Improving Communication Patterns in Polyhedral Process Networks

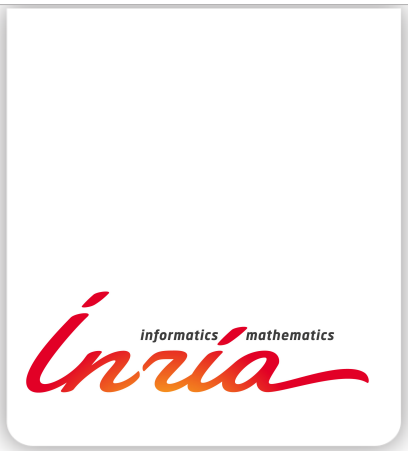
Christophe Alias

Improving Communication Patterns in Polyhedral Process Networks

Christophe Alias

RESEARCH REPORT
N° 9131
December 2017
Project-Team Roma

ISSN 0249-6399 ISRN INRIA/RR--9131--FR+ENG



Improving Communication Patterns in Polyhedral Process Networks

Christophe Alias*

Project-Team Roma

Research Report n° 9131 — version 1 — initial version December 2017 — revised version December 2017 — 13 pages

Abstract: Embedded systems performances are bounded by power consumption. The trend is to offload greedy computations on hardware accelerators as GPU, Xeon Phi or FPGA. FPGA chips combine both flexibility of programmable chips and energy-efficiency of specialized hardware and appear as a natural solution. Hardware design is long, fastidious and bug prone. Hardware compilers from high-level languages (High-level synthesis, HLS) are required to exploit all the capabilities of FPGA while satisfying tight time-to-market constraints. Compiler optimizations for parallelism and data locality restructure deeply the execution order of the processes, hence the read/write patterns in communication channels. This breaks most FIFO channels, which have to be implemented with addressable buffers. Expensive hardware is required to enforce synchronizations, which often results in dramatic performance loss. In this paper, we present an algorithm to partition the communications so that most FIFO channels can be recovered after a loop tiling, a key optimization for parallelism and data locality. Experimental results show a drastic improvement of FIFO detection for regular kernels at the cost of (few) additional storage. As a bonus, the storage can even be reduced in some cases.

Key-words: High-level synthesis, polyhedral compilation, FIFO, FPGA

* CNRS/ENS-Lyon/Inria/UCBL/Université de Lyon

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Amelioration des schémas de communication dans les réseaux de processus polyédriques

Résumé : Les performances des systèmes embarqués sont limitées par la consommation électrique. La tendance est de déléguer les calculs gourmands en ressources à des accélérateurs matériels comme les GPU, les Xeon Phi ou les FPGA. Les circuits FPGA allient la flexibilité d'un circuit programmable et l'efficacité énergétique d'un circuit spécialisé et apparaissent comme une solution naturelle. Des compilateurs de matériels à partir d'un langage haut-niveau sont requis pour exploiter au mieux les FPGA tout en remplissant les contraintes de mise sur le marché. Les optimisations de compilateur restructurent profondement les calculs et les schémas de communication (ordre de lecture/écriture). En conséquence, la plupart des canaux de communication ne sont plus des FIFOs et doivent être implémentées avec un tableau adressable, ce qui nécessite du matériel supplémentaire pour la synchronisation. Dans ce rapport, nous présentons un algorithme capable de partitionner les communications de sorte que la plupart des FIFO puissent être retrouvées après un tuilage de boucles. Les résultats expérimentaux confirment la puissance de note algorithme et son faible surcoût en stockage.

Mots-clés : Synthèse de circuit haut-niveau, compilation polyédrique, FIFO, FPGA

1 Introduction

Since the end of Dennard scaling, the performance of embedded systems is bounded by power consumption. The trend is to trade genericity (processors) for energy efficiency (hardware accelerators) by offloading critical tasks to specialized hardware. FPGA chips combine both flexibility of programmable chips and energy-efficiency of specialized hardware and appear as a natural solution. High-level synthesis (HLS) techniques are required to exploit all the capabilities of FPGA, while satisfying tight time-to-market constraints. Parallelization techniques from high-performance compilers are progressively migrating to HLS, particularly the models and algorithms from the polyhedral model – a powerful framework to design compiler optimizations. Additional constraints must be fulfilled before plugging a compiler optimization into an HLS tool. Unlike software, the hardware size is bounded by the available silicum surface. The bigger is a parallel unit, the less it can be duplicated, thereby limiting the overall performance. In particular, tough program optimizations are likely to spoil the performances if the circuit is not post-optimized carefully [5]. An important consequence is that the the roofline model is not longer valid in HLS [8]. Indeed, peak performance is no longer a constant: it decreases with the operational intensity. The bigger is the operational intensity, the bigger is buffer size and the less is the space remaining for the computation itself. Consequently, it is important to produce at source-level a precise model of the circuit which allows to predict accurately the resource consumption. Process networks are a natural and convenient intermediate representation for HLS [4, 13, 18]. A sequential program is translated to a process network by partitioning computations into processes and flow dependences into channels. Then, the the process and buffers are factorized and mapped to hardware.

In this paper, we focus on the translation of buffers to hardware. We propose an algorithm to restructure the buffers so they can be mapped to inexpensive FIFOs. Most often, a direct translation of a regular kernel – without optimization – produces to a process network with FIFO buffers [15]. Unfortunately, data transfers optimization [3] and generally loop tiling reorganizes deeply the computations, hence the read/write order in channels (communication patterns). Consequently, most channels may no longer be implemented by a FIFO. Additional circuitry is required to enforce synchronizations [4, 19, 14, 16] which result in larger circuits and causes performance penalties. In this paper, we make the following contributions:

- We propose an algorithm to reorganize the communications between processes so that more channels can be implemented as FIFO after a loop tiling. As far as we know, this is the first algorithm to recover FIFO communication patterns after a compiler optimization.
- Experimental results show that we can recover most of the FIFO disabled by communication optimization, and more generally any loop tiling, at almost no extra storage cost.

The remainder of this paper is structured as follows. Section 2 introduces polyhedral process network and discusses how communication patterns are impacted by loop tiling, Section 3 describes our algorithm to reorganize channels, Section 4 presents experimental results, Finally, Section 5 concludes this paper and draws future research directions.

1

2 Preliminaries

This section defines the notions used in the remainder of this paper. Section 2.1 and 2.2 introduces the basics of compiler optimization in the polyhedral model and defines loop tiling. Section 2.3

defines polyhedral process networks (PPN), shows how loop tiling disables FIFO communication patterns and outlines a solution.

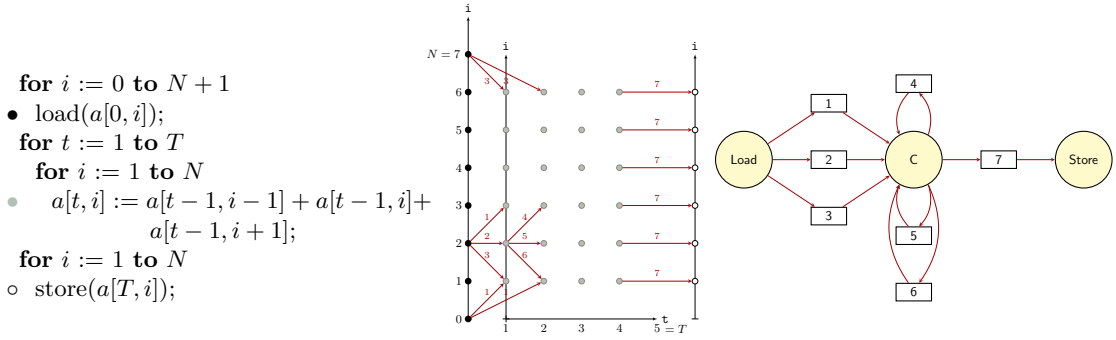
2.1 Polyhedral Model at a Glance

Translating a program to a process network requires to split the computation into processes and flow dependences into channels. The *polyhedral model* focuses on kernels whose computation and flow dependences can be predicted, represented and explored at compile-time. Specifically, the control must be predictable: (`for` loops, `if` with conditions on loop counters), arrays only; and loop bounds, conditions and array accesses must be affine functions of surrounding loop counters and structure parameters (typically the array size). This way, the computation may be represented with Presburger sets (typically approximated with convex polyhedra, hence the name). This makes possible to reason geometrically about the computation and to produce precise compiler analysis thanks to integer linear programming: flow dependence analysis [9], scheduling [7] or code generation [6, 12] to quote a few. Most compute-intensive kernels from linear algebra and image processing fit in this category. In some case, kernels with dynamic control can even fit in the polyhedral model after a proper abstraction [2]. Figure 1.(a) depicts a polyhedral kernel and (b) depicts the geometric representation of the computation for each assignment (• for assignment *load*, • for assignment *compute* and ∘ for assignment *store*). The vector $\vec{i} = (i_1, \ldots, i_n)$ of loop counters surrounding an assignment S is called an *iteration* of S. The execution of S at iteration $\vec{i}$ is denoted by $\langle S, \vec{i} \rangle$. The set $\mathcal{D}_S$ of iterations of S is called *iteration domain* of S. The original execution of the iterations of S follows the lexicographic order $\ll$ over $\mathcal{D}_S$. For instance, on the statement C: $(t, i) \ll (t', i')$ iff $t < t'$ or ($t = t'$ and $i < i'$). The lexicographic order over $\mathbb{Z}^d$ is naturally partitioned by depth: $\ll = \ll^1 \uplus \ldots \uplus \ll^d$ where $(u_1 \ldots u_d) \ll^k (v_1, \ldots, v_d)$ iff $\left(\wedge_{i=1}^{k-1} u_i = v_i\right) \wedge u_k < v_k$.

Dataflow Analysis On Figure 1.(b), red arrows depict several flow dependences (read after write) between executions instances. We are interested in flow dependences relating the production of a value to its consumption, not only a write followed by a read on the same location. These flow dependences are called *direct* dependences. Direct dependences represent the communication of values between two computations and allow to rule communications and synchronizations in the final process network. They are crucial to build the process network. Direct dependences can be computed exactly in the polyhedral model [9]. The result is a relation $\rightarrow$ relating each producer $\langle P, \vec{i} \rangle$ to one or more consumers $\langle C, \vec{j} \rangle$. Technically, $\rightarrow$ is a *Presburger relation* between vectors $(P, \vec{i})$ and vectors $(C, \vec{j})$ where assignments P and C are encoded as integers. For example, dependence 5 is summed up with the Presburger relation: $\{(t-1, i) \rightarrow (t, i),\ 0 < t \leq T \wedge 0 \leq i \leq N\}$. Presburger relations are computable and efficient libraries allow to manipulate them [17, 10]. In the remainder, direct dependence will be referred as flow dependence or dependence to simplify the presentation.

2.2 Scheduling and Loop Tiling

Compiler optimizations change the execution order to fulfill multiple goals such as increasing the parallelism degree or minimizing the communications. The new execution order is specified by a *schedule*. A schedule θ_S maps each execution $\langle S, \vec{i} \rangle$ to a timestamp $\theta_S(\vec{i}) = (t_1, \ldots, t_d) \in \mathbb{Z}^d$, the timestamps being ordered by the lexicographic order $\ll$. In a way, a schedule dispatches each execution instance $\langle S, \vec{i} \rangle$ into a new loop nest, $\theta_S(\vec{i}) = (t_1, \ldots, t_d)$ being the new iteration vector of $\langle S, \vec{i} \rangle$. A schedule θ induces a new execution order $\prec_\theta$ such that $\langle S, \vec{i} \rangle \prec_\theta \langle T, \vec{j} \rangle$ iff $\theta_S(\vec{i}) \ll \theta_T(\vec{j})$. Also, $\langle S, \vec{i} \rangle \preceq_\theta \langle T, \vec{j} \rangle$ means that either $\langle S, \vec{i} \rangle \prec_\theta \langle T, \vec{j} \rangle$ or $\theta_S(\vec{i}) = \theta_T(\vec{j})$. When



(a) Jacobi 1D kernel (b) Flow dependences (c) Polyhedral process network

Figure 1: Motivating example: Jacobi-1D kernel

a schedule is injective, it is said to be *sequential*: no execution is scheduled at the same time, hence everything is executed in sequence. In the polyhedral model, schedules are affine functions. They can be derived automatically from flow dependences [7]. On Figure 1, the original execution order is specified by the schedule $\theta_{\text{load}}(i) = (0, i)$, $\theta_{\text{compute}}(t, i) = (1, t, i)$ and $\theta_{\text{store}}(i) = (2, i)$. The lexicographic order ensures the execution of all the *load* instances (0), then all the *compute* instances (1) and finally all the *store* instances (2). Then, for each statement, the loops are executed in the specified order.

Loop tiling is a transformation which partitions the computation in tiles, each tile being executed atomically. Communication minimization [3] typically relies on loop tiling to tune the ratio computation/communication of the program beyond the ratio peak performance/communication bandwidth of the target architecture. Figure 2.(a) depicts the iteration domain of *compute* and the new execution order after tiling loops t and i. For presentation reasons, we depict a domain bigger than in Figure 1.(b) (with bigger N and M) and we depict only a part of the domain. In the polyhedral model, a loop tiling is specified by hyperplans with linearly independent normal vectors $\vec{\tau}_1, \ldots, \vec{\tau}_d$ where d is the number of nested loops (here $\vec{\tau}_1 = (0, 1)$ for the vertical hyperplanes and $\vec{\tau}_2 = (1, 1)$ for the diagonal hyperplanes). Roughly, hyper-plans along each normal vector $\vec{\tau}_i$ are placed at regular intervals b_i (here $b_1 = b_2 = 2$) to cut the iteration domain in tiles. Then, each tile is identified by an iteration vector $(\phi_1, \ldots, \phi_d)$, ϕ_k being the slice number of an iteration $\vec{i}$ along normal vector $\vec{\tau}_k$: $\phi_k = \vec{\tau}_k \cdot \vec{i} \div b_k$. The result is a new iteration domain (here $\hat{\mathcal{D}} = \{(\phi_1, \phi_2, t, i),\ 2\phi_1 \leq t < 2(\phi_1 + 1) \wedge 2\phi_2 \leq t + i < 2(\phi_2 + 1)\}$). In turn, this domain can be processed with polyhedral analysis: the polyhedral model is closed under loop tiling. In particular, the tiled domain can be scheduled. For instance, $\hat{\theta}_S(\phi_1, \phi_2, t, i) = (\phi_1, \phi_2, t, i)$ specifies the execution order depicted in Figure 2.(a)): tile with point (4,4) is executed, then tile with point (4,8), then tile with point (4,12), and so on. For each tile, the iterations are executed for each t, then for each i.

2.3 Polyhedral Process Networks

Given the iteration domains and the flow dependence relation, $\rightarrow$, we derive a *polyhedral process network* by partitioning iterations domains into process and flow dependence into channels. More formally, a polyhedral process network is a couple $(\mathcal{P}, \mathcal{C})$ such that:

- Each process $P \in \mathcal{P}$ is specified by an iteration domain $\mathcal{D}_P$ and a sequential schedule θ_P inducing an execution order $\prec_P$ over $\mathcal{D}_P$. Each iteration $\vec{i} \in \mathcal{D}_P$ realizes the execution

instance $\mu_P(\vec{i})$ in the program. The processes partition the execution instances in the program: $\{\mu_P(\mathcal{D}_P)$ for each process $P\}$ is a partition of the program computation.

- Each channel $c \in \mathcal{C}$ is specified by a producer process $P_c \in \mathcal{P}$, a consumer process $C_c \in \mathcal{P}$ and a dataflow relation $\rightarrow_c$ relating each production of a value by P_c to its consumption by C_c: if $\vec{i} \rightarrow_c \vec{j}$, then execution $\vec{i}$ of P_c produces a value read by execution $\vec{j}$ of C_c. $\rightarrow_c$ is a subset of the flow dependences from P_c to C_c and the collection of $\rightarrow_c$ for each channel c between two given processes P and C, $\{\rightarrow_c,\ (P,C) = (P_c, C_c)\}$, is a partition of flow dependences from P to C.

The goal of this paper is to find out a partition of flow dependences for each producer/consumer couple (P, C), such that most channels from P to C can be realized by a FIFO.

Figure 1.(c) depicts the PPN obtained with the canonical partition of computation: each execution $\langle S, \vec{i}\rangle$ is mapped to process P_S and executed at process iteration $\vec{i}$: $\mu_{P_S}(\vec{i}) = \langle S, \vec{i}\rangle$. For presentation reason the *compute* process is depicted as C. Dependence depicted as i on the dependence graph in (b) are solved by channel i. To read the input values in parallel, we use a different channel per couple producer/read reference, hence this partitioning. We assume that, *locally*, each process executes instructions in the same order than in the original program: $\theta_{\text{load}}(i) = i$, $\theta_{\text{compute}}(t, i) = (t, i)$ and $\theta_{\text{store}}(i) = i$. Remark that the leading constant (0 for *load*, 1 for *compute*, 2 for *store*) has disappeared: the timestamps only define an order local to their process: $\prec_{load}$, $\prec_{compute}$ and $\prec_{store}$. The global execution order is driven by the dataflow semantics: an operation is executed whenever its operands are available. The next step is to detect communication patterns to figure out how to implement channels.

Communication Patterns A channel $c \in \mathcal{C}$ might be implemented by a FIFO iff the consumer C_c read the values from c in the same order than the producer P_c write them to c (*in-order*) and each value is read exactly once (*unicity*) [13, 15]. The *in-order* constraint can be written:

$$\begin{aligned}&\text{in-order}(\rightarrow_c, \prec_P, \prec_C) :=\\ &\quad \forall x \rightarrow_c x', \forall y \rightarrow_c y' : x' \prec_C y' \Rightarrow x \preceq_P y\end{aligned}$$

The unicity constraints can be written:

$$\begin{aligned}&\text{unicity}(\rightarrow_c) :=\\ &\quad \forall x \rightarrow_c x', \forall y \rightarrow_c y' : x' \neq y' \Rightarrow x \neq y\end{aligned}$$

Notice that unicity depends only on the dataflow relation $\rightarrow_c$, it is independent from the execution order of the producer process $\prec_P$ and the consumer process $\prec_C$. Furthermore, $\neg$in-order$(\rightarrow_c, \prec_P, \prec_C)$ and $\neg$unicity$(\rightarrow_c)$ amount to check the emptiness of a convex polyhedron, which can be done by most LP solvers.

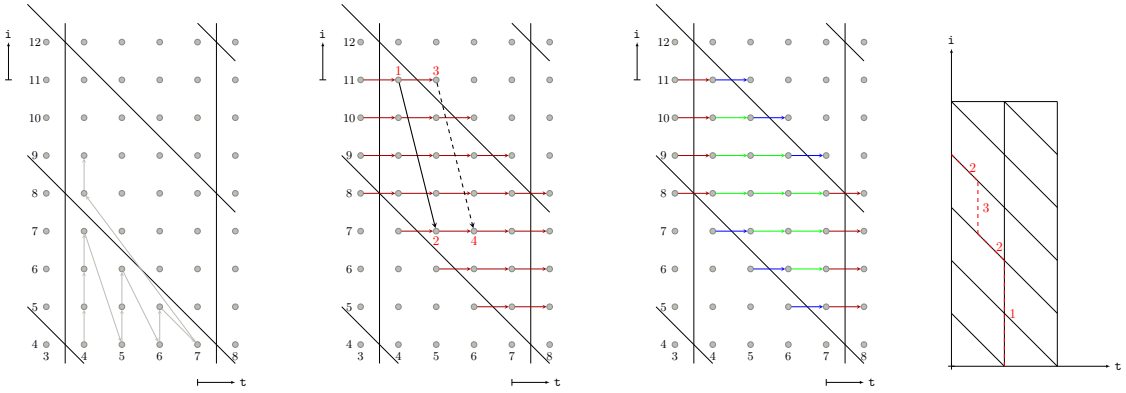
Finally, a channel may be implemented by a FIFO iff it verifies both in-order and unicity constraints:

$$\begin{aligned}&\text{fifo}(\rightarrow_c, \prec_P, \prec_C) :=\\ &\quad \text{in-order}(\rightarrow_c, \prec_P, \prec_C) \wedge \text{unicity}(\rightarrow_c)\end{aligned}$$

When the consumer reads the data in the same order than they are produced but a datum may be read several times: in-order$(\rightarrow_c, \prec_P, \prec_C) \wedge \neg$unicity$(\rightarrow_c)$, the communication pattern is said to be *in-order with multiplicity*: the channel may be implemented with a FIFO and a register keeping the last read value for multiple reads. However, additional circuitry is required to trigger the write of a new datum in the register [13]: this implementation is more expensive than a single FIFO. Finally, when we have neither in-order nor unicity: $\neg$in-order$(\rightarrow_c, \prec_P, \prec_C) \wedge \neg$unicity$(\rightarrow_c)$, the communication pattern is said to be *out-of-order without multiplicity*: significant hardware

resources are required to enforce flow- and anti- dependences between producer and consumer and additional latencies may limit the overall throughput of the circuit [4, 19, 14, 16].

Consider Figure 1.(c), channel 5, implementing dependence 5 (depicted on (b)) from $\langle \bullet, t-1, i \rangle$ (write $a[t,i]$) to $\langle \bullet, t, i \rangle$ (read $a[t-1,i]$). With the schedule defined above, the data are produced ($\langle \bullet, t-1, i \rangle$) and read ($\langle \bullet, t-1, i \rangle$) in the same order, and only once: the channel may be implemented as a FIFO. Now, assume that process *compute* follows the tiled execution order depicted in Figure 2.(a). The execution order now executes tile with point (4,4), then tile with point (4,8), then tile with point (4,12), and so on. In each tile, the iterations are executed for each t, then for each i. Consider iterations depicted in red as $1, 2, 3, 4$ in Figure 2.(b). With the new execution order, we execute successively 1,2,4,3, whereas an in-order pattern would have required 1,2,3,4. Consequently, channel 5 is no longer a FIFO. The same hold for channel 4 and 6. Now, the point is to partition dependence 5 and others so FIFO communication pattern hold.



(a) Loop tiling (b) Communication pattern (c) Our solution (d) Buffers size

Figure 2: Impact of loop tiling on communication pattern

Consider Figure 2.(c). Dependence 5 is partitioned in 3 parts: red dependences crossing tiling hyperplane ϕ_1 (direction t), blue dependences crossing tiling hyperplane $t+i$ (direction $t+i$) and green dependences inside a tile. Since the execution order in a tile is the same than the original execution order (actually a subset of the original execution order), green dependences will clearly verify the FIFO communication pattern. As concerns blue and red dependences, source and target are executed in the same order because the execution order is the same for each tile and dependence 5 happens to be short enough. In practice, this partitioning is effective to reveal FIFO channels. In the next section, we propose an algorithm to find such a partitioning.

3 Our Algorithm

Figure 3 depicts our algorithm for partitioning channels given a polyhedral process network $(\mathcal{P}, \mathcal{C})$ (line 5). For each channel c from a producer $P = P_c$ to a consumer $C = C_c$, the channel is partitioned by depth along the lines described in the previous section (line 7). $\mathcal{D}_P$ and $\mathcal{D}_C$ are assumed to be tiled with the same number of hyperplanes. P and C are assumed to share a schedule with the shape: $\theta(\phi_1, \ldots, \phi_n, \vec{i}) = (\phi_1, \ldots, \phi_n, \vec{i})$. This case arise frequently with tiling schemes for I/O optimization [4]. If not, the next channel $\rightarrow_c$ is considered (line 6). The split is realized by procedure SPLIT (lines 1–4). A new partition is build starting from the empty set. For each depth (hyperplane) of the tiling, the dependences crossing that hyperplane are filtered

```
1   SPLIT(→c,θP,θC)
2     for k := 1 to n
3       ADD(→c ∩{(x, y), θP(x) ≪k θC(y)});
4     ADD(→c ∩{(x, y), θP(x) ≈n θC(y)});

5   FIFOIZE((P, C))
6     for each channel c
7       {→c^1, ..., →c^{n+1}} := SPLIT(→c,Pc,Cc);
8       if fifo(→c^k, ≺θPc, ≺θCc) ∀k
9         REMOVE(→c);
10        INSERT(→c^k) ∀k;
```

Figure 3: Our algorithm for partitioning channels

and added to the partition (line 3): this gives dependences $\rightarrow_c^1, \ldots, \rightarrow_c^n$. Finally, dependences lying in a tile (source and target in the same tile) are added to the partition (line 4): this gives $\rightarrow_c^{n+1}$. $\theta_P(x) \approx^n \theta_C(y)$ means that the n first dimensions of $\theta_P(x)$ and $\theta_C(y)$ (tiling coordinates $(\phi_1, \ldots, \phi_n)$) are the same: x and y belong to the same tile. Consider the PPN depicted in Figure 1.(c). with the tiling and schedule discussed above : process *compute* is tiled as depicted in Figure 2.(c) with the schedule $\theta_{\text{compute}}(\phi_1, \phi_2, t, i) = (\phi_1, \phi_2, t, i)$. Since processes *load* and *store* are not tiled, the only channels processed by our algorithm are 4,5 and 6. SPLIT is applied on the associated dataflow relations $\rightarrow_4$, $\rightarrow_5$ and $\rightarrow_6$. Each dataflow relation is split in three parts as depicted in Figure 2.(c). For $\rightarrow_5$: $\rightarrow_5^1$ crosses hyperplane t (red), $\rightarrow_5^2$ crosses hyperplane $t + i$ (blue) and $\rightarrow_5^3$ stays in a tile (green).

This algorithm works pretty well for short uniform dependences $\rightarrow_c$: if fifo(c) before tiling, then, after tiling, the algorithm can split c in such a way that we get FIFOs. However, when dependences are longer, *e.g.* $(t, i) \rightarrow (t, i + 2)$, the target operations $(t, i + 2)$ reproduce the tile execution pattern, which prevents to find a FIFO. The same happens when the tile hyperplanes are "too skewed", e.g. $\tau_1 = (1, 1)$, $\tau_2 = (2, 1)$, dependence $(t - 1, i - 1) \rightarrow (t, i)$. Figure 2.(d) depicts the volume of data to be stored on the FIFO produced for each depth. In particular, dotted line with k indicates iterations producing data to be kept in the FIFO at depth k. FIFO at depth 1 (dotted line with 1) must store N data at the same time. Similarly, FIFO at depth 2 stores at most b_1 data and FIFO at depth 3 stores at most b_2 data. Hence, on this example, each transformed channel requires $b_1 + b_2$ additional storage. In general the additional storage requirements are one order of magnitude smaller than the original FIFO size and stays reasonable in practice, as shown in the next section.

4 Experimental Evaluation

This section presents the experimental results obtained on the benchmarks of the polyhedral community. We demonstrate the capabilities of our algorithm at recovering FIFO communication patterns after loop tiling and we show how much additional storage is required.

Experimental Setup We have run our algorithm on the kernels of PolyBench/C v3.2 [11]. Tables 1 and 2 depicts the results obtained for each kernel. Each kernel is tiled to reduce I/O while exposing parallelism [4] and translated to a PPN using our research compiler, DCC (DPN C Compiler). DCC actually produces a DPN (Data-aware Process Network), a PPN optimized for a specific tiled pattern. DPN features additional control processes and synchronization for

I/O and parallelism which have nothing with our optimization. So, we actually only consider the PPN part of our DPN. We have applied our algorithm to each channel to expose FIFO patterns. For each kernel, we compare the PPN obtained after tiling to the PPN processed by our algorithm.

Results Table 1 depicts the capabilities of our algorithm to find out FIFO patterns. For each kernel, we provide the channels characteristics on the original tiled PPN (Before Partitioning) and after applying our algorithm (After Partitioning). We give the total number of channels (#channel), the FIFO found among these channels (#fifo), the number of channels which were successfully turned to FIFO thanks to our algorithm (#fifo-split), the ratios #fifo/#channel (%fifo) and #fifo-split/#channel (%fifo-split), the cumulated size of the FIFO found (fifo-size) and the cumulated size of the channels found, including FIFO (total-size). On every kernel, our algorithm succeeds to expose more FIFO patterns (%fifo vs %fifo-split). On a significant number of kernels (11 among 15), we even succeed to turn all the compute channels to FIFO. On the remaining kernels, we succeed to recover all the FIFO communication patterns disabled by the tiling. Even though our method is not complete, as discussed in section 3, it happens that all the kernels fulfill the conditions expected by our algorithm (short dependence, tiling hyperplanes not too skewed).

Kernel	Before Partitioning							After Partitioning			
	#channel	#fifo	#fifo-split	%fifo	%fifo-split	fifo-size	total-size	#channel	#fifo	fifo-size	total-size
trmm	2	1	2	50%	100%	256	512	3	3	513	513
gemm	2	1	2	50%	100%	16	528	3	3	304	304
syrk	2	1	2	50%	100%	1	8193	3	3	8194	8194
symm	6	3	6	50%	100%	18	818	7	7	819	819
gemver	6	3	5	50%	83%	4113	4161	7	6	4146	4162
gesummv	6	6	6	100%	100%	96	96	6	6	96	96
syr2k	2	1	2	50%	100%	1	8193	3	3	8194	8194
lu	8	0	3	0%	37%	0	1088	11	6	531	1091
cholesky	9	3	6	33%	66%	513	1074	11	8	788	1076
atax	5	3	4	60%	80%	48	65	5	4	49	65
doitgen	3	2	3	66%	100%	8192	12288	4	4	12289	12289
jacobi-2d	10	0	10	0%	100%	0	8320	18	18	8832	8832
seidel-2d	9	0	9	0%	100%	0	49952	16	16	52065	52065
jacobi-1d	6	1	6	16%	100%	1	1153	10	10	1175	1175
heat-3d	20	0	20	0%	100%	0	148608	38	38	158992	158992

Table 1: Detailed results

Table 2 depicts the additional storage required after splitting channels. For each kernel, we compare the cumulative size of channels split and successfully turn to a FIFO (size-fifo-fail) to the cumulative size of the FIFOs generated by the splitting (size-fifo-split). The size unit is a datum *e.g.* 4 bytes if a datum is a 32 bits float. We also quantify the additional storage required by split channels compared to the original channel (Δ := size-fifo-split - size-fifo-fail / size-fifo-fail). It turns out that the FIFO generated by splitting use mostly the same data volume than the original channels. Additional resources are due to our sizing heuristic [1], which rounds channel size to a power of 2. Surprisingly, splitting can sometimes help the sizing heuristic to find out a smaller size (kernel `gemm`), and then reducing the storage requirements. Indeed, splitting decompose channel into channels of a smaller dimension, for which our sizing heuristic is more precise. In a way, our algorithm allows to finds out a nice piecewise allocation function whose

footprint is smaller than a single piece allocation. We plan to exploit this nice side effect in the future.

kernel	Size-fifo-fail	Size-fifo-split	Δ
trmm	256	257	0%
gemm	512	288	-44%
syrk	8192	8193	0%
symm	800	801	0%
gemver	32	33	3%
gesummv	0	0	
syr2k	8192	8193	0%
lu	528	531	1%
cholesky	273	275	1%
atax	1	1	0%
doitgen	4096	4097	0%
jacobi-2d	8320	8832	6%
seidel-2d	49952	52065	4%
jacobi-1d	1152	1174	2%
heat-3d	148608	158992	7%

Table 2: Impact of splitting on storage requirements

5 Conclusion

In this paper, we have proposed an algorithm to reorganize the channels of a polyhedral process network to reveal more FIFO communication patterns. Specifically, our algorithm operates producer/consumer processes whose iteration domain has been partitioned by a loop tiling. Experimental results shows that our algorithm allows to recover most of the FIFO disabled by loop tiling with almost the same storage requirement. Our algorithm is sensible to the dependence size and the loop tiling chosen. In the future, we plan to design a reorganization algorithm provably complete, in the meaning that a FIFO channel will be recovered whatever the dependence size and the tiling used. We also observe that splitting channels can reduce the storage requirements in some cases. We plan to investigate how such cases can be revealed automatically.

References

[1] Christophe Alias, Fabrice Baray, and Alain Darte. Bee+Cl@k: An implementation of lattice-based array contraction in the source-to-source translator Rose. In *ACM Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES'07)*, 2007.

[2] Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In *International Static Analysis Symposium (SAS'10)*, 2010.

[3] Christophe Alias, Alain Darte, and Alexandru Plesco. Optimizing remote accesses for offloaded kernels: Application to high-level synthesis for FPGA. In *ACM SIGDA Intl. Conference on Design, Automation and Test in Europe (DATE'13)*, Grenoble, France, 2013.

[4] Christophe Alias and Alexandru Plesco. Data-aware Process Networks. Research Report RR-8735, Inria - Research Centre Grenoble – Rhône-Alpes, June 2015.

[5] Christophe Alias and Alexandru Plesco. Optimizing Affine Control with Semantic Factorizations. *ACM Transactions on Architecture and Code Optimization (TACO)* , 14(4):27, December 2017.

[6] Cédric Bastoul. Efficient code generation for automatic parallelization and optimization. In *2nd International Symposium on Parallel and Distributed Computing (ISPDC 2003), 13-14 October 2003, Ljubljana, Slovenia*, pages 23–30, 2003.

[7] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 101–113, 2008.

[8] Bruno da Silva, An Braeken, Erik H D'Hollander, and Abdellah Touhafi. Performance modeling for fpgas: extending the roofline model with high-level synthesis tools. *International Journal of Reconfigurable Computing*, 2013:7, 2013.

[9] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[10] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and Dave Wonnacott. The omega calculator and library, version 1.1. 0. *College Park, MD*, 20742:18, 1996.

[11] Louis-Noël Pouchet. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/˜ pouchet/software/polybench/[cited July,]*, 2012.

[12] Fabien Quilleré, Sanjay Rajopadhye, and Doran Wilde. Generation of efficient nested loops from polyhedra. *International journal of parallel programming*, 28(5):469–498, 2000.

[13] Alexandru Turjan. *Compiling nested loop programs to process networks.* PhD thesis, Leiden Institute of Advanced Computer Science (LIACS), and Leiden Embedded Research Center, Faculty of Science, Leiden University, 2007.

[14] Alexandru Turjan, Bart Kienhuis, and E Deprettere. Realizations of the extended linearization model. *Domain-specific processors: systems, architectures, modeling, and simulation*, pages 171–191, 2002.

[15] Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. Classifying interprocess communication in process network representation of nested-loop programs. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(2):13, 2007.

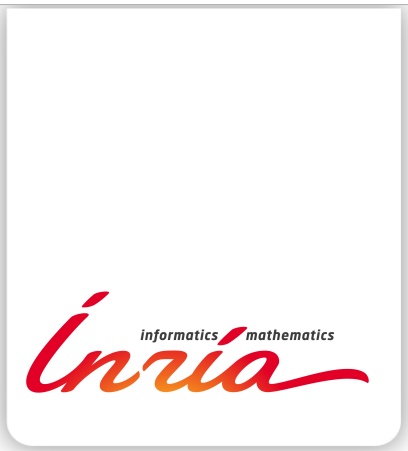
[16] Sven van Haastregt and Bart Kienhuis. Enabling automatic pipeline utilization improvement in polyhedral process network implementations. In *Application-Specific Systems, Architectures and Processors (ASAP), 2012 IEEE 23rd International Conference on*, pages 173–176. IEEE, 2012.

[17] Sven Verdoolaege. isl: An integer set library for the polyhedral model. In *ICMS*, volume 6327, pages 299–302. Springer, 2010.

[18] Sven Verdoolaege. *Polyhedral Process Networks*, pages 931–965. Handbook of Signal Processing Systems. 2010.

[19] C Zissulescu, A Turjan, B Kienhuis, and E Deprettere. Solving out of order communication using cam memory; an implementation. In *13th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC 2002)*, 2002.

Contents



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399