Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2007, Article ID 75947, 13 pages
doi:10.1155/2007/75947

Research Article

pn: A Tool for Improved Derivation of Process Networks

Sven Verdoolaege, Hristo Nikolov, and Todor Stefanov

Leiden Institute of Advanced Computer Science (LIACS), Leiden University, Niels Bohrweg 1, 2333 CA, Leiden, The Netherlands

Received 30 June 2006; Revised 12 December 2006; Accepted 10 January 2007

Recommended by Shuvra Bhattacharyya

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 for facilitating 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.

1. 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 System-on-Chip platforms have to be multiprocessor architectures. In 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, that is, 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 a 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 [1]. However, 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 a sequential program and a parallel model of computation motivates us to investigate 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 [2, 3], in this paper we consider the process network model of computation [4] 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 [5–14] 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 [15–17]. The techniques presented in this paper and implemented in the `pn` tool of our `isa` tool set 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. Our techniques try to address optimization issues in four main aspects.

Given an application specified as an 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.

2. RELATED WORK

The work in [11] presents a methodology and techniques implemented in a tool called ESPAM for automated multiprocessor system design, programming, and implementation. The ESPAM design flow starts with three input specifications at the system level of abstraction, namely a platform specification, a mapping specification, and an application specification. ESPAM requires the application specification to be a process network. Our compiler techniques presented in this paper are primarily intended to be used as a font-end tool for ESPAM. (Kahn) process networks are also supported by the Ptolemy II framework [3] and the YAPI environment [5] for concurrent modeling and design of applications and systems. In many cases, manually specifying an application as a process network is a very time consuming and error prone process. Using our techniques as a front-end to these tools can 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 multiprocessor architectures [6, 9, 12, 14]. 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 will be 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 [18]. 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 [19]. In particular, our FIFO sizes are a special case of the "reuse distance per statement" of [20]. For more advanced forms of copy propagation, we refer to [21].

The rest of this paper is organized as follows. In Section 3, we first introduce some concepts that we will need throughout this paper. We explain how to derive and optimize process networks in Section 4 and how to compute FIFO sizes in Section 5. Detailed examples are given in Section 6, with a further comparison to Compaan-generated networks in Section 7. In Section 8, we conclude the paper.

3. PRELIMINARIES

In this section, we introduce the process network model, discuss static affine nested loop programs (SANLPs) and our internal representation, and introduce our main analysis tools.

3.1. The process network model

As the name 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 particular, a process will block if it needs data from a channel that is not available yet. Similarly, a process will block if it tries to write to a "full" channel.

In the special case of a Kahn process network (KPN), the communication channels are unbounded FIFOs that can only block on a read. In the more general case, data can 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*. Furthermore, the FIFO channels have additional properties such as their size and the ability to be implemented as a shift register. Since both reads and writes may block, it is important 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 3.2. Our networks can be used as input for tools that expect Kahn process networks by ignoring the additional properties of FIFO channels and by changing the order in which a process reads from a reordering channel to match the order of the writes and storing the data that is not needed yet in an internal memory block.

3.2. Limitations on the input and internal representation

The SANLPs are programs or program fragments that can be represented in the well-known polytope model [22]. That is, an 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 can contain enclosing loop iterators and parameters as well as modulo and integer divisions, but no products of these elements. Such expressions are called quasi-affine. The parameters are symbolic constants, that is, their values may not change during the execution of the program fragment. These restrictions allow a compact representation of the program through sets and relations of integer vectors defined by linear (in)equalities, existential quantification, and the union operation. More technically, our (parametric) "integer sets" and "integer relations" are (disjoint) unions of projections of the integer points in (parametric) polytopes.

In particular, the set of iterator vectors for which a statement is executed is an integer set called the *iteration domain*. The linear inequalities of this set correspond to the lower and upper bounds of the loops enclosing the statement. For example, the iteration domain of statement `S1` in Figure 1 is $\{i \mid 0 \le i \le N-1\}$. The elements in these sets are ordered according to the order in which the iterations of the loop nest are executed, assuming the loops are normalized to have step +1. 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,

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

The 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, that is, $\{(\mathbf{i}, \mathbf{a}) \in I \times A \mid \mathbf{a} = L\mathbf{i} + \mathbf{m}\}$, where I is the iteration domain, A is the array space, and $L\mathbf{i} + \mathbf{m}$ is the affine access function.

The use of access *relations* allows us to impose additional constraints on the iterations where the access occurs.

```
for (i = 0; i < N; ++i)
S1: b[i] = f(i > 0 ? a[i-1] : a[i], a[i],
             i < N-1 ? a[i+1] : a[i]);
for (i = 0; i < N; ++i) {
    if (i > 0)
        tmp = b[i-1];
    else
        tmp = b[i];
S2: c[i] = g(b[i], tmp);
}
```

Figure 1: Use of temporary variables to express border behavior.

This is useful for expressing the effect of the ternary operator (`?:`) in C, or, equivalently, the use of temporary scalar variables. These frequently occur in multimedia applications where one or more kernels uniformly manipulate a stream of data such as an image, but behave slightly differently at the borders. An example of both ways of expressing border behavior is shown in Figure 1 on a 1D data stream. The second read access through `b` in line 9, after elimination of the temporary variable `tmp`, can be represented as

$$R = \{(i,a) \mid a = i-1 \wedge 1 \le i \le N-1\} \cup \{(i,a) \mid a = i = 0\}. \tag{2}$$

To eliminate such temporary variables, we first identify the statements that simply copy data to a temporary variable, perform a dataflow analysis (as explained in Section 4.1) on those temporary variables in a first pass and combine the resulting constraints with the access relation from the copy statement. A straightforward transformation of code such as that of Figure 1 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 reducing internode communication.

3.3. Analysis tools: lexicographical minimization and counting

Our main analysis tool is parametric integer programming [23], which computes the lexicographically smallest (or largest) 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 and possibly some additional existentially quantified variables. This result can be described as a union of parametric integer sets, where each set in the union contains a single point, or alternatively as a relation, or indeed a function, between (some of) the parameters and the corresponding lexicographical minimum. The existentially quantified variables that may appear will always be uniquely quantified, that is, the existential quantifier $\exists$ is actually a uniqueness quantifier $\exists!$. 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.

The `barvinok` library [24] efficiently computes the number of integer points in a parametric polytope. We can use it to compute the number of points in a parametric set provided that the existentially quantified variables are uniquely quantified, which can be ensured by first using PIP if needed. The result of the computation is a compact representation of a function from the parameters to the nonnegative integers, the number of elements in the set for the corresponding parameter values. In particular, the result is a piecewise quasipolynomial in the parameters. The `bernstein` library [25] can be used to compute an upper bound on a piecewise polynomial over a parametric polytope.

4. DERIVATION OF PROCESS NETWORKS

This section explains the conversion of SANLPs to process networks. We first derive the channels using a modified dataflow analysis in Section 4.1 and then we show how to determine channel types in Section 4.2 and discuss some optimizations on self-loop channels in Section 4.3.

4.1. Dataflow analysis

To compute the channels between the nodes, we basically need to perform array dataflow analysis [26]. That is, for each execution of a read operation of a given data element in a statement, we need to find the source of the data, that is, the corresponding write operation that wrote the data element. However, to reduce communication between different nodes and in contrast to standard dataflow analysis, we will also consider all previous read operations from the same statement as possible sources of the data.

The problem to be solved is then: given a read from an array element, what was the last write to or read (from that statement) from that array element? The last iteration of a statement satisfying some constraints can be obtained using PIP, where we compute the lexicographical *maximum* of the write (or read) source operations in terms of the iterators of the "sink" read operation. Since there may be multiple statements that are potential sources of the data and since we also need to express that the source operation is executed before the read (which is not a linear constraint, but rather a disjunction of n linear constraints (1), where n is the shared nesting level), we actually need to perform a number of PIP invocations. For details, we refer to [26], keeping in mind that we consider a larger set of possible sources.

For example, the first read access in statement `S2` of the code in Figure 1 reads data written by statement `S1`, which results in a channel from node "`S1`" to node "`S2`." In particular, data flows from iteration i_w of statement `S1` to iteration $i_r = i_w$ of statement `S2`. This information is captured by the integer relation

$$D_{\mathtt{S1}\to\mathtt{S2}} = \{(i_w, i_r) \mid i_r = i_w \wedge 0 \le i_r \le N-1\}. \tag{3}$$

For the second read access in statement `S2`, as described by (2), the data has already been read by the same statement after it was written. This results in a self-loop from `S2` to itself described as

$$\begin{aligned} D_{\mathtt{S2}\to\mathtt{S2}} = {} & \{(i_w, i_r) \mid i_w = i_r - 1 \wedge 1 \le i_r \le N-1\} \\ & \cup \{(i_w, i_r) \mid i_w = i_r = 0\}. \end{aligned} \tag{4}$$

In general, we obtain pairs of write/read and read operations such that some data flows from the write/read operation to the (other) 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

$$\bigcup_{j=1}^{m} D_j(\mathbf{i}_w, \mathbf{i}_r) \subset \mathbb{Z}^{n_1} \times \mathbb{Z}^{n_2}, \tag{5}$$

with n_1 and n_2 the number of loops enclosing the write and read operation respectively, that connect the specific iterations of the write/read and read operations such that the first is the source of the second. As such, each iteration of a given read operation is uniquely paired off to some write or read operation iteration. 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, a characteristic explained in the next section.

4.2. 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 if and only if there exist two pairs of write and read iterations, $(\mathbf{w}_1, \mathbf{r}_1), (\mathbf{w}_2, \mathbf{r}_2) \in \mathbb{Z}^{n_1} \times \mathbb{Z}^{n_2}$, such that the order of the write operations is different from the order of the read operations, that is, $\mathbf{w}_1 \succ \mathbf{w}_2$ and $\mathbf{r}_1 \prec \mathbf{r}_2$, or equivalently

$$\mathbf{w}_1 - \mathbf{w}_2 \succ \mathbf{0}, \qquad \mathbf{r}_1 \prec \mathbf{r}_2. \tag{6}$$

Given a union of integer relations describing the channel (5), then for any pair of relations in this union, (D_{j_1}, D_{j_2}), we therefore need to solve n_2 PIP problems

$$\begin{aligned} \operatorname{lexmax} \{ & (\mathbf{t}, (\mathbf{w}_1, \mathbf{r}_1), (\mathbf{w}_2, \mathbf{r}_2), \mathbf{p}) \mid \\ & (\mathbf{w}_1, \mathbf{r}_1) \in D_{j_1} \wedge (\mathbf{w}_2, \mathbf{r}_2) \in D_{j_2} \\ & \wedge \mathbf{t} = \mathbf{w}_1 - \mathbf{w}_2 \wedge \mathbf{r}_1 \prec \mathbf{r}_2 \}, \end{aligned} \tag{7}$$

where $\mathbf{r}_1 \prec \mathbf{r}_2$ should be expanded according to (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 $\mathbf{t} = \mathbf{w}_1 - \mathbf{w}_2$ 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 (7) therefore actually represents a nonparametric

```
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];
```

Figure 2: Outer product source code.

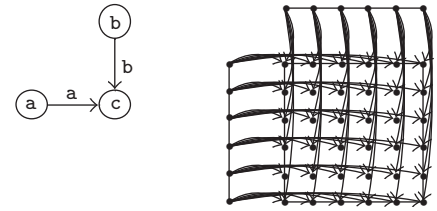

Figure 3: Outer product dependence graph with multiplicity.

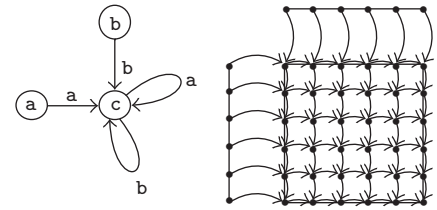

Figure 4: Outer product dependence graph without multiplicity.

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 [17], where it is formulated as $n_1 \times n_2$ PIP problems for a channel described by a single integer relation. A further difference is that the authors of [17] perform a more standard dataflow analysis and therefore also need to consider a second characteristic called *multiplicity*. Multiplicity occurs when the same data is read more than once from the same channel. Since we also consider previous reads from the same node as potential sources in our dataflow analysis, the channels we derive will never have multiplicity, but rather will be split into two channels, one corresponding to the first read and one self-loop channel propagating the value to subsequent reads.

Removing multiplicity not only reduces the communication between different nodes, it can also remove some artificial reorderings. A typical example of this situation is the outer product of two vectors, shown in Figure 2. Figure 3 shows the result of standard dataflow analysis. 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

$$D_{\mathtt{a}\to\mathtt{c}} = \{(i_{\mathtt{a}}, i_{\mathtt{c}}, j_{\mathtt{c}}) \mid 0 \le i_{\mathtt{c}} \le N-1 \wedge 0 \le j_{\mathtt{c}} \le N-1 \wedge i_{\mathtt{a}} = i_{\mathtt{c}}\} \tag{8}$$

and would be classified as nonreordering, 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

$$D_{\mathtt{b}\to\mathtt{c}} = \{(j_{\mathtt{b}}, i_{\mathtt{c}}, j_{\mathtt{c}}) \mid 0 \le i_{\mathtt{c}} \le N-1 \wedge 0 \le j_{\mathtt{c}} \le N-1 \wedge j_{\mathtt{b}} = j_{\mathtt{c}}\} \tag{9}$$

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 4. In this network, all channels, including the new self-loop channels, are FIFOs. For example, the channel with dependence relation $D_{\mathtt{b}\to\mathtt{c}}$ (9) is split into a channel with relation

$$D'_{\mathtt{b}\to\mathtt{c}} = \{(j_{\mathtt{b}}, i_{\mathtt{c}}, j_{\mathtt{c}}) \mid i_{\mathtt{c}} = 0 \wedge 0 \le j_{\mathtt{c}} \le N-1 \wedge j_{\mathtt{b}} = j_{\mathtt{c}}\} \tag{10}$$

and a self-loop channel with relation

$$D_{\mathtt{c}\to\mathtt{c}} = \{(i'_{\mathtt{c}}, j'_{\mathtt{c}}, i_{\mathtt{c}}, j_{\mathtt{c}}) \mid 1 \le i_{\mathtt{c}} \le N-1 \wedge 0 \le j_{\mathtt{c}} \le N-1 \wedge j'_{\mathtt{c}} = j_{\mathtt{c}} \wedge i'_{\mathtt{c}} = i_{\mathtt{c}} - 1\}. \tag{11}$$

4.3. Self-loops

When removing multiplicity from channels, our dataflow analysis introduces extra self-loop channels. Some of these channels can be further optimized. A simple, but important case is that where the channels hold 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 iteration $\mathbf{r}_1$ reading from the same channel in between. In other words, the situation does not occur if and only if 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 (6). The PIP problems that need to be solved to determine this condition are therefore nearly identical to the problems (7), namely.,

$$\begin{aligned}\operatorname{lexmax} \{(\mathbf{t}, (\mathbf{w}_1, \mathbf{r}_1), (\mathbf{w}_2, \mathbf{r}_2), \mathbf{p}) \mid \\ (\mathbf{w}_1, \mathbf{r}_1) \in D_{j_1} \wedge (\mathbf{w}_2, \mathbf{r}_2) \in D_{j_2} \\ \wedge \mathbf{t} = \mathbf{r}_1 - \mathbf{w}_2 \wedge \mathbf{r}_1 \prec \mathbf{r}_2\},\end{aligned} \tag{12}$$

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 (1).

If such a channel has the additional property that the single value it contains is always propagated to the next iteration of the node (a condition that can again be checked using PIP), then we remove the channel completely and attach the register to the input argument of the function and call the FIFO(s) that read the value for the first time "sticky FIFOs." This is a special case of the optimization applied to in-order channels with multiplicity of [17] that allows for slightly more efficient implementations due to the extra property.

Another special case occurs when the number of iterations of the node between a write to the self-loop channel and the corresponding read is a constant, which we can determine by simply counting the number of intermediate iterations (symbolically) and checking whether the result is a constant function. In this case, we can replace the FIFO by a shift register, which can be implemented more efficiently in hardware. Note, however, that there may be a trade-off since the size of the channel as a shift register (i.e., the constant function above) may be larger than the size of the channel as a FIFO. On the other hand, the FIFO size may be more difficult to determine (see Section 5.2).

5. 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, but we are dealing here with the easier case of networks generated from static affine nested loop programs. We first consider self-loops, with a special case in Section 5.1, and the general case in Section 5.2. In Section 5.3, we then explain how to reduce the general case of FIFOs to self-loops by scheduling the network.

5.1. Uniform self-dependences on rectangular domains

An 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 vectors 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 $D_{c\to c}$ (11) from Section 4.2 is a uniform dependence over a rectangular domain since the difference between the read and write iteration vectors is $(i_c, j_c) - (i'_c, j'_c) = (1, 0)$ and since the projection onto the read iterations is the rectangle $1 \le i_c \le N - 1 \wedge 0 \le j_c \le N - 1$.

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 using `barvinok`. In the example, the first read operation occurs at iteration (1,0) and so we need to compute

$$\#\left(S \cap \left\{(i'_c, j'_c) \mid i'_c < 1\right\}\right) + \#\left(S \cap \left\{(i'_c, j'_c) \mid i'_c = 1 \wedge j'_c < 0\right\}\right), \tag{13}$$

with S the set of write iterations

$$S = \left\{(i'_c, j'_c) \mid 0 \le i'_c \le N - 2 \wedge 0 \le j'_c \le N - 1\right\}. \tag{14}$$

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

5.2. General self-loop FIFOs

An easy approximation can be obtained by computing 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 `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 of 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.

For sufficiently regular problems, we can compute the above maximum symbolically by performing some simplifications and identifying some special cases. In the general case, we can apply Bernstein expansion [25] to obtain a parametric *upper bound* on the expression. For *nonparametric* problems, however, it is usually easier to *simulate* the communication channel. That is, we use `CLooG` [27] 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.

5.3. Nonself-loop FIFOs

Computing the sizes of self-loop channels is relatively easy because the order of execution within a node of the network is fixed. However, the relative order of iterations from different nodes is not known a priori since this order is determined at run-time. Computing minimal deadlock-free buffer sizes is a nontrivial global optimization 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

```
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]);
```

Figure 5: Source code of a Sobel edge detection example.

of computing the buffer sizes and is discarded afterward. 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 it does work for any network derived from an 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 [28], where it is applied to perform loop fusion on SANLPs. Note that unlike the case of loop fusion, we can ignore antidependences here, unless we want to use the declared size of an array as an estimate for the buffer size of the corresponding channels. (Antidependences 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 antidependences. 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 2 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 3. 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 [29].

6. 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, that is, 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 preprocessing stage of computer vision systems. The second algorithm is a forward discrete wavelet transform (DWT). The wavelet transform is a function for multiscale analysis and has been used for compact signal and image representations in denoising, compression, and feature detection processing problems for about twenty years.

6.1. 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 3×3 convolution mask. The mask is slid over the image, manipulating a square of 9 pixels at a time, that is, each time 9 image pixels are read and 1 value is produced. The value represents 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)

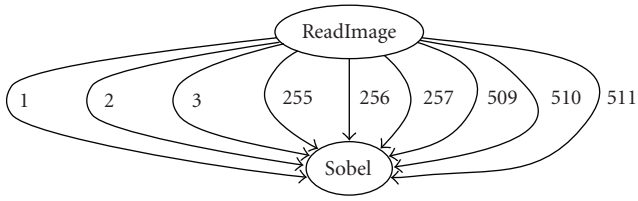

Figure 6: Compaan generated process network for the Sobel example.

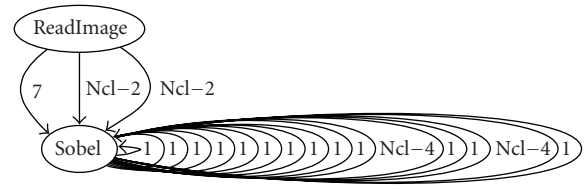

Figure 7: The generated process network for the Sobel example using the self reuse technique.

depicted in Figure 6. It contains 2 nodes (representing the `ReadImage` and `Sobel` functions) and 9 channels (representing the arguments 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 256×256 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 4.1, 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 5. 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 256×256 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 4.2). 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 (manually) 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 3.2, 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 256×256 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 number of communication channels and overhead, the final (real) implementation of the optimized PN will have a better performance.

6.2. 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 subbands, 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

```
#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));
```

Figure 8: Modified source code of the Sobel edge detection example.

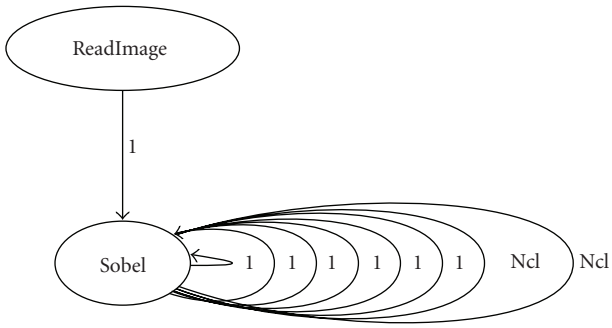

Figure 9: The generated PN for the modified Sobel edge detection example.

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 [30]. 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 subsampled by 2—lines 9 and 12). They are further processed by a 1D DWT applied in the horizontal direction and thus producing (again subsampled by 2) a four subbands 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 23 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 256×256 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 23 to 11 and the detection of self reuse technique reduces the communication between the nodes from 40 to 15 channels introducing 8 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 5 and the correctness of the process network is tested using the YAPI environment [5]. 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 256×256 pixels is 2585 compared to 2603 for the initial DWT PN. However, the topology of the optimized PN has been simplified significantly allowing an efficient HW and/or SW implementation.

7. COMPARISON TO COMPAAN AND COMPAAN-LIKE NETWORKS

Table 1 compares the number of channels in Compaan-like networks to the number of channels in our networks. The Compaan-like networks were generated by using standard dataflow analysis instead of also considering reads as sources and by turning off the copy propagation of temporary scalars and the combination of channels reading from the same write access. The table shows a decomposition of the channels into different types. 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

Table 1: Comparison to channel numbers of Compaan-like networks.

Algorithm name	Compaan-like networks				Our networks	
	IO sl + ed	IOM sl + ed	OO sl + ed	OOM sl + ed	IO 1r + sl + ed	OO sl + ed
LU-Factor	3 + 13	1 + 7	0 + 3	0 + 1	2 + 5 + 16	0 + 3
QR-Decomp	4 + 8	0 + 0	0 + 0	0 + 0	1 + 3 + 8	0 + 0
SVD	4 + 41	0 + 4	0 + 18	0 + 0	8 + 0 + 34	0 + 16
Faddeev	3 + 20	0 + 3	0 + 1	0 + 0	4 + 2 + 19	0 + 1
Gauss-Elim.	2 + 5	0 + 0	0 + 1	1 + 2	0 + 6 + 6	0 + 1
Motion Est.	27 + 66	0 + 0	0 + 0	0 + 0	0 + 54 + 66	0 + 0
M-JPEG	9 + 21	0 + 17	0 + 0	0 + 0	18 + 0 + 38	0 + 0

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

5   for (i = 0; i < Nrw; i++) {
      // 1D DWT in vertical direction with subsampling
      for (j = 0; j < 2*Ncl; j++) {
        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);
10
        tmp = (i==0) ? Hf[j] : oldHf[j];
        low_flt_vert(tmp, a[2*i][j], Hf[j], &oldHf[j], &Lf[j]);
      }

15    // 1D DWT in horizontal direction with subsampling ---------
      for (j = 0; j < Ncl; j++) {
        tmp = (j==Ncl-1) ? Lf[2*j] : Lf[2*j+2];
        HL[i][j] = high_flt_hor(Lf[2*j], Lf[2*j+1], tmp);

20      tmp = (j==0) ? HL[i][j] : HL[i][j-1];
        LL[i][j] = low_flt_hor(tmp, Lf[2*j],  HL[i][j]);
      }

      // 1D DWT in horizontal direction with subsampling ---------
25    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);

        tmp = (j == 0) ? HH[i][j] : HH[i][j-1];
30      LH[i][j] = low_flt_hor(tmp, Hf[2*j], HH[i][j]);
      }
    }
    // The Outputs  ------------------------------------------------
    for (i = 0; i < Nrw; i++)
35    for (j = 0; j < Ncl; j++) {
        Sink(LL[i][j]);
        Sink(HL[i][j]);
        Sink(LH[i][j]);
        Sink(HH[i][j]);
40    }
```

Figure 10: Source code of a discrete wavelet transform example.

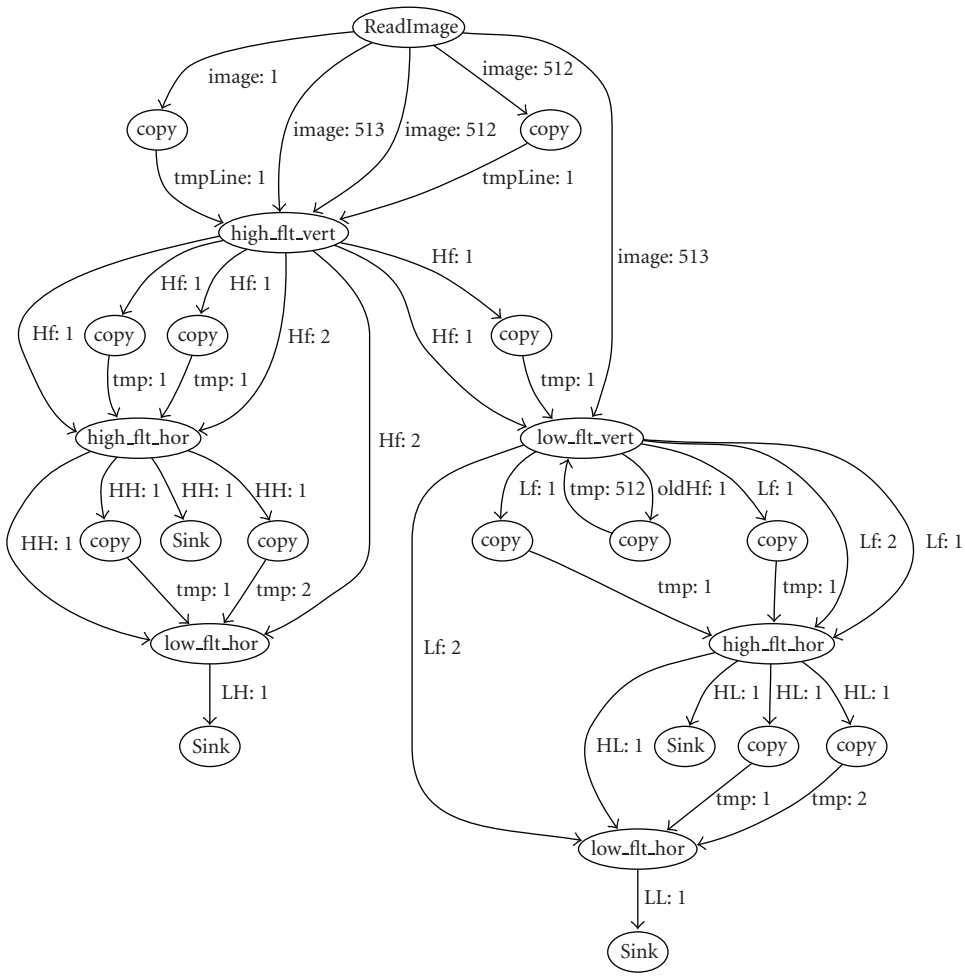

Figure 11: 2D-DWT process network with copy nodes.

self-loops+edges, or single-register+self-loops+edges for our FIFOs.

Note that our numbers on Compaan-like networks differ from those on Compaan networks reported in [17]. Due to a difference in internal representation, some of our channels are split into several Compaan-channels. In Compaan, these channels are recombined, with possibly further combinations, at a later stage. From the table, we can conclude that our techniques have split all OOM channels in examples LU-Factor and Gauss-Elim. into pairs of FIFOs. In general, we also have fewer channels between different nodes at the expense of more self-loops, which are a lot more efficient. For example, for SVD, the number of edges is reduced from 63 to 50, while for LU-Factor we have a reduction from 24 to 19 and for Faddeev from 24 to 20. Finally, we are able to identify (in examples LU-Factor, QR-Decomp, SVD, Faddeev, and M-JPEG) 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.

As to the time needed to derive the networks, Compaan itself takes 2.3 to 28.1 seconds on the examples in Table 1, while our tool takes 2.5 to 46.4 seconds. Most of the latter time is spent in the computation of the FIFO sizes, which Compaan does not compute.

8. 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 as 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. Our modified dataflow analysis leads 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 in nonparametric programs and approximately for other channels and self-loops in parametric programs.

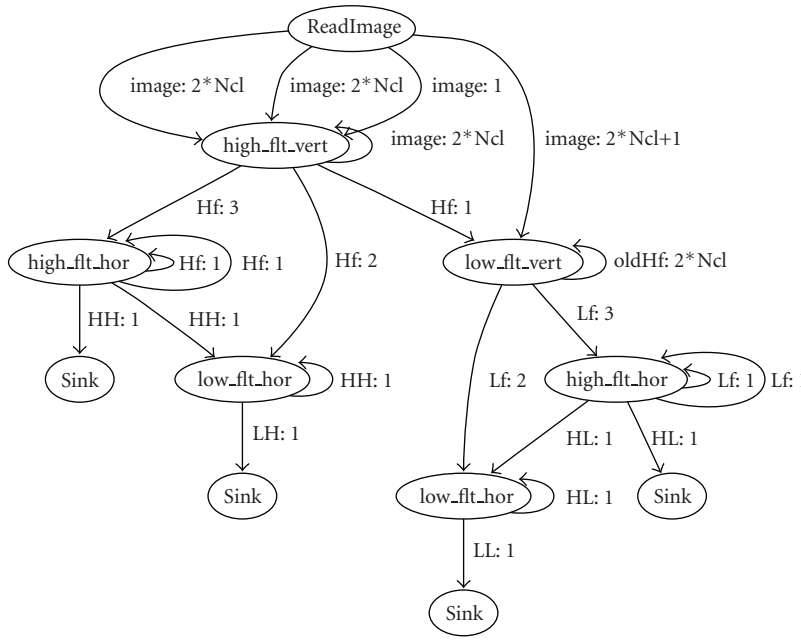

Figure 12: Optimized 2D-DWT process network.

ACKNOWLEDGMENTS

This research is partly supported by PROGRESS, the Embedded Systems and Software Research Program of the Dutch Technology Foundation STW—Project ARTEMISIA (LES.6389). We also thank Bart Kienhuis for his help and for the discussions on some of the topics in this paper.

REFERENCES

[1] A. Darte, R. Schreiber, and G. Villard, "Lattice-based memory allocation," *IEEE Transactions on Computers*, vol. 54, pp. 1242–1257, 2005.

[2] E. A. Lee and A. Sangiovanni-Vincentelli, "A framework for comparing models of computation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, no. 12, pp. 1217–1229, 1998.

[3] J. Davis, R. Galicia, M. Goel, et al., "PtolemyII: heterogeneous concurrent modeling and design in java," Tech. Rep. UCB/ERL M99/40, University of California, Berkeley, Calif, USA, 1999.

[4] G. Kahn, "The semantics of a simple language for parallel programming," in *Proceedings of the IFIP Congress*, pp. 471–475, North-Holland, Stockholm, Sweden, August 1974.

[5] E. A. de Kock, G. Essink, W. J. M. Smits, et al., "YAPI: application modeling for signal processing systems," in *Proceedings of the 37th Design Automation Conference (DAC '00)*, pp. 402–405, Los Angeles, Calif, USA, June 2000.

[6] E. A. de Kock, "Multiprocessor mapping of process networks: a JPEG decoding case study," in *Proceedings of the 15th International Symposium on System Synthesis (ISSS '02)*, pp. 68–73, Kyoto, Japan, October 2002.

[7] B. K. Dwivedi, A. Kumar, and M. Balakrishnan, "Automatic synthesis of system on chip multiprocessor architectures for process networks," in *Proceedings of the 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and Systems Synthesis, (CODES+ISSS '04)*, pp. 60–65, IEEE Computer Society, Stockholm, Sweden, September 2004.

[8] K. Goossens, J. Dielissen, J. van Meerbergen, et al., "Guaranteeing the quality of services in networks on chip," in *Networks on Chip*, pp. 61–82, Kluwer Academic Publishers, Hingham, Mass, USA, 2003.

[9] P. Lieverse, T. Stefanov, P. van der Wolf, and E. Deprettere, "System level design with SPADE: an M-JPEG case study," in *Proceedings of the International Conference on Computer-Aided Design (ICCAD '01)*, pp. 31–38, San Jose, Calif, USA, November 2001.

[10] A. Nieuwland, J. Kang, O. P. Gangwal, et al., *C-HEAP: A Heterogeneous Multi-Processor Architecture Template and Scalable and Flexible Protocol for the Design of Embedded Signal Processing Systems*, Kluwer Academic Publishers, Norwell, Mass, USA, 2002.

[11] H. Nikolov, T. Stefanov, and E. Deprettere, "Multi-processor system design with ESPAM," in *Proceedings of the 4th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '06)*, pp. 211–216, Seoul, Korea, October 2006.

[12] A. D. 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, pp. 99–112, 2006.

[13] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprettere, "System design using Kahn process networks: the Compaan/Laura approach," in *Proceedings of Conference Design, Automation and Test in Europe (DATE '04)*, vol. 1, pp. 340–345, Paris, France, February 2004.

[14] P. van der Wolf, P. Lieverse, M. Goel, D. La Hei, and K. Vissers, "MPEG-2 decoder case study as a driver for a system level

design methodology," in *Proceedings of the 7th International Workshop on Hardware/Software Codesign (CODES '99)*, pp. 33–37, Rome, Italy, May 1999.

[15] B. Kienhuis, E. Rijpkema, and E. Deprettere, "Compaan: deriving process networks from matlab for embedded signal processing architectures," in *Proceedings of the 8th International Workshop Hardware/Software Codesign (CODES '00)*, pp. 13–17, ACM Press, San Diego, Calif, USA, May 2000.

[16] 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.

[17] A. Turjan, B. Kienhuis, and E. Deprettere, "Translating affine nested-loop programs to process networks," in *Proceedings of International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '04)*, pp. 220–229, Washington, DC, USA, September 2004.

[18] A. Darte, R. Schreiber, and G. Villard, "Lattice-based memory allocation," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '03)*, pp. 298–308, ACM Press, San Jose, Calif, USA, October-November 2003.

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

[20] T. Vander Aa, M. Jayapala, F. Barat, H. Corporaal, F. Catthoor, and G. Deconinck, "A high-level memory energy estimator based on reuse distance," in *Proceedings of the 3rd Workshop on Optimizations for DSP and Embedded Systems (ODES '05)*, San Jose, Calif, USA, March 2005.

[21] P. Vanbroekhoven, G. Janssens, M. Bruynooghe, H. Corporaal, and F. Catthoor, "Advanced copy propagation for arrays," in *Proceedings of the ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '03)*, U. Kremer, Ed., pp. 24–33, ACM Press, San Diego, Calif, USA, June 2003.

[22] P. Feautrier, "Automatic parallelization in the polytope model," in *The Data Parallel Programming Model*, vol. 1132 of *Lecture Notes in Computer Science*, pp. 79–103, Springer, London, UK, 1996.

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

[24] 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 the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '04)*, pp. 248–258, Washington, DC, USA, September 2004.

[25] P. Clauss, F. J. Fernández, D. Gabervetsky, and S. Verdoolaege, "Symbolic polynomial maximization over convex sets and its application to memory requirement estimation," ICPS Research Reports 06–04, Université Louis Pasteur, Strasbourg, France, 2006, http://icps.u-strasbg.fr/upload/icps-2006-173.pdf.

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

[27] C. Bastoul, "Code generation in the polyhedral model is easier than you think," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT '04)*, pp. 7–16, IEEE Computer Society, Antibes Juanles-Pins, France, September 2004.

[28] S. Verdoolaege, M. Bruynooghe, G. Janssens, and F. Catthoor, "Multi-dimensional incremental loop fusion for data locality," in *Proceedings of the 14th IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP '03)*, D. Martin, Ed., pp. 17–27, The Hague, The Netherlands, June 2003.

[29] 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 *Proceedings of the 1st Workshop on Optimization for DSP and Embedded Systems (ODES '03)*, San Francisco, Calif, USA, March 2003.

[30] I. Daubechies and W. Sweldens, "Factoring wavelet transforms into lifting steps," *Journal of Fourier Analysis and Applications*, vol. 4, no. 3, pp. 247–269, 1998.