Reducing Memory Requirements of Stream Programs by Graph Transformations

Pablo de Oliveira Castro, Stéphane Louise
CEA, LIST, Embedded Real Time Systems Laboratory, F-91191 Gif-sur-Yvette, France;
pablo.de-oliveira-castro@cea.fr
stephane.louise@cea.fr

Denis Barthou
University of Bordeaux - Labri / INRIA
351, cours de la Libération, Talence, F-33405 France;
denis.barthou@inria.fr

ABSTRACT

Stream languages explicitly describe fork-join parallelism and pipelines, offering a powerful programming model for many-core Multi-Processor Systems on Chip (MPSoC). In an embedded resource-constrained system, adapting stream programs to fit memory requirements is particularly important. In this paper we present a new approach to reduce the memory footprint required to run stream programs on MPSoC. Through an exploration of equivalent program variants, the method selects parallel code minimizing memory consumption. For large program instances, a heuristic accelerating the exploration phase is proposed and evaluated. We demonstrate the interest of our method on a panel of ten significant benchmarks. Using a multi-core modulo scheduling technique, our approach lowers considerably the minimal amount of memory required to run seven of these benchmarks while preserving throughput.

KEYWORDS: Stream Languages, Data Flow, Memory, Graph Transformations.

1. INTRODUCTION

The recent low-consumption Multi-Processors Systems on Chip (MPSoC) enable new computation-intensive embedded applications. Yet the development of these applications is impeded by the difficulty of efficiently programming parallel applications. To alleviate this difficulty, two issues have to be tackled: how to express concurrency and parallelism adapted to the architecture and how to adapt parallelism to constraints such as buffer sizes and throughput requirements.

Stream programming languages [10][15][11] are particularly adapted to describe parallel programs. Fork-join parallelism and pipelines are explicitly described by the stream graph, and the underlying data-flow formalism enables powerful optimization strategies. As an example, the StreamIt [10] framework defines transformations of dataflow programs guided by greedy heuristics and enhancing parallelism through fission/fusion operations. However, to our knowledge, there is no method that explores the design space based on the different expressions of parallelism and communication patterns. The main difficulty comes from the definition of a space of semantically equivalent parallel programs and from the computational complexity of such exploration.

In this paper we propose a design space exploration technique that generates, from an initial program, a set of semantically equivalent parallel programs, but with different buffer, communication and throughput costs. Using an appropriate metric, we are able to select among all the variants, the one that best fits our requirements. We believe that our approach is applicable to memory, communication and throughput requirements, yet this paper focuses only on *memory requirements*.

The buffer requirements of stream programs depend not only on the rates of actors but also on the chosen mapping and scheduling. We illustrate the memory reduction achieved by our technique using the modulo scheduling execution model. Memory reduction on modulo scheduling has already been studied by Choi et al[6]. They propose an integer linear programming (ILP) approach to tackle the *Memory-constrained Processor Assignment for minimizing Initialization Interval* (MPA-ii) problem and measure the minimal memory requirements that their solution require. By exploring the memory design space of the input programs before mapping them with Choi *et al.* approach, we further reduce the memory requirements (sometimes as much as 80%).

Section 2 introduces the stream formalism used. The transformations producing semantically equivalent program variants are described in Section 3. Section 4 presents the exploration algorithm built upon these transformations and the algorithm termination proof. Section 5 reviews the *MPA-ii* problem that we use to illustrate the benefits of our exploration method and the metric used during our exploration. In Section 6 we describe our experimental methodology and results. Finally, Section 7 presents related works.

2. FORMALISM

Our formalism, very close to StreamIt[10], describes a program using a cyclo-static data flow (CSDF) graph[4] where nodes are actors that are fired periodically and edges represent communication channels. Consider the example of stream graph in fig. 1. Different types of actors are considered.

Source I and **Sink O** actor nodes model respectively a programs input and output. The source produces a stream of inputs elements, whereas the sink consumes all the elements it receives. If a source produces always the same element it is a **constant source C**. A sink whose elements are never observed is called a **trash sink T**.

Functions in the imperative programming paradigm are replaced by **Filter** actors $\mathbf{F}(\mathbf{c_1}, \mathbf{p_1})$. Each filter has one input and one output, and an associated pure (without internal state) function f. Each time there are at least c_1 elements on the input, the filter is fired: the function f consumes the c_1 input elements and produces p_1 elements on the output.

Then there are nodes that dispatch and combine streams of data from multiple filters, routing data streams through the program and reorganizing the order of elements within a stream. **Join round-robin** $\mathbf{J}(\mathbf{c_1} \dots \mathbf{c_n})$ gathers the elements received on its n inputs and writes them on its output. In its k^{th} firing the node consumes $c_u \in \mathbb{N}^\star$ elements on its u^{th} input, with $u = (k \mod n) + 1$ and writes them on its output. When the node has consumed elements on all its inputs, it has completed a *cycle*. **Split round-robin** $\mathbf{S}(\mathbf{p_1} \dots \mathbf{p_m})$ dispatches the elements it receives on its input among its m outputs. In its k^{th} firing the node takes $p_v \in \mathbb{N}^\star$ elements on its input, with $v = (k \mod m) + 1$ and writes them to the v^{th} output. **Duplicate** $\mathbf{D}(\mathbf{m})$ has one input and m outputs. Each time this node is fired, it takes one element on the input and writes it to every output, duplicating its input m times.

2.1. Schedulability

We can schedule a CSDF graph in bounded memory if it has no deadlocks (it is live) and if it admits a steady-state schedule (it is consistent) [4][14]. A graph deadlocks if the number of elements received by any sink remains bounded

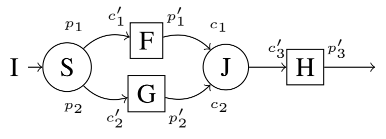


Figure 1. A Simple Stream Graph: the split node S consumes alternatively p_1 and p_2 elements from I. These are passed to filters F and G (that may be scheduled concurrently) and their outputs are combined by the join node J and passed to H.

no matter the number of elements streamed through the sources.

A graph is consistent if it admits a steady-state schedule, that is to say a schedule during which the amount of buffered elements between any two nodes remains bounded. A consistent CSDF graph G with N_G nodes admits a positive repetition vector $\vec{q_G} = [q_1, q_2, \dots, q_{N_G}]$ where q_i represents the number of cycles executed by node i in its steady-state schedule. Given two nodes u, v connected by edge e. We note $\beta(e)$ the elements exchanged on edge e during a steady-state schedule execution. If node u produces $prod(e)$ elements per cycle on edge e and node v consumes $cons(e)$ elements per cycle on edge e,

$$\beta(e) = prod(e) \times q_u = cons(e) \times q_v \qquad (1)$$

2.2. Executing a CSDF on a Multi-Core System

There are different approaches to map and execute a CSDF graph [10][14]. In this paper we use the Stream Graph Modulo Scheduling (SGMS) approach for multi-core systems proposed in [13]. SGMS is similar to classical modulo scheduling for instructions, each node u is assigned a single processing element (PE) and is scheduled on a particular activation stage $stage_u$. Stages are activated gradually, forming a software pipeline that executes the different nodes in the stream concurrently.

The first phase of SGMS is the PE assignment. By solving an ILP problem, we assign to each PE a group of nodes, so that every node is assigned to a exactly one PE and the computing load of the nodes is balanced among the PEs.

The second phase of SGMS is the stage assignment. Its role is to select an efficient temporal schedule for the software pipeline. This is achieved by enforcing two simple rules:

- Preservation of data dependences: The stage number of a producing actor must be greater or equal than the stage number of the consuming actor.
- Overlapping transfer latencies and computation time: Two actors u, v that are mapped to different PEs and

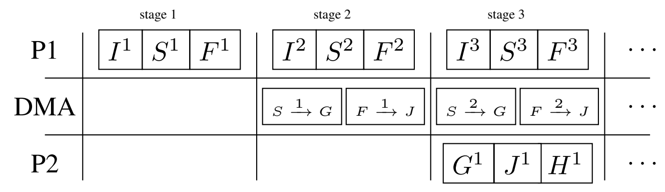


Figure 2. SGMS Software Pipeline on 2 PEs with a DMA for the Graph in fig. 1: the indexes indicate the current node appearance. An appearance of node N correspond to q_N cycles, that is to say an execution in the steady-state schedule.

connected by e need a DMA operation or a network operation to transfer the data from one PE to the other. In this case we schedule the data transfer of e in $stage_e$ and we enforce that $stage_u < stage_e < stage_v$. By scheduling the data transfer on a different stage than the consumer and producer on the software pipeline, we ensure that data transfer latencies and computation time are efficiently overlapped.

In fig. 2, we show one possible SGMS schedule for the program in fig. 1. Nodes I, S, F are assigned to processing element $P1$ and nodes G, J, H are assigned to processing element $P2$.

3. SEARCH SPACE

In this section we present the transformations that are used to generate variants preserving the semantics of the input program.

3.1. Graph Transformation Framework

The graph transformation framework presented here follows the formulation given in [3]. A transformation T applied on a graph G, generating a graph G' is denoted $G \xrightarrow{T} G'$. It is defined by a matching subgraph $L \subseteq G$, a replacement graph $R \subseteq G'$ and a glue relation g that binds the frontier edges of L (edges between nodes in L and nodes in $G \backslash L$) to the frontier edges of R.

The graph transformation works by deleting the match subgraph L from G thus obtaining $G \backslash L$. After this deletion, all the edges between $G \backslash L$ and L are left dangling. We then use the glue relation to paste R into $G \backslash L$ obtaining G'. We will describe transformations by giving the match and replacement subgraphs as in fig. 3. The input and output edges will be denoted i_x and o_y in both graphs. The natural correspondence between the i_x (resp. o_y) of L and R gives the glue relation.

Definition 1 *A derivation of a graph G_0 is a chain of transformations that can be successively applied to G_0: $G_0 \xrightarrow{T_0} G_1 \ldots \xrightarrow{T_n} \ldots$.*

Definition 2 *A transformation T preserves the semantics of G if T preserves consistency, liveness and if for the same inputs, the graph generated by T, G', produces at* least *the same outputs as G.*

The first two properties ensure that the transformation preserves the schedulability of the graph (cf. sec. 2). The last property ensures that the transformation does not change the observable outputs of the program. Because some of our transformations "relax" the consumption or production rates of split and join nodes, a variant may produce more values on the outputs than the original graph. If this was not allowed we would lose many desirable transformations. Besides, they still preserve the semantics since the extra values can be safely ignored by redirecting them to a fictitious Trash node (the number of extra values is known at compile time).

A formal definition of the semantic preservation of CSDF transformations is found in [7], also an illustration of the interest of allowing extra values is provided.

3.2. Simplifying Transformations

These transformations, remove nodes with either no effect or non-observed computations, and separate split-join structures into independent ones.

Dead code elimination(fig. 3(f)) is a dead-code elimination for stream graphs. A node for which all outputs go to a Trash can be replaced by a Trash itself without affecting the semantics. This transformation progressively removes nodes whose outputs are never observed.

Constant propagation(fig. 3(i)) when a constant source is split we can eliminate the split duplicating the constant source.

RemoveJS / RemoveSJ / RemoveD (not shown in the figure) are very simple transformations which remove nodes whose composed effect is the identity : a split and a join of identical consumption and productions, a single branch dup, a single branch split, etc.

CompactSS/CompactDD/CompactJJ (fig. 3(e)) fuse together a hierarchy of Splits, Joins or Duplicate nodes. CompactDD is always possible, we can replace any tree of Duplicate nodes by a single Duplicate node which copies its input stream to every output edge of the original tree. CompactSS is possible when when the lower split productions sum is equal to the production on the edge connecting the lower and upper splits. CompactJJ is possible when when the upper join consumptions sum is equal to the consumption on the edge connecting the lower and upper joins.

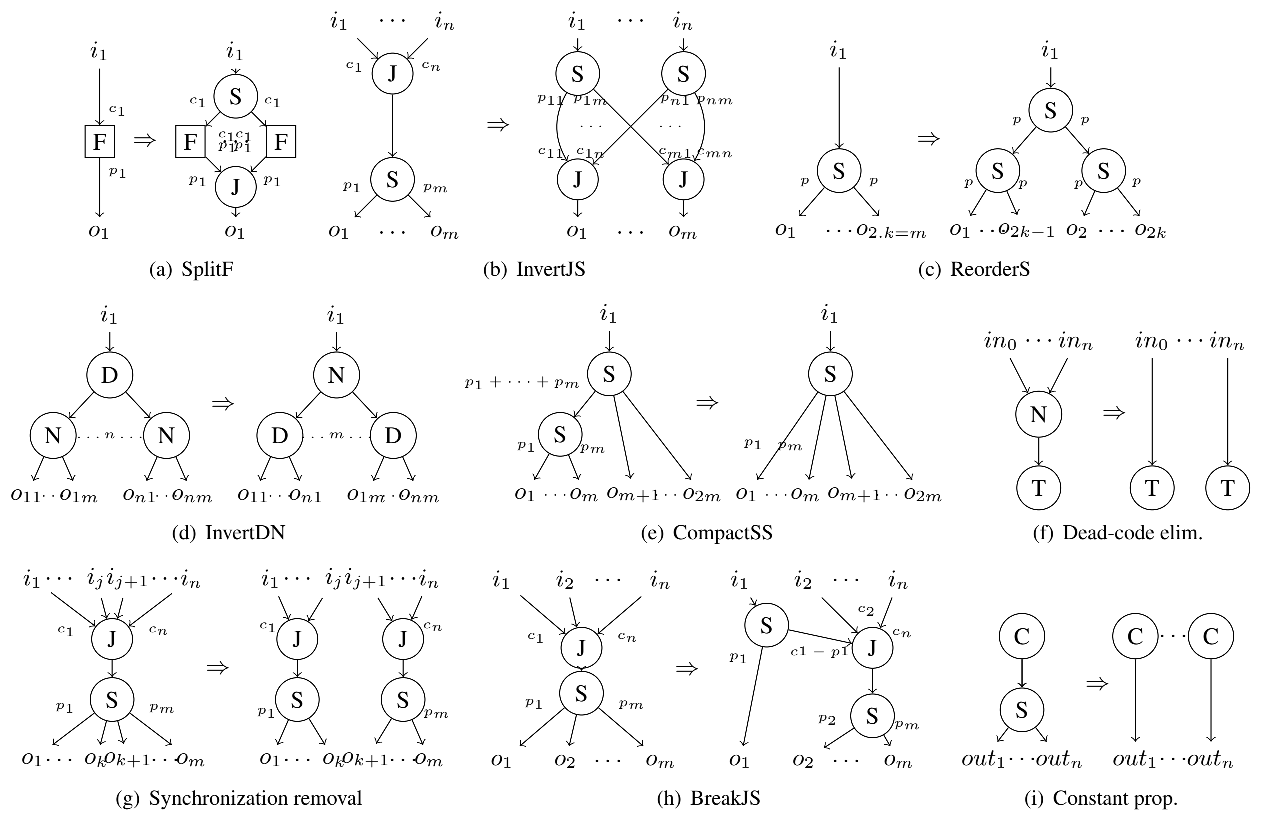

(a) SplitF (b) InvertJS (c) ReorderS (d) InvertDN (e) CompactSS (f) Dead-code elim. (g) Synchronization removal (h) BreakJS (i) Constant prop.

Figure 3. Set of Transformations Considered: each transformation is defined by a graph rewriting rule. Node N is a wildcard for any arity compatible node.

Synchronization Removal(fig. 3(g)) is possible when the sum of the join consumptions is equal to the sum of the split productions: $\sum_x c_x = \sum_y p_y$. The idea behind this transformation is to find a partition of the join consumptions and a partition of the split production so that their sum is equal: $\sum_{x \leq j} c_x = \sum_{y \leq k} p_k$. In that case we can split the Join-Split in two smaller and independant Join-Split.

BreakJS(fig. 3(h)) is triggered when $\sum_x c_x = \sum_y p_y$ but no partition of the productions/consumptions can be found. In that case the transformation breaks the split or join edge with the largest consumptions. BreakJS breaks Join-Split junctions into smaller constituents, often triggering Synchronization Removal separating the Join-Split junction into two smaller junctions.

3.3. Restructuring Transformations

These transformations restructure communication patterns, alone they do not reduce the memory requirements, but they can rewrite the graph and trigger some of the previous transformations.

SplitF(fig. 3(a)) This transformation splits a filter on its input. SplitF introduces split-join parallelism in the programs. Because filters are pure: we can compute each input block on a different filter concurrently. The degree of concurrency introduced by SplitF is parametric. Multiple splitting of the same filter are useless, since they can be achieved with a single SplitF of greater arity.

InvertJS(fig. 3(b)) This transformation inverts join and split nodes. To achieve this it creates as many split nodes as inputs and as many join nodes as outputs. It then connects together the new split and join stages as shown in the figure. Intuitively, this transformation works by "unrolling" the cycle in the original nodes (examining consecutive executions of the nodes) so that a common period between the join and split stage emerges. The transformation make explicit the communication pattern in this common period, by inverting the split and join stages.

The transformation is admissible in two cases:
1- Each p_j is a multiple of $C = \sum_i c_i$, the transformation is admissible choosing $p_{ij} = c_i.p_j/C$, $c_{ji} = c_i$.
2- Each c_i is a multiple of $P = \sum_j p_j$, the transformation is admissible choosing $p_{ij} = p_j$, $c_{ji} = p_j.c_i/P$.

ReorderS/ReorderJ(fig. 3(c)) create a hierarchy of split (resp. join) nodes. In the following we will only discuss ReorderS. The transformation is parametric in the split arity f. This arity must divide the number of outputs, $m = k.f$. In the figure, we have chosen $f = 2$. We forbid the trivial cases ($k = 1$ or $f = 1$) where the transformation is the identity. As shown in fig. 3(c), the transformation works by rewriting the original split using two separate stages: odd and even elements are separated then odd (resp. even) elements are redirected to the correct outputs. ReorderS and ReorderJ sometimes uncover possible simplifying transformations by explicitly separating elements by their f congruency (eg. even and odd elements when $f = 2$),

InvertDN(fig. 3(d)) This transformation inverts a duplicate node and its children, if they are identical. Since all nodes are pure, their outputs depend only on their inputs; thus applying a process k times to the same input is equivalent to applying the process once making k copies of the output. This transformation eliminates redundant computations in a program.

4. EXPLORATION ALGORITHM

The exploration is an exhaustive search of all the derivations of the input graph using our transformations. We use a branching algorithm with backtracking, as described recursively in algorithm 1. The exploration is particularly memory efficient because it never copies the graph when branching; transformations are in-place applied and in-place reverted when backtracking.

We prove that this algorithm terminates by showing for any initial graph, that once a large enough recursion depth is reached, no transformations will satisfy the **for all** condition in statement 6. This follows from the fact that the transformations considered cannot produce infinite derivations. We prove this result formally in [7], yet a general idea of the proof follows. We start by constructing a function on graphs $\tau : (Graphs) \mapsto (\mathcal{M}(\mathbb{N}), \succ)$, where $\mathcal{M}$ is the set of all finite multisets of $\mathbb{N}$ and $\succ$ its natural well-founded order[8]. We then show that for any of the transformations in sec. 3, $\tau(G) \succ \tau(G')$. Using the well-foundness of $\succ$ we prove that all the derivations are finite.

Algorithm 1 EXPLORE(G)

```
 1: Simpl ← {Dead Code, CompactSS, CompactJJ, ...}
 2: Restr ← {SplitF, InvertJS, ReorderS, ...}
 3: while $\exists N \in G \quad \exists S \in$ Simpl : $S$ applicable at $N$ do
 4:     $G \leftarrow S_N(G)$
 5: end while
 6: for all $N \in G$ and $T \in$ Restr : $T$ applicable at $N$ do
 7:     $G \leftarrow T_N(G)$
 8:     EXPLORE($G$)
 9:     $G \leftarrow$ RESTORE($G$)
10: end for
```

4.1. Dominance Rules

Generated graphs G are often obtained through application of transformations operating on distinct part of the graph. In that case, applications of these transformations commute, meaning that the order in which they are applied does not change the resulting graph. Therefore only one of the possible permutations of the application order is explored.

To select only one of the possible permutations, an arbitrary ordering ($\sqsubset$) is defined on the transformations. We say that T_1 dominates T_2 if $T_1 \sqsubset T_2$. A new transformation is applied only if it dominates all the earlier transformations that commute with it.

A sufficient condition for an early transformation T_1 to commute with a later transformation T_2 is that the nodes that are matched by L_2 were already present before applying T_1. This condition can be easily checked if nodes are tagged with a creation date during exploration.

4.2. Partitioning Heuristic

Exhaustive search may take too much time on large graph instances. The optimization presented in the previous section alleviates this complexity issue; however for large graphs like Bitonic (370 nodes) or DES (423 nodes) in section 6, the exploration still takes too much time. To bound exploration time, we have implemented an heuristic combining our exploration with a graph partitioning of the initial program.

For this to work, the metrics we optimize must be composable: there must exist a function f that verifies for every partition G_1, G_2 of G, $metric(G) = f(metric(G_1), metric(G_2))$.

We show that reducing the initial program size, reduces the exploration maximal size. Our heuristic, based on this result works by partitioning the initial program in partitions of at most $PSIZE$ nodes. If the metric we try to optimize is composable, exploring each partition separately, selecting the variant which optimizes f for each partition, and assembling them back together leads to valid results. To partition the graphs we use the freely available Metis [12] graph partitioning tool.

The downside of this heuristic is that we loose the transformations involving nodes of different partitions. Yet our benchmarks in section 6 show that the degradation of the final solution is acceptable (except for very small partitions).

5. MEMORY REQUIREMENTS IN SGMS

To evaluate our design-space exploration method, we apply it to memory reduction on a SGMS (cf. sec. 2) execution model.

The authors of [6] have studied the problem of reducing memory requirements in stream modulo scheduling. They introduce the problem *MPA-ii*, Memory-constrained Processor Assignment for minimizing Initiation Interval, and propose an ILP based solution; then they measure the minimal memory requirements achieved by their assignment. In the following we will show that if our exploration method is used we can significantly decrease these requirements.

To compute the buffer sizes required by two nodes u and v communicating through an edge e, the authors in [6] consider two kinds of situations:

- u and v are on the same PE, the output buffer of u can be reused by v.
- u and v are on different PE, buffers may not be shared, since a DMA operation d is needed to transfer the data. The stage in which d is scheduled is noted $stage_d$.

The number of elements produced by u and consumed by v on edge e during a steady-state execution is called $\beta(e)$ and given by eq. (1). Furthermore, since multiple executions of the streams may be on flight in the pipeline at the same time, buffers must be duplicated to preserve data coherency. The number of buffer copies needed is the number of stages between the two nodes, $\delta(u,v) = stage_v - stage_u + 1$.

The output buffer requirements for all the nodes but Duplicate are,

$$out\_bufs(i) = \sum_{\forall e \in \text{outputs}(i)} \delta(u,v) * \beta(e) \qquad (2)$$

For Duplicate nodes, since the output is the same on all the edges, buffers can be shared by recipients, so the formula becomes : $\max_{\forall e \in \text{outputs}(i)} \delta(u,v) * \beta(e)$.

The input buffer requirements are,

$$in\_bufs(i) = \sum_{\forall e \in \text{dma\_inputs}(i)} \delta(v,d) * \beta(e) \qquad (3)$$

Combining (2)(3) we obtain the total memory requirements for a node $b(i) = out\_bufs(i) + in\_bufs(i)$.

5.1. Metrics

In this section we propose a metric that selects candidates which improve the memory required for the modulo scheduling. From equations (2)(3) we observe that to reduce $b(i)$ we can either reduce the stage difference between producers and consumers, $\delta(u,v)$, or we can reduce the elements exchanged during a steady-state execution, $\beta(e)$.

The authors of [6] already optimize the $\delta(u,v)$ when possible. We attempt to reduce $b(i)$ by reducing $\beta(e)$. To reduce the influence of the $\beta(e)$ in the buffer cost we search the candidate that minimizes, $maxbuf(G) = \max_{\forall e \in G} \beta(e)$. if we have a tie between multiple candidates, we chose the one that minimizes, $totbuf(G) = \sum_{\forall e \in G} \beta(e)$.

We note that these metrics are composable, as required by the partitioning heuristic described in section 4. Indeed given a partition G_1, G_2 of G, we have $maxbuf(G) = \max(maxbuf(G_1), maxbuf(G_2))$ and $totbuf(G) = totbuf(G_1) + totbuf(G_2)$.

6. RESULTS

We consider a representative set of programs from the StreamIt benchmarks [1]: Matrix Multiplication, Bitonic sort, FFT, DCT, FMradio, Channel Vocoder. We also consider three benchmarks of our own: Hough filter and fine grained Matrix Multiplication (which both contain cycles in the stream graph), and Sobel filter.

We first compute for each benchmark the minimal memory requirements per core for which an *MPA-ii* mapping is possible. We will use this first measure, MPAii_mem, as our baseline. We then use our exploration method on each benchmark, and select the best candidate according to the metrics described in section 5. We compute the *MPA-ii* minimal memory per core requirements for the best candidate: Exploration_MPAii_mem.

Finally we compute the memory requirements reduction using the following formula:

$$\frac{(\text{MPAii\_mem} - \text{Exploration\_MPAii\_mem})}{\text{MPAii\_mem}}$$

As shown in [13] it should be possible to integrate filter splitting in Choi *et al.* approach. Thus to make the comparison fair, we have not taken into account any further memory reduction achieved by SplitF transformation.

6.1. Memory Reduction

The exhaustive search is used for all the benchmarks except DES and Bitonic for which we used the partitioning heuristic, the partition maximum size (PSIZE) was set to 60 which was empirically determined as a good compromise between speed and quality of the solution. A significant memory reduction is achieved in seven out of the ten benchmarks. This means that in these seven cases the application can be executed with significantly less memory than using the approach in [6]: either a larger application can be mapped to the same architecture or an architecture with smaller memory requirements can be designed for this application. In terms of throughput, our method does not degrade the performance since we obtain similar II (initialization interval) values than bare *MPA-ii*.

The experimental results obtained are summarized in fig. 4. We can distinguish three categories among the benchmarks:

No effect (DCT, FM, Channel), in this first group of benchmarks, we do not observe any improvement. Indeed these programs make little use of split, join and duplicate nodes. They are almost filter only programs. Because our transformations operate essentially on data reorganisation nodes, they have no effect on these examples. The very small improvement in Channel is due to a single SimplifyDD that compacts a serie of duplicate nodes.

Loop splitting (MM_Fine, Hough), these two benchmarks contain cycles. Using our set of transformations we are able to split the cycles, as we demonstrate in fig. 5 for the Hough benchmark. This is particularly interesting in the context of modulo scheduling where cycles must be fused [13]. In the Hough selected variant, the cycle is broken in three smaller cycles: Thus after fusion, it has more fine-grained filters, and achieves smaller memory requirements.

Dependencies breaking and communication reorganisation (MM_coarse, MM_fine, FFT, Bitonic, Sobel, DES), the other benchmarks show varying memory reductions, resulting from a better expression of communication patterns. Either non needed dependencies between nodes have been exposed, allowing for a better balance among the cores, or groups of split, join and dup have been simplified in a pattern that is less memory consumming.

6.2. Variation when Changing the Number of PE

In this section we study the memory reduction variance depending on the number of cores in the target architecture. We can identify two categories:

Plateau, graphs in this category, Bitonic, FFT, DES, show little change over the number of PEs. We observe in table 1 that for these graphs, $maxbuf$ remains at the same level but $totbuf$ has decreased.

Table 1. Metrics Variations between the Original Graph and the Best Selected Candidate (in number of elements).

Benchmark	*maxbuf variation*	*totbuf variation*
MM_fine	-1500	-630
MM_coarse	-1520	-209
Bitonic	0	-480
DES	0	-3514
FFT	0	-184
DCT	0	0
FM	-6	+6
Channel	0	-17
Sobel	-3392	-3841
Hough	-28000	-29933

Table 2. Measured Exploration Costs.

Benchmark	*Number of nodes*	*Exhaustive search time*	*Search time after partitioning (PSIZE = 60)*
MM_fine	17	2s	-
MM_coarse	14	<1s	-
Bitonic	370	>6hours	111s
DES	423	5hours	3.5s
FFT	106	33s	32.9s
DCT	40	<1s	-
FM	43	<1s	-
Channel	57	<1s	-
Sobel	32	9min18s	-
Hough	16	3min75s	-

Increasing, graphs in this category, MM_Fine, MM_Coarse, Sobel, Hough, show a better memory improvement the more PEs are used. These graphs increase both $totbuf$ and $maxbuf$.

Graph in category *Increasing* break the biggest memory consumer nodes into less greedy nodes. Thus when increasing the number of PEs, the mapping algorithm is able to balance memory requirements between PEs by spliting the biggest consumer(s) among multiple processors. This explains that these benches show better results as we increase the PE number. Once we add enough processors to distribute all the new nodes created by the splits of the biggest consumers, memory usage no longer improves (we can observe this effect on Hough and MM_Fine). In the other hand, the reduction in graphs of category *Plateau*, is more uniform, and thus the gain does not depend on the number of PE.

6.3. Exploration Cost and Heuristic Evaluation

We measure the exploration time (table 2) of our method for each of the benchmarks. The measures were taken using a Intel Pentium 4 3.8GHz computer with 4GB of RAM. The exploration algorithms are written in Python and executed using Python 2.5.4 interpreter running on Linux.

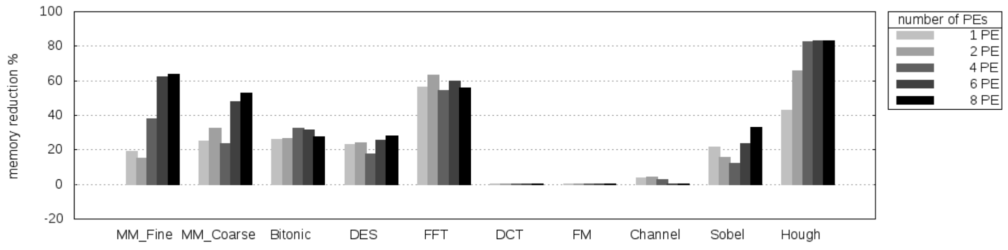
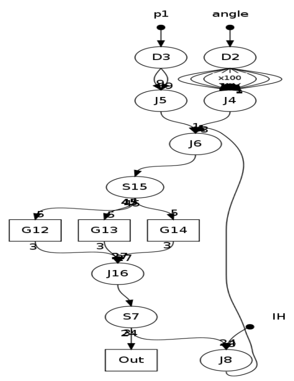
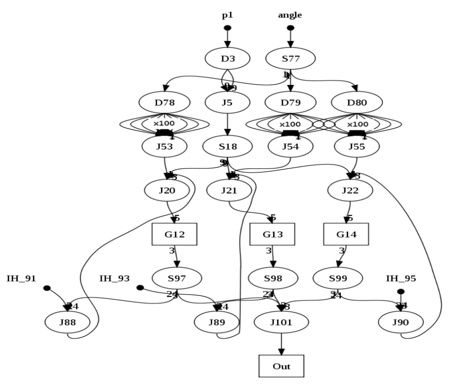


Figure 4. **Minimal Required Memory Reductions Achieved by our Proposed Method.**



(a) Original candidate after a SplitF



(b) Best candidate after transformations

1. InvertJS on J6 and S15
2. InvertJS on J16 and S7
3. InvertJS on J8 and S19
4. Constant Propagation on IH
5. RemoveJS on J26
6. InvertJS on J4 and S17
7. InvertDN on D2

(c) Best candidate derivation

Figure 5. **Using our Transformations to Split Cycles on the Hough Benchmark.**

If we do an exhaustive search, exploration times are fast (under 1min) for six of the benchmarks. Sobel and Hough, having multiple high-arity nodes, show moderate exploration times. Finally, without the partitioning heuristic, exploration for the largest benchmarks in our suite (DES and Bitonic) is too long: 5 hours for DES; more than 6 hours for Bitonic, we stopped the exploration after that point. Using the heuristic, with PSIZE=60, the exploration of DES and Bitonic is very fast (under 2min). The partitioning heuristic effectively reduces the running time of these very large graphs. The heuristic only marginally reduces the running time of FFT, because the exploration complexity is unbalanced in the original graph. Most of the transformations are found in the bottom half of the stream graph. Since the partitionning method separates the graph upper and lower part, exploring the upper part is cheap and the lower part is almost as costly as the original graph.

We evaluate the effect of the heuristic (fig. 6) on the solution quality by changing the parameter PSIZE. For each bench we choose a set of PSIZE values that separate the graph in 2,3,4 and 6 subgraphs. We have used smaller versions of DES (with 8 mixing stages instead of 16) and Bitonic (with 8 bit strings instead of 32) to evaluate the heuristic since using the original version was impractical because of the very long search time. The solutions for Bitonic8, Des8 and FFT are quite good even with 6 partitions, they remain close to the exhaustive solution. Solutions for MM_coarse, MM_fine, Hough and Sobel, on the other hand, quickly degrade as the number of partitions increases; for more than two partitions, there is no improvement over the original program. This result was expected and is explained by the small size of these benchmarks (less than 20 nodes). Indeed in this case, the partitioning process leaves very small partitions where no transformations can be applied. This is not a problem, since these small instances can be handled efficiently with the exhaustive approach.

7. RELATED WORKS

The authors of [13] were the first to apply Modulo Scheduling to stream graphs, evaluating their technique on the Cell BE. In [6] their work was extended considering an embedded target with memory and number of PE constraints, *MPA-ii* is introduced.

StreamIt is both a language[18], and an optimizing compiler[10]. As in our approach, StreamIt adapts the granularity and communications patterns of programs through graph transformation, which it separates in three classes: fusion transformations cluster adjacent filters,

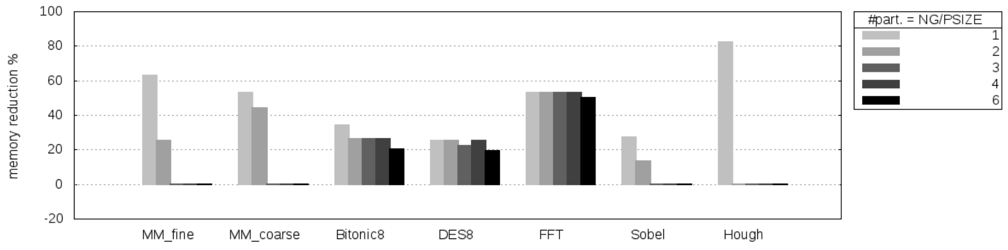


Figure 6. Evaluation of the partitioning heuristic on *8 PE*: for each benchmark, we show the memory reduction achieved when changing $\frac{N_G}{PSIZE}$ (the number of partitions used), exhaustive search is conducted independently in each partition.

coarsening their granularity; fission transformations parallelize stateless filters decreasing their granularity; reordering transformations operate on splits and joins to facilitate fission and fusion transformations.

We implement Fission transformations on stateless filters using the SplitF transformation. In our approach, Fusion transformations should be delegated to a later clustering phase, since early coarsening would reduce the number of explored variants. Instead, we have concentrated on proposing new Reordering transformations that explore alternate communication patterns. We implement StreamIt filter hoisting on duplicate nodes with InvertDN. We propose, through RemoveJS and RemoveSJ, StreamIt synchronization removal that eliminates neighbor split-joins with matching weights. We go further tackling split-join junctions of different weights with InvertJS and BreakJS, eliminating dead-code and removing unnecessary synchronization on constant sources. To the best of our knowledge StreamIt does not consider reordering transformations on feedback loops. The dead-code elimination transformation was already described in [16].

In the multi-dimensional dataflow model for signal processing Array-OL, the elements that are consumed by a task are defined by successively translating a selection pattern over a multidimensional array. In [2] the authors show that in some cases it is possible to share the patterns for two successive tasks. We achieve a similar effect using InvertDN, enabling a hoist of a common communication pattern over a duplicate node.

Dataflow transformations approaches have been proposed to optimize circuit design [5][19], but they usually target a much smaller granularity of nodes than our filters.

Many methods for optimizing CSDF multiprocessor schedules have already been proposed, some concentrate on maximizing the throughput [4][9], other on minimizing the memory requirements [20] and some search best pareto configurations for both criteria [17]. Our approach does not act at the schedule level but at the implementation level (changing structure of the dataflow graph), it is therefore complementary to these approaches focusing on schedule optimization (timing the dataflow graph).

8. CONCLUSION

We propose a new design-space exploration to reduce memory requirements of stream programs under modulo scheduling. Memory reduction is achieved through successive semantically preserving transformations. The transformations change the structure of the stream program, breaking cycles, dependencies and simplifying communication patterns.

We propose an efficient heuristic to explore the different transformations combinations based on graph partitioning. We demonstrate the interest of the approach on a significant panel of benchmarks, showing large memory gains while preserving throughput. We think that this approach is flexible and could also be used with a different metric to adapt stream programs to other constraints as communication bus capacities, maximum latencies, memory hierarchies.

REFERENCES

[1] Streamit benchmarks. `http://groups.csail.mit.edu/cag/streamit/shtml/benchmarks.shtml`.

[2] A. Amar, P. Boulet, and P. Dumont. “Projection of the Array-OL Specification Language onto the Kahn Process Network Computation Model”. In *Int. Symp. on Parallel Architectures, Algorithms and Networks*, 2005.

[3] M. Andries, G. Engels, A. Habel, B. Hoffmann, H.-J. Kreowski, S. Kuske, D. Plump, A. Schürr, and G. Taentzer. “Graph Transformation for Specification and Programming”. *Sci. Comput. Program.*, 34(1):1–54, 1999.

[4] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. “Cycle-static Dataflow”. In *IEEE Trans. on Signal Processing*, 1996.

[5] A. Chandrakasan, M. Potkonjak, J. Rabaey, and R. Brodersen. “An Approach For Power Minimization Using Transformations”. In *VLSI Signal Processing*, pages 41–50, 1992.

[6] Y. Choi, Y. Lin, N. Chong, S. Mahlke, and T. Mudge. "Stream Compilation for Real-Time Embedded Multicore Systems". In *Int. Symp. on Code Generation and Optimization*, pages 210–220, Washington, DC, USA, 2009. IEEE Computer Society.

[7] P. de Oliveira Castro, S. Louise, and D. Barthou. "Design-Space Exploration of Stream Programs through Semantic-Preserving Transformations", HAL-00447376. Technical report, 2010.

[8] N. Dershowitz and Z. Manna. "Proving Termination with Multiset Orderings". *Comm. ACM*, 22(8), 1979.

[9] J. Falk, J. Keinert, C. Haubelt, J. Teich, and S. S. Bhattacharyya. "A Generalized Static Data Flow Clustering Algorithm for MPSOC Scheduling of Multimedia Applications". In *ACM Conf. on Embedded Software*, 2008.

[10] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. "A Stream Compiler for Communication-Exposed Architectures". In *Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 291–303. ACM, 2002.

[11] T. Goubier, F. Blanc, S. Louise, R. Sirdey, and V. David. "Définition du Langage de Programmation ΣC", RT CEA LIST DTSI/SARC/08-466/TG. Technical report, 2008.

[12] G. Karypis and V. Kumar. "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs". *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.

[13] M. Kudlur and S. Mahlke. "Orchestrating the Execution of Stream Programs on Multicore Platforms". In *Proc. of the SIGPLAN conf. on Programming Language Design and Implementation*, pages 114–124. ACM, 2008.

[14] E. A. Lee and D. G. Messerschmitt. "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing". *IEEE Trans. Comput.*, 36(1):24–35, 1987.

[15] S.-w. Liao, Z. Du, G. Wu, and G.-Y. Lueh. "Data and Computation Transformations for Brook Streaming Applications on Multiprocessors". In *Proc. of the Int. Symp. on Code Generation and Optimization*, 2006.

[16] T. M. Parks, J. L. Pino, and E. A. Lee. "A Comparison of Synchronous and Cycle-Static Dataflow". In *Asilomar Conf. on Signals, Systems and Computers*, 1995.

[17] S. Stuijk, M. Geilen, and T. Basten. "Throughput-Buffering Trade-Off Exploration for Cyclo-Static and Synchronous Dataflow Graphs". *IEEE Trans. Comput.*, 57(10):1331–1345, 2008.

[18] W. Thies, M. Karczmarek, and S. P. Amarasinghe. "StreamIt: A Language for Streaming Applications". In *Proc. of the Int. Conf. on Compiler Construction*, pages 179–196, London, UK, 2002. Springer-Verlag.

[19] A. K. Verma, P. Brisk, and P. Ienne. "Data-Flow Transformations to Maximize the Use of Carry-Save Representation in Arithmetic Circuits". *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(10):1761–1774, 2008.

[20] M. Wiggers, M. Bekooij, and G. Smit. "Efficient Computation of Buffer Capacities for Cyclo-Static Dataflow Graphs". In *Proc. Int. Conf. on Design Automation*, pages 658–663. ACM, 2007.