FIFO Recovery by Depth-Partitioning is Complete on Data-aware Process Networks

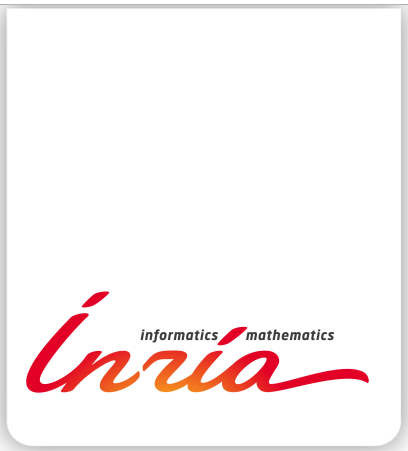
Christophe Alias

FIFO Recovery by Depth-Partitioning is Complete on Data-aware Process Networks

Christophe Alias

RESEARCH REPORT
N° 9187
Juin 2018
Project-Team Cash

ISSN 0249-6399 ISRN INRIA/RR--9187--FR+ENG



FIFO Recovery by Depth-Partitioning is Complete on Data-aware Process Networks

Christophe Alias*

Project-Team Cash

Research Report n° 9187 — version 1 — initial version Juin 2018 — revised version Juin 2018 — 17 pages

Abstract: Computing 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 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 build on our 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. We describe a class of process networks where the algorithm can recover all the FIFO channels. We point out the limitations of the algorithm outside of that class. Experimental results confirm the completeness of the algorithm on the class and reveal good performance outside of the class.

Key-words: High-level synthesis, automatic parallelization, polyhedral model, synchronization

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

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

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

Sur la complétude du partitionnement par profondeur pour réparer les FIFO des réseaux de processus DPN

Résumé : Les performances des ordinateurs 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 montrons que notre algorithme de partitionnement des communications par profondeur est complet sur les réseaux DPN: toutes les FIFO peuvent être retrouvées après un tuilage de boucles. Nous montrons l'incomplétude de notre algorithme sur des réseaux de processus polyédrique plus généraux. Ces résultats théoriques sont ensuite confirmés par des résultats expérimentaux.

Mots-clés : Synthèse de circuit haut-niveau, parallélisation automatique, modèle polyédrique, synchronisation

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 [7], 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 silicon surface. The bigger a parallel unit is, the less it can be duplicated, thereby limiting the overall performance. Particularly, tricky program optimizations are likely to spoil the performances if the circuit is not post-optimized carefully [5]. We believe that source-level optimizations (directly on the source program) should be avoided in HLS, and moved to *middle-end level*, on an intermediate representation *closed to the final circuit*. Process networks are such a natural and convenient intermediate representation for HLS [4, 12, 13, 18]. A sequential program is translated to a process network by partitioning computations into processes and flow dependences into channels. Then, the processes and buffers are factorized and mapped to hardware.

In this paper, we focus on the translation of the 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. This results in larger circuits and causes performance penalties [4, 19, 14, 16]. This a major lock which prevents, so far, to incorporate fine-grain polyhedral loop optimizations on the *middle-end* of HLS tools. In this paper, we build on our algorithm presented in [1] to reorganize the communications between processes so that more channels can be implemented by a FIFO after a loop tiling. We make the following contributions:

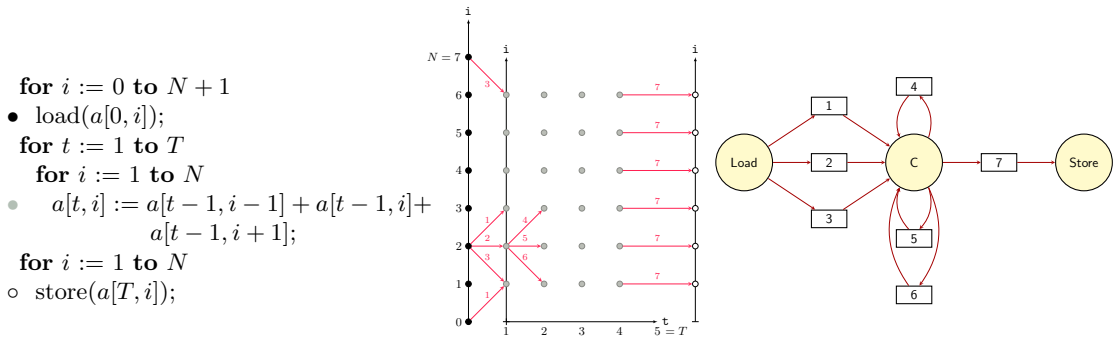
- We prove the completeness of our algorithm on Data-aware process networks (DPN) [4]: on a DPN, our algorithm can recover all the FIFO after a loop tiling. This feature is a step towards enabling polyhedral optimizations in HLS *at middle-end level*.
- We exhibit a counter-example which prove that our algorithm is no longer complete when the process network does not comply with the DPN process/channel partitioning scheme. Also, we discuss the criteria to be fullfilled by the kernel so all the FIFO can be recovered.
- Experimental results on Polybench/C kernels confirm the completeness of the algorithm on DPN. For non-DPN process networks, we show that the algorithm can recover a significant amount of FIFO.

The remainder of this paper is structured as follows. Section 2 introduces polyhedral process network, data-aware process networks and discusses how communication patterns are impacted by loop tiling, Section 3 recalls our algorithm to reorganize channels, prove the completeness of our algorithm on DPN and points-out the limitations of our algorithm on non-DPN process networks. 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. Sections 2.1 and 2.2 introduce 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. Finally Section 2.4 presents data-aware process networks (DPN), the particular kind of PPN on which our algorithm is proven to be complete.



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

Figure 1: Motivating example: Jacobi-1D kernel. (a) depicts a polyhedral kernel, (b) gives the polyhedral representation of loop iterations (•: load, •: compute, ○: store) and flow dependences (red arrows), then (c) gives a possible implementation as a polyhedral process network: each assignement (load, compute, store) is mapped to a different process and flow dependences (1 to 7) are solved through channels.

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. The control must be predictable: only `for` loops and `if` with conditions on loop counters are allowed. Data structures are bounded to arrays, pointers are not allowed. Also, 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 [8], scheduling [7] or code generation [6, 11] to quote a few. Most compute-intensive kernels from linear algebra and image processing fit in this category. In some cases, 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 $\left(\wedge_{i=1}^{k-1} u_i = v_i\right) \wedge u_k < v_k$. This property will be exploited by the partitioning algorithm. We now explain how producer/consumer relations are extracted from a polyhedral kernel and represented.

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 to the same location. These flow dependences are called *direct* dependences. Direct dependences represent the communication of values between two computations and drive 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 [8]. 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: $\{(\bullet, t-1, i) \rightarrow (\bullet, t, i),\ 0 < t \leq T \wedge 0 \leq i \leq N\}$. Presburger relations are computable and efficient libraries allow to manipulate them [17, 9]. 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 schedule is injective, it is said to be *sequential*: each execution is scheduled at a different 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 $\theta_{\text{load}}(i) = (0, i)$, $\theta_{\text{C}}(t, i) = (1, t, i)$ and $\theta_{\text{store}}(i) = (2, i)$, where C denotes the *compute* statement. 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 hyperplanes 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, hyperplanes along each normal vector $\vec{\tau}_i$ are placed at regular intervals b_i (here $b_1 = b_2 = 4$) 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 Presburger iteration domain, here $\hat{\mathcal{D}}_C = \{(\phi_1, \phi_2, t, i),\ 4\phi_1 \leq t < 4(\phi_1 + 1) \wedge 4\phi_2 \leq t + i < 4(\phi_2 + 1)\}$: the polyhedral model is closed under loop tiling. In particular, the tiled domain can be scheduled. For instance, $\hat{\theta}_C(\phi_1, \phi_2, t, i) = (\phi_1, \phi_2, t, i)$ 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

We derive a *polyhedral process network* by partitioning iterations domains into processes and the flow dependence relation into channels. Figure 1.(c) depicts a possible PPN from the Jacobi 1D kernel given on Figure 1.(a). For this example, we choose a canonical partition of the computation (one process per statement) and a partition of flow dependences such that there is single channel per couple producer/read reference, as motivated later. 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, $\{\rightarrow_c,\ (P_c, C_c) = (P, C)\}$, 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.

On Figure 1.(c), 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. Dependences depicted as k on the dependence graph in (b) are solved by channel k. 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: the next process operation is executed as soon as 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 reads the values from c in the same order than the producer P_c writes 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($\rightarrow_c$

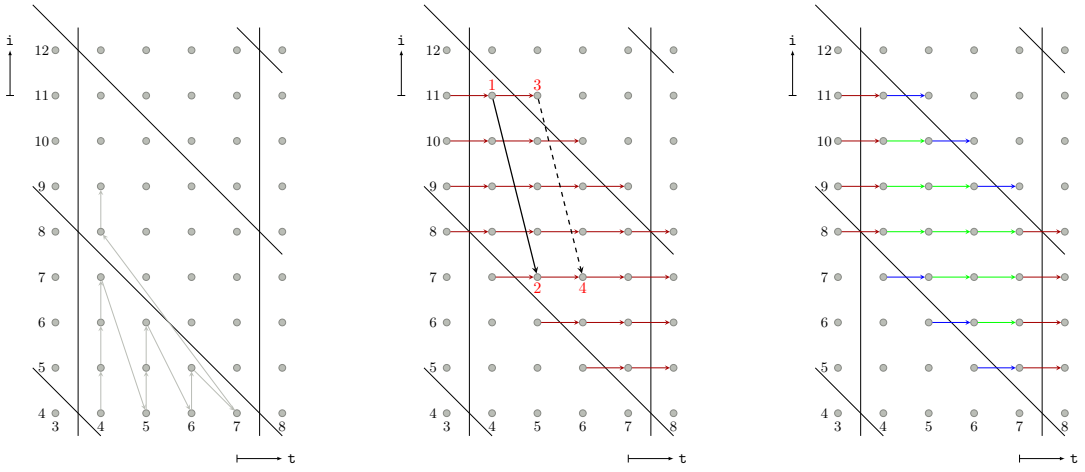
$, \prec_P, \prec_C)$ and $\neg\text{unicity}(\rightarrow_c)$ amount to check the emptiness of a convex polyhedron, which can be done by most LP solvers.

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

$$\text{fifo}(\rightarrow_c, \prec_P, \prec_C) := \\ \text{in-order}(\rightarrow_c, \prec_P, \prec_C) \wedge \text{unicity}(\rightarrow_c)$$

When the consumer reads the data in the same order than they are produced but a datum may be read several times: $\text{in-order}(\rightarrow_c, \prec_P, \prec_C) \wedge \neg\text{unicity}(\rightarrow_c)$, 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: $\neg\text{in-order}(\rightarrow_c, \prec_P, \prec_C) \wedge \neg\text{unicity}(\rightarrow_c)$, the communication pattern is said to be *out-of-order with 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 original sequential schedule, 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

Figure 2: **Impact of loop tiling on the communication patterns.** (a) gives the execution order of the Jacobi-1D main loop after a loop tiling, (b) shows how loop tiling disables the FIFO communication pattern, then (c) shows how to split the dependences into new channels so FIFO can be recovered.

Consider Figure 2.(c). Dependence 5 is partitioned in 3 parts: red dependences crossing tiling

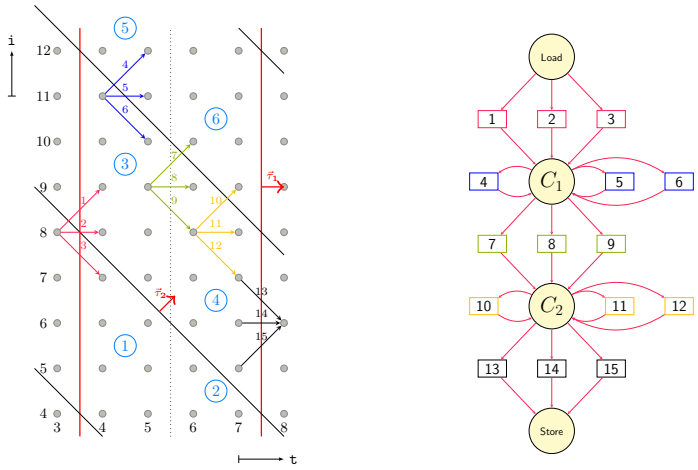
hyperplane ϕ_1 (direction t), blue dependences crossing tiling hyperplane ϕ_2 (direction $t+i$) and green dependences inside a tile. Since the execution order in a tile is the same as the original execution order (actually a subset of the original execution order), green dependences will verify the same FIFO communication pattern as in the non-tiled version. 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 recall our algorithm to find such a partitioning. The next subsection presents the Data-aware process networks, the particular kind of PPN on which the algorithm is complete.

2.4 Data-aware Process Networks

There are as many polyhedral process networks as possible partitions of the computations (processes) and dependences (channels) of the input kernel. A data-aware process network [4] is a process/channel partition template which allows, given a relevant loop tiling, to control the data transfers volume and the parallelism degree at the same time.

Figure 3 gives a possible DPN partition (b) from a loop tiling (a) on the motivating example. Given a loop tiling $\mathcal{D}_S \mapsto \hat{\mathcal{D}}_S$ for each statement S, we consider the execution order induced by the schedule $\hat{\theta}_S(\phi_1,\ldots,\phi_n,\vec{i}) = (\phi_1,\ldots,\phi_n,\vec{i})$: for each $(\phi_1,\ldots,\phi_{n-1})$, we execute the sequence of tiles $(\phi_n,\vec{i})$ such that $(\phi_1,\ldots,\phi_n,\vec{i}) \in \hat{\mathcal{D}}_S$. The set of executions $\langle S,\vec{i}\rangle$ given $(\phi_1,\ldots,\phi_{n-1})$ is called a *band* and written $\mathcal{B}(\phi_1,\ldots,\phi_{n-1})$. On Figure 3.(a), $\mathcal{B}(1)$ is the set of tiles surrounded by thick red lines. First, band $\mathcal{B}(0)$ (with $0 \leq t < 4$) is executed tile by tile, then band $\mathcal{B}(1)$ is executed tile by tile, and so on. On DPN, a band acts as a *reuse unit*. This means that incoming dependences (here 1,2,3) are loaded and outcoming dependences (here 13,14,15) are stored to an external storage unit. Dependences inside a band are resolved through channels (here 4 to 12). Inside a band, the computations may be split in parallel process thanks to surrounding hyperplanes $(\vec{\tau}_1,\ldots,\vec{\tau}_{n-1})$. On Figure 3.(a), each band is split in two sub-bands separated by a dotted line, thanks to hyperplane $\vec{\tau}_1 = (1,0)$. Each sub-band is implemented by a separate process on Figure 3.(b): C_1 for the left sub-band, C_2 for the right sub-band. The DPN partitioning allows to tune the arithmetic intensity ($A.I.$) by playing on the band width b (here $A.I. = 2 \times bN/2N = b$) and to select the parallelism independently Each parallel process is identified by its coordinate $\vec{\ell} = (\ell_1,\ldots,\ell_{n-1})$ along surrounding hyperplanes. Assuming p parallel instances along each surrounding hyperplanes we have $0 \leq \ell_k < p$. The parallel instance of C of coordinate $\vec{\ell}$, for $0 \leq \ell_k < p$ is written $C_{\vec{\ell}}$ (here we have parallel instances C_0 and C_1). We distinguish between *i/o dependences*, ($\rightarrow_{i/o}$ source or target outside of the band *e.g.* $1,2,3$ or $13,14,15$), *local dependences* to each parallel process ($\rightarrow_{local}$, source and target on the same parallel process *e.g.* $4,5,6$) and *synchronization dependences* between parallel process ($\rightarrow_{synchro}$, source and target on different parallel process *e.g.* $7,8,9$). For each array a loaded/stored through $\rightarrow_{i/o}$, a load (resp. store) process Load_a (resp. Store_a) is created. For each i/o dependence $(P_{\vec{\ell}},\phi_1,\ldots,\phi_n,\vec{i}) \rightarrow_{i/o} (C_{\vec{\ell'}},\phi'_1,\ldots,\phi'_n,\vec{j})$, assuming the data written by $P_{\vec{\ell}}$ is $a[u(\vec{i})]$ and the data read by $C_{\vec{\ell'}}$ is $a[v(\vec{j})]$, the dependence is removed and replaced by a dependence flowing to Store_a: $(P_{\vec{\ell}},\phi_1,\ldots,\phi_n,\vec{i}) \rightarrow_{i/o} (\text{Store}_a,\phi_1,\ldots,\phi_n,u(\vec{i}))$, and by a dependence flowing from Load_a: $(\text{Load}_a,\phi_1,\ldots,\phi_n,v(\vec{j})) \rightarrow_{i/o} (C_{\vec{\ell'}},\phi'_1,\ldots,\phi'_n,\vec{j})$ (on Figure (b), these processes are simply named Load/Store). Dependence $\rightarrow_{i/o}$ and processes Load_a and Store_b can be further optimized to improve data reuse through the band [4]. Finally each channel c of the original PPN is partitioned in such a way that each new channel c' connects a single producer process and consumer process (here original PPN channel 4 is split into DPN channels $4,7,10$). We keep track of the original PPN channel with the mapping μ: $\mu(c') := c$ (here $\mu(4) = \mu(7) = \mu(10) = 4$).

The DPN partitioning will be denoted by $(\mathcal{L}, \mathcal{P}', \mathcal{S}, \mathcal{C}')$ where $\mathcal{L}$ is the set of Load process, $\mathcal{P}'$ is the set of parallel processes, $\mathcal{S}$ is the set of store processes Store and $\mathcal{C}'$ is the set of channels.



(a) DPN partitioning (b) DPN

Figure 3: **Data-aware process network (DPN) for the Jacobi-1D kernel.** (a) computations are executed per band (between red thick lines), tile per tile. Incoming dependences (1,2,3) are loaded, outcoming dependences (13,14,15) are stored, internal dependences (4 to 12) are solved through local channels depicted in (b). Parallelization is derived by splitting a band with tiling hyperplanes (here the dotted line). With this scheme, arithmetic intensity and parallelism can be tuned easily.

3 The Partitioning Algorithm

In this section, we recall our algorithm presented in [1] to restructure the channels so FIFO channels can be recovered after a loop tiling. Then, we present the contributions of this paper. Section 3.1 proves that the algorithm is complete on DPN. Then, Section 3.2 points out the limits of the algorithm on general PPN without the DPN partitioning.

Figure 4 depicts the 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: $\theta(\phi_1, \ldots, \phi_n, \vec{i}) = (\phi_1, \ldots, \phi_n, \vec{i})$. In other words, the execution of a tile follow the order as in the original program. This case arises 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 built starting from the empty set. For each depth (hyperplane) of the tiling, the dependences crossing that hyperplane are filtered 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.

```
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,θPc,θCc);
8      if fifo(→c^k, ≺θPc, ≺θCc) ∀k
9        REMOVE(→c);
10       INSERT(→c^k) ∀k;
```

Figure 4: **Our algorithm for partitioning channels [1]**. The algorithm SPLIT produces the dependence partition described on Figure 2.(c), for each depth k of the producer schedule θ_P and the consumer schedule θ_C. The dependence partition is kept if each set of the dependence partition has a FIFO pattern.

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 the 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). Each of these new dataflow relations $\rightarrow_5^1$, $\rightarrow_5^2$,$\rightarrow_5^3$ exhibit a FIFO communication pattern w.r.t. θ. The total space occupied by these new FIFO is almost the same as the original channel plus a small overhead as discussed in [1].

Consider the DPN depicted in Figure 3. This DPN is derived from the PPN in Figure 1.(c), with the tiling described above. The dependences are pre-split into channels to fit with the DPN execution model: load/store dependences (red and black), intra-process dependences for each parallel process C_1 and C_2 (blue for C_1, yellow for C_2) and communication between parallel processes (green dependences). Dependence 4 and 5 lead to the same faulty communication pattern as in Figure 2.(b). Our algorithm will split them between dependence instances crossing the $t+i$ hyperplane (as on the figure) and dependence instances with source and target inside the same tile. Here, each split gives a FIFO. Similarly, the same splitting occurs for process C_2. Communication dependences (7,8,9, green) can be implemented directly by a FIFO: no split is required on this example. Finally, all the compute-to-compute channels (4 to 12) can be turned to FIFO. Load/Store dependences are not considered for splitting because load and store buffers are usually not FIFOs. Indeed, each loaded data will usually be read several times as DPN are tuned for a maximal data reuse. Also, the store process will read the data to prepare burst data transfers. Hence, the store read order is generally not the write order of the producer process (here process C_2).

3.1 Completeness on DPN partitioning

The algorithm always succeeds to recover all the FIFO on PPN with the DPN execution schema: if a PPN channel c is a FIFO before tiling, then, *after tiling and turning the PPN into a DPN*, the algorithm can split each DPN channel c' created from the PPN channel c ($c' = \mu(c)$) in such a way that we get FIFOs. We say that the splitting algorithm is *complete* over DPN. This is an important *enabling* property for DPN: essentially, it means that DPN allow to implement the

tiling transformation while keeping FIFO channels, which is necessary (though not sufficient) to map it to hardware. In the next section, we will show that general PPN (without DPN partitioning) are not complete. This shows that DPN is an important subset of PPN.

We start by a fundamental lemma, asserting that if a dependence relation $\rightarrow$ can be implemented with a FIFO w.r.t. a given schedule then any subset of $\rightarrow$ with the same schedule can also be implemented by a FIFO:

Lemma 3.1 *Consider a PPN $(\mathcal{P},\mathcal{C})$ and a channel $c \in \mathcal{C}$. If $\rightarrow' \subset \rightarrow_c$ and fifo$(\rightarrow_c, \theta_{P_c}, \theta_{C_c})$ then: fifo$(\rightarrow', \theta_{P_c}, \theta_{C_c})$.*

Proof: Any counter example $\rightarrow' \subset \rightarrow_c$ such that $\neg$fifo$(\rightarrow', \theta_{P_c}, \theta_{C_c})$ would imply that $\neg$fifo$(\rightarrow_c, \theta_{P_c}, \theta_{C_c})$: if $\neg$in-order$(\rightarrow', \theta_{P_c}, \theta_{C_c})$) then $\neg$in-order$(\rightarrow_c, \theta_{P_c}, \theta_{C_c})$), also: if $\neg$unicity$(\rightarrow')$ then $\neg$unicity$(\rightarrow_c)$. This contradicts the hypothesis. Q.E.D.

With this lemma, we can prove the completeness of the algorithm on any DPN partitioning:

Property 3.2 *Consider a PPN $(\mathcal{P},\mathcal{C})$ and a DPN partitioning $(\mathcal{L},\mathcal{P}',\mathcal{S},\mathcal{C}')$ w.r.t. a tiled schedule. Then, for each channel $c' \in \mathcal{C}'$ of the DPN: if the original channel in the PPN $c = \mu(c')$ is a FIFO, then the split of c' will be a FIFO as well.*

Proof: Since we only consider channels between compute processes of $\mathcal{P}'$, we can exclude the case $\rightarrow_{c'} \subset \rightarrow_{i/o}$ (load source or store target). Hence we have to consider the remaining cases $\rightarrow_{c'} \subset \rightarrow_{local}$ and $\rightarrow_{c'} \subset \rightarrow_{synchros}$.

Assume $\rightarrow_{c'} \subset \rightarrow_{local}$. This is the case for dependences $4,5,6$ and $10,11,12$. First, remark that dependences of $\rightarrow_{local}$ can only cross the last tiling hyperplane ($\vec{\tau}_2$ on figure 3.(a)). Indeed, the dependences crossing the remaining hyperplanes are all gathered in $\rightarrow_{i/o}$ by construction. Also, dependences between parallel processes are all gathered in $\rightarrow_{synchro}$ by construction. If the last tiling hyperplane does not break the fifo communication pattern, no splitting is required. Else, the splitting ends-up with a new dependence (and channel) partition: $\rightarrow_{c'} = \rightarrow_{c'}^{last} \uplus \rightarrow_{c'}^{tile}$. $\rightarrow_{c'}^{last}$ being the dependence instances crossing the last tiling hyperplane and $\rightarrow_{c'}^{tile}$ being the dependence instances lying inside a tile. Since in a tile, the schedule induces the same execution order as in the original DPN, $\rightarrow_{c'}^{tile} \subset \rightarrow_c$, and c is a FIFO with the original execution order, then $\rightarrow_{c'}^{tile}$ gives a FIFO channel (lemma 3.1). Similarly, $\rightarrow_{c'}^{last}$ links iterations from a producer tile (tile in the example) to the next consumer tile (tile on the example). Since $\rightarrow_{c'}^{last} \subset \rightarrow_c$ and fifo$(\rightarrow_c, \theta_{P_c}, \theta_{C_c})$, then fifo$(\rightarrow_{c'}^{last}, \theta_{P_c}, \theta_{C_c})$ by lemma 3.1. To conclude that $\rightarrow_{c'}^{last}$ gives a FIFO in the DPN, we need to check that the execution order on the *producer side* of $\rightarrow_{c'}^{last}$ is a subset of $\prec_{\theta_{P_c}}$ and that the execution order on the *consumer side* of $\rightarrow_{c'}^{last}$ is a subset of $\prec_{\theta_{C_c}}$. Provided the tile size is big enough, we can assume that starting from a given tile, the targets of $\rightarrow_{c'}^{last}$ belong the same tile (typically the next tile in the execution). By hypothesis, the execution order on a tile is a subset of the original execution order ($\hat{\theta}_S(\phi_1,\ldots,\phi_n,\vec{i}) = (\phi_1,\ldots,\phi_n,\vec{i})$ for each statement S). In particular, the execution order of the producers (resp. consumers) of $\rightarrow_{c'}^{last}$ is a subset of $\prec_{\theta_{P_c}}$ (resp. $\prec_{\theta_{C_c}}$). Hence the splitting ends-up with FIFO channels. This is a general apparatus which will be used to prove the next case.

Assume $\rightarrow_{c'} \subset \rightarrow_{synchro}$. On the example, this happens for dependences $7,8,9$. Remark that $\rightarrow_{c'}$ hold dependences from a producer tile (here tile 3) to tiles mapped to the same consumer process (here 4 and 6) by construction. Data sent to several consumer processes cannot flow through the same channel. Hence the only hyperplane which can separate consumer iteration is again the last tiling hyperplane. Either all the consumer iterations belong to the same tile (here 9), and we directly have a FIFO by lemma 3.1. Either consumer iterations are split by the last tiling hyperplane: $\rightarrow_{c'} = \rightarrow_{c'}^{last} \uplus \rightarrow_{c'}^{tile}$ (here 7,8). The same reasoning as in the previous paragraph allow to conclude that both cases can be implemented by a FIFO. Hence the splitting ends-up with FIFO channels.

This shows the completeness of the algorithm on DPN partitioning schema. Q.E.D.

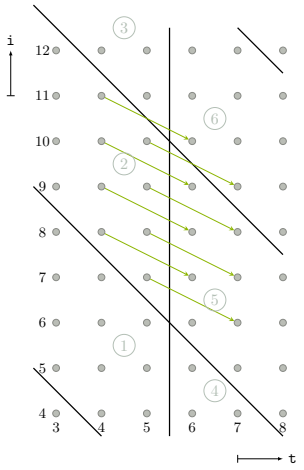


Figure 5: **Counter example on general PPN.** When dependences are too long, target iterations reproduce the tiling pattern ; hence breaking the FIFO pattern.

3.2 Limitations on general PPN

When the PPN does not follow the DPN partitioning, there is no guarantee that the algorithm will always succeed to recover FIFO. Figure 5 gives a counter example. We modify the kernel given on Figure 1.(a) to observe a single dependence $(t, i) \mapsto (t+2, i-1)$ on the compute domain ($\bullet$) and we keep the same loop tiling. The resulting PPN is almost like in (c) but with a single compute buffer (4 is kept to solve the dependence, $5, 6$ are removed). The algorithm will split the dependence in three parts: the dependences crossing the t hyperplane (partially depicted, in green), the dependences crossing the $t + i$ hyperplane (not depicted) and the dependence totally inside a tile (not depicted). In tile 2, iteration $(4, 11)$ is executed before iteration $(5, 7)$: $(4, 11) \prec_{\hat{\theta}} (5, 7)$. But the dependence target of iterations $(4, 11)$ and $(5, 7)$ are not executed in the same order because $(5, 7)$ targets tile 5 and $(4, 11)$ targets tile 6 (the next tile in the execution order). Hence the in-order property is not verified and the channel cannot be realized by a FIFO.

We observe that the algorithm works pretty well for short uniform dependences. However, when dependences are longer, the target operations reproduce the tile execution pattern, which prevents to find a FIFO. The same happens when the tile hyperplanes are "too skewed". Indeed, skewed hyperplanes can be viewed as orthogonal hyperplanes modulo a change of basis. When the hyperplanes are too skewed, the change of basis enlarge the dependence size which produce the same effect as in the counter-example. To summarize, the dependences must be uniform $\vec{i} \mapsto \vec{i} + \vec{d}$ and reasonably short (small $\|\vec{d}\|$). This means that tiling hyperplanes should not be too skewed. Indeed, skewed hyperplanes can always be viewed as orthogonal hyperplanes modulo a change of basis in the iteration space. The more skewed the hyperplanes are, the longer the dependence will be after the change of basis, thereby causing the same issue than on the counter example. The next section will, in particular, assess the capabilities of our algorithm to recover FIFO on general PPN.

4 Experimental Evaluation

This section presents the experimental results obtained on the benchmarks of the polyhedral community. We check the completeness of the algorithm on DPN process networks. Then, we assess the performances of the algorithm on PPN without DPN partitioning scheme. Finally, we show how much additional storage is produced by the algorithm.

Experimental Setup We have run the algorithm on the kernels of PolyBench/C v3.2 [10]. We have checked the completeness of the algorithm on PPN with DPN partitioning (Table 1), study the behavior of the algorithm on general PPN, without DPN partitioning (Table 2). Each kernel is tiled to reduce I/O while exposing parallelism [4] and translated both to a PPN and a DPN using our research compiler, Dcc (DPN C Compiler). On the DPN, the process are parallelized with a degree of $p = 2$ on each band dimension: for a kernel with a loop tiling of dimension n, each compute process is split in 2^{n-1} parallel processes.

Completeness on DPN Table 1 checks the completeness of our algorithm on PPN with DPN partitioning. For each kernel, column #buffers gives the total number of channels after applying our algorithm, column #fifos gives the total number of FIFO among these channels, the next columns provide the total size of FIFO channels and the total size of channels (unit: datum). Then, the next column assess the completeness of the algorithm. To do so, we recall the number of FIFO c in the original PPN before DPN partitioning and without tiling (#fifo basic). After tiling and DPN partitioning, each FIFO c is partitioned into several buffers c' ($\mu(c') = c$). Column #fifo passed gives the number of original FIFO c such that all target buffers c' are directly FIFO: $\#\{c \mid \forall c' : \mu(c') = c \Rightarrow c' \text{ is a FIFO}\}$. No splitting is required for these buffers c'. Column #fifo fail gives the number of original FIFO c such that at least one target buffer c' is not a FIFO: $\#\{c \mid \exists c' : \mu(c') = c \wedge c' \text{ is not a FIFO}\}$. If all the failing buffers c' can be split into FIFO by the algorithm, we say that c has been *restored*. The column #fifo restored count the restored FIFO c. Since the algorithm is complete, we expect to have always #fifo fail = #fifo restored. The last column gives the proportion of FIFO c not restored. We expect it to be always 0. As predicted, the results confirm the completeness of the algorithm on DPN partitioning: all the FIFO are restored.

Limits on PPN without DPN partitioning Table 2 shows how the algorithm can recover FIFO on a tiled PPN without the DPN partitioning. The columns have the same meaning as the table 1: columns #buffers and #fifos gives the total number of buffers (resp. fifos) after applying the algorithm. Column #fifo basic give the number of FIFO in the original untiled PPN: we basically want to recover all these fifos. Among these FIFO buffers: column #fifo passed gives the number of buffers which are still a FIFO after tiling and column #fifo fail gives the number of FIFO buffer broken by the tiling. These are the FIFO which need to be recovered by our algorithm. Among these broken FIFO: column #fifo restored gives the number of FIFO restored by the algorithm and column % fail gives the ratio of broken FIFO not restored by the algorithm. Since our algorithm is not complete on general PPN, we expect to find non-restored buffers. This happens for kernels `3mm`, `2mm`, `covariance`, `correlation`, `fdtd-2d`, `jacobi-2d`, `seidel-2d` and `heat-3d`. For kernels `3mm`, `2mm`, `covariance` and `correlation`, failures are due to the execution order into a tile, which did not reproduce the original execution order. This is inherently due to the way we derive the loop tiling. It could be fixed by imposing the intra tile execution order as prerequisite for the tiling algorithm. But then, other criteria (buffer size, throughput, etc) could be harmed: a trade-off needs to be found. For the remaining kernels: `fdtd-2d`, `jacobi-2d`, `seidel-2d` and `heat-3d`, failures are due to tiling hyperplanes which are too skewed. This fall

into the counter-example described in Section 3.2, it is an inherent limitation of the algorithm, and it cannot be fixed by playing on the schedule. The algorithm succeed to recover the all FIFO channels on a significant number of kernels (14 among 22): it happens that these kernels fulfill the conditions expected by the algorithm (short dependence, tiling hyperplanes not too skewed). Even on the "failing" kernels, the number of FIFO recovered is significant as well, though the algorithm is not complete: the only exception is the `heat-3d` kernel.

Kernel	#buffers	#fifos	total fifo size	total size	#fifo basic	#fifo passed	#fifo fail	#fifo restored	%fail
trmm	12	12	516	516	2	1	1	1	0
gemm	12	12	352	352	2	1	1	1	0
syrk	12	12	8200	8200	2	1	1	1	0
symm	30	30	1644	1644	6	5	1	1	0
gemver	15	13	4180	4196	4	3	1	1	0
gesummv	12	12	96	96	6	6	0	0	0
syr2k	12	12	8200	8200	2	1	1	1	0
lu	45	22	540	1284	3	0	3	3	0
trisolv	12	9	23	47	4	3	1	1	0
cholesky	44	31	801	1129	6	4	2	2	0
doitgen	32	32	12296	12296	3	2	1	1	0
bicg	12	12	536	536	4	2	2	2	0
mvt	8	8	36	36	2	0	2	2	0
3mm	53	43	5024	5664	6	3	3	3	0
2mm	34	28	1108	1492	4	2	2	2	0
covariance	45	24	542	1662	7	4	3	3	0
correlation	71	38	822	2038	13	9	4	4	0
fdtd-2d	120	120	45696	45696	12	5	7	7	0
jacobi-2d	123	123	10328	10328	10	2	8	8	0
seidel-2d	102	102	60564	60564	9	2	7	7	0
jacobi-1d	23	23	1358	1358	6	2	4	4	0
heat-3d	95	95	184864	184864	20	2	18	18	0

Table 1: Detailed results on PPN with DPN execution scheme. The algorithm is **complete on DPN**: all the FIFO were recovered (%fail = 0)

5 Conclusion

In this paper, we have studied an algorithm to reorganize the channels of a polyhedral process network to reveal more FIFO communication patterns. Specifically, the algorithm operates channels whose producer and consumer iteration domain has been partitioned by a loop tiling. We have proven the completeness of the algorithm on the DPN partitioning scheme. Also, we pointed out limitations of the algorithm on general PPN. Experimental results confirms the completeness of the algorithm on DPN as well as the limitations on general PPN. Even in that case, in a significant number of cases, the algorithm allows to recover the FIFO disabled by loop tiling with almost the same storage requirement. This means that our algorithm enables DPN as an intermediate representation for middle-end level polyhedral optimizations.

In the future, we plan to design a channel reorganization algorithm provably complete on general PPN with tiling, in the meaning that a FIFO channel will be recovered whatever the dependence size and the tiling used. Finally, we observed that FIFO failures can be avoided by playing with the intra-tile schedule. This may hinder other criteria (buffer size, throughput). We plan to study the trade-offs involved and to investigate how to constrain the scheduling algorithm while keeping acceptable trade-offs.

Kernel	#buffers	#fifos	total fifo size	total size	#fifo basic	#fifo passed	#fifo fail	#fifo restored	%fail
trmm	3	3	513	513	2	1	1	1	0
gemm	3	3	304	304	2	1	1	1	0
syrk	3	3	8194	8194	2	1	1	1	0
symm	9	9	821	821	6	3	3	3	0
gemver	8	7	4147	4163	4	2	2	2	0
gesummv	6	6	96	96	6	6	0	0	0
syr2k	3	3	8194	8194	2	1	1	1	0
lu	11	6	531	1091	3	0	3	3	0
trisolv	6	5	20	36	4	3	1	1	0
cholesky	12	9	789	1077	6	3	3	3	0
doitgen	4	4	12289	12289	3	2	1	1	0
bicg	6	6	532	532	4	2	2	2	0
mvt	4	4	34	34	2	0	2	2	0
3mm	10	6	1056	2848	6	2	4	2	50%
2mm	6	4	784	1296	4	2	2	1	50%
covariance	12	8	533	1317	7	4	3	2	33%
correlation	22	15	810	1642	13	9	4	3	25%
fdtd-2d	20	14	10054	36166	12	0	12	6	50%
jacobi-2d	12	4	1153	8385	10	0	10	2	80%
seidel-2d	12	6	803	49955	9	0	9	3	66%
jacobi-1d	13	13	1178	1178	6	1	5	5	0
heat-3d	20	0	0	148608	20	0	20	0	100%

Table 2: Detailed results on PPN. The algorithm is **not** complete on general PPN: some FIFO were not recovered (for some kernels, %fail $\neq 0$).

References

[1] Christophe Alias. Improving communication patterns in polyhedral process networks. In *Sixth International Workshop on High Performance Energy Efficient Embedded Systems (HIP3ES 2018)*.

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

[9] 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.

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

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

[12] Edwin Rijpkema, Ed F Deprettere, and Bart Kienhuis. Deriving process networks from nested loop algorithms. *Parallel Processing Letters*, 10(02n03):165–176, 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.

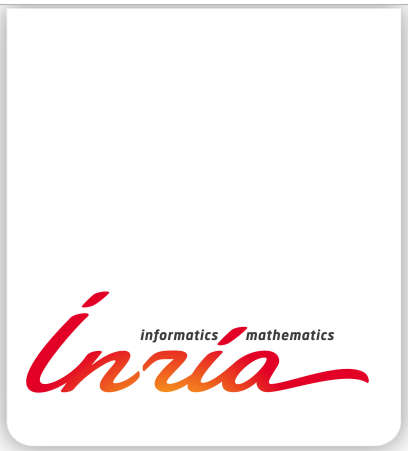
[16] Sven van Haastregt and Bart Kienhuis. Enabling automatic pipeline utilization improvement in polyhedral process network implementations. In *Application-Specific Systems, Architectures and Processors (ASAP), 2012 IEEE 23rd International Conference on*, pages 173–176. IEEE, 2012.

[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