sets, where each set in the union contains a single point. Parametric integer programming (PIP) can be used to project out some of the variables in a set. We simply compute the lexicographical minimum of these variables, treating all other variables as additional parameters, and then discard the description of the minimal element.

III. Derivation of Process Networks

This section explains the conversion of static affine nested loop programs (SANLPs) to Process Networks. We first discuss SANLPs and our internal representation in Section III-A. Then we recall how to perform dataflow analysis in Section III-B and how to determine channels types in Section III-C. In Section III-D we show how to extend the dataflow analysis to detect reuse, reducing the number of channels between nodes and in Section III-E we show we can enhance this effect by removing (artificial) copy nodes.

A. Limitations on the Input and Internal Representation

We impose the usual restrictions on the input programs we process, i.e., the SANLPs. A SANLP consists of a set of statements, each possibly enclosed in loops and/or guarded by conditions. The loops need not be perfectly nested. All lower and upper bounds of the loops as well as all expressions in conditions and array accesses are (quasi-)affine combinations of enclosing loop iterators and parameters. The values of the parameters may not change during the execution of the program, or at least the part we analyze. The reason for these restrictions is that they allow us to represent all relevant data using integer sets and relations, as defined in Section II-B.

In particular, the set of iterator vectors for which a statement is executed is an integer set called the *iteration domain*. These

iteration domains will form the basis of the description of the nodes in our process network, as each node will correspond to a particular statement. The channels are determined by the array (or scalar) accesses in the corresponding statements. All accesses that appear on the left hand side of an assignment or in an address-of (&) expression are considered to be *write accesses*. All other accesses are considered to be *read accesses*. Each of these accesses is represented by an *access relation*, relating each iteration of the statement to the array element accessed by the iteration, i.e., $\{ (\mathbf{i}, \mathbf{a}) \in I \times A \mid \mathbf{a} = L\mathbf{i} + \mathbf{m} \}$, where I is the iteration domain, A is the array space and $L\mathbf{i}+\mathbf{m}$ is the affine access function. The next section describes how the channels are derived from these access relations.

B. Dataflow Analysis

To compute the channels between the nodes, we basically need to perform dataflow analysis [10]. That is, for each execution of a read operation of a given data element in the sequential program, we need to find the corresponding write operation that wrote the data element. In the simplest case, where a given array is written by a single statement in a loop nest that furthermore precedes the loop nest reading from the array, we need to find the last iteration of the write statement that writes to the array element read by a given iteration of the read statement. This can be formulated as a single PIP problem. In particular, we need to solve

$$\operatorname{lexmax} \{ \mathbf{i}_{\mathrm{w}} \in I_{\mathrm{w}} \mid W(\mathbf{i}_{\mathrm{w}}, \mathbf{a}) \wedge R(\mathbf{i}_{\mathrm{r}}, \mathbf{a}) \wedge \mathbf{i}_{\mathrm{r}} \in I_{\mathrm{r}} \},$$

where W and R are the write and read access relations and $I_{\mathrm{w}} \subset \mathbb{Z}^{n_1}$ and $I_{\mathrm{r}} \subset \mathbb{Z}^{n_2}$ are the iteration domains of the write and read operation respectively. Note that the iterators of the read operation are treated as parameters in this problem, along with any possible structural parameters. The solution is therefore a map from the read operation to the corresponding write operation. The array index $\mathbf{a}$ is also treated as a vector of parameters in the above formulation, but it can be ignored in the solution since it is uniquely identified by the read iteration vector.

If the read and write operation share one or more enclosing loops, then we need to ensure that we only consider write operations that are executed before the read operation. However, the lexicographical order that we need to impose is not a linear constraint, but rather a disjunction of n linear constraints (1), where n is the shared nesting level. We therefore need to solve $n + 1$ PIP problems (the write and read operation may also occur in the same iteration of the loop nest). The presence of multiple statements that write to the same array complicates the analysis only slightly, since we now also have to impose lexicographical orderings between the different write operations to obtain the last of them. Our implementation of this analysis is a variation of the algorithm outlined in [10].

In the end, we obtain pairs of write and read operations such that some data flows from the write operation to the read operation. These pairs correspond to the channels in our process network. For each of these pairs, we further obtain a union of integer relations

$$\bigcup_{j=1}^{m} D_j(\mathbf{i}_{\mathrm{w}}, \mathbf{i}_{\mathrm{r}}) \subset \mathbb{Z}^{n_1} \times \mathbb{Z}^{n_2} \tag{2}$$

that connect the specific iterations of the write and read operations such that each iteration of a given read operation is uniquely paired off to some write operation iteration.

C. Determining Channel Types

In general, the channels we derived in the previous section may not be FIFOs. That is, data may be written to the channel in an order that is different from the order in which data is read. We therefore need to check whether such reordering occurs. This check can again be formulated as a (set of) PIP problem(s). Reordering occurs iff there exist two pairs of write and read iterations, $(\mathbf{w}_1, \mathbf{r}_1)$ and $(\mathbf{w}_2, \mathbf{r}_2)$, such that the order of the write operations is different from the order of the read operations, i.e., $\mathbf{w}_1 \succ \mathbf{w}_2$ and $\mathbf{r}_1 \prec \mathbf{r}_2$, or equivalently

$$\mathbf{w}_1 - \mathbf{w}_2 \succ \mathbf{0} \qquad \text{and} \qquad \mathbf{r}_1 \prec \mathbf{r}_2. \tag{3}$$

Given a union of integer relations describing the channel (2), then for any pair of relations in this union, (D_{j_1}, D_{j_2}), we therefore need to solve n_2 PIP problems

$$\begin{aligned} \operatorname{lexmax} \{ & (\mathbf{w}_1 - \mathbf{w}_2, (\mathbf{w}_1, \mathbf{r}_1), (\mathbf{w}_2, \mathbf{r}_2), \mathbf{p}) \mid \\ & (\mathbf{w}_1, \mathbf{r}_1) \in D_{j_1} \wedge (\mathbf{w}_2, \mathbf{r}_2) \in D_{j_2} \wedge \mathbf{r}_1 \prec \mathbf{r}_2 \}, \end{aligned} \tag{4}$$

where $\mathbf{r}_1 \prec \mathbf{r}_2$ should be expanded according to Equation (1) to obtain the n_2 problems. If any of these problems has a solution and if it is lexicographically positive or unbounded (in the first n_1 positions), then reordering occurs. Note that we do not compute the maximum of $\mathbf{w}_1 - \mathbf{w}_2$ in terms of the parameters $\mathbf{p}$, but rather the maximum over all values of the parameters. If reordering occurs for any value of the parameters then we simply consider the channel to be reordering. Equation (4) therefore actually represents a non-parametric integer programming problem. The large majority of these problems will be trivially unsatisfiable.

The reordering test of this section is a variation of the reordering test of [23], where it is formulated as $n_1 \times n_2$ PIP problems for a channel described by a single integer relation. The simplified computation for specific types of relations of [22] apply to pairs of the same relation and, with some modifications, also to pairs of different relations.

D. Detecting Self Reuse

The reordering check from the previous section is not sufficient to determine whether a channel is a FIFO or not. Even if the data elements are read in the same order as they are written, they may be read multiple times. Rather than detecting and handling this multiplicity, we opt to remove the multiplicity entirely by replacing the channel by two FIFOs, without any special cases. The added benefit of removing multiplicity is that some channels which would be classified as reordering, can also be decomposed into two FIFOs.

A typical example of a situation where the removal of multiplicity is very beneficial is the outer product of two

```
for (i = 0; i < N; ++i)
    a[i] = A(i);
for (j = 0; j < N; ++j)
    b[j] = B(j);
for (i = 0; i < N; ++i)
    for (j = 0; j < N; ++j)
        c[i][j] = a[i] * b[j];
```

Fig. 1. Outer product source code.

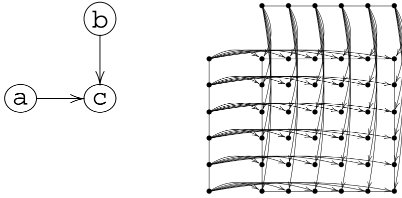

Fig. 2. Outer product dependence graph with multiplicity.

vectors, shown in Figure 1. Figure 2 shows the result of a straightforward application of the dataflow analysis from Section III-B. The left part of the figure shows the three nodes and two channels; the right part shows the data flow between the individual iterations of the nodes. The iterations are executed top-down, left-to-right. The channel between a and c is described by the relation

$$D_{\mathtt{a}\to\mathtt{c}} = \{\, (i_{\mathtt{a}}, i_{\mathtt{c}}, j_{\mathtt{c}}) \mid 0 \le i_{\mathtt{c}} \le N-1 \wedge 0 \le j_{\mathtt{c}} \le N-1 \wedge i_{\mathtt{a}} = i_{\mathtt{c}} \,\} \tag{5}$$

and would be classified as non-reordering, since the data elements are read (albeit multiple times) in the order in which they are produced. The channel between b and c, on the other hand, is described by the relation

$$D_{\mathtt{b}\to\mathtt{c}} = \{\, (j_{\mathtt{b}}, i_{\mathtt{c}}, j_{\mathtt{c}}) \mid 0 \le i_{\mathtt{c}} \le N-1 \wedge 0 \le j_{\mathtt{c}} \le N-1 \wedge j_{\mathtt{b}} = j_{\mathtt{c}} \,\} \tag{6}$$

and would be classified as reordering, with the further complication that the same data element needs to be sent over the channel multiple times. By simply letting node c only read a data element from these channels the first time it needs the data and from a newly introduced self-loop channel all other times, we obtain the network shown in Figure 3. In this network, all channels, including the new self-loop channels, are FIFOs.

To obtain this decomposition, we simply perform a dataflow analysis on each read access. That is, for each iteration of a read access, we determine the previous iteration of the same read access reading the same data element. This relation will

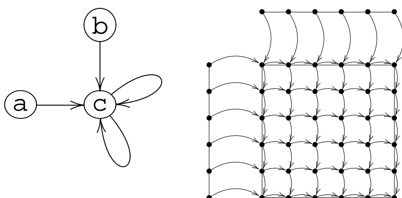

Fig. 3. Outer product dependence graph without multiplicity.

```
    for (i = 0; i < N; ++i)
        b[i] = f(i > 0 ? a[i-1] : a[i], a[i],
3                i < N-1 ? a[i+1] : a[i]);
    for (i = 0; i < N; ++i) {
        if (i > 0)
            tmp = b[i-1];
        else
8           tmp = b[i];
        c[i] = g(tmp, b[i]);
    }
```

Fig. 4. Use of temporary variables to express border behavior.

determine the new channel. Iterations for which there is no corresponding previous iteration continue to read from the original source. For example, the channel with dependence relation $D_{\mathtt{b}\to\mathtt{c}}$ (6) is split into a channel with relation

$$D'_{\mathtt{b}\to\mathtt{c}} = \{\, (j_{\mathtt{b}}, i_{\mathtt{c}}, j_{\mathtt{c}}) \mid i_{\mathtt{c}} = 0 \wedge 0 \le j_{\mathtt{c}} \le N-1 \wedge j_{\mathtt{b}} = j_{\mathtt{c}} \,\}$$

and a self-loop channel with relation

$$D^2_{\mathtt{c}\to\mathtt{c}} = \{\, (i'_{\mathtt{c}}, j'_{\mathtt{c}}, i_{\mathtt{c}}, j_{\mathtt{c}}) \mid 1 \le i_{\mathtt{c}} \le N-1 \wedge 0 \le j_{\mathtt{c}} \le N-1 \wedge j'_{\mathtt{c}} = j_{\mathtt{c}} \wedge i'_{\mathtt{c}} = i_{\mathtt{c}} - 1 \,\}. \tag{7}$$

Rather than performing this analysis as a separate step, we can also combine it with the regular dataflow analysis. That is, for each iteration of a read access, we determine the previous access to the same data element that is either a write *or* a read through the same read access. We can then apply the analysis from Section III-C and since there is no more multiplicity in any of the channels, any non-reordering channel will be a FIFO.

We can go one step further and not only consider reuse through the same read access, but also through other read accesses from the same statement. This analysis will not affect the types of any of the channels, but it may reduce the communication volume between different nodes. An example of such a case will be explained in detail in Section V-A. This further analysis can again be integrated in the regular dataflow analysis. Finally, channels that result from different read accesses from the same statement to data written by the same write access are combined into a single channel if this combination does not introduce reordering.

E. Simple Copy Propagation

A typical multimedia application has one or more kernels that uniformly manipulate a stream of data such as an image. At the borders of such data, however, the kernel will usually behave slightly differently. A common way to express such border behavior is to use temporary variables, possibly implicitly through the use of the ternary operator (`?:`) in C. Both an implicit and an explicit example are shown in Figure 4 on a 1D data stream.

A straightforward transformation of code such as that of Figure 4 would introduce extra nodes that simply copy the data from the appropriate channel to the input channel of the core node. Not only does this result in a network with more nodes than needed, it also reduces the opportunity for applying the self reuse detection described in Section III-D.

Our solution is to first identify the statements that simply copy data to a temporary variable and to perform a dataflow analysis on those temporary variables in a first pass. We then combine the (union of) dependence relation(s) between the copy node and the core node $D(\mathbf{i}_{\mathrm{copy}}, \mathbf{i}_{\mathrm{core}})$ with the corresponding read access relation $R_{\mathrm{copy}}(\mathbf{i}_{\mathrm{copy}}, \mathbf{a})$ of the copy node to obtain a new (union of) access relation(s) $R_{\mathrm{core}}(\mathbf{i}_{\mathrm{core}}, \mathbf{a})$ for the core node that bypasses the copy node. That is, we compute

$$R_{\mathrm{core}} = \{\, (\mathbf{i}_{\mathrm{core}}, \mathbf{a}) \mid \exists \mathbf{i}_{\mathrm{copy}} : D(\mathbf{i}_{\mathrm{copy}}, \mathbf{i}_{\mathrm{core}}) \wedge R_{\mathrm{copy}}(\mathbf{i}_{\mathrm{copy}}, \mathbf{a}) \,\} \tag{8}$$

by combining the constraints of D and R_{copy} and projecting out the iteration vector of the copy node $\mathbf{i}_{\mathrm{copy}}$. The regular dataflow analysis is then performed using these adapted access functions.

As an example, consider the copy statement in line 6 of Figure 4. Dataflow analysis yields the following relation between this statement and the statement in line 9:

$$D = \{\, (i_{\mathrm{copy}}, i_{\mathrm{core}}) \mid i_{\mathrm{copy}} = i_{\mathrm{core}} \wedge 1 \le i_{\mathrm{copy}} \le N-1 \,\}.$$

The read access in the copy statement from the `b` array is given by the relation

$$R_{\mathrm{copy}} = \{\, (i_{\mathrm{copy}}, a) \mid a = i_{\mathrm{copy}} - 1 \,\}$$

and so the propagated access function is given by the relation

$$R_{\mathrm{core}} = \{\, (i_{\mathrm{core}}, a) \mid a = i_{\mathrm{core}} - 1 \wedge 1 \le i_{\mathrm{core}} \le N-1 \,\}.$$

IV. Computing Channel Sizes

In this section, we explain how we compute the buffer sizes for the FIFOs in our networks at compile-time. This computation may not be feasible for Process Networks in general, be we are dealing here with the easier case of networks generated from static affine nested loop programs. We first consider self-loops, with two special cases in Section IV-A and Section IV-B, and the general case in Section IV-C. In Section IV-D, we then explain how to reduce the general case of FIFOs to self-loops by scheduling the network.

A. Single-Register Self-loops

We first consider a simple, but important special case of self-loop channels, namely the case where the channels holds at most one data element throughout the execution of the program. Such channels can be replaced by a single register. This situation occurs when for every pair of write and read iterations $(\mathbf{w}_2, \mathbf{r}_2)$, there is no other read iterations $\mathbf{r}_1$ reading from the same channel in between. In other words, the situation does *not* occur iff there exist two pairs of write and read iterations, $(\mathbf{w}_1, \mathbf{r}_1)$ and $(\mathbf{w}_2, \mathbf{r}_2)$, such that $\mathbf{w}_2 \prec \mathbf{r}_1 \prec \mathbf{r}_2$, or equivalently $\mathbf{r}_1 - \mathbf{w}_2 \succ \mathbf{0}$ and $\mathbf{r}_1 \prec \mathbf{r}_2$. Notice the similarity between this condition and the reordering condition (3). The PIP problems that need to be solved to determine this condition are therefore nearly identical to the problems (4), viz.,

$$\operatorname{lexmax} \{\, (\mathbf{r}_1 - \mathbf{w}_2, (\mathbf{w}_1, \mathbf{r}_1), (\mathbf{w}_2, \mathbf{r}_2), \mathbf{p}) \mid (\mathbf{w}_1, \mathbf{r}_1) \in D_{j_1} \wedge (\mathbf{w}_2, \mathbf{r}_2) \in D_{j_2} \wedge \mathbf{r}_1 \prec \mathbf{r}_2 \,\}, \tag{9}$$

where again (D_{j_1}, D_{j_2}) is a pair of relations in the union describing the channel and where $\mathbf{r}_1 \prec \mathbf{r}_2$ should be expanded according to Equation (1).

B. Uniform Self-Dependences on Rectangular Domains

Another important special case occurs when the channel is represented by a single integer relation that in turn represents a uniform dependence over a rectangular domain. A dependence is called uniform if the difference between the read and write iteration vector is a (possibly parametric) constant over the whole relation. We call such a dependence a uniform dependence over a rectangular domain if the set of iterations reading from the channel form a rectangular domain. (Note that due to the dependence being uniform, also the write iterations will form a rectangular domain in this case.) For example, the relation $D^2_{\mathtt{c}\to\mathtt{c}}$ (7) from Section III-D is a uniform dependence over a rectangular domain since the difference between the read and write iteration vector is $(i_{\mathtt{c}}, j_{\mathtt{c}}) - (i'_{\mathtt{c}}, j'_{\mathtt{c}}) = (1, 0)$ and since the projection onto the read iterations is the rectangle $1 \le i_{\mathtt{c}} \le N-1 \wedge 0 \le j_{\mathtt{c}} \le N-1$.

The required buffer size is easily calculated in these cases since in each (overlapping) iteration of any of the loops in the loop nest, the number of data elements produced is exactly the same as the number of elements consumed. The channel will therefore never contain more data elements than right before the first data element is read, or equivalently, right after the last data element is written. To compute the buffer size, we therefore simply need to take the first read iteration and count the number of write iterations that are lexicographically smaller than this read iteration. Although counting the number of elements in the resulting sets is very easy, we use the `barvinok` library [29] to perform this counting. This library is designed for more complicated cases but also detects and appropriately handles these easy cases. In the example, the first read operation occurs at iteration $(1, 0)$ and so we need to compute

$$\begin{aligned} &\#\left(S \cap \{\, (i'_{\mathtt{c}}, j'_{\mathtt{c}}) \mid i'_{\mathtt{c}} < 1 \,\}\right) + \\ &\#\left(S \cap \{\, (i'_{\mathtt{c}}, j'_{\mathtt{c}}) \mid i'_{\mathtt{c}} = 1 \wedge j'_{\mathtt{c}} < 0 \,\}\right), \end{aligned} \tag{10}$$

with S the set of write iterations

$$S = \{\, (i'_{\mathtt{c}}, j'_{\mathtt{c}}) \mid 0 \le i'_{\mathtt{c}} \le N-2 \wedge 0 \le j'_{\mathtt{c}} \le N-1 \,\}.$$

The result of this computation is $N + 0 = N$.

C. General Self-loop FIFOs

We currently do not have a completely satisfactory method of computing the buffer sizes for the general case. One option is to compute the number of array elements in the original program that are written to the channel. That is, we can intersect the domain of write iterations with the access relation and project onto the array space. The resulting (union of) sets can be enumerated symbolically using a library such as `barvinok`. The result may however be a large overestimate of the actual buffer size requirements.

The actual amount of data in a channel at any given iteration can be computed fairly easily. We simply compute the number

or read iterations that are executed before a given read operation and subtract the resulting expression from the number of write iterations that are executed before the given read operation. This computation can again be performed entirely symbolically and the result is a piecewise (quasi-)polynomial in the read iterators and the parameters. The required buffer size is the maximum of this expression over all read iterations. Computing this *parametric* maximum remains an obstacle, however. A possible approach is to use symbolic Bernstein expansion [3], but to the best of our knowledge this technique has not been implemented yet.

For sufficiently regular problems, we can still compute the above maximum symbolically by performing some simplifications and indentifying some special cases. We will not discuss these issues any further here. For *non-parametric* problems, it is usually easier to *simulate* the communication channel. That is, we use CLooG [1] to generate code that increments a counter for each iteration writing to the channel and decrements the counter for each read iteration. The maximum value attained by this counter is recorded and reflects the channel size.

D. Edge FIFOs

Computing the sizes of self-loop channels is relatively easy because the order of execution within a node of the network is fixed. The relative order of iterations from different nodes is not known a priori, however, since this order is determined at run-time. Computing minimal deadlock-free buffer sizes is a non-trivial global optimizations problem. This problem becomes easier if we first compute a deadlock-free schedule and then compute the buffer sizes for each channel individually. Note that this schedule is only computed for the purpose of computing the buffer sizes and is discarded afterwards. The schedule we compute may not be optimal and the resulting buffer sizes may not be valid for the optimal schedule. Our computations do ensure, however, that a valid schedule exists for the computed buffer sizes.

The schedule is computed using a greedy approach. This approach may not work for process networks in general, but is does work for any network derived from a SANLP. The basic idea is to place all iteration domains in a common iteration space at an offset that is computed by the scheduling algorithm. As in the individual iteration spaces, the execution order in this common iteration space is the lexicographical order. By fixing the offsets of the iteration domain in the common space, we have therefore fixed the relative order between any pair of iterations from any pair of iteration domains. The algorithm starts by computing for any pair of connected nodes, the minimal dependence distance vector, a distance vector being the difference between a read operation and the corresponding write operation. Then the nodes are greedily combined, ensuring that all minimal distance vectors are (lexicographically) positive. The end result is a schedule that ensures that every data element is written before it is read. For more information on this algorithm, we refer to [27], where it is applied to perform loop fusion on SANLPs. Note that unlike the case of loop fusion, we can ignore anti-dependences here, unless we want to use the declared size of an array as an estimate for the buffer size of the corresponding channels. (Anti-dependences are ordering constraints between reads and subsequent writes that ensure an array element is not overwritten before it is read.)

After the scheduling, we may consider all channels to be self-loops of the common iteration space and we can apply the techniques from the previous sections with the following qualifications. We will not be able to compute the absolute minimum buffer sizes, but at best the minimum buffer sizes for the computed schedule. We cannot use the declared size of an array as an estimate for the channel size, unless we have taken into account anti-dependences. An estimate that remains valid is the number of write iterations.

We have tacitly assumed above that all iteration domains have the same dimension. If this is not the case, then we first need to assign a dimension of the common (bigger) iteration space to each of the dimensions of the iteration domains of lower dimension. For example, the single iterator of the first loop of the program in Figure 1 would correspond to the outer loop of the 2D common iteration space, whereas the single iterator of the second loop would correspond to the inner loop, as shown in Figure 2. We currently use a greedy heuristic to match these dimensions, starting from domains with higher dimensions and matching dimensions that are related through one or more dependence relations. During this matching we also, again greedily, take care of any scaling that may need to be performed to align the iteration domains. Although our heuristics seem to perform relatively well on our examples, it is clear that we need a more general approach such as the linear transformation algorithm of [28].

V. Worked-out Examples

In this section, we show the results of applying our optimization techniques to two image processing algorithms. The generated Process Networks (PN) enjoy a reduction in the amount of data transferred between nodes and reduced memory requirements, resulting in a better performance, i.e., a reduced execution time. The first algorithm is the Sobel operator, which estimates the gradient of a 2D image. This algorithm is used for edge detection in the pre-processing stage of computer vision systems. The second algorithm is a forward Discrete Wavelet Transform (DWT). The Wavelet transform is a function for multi-scale analysis and has been used for compact signal and image representations in de-noising, compression, and feature detection processing problems for about twenty years.

A. Sobel Edge Detection

The Sobel edge detection algorithm is described by the source code in Figure 5. To estimate the gradient of an image the algorithm performs a convolution between the image and a 3x3 convolution mask. The mask is slid over the image, manipulating a square of 9 pixels at a time, i.e., each time 9 image pixels are read and 1 value is produced. The value represents

```
for (j=0; j < Nrw; j++)
  for (i=0; i < Ncl; i++)
    a[j][i] = ReadImage();

for (j=1; j < Nrw-1; j++)
  for (i=1; i < Ncl-1; i++)
    Sbl[j][i] = Sobel(a[j-1][i-1], a[j][i-1], a[j+1][i-1],
                      a[j-1][  i], a[j][  i], a[j+1][  i],
                      a[j-1][i+1], a[j][i+1], a[j+1][i+1]);
```

Fig. 5. Source code of a Sobel edge detection example.

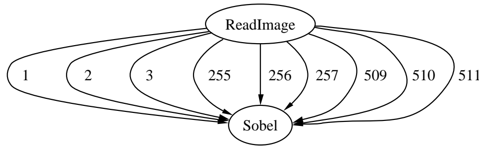

Fig. 6. Compaan generated Process Network for the Sobel example.

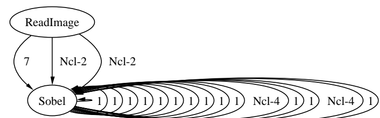

Fig. 7. The generated Process Network for the Sobel example using the self reuse technique.

```
#define A(j,i) (j>=0 && i>=0 && i<Ncl ? a[j][i] : noise)
#define S(j,i) (j>=1 && i>=1 && i<Ncl-1 ? Sbl[j][i] : noise)

for (j=0; j < Nrw; j++)
  for (i=0; i < Ncl; i++)
    a[j][i] = ReadImage();

for (j=-1; j < Nrw-1; j++)
  for (i=-1; i < Ncl+1; i++)
    S(j,i) = Sobel(A(j-1, i-1), A(j, i-1), A(j+1, i-1),
                   A(j-1,   i), A(j,   i), A(j+1,   i),
                   A(j-1, i+1), A(j, i+1), A(j+1, i+1));
```

Fig. 8. Modified source code of the Sobel edge detection example.

the approximated gradient in the center of the processed image area. Applying the regular dataflow analysis on this example using Compaan results in the Process Network (PN) depicted in Figure 6. It contains 2 nodes (representing the `ReadImage` and `Sobel` functions) and 9 channels (representing the parameters of the `Sobel` function). Each channel is marked with a number showing the buffer size it requires. These numbers were obtained by running a simulation processing an image of 256x256 pixels (`Nrw`=`Ncl`=256). The `ReadImage` node reads the input image from memory pixel by pixel and sends it to the `Sobel` node through the 9 channels. Since the 9 pixel values are read in parallel, the executions of the `Sobel` node can start after reading 2 lines and 3 pixels from memory.

After detecting self reuse through read accesses from the same statement as described in Section III-D, we obtain the PN in Figure 7. Again, the numbers next to each channel specify the buffer sizes of the channels. We calculated them at compile time using the techniques described in Section IV. The number of channels between the nodes is reduced from 9 to 3 while several self-loops are introduced. Reducing the communication load between nodes is an important issue since it influences the overall performance of the final implementation. Each data element transferred between two nodes introduces a communication overhead which depends on the architecture of the system executing the PN. For example, if a PN is mapped onto a multiprocessor system with a shared bus architecture, then the 9 pixel values are transferred sequentially through the shared bus, even though in the PN model they are specified as 9 (parallel) channels (Figure 6). In this example it is clear that the PN in Figure 7 will only suffer a third of the communication overhead because it contains 3 times fewer channels between the nodes. The self-loops are implemented using the local processor memory and they do not use the communication resources of the system. Moreover, most of the self loops require only 1 register which makes their implementations simpler than the implementation of a communication channel (FIFO). This also holds for PNs implemented as dedicated hardware. A single-register self-loop is much cheaper to implement in terms of HW resources than a FIFO channel. Another important issue (in both SW and HW systems) is the memory requirement. For the PN in Figure 6 the total amount of memory required is 2304 locations, while the PN in Figure 7 requires only 1033 (for a 256x256 image). This shows that the detection of self reuse reduces the memory requirements by a factor of more than 2.

In principle, the three remaining channels between the two nodes could be combined into a single channel, but, due to boundary conditions, the order in which data would be read from this channel is different from the order in which it is written and we would therefore have a reordering channel (see Section III-C). Since the implementation of a reordering channel is much more expensive than that of a FIFO channel, we do not want to introduce such reordering. The reason we still have 9 channels (7 of which are combined into a single channel) after reuse detection, is that each access reads at least some data for the first time. We can change this behavior by extending the loops with a few iterations, while still only reading the same data as in the original program. All data will then be read for the first time by access `a[j+1][i+1]` only, resulting in a single FIFO between the two nodes. To ensure that we only read the required data, some of the extra iterations of the accesses do not read any data. We can effectuate this change in C by using (implicit) temporary variables and, depending on the index expressions, reading from "`noise`", as shown in Figure 8. By using the simple copy propagation technique of Section III-E, these modifications do not increase the number of nodes in the PN.

The generated optimized PN shown in Figure 9 contains only one (FIFO) channel between the `ReadImage` and `Sobel` nodes. All other communications are through self-loops. Thus, the communication between the nodes is reduced 9 times compared to the initial PN (Figure 6). The total memory requirements for a 256x256 image have been reduced by a factor of almost 4.5 to 519 locations. Note that the results of the extra iterations of the `Sobel` node, which partly operate on "noise", are discarded and so the final behavior of the algorithm remains unaltered. However, with the reduced

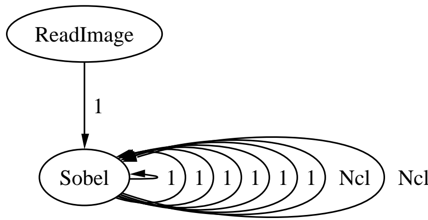

Fig. 9. The generated PN for the modified Sobel edge detection example.

```
    for( i=0; i<2*Nrw; i++ )
      for( j=0; j<2*Ncl; j++ )
 3      a[i][j] = ReadImage();

    for( i=0; i<Nrw; i++ ) {
      // 1D DWT in vertical direction with subsampling
      for( j=0; j<2*Ncl; j++ ) {
 8      tmpLine = (i==Nrw-1) ? a[2*i][j] : a[2*i+2][j];
        Hf[j] = high_flt_vert( a[2*i][j], a[2*i+1][j], tmpLine );

        tmp = (i==0) ? Hf[j] : oldHf[j];
        low_flt_vert( tmp,a[2*i][j],Hf[j],&oldHf[j],&Lf[j] );
13    }

      // 1D DWT in horizontal direction with subsampling ---------
      for( j=0; j<Ncl; j++ ) {
        tmp = (j==Ncl-1) ? Lf[2*j] : Lf[2*j+2];
18      HL[i][j] = high_flt_hor( Lf[2*j], Lf[2*j+1], tmp );

        tmp = (j==0) ? HL[i][j] : HL[i][j-1];
        LL[i][j] = low_flt_hor( tmp, Lf[2*j],  HL[i][j] );
      }
23
      // 1D DWT in horizontal direction with subsampling ---------
      for( j=0; j<Ncl; j++ ) {
        tmp = (j==Ncl-1) ? Hf[2*j] : Hf[2*j+2];
        HH[i][j] = high_flt_hor( Hf[2*j], Hf[2*j+1], tmp );
28
        tmp = (j == 0) ? HH[i][j] : HH[i][j-1];
        LH[i][j] = low_flt_hor( tmp, Hf[2*j],  HH[i][j] );
      }
    }
33 // The Outputs  -----------------------------------------------
    for( i=0; i<Nrw; i++)
      for( j=0; j<Ncl; j++) {
        Sink( LL[i][j] );
        Sink( HL[i][j] );
38      Sink( LH[i][j] );
        Sink( HH[i][j] );
      }
```

Fig. 10. Source code of a Discrete Wavelet Transform example.

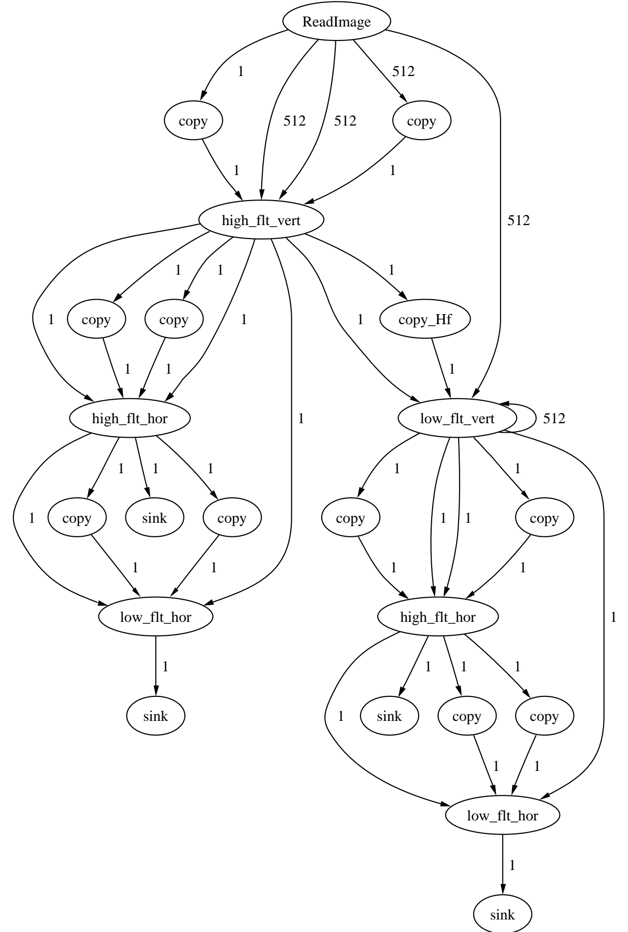

Fig. 11. 2D-DWT Process Network with Copy nodes.

number of communication channels and overhead, the final (real) implementation of the optimized PN will have a better performance.

B. Discrete Wavelet Transform

In the Discrete Wavelet Transform (DWT) the input image is decomposed into different decomposition levels. These decomposition levels contain a number of sub-bands, which consist of coefficients that describe the horizontal and vertical spatial frequency characteristics of the original image. The DWT requires the signal to be extended periodically. This periodic symmetric extension is used to ensure that for the filtering operations that take place at both boundaries of the signal, one signal sample exists and spatially corresponds to each coefficient of the filter mask. The number of additional samples required at the boundaries of the signal is therefore filter-length dependent.

The C program realizing one level of a 2D forward DWT is presented in Figure 10. In this example we use a lifting scheme of a reversible transformation with 5/3 filter [5]. In this case the image has to be extended with one pixel at the boundaries. All the boundary conditions are described by the conditions in code lines 8, 11, 17, 20, 26 and 29.

First, a 1D DWT is applied in the vertical direction (lines 7 to 13). Two intermediate variables are produced (low- and high-pass filtered images sub-sampled by 2—lines 9 and 12). They are further processed by a 1D DWT applied in the horizontal direction and thus producing (again sub-sampled by 2) a four sub-bands decomposition: HL (line 18), LL (line 21), HH (line 27), and LH (line 30). The Process Network generated by using the regular dataflow analysis (and Compaan tool) is depicted in Figure 11. The PN contains 22 nodes, half of them just copying pixels at the boundaries of the image. Channel sizes are estimated by running a simulation again processing an image 256x256 pixels. Although most of the channels have size 1, they cannot be implemented by a simple register since they connect nodes and additional logic (FIFO like) is required for synchronization. Obviously, the generated PN has considerable initial overhead.

The optimization goals for this example are to remove the `Copy` nodes and to reduce the communication between the nodes as much as possible. We achieve these goals by applying our techniques. The optimized Process Network is shown in Figure 12. The simple copy propagation technique reduces the number of the nodes from 22 to 11 and the detection of self

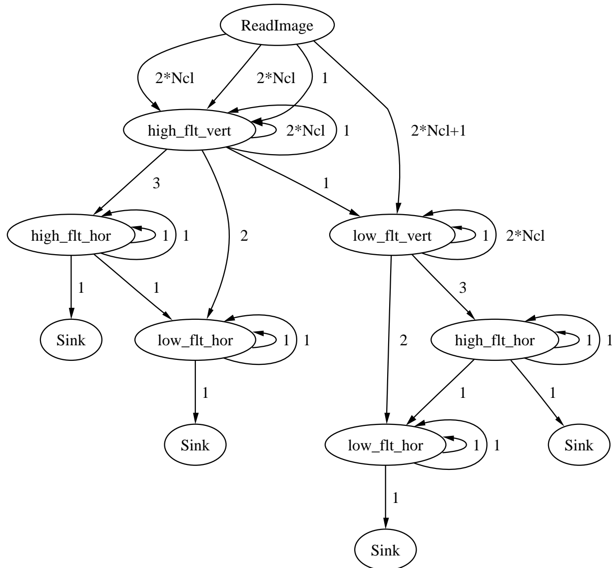

Fig. 12. Optimized 2D-DWT Process Network.

reuse technique reduces the communication between the nodes from 38 to 15 channels introducing 11 self-loop channels. There is only one channel connecting two nodes of the PN in Figure 12, except for the channels between the `ReadImage` and `high_filt_vert` nodes. In this case we detect that a combined channel would be reordering. As we mentioned in the previous example we prefer not to introduce reordering and therefore generate more (FIFO) channels. As a result, the number of channels emanating from the `ReadImage` has been reduced by only one compared to the initial PN (Figure 11). The buffer sizes are calculated at compile time using our techniques described in Section IV. Note that in this example applying the optimization techniques has little effect on the memory requirements: the number of memory locations required for an image of 256x256 pixels is 2586 compared to 2594 for the initial DWT PN. However, the topology of the optimized PN has been simplified significantly allowing an efficient HW and/or SW implementation.

VI. Comparison to Compaan-generated Networks

Table I compares the number of channels in Compaan-generated networks to the number of channels in our networks. The total number of channels is shown for each example as well as a decomposition into different types of channels. In-Order (IO) and Out-of-Order (OO) refer to FIFOs and reordering channels respectively and the M-suffix refers to multiplicity, which does not occur in our networks. Each column is further split into self-loops+edges, or single-register+self-loops+edges for our FIFOs.

Note that a direct comparison of the numbers is unfair since some of our channels are split into several Compaan-channels due to a difference in internal representation. In Compaan, these channels are recombined, with possibly further combinations, at a later stage. The numbers for combined channels are

Algorithm name	Compaan Networks					Our Networks		
	all	IO	IOM	OO	OOM	all	IO	OO
LU-Factor	35	4+17	1+5	0+5	0+1	27	3+ 5 +16	0+3
QR-Decomp	12	4+8	0+0	0+0	0+0	12	1+ 3 +8	0+0
SVD	118	4+80	0+4	0+30	0+0	60	10+ 0 +34	0+16
Faddeev	28	3+21	0+3	0+1	0+0	26	4+ 2 +19	0+1
Gauss-Elim.	11	2+5	0+0	0+1	1+2	13	0+ 6 +6	0+1
Motion Est.	98	27+71	0+0	0+0	0+0	120	0+54+66	0+0
M-JPEG	50	9+24	0+17	0+0	0+0	56	18+ 0 +38	0+0

TABLE I
Comparison to channel numbers of Compaan networks

reported in [23]. We can however conclude that our techniques have split all OOM channels in these examples into pairs of FIFOs, that in general we have fewer channels between different nodes at the expense of more self-loops and that many of these self-loops are "single-register" FIFOs, where "register" should be interpreted as "token", which may be a whole table in the case of M-JPEG.

VII. Related Work

Process Networks are supported by the Ptolemy II framework [15] and the YAPI environment [6] for concurrent modeling and design of applications and systems. The designer has to manually specify the application as a Process Network and to give this network as an input to the Ptolemy II or YAPI simulation and verification engines. In many cases manually specifying an application as a Process Network is a very time consuming and error prone process. Our work, presented in this paper, can be used as a front-end tool by Ptolemy II or YAPI. This will significantly speedup the modeling effort when Process Networks are used and avoid modeling errors because our techniques guarantee a correct-by-construction generation of Process Networks.

Process Networks have been used to model applications and to explore the mapping of these applications onto multi-processor architectures [7], [16], [19], [24]. The application modeling is performed manually starting from sequential C code and a significant amount of time (a few weeks) is spent by the designers on correctly transforming the sequential C code into Process Networks. This activity slows down the design space exploration process. The work presented in this paper gives a solution for fast automatic derivation of Process Networks from sequential C code that will contribute to faster design space exploration.

The relation of our analysis to Compaan has already been highlighted throughout the text. As to memory size requirements, much research has been devoted to optimal reuse of memory for arrays. For an overview and a general technique, we refer to [4]. These techniques are complementary to our research on FIFO sizes and can be used on the reordering channels and optionally on the data communication inside a node. Also related is the concept of reuse distances [2]. In particular, our FIFO sizes are a special case of the "reuse distance per statement" of [26]. For more advanced forms of copy propagation, we refer to [25].

VIII. Conclusions and Discussion

In this paper we have improved upon the state-of-the-art conversion of sequential programs to Process Networks in several ways. We have shown that we can reduce the number of reordering channels as well the total number of channels between different nodes by extending the standard dataflow analysis to detect reuse within a node. This effect is enhanced by first removing the (artificial) copy nodes introduced by Compaan through simple copy propagation. These techniques lead to a removal of all reordering channels with multiplicity that appear in our examples and a reduction of the communication volume by up to a factor 9 in the extreme case. We have further shown how to compute the FIFO sizes exactly for self-loops and approximately for other channels for some important special cases. Generalizing these results will likely require more advanced symbolic techniques such as Bernstein expansion [3].

Acknowledgment

We thank Bart Kienhuis for his help and for the discussions on some of the topics in this paper and Leids Universiteits Fonds (LUF) for the financial support.

References

[1] C. Bastoul, "Code generation in the polyhedral model is easier than you think," in *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, Juan-les-Pins, september 2004, pp. 7–16.

[2] K. Beyls and E. D'Hollander, "Generating cache hints for improved program efficiency," *Journal of Systems Architecture*, vol. 51, no. 4, pp. 223–250, 2005.

[3] P. Clauss and I. Tchoupaeva, "A symbolic approach to Bernstein expansion (in russian)," *Programmirovanie (Programming and Computer Software)*, vol. 30, no. 3, 2004.

[4] A. Darte, R. Schreiber, and G. Villard, "Lattice-based memory allocation," in *Proceedings of the international conference on Compilers, architectures and synthesis for embedded systems*. ACM Press, 2003, pp. 298–308.

[5] I. Daubechies and W. Sweldens, "Factoring wavelet transforms into lifting schemes," *Fourier Analysis and Applications*, vol. 4, pp. 249–269, 1998.

[6] 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)*, Los Angeles, CA, June 5-9 2000, pp. 402–405.

[7] E. de Kock, "Multiprocessor Mapping of Process Networks: A JPEG Decoding Case Study," in *Proc. 15th Int. Symposium on System Synthesis (ISSS'2002)*, Kyoto, Japan, Oct. 2-4 2002, pp. 68–73.

[8] B. Dwivedi, A. Kumar, and M. Balakrishnan, "Automatic Synthesis of System on Chip Multiprocessor Architectures for Process networks," in *Proc. Int. Conference on Hardware/Software Codesign and System Synthesis*, Stockholm, Sweden, Jan. 8-10 2004.

[9] P. Feautrier, "Parametric integer programming," *Operationnelle/Operations Research*, vol. 22, no. 3, pp. 243–268, 1988.

[10] ——, "Dataflow analysis of array and scalar references," *International Journal of Parallel Programming*, vol. 20, no. 1, pp. 23–53, 1991.

[11] K. Goossens, J. Dielissen, J. v. Meerbergen, P. Poplavko, A. Radulescu, E. Rijpkema, E. P. Waterlander, and P. Wielage, "Guaranteeing the Quality Of Services in Networks On Chip," in *Networks on Chip*. Kluwer Academic Publishers, 2003, pp. 61–82.

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

[13] B. Kienhuis, E. Rijpkema, and E. F. Deprettere, "Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures," in *Proc. 8th International Workshop on Hardware/Software Codesign (CODES'2000)*, San Diego, CA, USA, May 3-5 2000.

[14] E. Lee and A. Sangiovanni-Vincentelli, "A Framework for Comparing Models of Computation," *IEEE Transactions on CAD of Integrated Circuits and Systems*, vol. 17, no. 12, pp. 1217–1229, 1998.

[15] E. Lee et al., "PtolemyII: Heterogeneous Concurrent Modeling and Design in Java," University of California at Berkeley, Tech. Rep., 1999, uCB/ERL M99/40.

[16] P. Lieverse, T. Stefanov, P. van der Wolf, and E. Deprettere, "System Level Design with SPADE: an M-JPEG Case Study," in *Proc. Int. Conference on Computer Aided Design (ICCAD'01)*, San Jose CA, USA, Nov. 4-8 2001, pp. 31–38.

[17] A. Mihal and K. Keutzer, "Mapping Concurrent Applications onto Architectural Platforms," in *Networks on Chips*, A. Jantsch and H. Tenhunen, Eds. Kluwer Academic Publishers, 2003, pp. 39–59.

[18] A. Nieuwland, J. Kang, O. P. Gangwal, R. Sethuraman, N. Busa, K. Goosens, R. P. Llopis, and P. Lippens, *C-HEAP: A Heterogeneous Multi-processor Architecture Template and Scalable and Flexible Protocol for the Design of Embedded Signal Processing Systems*. Kluwer Academic Publishers, 2002.

[19] A. Pimentel, C. Erbas, and S. Polstra, "A Systematic Approach to Exploring Embedded System Architectures at Multiple Abstraction Levels," *IEEE Transactions on Computers*, vol. 55, no. 2, to appear 2006.

[20] E. Rijpkema, E. F. Deprettere, and B. Kienhuis, "Deriving Process Networks from Nested Loop Algorithms," *Parallel Processing Letters*, vol. 10, no. 2, pp. 165–176, 2000.

[21] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprettere, "System Design using Kahn Process Networks: The Compaan/Laura Approach," in *Proc. Int. Conference Design, Automation and Test in Europe (DATE'04)*, Paris, France, Feb. 16-20 2004, pp. 340–345.

[22] A. Turjan, B. Kienhuis, and E. Deprettere, "A Hierarchical Classification Scheme to Derive Interprocess Communication in Process Networks," in *Proceedings of the IEEE 14th Int. Conf. on Application-specific Systems, Architectures and Processors (ASAP'04)*, Galveston, Texas, Sept 27-29 2004.

[23] ——, "Translating affine nested-loop programs to process networks," in *Proc. International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES'04)*, Washington D.C., USA, Sept. 23-25 2004, pp. 220–229.

[24] P. van der Wolf, P. Lieverse, M. Goel, D. La Hei, and K. Vissers, "An MPEG-2 Decoder Case Study as a Driver for a System Level Design Methodology," in *Proc. 7th Int. Workshop on Hardware/Software Codesign (CODES'99)*, Rome, Italy, May 3-5 1999.

[25] P. Vanbroekhoven, G. Janssens, M. Bruynooghe, H. Corporaal, and F. Catthoor, "Advanced copy propagation for arrays," in *Proceedings of Languages, Compilers and Tools for Embedded Systems 2003, San Diego, California*, U. Kremer, Ed. ACM, June 2003, pp. 24–33.

[26] T. Vander Aa, M. Jayapal, F. Barat, H. Corporaal, F. Catthoor, and G. Deconinck, "A high-level memory energy estimator based on reuse distance," in *3rd Workshop on Optimization for DSP and Embedded Systems, ODES-3*, Mar. 2005.

[27] S. Verdoolaege, M. Bruynooghe, G. Janssens, and F. Catthoor, "Multi-dimensional incremental loop fusion for data locality," in *IEEE 14th International Conference on Application-specific Systems, Architectures and Processors*, D. Martin, Ed., The Hague, The Netherlands, June 2003, pp. 17–27.

[28] S. Verdoolaege, K. Danckaert, F. Catthoor, M. Bruynooghe, and G. Janssens, "An access regularity criterion and regularity improvement heuristics for data transfer optimization by global loop transformations," in *1st Workshop on Optimization for DSP and Embedded Systems, ODES*, Mar. 2003.

[29] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe, "Analytical computation of Ehrhart polynomials: Enabling more compiler analyses and optimizations," in *Proceedings of International Conference on Compilers, Architectures, and Synthesis for Embedded Systems, Washington D.C.*, Sept. 2004, pp. 248–258.