ENS DE LYON

HABILITATION À DIRIGER DES RECHERCHES
de l'École Normale Supérieure de Lyon

Spécialité : Informatique

Présentée par :
Christophe ALIAS

Contributions to Program Optimization and High-Level Synthesis

Soutenue le 15 mai 2019 devant le jury composé de :

Isabelle Guérin-Lassous, professeur, Université Lyon 1, France	Présidente
Corinne Ancourt, maître de recherche, MINES ParisTech, France	Rapporteur
Sebastian Hack, professeur, University of Saarland, Allemagne	Rapporteur
Michelle Strout, professeur, University of Arizona, USA	Examinateur
Jürgen Teich, professeur, Friedrich-Alexander University, Allemagne	Rapporteur

Remerciements

En premier lieu, je tiens à remercier les membres de mon jury d'habilitation pour m'avoir fait d'honneur d'évaluer mes travaux de recherche. Je tiens à remercier mes rapporteurs, Corinne Ancourt, Sebastian Hack et Jürgen Teich pour le temps qu'ils ont consacré à évaluer mes travaux. Je tiens également à remercier Michelle Strout pour m'avoir fait l'honneur de venir de l'Arizona pour participer à mon jury. Enfin, je tiens à remercier Isabelle Guérin-Lassous pour avoir accepté de présider mon jury malgré son emploi du temps chargé.

Il y a 30 ans, j'écrivais candidement "plus tard, je veux être informaticien car j'aime l'informatique et les ordinateurs". Après avoir voulu successivement être physicien, puis mathématicien, j'ai finalement réalisé mon rêve de recherche initial à base d'ordinateurs... Comme quoi les ZX-81, Alice et autres TO7/70 mènent à tout. Vive le plan "informatique pour tous" !

Après l'obtention de mon doctorat en 2005, j'ai suis allé rouler ma bosse aux USA où j'ai découvert une culture différente et passionnante. Je tiens à remercier tous les collègues (stagiaires, doctorants, post-docs et permanents) rencontrés à Texas A&M University, puis Ohio State University pour leur accueil chaleureux et la passion communicative avec laquelle ils exercent leur métier. J'ai beaucoup appris à leur contact et pas seulement scientifiquement.

Je tiens ensuite à remercier Inria (l'Institut national de recherche en informatique et en automatique, pour les non-initiés) pour m'avoir ouvert ses portes en 2009 et pour m'avoir laissé la liberté académique nécessaire à la maturation des concepts et des résultats présentés dans ce document. Je tiens particulièrement à remercier Nicolas Jourdan et tout le service transfert et innovation du centre de recherche Grenoble/Rhône-Alpes pour leur soutien lors de la création de la start-up XtremLogic – un évenement important dans ma vie de chercheur qui me fait dire que j'invente ma part du monde numérique ; ainsi que nos assistantes d'équipe pour leur patience et leur compétence.

Je tiens également à remercier Frédéric Vivien et Yves Robert pour m'avoir fait comprendre que le temps était venu de passer l'HDR et la team CASH pour m'avoir soutenu pendant la rédaction de ce manuscrit et pour avoir cuisiné une partie de mon pot.

Special thanks à mes deux ex-doctorants : Alexandru Plesco et Guillaume Iooss qui m'ont beaucoup appris sur la nature de la direction de thèse. Je vous le redis : je suis fier de ce que nous avons accompli ensemble pendant toutes ces années. Il faut dire qu'entre Guillaume le théoricien et Alex l'architecte/entrepreneur il y a eu matière à explorer des directions différentes, qui finiront par se croiser j'en suis convaincu.

Enfin, je tiens à adresser un remerciement tout particulier à ma famille et mes amis pour le précieux soutien qu'ils m'ont apporté pendant toutes ces années. Ce document leur est dédié.

Contents

1 Introduction

DESIGNING and programming supercomputers is a major challenge for computer scientists with deep societal implications – both technological and geopolitical. Supercomputers make possible major advances in science and technology, as recently popularized by the – stunning – picture supercomputation of the massive stellar object at the center of the M87 galaxy. My researches concern the automation of *supercomputer design and programming*. Specifically, my contributions concern the design of *compilation models and algorithms* to generate efficient *software* and *hardware* for *supercomputing kernels*. Being at the interface between software and hardware, my work is deeply influenced and challenged by the trends on computer architectures and programming languages for high-performance computing, detailed in the two next sections.

This work has been granted by several positions. First, as a post-doc at ENS de Lyon, Texas A&M University and Ohio State University (2006–2009). Then, as a permanent research scientist at Inria (2009–). In 2014, I have co-founded a start-up, XtremLogic, with Alexandru Plesco, my first PhD student. Since then, I spend a part of my time (20%) at doing scientific advising around the compiler I have developed and transferred to XtremLogic under an Inria licence, where it is now a part of the production compiler.

This chapter is structured as follows. Sections 1.1 and 1.2 discusses the general context and the challenges addressed by my researches. Section 1.3 outlines the contributions presented in this document. Finally, Section 1.4 outlines of this document.

1.1 Architecture trends: the emergence of reconfigurable circuits

Power wall Since the end of Dennard scaling, transistors can no longer be miniaturized under a constant power density. As a consequence, parallel architectures have become ubiquitous and energy efficiency (measured in flop/J) has become a major issue whenever the energy budget is limited, typically for embedded systems and high-performance computers (HPC). The current trend is to explore the *trade-off between circuit genericity and energy efficiency*. The more specialized is a circuit, the less energy it consumes. At the two extremes, an ASIC (application specific integrated circuit) finely tuned to realize a specific function is more energy efficient than a mainstream processor (Xeon, etc). Hence the rise of *hardware accelerators* [87] (Xeon-Phi, GPU, FPGA) which equip most embedded systems and high-performance computers. This way, *computer architectures are now heterogeneous*, and new programming environments (operating system/runtime, language/compiler) are required to exploit entirely their computing capabilities, as discussed in the next section.

Recently, reconfigurable FPGA circuits [50] have appeared to be a competitive alternative to GPU [175] in the race for energy efficiency. FPGAs combine both flexibility of programmable chips and energy-efficiency of specialized hardware and appear as a natural solution. However, designing a circuit is far more complex than writing a C program. Disruptive compiler technologies are required to generate automatically a circuit configuration from an algorithmic description (High-level synthesis, HLS) [66]. A substantial part of our contributions concerns HLS for FPGA.

Memory wall Since the early days of Von Neuman architecture, *computers are slowed down by off-chip memory accesses*. Whatever the technology used, the memory bandwidth (BW) is a limiting factor and the gap with compute power is continuously widening. Memory hierarchy reduces the bandwidth requirement by *cutting the memory traffic*, providing the application exhibits enough data reuse. However, this requires to *reorganize the computation* to enforce data locality. This issue has been widely addressed by the compilation community in the last decades. But still, mainstream compilers are not able to automate this task.
The *roofline model* [168] is a visual performance model which identifies memory bottleneck by leveraging the *operational intensity* (OI) of the program – the average number of computation per memory access. It simply states that the performance is bounded by the memory accesses (OI × BW) and the peak performance PP of the processing unit (processor, GPU, etc): $\text{perf(OI)} \leq \max\{\text{OI} \times \text{BW}, \text{PP}\}$. It is an upper bound, which cannot be reached without hiding the memory latency by the computation. The operational intensity depends on the program execution order, which must be tuned so $\text{OI} \times \text{BW} \geq \text{PP}$, and then become *compute-bounded*. However, there is no magic: the operational intensity of a program is inherently bounded by the ratio between the volume of computation and the volume of input/output data. Hence, some programs might stay memory bounded whatever the execution order.
The roofline model is redefined in light of FPGA constraints [67]. With FPGA, the peak performance is no longer constant, it decreases with the operational intensity! Indeed, the more operational intensity we need, the more local memory we use, the less parallelization we get (since FPGA resources are limited), and finally the less performance we get. Hence, there exists *an* optimal operational intensity, which may require multiple iterations to be reached.
Clearly, writing a kernel to exploit the fine-grain parallelism of a processing unit while minimizing the memory traffic is out of reach for a programmer. This must be achieved by a compiler. In particular, there is a need for a *unified compilation model* which captures both parallelism and data transfers with the memory. We propose a new HLS-oriented compilation methodology towards this goal.

1.2 Dealing with parallelism and heterogeneity

Languages When it comes to program a parallel computer with hardware accelerators, the programmer is left to a myriad of standards with different levels of abstraction, allowing to control both *coarse-grain* and *fine-grain* parallelism. The general philosophy is to exploit coarse-grain parallelism *dynamically*: tasks are expressed at the langage-level, then a runtime system schedules the tasks on the processing units and rules the data transfers across processing units. Then, to exploit fine-grain parallelism *statically*: ideally, it should be up to the compiler to schedule the computations of a task to reveal parallelism and improve data locality.

▶ *Low-level models* like OpenCL [122] or CUDA [64] provide a software stack that exposes hardware accelerators through a low level stream abstraction. It is possible to express both *data parallelism* (into the kernels to be offloaded) and *task parallelism* (between the kernels). Two remarks: *data parallelism* was chosen to program GPUs. When the hardware accelerator is not a GPU and/or the kernel exhibits pipeline parallelism with fine-grain synchronisations, this model is simply not appropriate. This explains why the tentatives to port OpenCL to FPGA led to a failure. *Task parallelism* is entirely driven by the programmer at host level. Hence, it is entirely up to the programmer to place and to schedule the kernels and to issue the data transfers. With such a degree of freedom, these languages are perfect for source-level compiler optimizations (as an intermediate language); but particularly tedious for a programmer.

▶ *Kernel offloading models* such as OpenACC [65], OpenHMPP [78], or X10 [57] enrich a *sequential*, mainstream language (e.g. C, Java, Python) with directives to delimit the kernels offloaded to hardware accelerators and parallel programming constructs. The memory of the hardware accelerator and the data transfers are managed automatically before and after the kernel execution. Some approaches provide directives to tune the schedule of data transfers to overlap data transfers and computations. Extending a mainstream sequential programming langage with parallel constructions is appealing. However, it sometimes make unnatural the expression of the parallelism, which fits better in dataflow languages.

▸ *Task-based programming* models such as StarPU [33], Quark [173] or SMPSs [124] view the program as a composition of coarse-grain tasks to be scheduled at runtime in a dataflow fashion. The runtime analyses the data dependencies and builds a task graph (DAG). Then, the tasks are scheduled and mapped to processing units to improve the usual metrics (load balancing, data locality, data movements). This approach is generic enough to fit most heterogeneous systems [33]. However, FPGA raise additional challenges, which must be addressed:

– With FPGA, the silicon surface may be *reconfigured dynamically* to realize *several* computations in parallel. Hence, a FPGA is – *virtually* – several hardware accelerators whose features (e.g. silicon surface) may be tuned *dynamically*. This dynamicity of hardware must be modeled and taken into account by the runtime system. As well, a virtualization layer is clearly required.
– Unlike software parallelization, the nature of FPGA requires that many configuration decisions must be made at *compile time*. It is our belief that these configurations will require a double-staged process with an offline compilation *keeping some parameters symbolic* (like the size of the processor array in [147]), coupled with a runtime which plays on these parameters to reconfigure the circuit. This raise the challenge of *partial compilation*, as discussed later.

Libraries With optimized HPC libraries, the programmer can exploit parallelism without tedious parallel programming. At first glance, a kernel optimized by hand by an expert will always outperform an automated compiler optimization. For instance, the Basic Linear Algebra Subprograms (BLAS) [112] provides optimized building blocks for performing basic vector and matrix operations. Many HPC libraries were designed on top of BLAS (LAPACK [32], ScaLAPACK [61], PLASMA and MAGMA [30]). In particular, PLASMA addresses architectures with multicore CPU ; and MAGMA addresses heterogeneous architectures with CPU and GPU. Both rely on task decomposition of the routines into BLAS routines. This decomposition into subcomputations is wrongly referred to as a *tiling* in the litterature. It is actually a *semantic tiling* (the associativity of the computation is no longer the same). We address the formalization and the automation of semantic tiling in this document.
Adaptative libraries (ATLAS [167], FFTW [86], PhiPAC [49]) apply an *autotuning* at installation time by varying parameters that affect performances (typically the blocking size). This actually relies on a telescopic approach with a *partial compilation*: the compiled code contains optimization parameters which may be tuned at installation time (*adaptative libraries*), and even at runtime (*iterative compilation*). This way, the library vendor does not have to provide the source code and to let several recompilations at installation time, which takes times, in addition to be risky from a business perspective. This clearly raises the challenge of partial compilation: how to parametrize a program transformation, and how to let the parameter(s) survive the compilation. We address this challenge in this document.
However, there is no magic: a pure library-driven approach raises many challenges:

– The optimization coverage is bounded to library functions, which excludes *de facto* global optimizations
– The choice of computations to take from a library is left to the programmer, which is guided by algorithmic considerations. However, non-trivial factorizations might perform better.

Learning and using a new library remains fastidious and may refrain the adoption of a new standard: old habits die hard. Hence, we believe that the compiler should *automate the refactoring of a program* with library functions, in the same way as an instruction selector selects the best set of instructions to program a processor. We propose a complete system based on semantic tiling to automate this task.

Compilers While runtimes exploit *coarse-grain* parallelism, compilers generate the machine code offloaded to processing units while revealing *fine-grain* parallelism. Compilers enhances programmer productivity and code portability when compiling from a mainstream langage. However, compiler optimizations are still fragile and highly depend on the shape of the source code. This is because most compiler analysis (dependences, alias) are locked by undecidable problems, which push towards *conservative approximations* that limit there impact. Hence the emergence of *domain-specific compilers*, which trade the genericity for precision.

A typical example is the *polyhedral model*, a theoretical framework to reason about kernels with nested `for` loops and to restructure the code, mainly for extracting parallelism while ensuring data locality. Since it is a source-to-source approach, it can be connected to any mainstream compiler. Thanks to the emergence of robust algorithms and tools for getting the code *to* and *from* the polyhedral representation, polyhedral code optimizations are progressively adopted by industry and transfered to production compilers.

With FPGA, the compiler must produce a *circuit configuration* from a high-level algorithmic description. The process consists into two steps: first, *high-level synthesis* compiles the source code to a circuit expressed in a hardware description language. Then, *synthesis* compiles the circuit to a binary file describing the FPGA configuration (*bitstream*). That bitstream is uploaded to the FPGA chip when the runtime proceeds to the configuration. High-level synthesis tools (VivadoHLS [165], OpenCL SDK [122], Xilinx SDAccel [85]) are now mature enough to produce circuits with an optimized internal structure thanks to efficient scheduling techniques, resource sharing, and finite state machines generation. This enhance the spreading of FPGA across software developers, which can take advantage of the speed and the energy efficiency of FPGA without an hardware expertise.

However, HLS tools suffer from many limitations.

– *The input langage lacks a clear semantics*. It is usually presented as *locally sequential, globally dataflow*. However interprocess synchronizations often assumes a DAG of process (e.g. with Altera/OpenCL, C2H, CatapultC). Hence, it is legal for the HLS scheduler to change the order of channels reads and writes. This limitation forbids dataflow programs with cycles between processes, which arises frequently with fine-grain processes.

– *The I/O of the circuit are hard to program*, in particular when it comes to optimize the communications between the circuit and an off-chip memory. So far, the memory hierarchy must be implemented by the programmer (local memory sizing and allocation, data transfers scheduling). We propose a complete system to address this challenge.

– *Fine-grain parallelism is still extracted with old fashioned techniques* like loop unrolling and loop pipelining. Hence the need to push high-level parallelization techniques from the polyhedral model to HLS tools. This is the goal of our work on HLS.

1.3 Contributions

Our contributions are twofold. We propose program transformations for automatic parallelization, and more generally program optimization. Then, we propose models and algorithms for high-level synthesis of circuit configurations for FPGA chips.

On the first part, we propose several extensions of *loop tiling*, a fundamental program transformation in automatic parallelization (**generalized loop tiling**). In particular, we address *parametric loop tiling*, with opens the way to the *partial compilation challenge*. Then, **algorithm recognition** automates the *refactoring of a program with a performance library*.

On the second part, we automate the synthesis of data spilling between an FPGA chip and an off-chip memory while hiding the communications by computation (**communication synthesis**). Then, we propose an HLS model which explicits data spilling and fine-grain parallelism (**data-aware process networks**). Our HLS algorithms have been implemented and transferred to the XtremLogic startup, where they are used in the production compiler.

Generalized loop tiling Many program transformations require dividing the program into *blocks* of computation. For automatic parallelization, the blocks serve as subcomputations to be executed in parallel. For memory traffic minimization, the blocks serve as *reuse units*: into a reuse unit, the data are communicated via a local memory; between two reuse units, the data are communicated through the remote memory. Loop tiling [55, 68, 99, 111, 142, 149, 171] is a very natural and standard way to obtain such a partitioning.
We propose to extend loop tiling in two directions. First, we allow to parametrize the tile size *with a scaling parameter* [26], for *any convex polytopic tile shape* [25]. The scaling parameter survives the compilation, and may be tuned at installation time and even at execution time, which opens the way to *partial compilation*. The main novelty is to express this transformation into the polyhedral framework, and not as a post-processing step. The ability to compose with other polyhedral transformations opens new perspectives for polyhedral *partial compilation*. Then, we show how our tiling transformation might be extended to be a *semantic tiling* [23], a transformation which increases the granularity of operators (scalar → matrix). Semantic tiling leverages associativity and commutativity of reduction operators to partition the computation. As we have seen, semantic tiling is usually confused with tiling by the numerical community. Hence, it is already widely applied by hand for supercomputing: it typically defines the tasks in task-level programming models. When the computations are identified (see our next contribution), we may automate the refactoring with performance libraries. Our contributions on semantic tiling and its application to algorithm recognition are under publication. Hence a sufficient level of detail is provided in this document. So far, a complete description may be found in the PhD thesis of Guillaume Iooss [98], in the context of which all this work (generalized loop tiling & algorithm recognition) was developed.

Algorithm recognition A compiler optimization will never replace a good algorithm, hence the idea to recognize algorithm instances in a program and to substitute them by an optimized version. In particular, this automates the refactoring of a program with a *performance library*, and free the programmer from this tedious task. Algorithm recognition raises many challenges: *program slicing* (divide the program into subcomputations), *program equivalence* (compare with an algorithm) and *performance evaluation* (is the substitution beneficial?). During my PhD thesis, I have addressed the recognition of algorithm templates – functions with first-order variables – into programs [3, 6, 7, 9] and the substitution by a call to a performance library [2, 8]. The slicing was driven by a relaxed version of the equivalence checking algorithm. However, the subcomputations were not clearly separated from the remainder of the program, which complicates the generation of the final code and tends to produce inefficient factorizations.
We address these issues by leveraging *semantic loop tiling* to split the program into *slices* to be compared with the templates. Though the recognition is more focused, it provides a natural algorithmic partitioning and it eases the substitution by a call to the library. Our algorithm recognition system applies iteratively a new *template matching algorithm*, which process the slices (tiles) until all the library idioms have been recognized. We use a relaxed notion of template where the unknowns are simply the inputs. This simplifies the process to find a composition of algorithms: match a template, and apply recursively the recognition on the template inputs.
As stated in the previous paragraph, this work was developed in the context of the PhD thesis of Guillaume Iooss where all the details may be found [98]. In addition, we addressed the equivalence checking of programs with explicit reductions [24]. It is another way to tackle semantic equivalence of programs. It was actually one of our first contributions towards semantic tiling. Finally, we this algorithm was not used in our system: we prefered a two-step approach with a correct-by-construction semantic transformation (semantic tiling), followed by template matching.

Communication synthesis FPGAs come with pretty few local memory (8 MB on an Intel Stratix 10 GX1150 FPGA). This does not fit supercomputing kernels requirements, which often feature large memory footprint. Hence, compiler techniques are required to *spill* efficiently the data to an off-chip memory, as would do a register allocation, and to schedule the operations so the *computation hides the communications*. All the more so due to the small bandwidth of off-chip memories found on FPGA boards (typically a DDR4 memory with 25 GB/s).

We propose a complete HLS algorithm to automate the data spilling to the off-chip memory as a source-to-source transformation in front of an HLS tool, which serves as an hardware backend. First, we propose a template of architecture for optimized data transfers, and we show how this architecture can be specified as a C program in front of an HLS tool [12, 13]. Then, we propose an algorithm to schedule the data transfers [14, 15] so that the resulting application is highly-optimized, with minimized off-chip memory traffic. Communications are covered by computations by double-buffering the subcomputations revealed by a loop tiling. This way, the maximal throughput is reachable providing a sufficient parallelization. By playing on the tiling we can simply explore several trade-offs. *Local memory size for memory traffic:* the bigger is the local memory, the less memory traffic is issued. *Local memory size for peak performance:* the bigger is the local memory, the less FPGA resources remain for the computation. Our compilation model gives a general methodology to *explore the FPGA roofline model* for the best configuration.

These results were obtained in the context of the PhD thesis of Alexandru Plesco [126] and have opened the way to a fruitful collaboration towards a complete polyhedral HLS approach in the context of the XtremLogic start-up, as described in the next paragraph.

Data-aware process networks We propose a complete polyhedral-powered approach for high-level synthesis (HLS) of supercomputing kernels to FPGA. We cross fertilize polyhedral process networks (PPN) [160] with our communication synthesis approach to propose a new dataflow intermediate representation which explicits both parallelism and data spilling to the off-chip memory: the *data-aware process networks (DPN)*. We show how DPN and PPN might be view as instances of *regular process networks (RPN)*, a general compilation-oriented dataflow model of computation. RPN induces a *general methodology* of automatic parallelization, that we believe to be appropriate for HLS of supercomputing kernels to FPGA.

We propose algorithms for both *front-end* C → DPN and *back-end* DPN → circuit. We addressed the compilation of the *process control* (control compaction [21], single process scheduling [17, 18]), the compilation of *channels* (typing [1, 4], allocation/sizing [5]) and the *synchronization of communications* (Xtremlogic/Inria patent [19]). We have developed entirely the *front-end* C → DPN, which was transferred to the XtremLogic startup, that we co-founded in 2014 with Alexandru Plesco, three years after his defense. XtremLogic is in charge of developing the *back-end* DPN → circuit. Though the back-end is still under development today (and then no global, system-level, results are available yet), we obtained a substantial number of results, which apply to DPN while being general enough to apply to other contexts.

1.4 Outline of the document

This manuscript is structured as follows:

– Chapter 2 presents our contributions to *generalized loop tiling*. We show how polyhedral loop tiling can be extended to *any polytopic shape with a scaling parameter*, while staying in the polyhedral representation. Then we propose an algorithm for *semantic tiling*.
  ▷ *Context: Guillaume Iooss' PhD.*

– Chapter 3 presents our contributions to *algorithm recognition*, based on semantic tiling. Aside, we propose an algorithm program equivalence in the presence of reductions.
  ▷ *Context: Guillaume Iooss' PhD*

– Chapter 4 presents our contributions to *compile off-chip memory accesses in HLS*. We propose a model of architecture to rule the I/O with the memory, an algorithm to schedule the memory transfers while avoiding redondant communications (*communication coalescing*). This works serve as a basis to the high-level synthesis methodology presented in the next chapter.
  ▷ *Context: Alexandru Plesco's PhD.*

– Chapter 5 presents an high-level synthesis methodology based on data-aware process networks (DPN), a model of computation which explicit the parallelism and memory transfers. We discuss several compilation algorithms related to DPN, addressing the generation of process control, channels and synchronizations
  ▷ *Context: start-up project with Alexandru Plesco.*

– Chapter 6 concludes and draws research perspectives.

– Appendix A outlines our further contributions.

Part I

Program Transformations for Automatic Parallelization

2 Generalized Loop Tiling

Loop tiling [55, 68, 99, 111, 142, 149, 171] is a standard loop transformation which divides the computation into subcomputations by grouping the loop iterations into *tiles* to be executed atomically. With loop tiling, an application might be distributed across the nodes of a parallel computer, while tuning the ratio computation/communication to fit the architecture balance. Loop tiling is probably the most appropriate and powerful program transformation for automatic parallelization. State-of-the-art tiling algorithms leverage the *polyhedral model* [81, 82, 83, 135, 136], a mathematical formalism to develop compiler optimizations. The polyhedral model provides a compact, mathematical representation of both programs and their transformations. However, many locks refrain to exploit the full potential of loop tiling:

- *Tile size* is a major parameter. In automatic parallelization, it impacts directly the parallelism, the operational intensity and the local memory size. In particular, trade-offs might be explored by tuning the tile size (typically, communication volume for local memory size). Hence, it is desirable to have a *symbolic* compilation where parameters survive the compilation and can be tuned afterwards, without having to recompile [86, 132, 167]. However, parametric tiling does not fit directly in the polyhedral model, it is usually implemented directly in back-end code generators of polyhedral compilers [39, 107, 139]. How to express the parametric tiling *in* the polyhedral model and make possible to *compose* with further polyhedral transformations is still an open problem today.
- *Tile shape*. Historically, loop tiling was defined as a direct partition of the iteration space with hyperrectangles [171] and hyper-parallelepiped, defined by its boundary hyperplanes [99, 172], to preserve the atomicity. Trapezoidal tiling was introduced to allow concurrent start [108, 137, 176], at the price of redundant computations. Then, redundant computations were removed by using diamonds [52] and hexagons [91, 138]. Actually, any covering of the iteration space can define a tiling, there is no need to have a partition (redundant computations are allowed) or tiles defined as convex polyhedra. How to find the best tile shape for a given optimization criteria, how to transform a program with an arbitrary polyhedral tile shape while staying in the polyhedral model, then how to produce a reasonable loop nest out of it are still challenging problems today.

Summary and outline This chapter summarizes the contributions made to loop tiling. All these results were obtained in the context of the PhD thesis of Guillaume Iooss [98]. Section 2.1 outlines the polyhedral model and defines the tiling transformation. Then we present our contributions:

- Section 2.2 summarizes our work on hyperrectangular tiling *with a scaling parameter*. We show that this transformation fits in the polyhedral model under certain restrictions [26].
- Section 2.3 generalizes the results of section 2.2 *to any polytopic tile shape* (e.g. diamond, hexagon). Likewise, it is a polyhedral transformation under the same restrictions [25].
- Section 2.4 extends our parametric tiling transformation to a *semantic tiling* [23], a transformation which increases the granularity of operators (scalar → matrix). This part is described in the PhD thesis of Guillaume Iooss and is under publication with some of the contributions described in Chapter 3.

2.1 Polyhedral model

This section introduces the polyhedral model (Section 2.1.1) and our program model (Section 2.1.2). Then, we define the polyhedral intermediate representation on which polyhedral transformations are applied (Section 2.1.3). Finally, we introduce polyhedral transformations and show how loop tiling is expressed in the polyhedral model (Section 2.1.4). In particular, we describe the locks addressed in this Chapter.

2.1.1 Polyhedral compilers

The *polyhedral model* [81, 82, 83, 135, 136] is a well established framework to design automatic parallelizers and compiler optimizations. It abstracts loop iterations as a union of convex polyhedra and data access as affine functions. This way, precise—iteration-level—analyses can be designed thanks to geometric operations and integer linear programming: exact array dataflow analysis [81], scheduling [82, 83], memory allocation [70, 133] or code generation [31, 43, 134]. A program can fit in the polyhedral model, given an appropriate abstraction. We will describe in the next section a class of programs which fits directly in the polyhedral model.
A *polyhedral compiler* is usually a source-to-source compiler, that transforms a source program to optimize various criteria: data locality [43], parallelism [83], a combination of locality and parallelism [53, 170] or memory footprint [70] to name a few. The input language is usually imperative (C like) [5, 53]. It may also be an equational language [117, 174]. Today, polyhedral optimizations can also be found in the heart of production compilers [20, 92, 128, 152]. A polyhedral compiler follows a standard structure. A *front-end* parses the source program, identifies the regions that are amenable to polyhedral analysis, and builds an intermediate *polyhedral representation* using array dataflow analysis [81]. This representation is typically an iteration-level dependence graph, where a node represents a polyhedral set of iterations (e.g., a statement and its enclosing loops), and edges represent affine relations between source and destination polyhedra (e.g., dependence functions). Then, *polyhedral transformations* are applied on the representation. Because of the *closure properties* of the polyhedral representation [117, 136] the resulting program remains polyhedral, and transformations can be composed arbitrarily. Finally, a *polyhedral back-end* generates the optimized output program from the polyhedral representation [31, 43, 134].

2.1.2 Program model

Polyhedral programs A polyhedral program consists of nested `for` loops and `if` conditions manipulating arrays and scalar variables, which satisfies an *affinity* property: loop bounds, `if` conditions, and array access functions are *affine expressions* of surrounding loops counters and structure parameters. With polyhedral programs, the control is *static*: it only depends on the input size (the structure parameters), not the input values. This way, loop iterations and array accesses might represented statically, and with decidable objects (Presburger sets) thanks to the affinity property. Polyhedral programs covers an important class of compute- and data-intensive loop kernels usually found in linear algebra and signal processing applications [41, 129].
Figure 2.1.(a) depicts a polyhedral program computing the product of two polynomials given the arrays of coefficients `a` and `b` for each monomial.
With polyhedral programs, each iteration of a loop nest is uniquely represented by the vector of surrounding loop counters $\vec{i}$. The execution of a program statement S at iteration $\vec{i}$ is denoted by $\langle S, \vec{i} \rangle$. The set $\mathscr{D}_S$ of iteration vectors is called the *iteration domain* of S.
Figure 2.1.(b) depicts the iteration domains $\mathscr{D}_S$ and $\mathscr{D}_T$ (grey points). When the loop steps are equal to one, an iteration domain is the set of *integral points* satisfying the affine constraints induced by the enclosing loop bounds and tests. Such a set is called an *integer polyhedron*:

Definition 2.1 (integer polyhedron)**.** *An* integer polyhedron *is a set $\mathscr{P}$ of integral points $\vec{x} \in \mathbb{Z}^n$ satisfying a conjunction of affine constraints $\vec{a}_1 \cdot \vec{x} + b_1 \geq 0 \wedge \ldots \wedge \vec{a}_p \cdot \vec{x} + b_p \geq 0$ where $\vec{a}_i \in \mathbb{Z}^n$ and $b_i \in \mathbb{Z}$ for each i. If A is the matrix whose rows are $\vec{a}_1, \ldots, \vec{a}_p$, the* matrix representation *of $\mathscr{P}$ is: $\{\vec{x} \in \mathbb{Z}^n, A\vec{x} + \vec{b} \geq \vec{0}\}$.*

An iteration domain may depend on *parameters* (typically, the array size N on the example). Hence, we extend the definition by considering some variables as parameters:

Definition 2.2 (parametric integer polyhedron)**.** *A* parametric integer polyhedron *is a set $\mathscr{P}$ of integral points $\vec{x} \in \mathbb{Z}^n$ satisfying $A\vec{x} + B\vec{n} + \vec{c} \geq \vec{0}$ where A and B are integral matrices and $c \in \mathbb{Z}^p$. $\vec{n} \in \mathscr{C} \subseteq \mathbb{Z}^q$ is the vector of* parameters*. $\mathscr{C}$ is called the* context domain.

A parametric integer polyhedron $\mathscr{P}$ must be seen as an application $\mathscr{C} \rightarrow 2^{\mathbb{Z}^n}, \vec{n} \mapsto \mathscr{P}(\vec{n})$, and a point $\vec{x}$ of $\mathscr{P}$ as an application $\mathscr{C} \rightarrow \mathbb{Z}^n, \vec{n} \mapsto \vec{x}(\vec{n})$. It is usually a piecewise affine function discussing the value of $\vec{x}(\vec{n})$ depending on the parameters $\vec{n}$. For instance, the first iteration of T reading $c[2]$ is the point: $N \mapsto (N \geq 2 : (0,2), N = 1 : (1,1), N < 1 : \perp)$. The notation $\perp$ means that the point is not defined. We avoid that mapping notation when the parameters are clear from the context. The ability to reason on non-constant—parametric—execution traces is a essential feature of the polyhedral model.

Beyond polyhedral programs In general, any program may fit in the polyhedral model provided a *sound abstraction* to the polyhedral representation (a *front-end* program → polyhedral representation). In that case, *sound* means that the abstraction should convey enough information to enforce the correctness of polyhedral transformation. The compiler should also feature a code generator from the polyhedral representation (*back-end*) able to convey dynamic control and memory references. This way, non-polyhedral constructs (non affine array accesses, while loop, early exits) might be handled [10, 47], at the price of precision: for instance, parallelization opportunities might be missed because of dependence over-approximation.
Further polyhedral approaches rely on mixed static/dynamic compilation to cope with dynamic constructs. In particular, the *sparse polyhedral model* [145, 158] defines composable inspector/executor schemes using the polyhedral formalism (affine relations) extended with uninterpreted functions to cope with irregularities (non affine loop bounds and array indices).
In [10], we proposed an application of the polyhedral scheduling to checking the termination of programs with while loops. The control flow graph was encoded in the polyhedral model in such a way that polyhedral scheduling gives a ranking function (a loop variant), thereby proving the termination. An extended summary of this work may be found in Appendix A.

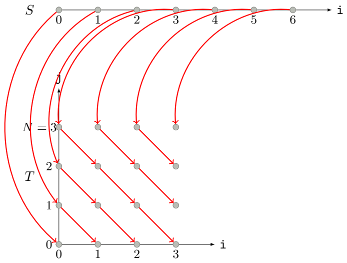
Execution order The *sequential execution order* $\prec$ is locally defined by the lexicographic ordering: $\langle T, i, j\rangle \prec \langle T, i', j'\rangle$ iff $(i, j) \ll (i', j')$, where $(i, j) \ll (i', j')$ iff $i < i'$, or $i = i' \wedge j < j'$. Between instances of different statements $\langle S_1, \vec{i}\rangle$ and $\langle S_2, \vec{j}\rangle$, we consider the restriction of $\vec{i}$ and $\vec{j}$ to the common loop counters: $\vec{i}'$ and $\vec{j}'$: $\langle S_1, \vec{i}\rangle \prec \langle S_2, \vec{j}\rangle$ iff $\vec{i}' \ll \vec{j}'$, or $\vec{i}' = \vec{j}'$ and S_1 is before S_2 in the text of the kernel. Here, we always have $\langle S, i\rangle \prec \langle T, i', j'\rangle$ as S and T do not share any loop and S is before T in the text of the kernel.

Lexicographic optimization In the polyhedral model, we often need to compute the first/last iteration satisfying a property. Computing the lexicographic minima/maxima of an integer polyhedron can be done with standard ILP algorithms. The simplex algorithm with Gomory cuts has been extended to parametric integer polyhedra [80]. The result is a point in the meaning defined above: a piecewise function discussing the result depending on the parameters. For instance, the first iteration of T reading c[2] can be expressed by the *parametric* integer program: $\min_{\ll}\{(i, j) \mid i + j = 2 \wedge 0 \leq i, j \leq N\}$. In general, this function is piecewise *quasi-affine*, meaning that the pieces and the functions may involve integer divisions and modulo. This can still be expressed with affine forms if existential quantifiers are allowed, hence the notion of *polyhedral domain* [159]:

```
    for i := 0 to 2N
S:    c[i] := 0;

    for i := 0 to N
      for j := 0 to N
T:      c[i+j] := c[i+j] + a[i]*b[j];
```

(a) Product of polynomials



(b) Iteration domains and direct dependences

$$S[i] = \; 0 \le i \le 2N : 0$$

$$T[i,j] = \begin{cases} i = 0 \vee j = N : \\ \quad S[i+j] + a[i] * b[j] \\ 2 \le i \le N \wedge 1 \le j \le N-1 : \\ \quad T[i-1, j+1] + a[i] * b[j] \end{cases}$$

(c) SARE

Figure 2.1 – Polyhedral program (a), execution trace (iteration domains) and direct dependences (b), and corresponding system of affine recurrence equations (SARE) which captures the dataflow dependences and the computation (c). The SARE is a possible intermediate representation in a polyhedral compiler

Definition 2.3 (polyhedral domain)**.** *A* polyhedral domain *is defined as:*

$$\left\{ \vec{x} \in \mathbb{Z}^n \mid \bigvee_{i=1}^{p} \exists \vec{\alpha}_i \in \mathbb{Z}^{m_i} : A_i \vec{x} + B_i \vec{\alpha}_i + C_i \vec{n} + \vec{d}_i \ge \vec{0} \right\}$$

Where A_i, B_i and C_i are integral matrices and $\vec{d}_i \in \mathbb{Z}^n$. Again, $\vec{n} \in \mathscr{C} \subseteq \mathbb{Z}^q$ is the vector of parameters *and $\mathscr{C}$ is called the* context domain.

For example, the set of even integers $\{x \in \mathbb{Z} \mid \exists y \in \mathbb{Z} : x = 2y\}$ is a polyhedral domain. We point out that polyhedral domains are exactly Presburger definable sets: $\mathscr{D}$ is a polyhedral domain iff there exists a Presburger formula ϕ such that $\mathscr{D} = \{\vec{x} \in \mathbb{Z}^n \mid \vec{x} \models \phi\}$. Emptiness checking, geometric operations (projection, union, intersection) and lexicographic minima/maxima are computable over polyhedral domains [95, 131, 159], though the algorithms are more expensive and require more machinery than for integer polyhedra. The class of polyhedral domains is closed under usual operations: projection, union, intersection. *Integer lattices* are particular polyhedral domains, required to define tiling:

Definition 2.4 (integer lattice)**.** *An* integer lattice *is the sub-group of* $(\mathbb{Z}^n, +)$ *generated by vectors $u_1, \ldots, u_p$ of $\mathbb{Z}^n$:*

$$\mathscr{L}(u_1, \ldots, u_p) = \{\vec{z} \in \mathbb{Z}^n \mid \exists \vec{\lambda} \in \mathbb{Z}^p : \vec{z} = \lambda_1 \vec{u}_1 + \ldots + \lambda_p \vec{u}_p\} = L\mathbb{Z}^n$$

where L is the matrix with column vectors $u_1, \ldots, u_p$.

2.1.3 Polyhedral intermediate representations

We now discuss briefly the intermediate representations used in a polyhedral compiler. As always, several representations are possible. They all capture the flow of data dependences, required for a sound reorganization of the computations.

Dependences There exists a dependence $\langle S_1, \vec{i} \rangle \to \langle S_2, \vec{j} \rangle$ iff $\langle S_1, \vec{i} \rangle \prec \langle S_2, \vec{j} \rangle$ and both operations access (write or read) the same data. Depending on the access type, we classify the dependences in *flow dependence* (write then read), *anti dependences* (read then write), *output dependences* (write then write) and *read dependences* (read then read). By nature, polyhedral transformations as loop tiling will change the read order, hence read dependences are ignored. From a HLS perspective, read dependences should preserved when the data come with a prescribed order e.g. when the data are read from an external FIFO. Also, anti- and output- dependences may be removed by array expansion [81]. Hence we focus on flow dependences $\to_{\text{flow}}$. Specifically, we consider producer/consumer dependences, relating the production of a value to its read, called *direct dependences*:

$$\to_{\text{pc}} = \{x_{max} \to_{\text{flow}} y \mid x_{max} = \max_{\prec}\{x,\ x \to_{\text{flow}} y\}\}$$

For each read y, we keep the last write x_{max} defining y. This is solved with lexicographic optimization [81]. Direct dependences are depicted in red in Figure 2.1.(b). We have: $\langle S, i+j\rangle \to_{\text{pc}} \langle T, i, j\rangle$ if $i = 0$ or $j = N$ and $\langle T, i-1, j+1\rangle \to_{\text{pc}} \langle T, i, j\rangle$ if $2 \le i \le N$ and $1 \le j \le N-1$. Modulo a proper encoding of S and T with integers, dependences relations may be represented by polyhedral relations:

Definition 2.5 (polyhedral relation)**.** *Given a polyhedral set* $\{\vec{x} \in \mathbb{Z}^{n+p} \mid \phi(\vec{x}, \vec{n})\}$ *where* $\vec{n}$ *is the vector of parameters, we define a* polyhedral relation *by distinguishing input variables* $x_1, \dots x_n$ *from output variables* $x_{n+1}, \dots x_{n+p}$*:*

$$(x_1, \dots, x_n) \to (x_{n+1}, \dots, x_{n+p}) \quad s.t. \quad \phi(\vec{x}, \vec{n})$$

If we write $\to$ the direct dependence relation, and we define $x \hat{\to} (y, k)$ iff $x \to y$ on the k-th read of y, then $\sigma = \hat{\to}^{-1}$ is a piecewise-affine function, called the *source function* [81]. For instance, $\sigma(\langle T, i, j\rangle, 1)$ gives the execution instance producing the value consumed by the read c[i+j] of statement T:

$$\sigma(\langle T, i, j\rangle, 1) = \begin{cases} \langle T, i-1, j+1\rangle \text{ if } i \ge 1 \wedge j < N \\ \langle S, i+j\rangle \text{ else} \end{cases}$$

When σ is expressed with translations (clauses $i \in \mathcal{D} : \langle S, \vec{i} + \vec{t}\rangle$), the dependences are said to be *uniform*. $\to_{pc}$ and σ are the same representation of the flow of dependences of the program. They naturally lead to an equational representation of polyhedral programs.

Systems of Affine Recurrence Equations (SARE) A SARE is an *equational normal form* of the program, which focuses on the computation itself and abstracts away the storage allocation and the execution order. A SARE is a collection of recurrence equations between single-assignment arrays, where equation domains are polyhedral domains and array index functions are affine:

Definition 2.6 (SARE)**.** *A SARE is a list of equations of the form*

$$\mathtt{Var}[\vec{i}] \;=\; \begin{cases} \dots \\ \vec{i} \in \mathcal{D}_k : \mathtt{Expr}_k \\ \dots \end{cases}$$

where the $\mathcal{D}_k$ *are disjoint* polyhedral domains*, and where:*
- `Var` *is a* variable*, is defined over a polyhedral domain* $\mathcal{D}$ *and is either an input, an output or a local variable*
- `Expr` *is an* expression*, and can be either:*
  - *A variable* `Var[f(`$\vec{i}$`)]` *where* f *is an affine function*
  - *A constant* `Const`*,*
  - *An affine function of the indices* $f(\vec{i})$
  - *An operation* `Op(Expr`$_1$`, ..., Expr`$_k$`)` *of arity* k *(i.e., the operation has* k *arguments)*

Moreover, we assume that `Expr` *depends strictly on all of its arguments (i.e., the value of each of the argument impacts the value of* `Expr`*).*

Figure 2.1.(c) gives the SARE representation for the program given in (a). We point out that a SARE is nothing more than a finite representation of direct dependences [81]. Somehow, it might be viewed as a reduced dependence graph whose nodes are arrays and edges are driven by source functions of read arrays. SARE arrays are completely expanded to ensure the single assignment property: each array cell is uniquely defined.

The representation of computations by systems of recurrence equations was pioneered in 1967 by Karp, Miller and Winograd [104], which paved the way to the polyhedral model. Since their *uniform recurrence equation*, more general representations were addressed. *System* of uniform recurrence equations (with several equations and uniform dependences) were addressed by the systolic community [46, 135, 136, 150] where they are sometimes referred to as *piecewise regular algorithms*. Then, SAREs were adopted to address programs with non-uniform dependences. SAREs were enforced as a standard program representation for scheduling and automatic parallelization as soon as Feautrier found a *front-end* algorithm from polyhedral programs based on the computation of the source function σ [81].
In a slightly more general form of SARE, we allow *hierarchical SAREs* where it is possible to "call" another SARE , called *subsystem* [74] and to use a *reduction* operator, as defined in the two next paragraphs. This is the representation used in the Alpha language [113, 117], leveraged by this Chapter.

Hierarchical SARE We enrich SAREs with the ability to call another SARE (called a *subsystem* [74]). The call to a subsystem is specified by a new type of equation, called a *use equation.* A use equation provides the inputs to this program, and retrieves its outputs. A use equation provides a formalism somehow similar to the `map` construction of functional languages: the subsystem is applied iteratively on variable elements specified by an *extension domain*. For instance, the matrix multiplication is an iterative application of a dot product on line and columns vectors. The following example illustrates this notion.

Example Let us assume that we want to implement a matrix-vector product, where the matrix is lower-triangular, by using a subsystem which implements a scalar product:

Program "scalProd" : inputs: *Vect1*, *Vect2* (both defined on $\{i|0 \le i < M\}$)
output: *Res* (scalar)
parameter: M

$$Res = \sum_{0\le k<M} Vect1[k] * Vect2[k];$$

Program "triMatVectProd" : inputs: *Vect* (defined on $\{i|0 \le i < N\}$)
L (defined on $\{i, j|0 \le i \le j < N\}$)
output: *vectRes* (defined on $\{i|0 \le i < N\}$)
parameter: N

`use`$\{k|0 \le k < N\}$ scalProd $[k]$
$((k, i \to i)@Vect,\ \ L)$
`returns` (*vectRes*);

where $(k, i \to i)@Vect$ is a 2-dimensional expression whose value at (k, i) is $Vect[i]$.
In this example, we use the extension domain $\{k|0 \le k < N\}$ which specifies N different subsystems call. The k-th call computes the product of two vectors of size k. The first one is the first k elements of $Vect$, the second one is the kth row of L. The value produced by the k-th instance of the subsystem is the k-th element of $vectRes$. □

Reductions A reduction is the successive application of an *associative and commutative* binary operator $\oplus$ over a list of values. It can be viewed as an accumulation, where any accumulation order is allowed. For example, a matrix multiplication can be written by using a reduction:

$$C[i,j] \quad = \quad \sum_{k=0}^{N-1} A[i,k] * B[k,j];$$

In general, the value of a reduction at the point $\vec{i}$ is $\bigoplus_{\pi(\vec{k})=\vec{i}} E[\vec{k}]$, where E is an expression, and π is a quasi-affine function (possibly with integer divisions and modulos), called the *projection function*. In the example above, we sum over the index k, thus $\pi : (i,j,k \mapsto i,j)$, and the result of the reduction is a two-dimensional variable whose indices (i,j) belong to the image of π.
Reductions are powerful programming and computational abstractions. They ease program transformations which change the associativity of the computation by changing the accumulation order. Such transformations are called *semantic transformations*: the transformed program achieves the same computation modulo associativity/commutativity, but the dependences are no longer respected. This opens a wide range of opportunities for program optimization [109]. This said, semantic transformations must be applied with care on computations over floating point values. In that case, addition and multiplication are no longer associative/commutative and a direct transformation without analyzing the context may cause numerical unstability.

2.1.4 Polyhedral transformations

Polyhedral transformations tune the program structure, the execution order and the data layout to fulfill various optimization goals. The outcome is a *SARE* (possibly changed), a *schedule* defining the evaluation order of SARE array elements $S[\vec{i}]$ (or equivalently program operations $\langle S,\vec{i}\rangle$), and an *allocation* mapping the SARE arrays (or program arrays) to physical storage. At the end of the process, these elements (SARE, schedule, allocation) are provided to a polyhedral code generator [31, 42, 44], which produces the transformed program. Allocation functions will be studied in Chapter 4. We focus here on *schedules* and we present the *tiling* transformation addressed in this chapter.

Scheduling A *schedule* θ_S maps each operation $\langle S,\vec{i}\rangle$ to a timestamp $\theta_S(\vec{i}) = (t_1,\dots,t_d) \in \mathbb{Z}^d$, the timestamps being ordered by the lexicographic order $\ll$. In a way, a schedule dispatches the execution instances $\langle S,\vec{i}\rangle$ into a new loop nest, $\theta_S(\vec{i}) = (t_1,\dots,t_d)$ being the new iteration vector of $\langle S,\vec{i}\rangle$. A schedule θ prescribes 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 $\theta_S(\vec{i}) = \theta_T(\vec{j})$. A schedule is *correct* if it preserves flow dependences: $\rightarrow_{pc} \subseteq \prec_\theta$. Again, the SARE representation removes anti and output dependences [81], which are then ignored. This correctness condition enforces the scheduled program to be *structurally equivalent* to the source program. In [2], we refer this program equivalence as *Herbrand equivalence*. Herbrand equivalence will be addressed in the next Chapter in the context of algorithm recognition. However, *this is not a necessary condition*: a schedule could perfectly describe an equivalent computation by playing on semantic properties of operators, while breaking flow dependences (for instance, by reorganizing a reduction in a different order). In that case, the schedule prescribes a *semantic transformation*. Semantic transformations open a wide range of opportunities for program optimization. This chapter presents a semantic version of loop tiling, called *semantic tiling*.
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 [53, 83].
On Figure 2.1, the original execution order is specified by the schedule $\theta_S(i) = (0,i)$, $\theta_T(i,j) = (1,i,j)$. The lexicographic order ensures that the instances of S are executed before the instances of T (2). Then, for each statement, the loops are executed in the specified order. Likewise, a schedule prescribing parallelism could be $\theta_S(i) = 0$ and $\theta_T(i,j) = i+1$. With this schedule, all the iterations of S are executed in parallel at timestamp 0, then each front $i+1 = t$ of T is executed at timestamp t.

Tiling Tiling groups the iterations into *tiles* (or *blocks*) to be executed *atomically*. After tiling, the iteration domains are reindexed with *block indices*, which iterates over the tiles, and *local indices* which iterates within a tile. In the resulting loop nest, the loop structure consists of *tile loops* (iterating over *block indices*) and *intra-tile loops* (iterating *local indices*). Figure 2.2.(c) illustrates the tiling transformation for the product of polynomials given Figure 2.1, tiles are delimited by bold lines.
The atomicity condition means that tiles can be scheduled as *atomic* operations. Hence, no interdependence is allowed between two tiles.
Providing the dependences, we may need different tile shapes to satisfy the atomicity condition. The most commonly used shape is a hyper-parallelepiped, defined by its boundary hyperplanes [99, 172]. A particular case is *hyperrectangular tiling* where the tile boundaries are normal to the canonic axes. For optimization purpose, other shapes may be required such as trapezoid (with redundant computation [108, 137, 176]), diamond [35] or hexagonal [91, 138] (see Figure 2.7). In particular, hexagonal tiling can expose synchronization-free parallelism for GPU (still, a single global barrier is required). Parallelepipedic tiling exhibits pipelined parallelism, and a common belief is that performances are *bounded by iterated flush-in and flush-out*. This happens when the parallelism is described by a sequential program with parallel `for` constructs, but *no longer when we target a dataflow program*, as the DPN representation discussed in Chapter 5.
The tiling transformation $\mathcal{T}_S$ maps each iteration $\vec{i} \in \mathbb{Z}^n$ of $\mathcal{D}_S$ to a tiled iteration $(\vec{i}_b, \vec{i}_l) \in \mathbb{Z}^{2n}$, where $\vec{i}_b$ denotes the block indices and $\vec{i}_l$ denotes the local indices. For instance, a tiling with hyperrectangles of size $(b_1, \ldots, b_n) = \vec{b} \in \mathbb{Z}^n$ would be specified by $\mathcal{T}_S(\vec{i}) = \left(\left\lfloor \frac{\vec{i}}{\vec{b}} \right\rfloor, \vec{i} \bmod \vec{b}\right)$, with elementwise integer division and modulo.

Tiling a SARE. Given a tiling transformation $\mathcal{T}_S$ for each SARE array S, tiling a SARE means:

- Reindexing arrays with block and local indices $S[\vec{i}] \mapsto \hat{S}[\vec{i}_b, \vec{i}_l]$.
- Tiling each equation domain of $\hat{S}$ with $\mathcal{T}_S$: $\mathcal{D} \mapsto \hat{\mathcal{D}}$.
- Rephrasing array index functions with block and local indices: $f \mapsto \hat{f}$.

Example In Figure 2.2 the tiling is defined by:

$$\mathcal{T}_S(i) = (\lfloor i/2 \rfloor, i \bmod 2) \qquad \text{and} \qquad \mathcal{T}_T(i,j) = (\lfloor i/2 \rfloor, \lfloor j/2 \rfloor, i \bmod 2, j \bmod 2)$$

The original SARE is depicted in (a), domains and index functions are made symbolic for the presentation. (b) gives the tiled SARE, and (c) illustrates the iteration domains of S and T after the tiling transformation. Tiled iteration domains are obtained by writing the euclidian division:

$$\hat{\mathcal{D}}_S = \{(i_b, i_l) \mid 0 \le \mathcal{T}_S^{-1}(i_b, i_l) < 2N \wedge 2i_b \le \mathcal{T}_S^{-1}(i_b, i_l) \le 2(i_b + 1)\}$$

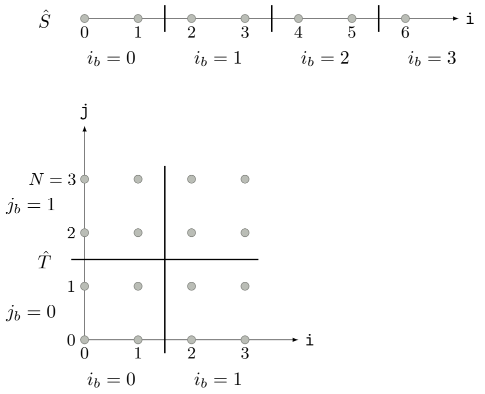
Since $\mathcal{T}_S^{-1}(i_b, i_l) = 2i_b + i_l$ is affine, $\hat{\mathcal{D}}$ is a polyhedron. Hence it satisfies the constraints of the SARE representation. This closure property is unclear when the tile size b is *parametric*, since $\mathcal{T}_S^{-1}(i_b, i_l) = b.i_b + i_l$ is then a quadratic form.
Index functions are rephrased to account for tiled indices. The source and the target domains might be different, with different dimensions and partitionings (e.g., with $f : \mathbb{Z}^2 \to \mathbb{Z}$), which makes it challenging to determine them automatically. For instance, in the original SARE, the index function g is defined by: $g(i,j) = (i-1, j+1)$. In the tiled SARE, we substitute g with its tiled counterpart, $\hat{g} = \mathcal{T}_S \circ g \circ \mathcal{T}_T^{-1}$. Tiled index functions are piecewise: when $(i_l - 1, j_l + 1)$ is a valid local index, $\hat{g}(i_b, j_b, i_l, j_l) = (i_b, j_b, i_l - 1, j_l + 1)$. On the corner case, we take the value from a neighbor tile. Finally, we end-up with a piecewise affine definition of $\hat{g}$, which fits the SARE constraints. Again, the closure is not clear when the tile size $\vec{b}$, is parametric, as a direct application of $\hat{g} = \mathcal{T}_S \circ g \circ \mathcal{T}_T^{-1}$ would lead a non-affine function. One of the contributions of this chapter is to show that when the tile size depends on a single scaling parameter, we still have the closure. □

$$S[i] = \; i \in \mathscr{D}_S : 0$$
$$T[i,j] = \begin{cases} (i,j) \in \mathscr{D}_{T1} : \\ \quad S[f(i,j)] + a[i] * b[j] \\ (i,j) \in \mathscr{D}_{T2} : \\ \quad T[g(i,j)] + a[i] * b[j] \end{cases}$$

(a) SARE

$$\hat{S}[i_b, i_l] = \; (i_b, i_l) \in \hat{\mathscr{D}}_S : 0$$
$$\hat{T}[i_b, j_b, i_l, j_l] = \begin{cases} (i_b, j_b, i_l, j_l) \in \hat{\mathscr{D}}_{T1} : \\ \quad \hat{S}[\hat{f}(i_b, j_b, i_l, j_l)] + \hat{a}[i_b, i_l] * b[j_b, j_l] \\ (i_b, j_b, i_l, j_l) \in \hat{\mathscr{D}}_{T2} : \\ \quad \hat{T}[\hat{g}(i_b, j_b, i_l, j_l)] + a[i_b, i_l] * b[j_b, j_l] \end{cases}$$

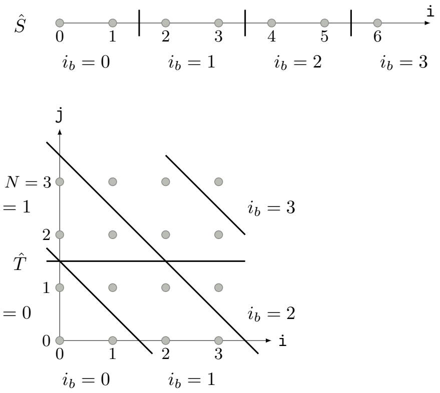
(b) Tiled SARE



(c) Tiled iteration domains

Figure 2.2 – Tiling transformation

Parallelepipedic tiling.
The atomicity constraint may prevent hyperrectangular tiling. In most cases, a *parallelepipedic tiling* with relevant hyperplane boundaries may apply. On perfect loop nests, a *parallelepipedic tiling* is classically defined as the composition of a unimodular transformation (change of basis) followed by an hyperrectangular tiling [172]. This is naturally extended to unperfect loop nests, or SARE with several arrays, by transforming the iteration domain $\mathscr{D}_S$ of each statement S with a bijective *affine transformation* ϕ_S, then by applying an hyperrectangular tiling [53]. The final tiling transformation is then $\mathscr{T}_S \circ \phi_S$ for each statement S. On the example, the transformation $\phi_S(i) = i$ and $\phi_T(i,j) = (i+j, j)$, followed by the same hyperrectangular tiling defines the tiling depicted on the right. Sometimes, the dimensions of ϕ ($\phi_S[0]$ and $\phi_T[0]$, then $\phi_T[1]$) are referred to as "tiling hyperplanes". This terminology will be used in Chapter 4 and Chapter 5.



Monoparametric tiling The tiling transformation is central in automatic parallelization and the *tile size* plays a crucial role. In this chapter, we show that, when the tile size depends on a scaling parameter b (*monoparametric tiling*), hyperrectangular tiling *is a polyhedral transformation* and we present the polyhedral machinery to derive the tiled SARE. Also, *we extend this result to any polytopic tile shape (e.g. diamond, hexagon)* where the lattice of tile origins and the tile shape depends on a scaling parameter.
With this result, it is now possible the reason analytically on the tile size, directly in the polyhedral transformation. This way, we go beyond the back-end approaches [39, 107, 139], which allow to tune the tile size after the compilation. As for the back-end techniques, the scaling parameter b may survive the compilation, which enables runtime tuning for any composition of monoparametric tilings and further polyhedral transformations.

Contributions and outline In this chapter, we present the following contributions:

- We propose the *monoparametric tiling transformation* and we show that it is a *polyhedral transformation*. This closure property is the main contribution of this chapter. Section 2.2 describes the monoparametric tiling transformation for *hyperrectangular tiles*. Then, Section 2.3 generalizes these results to tilings with *any polytopic shape*.

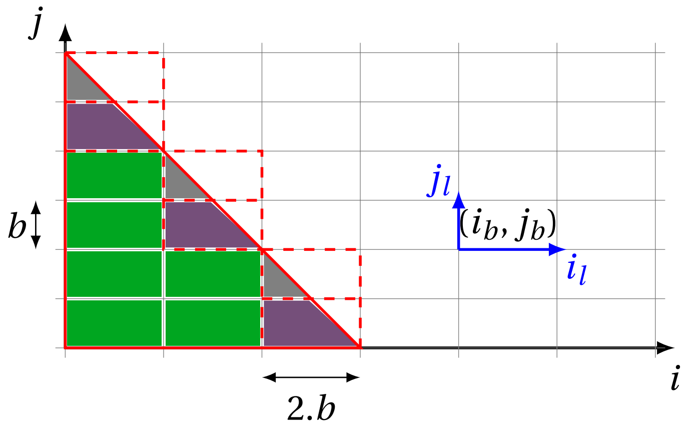


Figure 2.3 – A 2 dimensional monoparametric tiling. The tiles are rectangles of ratio 2 × 1, and the domain is $\mathcal{D} = \{i, j \mid 0 \leq i, j \wedge i + j < N\}$. Each tile is uniquely identify by the block indices (i_b, j_b). A point inside a tile is identify by the local indices (i_l, j_l). When tiling $\mathcal{D}$, we observe 3 kinds of tiles: the full ones (in green), the triangle ones (in gray) and the trapezoid ones (in purple). The shape of each kind of tiles and their placement can both be expressed as polyhedral domains.

- We leverage monoparametric tiling to propose a *semantic tiling* transformation, a transformation increasing the granularity of operators (scalar → matrix). This is discussed in Section 2.4. Semantic tiling is decomposed in reduction tiling (Section 2.4.2), followed by an outlining of the tiles into subsystems (Section 2.4.1).

Finally, Section 2.5 concludes this chapter and draw perspectives.

2.2 Hyperrectangular monoparametric tiling

In this section, we consider the monoparametric tiling transformation with hyperrectangular tiles. Composed with a proper affine transformation, it covers any parallelepipedic tiling.

With hyperrectangular tiling, the tile shape is an hyperrectangle $b.d_1 \times \ldots \times b.d_n$, where b is a scaling parameter and $(d_1, \ldots, d_n) \in \mathbb{N}_*^n$ is called the *tile ratio*. For instance, Figure 2.3 illustrates a $2b \times b$ monoparametric tiling on a triangular domain.

We define the hyperrectangular tiling transformation as follows:

Definition 2.7. *Given the block size parameter b and a diagonal matrix $D = \operatorname{diag}(d_1, \ldots, d_n)$ of tile ratio, the monoparametric tiling transformation associated to this tiling is:*

$$\mathcal{T}_{b,D} = \begin{cases} \mathbb{Z}^n & \mapsto \quad \mathbb{Z}^{2n} \\ \vec{i} & \mapsto \quad (\vec{i}_b, \vec{i}_l) = (\left\lfloor \frac{\vec{i}}{b.D.\vec{1}} \right\rfloor, \vec{i} \bmod (b.D.\vec{1})) \end{cases}$$

where we have extended the division, modulo and floor operation elementwise to vectors.

The inverse of a monoparametric tiling is quadratic: $\mathcal{T}_{b,D}^{-1}(\vec{i}_b, \vec{i}_l) = b.D.\vec{i}_b + \vec{i}_l$, no surprise.

We assume that the parameters $\vec{p}$ can be decomposed in the same fashion: $\vec{p} = b.\vec{p}_b + \vec{p}_l$ where $\vec{p}_b$ is the vector of *tiled parameters*, $\vec{p}_l$ the *local parameters* and $\vec{0} \leq \vec{p}_l < b.\vec{1}$.

The goal of this section is to show that SARE equation domains are still polyhedral domains after the application of $\mathcal{T}_{b,D}$ (Section 2.2.1), and that SARE index functions can be expressed with piecewise affine functions in the tiled domain (Section 2.2.2).

2.2.1 Tiling polyhedra

In this section, we show that polyhedral domains are closed under the monoparametric tiling transformation: if $\mathcal{D}$ is a polyhedral domain, then $\hat{\mathcal{D}} = \mathcal{T}_{b,D}(\mathcal{D})$ is still a polyhedral domain. We consider a convex polyhedron $\mathcal{D} = \{\vec{i},\ Q.\vec{i} + \vec{q} + Q^{(p)}.\vec{p} \geq \vec{0}\}$, then we show that the monoparametric tiling can be expressed with a *finite* union of convex polyhedra.
By definition, $\vec{i} = b.D.\vec{i}_b + \vec{i}_l$ and $\vec{p} = b.\vec{p}_b + \vec{p}_l$. Hence:

$$b.Q.D.\vec{i}_b + Q.\vec{i}_l + b.Q^{(p)}.\vec{p}_b + Q^{(p)}.\vec{p}_l + \vec{q} \geq \vec{0}$$

These constraints are *no longer* polyhedral (b is a parameter and $\vec{i}_b$ are indices). To get rid of the quadratic part, we divide both sides by the tile size parameter b (which is strictly positive):

$$Q.D.\vec{i}_b + Q^{(p)}.\vec{p}_b + \frac{Q.\vec{i}_l + Q^{(p)}.\vec{p}_l + \vec{q}}{b} \geq \vec{0}$$

In general, the fraction is a rational vector. Thus, to come back into the integer world, we take the floor of the previous constraints (it is valid because $a \geq 0 \Leftrightarrow \lfloor a \rfloor \geq 0$):

$$Q.D.\vec{i}_b + Q^{(p)}.\vec{p}_b + \left\lfloor \frac{Q.\vec{i}_l + Q^{(p)}.\vec{p}_l + \vec{q}}{b} \right\rfloor \geq \vec{0}$$

Let $\vec{k}(\vec{i}_l) = \left\lfloor \frac{Q.\vec{i}_l + Q^{(p)}.\vec{p}_l + \vec{q}}{b} \right\rfloor$. We show [26] that $\vec{k}(\vec{i}_l)$ can only take a *constant* number of values, since $\vec{0} \leq \vec{i}_l < b.D.\vec{1}$ and $\vec{0} \leq \vec{p}_l < b.\vec{1}$, bounded in terms of Q, $Q^{(p)}$, p_l and D. Then, it is sufficient to *enumerate* these values to obtain the *union of polyhedra* describing the tiled iteration domain.

Example To illustrate this algebraic manipulation, consider the following parameterized triangle:

$$\mathcal{D} = \{i, j \mid N - 1 - i - j \geq 0 \wedge i \geq 0 \wedge j \geq 0\}$$

We consider the monoparametric tiling with tiles $b \times b$, $\begin{pmatrix} i \\ j \end{pmatrix} = b.\begin{pmatrix} i_b \\ j_b \end{pmatrix} + \begin{pmatrix} i_l \\ j_l \end{pmatrix}$, as depicted on Figure 2.4.
To simplify the presentation, we assume that the parameter N is a multiple of the size parameter b: $N = N_b.b$. Then, the first inequality becomes:

$$\begin{aligned} N - 1 - i - j \geq 0 \quad & \Leftrightarrow \quad N_b.b - 1 - b.i_b - i_l - b.j_b - j_l \geq 0 \\ & \Leftrightarrow \quad N_b - i_b - j_b + \left\lfloor \frac{-i_l - j_l - 1}{b} \right\rfloor \geq 0 \end{aligned}$$

Let us study the values of $k_1(i_l, j_l) = \left\lfloor \frac{-i_l - j_l - 1}{b} \right\rfloor$. Because of the sign of the numerator coefficients, the maximum is -1 ($i_l = j_l = 0$) and the minimum is -2 ($i_l = j_l = b - 1$). After analyzing the two other inequalities, we obtain:

$$\hat{\mathcal{D}} = \left\{ i_b, j_b, i_l, j_l \mid \begin{array}{c} N_b - i_b - j_b - 1 = 0 \\ i_b, j_b \geq 0 \\ 0 \leq i_l, j_l < b \\ -b \leq -i_l - j_l - 1 \end{array} \right\} \uplus \left\{ i_b, j_b, i_l, j_l \mid \begin{array}{c} N_b - i_b - j_b - 2 \geq 0 \\ i_b, j_b \geq 0 \\ 0 \leq i_l, j_l < b \end{array} \right\}$$

This union of polyhedra is shown in Figure 2.4. □

We point out that, on the first polyhedron, we changed the constraint $N_b - i_b - j_b - 1 \geq 0$ to the constraint $N_b - i_b - j_b - 1 = 0$ in order to obtain a partition. Indeed, if $\mathcal{P}_{\vec{r}}$, denotes the polyhedron obtained for $\vec{k}(\vec{i}_l) = \vec{r}$, then $\mathcal{P}_{\vec{r}} \subseteq \mathcal{P}_{\vec{s}}$ when $r_j < s_j$ for each dimension j: $f(\vec{i}) + \vec{r} \geq 0 \Rightarrow f(\vec{i}) + \vec{s} \geq 0$. Hence the idea to keep a constraint “ $\geq 0''$ for the smallest value of $k_1(i_l, j_l)$ (here, -2), and write “=0” constraints for the remaining values of $k_1(i_l, j_l)$ (here, -1).

This observation leads to the following general formulation:

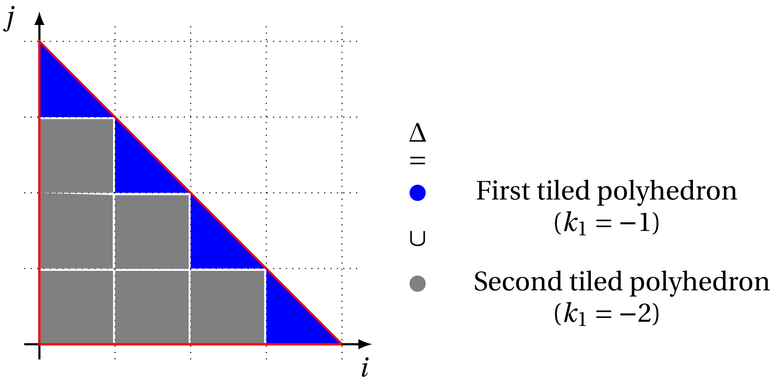


Figure 2.4 – Obtained union of tiled polyhedra Δ. The original polyhedron is a triangle, and we have assume that the tile sizes divide its sizes. We have two polyhedra in Δ: one corresponding to the full tiles, and another for the diagonal lower-triangular tiles

Theorem 2.8. *The image of a polyhedron $\mathscr{D} = \{\vec{i} \mid Q.\vec{i} + Q^{(p)}.\vec{p} + \vec{q} \geq \vec{0}\}$ by a monoparametric tiling transformation $\mathscr{T}_{b,D}$ is the polyhedral domain:*

$$\hat{\mathscr{D}} = \bigcap_{c=1}^{m} \Big[\biguplus_{k_c^{\min} < k_c \leq k_c^{\max}} \left\{ \vec{i}_b, \vec{i}_l \;\middle|\; \begin{array}{c} Q_c.D.\vec{i}_b + Q_c^{(p)}.\vec{p}_b + k_c = 0 \\ b.k_c \leq Q_c.\vec{i}_l + Q_c^{(p)}.\vec{p}_l + q_c \\ \vec{0} \leq \vec{i}_l < b.D.\vec{1} \end{array} \right\}$$
$$\uplus \left\{ \vec{i}_b, \vec{i}_l \;\middle|\; \begin{array}{c} Q_c.D.\vec{i}_b + Q_c^{(p)}.\vec{p}_b + k_c^{min} \geq 0 \\ \vec{0} \leq \vec{i}_l < b.D.\vec{1} \end{array} \right\} \Big]$$

where $\vec{k}$ enumerates the possible values of $\left\lfloor \frac{Q.\vec{i}_l + Q^{(p)}.\vec{p}_l + \vec{q}}{b} \right\rfloor \in [\![\vec{k}^{\min}; \vec{k}^{\max}]\!]$, where $[\![\vec{a}, \vec{b}]\!]$ is the set of integral points in the rectangle whose corners are $\vec{a}$ and $\vec{b}$.

The first union ranges on the m constraints of $\mathscr{D}$. Then, for each constraint c, we bound k_c with k_c^{min} and k_c^{max}. The second member of the $\uplus$ is a special case for $k_c = k_c^{min}$ (note the constraint "≥ 0"). Then, the first member of $\uplus$ add disjoint domains (thanks to the constraint "$= 0$"). After simplification and elimination of empty domains, we obtain a family of convex polyhedra corresponding to the different possible tile shapes. This provides a general monoparametric tiling algorithm.
The scalability of our approach is assessed experimentally on the Polybench kernels [129] (see Section 2.3.3).

2.2.2 Tiling affine functions

After tiling the SARE equation domains, we need to modify accordingly the SARE array index functions. This is a challenging issue, for which we propose a general solution described in this section. Consider an affine function $f : \mathbb{Z}^n \to \mathbb{Z}^p, \vec{i} \mapsto Q.\vec{i} + Q^{(p)}.\vec{p} + \vec{q}$ of the original SARE, used in an equation:

$$A[\vec{i}] = \vec{i} \in \mathscr{D} \; : \; \ldots B[f(\vec{i})] \ldots$$

After applying a monoparametric partitionning to A ($\mathscr{T}'_{b,D'} : \vec{i} \mapsto \vec{I}$) and B ($\mathscr{T}_{b,D}$), we expect an equation of the form:

$$\hat{A}[\vec{I}] = \vec{I} \in \hat{\mathscr{D}} \; : \; \ldots \hat{B}[\hat{f}(\vec{I})] \ldots$$

Hence, we want to derive from f a new index function $\hat{f}$ which operates on the tiled domains:

$$\hat{f} = \mathscr{T}'_{b,D'} \circ f \circ \mathscr{T}_{b,D}^{-1}$$

$$\hat{f}(i_b, j_b, i_l, j_l) = \begin{cases} (4i_b, M-2j_b-1, i_b+j_b, & 2i_l, b-j_l-1, i_l+j_l) \\ & \quad \text{if } \ 0 \le i_l < b \wedge 0 \le j_l < b \wedge 0 \le i_l+j_l < 2b \\ (4i_b+1, M-2j_b-1, i_b+j_b, & 2i_l-b, b-j_l-1, i_l+j_l) \\ & \quad \text{if } \ b \le i_l < 2b \wedge 0 \le j_l < b \wedge 0 \le i_l+j_l < 2b \\ (4i_b, M-2j_b-2, i_b+j_b, & 2i_l, 2b-j_l-1, i_l+j_l) \\ & \quad \text{if } \ 0 \le i_l < b \wedge b \le j_l < 2b \wedge 0 \le i_l+j_l < 2b \\ (4i_b, M-2j_b-2, i_b+j_b+1, & 2i_l, 2b-j_l-1, i_l+j_l-2b) \\ & \quad \text{if } \ 0 \le i_l < b \wedge b \le j_l < 2b \wedge 2b \le i_l+j_l < 4b \\ (4i_b+1, M-2j_b-1, i_b+j_b+1, & 2i_l-b, b-j_l-1, i_l+j_l-2b) \\ & \quad \text{if } \ b \le i_l < 2b \wedge 0 \le j_l < b \wedge 2b \le i_l+j_l < 4b \\ (4i_b+1, M-2j_b-2, i_b+j_b+1, & 2i_l-b, 2b-j_l-1, i_l+j_l-2b) \\ & \quad \text{if } \ b \le i_l < 2b \wedge b \le j_l < 2b \wedge 2b \le i_l+j_l < 4b \end{cases}$$

Figure 2.5 – Rephrased index function $(i, j) \mapsto (2i, N-j-1, i+j)$ for an input tiling $2b \times 2b$ and an output tiling $b \times b$.

while staying in the polyhedral model: $\hat{f}$ must be affine, or at least piece-wise affine.
We start from the definition of f: $\vec{i}' = Q.\vec{i} + Q^{(p)}.\vec{p} + \vec{q}$. With similar arguments as at the beginning of the proof of Theorem 2.8, we get rid of $\vec{i_l}'$ to obtain:

$$\vec{i_b}' = \left\lfloor D'^{-1}.Q.D.\vec{i_b} + D'^{-1}.Q^{(p)}.\vec{p_b} + \frac{D'^{-1}.(Q.\vec{i_l} + Q^{(p)}.\vec{p_l} + \vec{q})}{b} \right\rfloor \tag{2.1}$$

Now, *assume that* $(D'^{-1}.Q.D)$ *and* $(D'^{-1}.Q^{(p)})$ *are integer matrices*. We obtain:

$$\vec{i_b}' = D'^{-1}.Q.D.\vec{i_b} + D'^{-1}.Q^{(p)}.\vec{p_b} + \left\lfloor \frac{D'^{-1}.(Q.\vec{i_l} + Q^{(p)}.\vec{p_l} + \vec{q})}{b} \right\rfloor$$

Again, $\vec{k}(\vec{i_l}) = \left\lfloor \frac{D'^{-1}.(Q.\vec{i_l}+Q^{(p)}.\vec{p_l}+\vec{q})}{b} \right\rfloor$ can take a finite number of values. By enumerating the possible integer values, we obtain a piecewise expression of $\vec{i_b}'$, where each branch corresponds to a value $\vec{k}$ of $\vec{k}(i_l)$:

$$\begin{aligned} &\vec{i_b}' = D'^{-1}.Q.D.\vec{i_b} + D'^{-1}.Q^{(p)}.\vec{p_b} + \vec{k} \\ &\quad \text{if } \ b.\vec{k} \le D'^{-1}.Q.\vec{i_l} + D'^{-1}.Q^{(p)}.\vec{p_l} + D'^{-1}.\vec{q} < b.(\vec{k} + \vec{1}) \end{aligned}$$

Hence the main result:

Theorem 2.9. *Consider an affine function $f : \mathbb{Z}^n \to \mathbb{Z}^p, \vec{i} \mapsto Q.\vec{i} + Q^{(p)}.\vec{p} + \vec{q}$ and a monoparametric tiling for the source domain ($\mathcal{T}_{b,D}$) and for the target domain (${\mathcal{T}'}_{b,D'}$). Assume that $(D'^{-1}.Q.D)$ and $(D'^{-1}.Q^{(p)})$ are integer matrices.*
Then, the composition $\hat{f} = {\mathcal{T}'}_{b,D'} \circ f \circ \mathcal{T}_{b,D}^{-1}$ is a piecewise quasi-affine function, whose branches are:

$$\begin{aligned} &\hat{f}(\vec{i_b}, \vec{i_l}) = \begin{pmatrix} D'^{-1}.Q.D.\vec{i_b} + D'^{-1}.Q^{(p)}.\vec{p_b} + \vec{k} \\ Q.\vec{i_l} + Q^{(p)}.\vec{p_l} + \vec{q} - b.D'.\vec{k} \end{pmatrix} \\ &\qquad \textit{if } \ b.\vec{k} \le D'^{-1}.Q.\vec{i_l} + D'^{-1}.Q^{(p)}.\vec{p_l} + D'^{-1}.\vec{q} < b.(\vec{k} + \vec{1}) \end{aligned}$$

for each $\vec{k} \in [\![\vec{k}^{\min}; \vec{k}^{\max}]\!]$.

Figure 2.6 illustrates the condition of integrality of $(D'^{-1}.Q.D)$ and $(D'^{-1}.Q^{(p)})$. To simplify, consider the identity function $f(i) = i$. On the left, the target tile size is b and the source tile size is $2b$. Hence: $D'^{-1}.Q.D = 1/1 \times 1 \times 1 = 1 \in \mathbb{Z}$ and $D'^{-1}.Q^{(p)} = 1/1 \times 1 = 1 \in \mathbb{Z}$. In that case, the $\hat{f}$ might be expressed with a piecewise affine discussion, as illustrated on the figure. On the right, we have the opposite situation: the target tile size is $2b$ and the source tile size is b. Hence, $D'^{-1}.Q.D = 1/2 \times 1 \times 1 = 1/2 \notin \mathbb{Z}$. In that case, $\hat{f}$ requires integer divisions and modulos: it is piecewise *quasi*-affine.
In the next paragraph, we propose an algorithm to derive $\hat{f}$ in this situation. Piecewise *quasi*-affine functions still fit in the polyhedral representation, at the price of adding existential quantifiers to the SARE domains. This may hinder the polyhedral transformations which expect integer polyhedra rather than Presburger sets.

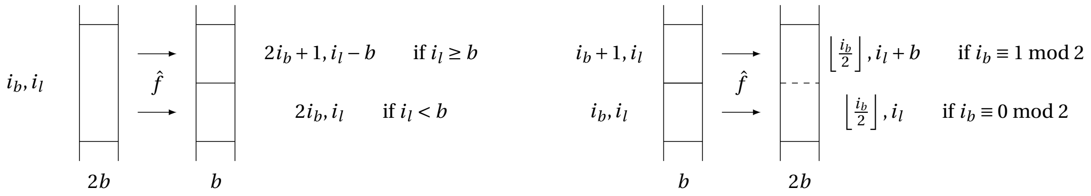

i_b, i_l
$\hat{f}$
$2i_b+1, i_l-b$ if $i_l \geq b$
$2i_b, i_l$ if $i_l < b$
$2b$
b
i_b+1, i_l
i_b, i_l
$\hat{f}$
$\left\lfloor \frac{i_b}{2} \right\rfloor, i_l+b$ if $i_b \equiv 1 \bmod 2$
$\left\lfloor \frac{i_b}{2} \right\rfloor, i_l$ if $i_b \equiv 0 \bmod 2$
b
$2b$

Figure 2.6 – Condition of integrality of $(D'^{-1}.Q.D)$ and $(D'^{-1}.Q^{(p)})$ when f is the identity function. On the left, f maps a tiling $2b$ to a tiling b. The condition is respected: $\hat{f}$ is piecewise affine. On the right, f maps a tiling b to a tiling $2b$. The condition is no longer respected, $\hat{f}$ requires integer divisions and modulos. In general, $\hat{f}$ is piecewise *quasi*-affine.

Example Consider the affine function $f : (i, j \mapsto 2i, N-j-1, i+j)$ and the following monoparametric tiling for the *source domain* and the *target domain*:

- *Source domain*: tiles $2b \times 2b$, $\begin{pmatrix} i \\ j \end{pmatrix} = b.\begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}.\begin{pmatrix} i_b \\ j_b \end{pmatrix} + \begin{pmatrix} i_l \\ j_l \end{pmatrix}$ where $0 \leq i_l, j_l < 2b$, hence $D = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}$.
- *Target domain*: tiles $b \times b$, $\begin{pmatrix} i' \\ j' \end{pmatrix} = b.\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.\begin{pmatrix} i'_b \\ j'_b \end{pmatrix} + \begin{pmatrix} i'_l \\ j'_l \end{pmatrix}$ where $0 \leq i'_l, j'_l < b$, hence $D' = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$.

We assume that the parameter N is divisible by b, and we introduce $N = N_b.b$. We check that $(D'^{-1}.Q.D)$ and $(D'^{-1}.Q^{(p)})$ are both integral, thus we will have purely affine constraints. Roughly, when the target tile size are a multiple
After performing the operations described previously, we obtain an expression of $\vec{i'_b}$:

$$\begin{bmatrix} i'_b \\ j'_b \\ k'_b \end{bmatrix} = \begin{pmatrix} 4 & 0 \\ 0 & -2 \\ 1 & 1 \end{pmatrix} \begin{bmatrix} i_b \\ j_b \end{bmatrix} + \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}.\left[N_b\right] + \begin{bmatrix} k_1 \\ k_2 \\ k_3 \end{bmatrix}$$

where $k_1 = \left\lfloor \frac{2i_l}{b} \right\rfloor$, $k_2 = \left\lfloor \frac{-j_l-1}{b} \right\rfloor$ and $k_3 = \left\lfloor \frac{i_l+j_l}{2b} \right\rfloor$. Thus, $0 \leq k_1 \leq 1$, $-2 \leq k_2 \leq -1$ and $0 \leq k_3 \leq 1$.
By enumerating $(k_1, k_2, k_2) \in \{0,1\} \times \{-1,-1\} \times \{0,1\}$, we obtain 8 branches, 2 branches of which have unsatisfiable conditions. After pruning them out, we obtain the expression of $\hat{f}$ described in Figure 2.5. □

Derivation when $(D'^{-1}.Q.D)$ and $(D'^{-1}.Q^{(p)})$ are not integer matrices If this condition is not satisfied, we cannot separate directly the integer part in Equation 2.2.2. Thus, we write explicitly the integer division of $\vec{i}_b$ and $\vec{p}_b$ with each diagonal coefficient of D', $D'_{l,l}$: $\vec{i}_{bl} = \vec{i}_b^{\,(div),l}.D'_{l,l} + \vec{i}_b^{\,(mod),l}$ and $\vec{p}_{bl} = \vec{p}_b^{\,(div),l}.D'_{l,l} + \vec{p}_b^{\,(mod),l}$.
Now, Equation 2.2.2 becomes:

$$\vec{i}_{bl}^{\,\prime} = Q_l.D.\vec{i}_b^{\,(div),l} + Q_l^{(p)}.\vec{p}_b^{\,(div),l} + \left\lfloor \frac{Q_l.D.\vec{i}_b^{\,(mod),l} + Q_l^{(p)}.\vec{p}_b^{\,(mod),l}}{D'_{l,l}.b} + \frac{Q_l.\vec{i}_l + Q_l^{(p)}.\vec{p}_l + q_l}{D'_{l,l}.b} \right\rfloor$$

Again, $k_l(\vec{i}_b^{\,(mod),l}, \vec{p}_b^{\,(mod),l}) = \left\lfloor \frac{Q_l.D.\vec{i}_b^{\,(mod),l} + Q_l^{(p)}.\vec{p}_b^{\,(mod),l}}{D'_{l,l}.b} + \frac{Q_l.\vec{i}_l + Q_l^{(p)}.\vec{p}_l + q_l}{D'_{l,l}.b} \right\rfloor$ can take a finite number of values. It is sufficient to enumerate the values of the triplet $(\vec{i}_b^{\,(mod),l}, \vec{p}_b^{\,(mod),l}, k_l)$ for each dimension l. For each value, we obtain a new piece for $\hat{f}$, hence the result:

Theorem 2.10. *Given two monoparametric tiling transformation ($\mathcal{T}_{b,D}$ and $\mathcal{T}'_{b,D'}$) and any affine function ($f(\vec{i}) = Q.\vec{i} + Q^{(p)}.\vec{p} + \vec{q}$), if $(D'^{-1}.Q.D)$ or $(D'^{-1}.Q^{(p)})$ is not an integer matrix, the composition $\hat{f} = \mathcal{T}'_{b,D'} \circ f \circ \mathcal{T}_{b,D}^{-1}$ is a piecewise quasi-affine function with modulo conditions in its branches.*

Example Let us consider $f:(i,j)\mapsto(i,j)$ where the input indices are tiled as $\begin{pmatrix}i\\j\end{pmatrix}=\begin{pmatrix}i_b\\j_b\end{pmatrix}.b+\begin{pmatrix}i_l\\j_l\end{pmatrix}$ and the output indices are tiled as $\begin{pmatrix}i'\\j'\end{pmatrix}=\begin{pmatrix}2i'_b\\3j'_b\end{pmatrix}.b+\begin{pmatrix}i'_l\\j'_l\end{pmatrix}$. Let us consider the first output dimension:

$$\begin{aligned} i'=i \quad &\Leftrightarrow \quad 2.i'_b.b+i'_l=i_b.b+i_l \\ &\Rightarrow \quad i'_b=\left\lfloor \frac{i_b}{2}+\frac{i_l}{2b}\right\rfloor = i_b^{(div)}+\left\lfloor \frac{i_b^{(mod)}}{2}+\frac{i_l}{2b}\right\rfloor \end{aligned}$$

where $i_b=2.i_b^{(div)}+i_b^{(mod)}$ and $0\le i_b^{(mod)}\le 1$.
Now, consider the second output dimension. Likewise, we have:

$$j'_b=j_b^{(div)}+\left\lfloor \frac{j_b^{(mod)}}{3}+\frac{j_l}{3b}\right\rfloor$$

where $j_b=3.j_b^{(div)}+j_b^{(mod)}$ and $0\le j_b^{(mod)}\le 2$.
Finally, we build the pieces of $\hat{f}$ by enumerating all the possible values of $i_b^{(mod)}$ and $j_b^{(mod)}$. For example, for $i_b^{(mod)}=j_b^{(mod)}=0$:

$$\vec{k}(i_l,j_l)=\left(\left\lfloor\frac{i_l}{2b}\right\rfloor \quad \left\lfloor\frac{j_l}{3b}\right\rfloor\right)^T$$

$k_1(i_l,j_l)$ and $k_2(i_l,j_l)$ can only take the value 0, thus we will obtain the branch:

$$(i_b/2, j_b/3, \quad i_l, j_l)^T \quad \text{if } \ i_b\equiv 0 \bmod 2 \wedge j_b \equiv 0 \bmod 3$$

After enumerating all the possible values for $(i_b^{(mod)},k_1,j_b^{(mod)},k_2)$, we obtain:

$$\hat{f}:\begin{bmatrix}i_b\\j_b\\i_l\\j_l\end{bmatrix}\mapsto\begin{cases}(i_b/2, j_b/3, & i_l, j_l)^T & \text{if } \ i_b\equiv 0 \bmod 2\wedge j_b\equiv 0 \bmod 3\\ (i_b/2,(j_b-1)/3, & i_l, j_l+b)^T & \text{if } \ i_b\equiv 0 \bmod 2\wedge j_b\equiv 1 \bmod 3\\ (i_b/2,(j_b-2)/3, & i_l, j_l+2b)^T & \text{if } \ i_b\equiv 0 \bmod 2\wedge j_b\equiv 2 \bmod 3\\ ((i_b-1)/2, j_b/3, & i_l+b, j_l)^T & \text{if } \ i_b\equiv 1 \bmod 2\wedge j_b\equiv 0 \bmod 3\\ ((i_b-1)/2,(j_b-1)/3, & i_l+b, j_l+b)^T & \text{if } \ i_b\equiv 1 \bmod 2\wedge j_b\equiv 1 \bmod 3\\ ((i_b-1)/2,(j_b-2)/3, & i_l+b, j_l+2b)^T & \text{if } \ i_b\equiv 1 \bmod 2\wedge j_b\equiv 2 \bmod 3\end{cases}$$

□

Deriving the ratios to avoid modulo conditions If we assume that the ratio of all variables were chosen beforehand, we just have to check for their compatibility, i.e., we have to check that tiling the dependence functions do not introduce non-polyhedral modulo constraints. This means that we have to check, for any dependence function $(\vec{i}\mapsto Q.\vec{i}+Q^{(p)}.\vec{p}+\vec{q})$ and ratio D and D', that $(D'^{-1}.Q.D)$ and $(D'^{-1}.Q^{(p)})$ are integral.
In a more general situation, we assume that the ratio of some variables were chosen beforehand (either by the user or by the compiler), but not all ratios were decided. In order to apply the monoparametric tiling transformation, we propose an algorithm to find the ratio for all the remaining variables, such that no modulo constraints are introduced in their equations [98].

2.3 General monoparametric tiling

In Section 2.2, we have focused on monoparametric tiling with *hyperrectangular* shapes. In this section, we show how this theory can be extended to any polyhedral tile shape (hexagonal [91], diamond [35], etc). In particular we show the closure of polyhedral representations under our *general monoparametric partitionning*. We first recall the definition of the *fixed size* general tiling, then we define our general monoparametric tiling.

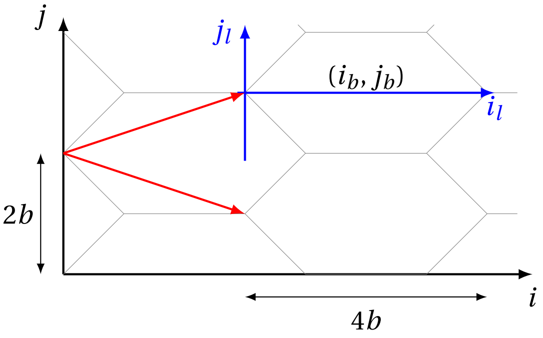


Figure 2.7 – Example of hexagonal monoparametric tiling for a 2D space. (i_b, j_b) are the block indices, which identify a tile, (i_l, j_l) are the local indices, which identify the position of a point inside a tile. The tile shape is an hexagon with 45° slopes and of size $4b \times 2b$, and can be viewed as the homothetic scaling of a 4×2 hexagon. The red arrows correspond to a basis of the lattice of tile origins.

General fixed-size tiling A *general fixed-size tiling* is defined by:

- A bounded convex polyhedron $\mathscr{P}$
- An integer lattice $\mathscr{L}$ of the tile origins (which admits a basis L) and,
- A function $\mathscr{T}$ which decomposes any point $\vec{i}$ in the following way:

$$\mathscr{T}(\vec{i}) = (\vec{i}_b, \vec{i}_l) \Leftrightarrow \vec{i} = L.\vec{i}_b + \vec{i}_l \quad \text{where } L.\vec{i}_b \in \mathscr{L} \text{ and } \vec{i}_l \in \mathscr{P}$$

When this decomposition is unique, this tiling defines a partition of the space. Otherwise, we have overlapping tiles. Some tilings do not have an integral lattice of tile origins (such as diamond tiling with non-unimodular hyperplanes). We do not address overlapped tiles or non-integral tile origins. We now extend this definition with a scaling parameter b, as for the hyperrectangular case:

Definition 2.11. *A* monoparametric general tiling *is defined by:*

- *A tile shape $\mathscr{P}_b = b \times \mathscr{P}$, where $\mathscr{P}$ is a convex polyhedron ($\times$ is the homothetic scaling)*
- *A lattice of tile origins $\mathscr{L}_b = b \times \mathscr{L}$, where $\mathscr{L}$ is an integer lattice.*
- *A function $\mathscr{T}_b$ which decomposes any point $\vec{i}$ in the following way:*

$$\mathscr{T}_b(\vec{i}) = (\vec{i}_b, \vec{i}_l) \Leftrightarrow \vec{i} = b.L.\vec{i}_b + \vec{i}_l \quad \textit{where } (L.\vec{i}_b) \in \mathscr{L} \textit{ and } \vec{i}_l \in b \times \mathscr{P}$$

We now show that the monoparametric general tiling *is a polyhedral transformation*, just as we did for the hyperrectangular case. First, we show that tiled domains $\hat{\mathscr{D}} = \mathscr{T}_b(\mathscr{D})$ are union of convex polyhedra (Section 2.3.1). Then, we show that index functions between tiled domains, $\hat{f}$ are piecewise quasi affine. As for the hyperrectangular case, we give a condition that makes $\hat{f}$ piecewise affine (Section 2.3.2).

2.3.1 Tiling polyhedra

Let us consider a n-dimensional polyhedron $\mathscr{D} = \{\vec{i} \mid Q.\vec{i} + Q^{(p)}.\vec{p} + \vec{q} \geq \vec{0}\}$ where $\vec{p}$ are the program parameters. As in Section 2.2.1, we replace $\vec{i}$ by the *block indices* $\vec{i}_b$ and the *local indices* $\vec{i}_l$, such that $\mathscr{T}_b(\vec{i}) = (\vec{i}_b, \vec{i}_l)$ (cf Figure 2.7).

Consider the c-th constraint of $\mathscr{D}$: $Q_c.\vec{i} + Q_c^{(p)}.\vec{p} + q_c \geq 0$. We substitute $\vec{i}$ by $b.L.\vec{i}_b + \vec{i}_l$ where $\vec{i}_l \in \mathscr{P}_b$. By doing exactly the same operations as in the proof of Theorem 2.8, we obtain the following expression:

$$Q_c.L.\vec{i}_b + Q_c^{(p)}.\vec{p}_b + \left\lfloor \frac{Q_c.\vec{i}_l + Q_c^{(p)}.\vec{p}_l + q_c}{b} \right\rfloor \geq \vec{0}$$

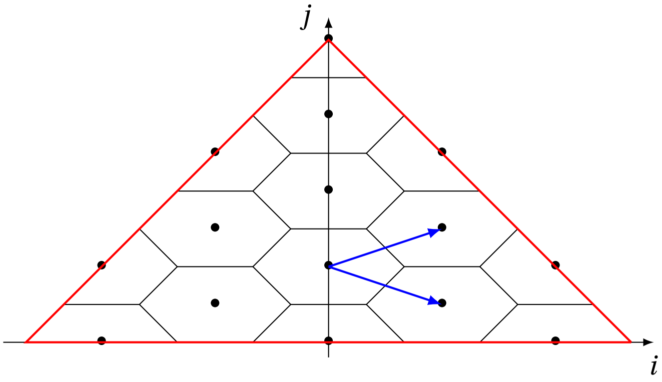


Figure 2.8 – Polyhedron and tiling of the example. The dots correspond to the tile origins of the tiles contributing to the polyhedron. The blue arrows show the basis of the lattice of tile origins.

Again, the quantity $k_c(\vec{i_l}) = \left\lfloor \frac{Q_c.\vec{i_l}+Q_c^{(p)}.\vec{p_l}+q_c}{b} \right\rfloor$ can take a finite number of values because $\vec{i_l} \in \mathscr{P}_b = b \times \mathscr{P}$ where $\mathscr{P}$ is bounded. This way, we obtain a polyhedral formulation of the tiled iteration domains $\hat{\mathscr{D}}$ very similar to Theorem 2.8 for the hyperrectangular case:

Theorem 2.12. *The image of a polyhedron* $\mathscr{D} = \{\vec{i} \mid Q.\vec{i} + Q^{(p)}.\vec{p} + \vec{q} \geq \vec{0}\}$ *by a general monoparametric tiling transformation is the polyhedral domain:*

$$\hat{\mathscr{D}} = \bigcap_{c=1}^{m} \left[\biguplus_{k_c^{\min} < k_c \leq k_c^{\max}} \left\{ \vec{i_b}, \vec{i_l} \;\middle|\; \begin{array}{c} Q_c.L.\vec{i_b} + Q_c^{(p)}.\vec{p_b} + k_c = 0 \\ b.k_c \leq Q_c.\vec{i_l} + Q_c^{(p)}.\vec{p_l} + q_c \\ \vec{i_l} \in \mathscr{P}_b \end{array} \right\} \right.$$
$$\left. \uplus \left\{ \vec{i_b}, \vec{i_l} \;\middle|\; \begin{array}{c} Q_c.L.\vec{i_b} + Q_c^{(p)}.\vec{p_b} + k_c^{min} \geq 0 \\ \vec{i_l} \in \mathscr{P}_b \end{array} \right\} \right]$$

where $\vec{k}$ *enumerates the possible values of* $\left\lfloor \frac{Q.\vec{i_l}+Q^{(p)}.\vec{p_l}+\vec{q}}{b} \right\rfloor$.

After distributing the intersection across the unions and eliminating the empty polyhedral, we obtain as many polyhedra as the number of different tile shapes of the partitioned version of $\mathscr{D}$ (which is, at most, the number of different values of $\vec{k}$).

Example Consider the polyhedron: $\mathscr{D} = \{i, j \mid j - i \leq N \wedge i + j \leq N \wedge 0 < j\}$ and the tiling:

- $\mathscr{P}_b = \{i, j \mid -b < j \leq b \;\wedge\; -2b < i + j \leq 2b \;\wedge\; -2b < j - i \leq 2b\}$
- $L_b = L.b.\mathbb{Z}^2$ where $L = \begin{bmatrix} 3 & 3 \\ 1 & -1 \end{bmatrix}$

For simplicity, we assume that $N = 6.b.N_b + 2b$, where N_b is a positive integer. A graphical representation of the polyhedron and of the tiling is shown in Figure 2.8.

We now unroll the enclosing intersection by considering independently each constraint of $\mathscr{D}$.

- Consider the first constraint of the polyhedron.

$$\begin{array}{rcl} j - i \leq N & \Leftrightarrow & 0 \leq 6.b.N_b + 2.b + b.(3.i_b + 3.j_b) + i_l - b.(i_b - j_b) - j_l \\ & \Leftrightarrow & 0 \leq 6.N_b + 2 + 2.i_b + 4.j_b + \left\lfloor \frac{i_l - j_l}{b} \right\rfloor \end{array}$$

where $-2b \leq i_l - j_l < 2b$. Therefore, $k_1 = \left\lfloor \frac{i_l - j_l}{b} \right\rfloor \in [\![-2, 1]\!]$. For $k_1 = -1$ and 1, the equality constraint $6.N_b + 2.i_b + 4.j_b + 2 + k_1 = 0$ is not satisfied (because of the parity of its terms), thus $k_1 \in \{-2, 1\}$.

- Consider the second constraint of the polyhedron.

$$\begin{aligned} i+j \leq N &\Leftrightarrow 0 \leq 6.b.N_b + 2.b - b.(3.i_b + 3.j_b) - i_l - L.b.(i_b - j_b) - j_l \\ &\Leftrightarrow 0 \leq 6.N_b + 2 - 4.i_b - 4.j_b + \left\lfloor \frac{-i_l - j_l}{b} \right\rfloor \end{aligned}$$

  where $-2b \leq -i_l - j_l < 2b$. Therefore $k_2 = \left\lfloor \frac{-i_l-j_l}{b} \right\rfloor \in [\![-2, 1]\!]$. For the same reason as the previous constraint, $k_2 = -1$ and 1 lead to empty polyhedra. Hence $k_2 \in \{-2, 0\}$.
- Finally, consider the third constraint of the polyhedron.

$$\begin{aligned} 0 \leq j-1 &\Leftrightarrow 0 \leq b.(i_b - j_b) + j_l - 1 \\ &\Leftrightarrow 0 \leq i_b - j_b + \left\lfloor \frac{j_l - 1}{b} \right\rfloor \end{aligned}$$

  where $-b \leq j_l - 1 < b$. Therefore $k_3 = \left\lfloor \frac{j_l-1}{b} \right\rfloor \in [\![-1, 0]\!]$

Therefore, we obtain a union of $2 \times 2 \times 2 = 8$ polyhedra, which are the result of the following intersections:

$$\begin{aligned} &\left[\begin{array}{l} \{i_b, j_b, i_l, j_l | 0 \leq 6.N_b + 2.i_b + 4.j_b \wedge (i_l, j_l) \in \mathscr{T}_b\} \\ \uplus \{i_b, j_b, i_l, j_l | 0 = 6.N_b + 2.i_b + 4.j_b + 2 \wedge (i_l, j_l) \in \mathscr{T}_b \wedge 0 \leq i_l - j_l\} \end{array} \right] \\ \cap &\left[\begin{array}{l} \{i_b, j_b, i_l, j_l | 0 \leq 6.N_b - 4.i_b - 4.j_b \wedge (i_l, j_l) \in \mathscr{T}_b\} \\ \uplus \{i_b, j_b, i_l, j_l | 0 = 6.N_b - 4.i_b - 4.j_b + 2 \wedge (i_l, j_l) \in \mathscr{T}_b \wedge 0 \leq -i_l - j_l\} \end{array} \right] \\ \cap &\left[\begin{array}{l} \{i_b, j_b, i_l, j_l | 0 \leq i_b - j_b - 1 \wedge (i_l, j_l) \in \mathscr{T}_b\} \\ \uplus \{i_b, j_b, i_l, j_l | 0 = i_b - j_b \wedge (i_l, j_l) \in \mathscr{T}_b \wedge 0 \leq j_l - 1\} \end{array} \right] \end{aligned}$$

□

2.3.2 Tiling affine functions

As for the hyperrectangular case described in Section 2.2.2, we need to modify the SARE index functions to account for the tiling reindexation.

Let us consider an affine function $f : (\vec{i} \mapsto Q.\vec{i} + Q^{(p)}.\vec{p} + \vec{q})$ and two tilings: one for the input indices ($\mathscr{T}_b$) and one for the output indices (${\mathscr{T}'}_b$). Note that the "tile shapes" in the input and output dimensions, $\mathscr{P}_b$ and $\mathscr{P}'_b$ might be different.

We adapt the derivation of Theorem 2.9 to obtain a close definition of $\hat{f} = {\mathscr{T}'}_b \circ f \circ \mathscr{T}_b^{-1}$. From $\vec{i}' = f(\vec{i}) = Q.\vec{i} + Q^{(p)}.\vec{p} + \vec{q})$, we obtain:

$$\vec{i_b}' + \left\lfloor \frac{L'^{-1}.\vec{i_l}'}{b} \right\rfloor = \left\lfloor L'^{-1}.Q.L.\vec{i_b} + L'^{-1}.Q^{(p)}.\vec{p_b} + \frac{L'^{-1}.(Q.\vec{i_l} + Q^{(p)}.\vec{p_l} + \vec{q})}{b} \right\rfloor$$

Now, *assume that $L'^{-1}.Q.L$ and $L'^{-1}.Q^{(p)}$ are integral.* This can be viewed as a generalization of the hypothesis on $(D'^{-1}.Q.D)$ and $(D'^{-1}.Q^{(p)})$ on the hyperrectangular case. We show that the vectors $\vec{k}(\vec{i_l}) = \left\lfloor \frac{L'^{-1}.(Q.\vec{i_l}+Q^{(p)}.\vec{p_l}+\vec{q})}{b} \right\rfloor$ (on the right) and $\vec{k}'(\vec{i_l}') = \left\lfloor \frac{L'^{-1}.\vec{i_l}'}{b} \right\rfloor$ (on the left) can take a finite number of values. Each couple of vectors for $\vec{k}(\vec{i_l})$ and $\vec{k}'(\vec{i_l}')$ defines a new affine piece of $\hat{f}$:

Theorem 2.13. *Given two general monoparametric tiling transformations ($\mathscr{T}_b$ and ${\mathscr{T}'}_b$) and any affine function ($f(\vec{i}) = Q.\vec{i} + Q^{(p)}.\vec{p} + \vec{q}$), the composition (${\mathscr{T}'}_b \circ f \circ \mathscr{T}_b^{-1}$) is a piecewise quasi-affine function, whose branches are of the form:*

$$\hat{f}(\vec{i_b}, \vec{i_l}) = \begin{pmatrix} L'^{-1}.Q.L.\vec{i_b} + L'^{-1}.Q^{(p)}.\vec{p_b} + \vec{k} - \vec{k}' \\ Q.\vec{i_l} + Q^{(p)}.\vec{p_l} + \vec{q} + b.L'(\vec{k}' - \vec{k}) \end{pmatrix}$$
$$\textit{if} \left\{ \begin{array}{l} b.\vec{k} \leq L'^{-1}\left(Q.\vec{i_l} + Q^{(p)}.\vec{p_l} + \vec{q}\right) < b.(\vec{k} + \vec{1}) \\ Q.\vec{i_l} + Q^{(p)}.\vec{p_l} + \vec{q} + b.L'(\vec{k}' - \vec{k}) \in \mathscr{P}'_b \\ \vec{i_l} \in \mathscr{P}_b \end{array} \right.$$

for each $\vec{k} \in [\![\vec{k}^{\min}; \vec{k}^{\max}]\!]$, for each $\vec{k}' \in [\![\vec{k}'^{\min}; \vec{k}'^{\max}]\!]$, where L, L' are bases of the lattices of tile origins of respectively $\mathscr{T}$ and $\mathscr{T}'$, and assuming that $(L'^{-1}.Q.L)$ and $(L'^{-1}.Q^{(p)})$ are integer matrices.

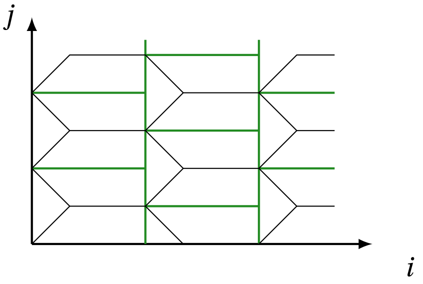


Figure 2.9 – Overlapping of rectangular (in green) and the hexagonal tiles

Example Consider the identity function $(i,j) \mapsto (i,j)$, with the two following tilings:

- For the input space, we choose an hexagonal tiling:
  - $\mathcal{T}_b = \{i,j \mid -b < j \leq b \ \wedge\ -2b < i+j \leq 2b \ \wedge\ -2b < j-i \leq 2b\}$
  - $L_b = L.b.\mathbb{Z}^2$ where $L = \begin{bmatrix} 3 & 3 \\ 1 & -1 \end{bmatrix}$
- For the output space, we choose a rectangular tiling, with the same lattice:
  - $\mathcal{T}'_b = \{i,j \mid 0 \leq i < 3b \ \wedge\ 0 \leq j < 2b\}$
  - $L'_b = L'.b.\mathbb{Z}^2$ where $L' = \begin{bmatrix} 3 & 3 \\ 1 & -1 \end{bmatrix}$

An overlapping of these two tilings is shown in Figure 2.9.
The derivation goes as follow:

$$\begin{aligned} & \begin{bmatrix} i' \\ j' \end{bmatrix} = \begin{bmatrix} i \\ j \end{bmatrix} \\ \Leftrightarrow\quad & L'.b.\begin{bmatrix} i'_b \\ j'_b \end{bmatrix} + \begin{bmatrix} i'_l \\ j'_l \end{bmatrix} = L.b.\begin{bmatrix} i_b \\ j_b \end{bmatrix} + \begin{bmatrix} i_l \\ j_l \end{bmatrix} \\ \Leftrightarrow\quad & \begin{bmatrix} i'_b \\ j'_b \end{bmatrix} + L'^{-1}.\tfrac{1}{b}.\begin{bmatrix} i'_l \\ j'_l \end{bmatrix} = \begin{bmatrix} i_b \\ j_b \end{bmatrix} + L'^{-1}.\tfrac{1}{b}.\begin{bmatrix} i_l \\ j_l \end{bmatrix} \end{aligned}$$

Because $L'^{-1} = \frac{1}{6}.\begin{bmatrix} 1 & 3 \\ 1 & -3 \end{bmatrix}$, then the constraints become:

$$\begin{cases} i'_b + \frac{i'_l+3.j'_l}{6b} = i_b + \frac{i_l+3.j_l}{6b} \\ j'_b + \frac{i'_l-3.j'_l}{6b} = j_b + \frac{i_l-3.j_l}{6b} \end{cases}$$

After taking the floor of these constraints:

$$\begin{cases} i'_b + \left\lfloor \frac{i'_l+3.j'_l}{6b} \right\rfloor = i_b + \left\lfloor \frac{i_l+3.j_l}{6b} \right\rfloor \\ j'_b + \left\lfloor \frac{i'_l-3.j'_l}{6b} \right\rfloor = j_b + \left\lfloor \frac{i_l-3.j_l}{6b} \right\rfloor \end{cases}$$

We define $k'_1 = \left\lfloor \frac{i'_l+3.j'_l}{6b} \right\rfloor$, $k_1 = \left\lfloor \frac{i_l+3.j_l}{6b} \right\rfloor$, $k'_2 = \left\lfloor \frac{i'_l-3.j'_l}{6b} \right\rfloor$ and $k_2 = \left\lfloor \frac{i_l-3.j_l}{6b} \right\rfloor$. After analysis of the extremal values of these quantities, we obtain:

- $k_1 \in [\![-1;0]\!]$ and $k_2 \in [\![-1;0]\!]$
- $k'_1 \in [\![0;1]\!]$ and $k'_2 \in [\![-1;0]\!]$

Therefore, we obtain a piecewise quasi-affine function with 16 branches (one for each value of (k_1, k'_1, k_2, k'_2)). Each branch has the following form:

$$\begin{aligned} &\Big(i_b + k_1 - k'_1, j_b + k_2 - k'_2,\ \ i_l + 3b(k'_1 + k'_2 - k_1 - k_2), j_l + b(k'_1 + k_2 - k_1 - k'_2)\Big) \\ &\text{when } 0 \leq i_l + 3b(k'_1 + k'_2 - k_1 - k_2) < 3b \ \wedge\ 0 \leq j_l + b(k'_1 + k_2 - k_1 - k'_2) < 2b \\ &\qquad k_1.b \leq i_l + 3j_l < (k_1+1).b \ \wedge\ k_2.b \leq i_l - 3j_l < (k_2+1).b \\ &\qquad -b < j_l \leq b \ \wedge\ -2b < i_l + j_l \leq 2b \ \wedge\ -2b < j_l - i_l \leq 2b \end{aligned}$$

□

Case where $L'^{-1}.Q.L$ or $L'^{-1}.Q^{(p)}$ are not integral As for the hyperrectangular case, we want to address the case where $L'^{-1}.Q.L$ and $L'^{-1}.Q^{(p)}$ are not integral. When L and L' are diagonal, we may apply the same algorithm as for the hyperrectangular case. Otherwise, writing explicitly the integer division and to enumerating the combination of values for the quotient and the modulo would produce a huge number of branches. Indeed, we would end-up with, at worse, as many integer divisions as non-integral coefficients in $L'^{-1}.Q.L$ and $L'^{-1}.Q^{(p)}$. Though it would work, we believe that computing a closed form for $\hat{f}$ might not be relevant in that case. As for the hyperrectangular case, we need to infer the tile ratio of the intermediate arrays to enforce the integrality of $L'^{-1}.Q.L$ and $L'^{-1}.Q^{(p)}$.
This gives a complete algorithm to express general monoparametric tiling as a polyhedral representation. The next section evaluates experimentally the scalability of our technique.

2.3.3 Experimental validation

In this section, we evaluate the scalability of our monoparametric tiling transformation and the quality of the tiled code generated.
We have two implementations of monoparametric tiling:

- An implementation in the *AlphaZ* system [174], which covers only rectangular tile shapes
- A standalone C++ implementation [1] interfaced with a source-to-source C compiler, `mppcodegen` [2], able to produce tiled code out of any mix of rectangular and general monoparametric tiling.

We first report on the scalability of the transformation itself, by studying the AlphaZ implementation. Then, we will study the quality of the tiled code generated using the output of the C compiler.
We run our experiment on a machine with an Intel Xeon E5-1650 CPU with 12 cores running at 1.6 GHz (max speed at 3.8GHz), and 31GB of memory.

Scalability of the monoparametric tiling transformation

We use Polybench/Alpha [3] benchmarks, an hand-written *Alpha* implementation of the *Polybench 4.0* benchmark suite.
Since the tiling transformation is the reindexing part of a tiling, there is no legality condition to respect. Hence, we partition each kernel with a rectangular tiling of ratio 1^d where d is the number of dimensions of a variable. Then, we apply a polyhedral analysis on the transformed program. We choose to compute the *context domain* of each node n of the Alpha program's AST. Alpha programs are not exactly SAREs, they feature piecewise definitions (`case`), reductions, calls to subsystems, and any imbrication thereof. This is more flexible than a SARE, but it requires additional analyses to be compiled. The context domain of a node is the set of indices on which the expression value is required to compute the output of a program. This analysis performs polyhedral operations (such as image and preimage) for certain nodes of the AST. Its complexity increases with the size of the AST.

Results and analysis Figure 2.10 reports the time taken by each phase for all the kernel of Polybench/Alpha, and the number of node of the AST of the program after the tiling transformation.
The time taken by the transformation itself remains reasonable (no more than about 2 seconds for heat-3d). However, the time taken by the subsequent polyhedral analysis (i.e., the context domain calculation) is huge for the stencil kernels (the last six kernels in the bottom table), with heat3d taking up to about 37 minutes). This is due to the size of the program after tiling and the fact that the context domain analysis builds a polyhedral set per node of the AST.

[1] Available at https://github.com/guillaumeiooss/MPP
[2] Available at https://foobar.ens-lyon.fr/mppcodegen
[3] http://www.cs.colostate.edu/AlphaZsvn/Development/trunk/mde/edu.csu.melange.alphaz.polybench/polybench-alpha-4.0/

Time taken (ms)	correlation	covariance	gemm	gemver	gesummv	symm	syr2k	syrk	trmm	2mm	3mm	atax	bicg	doitgen	mvt	cholesky
Parsing	121	69	62	83	50	118	83	54	43	93	112	51	51	54	55	389
Tiling	300	157	151	178	93	282	439	119	82	308	482	112	113	187	159	369
Context Domain	1147	504	163	230	162	1257	685	153	207	319	451	153	153	185	201	1197
Num AST Nodes	110	66	21	47	29	136	36	21	25	34	39	25	25	13	29	113
Num Equations	10	6	2	5	3	14	3	2	3	4	6	4	4	2	4	15

Time taken (ms)	durbin	gramschmidt	lu	ludcmp	trisolv	deriche	floyd-warshall	nussinov	adi	fdtd-2d	jacobi-1d	jacobi-2d	seidel-2d	heat-3d
Parsing	121	147	106	179	74	468	220	122	546	331	139	134	183	278
Tiling	266	398	284	472	139	1213	390	380	2393	1048	678	628	550	3275
Context Domain	2182	1867	1208	2672	203	2843	335	6845	2m 32s	1m 52s	2913	58s	1m 28s	37m 13s
Num AST Nodes	315	123	138	216	39	659	27	537	11931	4194	334	2836	4684	50170
Num Equations	34	20	20	30	5	40	4	57	570	495	38	194	210	1242

Figure 2.10 – Time taken by the hyperrectangular monoparametric tiling transformation inside the compiler, number of nodes of the AST of the program after the tiling transformation and number of equations of the partitioned program. All the considered stencil computations (adi to heat-3d) have an order of 1.

The main reason a partitioned stencil computation is so big is because of the multiple uniform dependences (of the form $(\vec{i} \mapsto \vec{i}+\vec{c})$ where $\vec{c}$ are constants) in its computation. For each such dependence, the partitioned piecewise affine function has a branch per block of data accessed. Thus the normalized partitioned program will have a branch of computation per combination of block of the data accessed. Even if we progressively eliminate empty polyhedra during normalization, we still have a large number of branches that cannot be merged. Because all the branches contain useful information, we cannot further reduce the size of this program.

Quality of the monoparametric tiled code

We now consider the source-to-source C compiler implementation. Our goal is the compare the quality of a monoparametric tiled code with a fixed-size tiled code. We produce these two codes with the same compiler framework, the only different optimization decision being the nature of the tiling performed. For each Polybench/C kernel, we generate a monoparametric tiled code and a fixed-size tiled code with a tile size of 16 by using the tiling hyperplanes found by the pluto compiler [53]. In order to conserve the memory mapping, we apply the monoparametric tiling transformation only on the iteration space. We simply express the original index by using the tile index. For example, in the case of a square rectangular tiling of tile size b and an array access `A[i][k]`, we would generate `A[ib*b+il][kb*b+kl]`.
The execution timesare shown in Figure 2.11. For most of the kernels, the execution time of both tiled code are comparable. However, the monoparametric code is sometimes twice as fast as the fixed-size code. When substituting the tile size parameter with a constant in the monoparametric tiled code, we obtain similar performance. Thus, this is caused by the difference of the structure of the code generated by ISL. Indeed, the inner loop iterator is not the same: the original iterator is used for the fixed-size tiled code (starting at the origin of the current tile) while the monoparametric code uses i_l (starting at 0). Also, the monoparametric code explicitly separates the tile shapes into internal loops. This leads to bigger code, but allows the factorization of some terms across loops.

Time taken (ms)	correlation	covariance	gemm	gemver	gesummv	symm	syr2k	syrk	trmm	2mm	3mm	atax	bicg
Fixed-size	949	944	776	27.5	3.32	1416	939	617	618	1420	2460	19.2	21.3
Monoparametric	843	945	945	33.2	3.20	1616	928	575	700	1279	2814	17.6	22.7

Time taken (ms)	doitgen	mvt	cholesky	gramschmidt	lu	trisolv	floyd-warshall	fdtd-2d	jacobi-1d	jacobi-2d	seidel-2d	heat-3d
Fixed-size	424	21.0	2054	3093	4255	2.94	20179	2515	5.22	2797	13540	5395
Monoparametric	418	20.8	1108	3107	2357	1.63	19021	1636	7.84	2300	13438	3746

Figure 2.11 – Comparison of the execution time between a fixed-size tiled code and a monoparametric tiled code, given the same compiler framework and optimization parameters. Each number reported is the average of 50 executions.

2.4 Semantic tiling

In this section, we leverage our monoparametric tiling to derive a semantic tiling. We show that it suffices to combine reduction tiling (Section 2.4.2) and an an *outlining transformation* able to encapsulate the tiles into subsystems (Section 2.4.1) to do so. Finally, we evaluate the scalability of our transformation.

Motivating Example Consider a matrix multiplication program with a reduction:

$$(\forall 0 \le i, j < N)\ C[i,j] = \sum_{k=0}^{N-1} A[i,k] * B[k,j]$$

We assume that N is divisible by the block size b. After applying the tiling transformation, we obtain the following program:

$$(\forall 0 \le i_b, j_b < N_b)(\forall 0 \le i_l, j_l < b)\ \hat{C}[i_b, j_b, i_l, j_l] = \sum_{k_b, k_l} \hat{A}[i_b, k_b, i_l, k_l] * \hat{B}[k_b, j_b, k_l, j_l];$$

The reduction introduces an additional dimension, k, which might be tiled by scheduling the evaluation of $\hat{A}[i_b, k_b, i_l, k_l] * \hat{B}[k_b, j_b, k_l, j_l]$ at timestamp $(i_b, j_b, k_b, i_l, j_l, k_l)$. However, with that scheme, the *reduction itself is not tiled*. Hence, we proposed to tile the reduction by *modifying the structure of the SARE*:

$$\begin{aligned} \hat{C}[i_b, j_b, i_l, j_l] &= \sum_{k_b} T[i_b, j_b, k_b, i_l, j_l]; \\ T[i_b, j_b, k_b, i_l, j_l] &= \sum_{k_l} \hat{A}[i_b, k_b, i_l, k_l] * \hat{B}[k_b, j_b, k_l, j_l]; \end{aligned}$$

This transformation is somehow similar to a strip-mining where $T[i_b, j_b, k_b, i_l, j_l]$ corresponds to the intermediate result of the accumulation over the k_bth tile. This transformation is always correct as a reduction applies an associative operator. With that tiling, each tile (i_b, j_b, k_b) of variable T computes a small matrix product. If we *outline* that tile computation into a function ϕ (implemented as a subsystem), we obtain a SARE of the form:

$$\hat{C}[i_b, j_b, i_l, j_l] = \sum_{k_b} \phi(\hat{A}[i_b, k_b, \bullet, \bullet], \hat{B}[k_b, j_b, \bullet, \bullet])$$

The original SARE structure is preserved, but the granularity of the computation has increased: the scalar product has been substituted by a matrix product ϕ. This transformation is called *semantic tiling*. This transformation belongs to the class of *semantic transformations*: the dependences are

not respected, but the transformed computation is equivalent modulo the associativity/commutativity of the reduction operator. The steps required are: (i) partitioning the domains and the index functions, (ii) tiling the reductions, (iii) outlining the tiles. The next chapter completes our semantic tiling with an equivalence checking tile/matrix operator to *recognize* the operator ϕ and let an optimized library implement it.
To ease the presentation, we first discuss the outlining step (iii), then we explain the reduction tiling (ii). This is because the outlining requires a tile classification which is updated by the reduction tiling.

2.4.1 Outlining the tiles

This section describes an *outlining transformation* that encapsulates the tiles into subsystems. In particular, we expose the underlying tile classification which will be used in Section 2.4.2 to tile the reductions.

Step 1 - Monoparametric tiling We apply the monoparametric tiling on the *domains* and the *index functions* of each equation of the program. Remark that the blocked and local indices are cleanly separated, as stated in Theorem 2.9. This property will be used in the outlining (step 3).

$$\begin{array}{ll}
(\forall \vec{i_b} \in \mathcal{D}_{bl,1})(\forall \vec{i_l} \in \mathcal{D}_{loc,1,1}) & Var[\vec{i_b}, \vec{i_l}] = \mathrm{SExpr}_{1,1}[\vec{i_b}, \vec{i_l}]; \\
(\forall \vec{i_b} \in \mathcal{D}_{bl,1})(\forall \vec{i_l} \in \mathcal{D}_{loc,1,2}) & Var[\vec{i_b}, \vec{i_l}] = \mathrm{SExpr}_{1,2}[\vec{i_b}, \vec{i_l}]; \\
\dots & \dots \\
(\forall \vec{i_b} \in \mathcal{D}_{bl,2})(\forall \vec{i_l} \in \mathcal{D}_{loc,2,1}) & Var[\vec{i_b}, \vec{i_l}] = \mathrm{SExpr}_{2,1}[\vec{i_b}, \vec{i_l}]; \\
(\forall \vec{i_b} \in \mathcal{D}_{bl,2})(\forall \vec{i_l} \in \mathcal{D}_{loc,2,2}) & Var[\vec{i_b}, \vec{i_l}] = \mathrm{SExpr}_{2,2}[\vec{i_b}, \vec{i_l}]; \\
\dots & \dots
\end{array}$$

After this step, the equation domains are tiled *separately*. The challenge is to outline the tile computation into subsystems out of this.

Step 2 - Extracting and classifying the tiles We proceed in two steps. First, *we tile the variables separately*: for each variable, we classify the tiles according to their computation. If two tiles share the *same equations*, then they can be realized by the same subsystem. This classification is called *kind of tiles*. Then, *we group the variables interdependent on the same tile*. Indeed, putting them into separate subsystems would break the atomicity constraint of the tiling. Hence, they have to be gathered on the same kind of tile. We keep track of this interdependence by classifying interdependent variables in the same *tile group*. These steps are detailed thereafter.

- *a) Tile the variables separately*. For each equation defining a variable Var, we retrieve the constraints $\mathcal{D}_{bl,k}$ on the blocked indices of the domain. The tile domains $\mathcal{D}_{bl,k}$ may intersect, hence they are *separated* to form a partition. For instance, the separation of $\mathcal{D}_{bl,1}$ and $\mathcal{D}_{bl,2}$ would be $\{\mathcal{D}_{bl,1} \setminus \mathcal{D}_{bl,2}, \mathcal{D}_{bl,1} \cap \mathcal{D}_{bl,2}, \mathcal{D}_{bl,2} \setminus \mathcal{D}_{bl,1}\}$. Each set of the partition defines a kind of tile. Indeed, each tile uses the same equations and then do the same computation. For example, tiles of $\mathcal{D}_{bl,1} \setminus \mathcal{D}_{bl,2}$ use equations with $SExpr_{1,\bullet}$, tiles $\mathcal{D}_{bl,1} \cap \mathcal{D}_{bl,2}$ use equations with $SExpr_{1,\bullet}$ and $SExpr_{2,\bullet}$, tiles $\mathcal{D}_{bl,2} \setminus \mathcal{D}_{bl,2}$ use equations with $SExpr_{2,\bullet}$.
- *b) Group interdependent variables*. For each variable of a tile group, we consider the family of block constraints $(\mathcal{D}^{Var}_{bl,k})_k$. The list of non-empty intersections of these families corresponds to the different kind of tiles. The corresponding equation of each one of these kind of tiles are the ones which contributes to the intersection.

Step 3 - Building the subsystems For each kind of tile, we build the subsystem which perform its computation. The equations can be obtained by removing the blocked dimensions of every variable and dependence functions. On the following equation:

$$(\forall \vec{i_b} \in \mathcal{D}_{bl})(\forall \vec{i_l} \in \mathcal{D}_{loc}) \quad Var[\vec{i_b}, \vec{i_l}] = f(\mathrm{Var}_1[u_{b,1}(\vec{i_b}), u_{l,1}(\vec{i_l})], \dots, \mathrm{Var}_k[u_{b,k}(\vec{i_b}), u_{l,k}(\vec{i_l})]);$$

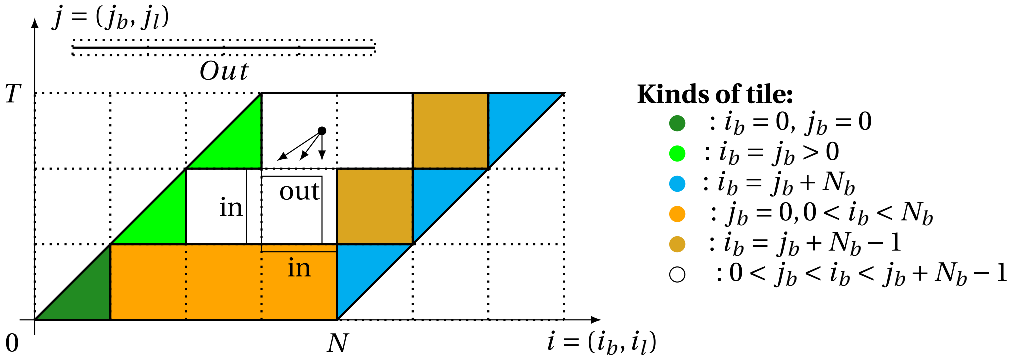


Figure 2.12 – Kinds of tile for a Jacobi1D skewed program

we would remove the blocked dimensions $\vec{i_b}$ to obtain:

$$(\forall \vec{i_l} \in \mathscr{D}_{loc}) \quad Var'[\vec{i_b}, \vec{i_l}] = f(\text{Var'}_1[\vec{i_l}], \ldots, \text{Var'}_k[\vec{i_l}]);$$

Note that this is possible only because the block and local indices are cleanly separated in a partitioned affine function, as shown by Theorem 2.9.
The inputs can be determined by examining the dependences of the computation of the subsystem. For example, if we have originally a dependence $Var[i_b - 1, j_b - 1, b - 1, b - 1]$, we can immediately deduce that we need a data from the block $(i_b - 1, j_b - 1)$ of the tile group of the variable Var. We create one input variable in the subsystem, per external block accessed in the computation of the subsystem. For the outputs of a subsystem, we determine which data from the tile is needed by other tiles. For example, if the data is used by $(i_b, j_b + 1)$ and $(i_b + 1, j_b + 1)$, we produce two outputs. Then, we emit the equations feeding the outputs from each variable A computed in the subsystem.

Step 4 - Building the main system Finally, we need to form the main system. In particular, we need to gather the outputs of the subsystems to send them as input of others. We first create use equation per subsystem generated, whose extension domain corresponds to the kind of tile. We also create one new local variable per outputs of the use equation, in order to retrieve the results of the subsystem. We also create local variables to gather the values of all the outputs of the same type and the same variable. These variables are used inside the input expressions of the use equations.

Example Consider a skewed Jacobi1D computation:

$$\begin{array}{l}
(\forall 0 < i < N)\ Out[i] = temp[T-1, i+T-1]; \\
(\forall t = 0, 0 < i < N)\ temp[t,i] = A[i]; \\
(\forall t = i > 0)\ temp[t,i] = temp[t-1, i-1]; \\
(\forall t > 0, i = N-1+t)\ temp[t,i] = temp[t-1, i-1]; \\
(\forall t > 0, t < i < N-1+t)\ temp[t,i] = (temp[t-1, i-2]+ \\
\qquad\qquad\qquad\qquad\qquad temp[t-1, i-1] + temp[t-1, i])/3;
\end{array}$$

We consider a monoparametric tiling $b \times b$ and we assume that N and T are divisible by the tile size parameter b. To make a rectangular tiling legal, the domain of the variable $temp$ has been skewed with a change of basis $(i, t) \mapsto (i + t, t)$. The domains are depicted in Figure 2.12.
There are no interdependence between variables: the data flows exclusively from $temp$ to Out. Consequently the tile groups are $\{temp\}$ and $\{Out\}$. The separation produces 20 differents equations gathered in 7 kind of tiles (subsystems): 6 for the $temp$ variable (listed in the figure 2.12), and one for the Out variable. Once we have determined the equations and the inputs of each subsystem, we

determine that the output of a tile are the 2 last columns on the right and the last row (needed for the right, above and the diagonal above-right tiles).
For example, the subsystem corresponding to the kind of tile • $(i_b = j_b + N_b - 1)$ consists of 10 equations. □

2.4.2 Tiling the reductions

In this section, we present the tiling transformation for the reduction operator, and we show how tile classification must be modified to preserve the atomicity of the tiling. Consider a reduction:

$$Var[\vec{i}] = \bigoplus_{\pi(\vec{i},\vec{j})=\vec{i}} Expr[\vec{i},\vec{j}]$$

where $\pi : (\vec{i},\vec{j} \mapsto \vec{i})$ is a *canonic* projection function. The reductions of an Alpha program can always be preprocessed to make their projection function canonic [98]. This way, the monoparametric tiling of π is always of the form $\hat{\pi}(\vec{i_b},\vec{j_b},\vec{i_l},\vec{j_l}) = (\vec{i_b},\vec{i_l})$. After tiling, this reduction becomes:

$$\hat{Var}[\vec{i_b},\vec{i_l}] = \bigoplus_{\vec{j_b}} \left(\bigoplus_{\vec{j_l}} \hat{Expr}[\vec{i_b},\vec{j_b},\vec{i_l},\vec{j_l}] \right)$$

We can separate the two reductions because the reduction operator is associative. To do so, we introduce a fresh variable $T[\vec{i_b},\vec{j_b},\vec{i_l}]$ to represent the intermediate result of the “summation” on one block of the reduction. T contains the result of the second “summation” in the previous equation. The original reduction equation becomes:

$$\hat{Var}[\vec{i_b},\vec{i_l}] = \bigoplus_{\vec{j_b}} T[\vec{i_b},\vec{j_b},\vec{i_l}]$$

The equation of T is:

$$T[\vec{i_b},\vec{j_b},\vec{i_l}] = \bigoplus_{\vec{j_l}} \hat{Expr}[\vec{i_b},\vec{j_b},\vec{i_l},\vec{j_l}];$$

After this preprocessing the program is almost ready to be processed by our outlining algorithm. There remains to decide in which tile group T should be put.

Tile groups and reductions We analyse the interdependences between the blocks of T and the blocks of $\hat{V}ar$. Then, we gather the interdependant blocks to preserve the atomicity constraint. By definition, the block $\hat{Var}[\vec{i_b}]$ reads the blocks $T[\vec{i_b},\vec{j_b}]$ for each $\vec{j_b}$. Hence, the blocks $T[\vec{i_b},\vec{j_b}]$ reading $\hat{Var}[\vec{i_b}]$ must be included in the same tile as $\hat{Var}[\vec{i_b}]$ to avoid interdependence. In general, if the block $T[\vec{i_b},\vec{j_b}]$ reads $\hat{Var}[f_k(\vec{i_b},\vec{j_b})]$ for $k = 1, n$, then the set of blocks to be included in the same tile as $\hat{Var}[\vec{i_b}]$ must contain at least the following set of tiles of T: $\{\vec{i_b},\vec{j_b} \mid f_1(\vec{i_b},\vec{j_b}) = \vec{i_b} \vee \ldots \vee f_n(\vec{i_b},\vec{j_b}) = \vec{i_b}\}$
We use this set to split the variable T into two variables: T_same (Same Group), corresponding to the tiles to be put into the same tile group as $\hat{Var}$, and T_sep, corresponding to the tiles which can be tiled separately.

Example Let us consider a program which solves the linear system $L.\vec{x} = \vec{b}$ where L is a lower-triangular matrix:

$$(\forall 0 \le i < N)\ x[i] = (b[i] - \sum_{k<i} L[i,k] \times x[k])/L[i,i];$$

We assume that x and $temp$ belong to the same tile group. The tiling step introduces a new variable T and transform the program into the following equations, assuming that the parameters are divisible:

$$\begin{array}{l}
(\forall 0 \le i_b < N_b)\ (\forall 0 \le i_l < b)\ x[i_b,i_l] = (b[i_b,i_l] - \sum_{k_b \le i_b} T[i_b,k_b,i_l])/L[i_b,i_b,i_l,i_l];\\
(\forall 0 \le i_b = k_b < N_b)\ (\forall 0 \le i_l < b)\ T[i_b,k_b,i_l] = \sum_{0 \le k_l < i_l} L[i_b,k_b,i_l,k_l] \times x[k_b,k_l];\\
(\forall 0 \le k_b < i_b < N_b)\ (\forall 0 \le i_l < b)\ T[i_b,k_b,i_l] = \sum_{0 \le k_l < b} L[i_b,k_b,i_l,k_l] \times x[k_b,k_l];
\end{array}$$

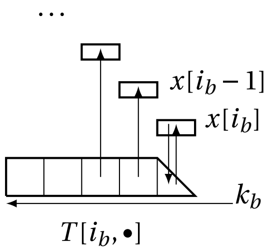


Figure 2.13 – Dependences between the tiles of T and the tiles of $x/temp$

Let us analyze the dependences involving T to decide in which tile group we should insert it. The only dependence which might introduce a cycle is the one corresponding to $x[k_b,k_l]$ in the equations of T (as shown in Figure 2.13). A cycle is introduced when $k_b = i_b$, thus we need to split this tile of T from the other tiles. Therefore, we obtain the following program after normalization:

$$\begin{aligned}
&(\forall 0 \le i_b < N_b)\ (\forall 0 \le i_l < b)\ x[i_b,i_l] = (b[i_b,i_l] - \sum_{k_b<i_b} T\_sep[i_b,k_b,i_l] \\
&\qquad\qquad - \sum_{k_b=i_b} T\_same[i_b,k_b,i_l])/L[i_b,i_b,i_l,i_l]; \\
&(\forall 0 \le i_b = k_b < N_b)\ (\forall 0 \le i_l < b)\ T\_same[i_b,k_b,i_l] = \sum_{0\le k_l<i_l} L[i_b,k_b,i_l,k_l] \times x[k_b,k_l]; \\
&(\forall 0 \le k_b < i_b < N_b)\ (\forall 0 \le i_l < b)\ T\_sep[i_b,k_b,i_l] = \sum_{0\le k_l<b} L[i_b,k_b,i_l,k_l] \times x[k_b,k_l];
\end{aligned}$$

We have two tile groups: one containing the variables (x and T_same), and another containing the variable T_sep. As a side note, we can notice that each tile of the first tile group correspond to a small forward substitution computation, and that each tile of the second tile group correspond to a small matrix multiplication: *we obtained a semantic tiling*. The next chapter will explain how to recognize these computations. □

2.4.3 Experimental results

In this section, we evaluate the scalability of our semantic tiling transformation. The next chapter will present an application of semantic tiling to program optimization and give final performance results.

Setup We have implemented the reduction tiling and the outlining transformations in the AlphaZ research compiler. We use the benchmarks from Polybench/Alpha[4] , an hand-written *Alpha* implementation of the *Polybench 4.0* benchmark suite. We run our experiment on a machine with an Intel Xeon E5-1650 CPU with 12 cores running at 1.6 GHz (max speed at 3.8GHz), and 31GB of memory. For each kernel, we apply the complete transformation (monoparametric tiling, reduction tiling, outlining). The default rectangular tiling is legal for all the benchmarks, except `gramschmidt` (where no legal tiling could be found) and all the stencils which required to skew the iteration domain beforehand.
We report the following informations:

- The time taken by the monoparametric tiling transformation (step 1).
- The time taken by the postprocessing of the monoparametric partitioned program by the alphaZ compiler. This housekeeping step is mandatory: it computes the the counter range reaching each subexpression of the program (context domain).
- The time taken by the outlining transformation (step 2: extracting and classifying the tiles, step 3: building the subsystems and step 4: building the main system).

[4] http://www.cs.colostate.edu/AlphaZsvn/Development/trunk/mde/edu.csu.melange.alphaz.polybench/polybench-alpha-4.0/

Time taken (ms)	correlation	covariance	gemm	gemver	gesummv	symm	syr2k	syrk	trmm	2mm	3mm	atax	bicg	doitgen	mvt	cholesky
Step 1) Tiling	166	102	109	91	53	190	110	69	66	233	433	71	217	152	62	165
Postprocessing	432	322	276	160	137	905	244	126	296	259	382	144	175	178	143	1139
Step 2) Kind of tiles	4	3	2	2	1	6	2	1	2	4	7	1	7	3	1	4
Step 3) Subsystems	382	277	282	75	55	660	176	112	87	646	1479	148	236	406	68	224
Step 4) Main System	34	18	12	18	7	19	13	9	8	25	44	12	13	18	9	25
Total Time	1206	817	754	457	309	1928	658	383	516	1297	2471	436	706	828	346	1806
Context Domain	1030	651	390	363	237	2617	456	274	277	632	837	871	273	300	265	1057
Num SubSystem	10	6	2	5	3	12	3	2	3	4	6	4	4	2	4	7
Av. Nodes in Subsystems	14	14	12	11	11	15	13	12	9	10	8	7	7	8	8	36
Num Equations Main	24	14	5	15	7	25	7	5	7	9	13	9	10	5	10	31

Time taken (ms)	durbin	gramschmidt	lu	ludcmp	trisolv	deriche	floyd-warshall	nussinov	adi	fdtd-2d	jacobi-1d	jacobi-2d	seidel-2d	heat-3d
Step 1) Tiling	177	–	313	450	52	329	72	248	1907	1301	119	529	538	6088
Postprocessing	1130	–	1522	1946	193	546	78	3605	18535	19138	867	19037	28497	$8m2s$
Step 2) Kind of tiles	4	–	6	7	1	4	2	8	72	122	4	28	28	297
Step 3) Subsystems	230	–	304	426	52	227	19	592	1874	2740	107	1764	2823	88267
Step 4) Main System	49	–	33	53	7	69	28	51	427	940	28	242	215	2905
Total Time	1772	–	2350	3188	363	1928	302	4689	23719	24871	1251	21859	32474	$9m40s$
Context Domain	2143	–	1555	2444	358	1719	629	4968	53348	98721	1252	28482	34515	$21m44s$
Num SubSystem	8	–	10	16	3	24	3	25	48	53	9	33	33	129
Av. Nodes in Subsystems	70	–	33	30	23	30	16	45	319	234	54	184	259	1029
Num Equations Main	63	–	44	67	9	59	15	118	676	1067	45	293	333	1985

Figure 2.14 – Time taken by our semantic tiling transformation inside the compiler. We also report the number of subsystems produced, the average number of nodes of a AST of a subsystem, and the number of equations in the main system.

- The total time spent in the transformation.
- The time taken by a polyhedral analysis on the transformed program: the computation of the context domains of all the subexpressions of the final tiled program.
- The number of subsystems generated.
- The average number of nodes in the AST of a subsystem.
- The number of equations inside the main system.

The result of our experiments are presented in Figure 2.14.

Results and analysis Most of the time is spent during the construction of the subsystems and the postprocessing of the program obtained after step 1 and before running step 2. This postprocessing step traverses the monoparametric partitioned program to compute the context domain of the sub-expressions of the form $Var[u_b(\vec{i_b}), u_l(\vec{i_l})]$. The construction of the subsystems also contains a traversal of the monoparametric partitioned program, in order to build the equations of the subsystems. The time taken by these steps is mostly caused by the size of the program after applying the monoparametric tiling transformation. We also notice that the time taken by a context domain computation following the monoparametric tiling transformation is reduced compared to the time taken by the same polyhedral analysis after the monoparametric tiling transformation (cf Figure 2.10 Page 30). Thus, distributing the computation across subsystem helps reducing the time taken by the polyhedral analysis on the transformed program.

2.5 Conclusion

In this chapter we summarized our contributions made to loop tiling. We extend loop tiling in several directions. First we show that parametric loop tiling is a polyhedral transformation when the tile shape and the lattice of tile origin depends on a single scaling parameter. We referred this transformation to as *monoparametric tiling*. We provide the compilation algorithms to transform a polyhedral representation with a monoparametric tiling: first with hyperrectangular tile shape, then with any polytopic tile shape. Finally, we describe an extension of monoparametric tiling to *semantic tiling*, a semantic transformation which increases the grain of operators (scalar → matrix).
All these contributions open many perspectives. We believe we opened a new path to partial compilation (referred to as *symbolic compilation* in [148]) by letting the tile parameter survive to the compilation. Also semantic tiling opens a new way to automated library refactoring. We present our contributions to this challenging problem in the next chapter. Finally, we believe that the numerous polyhedral algorithms leveraging a tiling can benefit from parametrization (this includes all the contributions described in this document). At least because it will make possible to integrate the computation of the tile size directly in the transformation.

3 Algorithm Recognition

Most compiler optimizations are based on low-level program transformations without any knowledge of the *meaning* of the program. Although they produce satisfactory results, they will never replace a good algorithm. Hence the idea to *recognize* algorithm instances in a program and to substitute them by a call to a performance library [2]. More generally, algorithm recognition allows to re-engineer a program and to build a hierarchical algorithmic description of a program [169], which could enable high-level algorithmic optimizations. Algorithm recognition is based on *program equivalence*, an old and well-known problem in Computer Science. Program equivalence has many other applications, such as program comprehension, program verification [84, 88], semi-automated debugging, compiler verification [97], translation validation [110, 123, 127, 178] to name just a few. This problem is known to be undecidable as soon as the considered program class is rich enough to be interesting. Moreover, if we account for the semantic properties of objects manipulated (e.g. associativity of +) in the program, the problem becomes harder.
During my PhD thesis, I have addressed the problem of *template recognition* and its application to program optimization [2, 3, 6, 7, 8, 9]. Aside the complexity of template matching, an important issue was to split the program into *slices* to be compared with the template, and substituted by a call to the library function. An important slicing criterion is that the slice must be *separable* from the program – in other words, *atomic*. Hence the idea of using iteration space tiling as a way to reveal library functions. This was the starting point of the PhD thesis of Guillaume Iooss [98], in the context of which all the ideas presented in this chapter were developed.

Summary and outline This chapter summarizes the contributions made to program equivalence and algorithm recognition. Section 3.1 recalls the notion of program equivalence considered and present the equivalence checking algorithm of Barthou et al. [37], whose concepts have largely influenced our work. Then the chapter presents three contributions to algorithm recognition. The contributions to template matching (Section 3.3) and template recognition (Section 3.4) are described in the PhD thesis of Guillaume Iooss [98], and are under publication:

- Section 3.2 discusses our contributions to check the equivalence of programs with *reductions* [24].
- Section 3.3 presents our contributions to *template matching*. We consider a relaxed form of template where the unknown functions are the inputs. A solution to a matching between a template and a program is a substitution of the template inputs so the template matches the program.
- Section 3.4 presents our method for algorithm recognition. We apply our semantic tiling algorithm to partition the computation into tiles, then we decompose the tile into templates applying iteratively our template matching algorithm. We report the results obtained by matching the BLAS [112] functions on several medium size linear algebra applications.

3.1 Program equivalence in the polyhedral model

In this section, we define the program equivalence considered in this chapter, the *Herbrand equivalence*. Then, we present the equivalence checking semi-algorithm of Barthou et al. [37] on which we built for the contributions presented in Section 3.2 and Section 3.3.

Program equivalence There several notions of program equivalence. The simplest one is called *Herbrand equivalence* [2] and corresponds to a structural equivalence of the computation. Two programs are Herbrand equivalent iff they compute the same mathematical formula, syntactically (no associativity, no commutativity involved for instance). In other words, if we consider the output values computed by a program as terms depending on the inputs, two programs are Herbrand equivalent iff they compute exactly the same terms. Herbrand-equivalence is typically preserved by any program transformation which satisfies data dependencies. Hence, Herbrand equivalence is sufficient to check the correctness of polyhedral program transformations [2]. The problem of checking the Herbrand equivalence of two SAREs is undecidable [37].
By definition, Herbrand equivalence does not consider any semantic property. For example, if we compare two programs, one computing $(a+b)+c$ and the other one $a+(b+c)$, they will not be Herbrand equivalent, because the operations performed are different. Likewise, a program computing $a+b$ will not be equivalent to a program computing $b+a$.
Yet, several versions of an algorithm are likely to exhibit such *semantic variations* [169], hence the need to go beyond Herbrand equivalence. Since my PhD thesis, a few approaches addressed this issue. Some extend Barthou's algorithm and manages associativity and commutativity over a finite number of elements [144] or by testing every permutation of the arguments of operators [163]. Some address non-parametrized computation by unrolling and normalization [103].
In Section 3.2, we propose a semi-algorithm to check the Herbrand-equivalence of programs with *reductions*, an abstraction presented in Chapter 2 which normalizes the accumulation of a pool of values with an associative/commutative operator. This way, we address all the semantic variations leveraging reductions. This concerns, in our opinion, most of the semantic variations modulo associativity/commutativity.
We now present Barthou's algorithm, which serves as a basis for the contributions described in this chapter.

Barthou's equivalence semi-algorithm Barthou et al. [37] proposed a semi-algorithm to check the Herbrand equivalence of two SAREs. It first builds an *equivalence* automaton encoding the equivalence problem and then studies the *accessibility set* of particular states of this automaton. Starting from the outputs of the two programs, the automaton unrolls the two computations and check that the operators are the same at each step. The final states of the automaton contains atomic comparisons between input arrays. Their accessibility set is computed to check that the array indices are the same. In a way, the equivalence automaton is an equivalence algorithm specialized on the two SAREs to compare, which is then executed symbolically to conclude.
The equivalence automaton is a Memory State Automaton [37] (also referred to as integer interpreted automata [10]):

Definition 3.1. *A* Memory State Automaton *(MSA) is a finite automaton where:*
- *Every state p is associated with an integer vector $\vec{v}_p$ of some dimension n_p.*
- *Every transition from p to q is associated with a firing relation $F_{p,q} \in \mathbb{Z}^{n_p} \times \mathbb{Z}^{n_q}$, which is a* polyhedral relation. *On our context,* this relation is always a function.
- *A transition from $\langle p, \vec{v}_p\rangle$ to $\langle q, \vec{v}_q\rangle$ can only happen if $(\vec{v}_p, \vec{v}_q) \in F_{p,q}$.*

We say that a state p is *accessible* iff there exists a finite path from the initial state p_0 to p for some associated vector. The *accessibility relation* of a state p is:

$$\mathscr{R}_p = \{(\vec{v}_0, \vec{v}_p) \mid \langle p_0, \vec{v}_0\rangle \to^* \langle p, \vec{v}_p\rangle\}$$

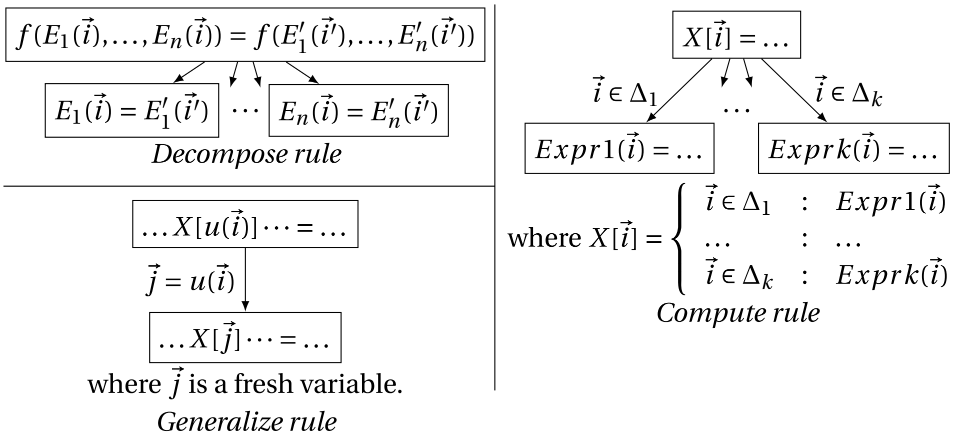

$f(E_1(\vec{i}),\dots,E_n(\vec{i})) = f(E'_1(\vec{i}'),\dots,E'_n(\vec{i}'))$
$E_1(\vec{i}) = E'_1(\vec{i}')$
$\cdots$
$E_n(\vec{i}) = E'_n(\vec{i}')$
Decompose rule
$\dots X[u(\vec{i})]\cdots = \dots$
$\vec{j} = u(\vec{i})$
$\dots X[\vec{j}]\cdots = \dots$
where $\vec{j}$ is a fresh variable.
Generalize rule
$X[\vec{i}] = \dots$
$\vec{i} \in \Delta_1$
$\dots$
$\vec{i} \in \Delta_k$
$Expr1(\vec{i}) = \dots$
$Exprk(\vec{i}) = \dots$
where $X[\vec{i}] = \begin{cases} \vec{i} \in \Delta_1 & : \; Expr1(\vec{i}) \\ \dots & : \; \dots \\ \vec{i} \in \Delta_k & : \; Exprk(\vec{i}) \end{cases}$
Compute rule

Figure 3.1 – Construction rules for the equivalence automaton

The *reachability set* of a state p is the range (image set) of the accessibility relation $\mathscr{R}_p$.
When the automaton has a cycle, the computation of the accessibility set involves the transitive closure of a polyhedral relation, which is not computable in general [51]. In most cases, the transitive closure can be computed with a fast heuristic [105, 162].

Step 1: Building the equivalence automaton: We assume that a correspondance is given between the outputs and the inputs of the two SAREs to compare. We use the convention that expressions, operators and indices of the second SARE are "primed" (e.g., X', E'). The equivalence automaton is defined as follows:

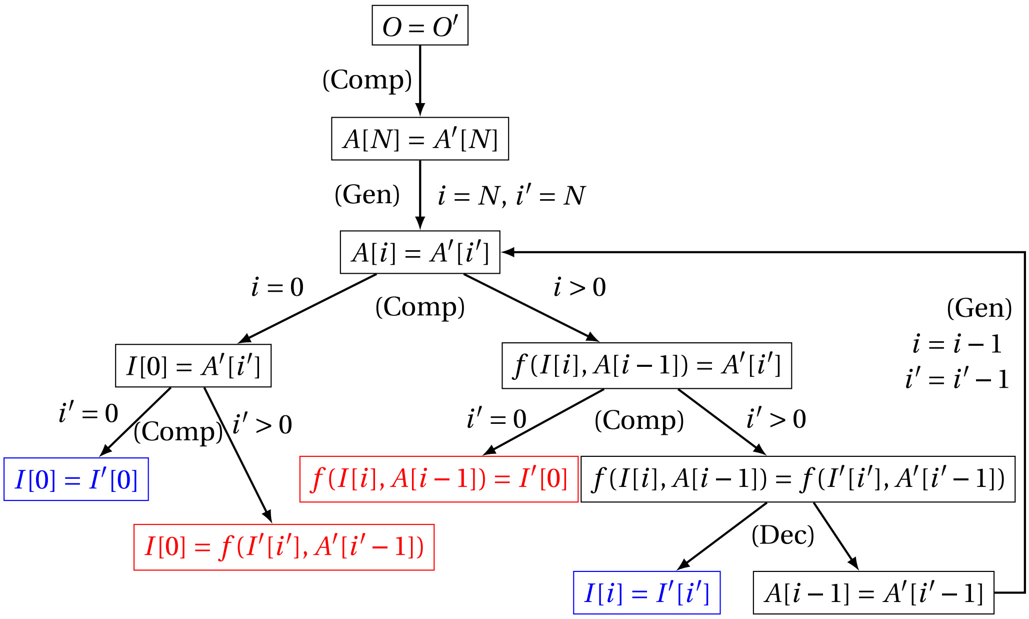
- **States:** A state corresponds to a point of comparison between the two SAREs. It is labeled by an equation $e(\vec{i}) = e'(\vec{i}')$ and is associated with the vector $(\vec{i}, \vec{i}')$.
- **Initial state:** The initial state of the automaton is $O[\vec{i_0}] = O'[\vec{i'_0}]$ where O and O' are the outputs of the two SAREs.
- **Transitions:** The transitions unroll the computation of the two SAREs and compare the operators. We have 3 types of transitions (*rules*) in the equivalence MSA: *Decompose*, *Compute* and *Generalize*, as described in Fig 3.1. The *Decompose* compare two operators and expresses that two expressions using the same operator are equivalent if their arguments are equivalent. The *Compute* executes a SARE by "unrolling" a definition and creates a state per case. Note that for each value $(\vec{i}, \vec{i}')$ associated with the source state, there is only one accessible state among the created states. The rules *Decompose* and *Compute* allow to traverse the two SAREs and to check the equivalence. The *Generalize* rule is useful to deal with recursions. It replaces an affine expression by a fresh index, allowing us to go into a state we may have already encountered, but with different index values.
- **Final state:** The final states of the automaton corresponds to comparisons where nothing can be eliminated or further unrolled. There are two kinds of final states: the *success states* and the *failure states*. The *accept states* are:
  - $f() = f()$ (i.e., two identical constants)
  - $I_k = I'_{k'}$ where I_k and $I'_{k'}$ are corresponding inputs.

  The *failure states* are:
  - $f(\dots) = f'(\dots)$ where f and f' are different operators,
  - $I_k[\vec{i}] = f'(\dots)$ or $f(\dots) = I'_k[\vec{i}']$,
  - $I_k[\vec{i}] = I'_{k'}[\vec{i}]$ where I_k and $I'_{k'}$ are not corresponding inputs.

Example. Let us compare the following SARE with itself:

$$\begin{array}{rcl} O & = & A[N] \\ A[i] & = & \left\{ \begin{array}{lcl} 1 \le i \le N & : & f(I[i], A[i-1]) \\ i = 0 & : & I[0] \end{array} \right. \end{array}$$

where O is the output of the SARE and I the input. The equivalence automaton is the following (success states are in blue and failure states in red):



The automaton has a cycle: it corresponds to the comparison between the recursions of both SARE. We can notice that, for every state of the automaton, we have $i = i'$ (indeed, for each transition we are modifying i, we are also modifying i' in the same way).

□

Step 2: Equivalence and reachability problem in the equivalence automaton: The two SAREs are Herbrand equivalent if they compute the same expression from the same inputs. Hence, we have to check than none of the failure states are reachable and that, for each accept state $I_k[\vec{i}] = I'_{k'}[\vec{i}']$, the two indices are the same: $\vec{i} = \vec{i}'$. This means that, if we compare the same element of the outputs, then when we end up on a reachable success state, the compared elements must be the same:

Theorem 3.2 (from [37]). *Two SAREs are equivalent iff, in their equivalence MSA:*

- *All failure states are not reachable from the start state: their accessibility set is empty.*
- *The accessibility relation of each* reachable *success state $I_k[\vec{i}] = I'_{k'}[\vec{i}']$ is included in the identity relation $\{(\vec{i}, \vec{i}') \mid \vec{i} = \vec{i}'\}$.*

Example (cont'd) The accessibility set of the failure states are empty (we can never have $i = 0 \wedge i' > 0$ or $i > 0 \wedge i' = 0$). Moreover, the equalities that need to be satisfied when reaching a success state are respectively $0 = 0$ (trivially satisfied) and $i = i'$ (satisfied). Thus, according to Theorem 3.2 these two SARE are equivalent. □

Limitation of the equivalence algorithm: This algorithm only checks Herbrand equivalence, semantic properties like associativity/commutativity of operators are not taken into account. For instance, if we try to compare the SAREs $O = I_1 + I_2$ and $O' = I'_2 + I'_1$, the equivalent automaton will have a *decompose* rule which will generate two failure states with respective labels $(I_1 = I'_2)$ and $(I_2 = I'_1)$.

3.2 Equivalence of programs with reductions

We now discuss our an extension of Barthou's equivalence algorithm to manage programs with reductions [24]. The main challenge in this extension is to find a bijection between the terms of two compared reductions, so that we can conclude for equivalence or not. We add the following construction rule (called *Decompose Reduce*):

$$\begin{tikzcd}
\boxed{\bigoplus_{\pi(\vec{k})=\vec{i}} E[\vec{k}] = \bigoplus_{\pi'(\vec{k}')=\vec{i}'} E'[\vec{k}']} \arrow[d, "\sigma(\vec{k}) = \vec{k}'"] \\
\boxed{E[\vec{k}] = E'[\vec{k}']}
\end{tikzcd}$$

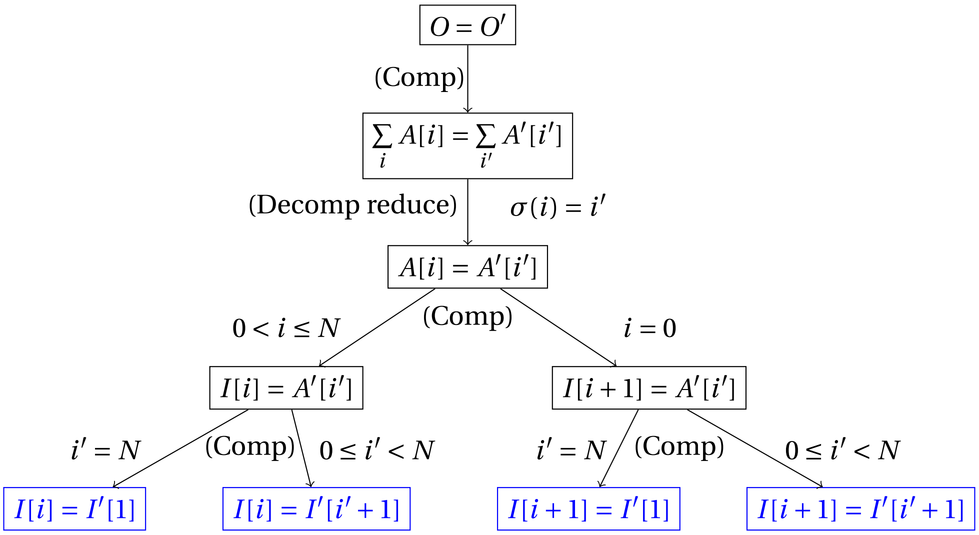
The idea of this rule is to map every term of the left reduction $E[\vec{k}]$ to an equivalent term $E'[\vec{k}']$ on the right reduction, such that these two instances are equivalent. If we manage to find a bijection σ between the instances $\vec{k}$ of the left reduction and the occurrences $\vec{k}'$ of the right reduction such that $E[\vec{k}]$ is equivalent to $E'[\vec{k}']$, then both reductions are equivalent. During the equivalence automaton construction step, we leave σ as a symbolic function (it does not impact the construction of the rest of the automaton). The rest of the algorithm will focus on inferring σ.

3.2.1 Illustrative example

To give the intuition of the algorithm, let us first consider the following SAREs:

$$\begin{array}{lcl}
O & = & \sum_i A[i] \\
A[i] & = & \begin{cases} 0 < i \leq N & : \quad I[i] \\ i = 0 & : \quad I[i+1] \end{cases}
\end{array}
\qquad\qquad
\begin{array}{lcl}
O' & = & \sum_{i'} A'[i'] \\
A'[i'] & = & \begin{cases} i' = N & : \quad I'[1] \\ 0 \leq i' < N & : \quad I'[i'+1] \end{cases}
\end{array}$$

These two SAREs sum up all the $I[i]$, except for $I[0]$, but with $I[1]$ being counted twice (for index points $i = 0,1$ in the first SARE, and for $i' = 0, N$ in the second one). Thus, when we compare the terms of the two bijections, we should map $A[0]$ to either $A'[0]$ or $A'[N]$ and $A[1]$ to the other one, and each remaining $A[i]$ to $A'[i']$. Let us derive these bijections from the equivalence automaton, which is shown below, where $\sigma : [\![0;N]\!] \mapsto [\![0;N]\!]$:



Extraction of the constraints: We want to find a bijection σ such that the conditions of Theorem 3.2 are satisfied, i.e., one that makes all failure state unreachable, and the indices of all success states equal. For example, let us consider the second success state $s : I[i] = I'[i'+1]$: we need $i = i'+1$ at this state, for each (i,i') of its accessibility set $\mathscr{R}_s = \{(i,i') \mid (0 < i \leq N) \wedge (0 \leq i' < N)\}$. By studying the accessibility set of each final state, we obtain the following constraints:

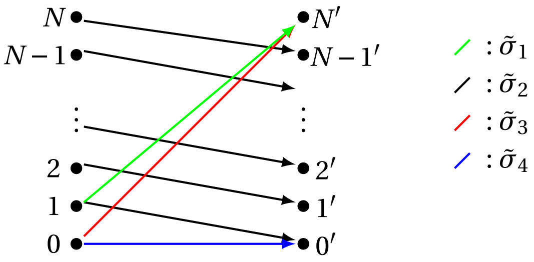
$$\begin{array}{lccccccccl}
 & [& \sigma(i) = i' & \wedge & (0 < i \leq N) & \wedge & (i' = N) & \wedge & i = 1 &] \\
\vee & [& \sigma(i) = i' & \wedge & (0 < i \leq N) & \wedge & (0 \leq i' < N) & \wedge & i = i'+1 &] \\
\vee & [& \sigma(i) = i' & \wedge & (i = 0) & \wedge & (i' = N) & \wedge & i+1 = 1 &] \\
\vee & [& \sigma(i) = i' & \wedge & (i = 0) & \wedge & (0 \leq i' < N) & \wedge & i+1 = i'+1 &]
\end{array}$$

Obtaining the partial bijections: Notice that our constraints admit a kind of structure: the constraints on i and i' are separated, except for the equalities coming from the success state itself. In our example, each Diophantine equation admits a unique solution, and we can express i' as a function of i:

$$\tilde{\sigma}_1 : \left\{ \begin{array}{ccc} \{1\} & \mapsto & \{N\} \\ 1 & \mapsto & N \end{array} \right. \quad \tilde{\sigma}_2 : \left\{ \begin{array}{ccc} [\![1;N]\!] & \mapsto & [\![0;N-1]\!] \\ i & \mapsto & i-1 \end{array} \right. \quad \tilde{\sigma}_3 : \left\{ \begin{array}{ccc} \{0\} & \mapsto & \{N\} \\ 0 & \mapsto & N \end{array} \right. \quad \tilde{\sigma}_4 : \left\{ \begin{array}{ccc} \{0\} & \mapsto & \{0\} \\ 0 & \mapsto & 0 \end{array} \right.$$

At this point, if σ is equal to $\tilde{\sigma}_k$ on its definition domain $\mathcal{D}_k$, then its success state is reachable from the reduction state for each $(\vec{i}, \tilde{\sigma}(\vec{i}))$ and the indices are equal. Hence, the conditions of Theorem 3.2 are satisfied for $\vec{i} \in \mathcal{D}_k$. The next paragraph explains how to find a combination of partial bijections into a *complete* bijection (which *covers all the values of* $\vec{i}$). If we can find such a bijection, then we can conclude the equivalence of the two SAREs.

Sticking the partial bijections: We can represent the definition and image domain of σ as the nodes of a bipartite graph and the partial bijections as the edges of this graph:



There is an edge from a node i to i' iff there is a partial bijection which maps i to i', i.e., iff selecting $\sigma(i) = i'$ satisfies the conditions of Theorem 3.2. Thus, finding a bijection σ which satisfies Theorem 3.2 is identical to finding a matching in this graph. In our example, we do not have any choice for $i \in [\![2;N]\!]$, but we can map $i = 0$ and 1 to either $i' = 0'$ or N'. Thus, we have two possible bijections:

$$\sigma : \left\{ \begin{array}{cclll} 1 & \mapsto & N & & (\tilde{\sigma}_1) \\ i & \mapsto & i-1 & \text{when } i \geq 2 & (\tilde{\sigma}_2) \\ 0 & \mapsto & 0 & & (\tilde{\sigma}_4) \end{array} \right. \quad \text{or} \quad \sigma : \left\{ \begin{array}{cclll} i & \mapsto & i-1 & \text{when } i \geq 1 & (\tilde{\sigma}_2) \\ 0 & \mapsto & N & & (\tilde{\sigma}_3) \end{array} \right.$$

We managed to build a bijection, thus we can conclude that the two reductions are equivalent. Therefore, the two considered SAREs are equivalent. In summary, our semi-algorithm proceeds in the three steps below, whose details may be found in [24]:

1. Extract the constraints on σ from the equivalence automaton. These informations are directly available on the reachability sets of the terminal states.
2. Transform these constraints into partial bijections $\tilde{\sigma}$, which are portions of σ where the equivalence constraints are satisfied. The challenge is to normalized the reachability set of terminal states to the form $(\sigma(i), i')$ where σ is a function. To do so, we leveraged mathematical recipes.
3. Combine these partial bijections to obtain σ. This is the most challenging step, since it amounts to find a matching on a bipartite graph with a parametric number of nodes. We proposed a *greedy heuristic* and an extension of the *augmenting path algorithm* [166].

3.2.2 Experimental results

We have implemented a prototype of the equivalence algorithm [1] on the *AlphaZ* polyhedral compilation framework. We have run our implementation on several examples, and we report their execution time in Figure 3.1.

[1] The prototype and the benchmarks are available at: http://cs.colostate.edu/AlphaZ/equivalence/index.html

Table 3.1 – Execution time (in milliseconds) that the algorithm spends in each phase for a few simple examples. Experiments were done on an Intel core i5-3210M running Linux.

Example	#states	Automaton	Partial Bijections	Gathering	Total
Illustrating example	15	74 ms	206 ms	133 ms	413 ms
Loop reverse	5	48 ms	56 ms	21 ms	125 ms
Distributed summation	15	75 ms	788 ms	297 ms	1160 ms
Non equivalence	4	45 ms	48 ms	84 ms	177 ms
Tiling	7	71 ms	245 ms	83 ms	399 ms

Every example manages parametric reductions, thus their equivalence cannot be decided by the previous work:

- *Loop reverse.* This example compares two 1D summations, one summing in increasing order and the other one in decreasing order.
- *Distributed summation.* This example compares two sums of reductions, summing the terms $I[i]$ for $0 \le i < 2N$ differently. In the first sum, the terms are split across the two reductions according to the parity of i. In the second sum, the terms are split between $0 \le i < N$ and $N \le i < 2N$.
- *Non Equivalence.* This example compares two reductions which are not equivalent (the second reduction has one extra term).
- *Tiling.* This example compares a 1D summation ($\sum_i I[i]$) with its tiled counterpart ($\sum_{ib} \sum_{il} I[16ib + il]$ where $0 \le il < 16$).

For all these examples, the greedy heuristic (for combining the partial bijections) happened to be sufficient to conclude in a reasonnable amount of time.
Finally, we did not use this algorithm in our algorithm recognition framework. We preferred to rely on a semantic tiling (which is correct-by-construction) to reveal the potential algorithm, and then use the template matching presented in the next section to retrieve the algorithms.

3.3 Template matching

In this section, we present our contribution to match a program to a template. A *template* is an algorithm whose inputs are unknown and might correspond to an arbitrarily elaborate computation. Given a template and a program, we look for a substitution of the template inputs by a computation so the template and the program become Herbrand equivalent. This problem is somehow similar to the matching problem over Herbrand algebra [34], with the notable difference that *we do not deal with terms, but with programs producing the terms to unify.*
In a way, we try to match the template to the head of the program's computation. This extension is a central building block of the template recognition algorithm presented in the next section.
One of the main challenges of template matching is to deduce these inputs. In particular, if an input appears in several places in a template, we should check that the corresponding computation is coherent across all of these places.

Motivating example Here is an example of template matching between a *program* and a *template.* Consider the *program*, corresponding to a serialized reduction over two arrays of size N (I_2 and I_1, I_2 being summed in the reverse order), and an element $I_0[0]$:

$$\begin{aligned} O =& \quad Temp[2N-1] \\ (\forall N \le i < 2N)\ Temp[i] =& \quad Temp[i-1] + I_2[2N-1-i] \\ (\forall 0 < i < N)\ Temp[i] =& \quad Temp[i-1] + I_1[i] \\ (\forall i = 0)\ Temp[i] =& \quad I_0[0] \end{aligned}$$

where I_0, I_1 and I_2 are inputs of the program.
Now, consider the following *template* (a serialized reduction along an array of size N'):

$$\begin{aligned} O' = \quad & Temp'[N'-1] \\ (\forall 0 < i' < N')\ Temp'[i'] = \quad & Temp'[i'-1] + I'[i'] \\ (\forall i' = 0)\ Temp'[i'] = \quad & I'[0] \end{aligned}$$

We expect the following *match*:

$$\begin{aligned} & (\forall i' = 0)\ I'[i'] = I_0[0] \\ & (\forall 0 < i' < N)\ I'[i'] = I_1[i'] \\ & (\forall N \le i' < 2N)\ I'[i'] = I_2[2N-1-i'] \end{aligned}$$

□

Our algorithm exploits the equivalence automaton introduced by Barthou to infer the match. We proceed into 3 steps, detailed thereafter. In Step 1, we build the equivalence automaton and we collect the states reaching a template input (here I'). In step 2, we extract the constraints on template inputs from the reachability sets. Finally Step 3 infers the match from the constraints.

Step 1 - Construction of the equivalence automaton We keep the construction rules of the equivalence automaton and we modify the notion of success and failure state of Barthou's equivalence automaton to account for the inputs of a template. Template expressions appear on the right hand side with a prime (e.g. $Expr'$):

- A *template-accept state* is a state which is labeled by an equation of the form $Expr = I'$, where I' is an input of the template. These states define a possible substitution of the template inputs: the template input $I'[\vec{i'}]$ might be defined by the program expression $Expr[\vec{i}]$ for each $(\vec{i}, \vec{i'})$ of the accessibility set.
- A *template-failure state* is a state which is labeled by an equation of either:
  - $f(\ldots) = f'(\ldots)$ where f and f' are different operators
  - $I = f'(\ldots)$ where f' is an operator and I is an input of the program

  In other words, the matching fails when the computation is not Herbrand equivalent ($f \neq f'$) or when the template does more computation than the program.

Because we assume that the output of the template matches the output of the program, it might impose some constraints on the parameters of the template (typically, both output arrays must be of the same size). We extract these constraints and keep them.

Example (cont'd). We obtain the following equivalence automaton, with five template-success states and one template-failure state depicted on Figure 3.2. The failure state $I_0[0] = Temp'[i'-1] + I'[i']$ describes the case where the template computation is longer than those of the program. □

Step 2 - Extracting the constraints on the inputs of the template We collect the constraints required to build the inputs of the template. The templates matches the program iff:

- *There exists a template input which satisfies all the accessible template-accept states.* For instance, the template $O = I' + I'$ does not match the program $O = I_1 + I_2$, as I' cannot have two different definitions ($I' = I_1$ and $I' = I_2$).

  Hence, we examine the automaton and we extract the constraints on the template inputs. For each template-accept state $Expr_k[\vec{i}] = I'[\vec{i'}]$, we keep its accessibility set $S_{I',k}$ which contains all the possible couples of $(\vec{i}, \vec{i'})$ reaching the state. The construction of the template input is achieved in the next step.
- *The template-failure states are not accessible.*

  Hence, if a template-failure state is accessible only for certain values of the template parameters, we extract the constraints on the template parameters which makes the corresponding accessibility set empty and consider them as constraints on the parameters of the template.

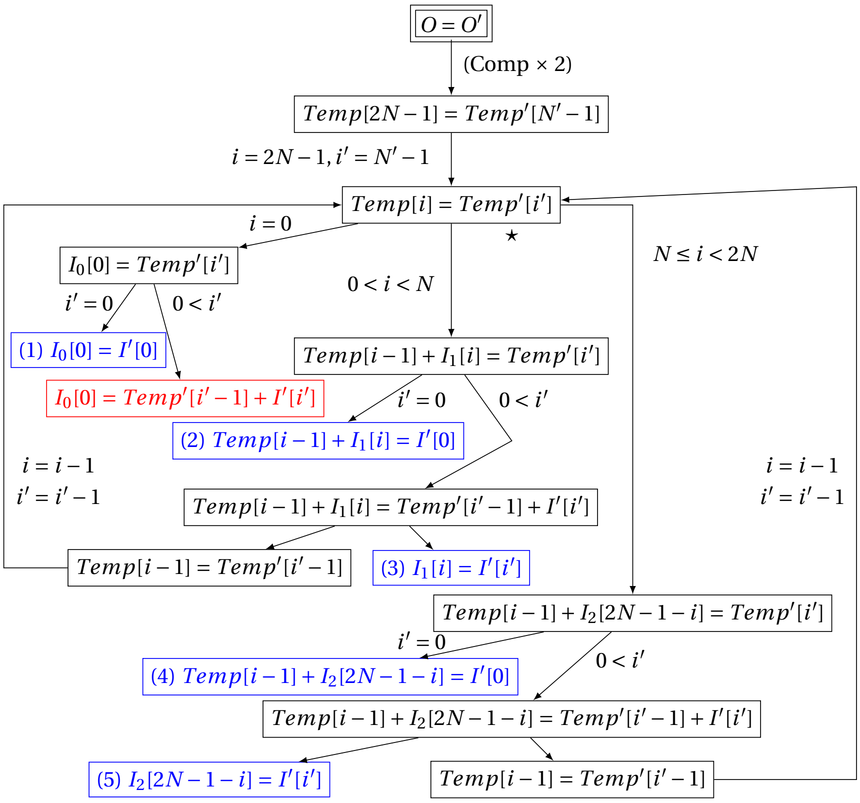

$O = O'$
(Comp × 2)
$Temp[2N-1] = Temp'[N'-1]$
$i = 2N-1, i' = N'-1$
$Temp[i] = Temp'[i']$
$i = 0$
$\star$
$N \leq i < 2N$
$I_0[0] = Temp'[i']$
$0 < i < N$
$i' = 0$
$0 < i'$
(1) $I_0[0] = I'[0]$
$Temp[i-1] + I_1[i] = Temp'[i']$
$I_0[0] = Temp'[i'-1] + I'[i']$
$i' = 0$
$0 < i'$
(2) $Temp[i-1] + I_1[i] = I'[0]$
$i = i-1$
$i' = i'-1$
$i = i-1$
$i' = i'-1$
$Temp[i-1] + I_1[i] = Temp'[i'-1] + I'[i']$
$Temp[i-1] = Temp'[i'-1]$
(3) $I_1[i] = I'[i']$
$Temp[i-1] + I_2[2N-1-i] = Temp'[i']$
$i' = 0$
$0 < i'$
(4) $Temp[i-1] + I_2[2N-1-i] = I'[0]$
$Temp[i-1] + I_2[2N-1-i] = Temp'[i'-1] + I'[i']$
(5) $I_2[2N-1-i] = I'[i']$
$Temp[i-1] = Temp'[i'-1]$

Figure 3.2 – Template matching automaton

Example (cont'd). The reachability sets of template-success states gives the following $S_{I',k}$ constraints. We recall the (number) annotating the state, it sets the k of $S_{I',k}$:
- (1), $S_{I',1}$: $(i,i') \in \{i,i'|i = i' = 0 \ \wedge \ i = 2N - N' + i'\}$ $I_0[0] = I'[0]$
- (2), $S_{I',2}$: $(i,i') \in \{i,i'|0 < i < N \ \wedge \ i' = 0 \ \wedge \ i = 2N - N' + i'\}$ $Temp[i-1] + I_1[i] = I'[0]$
- (3), $S_{I',3}$: $(i,i') \in \{i,i'|0 < i < N \ \wedge \ 0 < i' \ \wedge \ i = 2N - N' + i'\}$ $I_1[i] = I'[i']$
- (4), $S_{I',4}$: $(i,i') \in \{i,i'|N \leq i < 2N \ \wedge \ i' = 0 \ \wedge \ i = 2N - N' + i'\}$ $Temp[i-1] + I_2[2N-i-1] = I'[0]$
- (5), $S_{I',5}$: $(i,i') \in \{i,i'|N \leq i < 2N \ \wedge \ 0 < i' \ \wedge \ i = 2N - N' + i'\}$ $I_2[2N-i-1] = I'[i']$

We have one template-failure state $I_0[0] = Temp'[i'-1] + I'[i']$, whose accessibility set is $\{i,i'|i' = N'-2N+i \ \wedge \ i = 0 \ \wedge \ 0 < i'\} = \{i,i'|2N < N' \ \wedge \ i = 0 \ \wedge \ 0 < i'\}$. This set is empty iff $N' \leq 2N$. This means that the reduction we try to detect with our template *must* not be too long: $N' = 2N$ corresponds to detecting the whole program as a reduction (with a piece-wise input) and $N' < 2N$ corresponds to detecting only part of the program as a reduction.

□

Step 3 - Determining the inputs of the template We examine each input I' of the template and we try to determine a valid value from the associated constraints. For each $\vec{i'}$, we examine how many pairs $(k,\vec{i})$ are such that $I'[\vec{i'}] = Expr_k[\vec{i}], (\vec{i}, \vec{i'} \in S_{I',k})$, i.e., how many expressions $Expr_k[\vec{i}]$ are matched to the same $\vec{i'}$. In general, there is a parametric number of $\vec{i'}$, so it is not possible to iterate over all them. Instead, we can compute separately the projections of the $S_{I',k}$ on $\vec{i'}$ (referred to as $\mathscr{P}_{I',k}$), then consider the non-empty intersections pieces between a subset of these projected sets: if $\mathscr{P}_{I',k_1} \cap \mathscr{P}_{I',k_2} \neq \emptyset$, then both $Expr_{k_1}[\vec{i}]$ and $Expr_{k_2}[\vec{i}]$ must be mapped to $I[\vec{i'}]$, for all $\vec{i'} \in \mathscr{P}_{I',k_1} \cap \mathscr{P}_{I',k_2}$. If all the pairs mapped to the same $I'[\vec{i'}]$ are equivalent, we can select any of them. Another possibility is that the pairs are equivalent only for some values of the parameters: in that case, we extract the constraints on the parameters (by projecting the constraints on the parameters). If there exists a non-equivalent pair mapped to the same $I'[\vec{i'}]$ for any value of the parameters then the program does not match the template.

Final step If a value is found for every input of the template, and if the constraints on the parameters are satisfiable, then the template matches the program. In some situations, several values of the template parameters are valid: in that case, we choose to select the biggest values of the parameters, such that we match as much operations as possible from the program with the template.

Example (cont'd). Let us determine the value of the template input I'. By projecting the constraints $S_{I,k}$ on i', we deduce that:

- $I'[0]$ is bind by $S_{I',1}$, $S_{I',2}$, $S_{I',4}$, to values $I_0[0]$, $Temp[i-1]+I_1[i]$ and $Temp[i-1]+I_2[2N-i-1]$, respectively.
- $I'[i']$, $i'>0$ is bind by $S_{I',3}$ and $S_{I',5}$ to values $I_1[i]$ and $I_2[2N-i-1]$ respectively.

For each pool of constraints ($S_{I',1}$, $S_{I',2}$, $S_{I',4}$ for $i'=0$, and $S_{I',3}$ and $S_{I',5}$ for $i'>0$), we compute the parameter domain where each constraint apply. If the parameter domains are disjoint, then we have a match.

- For $i'=0$, the constraint $S_{I',1}$ imposes that the template parameter N' is equal to $2N$. The constraint $S_{I',2}$ imposes that $0<2N-N'<N$, i.e., $N<N'<2N$ and the constraint $S_{I',5}$ imposes that $N\le 2N-N'<2N$, i.e. $0<N'\le N$. Therefore, these 3 constraints are disjoints, and we have:

$$I'[0]=\begin{cases} N'=2N & : \quad I_0[0] \\ N<N'<2N & : \quad Temp[2N-N'-1]+I_1[2N-N'] \\ 0<N'\le N & : \quad Temp[2N-N'-1]+I_2[N'-1] \end{cases}$$

For $i'>0$, the constraint $S_{I',3}$ imposes $0<2N-N'+i'<N$, i.e., $N'-2N<i'<N'-N$. Because we have already determined that $N'\le 2N$, $0\le i'<N'-N$. The constraint $S_{I',5}$ imposes $N\le 2N-N'+i'<2N$, i.e., $N'-N\le i'<N'$. Thus, both of them are disjoints and we have:

$$I'[i']=\begin{cases} 0<i'<N'-N & : \quad I_1[2N-N'+i'] \\ N'-N\le i'<N' & : \quad I_2[N'-1-i'] \end{cases}$$

Therefore, the template matches for any $N'\le 2N$. To maximize the part of the program covered by the template, we pick $N'=2N$, which gives us, as the input of the template:

$$\begin{array}{l} (\forall i'=0)\ I'[i']=I_0[0] \\ (\forall 0<i'<N)\ I'[i']=I_1[i'] \\ (\forall N\le i'<2N)\ I'[i']=I_2[2N-1-i'] \end{array}$$

Therefore, we conclude that the template matches. □

Over-approximations and soundness This matching algorithm relies on a transitive closure, which might not be exact. When the transitive closure is over-approximated, our algorithm is still sound:

- We consider the negation of the accessibility set of a template failure-state to extract constraints on the parameters. If the accessibility set of a template failure-state is over-approximated, these constraints might be more restrictive than needed, but are sound.
- If the accessibility set of a template accept-state is over-approximated, then the input constraints would span a larger domain than needed. It might create an intersection with a conflicting non-equivalent constraint and might fail the algorithm. Hence, the approximation might only produce false negatives.

3.4 Template recognition

In this section, we combine our semantic tiling transformation (Chapter 2) with our template matching algorithm (Section 3.3) in order to recognize instances of templates from a template

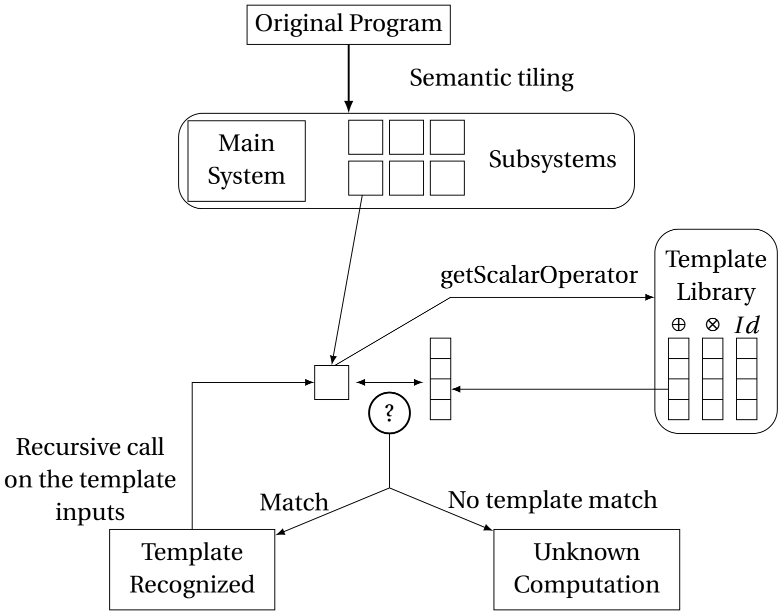


Figure 3.3 – Template recognition procedure

library (Section 3.3). These instances are then substituted by a call to the template library. This provides a complete algorithm to refactor a linear algebra program with matrix operations. In turn, these matrix operations might be realized with an optimized library. The next section will show the acceleration obtained using the BLAS library. The whole process is shown in Figure 3.3.
Thanks to SARE normalization, we are not bounded by syntactic artifact unlike *pattern matching* approaches for algorithm recognition [48, 106, 119]. Since the SARE provide an *exact* dependence representation, this makes our approach more precise than techniques based on statement-level dependence graphs [125, 169]. As far as we know, this approach is, with our PhD, the only existing approach for algorithm *recognition* leveraging the power of the polyhedral model.

3.4.1 Method

Step 1: Template library preprocessing In order to accelerate the recognition process, we classify the library templates according to their *scalar operator*: the operation obtained when the size of matrices and vectors are set to 1. For instance, the scalar operator of a matrix multiplication would be ×; the scalar operator of a Cholesky decomposition would be $x \mapsto \sqrt{x}$.
This gives a simple *hash function*, that will accelerate the recognition process: since we match library templates to program tiles, we set the program tile size to 1 to obtain the scalar operator, and we deduce the potential templates to match.

Step 2: Semantic tiling We apply the semantic tiling transformation to the program. We use hypersquare tiles for every variable (identity tile ratio).
This transformation produces a main program and a list of *subsystems* corresponding to *tile computations*. Thus, *we consider each subsystem* independently in the rest of the procedure.

Step 3: Template recognition We compare each subsystem with the templates of the library, using our template matching algorithm described in Section 2.4.2. If a subsystem is matched to a template, we build a node corresponding to this template. Then, we consider the expressions of the subsystem that correspond to the inputs of the template. If the expression is not an input variable of the subsystem or a constant, we build a new subsystem which corresponds to the reminder of the computation and apply our procedure recursively on this new system. Finally, we retrieve

the produced *tree* of templates and link it to the node of the recognized template.
The output of our procedure is a *tree of templates*. Each node of this tree corresponds to a template, whose inputs are the children of this node. The leaves of the template tree are either an input of the program, a constant, or a non recognized computation. Figure 3.5 gives several trees of templates produced by our algorithm.

3.4.2 Experimental results

In this section, we present an experimental validation of our our algorithm recognition framework. We apply our framework to detect BLAS calls into medium-size linear algebra applications. This assess the scalability and the recognition capabilities of our framework.

Template library We consider a modified version of the BLAS library [112] specified in Figure 3.4. Level 1 operators output a scalar, level 2 operators output a vector, and level 3 operators output a matrix. We will focus only on the double precision variant, but our approach can easily be extended to any other data types.
In the original BLAS specification, most of the operations are scaled with a coefficient α. To reduce the number of templates, we assume that $\alpha = 1$ everywhere, and add the operations $C \leftarrow \alpha.A$ (where $\alpha \neq 1$). A post-processing can be used to merge these operations, if they are detected in succession, so that a single BLAS kernel can be used instead of two.
We also notice that BLAS has many variants of the same operation, depending on whether or not one of its argument is transposed. To reduce the number of variants, we separate the transpose operation. Once again, a post-processing can merge the transpose operation with the matrix multiplication operation, if these operations are detected in succession.

Applications We study the detection of templates in four applications:

- *Symmetric Positive semi-Definite Matrix Inversion*. This application computes the inverse of a symmetric positive semi-definite A thanks to Cholesky decomposition $A = L.L^T$ and triangular matrix inversion $A^{-1} = L^{-T}.L^{-1}$. This application has a total of 7 equations.
- *Sylvester Equation Solver*. This application computes the solution of a *Sylvester equation* $A.X + X.B = C$, where A, B and C are given square matrices, and X is an unknown square matrix. This application has a total of 4 equations.
- *Algebraic Path Problem*. The Algebraic Path Problem (APP) [36] is a graph algorithm which can be viewed as a generalization of the Floyd-Warshall algorithm. This application is defined by 6 equations.
- *McCaskill*. This application is a subset of the computation of a bioinformatics application called *piRNA* (Partition function of Interacting RNAs [60]). This application has a total of 11 equations.

The experiments presented in this section were run on a machine with an Intel Xeon E5-1650 CPU with 12 cores running at 1.6 GHz (max speed at 3.8GHz), and 31GB of memory.

Results Table 3.2 summarizes the results. We give the number of subsystems obtained after the semantic tiling (#subsystems), the number of comparisons template/program (#matchings), the total number of states for all the equivalence automata built (states), the number of templates detected (#templates detected) and the total time ellapsed (Total time).

SPDMI. We have detected 27 templates in this application, if we ignore the 3 “transpose” nodes that precede a “non-recognized” node. The corresponding template trees are presented in Figure 3.5. We managed to recognize completely the computation, except the Cholesky decomposition, which was not part of our template library. The time taken by the monoparametric tiling is about 6 seconds.
Sylvester. We have detected 8 templates in this application. These templates cover completely the subsystems created from reductions, which contains the majority of the computation of the pro-

Extra:
- Transpose: $C \leftarrow A^T$
- Scalar multiplication - vector : $\vec{y} \leftarrow \alpha.\vec{x}$ where $\alpha \notin \{0,1\}$
- Scalar multiplication - matrix : $C \leftarrow \alpha.A$ where $\alpha \notin \{0,1\}$
- Addition - vector : $\vec{y} \leftarrow \vec{x}_1 + \vec{x}_2$
- Addition - matrix : $C \leftarrow A + B$
- Reduction - vector : $\vec{y} \leftarrow \sum\limits_k \vec{x}_k$
- Reduction - matrix : $C \leftarrow \sum\limits_k A_k$

Level 1:
- DSCAL : $y \leftarrow \alpha.x$
- DDOT : $\alpha \leftarrow \vec{x}^T.\vec{y}$

Level 2:
- DGEMV : $\vec{y} \leftarrow A.\vec{x}$
- DSYMV : $\vec{y} \leftarrow S.\vec{x}$ where S is symmetric
- DTRMV : $\vec{y} \leftarrow L.\vec{x}$ where L is lower-triangular
  $\vec{y} \leftarrow U.\vec{x}$ where U is upper-triangular
- DTRSV : $\vec{y} \leftarrow L^{-1}.\vec{x}$ where L is lower-triangular
  $\vec{y} \leftarrow U^{-1}.\vec{x}$ where U is upper-triangular
- DGER : $A \leftarrow \vec{x}.\vec{y}^T$
- DSYR : $A \leftarrow \vec{x}.\vec{x}^T$
- DSYR2 : $A \leftarrow \vec{x}.\vec{y}^T + \vec{y}.\vec{x}^T$

Level 3:
- DGEMM : $C \leftarrow A.B$
- DSYMM : $C \leftarrow S.B$ or $C \leftarrow B.S$ where S is symmetric
- DSYRK : $C \leftarrow A.A^T$
- DSYR2K : $C \leftarrow A.B^T + B.A^T$
- DTRMM : $C \leftarrow L.B$ or $C \leftarrow B.L$ where L is lower-triangular
  $C \leftarrow U.B$ or $C \leftarrow B.U$ where U is upper-triangular
- DTRSM : $C \leftarrow L^{-1}.B$ or $C \leftarrow B.L^{-1}$ where L is lower-triangular
  $C \leftarrow U^{-1}.B$ or $C \leftarrow B.U^{-1}$ where U is upper-triangular

Figure 3.4 – List of templates in our library, after simplification

gram. The time taken by the monoparametric tiling is about 7 seconds.
APP. We have detected 44 templates in this application. The operations detected are mostly matrix multiplications ($A.B$, $B.U$ where U is upper-triangular, diagonal matrix multiplication), but also some matrix and vector additions, point to point multiplications, reduction on a vector ($\vec{y} = \sum_k \vec{x}_k$). We managed to recognize the totality of the computation of 5 of these subsystems, over 6. The time taken by the monoparametric tiling is about 20 seconds.
McCaskill. We have detected 80 templates in this application. However, only 8 of them are not a "transpose" preceding a non-recognized computation. This poor result can be explained by the fact that using a linear algebra library for this computation is not a good fit. However, the computation of many subsystems have the same kind of structure. Thus, we might be able to identify a common operator which can be recognized over many subsystems. The time taken by the monoparametric tiling is about 57 seconds.

Application	#subsystems	#matchings	#states	#templates detected	Total time
SPDMI	16	200	9815	27	8 min
Sylvester	8	28	1602	8	2 min
APP	60	660	13054	44	27 min
McCaskill	113	2245	90812	80	1 h 10 min

Table 3.2 – Template recognition: experimental validation

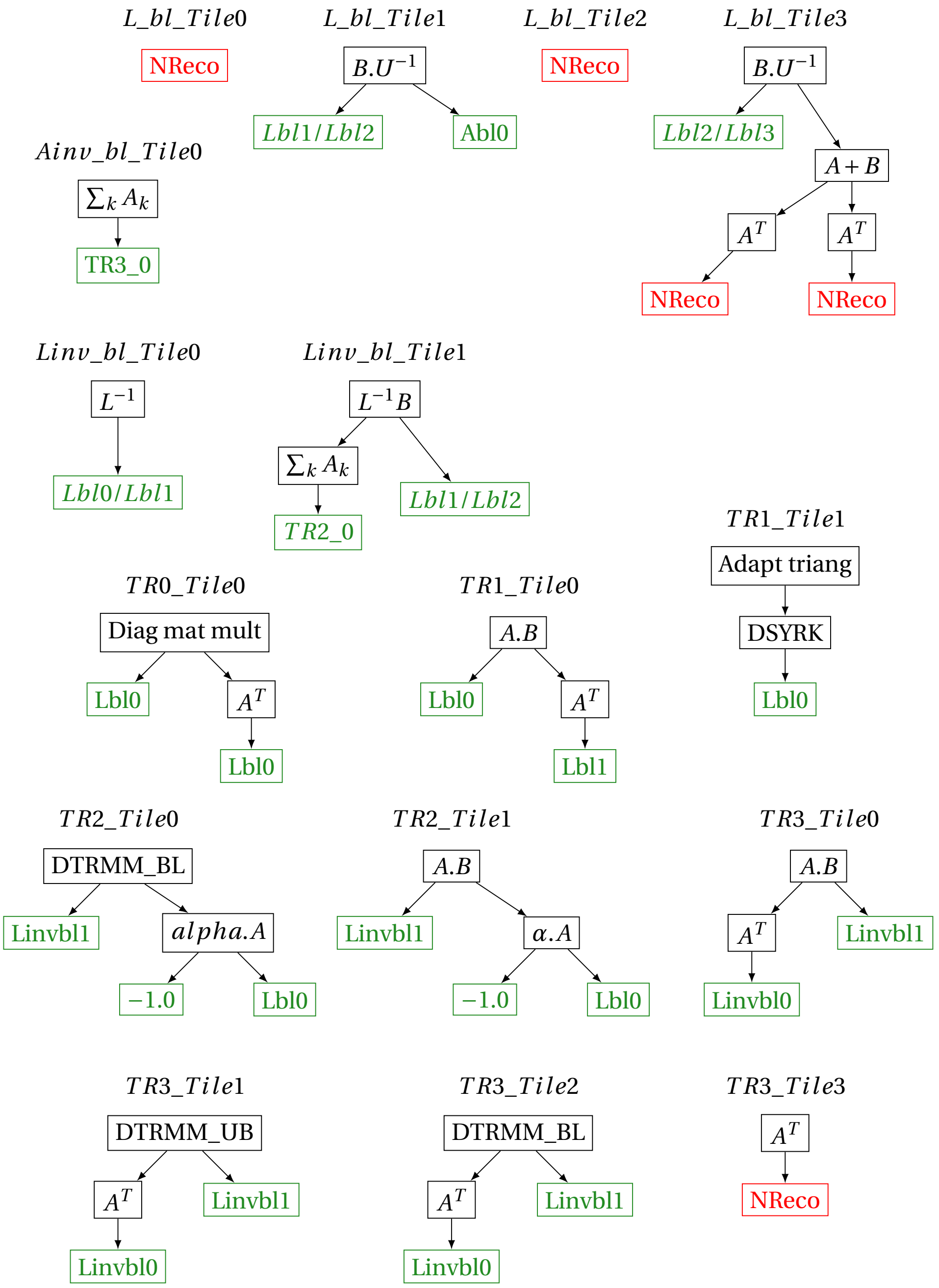


Figure 3.5 – Output of our template recognition framework: trees of recognized templates for each subsystem found in the SPDMI example. The nodes in green correspond to the input of the template, a constant, or a switch between inputs and constants. The nodes in red correspond to the non-recognized computation.

3.5 Conclusion

In this chapter we summarized our contributions made to algorithm recognition for the purpose of program refactoring with a performance library. We address the problem of program equivalence modulo associativity/commutativity by handling *reductions*. Then, we propose an algorithm recognition framework leveraging the semantic tiling transformation, presented in Chapter 2. We show how the semantic tiles might be decomposed in a *template tree* by leveraging a template matching algorithm.

In the future, we plan to explore the possible performance models to choose a relevant refactoring among all the possible factorization found by our algorithm. Semantic tiling is a building block for more general semantic transformations. We plan to model and to mechanize the notion of semantic transformation where we believe semantic tiling will play an important role. From a more fundamental point of vue, we plan to address and to explain the apparent fractality of semantic tiling – a beautiful and unexpected feature.

Part II

Models and Algorithms for High-Level Synthesis

4 Communication Synthesis

Since the end of Dennard scaling, high-performance computing applications are accelerated by offloading kernels to specialized hardware accelerators such as GPGPU or FPGA. With FPGA (field programmable gate array) accelerators, the designer must specify directly a low-level *circuit configuration*.

High-level synthesis tools are compilers able to produce such a circuit configuration from a higher level of abstraction (such as C) [29, 56, 90, 116, 156]. High-level synthesis tools are now quite efficient for generating finite-state machines, for exploiting instruction-level parallelism, operator selection, resource sharing, and even for performing some form of software pipelining, for one given kernel. In other words, it is acceptable to rely on them for optimizing the heart of accelerators, i.e., the equivalent of back-end optimizations in standard (software) compilers.

However, the designer has still the responsibility to restructure the code and the communications so the circuit can complete its computation without waiting for the off-chip memory. In particular, the *operational intensity* [67, 168] of the code must be tuned so it becomes *compute-bounded*, which is possible only when the code exhibits enough data reuse. Also, the usual solutions to overcome latency (prefetch) and bandwidth (local data reuse) must be *specialized* on the offloaded kernel to gain efficiency and silicon surface.

How to decompose an application into smaller communicating processes, how to define the adequate memory organization or communicating buffers, and how to integrate all processes in one complete design with suitable synchronization mechanisms is extremely difficult, time-consuming, and error-prone. This Chapter presents our contributions to these challenging issues.

Summary and outline This chapter summarizes our contributions to the compilation of data transfers with an off-chip memory in the context of high-level synthesis for FPGA. All these results were obtained in the context of the PhD thesis of Alexandru Plesco [126]. Section 4.1 outlines the challenges related to memory access and to the HLS tool chosen for this study. Then:

- Section 4.2 identifies the features that make DDR optimization hard to perform, presents our template of architecture for optimized data transfers, and show how this architecture can be specified as a C program in front of the HLS tool [12, 13].

  The next sections presents our contributions to compile the components of this architecture from a C program.
- Section 4.3 presents our algorithm to schedule the data transfers [14, 15] with the the external memory so as to reduce communications and reuse data as much as possible in the accelerator.
- Section 4.4 outlines our contribution to local memory allocation [5].

4.1 Background

This section briefly outlines the challenges related to C2H, the HLS tool we chose to develop our source-to-source optimization ; and the DDR memory, to which we want to optimize the accesses.

4.1.1 C2H at a glance

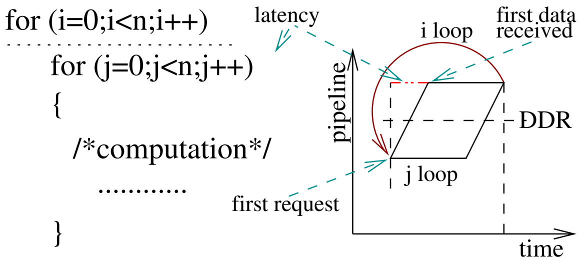
Some HLS tools rely on a quite direct mechanism to map the C syntax elements to the corresponding hardware, e.g., encoding a loop with a simple finite-state machine (FSM) instead of unrolling it, mapping each scalar variable to a register and each array to a distinct local memory, etc. This may seem a limitation but, at the same time, it gives a mean to control what the HLS tool produces, which is particularly important when used with source-to-source preprocessing, as we do. This is one of the reasons why we chose C2H [56], the HLS tool designed by Altera Corporation, as a target for our source-level optimizations.

The accelerator is controlled by several synchronized FSMs, one for each function or loop. Each loop is software-pipelined to optimize its CPLI (cycles per loop iteration). Memory transactions are pipelined with an optimistic latency (the FSM stalls if the data arrives later) and implicit fifos are created to store transferred data.

Dependence analysis in C2H is limited to an analysis of "names", with no analysis of array elements. Some potential aliasing can be removed, thanks to the pragma `restrict` but, still, this weakness is a real difficulty for source-to-source transformations.

4.1.2 Optimization challenges at a glance

Data fetch penalty If a loop contains another loop, the FSM of the outer loop stalls at the cycle containing the inner loop and waits for its end. In particular, it waits for the communications to be completed, as illustrated in Figure 4.1 for 2 nested loops. This is referred to as *data fetch penalty*.



```
for (i=0;i<n;i++)
   for (j=0;j<n;j++)
   {
     /*computation*/
       ...........
   }
```

Figure 4.1 – Latency penalty for an outer loop

Row penalty The DDR memory is organized in matricial banks, in turn organized in rows [101]. When accessing to a memory element, the corresponding row is selected, which takes times. Then, the elements of the row can be accessed. On the DDR used in our study (DDR-400 128Mbx8, size of 16MB, and CAS of 3 at 200MHz memory clock), accessing a different row takes at least 65 ns, while successive accesses to the same row would take only 10 ns.

4.2 A solution with communicating accelerators

We first show, with the vector sum example, why direct approaches do not work, due to C2H features/limitations. We then detail our solution, based on multiple accelerators that orchestrate communications, and how the required transformations and code generations can be automated. Our compiler algorithms are then described in Section 4.3.

4.2.1 First attempts toward a solution

Strip-mining and loop distribution To get accesses per block, a natural solution is to use strip-mining and loop distribution as follows:

```
for (i=0; i<MAX; i=i+BLOCK) {
  for(j=0; j<BLOCK; j++) a_tmp[j] = a_in[i+j];//prefetch
  for(j=0; j<BLOCK; j++) b_tmp[j] = b_in[i+j];//prefetch
  for(j=0; j<BLOCK; j++) c_out[i+j] = a_tmp[j] + b_tmp[j];
}
```

The two prefetching loops occur in parallel since there is no data dependence between them, thus requests of a and b are still interleaved. To avoid the *data fetch penalties*, one can unroll the inner loops and store each data read in a different scalar variable. If BLOCK is 8, the i loop has a CPLI of 16 and performs 16 read accesses, optimally. It happens (is it a feature of the scheduler of C2H?) that the data are fetched in the textual order of the requests. The downside of this approach is the code explosion and hence the resource need explosion. Also, it requires a non parametric unrolling factor.

Juggling A more involved solution, similar to the juggling technique [71], is to linearize the 3 inner loops into one loop k, emulating the desired behavior thanks to an automaton that retrieves the original indices. The code then looks like:

```
int ptr_local, ptr_ddr;
bi = 0; j = 0;
for (k=0; k<3*MAX; k++) {
 if (j==2) { *(c_out+i+bi) = *(a_tmp+i) + *(b_tmp+i); }
 else {
   if (j==0) { ptr_ddr = a+(i+bi); ptr_local = a_tmp+i;}
   else { ptr_ddr = b+(i+bi); ptr_local = b_tmp+i;}
   *ptr_local = *ptr_ddr; /* data transfer */}
 if (i++==BLOCK) { i=0; if (j++==3) {j=0; bi += BLOCK}}
}
```

Unfortunately, due to false dependences, C2H is now unable to pipeline the accesses. If the restrict pragma is added for ptr_local and ptr_ddr, the loop is pipelined, with CPLI equal to 1, but in some cases, depending on the scheduler and of runtime latencies, the code is incorrect: the computation should start after the last data of array b has arrived, not just after the request itself. The restrict pragma is too global, it cannot express the restriction between only two pointers. Also, with C2H, data requests in if instructions are still initiated, speculatively, so as to enable their pipelining, which, here, leads to interleaved reads and writes again. We tried many other variants of this code, with different pointers, different writing, trying to enforce dependences when needed and remove false dependences. We did not find any satisfactory solution. Either the code is potentially incorrect, depending on the schedule, or its CPLI increases, or it is not pipelined at all.

4.2.2 Multiple accelerators to overlap communications and computations

The previous discussion shows that it is inefficient, if not impossible, to write the code managing the communication in the code managing the computation.

If it is placed in a previous loop, the accelerator has to wait for the data to arrive before starting the computation (data fetch penalty). If it is interleaved within the computation code, controlled by an automaton as for the juggling technique, it is very difficult to ensure that this extra housekeeping code does not alter the optimal data rate. A natural solution would implement the data transfer in a single-loop accelerator, synchronized with the computation accelerator. However, since all instructions from a loop are guaranteed to be executed only when the loop state machine finishes its execution, it is again not possible to enforce data coherency with a correct synchronization between the two accelerators. This suggests to split the computation in *blocks*, and to synchronize the communications between two blocks.

All these considerations pushed us toward a more involved solution that we now expose.

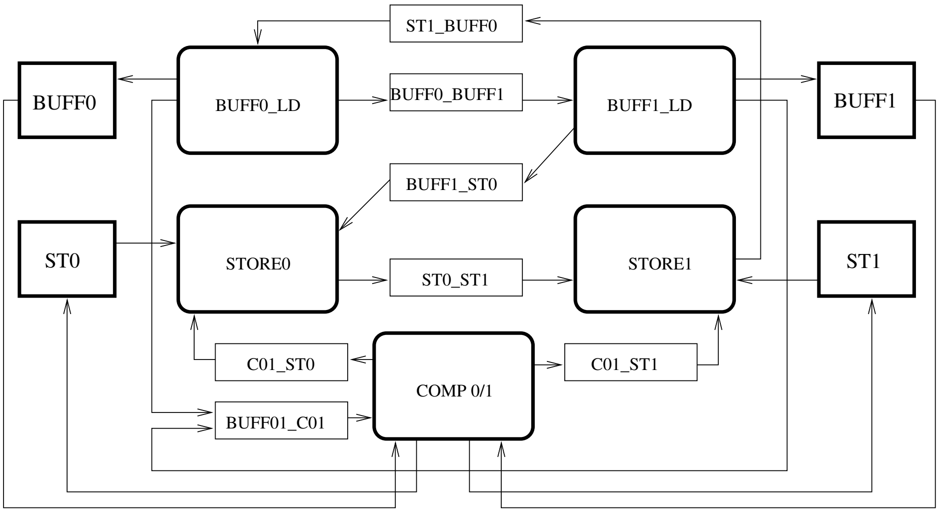
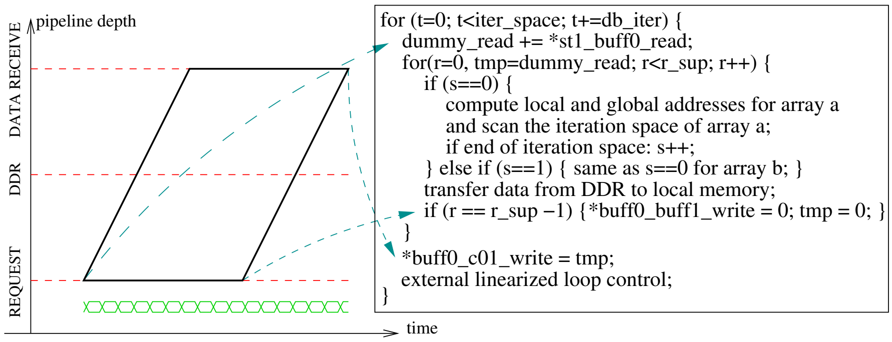


Figure 4.2 – Accelerators module architecture

Template architecture The template architecture is presented in Fig. 4.2. The generated accelerators are represented as bold rounded rectangles, local memories as normal bold rectangles, and the rest are FIFOs. In this design, FIFOs are used only for synchronization. The arrays on which the computations are performed are located in local memories.
The computation is split into *blocks*, then, the data required for a block of computations are transferred using a double buffering approach implemented by two accelerators `BUFF0_LD` and `BUFF1_LD`. This approach allows to use single-port local memories for buffers `BUFF0` and `BUFF1`, which consume fewer resources and are preferred over dual port ones, even if they are now usually available on FPGA platforms.

Template accelerator Fig. 4.3 shows the template code of the `BUFF0_LD` accelerator. The code has two nested loops. The outer loop iterates over the blocks (tiles), and each tile loads a block of array `a`, and a block of array `b`. The inner loop uses the same mechanism as the juggling code of Section 4.2.1 to emulate the traversal of the read requests, in the right order. After the desired local and external addresses are computed, the data is transferred from external to local memory. Here, as there are only reads, the code can be fully pipelined with CPLI equal to 1. The same holds for the writing accelerators.

Synchronizations The key point is how the `BUFF0_LD` accelerator is synchronized, at C level, with the other ones. Before each invocation of the inner loop, the accelerator performs a blocking read from a synchronization fifo. Since C2H may schedule independent instructions in parallel, we guarantee that the inner loop starts after this synchronization by introducing a dependence with



```
for (t=0; t<iter_space; t+=db_iter) {
  dummy_read += *st1_buff0_read;
  for(r=0, tmp=dummy_read; r<r_sup; r++) {
    if (s==0) {
      compute local and global addresses for array a
      and scan the iteration space of array a;
      if end of iteration space: s++;
    } else if (s==1) { same as s==0 for array b; }
    transfer data from DDR to local memory;
    if (r == r_sup −1) {*buff0_buff1_write = 0; tmp = 0; }
  }
  *buff0_c01_write = tmp;
  external linearized loop control;
}
```

Figure 4.3 – Simplified template C code

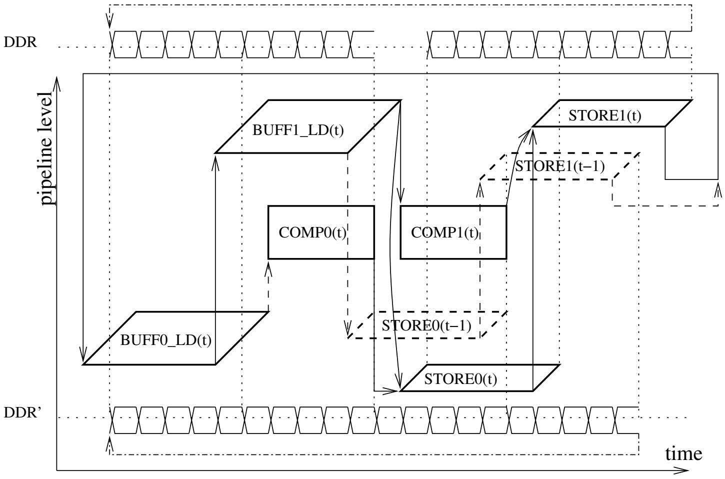


Figure 4.4 – Synchronization diagram

the variable `dummy_read` (`dummy_read += ...`). At the last iteration of the innermost loop, the accelerator has finished sending requests to the memory, and a synchronization token is sent so that another accelerator can start requesting data from the memory controller (`if(r == r_sup-1) ...`). When the innermost loop state machine receives all the requested data from the memory, a synchronization token is sent to the computation accelerator. Similar to `dummy_read`, the variable `tmp` guarantees that this synchronization takes place after the loop.
With this generic technique, it is possible to fetch, in an optimized blocked manner, many blocks of different sizes, each with its individual access addresses, without increasing the hardware resources too much (the only increase is the state machine size of the inner loop). Another advantage is that we can dispatch one or multiple arrays to multiple memories. This should be used jointly with optimizations of the computation accelerator so that parallel computations can be performed on data from different local memories. This point is addressed in the next Chapter.

Pipelined execution Fig. 4.4 shows a possible synchronization of the whole system, with two kinds of synchronizations, due to data dependencies (e.g., from `BUFF0_LD` to `COMP0`) and due to resource utilization (e.g., from `BUFF0_LD` to `BUFF1_LD`). Here, the `DDR` transfers are still not optimal: there is a small gap between the load and the store, due to the conservative synchronization between `BUFF1_LD` and `STORE0`. We indeed assumed here, for the sake of illustration, that `STORE0` could write to the location that `BUFF0_LD` reads. Without this assumption, the synchronization could be moved to the last request of `BUFF0_LD`. However, if the computation finishes later, there is still a gap due to the synchronization between `COMP0` and `STORE0`. It can be eliminated by reducing the computation time with parallelization techniques. Or one can shift all stores to the right (i.e., to delay them by one iteration) as depicted with dotted lines in Fig. 4.4. But this requires duplicating the local memory of the computed data, as `COMP0(t)` now overlaps with `STORE0(t-1)`. To avoid the gap when `COMP0(i)` delays `STORE0(i)`, it is possible to find a solution without an overlap between `STORE0(i-1)` and `COMP0(i)`, thus with no extra memory duplication, as shown by the (non intuitive) *software pipeline* of Fig. 4.5.

4.2.3 Automation of the process

Section 4.2.2 defined the synchronization mechanisms that enable to pipeline communications by blocks, avoiding both the row change penalty, due to the `DDR` specification, and the data fetch penalty, due to the way C2H schedules nested loops.
Compiler algorithms are required to fill out the accelerators templates and to allocate the buffers. We list here the main steps that are necessary, as well as related references. Then, the next sections

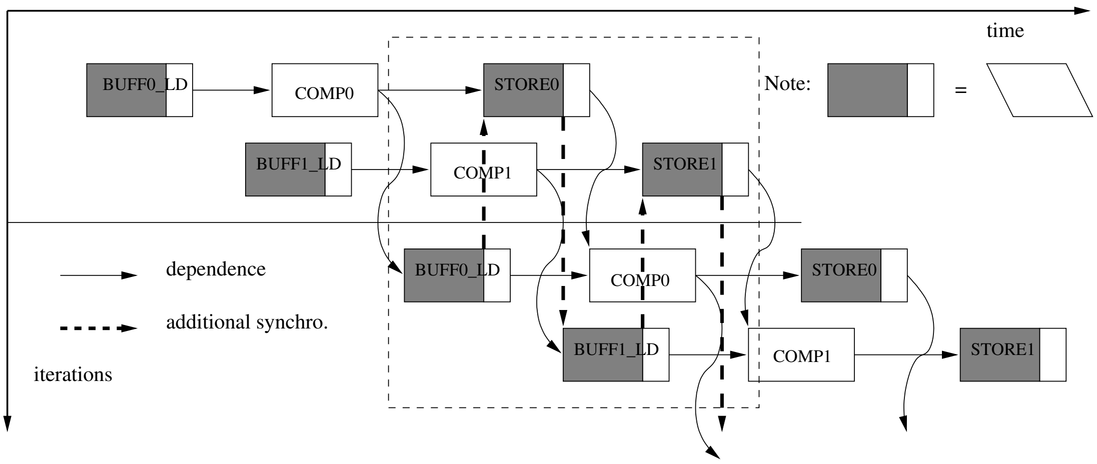


Figure 4.5 – Software pipeline

will describe our contributions to achieve these steps.
Section 4.3 describes our algorithms for steps 1 and 2. Then, Section 4.4 presents our algorithm for steps 3.

Step 1. Loop tiling First, loop tiling divides the kernel into elementary blocks of computations to be executed with a double buffering scheme. For the matrix-matrix product, this corresponds to a block-based version. The double buffering scheme is then performed along the last loop describing tiles (loop unrolling by 2). A good tiling should reduce the volume of necessary data transfers, as well as the sizes of the local memories needed (memory footprint optimization), while trying to leave some data in local memory from one tile to another (temporal reuse). Many approaches exists in the HPC community to compute such a tiling (see references in [172]). We focus on *polyhedral programs* where the *tiling* transformation is well formalized and general enough to handle non perfectly-nested loops [53]. Also, we consider only loop tiling with *parallelepipedic tiles* (see Chapter 2, Section 2.1.4).

Step 2. Communication coalescing The second step is to identify, for a given tile, the data to read from and to write to the DDR, excluding those already stored locally or that will be overwritten before their final transfer to the DDR (as the array c in the product of polynomial given Figure 2.1). This is a particular form of *communication coalescing* as described in [58], for HPF, to host communications outside loops (here the loops describing one tile). Then, the set of transferred data is scanned [42, 54], preferably row by row. Even if an array is accessed by column in a tile, as for the matrix-matrix product, the corresponding data can be transferred by row.

Step 3. Contraction of local arrays Then, a mapping function must be defined that convert indices of the global array (in the DDR) to local indices of a smaller array in which the transferred data are stored. Standard array contraction techniques [5, 75, 114] can be used for that.

Step 4. Code specialization The final transformation is to replace nested loops that scan data sets or that define the computations in a tile by a linearization, as in the juggling code of Section 4.2.1. This, again, is to avoid any data fetch penalty. Also, once the different accelerators are defined, synchronizations (`fifo_read` or `fifo_write`) are placed as explained in Section 4.2.2.

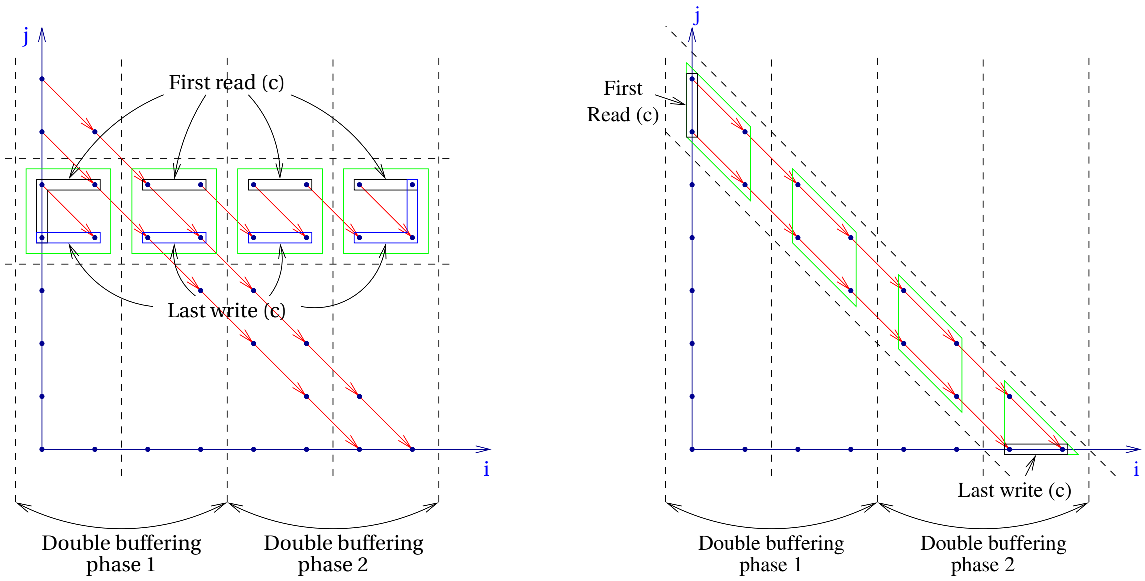


Figure 4.6 – Different tilings and communications

4.3 Communication scheduling

This section presents our compiler algorithms to split the computation into blocks to be executed with the double-buffering scheme (step 1) ; and to derive of the set of data to be loaded/stored for each block so the communication volume is minimized and the liveness of local values is reduced (step 2).

Tile bands Recall that, after loop tiling, the iteration domain is partitioned into blocks (tiles) of iterations to be executed atomically. Also, the loop structure consists of *tile loops* $(I_1,\ldots,I_n)$, which iterate over the tiles, and the *intra-tile loops* $\vec{i}$, which iterate within a tile.
A *tile band* is the set of tiles described by the *innermost tile loop* (along I_n), for a given iteration of the outer tile loops $\vec{I}_{band} = (I_1,\ldots,I_{n-1})$.

Example. Consider the following code, which computes the product of two polynomials stored in arrays a and b.

```
    for (i=0; i<=2*N; i++)
S1:    c[i] = 0;

    for (i=0; i<=N; i++)
      for (j=0; j<=N; j++)
S2:      c[i+j] = c[i+j] + a[i]*b[j];
```

The offloaded kernel is the set of nested loops containing S_2. Figure 4.6 depicts a *tile band* for two possible tilings: on the left: $\phi : (i,j) \mapsto (N-j,i)$, on the right: $\phi' : (i,j) \mapsto (i+j,i)$ □

Our approach We consider tile bands as *reuse units*: inside a tile band, data are communicated through the *local memory* ; between two tile bands, data are communicated through the *off-chip memory*. This is a usual partitioning for this purpose [77, 104]. To reduce the liveness of array elements, hence the size of local memory, we load the array elements right before their *first read* (black boxes in Figure 4.6) and we store the array elements *right after their last write* (blue boxes in Figure 4.6).
The tiles of a band are executed two-by-two, following the execution scheme presented in Section 4.2. This way, the execution of each tile T is decomposed into three pipelined processes, for loading

data, storing data, and performing the computations. Also, two successive blocks of computation in a tile band are pipelined with two blocks of communications, which results in an overlapping between communications and computations.

Example (cont'd).

- Consider the tiling on the left. There is maximal inter-tile reuse of b within a tile band (along the j axis), maximal intra-tile reuse of a within a tile (along the i axis), and some intra- and inter-tile reuse for c between two successive tiles. In black boxes are shown the elements of c that must be loaded by each tile, if maximal data reuse is exploited, and in blue boxes those that must be stored back by each tile.
- Consider the tiling on the right. The data dependences on c are always kept in the tile band. This way, the loads and stores for array c only arise on the first and last tiles of the tile band. Notice that the loads and stores for the array a are the same in both cases. However, the number of transfers for array b now increases compared to the first tiling. As an illustration, for this second tiling, the tiled iteration domain is $\hat{\mathscr{D}}_{S_2} = \{(I_1, I_2, i, j) \mid (i, j) \in \mathscr{D}_{S_2}$ and $bI_1 \leq i + j < b(I_1 + 1)$ and $bI_2 \leq i \leq b(I_2 + 1)\}$ and the full sequential schedule of iterations, θ, is $(I_1, I_2, i, j) \mapsto (I_1, I_2, i, j)$. The tile bands are defined by the outer tile loop counters $\vec{I}_{band} = (I_1)$.

□

This way, we outperform the usual approaches for communication coalescing [38, 58, 59, 63, 94, 100, 115, 120] which load, just before executing a tile, all the data read in the tile, then store to the DDR all the data written in the tile. With our approach, we exploit not only intra-tile reuse but also *inter-tile reuse*, even if data dependences exist between tiles, *at the granularity of individual array elements*. Furthermore, we load from (resp. store to) the DDR any data read (resp. written) in the current tile band *only once*. As a bonus, this method handles naturally the case where dependences exist between tiles of a tile band.

Also, our scheme inherits the capabilities of tiling to explore the trade-off communications/local memory simply by playing with the tiling (tile shape, tile size).

Transfer sets The *transfer sets* $\textsc{Load}(T)$ and $\textsc{Store}(T)$ express the data to be loaded/stored before/after the execution of a tile indexed by T. For a tile T, let $\textsc{In}(T)$ be the data read in T, but not defined earlier in the tile, i.e., used in T and live-in for T, and let $\textsc{Out}(T)$ be the data written in T. We assume $\textsc{In}(T)$ and $\textsc{Out}(T)$ to be exact: they can always be derived with polyhedral techniques. The following theorem gives a solution where loads are performed as late as possible and stores as soon as possible. This has the effect of minimizing the lifetime of data in the local memory, which tends to reduce its size.

Theorem 4.1. *The functions* $\textsc{Load}$ *and* $\textsc{Store}$ *defined by*

- $\textsc{Load}(T) = \textsc{In}(T) \setminus \{\textsc{In}(t < T) \cup \textsc{Out}(t < T)\}$
- $\textsc{Store}(T) = \textsc{Out}(T) \setminus \textsc{Out}(t > T)$

avoid useless transfers and reduce lifetimes in local memory.

Method with first read/last write We use an equivalent formulation where the first read and the last write of an array element $\vec{m}$ are computed explicitly. We define:

- $\textsc{FirstRead}(\vec{m})$, the *first operation of the tile band* accessing $\vec{m}$ as a read. This remove the first reads preceded by a write in the tile band. Indeed, it would be incorrect to load an element already defined in the tile band.
- $\textsc{LastWrite}(\vec{m})$, the *last operation of the tile band* accessing $\vec{m}$ as a write.

The scope of $\textsc{FirstRead}(\vec{m})$ and $\textsc{LastWrite}(\vec{m})$ is a *tile band* of the iteration domain, defined by the outer tile loop counters $\vec{I}_{band} = (I_1, \ldots, I_{n-1})$. $\vec{I}_{band}$ is kept as a *parameter* in the iteration domains considered in the following. It is an implicit parameter of $\textsc{FirstRead}$ and $\textsc{LastWrite}$.

Theorem 4.1 can then be reformulated as follows:

Theorem 4.2. *The operators of Theorem 4.1 can be defined as:*
- $\text{LOAD}(T) = \{\vec{m} \mid \text{FIRSTREAD}(\vec{m}) \in T\}$
- $\text{STORE}(T) = \{\vec{m} \mid \text{LASTWRITE}(\vec{m}) \in T\}$

$\text{FIRSTREAD}(\vec{m})$ is obtained by first extracting the set of operations accessing $\vec{m}$. Then, by computing the access that is scheduled first (with respect to θ, the tiled schedule) in the tile band. This boils down to compute the lexicographic minimum in a union of polytopes [14, 15].

Example (cont'd). Consider Figure 4.6 with the left tiling and the transfer sets for the array c. Our technique determines that the elements be loaded are those depicted in black boxes ("First read (c)"), and that the elements to be stored are those depicted in blue boxes ("Last write (c)"). For $b = 10$ we obtain:
- $\text{FIRSTREAD}(m) = (0, m)$ if $0 \leq -10I_1 + N - m \leq 9$ (vertical portion of c);
- $\text{FIRSTREAD}(m) = (10I_1 - N + m, N - 10I_1)$ if $1 \leq 10I_1 - N + m \leq N$ (horizontal portion).

$\text{FIRSTREAD}(m)$ is parametrized by the considered tile band I_1.
Finally, we derive $\text{LOAD}(T)$ as the set of data m accessed by the first read iterations of T:

$$\{m \mid \max(0, N - 10I_1 - 9) \leq m \leq N - 10I_1,\ T = 0\} \cup$$
$$\{m \mid \max(1, 10T) \leq m + 10I_1 - N \leq \min(N, 10T + 9)\}$$

□

Since this work, several approaches were developed to optimize the communications with off-chip memory, by enabling parametric tiling size [69] or relaxing the load/store domains [40, 45, 130]. Our approach relies on a subtraction of polyhedra, which tends to produce large unions of polyhedra. Hence, relaxations, or abstractions are required. The right level of abstraction is still to be found. For instance, [130] relaxes our equations but still uses a subtraction. As for [40, 45], no reuse is allowed between tiles, which limits the impact.

4.4 Local storage management

This section shows how the local memories (size and access function) are defined with respect to the software-pipelined schedule of the processes. With our method, all the computations use variables from the local memory. The lifetime of a variable starts with it first access (possibly from a load operation) and ends at its last access (possibly from a store operation). Variables are mapped to the local memory so that (i) two data alive at the same time are not mapped to the same local address, (ii) the local memory size is as small as possible.

Array contraction Unlike the methods developed in [93], which try to pack data optimally (in size), possibly with complex and expensive mapping functions and reorganization, we rely on *array contraction* based on modular mappings [72, 75, 114]: an array cell $a(\vec{i})$ is mapped to a *local* array cell $a_l(\sigma(\vec{i}))$ where $\sigma(\vec{i}) = A\vec{i} \bmod \vec{b}$, A is an integer matrix, and $\vec{b}$ is an integral vector defining a modulo operation component-wise. When the array index functions are translations w.r.t. the loop indices, as in `a[i][j-1]`, the set of live array cells is a window sliding during a tiled program execution. This is handled efficiently with the modulo.
In a polyhedral compiler, array contraction produces an *allocation*, which maps SARE array elements to physical storage, providing the liveness constraints induced by a schedule (schedule → allocation). In a way, array contraction counteracts array expansion (of SARE arrays) under scheduling constraints. A fundamental application of array contraction is the *allocation of communication channels in process networks*. This will be addressed in the next Chapter.

Back to the main example. Consider the left tiling of Fig. 4.6. Communication scheduling informs us that every tile (I_1, I_2) must load the following region of a to the local variable a_l:

$$a_l \longleftarrow \text{LOAD}_a(I_1, I_2) = a[bI_2 : bI_2 + b - 1]$$

where b denotes the tile size (on the figure, $b = 2$). Meanwhile, the double buffering process loads the data for the next tile in the band, $(I_1, I_2 + 1)$:

$$a_l \longleftarrow \text{LOAD}_a(I_1, I_2 + 1) = a[bI_2 + b : bI_2 + 2b - 1]$$

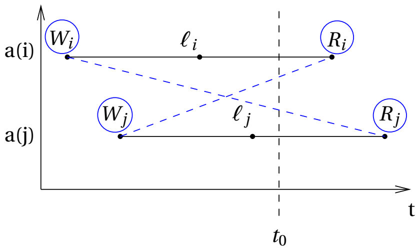
This means that *at the same time*, a_tmp needs at most $2b$ array cells. The following *allocation* satisfies these constraints, while minimizing the local storage size:

$$\sigma : a[i] \mapsto a_l[i \bmod 2b]$$

□

This way, it is sufficient to encode the software-pipelined schedule of the processes with a polyhedral program / schedule in order to derive the allocation function for each array of the kernel.

Array liveness analysis We defined a polyhedral analysis to compute array liveness from a polyhedral program with respect to any polyhedral schedule [5]. The output is a *conflict polyhedron*, a set of differences $\delta = \vec{i} - \vec{j}$ such that $a(\vec{i})$ conflicts with $a(\vec{j})$ according to the schedule, which serves as an input to array contraction techniques [72, 114].
We consider all the configurations where $a(\vec{i})$ conflicts with $a(\vec{j})$:



To be exact, $[W_k, R_k]$ should be a life interval, hence W_k must be the write defining R_k. However, this requires an exact array dataflow analysis to be settled [81], which leads to complex domains (possibly one per piece of the source function).
Hence, we propose the following conservative approximation. Instead of the last writes, we consider *any* write W_i of $a(\vec{i})$ executed before R_i. This over-approximation has the effect of considering that an array cell is live *from its very first write to its very last read*, even it is live only on several smaller "intervals".
For our specific usage, the optimization of the local memory used to store data from the DDR within a tile band, this means that a local array cell is considered live from the time it is loaded to its last use in the tile band, even if it is overwritten one or more times in the tile band. This case is unlikely to happen, since, most often, array indices are translations (e.g. a[i,j-1]), resulting in a *single* sliding interval for each array cell.

▷ **Software:** BEE**.** We have developed a tool called BEE based on these principles. It takes as input a program annotated with pragmas specifying the schedule. BEE analyzes the program, computes the conflict polyhedron DS for each array, then computes the mapping using [114] (tuned home version) or [72] (calling the CLAK library), and finally outputs the program with the contracted arrays. BEE can be tried online [a].

[a] http://compsys-tools.ens-lyon.fr/bee

4.5 Experimental results

We implemented our source-to-source optimizer using the polyhedral libraries Polylib and PIP. Our prototype inputs the C source code of the kernel to be optimized, annotated with pragmas

Kernel	ALUT	Reg.	T. reg.	IP	M. freq.	Speed-up
System alone	4406	3474	3606	8	205.85	
DMA original	4598	3612	3744	8	200.52	1
DMA manual	9853	10517	10649	8	162.55	6.01
DMA autom.	11052	12133	12265	48	167.87	5.99
VS original	5333	4607	4739	8	189.04	1
VS manual	10881	11361	11493	8	164	6.54
VS autom.	11632	13127	13259	48	159.8	6.51
MM original	6452	4557	4709	40	191.09	1
MM manual	15255	15630	15762	188	162.02	7.37
MM autom.	24669	32232	32364	336	146.25	7.32

Table 4.1 – Synthesis: original, manual, automatic

specifying the loop tiling. Then, it generates automatically a C source code implementing a double-bufferized version of the kernel. The 5 driver codes are then synthesized using C2H, which integrates them automatically in the system instantiated using Altera SOPC builder.

We apply our method to optimize three kernels: a copy $a \leftarrow b$ where a and b are 1-dimensional arrays (DMA), the sum of two vectors (VS) and the product of two matrices (MM).

We used ModelSim to evaluate our designs, which were synthesized on the FPGA Altera Stratix II EP2S180F1508C3, running at 100 MHz, and connected to an outside DDR memory, of specification JEDEC DDR-400 128 Mb ×8, CAS of 3.0, running at 200 MHz. The optimized versions can run 6x or more faster than the direct implementations (remember that the maximal speed-up is at most 8, if we start from a code where successive DDR accesses are in different rows). Note that these speed-ups are obtained neither because computations are parallelized (tiles are run in sequential), nor because the communications are pipelined (this is also the case in the original versions), but (i) because DDR requests are reorganized to get successive accesses on the same row as much as possible, (ii) because some communications overlap computations, and (iii) because some data reuse is exploited.

However, to achieve this, there is a (moderate) price to pay in terms of hardware resources, in addition to the local memories involved to store the data locally. This is illustrated in Table 4.1, which gives different parameters measuring the hardware usage: the number of look-up tables (column “ALUT”), of registers (“Reg.”), of all registers including those used by the synthesis tool (“T. reg.”), and of hard 9-bit multiplication IP cores (“IP”). Compared to the manually-optimized versions, the automatic ones use slightly more ALUT and registers, mostly because they use 2 separate FIFOs for synchronization between the drivers Load0 and Load1, and the driver Compute (we changed the design to make it more generic). They also use more multipliers to perform tile address calculations, which could be removed by strength reduction.

Speed-ups are given in the column “Speed-up”. Optimized versions have a slightly smaller *maximal* running frequency than the original ones (column “M. freq.” in MHz). But, if the designs already saturate the memory bandwidth at 100 MHz, running the systems at a higher frequency will not speed them up anyway. This maximal frequency reduction could come from more complex codes, the Avalon interconnect routing, and the use of double-port memories available in the FPGA, which induces additional synthesis constraints.

4.6 Conclusion

In this chapter, we have proposed an HLS algorithm to optimize the communications with the off-chip memory. We built our algorithm as a source-to-source post-pass in front of C2H, the C-to-VHDL compiler of Altera. We propose a template of architecture to optimize off-chip memory accesses. Then, we propose original techniques to analyze, optimize, and generate the final code.

In particular, we propose an communication scheduling algorithm which minimize the memory transfers, while reducing the local memory size. Finally, we presented our contributions to local storage allocation.
The template of architecture and the tiling strategy developed in this chapter were the basis to the HLS methodology presented in the next chapter. Many improvements are possible. In particular, the complexity of LOAD and STORE sets could be set could be trade for the communication volume and/or the local memory size, if access are scheduled in different tiles.

5 Data-aware Process Networks

Process networks [102, 160] are dataflow models of computation expressing naturally task-level parallelism for streaming applications. In this context, they are a relevant *intermediate representation* for parallelizing compilers, where a *front-end* extracts the parallelism and derive the process network, then a *back-end* maps the process network to the target architecture.
For HPC applications manipulating a huge volume of data with large reuse distances, process networks are no longer relevant, since the buffers size would literally explodes. Rather, automatic parallelizers traditionally rely on partitioning, scheduling and allocation techniques to distribute the computations while orchestrating communications.
In the context of *high-level synthesis of HPC applications for FPGA*, we leverage the *polyhedral process networks* developed in the Compaan project [118, 140, 153, 160] to define a new dataflow model of computation, the data-aware process networks (DPN), which *combines* the expressivity of process networks *with the partitioning and the communication scheduling presented in Chapter 4* to orchestrate the *spilling* with the off-chip memory.
We develop a complete *front-end* C → DPN by leveraging some of the ideas developed in the Compaan project. Then, we propose compiler algorithms for the *back-end* DPN → circuit.

Summary and outline This chapter summarizes our contributions to *high-level synthesis of HPC applications for FPGA in the polyhedral model*. Most of these results were obtained in collaboration with Alexandru Plesco, in the context of the XtremLogic start-up project, and are now used in the production compiler of the Xtremlogic start-up.
Section 5.1 introduces *regular process networks* (RPN), a general model of computation derived from polyhedral programs, which *subsumes polyhedral process networks*, and serves as a foundation to our model. Then:

- Section 5.2 presents *data-aware process networks (DPN)*, a parallel intermediate representation which leverages the execution model proposed in chapter 4 to unify the parallelism and the data transfers [20].
  Our contributions to the synthesis of a DPN to a circuit are presented in Sections 5.4, 5.5 and 5.6.
- Section 5.3 presents a loop scheduling algorithm to *optimize the throughput* when the datapath involves *pipelined arithmetic operators* [17, 18].
- Section 5.4 discusses our contribution to *synthesize the control* of a DPN [21].
- Section 5.5 presents our contribution to *restructure the channels* so FIFO channels are guaranteed after a loop tiling [1, 4].
- Section 5.6 outlines our contribution to *synchronize* interprocess communications [19].

5.1 Regular process networks

Since the early days of high-level synthesis, *partitioning techniques* were designed to distribute the computations across parallel units, while exploring the parallelism trade-offs [121, 141, 143, 150]. On the other hand, dataflow representations are natural candidates to represent the parallelism of an application. For polyhedral programs, the Compaan project propose to represent programs with *polyhedral process networks* (PPN) [118, 140, 153, 160], a dataflow model of computation derived from SARE, with the Kahn process networks semantics [102].
This section describes a simple dataflow representation for polyhedral programs inspired by PPN, which captures the partitioning of computations *and communications*: the *regular process networks* (Section 5.1.1). Although there is nothing fundamentally new behind RPN (a RPN is nothing more than a PPN with a generic partitioning strategy), the cross fertilization between the concept of partitioning and dataflow representation gives us a general HLS design methodology (Section 5.1.2), which drives the contributions presented in this chapter.
We only use RPN in the next section to define properly our DPN model and to make a fair comparison with PPN.

5.1.1 Regular process networks

Given a polyhedral program, we build a *regular process network* (RPN) with the following operations:

- We *partition the computation* (iteration domains) into *processes*: if $\mathscr{D}$ denotes the union of the iteration domains of the program, any partition $\mathscr{D} = \mathscr{D}_1 \uplus \ldots \uplus \mathscr{D}_n$ defines a set of processes $\{P_1, \ldots, P_n\}$, each P_i iterating over $\mathscr{D}_i$.
- We provide a *schedule* θ_i for each process P_i.
- Then, we *partition the direct dependences* into *channels*: if $\to$ denotes the direct dependence relation, any partition $\to = \to_1 \uplus \ldots \uplus \to_n$ defines a set of channels $\{1, \ldots, n\}$ such that the data transferred by $\to_c$ is conveyed through channel c. We say that $\to_c$ is *resolved* by channel c.
  With this definition, the channel implementation is completely abstracted. It could be a FIFO, a global/local memory, a link between communicating FPGA, and so on.

More formally:

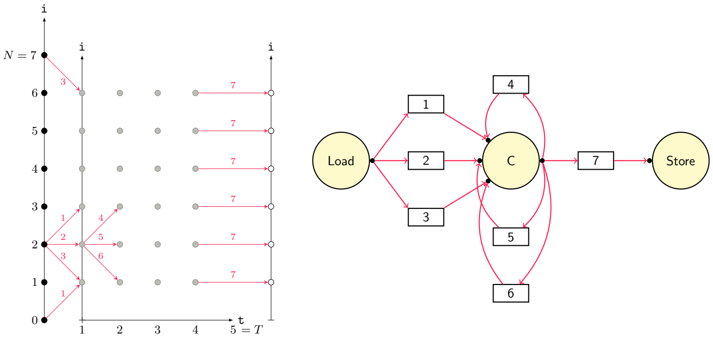
Definition 5.1 (Regular process network (RPN))**.** *Given a program, a* regular process network *is a couple* $(\mathscr{P}, \mathscr{C}, \theta, \omega)$ *such that:*

- *Each process* $P \in \mathscr{P}$ *is specified by an iteration domain* $\mathscr{D}_P$ *and a sequential schedule* θ_P *inducing an execution order* $\prec_P$ *over* $\mathscr{D}_P$*. Each iteration* $\vec{i} \in \mathscr{D}_P$ *realizes the execution instance* $\omega_P(\vec{i})$ *in the program.*
  The processes partition the execution instances in the program: $\{\omega_P(\mathscr{D}_P)$ *for each process* P *of* $\mathscr{P}\}$ *is a partition of the program computation.*
- *Each channel* $c \in \mathscr{C}$ *is specified by a producer process* $P_c \in \mathscr{P}$*, a consumer process* $C_c \in \mathscr{P}$ *and a dataflow relation* $\to_c$ *relating each production of a value by* P_c *to its consumption by* C_c*: if* $\vec{i} \to_c \vec{j}$*, then execution* $\vec{i}$ *of* P_c *produces a value read by execution* $\vec{j}$ *of* C_c*.* $\to_c$ *is a subset of the direct dependences from the iterations of* P_c *in the program* $(\omega_{P_c}(\mathscr{D}_{P_c}))$ *to the iterations of* C_c *in the program* $(\omega_{C_c}(\mathscr{D}_{C_c}))$ *and the set of* $\to_c$ *for each channel* c *between two given processes* P *and* C*,* $\{\to_c, (P_c, C_c) = (P, C)\}$*, is a partition of direct dependences from* P *to* C*.*

Example. Figure 5.1.(a) depicts a polyhedral kernel and (b) provides the iteration domains for each assignment (• for assignment *load*, • for assignment *compute* and ∘ for assignment *store*). On Figure 5.1.(c), each execution $\langle S, \vec{i} \rangle$ is mapped to process P_S: load iterations (•) are mapped to process Load, compute iterations (•) are mapped to process C, and store iterations (∘) are mapped to process Store. With this straightforward partitioning, the instances mapped to a process come from the same statement, hence the input ports are the reads of the statement. The input/out-

```
  for $i := 0$ to $N+1$
• load($a[0,i]$);
  for $t := 1$ to $T$
    for $i := 1$ to $N$
•     $a[t,i] := a[t-1,i-1] + a[t-1,i]+$
            $a[t-1,i+1]$;
  for $i := 1$ to $N$
∘ store($a[T,i]$);
```

(a) Jacobi 1D kernel



(b) Flow dependences

(c) Regular process network

Figure 5.1 – Motivating example: Jacobi-1D kernel. (a) depicts a polyhedral kernel, (b) gives the polyhedral representation of loop iterations ($\bullet$: load, $\bullet$: compute, $\circ$: store) and flow dependences (red arrows), then (c) gives a possible implementation as a regular process network: each assignment (load, compute, store) is mapped to a different process and flow dependences (1 to 7) are solved through channels.

put ports are depicted with black points. Dependences labeled by k on the dependence graph in (b) are solved by channel k. With this dependence partitioning, we have a different channel per couple producer/read reference. This way, the input values can be read in parallel. 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. □

Execution semantics The execution of a RPN is *locally sequential* (each process executes its operations sequentially) and *globally dataflow* (for each operation, the process executes once the input data are available). Hence, the execution order is constrained by the direct dependences between processes $\to$ for the global dataflow part; and the local sequential execution for each process P_i, $\prec_{\theta_i}$ for the local sequential part:

$$\prec_{RPN} \quad \supseteq \quad \to \cup \left(\bigcup_i \prec_{\theta_i} \right)$$

Partitioning strategy The *partitioning strategy* presented on the example (processes with instances of the same statement, statement instances possibly split across several processes, one channel per couple producer/read) defines the class of *polyhedral process networks* (PPN) [118, 140, 153, 160] developed in the context of the Compaan project. In this chapter, *we build on the PPN partitioning strategy* to define an RPN partitioning strategy (referred to as DPN, for data-aware process networks), which explicit the data transfers between the RPN and a remote memory.

Correctness We may wonder how to constrain the partitioning strategy and the process scheduling to obtain a correct RPN *i.e.* whose execution terminate and produces the same output value than the program given the same inputs. Actually, this is pretty open: for any partition, there exists a process schedule leading to a correct execution. This schedule is simply the original program schedule, or any valid sequential program schedule.
Under this scheduling constraint, the channels can always be sized to ensure a correct execution, and avoid deadlocks: at worse, the channels can be kept single-assignment. We point out that de-

pending on the partitioning of the channels, the synchronizations may be hard to enforce. This optimization criterion will be discussed later.

5.1.2 Compilation methodology

Regular process networks are a very general family of intermediate representations, that we believe to be appropriate and effective for automatic parallelization and specifically high-level synthesis of polyhedral kernels. RPN enforces the following compilation methodology, that drives the contributions discussed in this chapter.

Partitioning strategy At compiler design time, *choose a partitioning strategy*. This is clearly driven by the features of the target architecture. In the specific context of HLS of high-performance kernels which manipulate a huge volume of data with an important data reuse, we propose the DPN (data-aware process network) partitioning strategy [4, 20] described in Section 5.2.

Front-end: derive a RPN from a program.

1. Apply the computation partitioning and *find a schedule* θ_i for each process P_i. Many criteria can drive the derivation of θ_i: throughput, parallelism, channel size to quote a few. In Section 5.3, we propose a loop scheduling algorithm which maximize the throughput of a *single process* by reducing bubbles in the arithmetic datapath [17, 18]. This is an important building block for the future derivation of process schedules.
2. *Compile the process control* (multiplexers for input ports, demultiplexers for output ports,control loop nest or FSM). This step has been addressed in the context of PPN [153]. This basically follows the line of the generation of a SARE from the source function σ. We reuse and adapt these algorithms in the context of DPN [20].
3. *Compile the channels* (type of channel, size/allocation function, synchronizations).

   In Section 4.4, we have proposed an array liveness analysis to size and allocate channels [5]. In Section 5.5, we propose an algorithm to restructure the channels of a PPN to enforce a FIFO communication pattern after a loop tiling [1]. We show that our algorithm is complete on the DPN partitioning: all the FIFO broken by the the loop tiling can be restored by our channel partitioning [4]. In case a channel is not a FIFO, we propose an alternative synchronization apparatus [19] based on the iteration vectors of the producer and the consumer processes.

We end-up with a rich RPN representation, which is ready to by mapped to the target.

Back-end: map the RPN to the target architecture In addition to target-specific passes, the back-end should feature simplification passes of the RPN representation. With the PPN partitioning strategy, it is possible to factor the channels [164] and the processes which share a lot of common control. In Section 5.4, we propose a back-end algorithm to compact affine control [21].

5.2 Data-aware process networks

PPN were developed in the context of HLS of data-centric embedded applications [76]. These applications manipulate streams of data and/or reasonably small matrices, while keeping a small number of data alive at the same time [151]. This way, *the total size of the channels stays reasonable*. When it comes to compute-intensive applications for high-performance computing, things are totally different. Indeed, the computation volume and the data reuse are such that *the total channel size would literally explodes*. For instance, a matrix multiplication $A \times B$ *à la* PPN would require to store the whole matrices A and B in channels, which is clearly not possible with FPGA, as the local memory size is at most a few tens of megabytes.

Hence, *spilling strategies* must be found, just as for register allocation. As shown in Chapter 4, partitioning strategies must be designed to split the application into *reuse units* (here tile bands) where *into a reuse unit, the data flows through local memory* and *between two reuse units, the data flows through a remote memory*. This general principle inspires the DPN partitioning.

DPN: a RPN partitioning strategy to tune operational intensity A data-aware process network [4, 20] is a RPN partitioning strategy where several dependences are *explicitly solved through the remote memory*. For these dependences, the source (writing the remote data) flows the data to a channel connected to a *store* process, assumed to write the data to the remote memory. Then the dependence target is connect to a *load* process through a distinct channel, which reads that data from the remote memory. Additional synchronizations are required to enforce the dependence. In other words, with the DPN partitioning strategy, the *reuse units* are made explicit and the ratio computation/data transfers can be tuned at dataflow level, by playing on a few parameters as explained later. From a HLS perspective, the DPN models a circuit with the adequate processes to load/store data from the outside world, the computation processes and the channels which organizes the local memory. It naturally extends the partitioning strategy by tile bands described in Chapter 4.

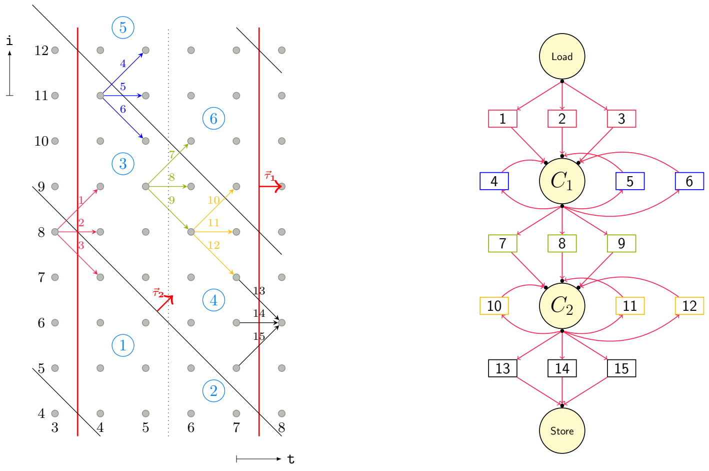
Making data transfers explicit Given a loop tiling, we consider the execution per tile band along the lines developed in Chapter 4: a tile band acts as a *reuse unit*. Dependences into the tile band (source and target in the same tile band) are resolved by a channel (in the local memory) and dependences from/to another tile band are resolved through a remote memory: *incoming dependences are loaded* from the remote memory, *outcoming dependences are stored* to the remote memory. Figure 5.2 gives a possible DPN partition (b) from a loop tiling (a) on the motivating example. The loop tiling is defined by hyperplanes (ϕ_1,ϕ_2) where $\phi_1(t,i) = t = \vec{\tau}_1 \cdot (t,i)^T$ with $\vec{\tau}_1 = (0,1)^T$ and $\phi_2(t,i) = t+i = \vec{\tau}_2 \cdot (t,i)^T$ with $\vec{\tau}_2 = (1,1)^T$. By definition, the iteration domain is sliced into tile bands by all the hyperplanes except the last one (here $\vec{\tau}_1$, hence we obtain vertical bands). (b) depicts the tile band with $4 \leq t \leq 7$. Since tile band acts as a *reuse unit*, 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).

Adding parallelism Inside a band, the computations may be split in parallel process thanks to surrounding hyperplanes with normals $(\vec{\tau}_1,\ldots,\vec{\tau}_{n-1})$. On Figure 5.2.(a), each band is split in $p = 2$ sub-bands by the hyperplane $\vec{\tau}_1 = (1,0)^T$, separated by a dotted line. Each sub-band is mapped to separate process on Figure 5.2.(b): C_1 for the left sub-band, C_2 for the right sub-band. With this partitioning strategy, we can *tune the arithmetic intensity* ($A.I.$) by playing on the band width b (here $A.I. = 2 \times bN/2N = b$) and *select the parallelism independently*. Each parallel process is identified by its coordinate $\vec{\ell} = (\ell_1,\ldots,\ell_{n-1})$ across surrounding hyperplanes.The parallel instance of C of coordinate $\vec{\ell}$ is written $C_{\vec{\ell}}$ (here we have parallel instances C_0 and C_1).

Dependence partitioning 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). Again, *i/o dependences* are solved through *remote memory* whereas *local dependences* and *synchronization dependences* are solved through *local memory*.

Load/store processes For each array a loaded/stored through $\rightarrow_{i/o}$, a load (resp. store) process $\textsc{Load}_a$ (resp. $\textsc{Store}_a$) 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 $\textsc{Store}_a$: $(P_{\vec{\ell}}, \phi_1, \ldots, \phi_n, \vec{i}) \rightarrow_{i/o} (\textsc{Store}_a, \phi_1, \ldots, \phi_n, u(\vec{i}))$, and by a dependence flowing from $\textsc{Load}_a$: $(\textsc{Load}_a, \phi_1, \ldots, \phi_n, v(\vec{j})) \rightarrow_{i/o} (C_{\vec{\ell}'}, \phi'_1, \ldots, \phi'_n, \vec{j})$ (on Figure (b), these processes are simply named Load/Store). We point out that $\rightarrow_{i/o}$ defines the sets $In(T)$ and $Out(T)$ of data read/defined into a tile T used in Chapter 4 to derive the processes $\textsc{Load}_a$ and $\textsc{Store}_b$ which minimize the data transfers.

Connection with the canonic PPN partitioning Finally, the dependences are 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 $4, 7, 10$): this is the property inherited from PPN. 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 $(\mathscr{L}, \mathscr{P}', \mathscr{S}, \mathscr{C}')$ where $\mathscr{L}$ is the set of $\textsc{Load}$ process, $\mathscr{P}'$ is the set of parallel processes, $\mathscr{S}$ is the set of store processes $\textsc{Store}$ and $\mathscr{C}'$ is the set of channels.



(a) DPN partitioning

(b) DPN

Figure 5.2 – **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.

▷ **Software:** Dcc **and** PoCo**.** All our contributions around DPN and communication coalescing (described in Chapter 4) have been implemented in our research compiler, Dcc (Data-aware process network C compiler). Dcc analyzes a polyhedral program written in C and annotated with pragmas specifying the tiling and the schedule θ, then it produces a DPN. Dcc comes with a polyhedral compilation library, PoCo (Polyhedral Compilation Library) which implements most of the basic polyhedral analysis (mainly dataflow analysis [81], liveness analysis [5], array contraction [5, 114], and polyhedral code generation [54]). We have entirely implemented these tools, which serves now as a front-end for the XtremLogic HLS tool. So far, the XtremLogic back-end is still under construction. This prevent to have global results on DPN. Instead, we present local results for our different contributions.

The next sections presents our contribution to the synthesis process, both on the front-end and the back-end part.

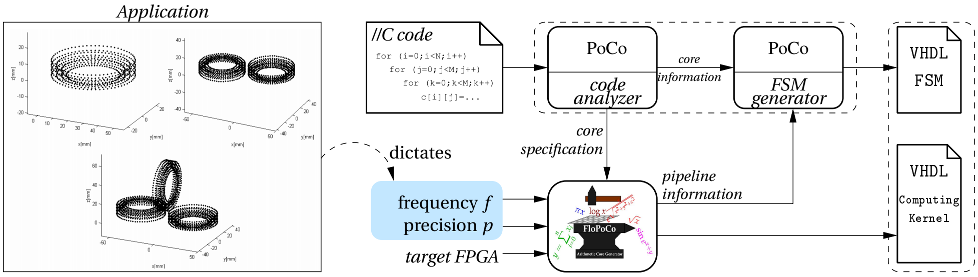


Figure 5.3 – Automation flow

5.3 Loop scheduling for pipelined arithmetic operators

This section presents a scheduling technique to keep busy the pipeline of the arithmetic datapath, thereby improving the overall latency of the design [17, 18]. This contribution focuses on perfect loop nests with uniform dependences. It is not intended to schedule globally a DPN, but we believe it provides important insights to address this problem.

Contribution The arithmetic operators $(+, -, \times, /, \sqrt{.}, \ldots)$ used in the datapath of hardware accelerators are usually pipelined to keep the frequency of the circuit at a reasonable level. As a consequence, an operation scheduled at date t will output its result at date $t+d$ where d is the number of stages of the pipeline. If the result is consumed at date t', we must have $t' > t+d$ (*pipeline constraint*). Otherwise, the pipeline of the consumer will be frozen until the data is available, which would degrade the throughput of the circuit. We propose a method to derive automatically an efficient, sequential, hardware using pipelined arithmetic operators in the data-path. In particular, we propose a scheduling algorithm [17, 18] which reorganizes the computations to reduce the total execution time while satisfying the pipeline constraints. Figure 5.3 depicts our compilation flow. In this study, we consider the floating-point arithmetic custom operators generated by the tool FloPoCo [73], from a specification including the precision and the pipeline depth. We point out that our method could apply to any pipelined datapath. The operations are carefully scheduled to keep the FloPoCo operators busy, hence making optimal use of their pipelines. The input kernel is specified by a naive sequential C program, as depicted in Figure 5.4.(a) for matrix multiplication. The user must also specify the pipeline depth for each FloPoCo operator. These are the only inputs required.

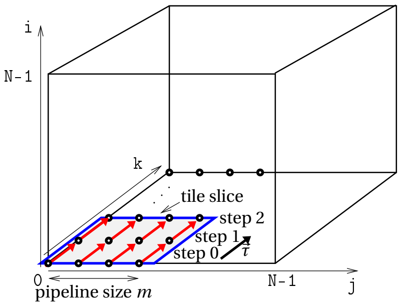
Program model Our algorithm operates on *perfect loop nests with uniform dependences* iterating over a single statement S. A typical example is the matrix multiply kernel given in Figure 5.4.(a). Hence, we can represent the dependences with *dependence vectors* $\Delta D = \{\vec{j} - \vec{i} \mid \langle S, \vec{i}\rangle \rightarrow_{pc} \langle S, \vec{j}\rangle\}$ where $\rightarrow_{pc}$ denotes the direct dependences (see Section 2.1.3, paragraph *Dependences*). The dependences are uniform when ΔD is finite (non parametric). This applies to matrix multiply (as $\Delta D = \{(0,0,1)\}$). This model captures stencil operations and basic linear algebra operators. We point out that our scheduling algorithm does not require uniform dependences. They are required by our hardware generation scheme to produce shift registers. This constraint can be relaxed if we use FIFO instead of shift registers, we will discussed this point later. However, how to extend this approach to a general loop nest (or equivalently on to the processes of a DPN) is still an open problem.

5.3.1 Motivating examples

In this section, we illustrate the feasibility of our approach on two examples. The first example is the matrix multiplication, that has one uniform data dependency that propagates along one

```
typedef float fl;
void mmm(fl* a, fl* b, fl* c, int N) {
  int i, j, k;
  for (i = 0; j < N; j++)
    for (j = 0; i < N; i++){
      for (k = 0; k < N; k++)
        c[i][j] = c[i][j] + a[i][k]*b[k][j];//S
    }
}
```

(a)



(b)

Figure 5.4 – Matrix-matrix multiplication: (a) C code, (b) iteration domain with tiling

axis. The second example is the Jacobi 1D algorithm. It is more complicated because it has three uniform data dependencies with different distances.

Matrix multiplication

The original code and its iteration domain is depicted in Figure 5.4. Each point (i, j, k) of the iteration domain represents an execution of the assignment S with the loop counters i, j and k. There is a unique data dependency carried by the loop k, which can be expressed as a vector $\vec{d} = (\delta i, \delta j, \delta k) = (0, 0, 1)$ (Figure 5.4.(b)). With the original sequential schedule, the computation applies sequentially a multiply and accumulate operation $(x, y, z) \mapsto x + (y * z)$ along the k axis, that we want to implement with a specialized FloPoCo operator (Figure 5.6.(a)). It consists of a pipelined multiplier with ℓ pipeline stages that multiplies the elements of matrices a and b, followed by a pipeline adder with m pipelined stages. When $k = 0$, the accumulation is initialized thanks to a delayed control signal and the corresponding steering logic. For $k > 0$, the multiplication result is accumulated with the current sum, available *via* the feedback loop (when the delayed control signal S is 1). This result will be available m cycles later (m is the adder pipeline depth), for the next accumulation.

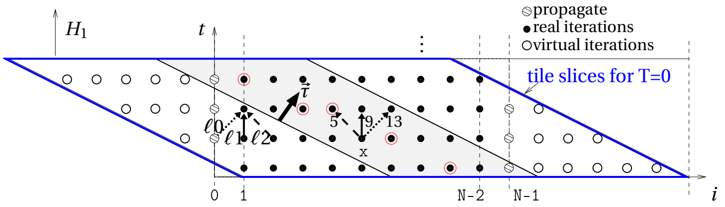
The original execution order (with schedule $\theta(i, j, k) = (i, j, k)$), would not exploit at all the pipeline, causing a stall of $m - 1$ cycles for each iteration of the loop k. Indeed, the iteration $(0, 0, 0)$ would be executed, then wait $m - 1$ cycles for the result to be available, then the iteration $(0, 0, 1)$ would be executed, and so on. To address this issue, the loop schedule must enforce at least m iterations between the definition of a value at iteration $\vec{i} = (i, j, k)$ and its use at iteration $\vec{i} + \vec{d} = (i, j, k + 1)$. In other words, the *dependence distance* $\Delta(\vec{d})$ must be greater than m.

We enforce this property with a loop tiling, where the last iteration traverses an hyperplane (called the *parallel hyperplane* in the remainder) without carrying any dependence. Then, we can customize the dependence distance simply by playing on the tile size. On to the example, we can choose the parallel hyperplane $H_{\vec{\tau}}$ with $\vec{\tau} = (0, 0, 1)$. $H_{\vec{\tau}}$ does not *contain* the dependence $\vec{d}$. Hence, each iteration on this hyperplane can be executed independently, so it is possible to insert in the arithmetic operator pipeline one computation every cycle. Then, we complete the tiling with hyperplanes (H_1, H_2) using standard tiling techniques [53] (here, the normal vectors are $(1, 0, 0)$ and $(0, 1, 0)$), $\mathscr{H} = (H_1, H_2, H_{\vec{\tau}})$. Finally, we consider the schedule $\theta(I_1, I_2, I_3, i, j, k) = (I_1, I_2, I_3, i, k, j)$ where the last dimension "slides" along $H_{\vec{\tau}}$. Basically, on this example, the tile width along H_2 is exactly $\Delta(\vec{d})$. Hence, we set it to the pipeline depth m.

This way, the result is scheduled to be used exactly at the cycle it gets out of the operator pipeline, without any temporary buffering. In a way, the pipeline registers of the arithmetic operator are used as a temporary buffer. We point out that this is possible because dependences are uniform. Otherwise, we would not be able to guarantee, in general, a constant dependence distance and we would have to use a FIFO, whose size can by bounded with array contraction techniques, for instance, those described in section 4.4.

```
typedef float fl;
void jacobi1d(fl a[T][N]){
  fl b[T][N];
  int i,t;
  for (t = 0; t < T; t++)
    for (i = 1; i < N-1; i++)
      a[t][i] = (a[t-1][i-1]+a[t-1][
          i]+
           a[t-1][i+1])/3;
}
```

(a)



(b)

Figure 5.5 – Jacobi 1D: (a) source code, (b) iteration domain with tiling

Jacobi 1D

The kernel is given in Figure 5.5.(a), this is a standard stencil computation with two nested loops. We want to implement the datapath with the pipelined operator depicted on Figure 5.6.(b). On this example, we have several dependence vectors, $\Delta D = \{\vec{d_1} = (\delta t, \delta i) = (1,-1), \vec{d_2} = (1,0), \vec{d_3} = (1,1)\}$ (Figure 5.5.(b)).

Again, we build a tiling $\mathscr{H} = (H_1, H_{\vec{\tau}})$ with a parallel hyperplane $H_{\vec{\tau}}$. We choose $H_{\vec{\tau}}$ with normal vector $\vec{\tau} = (\delta t, \delta i) = (2,1)$. We verify that $H_{\vec{\tau}}$ does not contain any dependence: $\vec{\tau} \cdot \vec{d} > 0$, for any dependence $\vec{d} \in \Delta D$. Equivalently, we say that $H_{\vec{\tau}}$ *satisfies* all the dependences. Then, we complete the tiling with the hyperplane H_1 with normal vector $(1,0)$. Finally, we choose a tiled schedule whose last dimension slides along $H_{\vec{\tau}}$: $\theta(I_1, I_2, t, i) = (I_1, I_2, 2t+i, t)$. Figure 5.5.(b) shows the consecutive tile slices with $I_1 = 0$ with a tile height of 4 iterations in the direction of H_1.

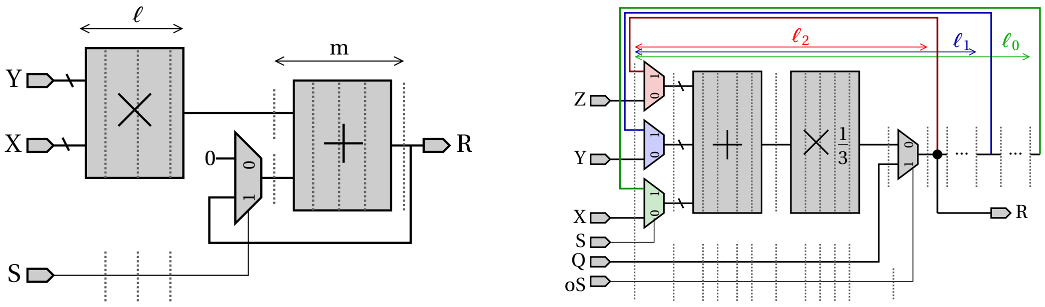
With this schedule, the dependence distances are $\Delta(\vec{d_1}) = 5$ iterations, $\Delta(\vec{d_2}) = 9$ iterations and $\Delta(\vec{d_3}) = 13$ iterations, which means that the data flowing through the dependence $\vec{d_1}$ (resp. $\vec{d_2}$, $\vec{d_3}$) must be available at least 5 (resp. 9, 13) iterations later. Notice that the dependence distances are the same for any point of the iteration domain, as the dependencies are uniform. In hardware, this translates to adding delay shift registers at the operator's output and connecting this output to the operator's inputs *via* feedback lines, according to the data dependency distance levels ℓ_0, ℓ_1 and ℓ_2 (see Figure 5.5.(b)). Once again, the intermediate values are kept in the pipeline, no additional storage is needed in a slice.

As the tiling hyperplanes are not parallel to the original axis, some tiles in the borders are not full parallelograms (see left and right triangles in Figure 5.5.(b)). Inside these tiles, the dependence vectors are not longer constant. To overcome this issue, we extend the iteration domain with virtual iteration points where the pipelined operator will compute on dummy data. This data is discarded at the border between the real and extended iteration domains (propagate iterations, when $i = 0$ and $i = N-1$). For the border cases, the correctly delayed data is fed via line `Q` (`oS`=1).

5.3.2 Method

The key idea is to tile the program in such a way that each dependence distance can be customized by playing on the tile size. Then, it is always possible to set the minimum dependence distance to the pipelined depth of the FloPoCo operator, and to handle the remaining dependencies with additional (pipeline) registers in the way described for the Jacobi 1D example.

This amounts to find a parallel hyperplane $H_{\vec{\tau}}$, and to complete the tiling with independent hyperplanes: $H_1, \ldots, H_{n-1}$, assuming the depth of the loop kernel is n. Now, it is easy to see that the hyperplane $H_{\vec{\tau}}$ should be the $(n$-1)-*th* hyperplane (implemented by L_{it}), any hyperplane H_i being the last one (implemented by L_{par}). Roughly speaking, L_{it} pushes $H_{\vec{\tau}}$, and L_{par} traverses the current 1D section of $H_{\vec{\tau}}$. It remains in *step c* to compute the tile size to fit the fixed FloPoCo operator pipeline depth. If several dependencies exist, the minimum dependence distance $\Delta(\vec{d})$ must be set to the pipeline depth of the operator.



(a) (b)

Figure 5.6 – Pipelined arithmetic operators produced by FloPoCo

We propose an ILP formulation to find an parallel hyperplane $H_{\vec{\tau}}$ while reducing the dependence distances $\Delta(\vec{d})$ (hence the number of shift registers). We compute the dependence distances $\Delta(\vec{d})$ as a symbolic expression of the tile height ℓ along the before-last hyperplane (H_{n-1} for a tiling $(H_1, \ldots, H_{n-1}, H_{\vec{\tau}})$. Finally, ℓ can be derived by solving the equation $\Delta(\vec{d}) = m$ where m is the pipeline depth and $\Delta(\vec{d})$ is the smallest dependence distance [17, 18].

5.3.3 Experimental results

Table 5.1 presents synthesis results for both our running examples, using a large range of precisions, and two different FPGA. The results presented confirm that precision selection plays an important role in determining the maximum number of operators to be packed on one FPGA. As it can be remarked from the table, our automation approach is both flexible (several precisions) and portable (Virtex5 and StratixIII), while preserving good frequency characteristics.

Table 5.1 – Synthesis results for the full (including FSM) MMM and Jacobi1D codes. Results obtained using using Xilinx ISE 11.5 for Virtex5, and QuartusII 9.0 for StratixIII

Application	**FPGA**	**Precision**	**Latency**	**Frequency**	**Resources**		
		(w_E, w_F)	**(cycles)**	**(MHz)**	**REG**	**(A)LUT**	**DSPs**
Matrix-Matrix Multiply N=128	Virtex5(-3)	(5,10)	11	277	320	526	1
		(8,23)	15	281	592	864	2
		(10,40)	14	175	978	2098	4
		(11,52)	15	150	1315	2122	8
		(15,64)	15	189	1634	4036	8
	StratixIII	(5,10)	12	276	399	549	2
		(9,36)	12	218	978	2098	4
Jacobi1D stencil N=1024 T=1024	Virtex5(-3)	(5,10)	98	255	770	1013	-
		(8,23)	98	250	1559	1833	-
		(15,64)	98	147	3669	4558	-
	StratixIII	(5,10)	98	284	1141	1058	-
		(9,36)	98	261	2883	2266	-
		(15,64)	98	199	4921	3978	-

The generated kernel performance for one computing kernel is: 0.4 GFLOPs for matrix-matrix multiplication, and 0.56 GFLOPs for Jacobi, for a 200 MHz clock frequency. Thanks to program restructuring and optimized scheduling in the generated FSM, the pipelined kernels are used with very high efficiency. Here, the efficiency can be defined as the percentage of useful (non-virtual) inputs fed to the pipelined operator. This can be expressed as the ratio $\#(\mathscr{I} \setminus \mathscr{V})/\#\mathscr{I}$, where $\mathscr{I}$ is the iteration domain and $\mathscr{V} \subseteq \mathscr{I}$ is the set of virtual iterations. The efficiency represents more than 99% for matrix-multiply, and more than 94% for Jacobi 1D. Taking into account the kernel size and operating frequencies, tens, even hundreds of pipelined operators can be packed per FPGA, resulting in significant potential speedups.

Table 5.2 – Synthesis results for the parallelized MMM and Jacobi1D. Results obtained using using Quartus II 10.1 for StratixIII with $w_E = 8$, $w_F = 23$

Application	**Par. factor**	**Frequency**	**Resources**			
		(MHz)	**REG**	**(A)LUT**	**M9K**	**DSPs**
Matrix-Matrix Multiply `N=128`	1	308	701	614	3	4
	2	282	1317	999	5	8
	4	303	2473	1789	12	16
	8	302	4842	3291	20	32
	16	281	9582	6291	32	64
Jacobi1D stencil `N=1024` `T=1024`	1	311	1217	1199	9	-
	2	295	2394	2095	21	-
	4	283	4600	3853	38	-
	8	274	9018	7314	69	-
	16	251	17806	14218	132	-

Adding parallelism Table 5.2 presents synthesis results of the parallelization for both our running examples on the StratixIII FPGA using the single precision format. Matrix-multiply parallelization follow the DPN scheme (invented after this contribution), the tiling bands are simply along the i axis. As for the Jacobi 1D, we used a different parallelization scheme where the tile band is split in 2 processes along the i axis. The left process iterates with schedule $\theta_1(t,i) = (2t+i,t)$, while the right process iterates with schedule $\theta_2(t,i) = (2t-i,t)$ (see Figure 5.7). Since both process are driven by the same FSM, no synchronizations are required (more details in [18], p 10). As expected, due to massive parallelism and no inter parallel process communication, for matrix multiplication example the scaling in terms of resources is proportional to the parallelization factor. The maximum operating frequency remains fairly constant. Jacobi 1D scales very well too. A small increase in utilized resources is due to the increase in the multiplexer size in order to fit signals from neighbor computational cores. The frequency remains fairly constant. This proves that our method is well suited for FPGA implementation.

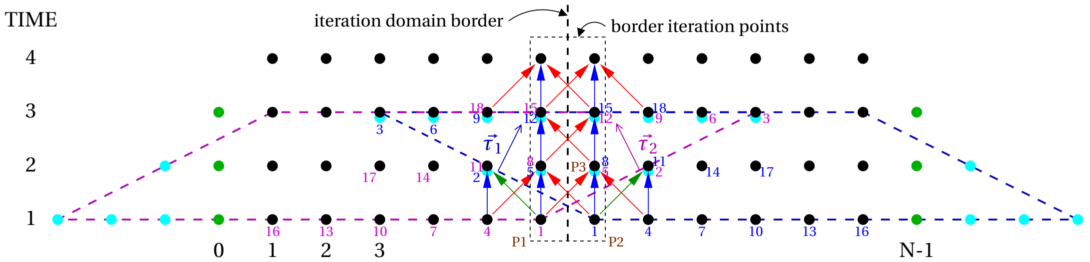


Figure 5.7 – An alternative to executing the Jacobi Kernel using 2 processing elements.

5.4 Synthesizing the control

Hardware accelerators derived with polyhedral compilation techniques make an extensive use of affine expressions (affine functions and convex polyhedra) in control and steering logic. Since the control is pipelined, these affine objects must be evaluated at the same time for different values, which forbids aggressive reuse of operators. This section presents a method to factorize a collection of affine expressions without preventing pipelining. Our key contributions are (i) to use *semantic factorizations* exploiting *arithmetic properties of addition and multiplication* and (ii) to rely on a cost function whose minimization ensures a correct usage of FPGA resources. Our algorithm is totally parametrized by the cost function, which can be customized to fit a target FPGA. Experimental results on a large pool of linear algebra kernels show a significant improvement compared to traditional low-level RTL optimizations. In particular, we show how our method reduces resource consumption by revealing hidden strength reductions.

5.4.1 Affine control

Multiplexers Consider process C_1 on the DPN depicted on Figure 5.2. Each input port is associated a multiplexer which selects the source channel depending on the iteration (I_1, I_2, t, i) being executed. Recall that I_1 and I_2 are the tile counters for the tiling with hyperplanes $\phi_1(t,i) = t$ and $\phi_2(t,i) = t+i$. This multiplexer is nothing but the *source function* σ (see section 2.1.3, paragraph *"Dependences"*) and might be computed with with parametric integer programming techniques [81]. The result is an *integer piecewise quasi-affine mapping*, a mapping from $\mathscr{D} \subseteq \mathbb{Z}^n$ to $\mathbb{Z}^p$ defined, given a partition $\mathscr{D} = \mathscr{P}_1 \uplus \ldots \uplus \mathscr{P}_q$, by a quasi-affine function u_i (affine with possible integer divisions by a constant) on each part $\mathscr{P}_i$. When no divisions are involved, the mapping is said to be *integer piecewise affine*.

We point out that integer piecewise affine mappings are not necessarily continuous. Some results on piecewise affine mappings, for instance lattice-based representation [146], may no longer apply. For example, on port 1 of process C_1 (corresponding to read `a[t-1,i-1]` on Figure 5.1.(a)) we have the multiplexer:

$$\sigma(\langle C_1, I_1, I_2, t, i\rangle, 1) = \begin{cases} t = 4I_1: & \langle \text{buffer1}, i-1\rangle \\ t > 4I_1 \wedge i = 1: & \langle \text{buffer1}, i-1\rangle \\ t > 4I_1 \wedge i > 1: & \langle \text{buffer4}, t-1, i-1\rangle \end{cases}$$

Multiplexers are always *integer piece-wise quasi-affine* mappings modulo the encoding of buffer identifiers with integers and the padding of iteration vectors so they have the same dimension. The complexity of the multiplexer (number of clauses, number of affine constraints per clause) may increase exponentially with the dimension of the iteration domain. Hence, efficient compaction techniques are required.

Finite-state machine Once the schedule is found for the process, it remains to generate the control which executes the process in the order prescribed by the schedule. Many approaches were developed [42, 54]. Usually, we synthesize a control automaton per process P, which issues a new iteration vector $\vec{i}$ of P at each clock cycle [54]. Given the schedule θ_P of process P, two integer piecewise quasi-affine functions are required. A function First_P, which issues the first iteration of P w.r.t θ_P (initial state), and a function Next_P which maps each iteration of P to the next iteration of P to be executed w.r.t. θ_P (transition function). On the running example, with $T = N = 16$, tiles of size 4 × 4, and a schedule $\theta_{C_1}(I_1, I_2, t, i) = (I_1, I_2, t, i)$, we have:

$$\text{First}_P = (0,0,0,1)$$

$$\text{Next}_P(I_1, I_2, t, i) = \begin{cases} \text{(depth } I_1\text{)} \\ 2 - I_1 \geq 0: \\ \quad (1+I_1, 1+I_1, 4+4I_1, 1) \\ \\ \text{(depth } I_2\text{)} \\ 2 + I_1 - I_2 \geq 0: \\ \quad (I_1, 1+I_2, 4I_1, 4-4I_1+4I_2) \\ \\ \text{(depth } t\text{)} \\ 2 + t - 4I_2 < 0 \wedge -t + 4I_1 \geq 0: \\ \quad (t - (3t)/4, I_2, 1+t, -1-t+4I_2) \\ 2 + t - 4I_2 \geq 0 \wedge -t + 4I_1 \geq 0: \\ \quad (t - (3t)/4, I_2, 1+t, 1) \\ \\ \text{(depth } i\text{)} \\ 13 - i \geq 0 \wedge 2 - t - i + 4I_2 \geq 0: \\ \quad (I_1, I_2, t, 1+i) \end{cases}$$

We point out that Next_P is an integer piecewise quasi-affine *per depth of the schedule*. When several domains overlap, the deeper clause is chosen. Intuitively, each depth can be seen as a loop from a perfect loop nest: for I_1, for I_2, for t, for i. It is another reason why techniques to simplify generic piecewise affine functions do not apply, even though no integer divisions are involved.
All in all, the multiplexing and the control involved on this simplified jacobi-1d example have a set of 123 affine constraints and 69 affine expressions. Clearly, they should be compacted before being mapped to an FPGA. This section presents our contribution to compact several integer piecewise affine functions, provided as a pool of affine constraints and expressions, as a DAG using efficiently FPGA resources.

5.4.2 Cost model

We present here the cost model that drive our experiments. We point out that the cost function $|.|$ is a parameter of our algorithm. It could perfectly be refined/redefined to fit a different target.

Cost of a DAG An FPGA consists of reconfigurable building blocks with lookup tables, 1 bit adders and 1 bit registers (ALM with Altera, CLB with Xilinx). In addition, RAM blocks and DSP blocks are usually provided. Our DAGs use only integer operators (integer addition, integer multiplication by an integer constant) which require an amount of building blocks proportional to the bitwidth of the result. Hence, the resource usage of a DAG $\mathscr{D} = (N, E)$ can be modeled as a simple weighted sum:

$$|\mathscr{D}| = \sum_{n \in N} \mathbf{w}(n) \cdot \mathbf{bw}(n)$$

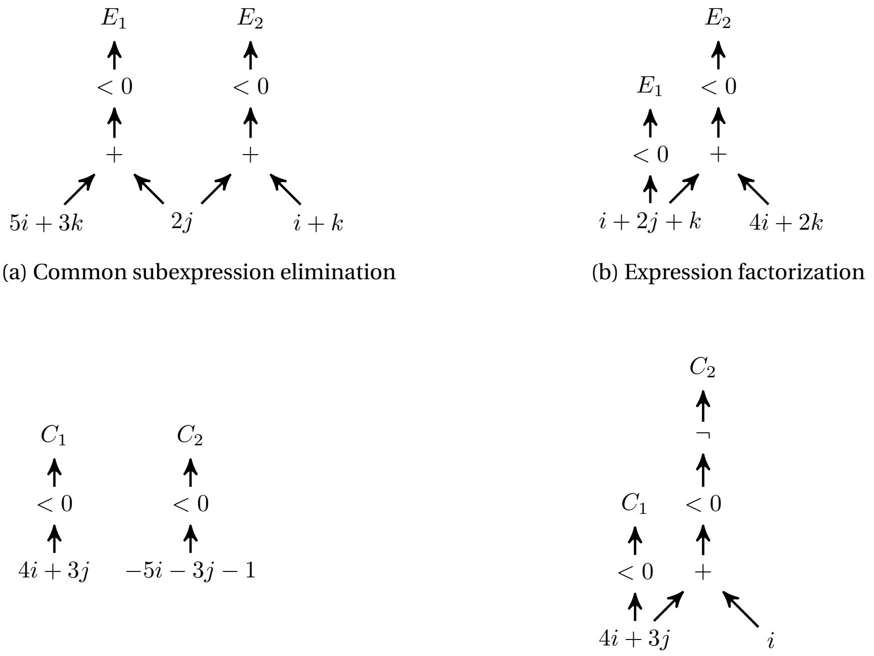
Where $\mathbf{bw}(n)$ denotes the bitwidth of the result computed by the operator n of the DAG and $\mathbf{w}(n)$ denotes, roughly, the number of building blocks required by n to compute 1 bit of result. $\mathbf{bw}$ is simply computed for each node of the DAG by a bottom-up application of the rules $\mathbf{bw}(x+y) = 1+\max(\mathbf{bw}(x), \mathbf{bw}(y))$ and $\mathbf{bw}(x*y) = \mathbf{bw}(x)+\mathbf{bw}(y)$ starting from the bitwidth of the input variables. Furthermore, $\mathbf{w}$ can be customized at will to fit a target FPGA. In the following, we will assume $\mathbf{w}(+) = 1$ and $\mathbf{w}(*) = 100$. With that choice, our algorithm will tend to decompose affine expressions with multiplications by a power of 2. Note that the cost model is not intended to reflect the actual resources requirement. Rather, it should be viewed as an objective function whose minimization leads to desired properties. We point out that special cases (multiplication by a power of 2, multiplication by a negative constant) are also taken into account by our model [21].

Affine forms Our algorithm builds the DAG by adding affine forms incrementally, and needs an upper bound on the cost of the sub-DAG computing an affine form. Consider an affine form $u = \sum_{i=1}^{n} a_i x_i + b$ where the x_i are integer variables and the coefficients a_i and b are integer constants. In the worst case, the term $a_i x_i$ (or b) with the largest bitwidth is evaluated first, each addition increasing the size of the result of 1 bit. Hence, the worst possible bitwidth for the result of u is $\mathbf{bw}_{\text{worst}}(u) = n-1+\max\{\mathbf{w}(b)\} \cup \{\mathbf{w}(a_i)+\mathbf{w}(x_i),\ i \in [\![1,n]\!]\}$. Therefore, an upper bound for $|u|$ is:

$$\lceil u \rceil = n \cdot \mathbf{bw}_{\text{worst}}(u) \cdot \mathbf{w}(+) + \sum_{i=1}^{n} |a_i * x_i|$$

5.4.3 Semantic factorizations

We classically reduce the resources to evaluate a pool of expressions by factorizations, the most common being common subexpression factorization. Our contribution is to investigate *semantic factorizations*, which leverage associativity and commutativity of operators + and ×. We present briefly the two kinds of semantic factorizations considered: *expression factorization* and *constraint factorization*. We illustrate these notions on the affine expressions $E_1 = i+2j+k$ and $E_2 = 5i+$



(a) Common subexpression elimination

(b) Expression factorization

(c) Common subexpression elimination

(d) Constraint factorization

Figure 5.8 – Semantic factorizations: expression factorization (b) and constraint factorization (d).

$2j + 3k$ where i, j and k are input variables. Figure 5.8 depicts the different kind of factorizations considered.

Expression factorization Common subexpression elimination would produce the DAG sketched in Figure 5.8.(a). The resources used are 4 adders, 2 multipliers by a constant and 1 shifter. Now, observe that $E_2 = E_1 + 4i + 2k$. This leads to the DAG in Figure 5.8.(b). This semantic factorization is called *expression factorization* in the remainder. With expression factorization, the resources required are now 4 adders and 3 shifters, which is better than the first solution. We point out that expression factorization is not always beneficial. If one tries to derive E_1 from E_2, with $E_1 = E_2 + (-4i - 2k)$, the resource usage would be worse than the direct solution (4 adders and 5 multipliers by a constant). A best combination of factorizations must be found among all the possible combinations.

Constraint factorization A similar factorization scheme can be applied to affine constraints. Consider the normalized affine constraints $C_1 : 4i + 3j < 0$ and $C_2 : -4i - 3j + 1 < 0$. Writing $C_2 : 4i + 3j \geq 0$, it is easy to detect that $C_2 = \neg C_1$. Now, consider the affine constraint $C_2 : -5i - 3j - 1 < 0$. There is no direct connexion with C_1. But if we write $C_2 : 5i + 3j \geq 0$, the affine expression of C_2 $(5i + 3j)$ can be obtained from the affine expression of C_1 $(4i + 3j)$, giving the improved DAG depicted in Figure 5.8.(d). With that *constraint factorization*, the resources used are reduced to 2 adders, 1 multiplier by a constant and 1 shifter. We point out that constraint factorization (C_2 from C_1) is a terminal transformation. Indeed, the expression of C_2 (e_2 s.t $C_2 : e_2 < 0$) is never computed. Hence, subsequent factorizations involving the expression e_2 are not possible. For this reason, expression factorizations will be preferred over constraint factorizations.
We propose a unified way to represent the possible sequence of factorizations of affine expressions and constraints and to select combination of factorizations minimizing the resource consumption.

5.4.4 Method

Realization graph Given a set of affine constraints $\mathscr{C}$ and affine expressions $\mathscr{E}$, we summarize all the possible semantic factorizations by means of a *realization graph* $\mathscr{G}_r$. Basically, the nodes of $\mathscr{G}_r$ are affine expressions and constraints, and an edge $u \xrightarrow{\Delta} v$ means that v can be realized from u with a cost of Δ. Also, $\mathscr{G}_r$ has a root node called *initial_node* which serves as a starting point to build affine expressions and constraints. Intuitively, a rooted path in $\mathscr{G}_r$ would give a realization of the reached nodes. Figure 5.9 depicts the realization graph built from the constraints:

$$\mathscr{C} = \{i+2j+k<0, \quad 5i+2j+3k<0, \quad 4i+3j<0, \quad -5i-3j-1<0\}$$

Depending on the factorization (expression or constraint) a specific edge (plain or dashed) is issued.Also, we point out that $\mathscr{G}_r$ contains constraints and expressions to synthesize (yellow nodes) and *intermediate nodes* (white nodes). We explain later how intermediate nodes are generated.

Expression factorization Given a DAG node computing an expression u, an expression v can be computed by applying the expression factorization rule $v = u + (v - u)$. In that case, we would add to the DAG the following components:
- A sub-DAG computing $v - u$ (to be optimized as well)
- An adder taking the output nodes of u and $v - u$.

The additional resource cost is the cost of the operator + plus the cost of the affine form $v - u$:

$$\Delta = (1 + \max\{\mathbf{bw}_{\text{worst}}(u), \mathbf{bw}_{\text{worst}}(v-u)\}) \cdot \mathbf{w}(+) + \lceil v - u \rceil$$

When that additional cost Δ is less than the cost of computing directly v, we register this possible design choice as an expression factorization edge $u \xrightarrow{\Delta} v$ to the realization graph $\mathscr{G}_r$ (plain edges on Figure 5.9.(a)). When the source (resp. target) node is a constraint $u < 0$ (resp. $v < 0$), the edge has the same meaning.

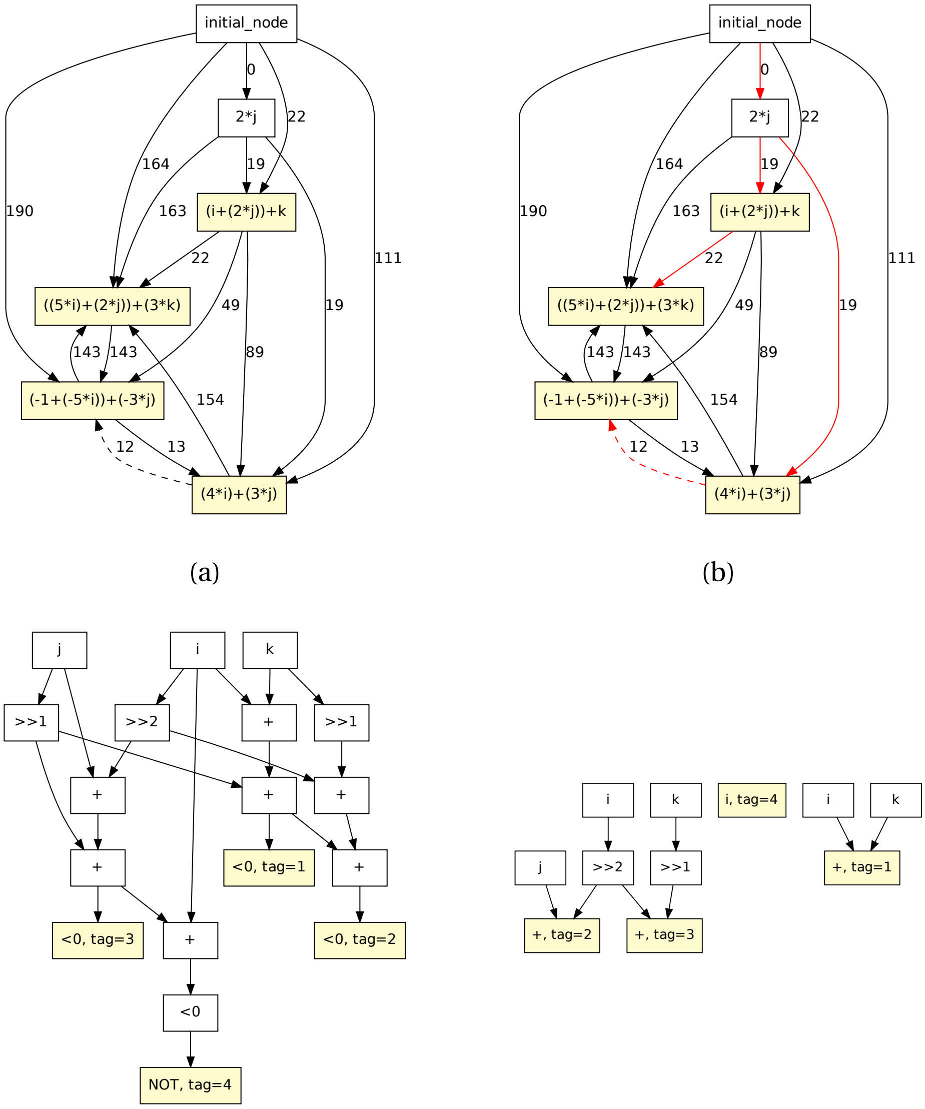
Constraint factorization Given a DAG node computing an affine constraint normalized as $u < 0$, the constraint $v < 0$ can be derived from $u < 0$ with a simple logic negation when $v < 0 \equiv \neg(u < 0)$, which means: $v < 0 \equiv -u - 1 < 0$ or more simply: $u + v = -1$. This gives a first simple test to detect negations. Otherwise, remark that $(u + (-1 - u - v)) + v = -1$. This means that $v < 0 \equiv \neg(u + (-1 - u - v) < 0)$. Hence $v < 0$ can be computed from u by adding the following components to the DAG:
- A sub-DAG computing $-1 - u - v$ (to be optimized)
- An adder taking the output nodes of u and $-1 - u - v$.
- The result of the adder is checked by connecting the most significant bit (to have < 0) to a negation.

The additional resource cost would then be the cost of the operator +, plus the cost of the affine form $-1 - u - v$:

$$\Delta = (1 + \max\{\mathbf{bw}_{\text{worst}}(u), \mathbf{bw}_{\text{worst}}(-1-u-v)\}) \cdot \mathbf{w}(+) + \lceil -1 - u - v \rceil$$

When that additional cost Δ is less than the cost of computing directly v, we register this possible design choice as a constraint factorization edge $u < 0 \xrightarrow{\Delta}_{\neg} v < 0$ in the realization graph $\mathscr{G}_r$ (dashed edges on Figure 5.9.(a)).

Building the realization graph We use the following algorithm to build the realization graph from a set of affine constraints $\mathscr{C}$ and affine expressions $\mathscr{E}$ (Algorithm 1). This is a summary, the detailed algorithm can be found in [21]. The graph is initialized with the node *initial_node* (line 1). Then each expression and constraint is inserted as a node u and expression/constraint factorizations edges from/to u are inserted whenever the factorization is beneficial, as explained in the



(a)

(b)

(c)

(d)

Figure 5.9 – (a) Realization graph $\mathscr{G}_r$ obtained from $\mathscr{C}$, (b) Resource-efficient realization tree $\mathscr{T}$ in $\mathscr{G}_r$ (in red), (c) Resource-efficient DAG from realization tree $\mathscr{T}$ in $\mathscr{G}_r$, (d) Recursive compaction of fresh expressions $\mathscr{E}_{\text{new}}$

previous paragraphs (lines 3–4). Finally, the maximal strict subexpression with each node c' of $\mathscr{G}_r$, $v = \text{CS}(c, c')$ is inserted, and the factorization edges from/to v are inserted whenever it is beneficial (lines 5 – 8). This is required, as semantic factorizations do not include common subexpression factorizations. This way, all the solutions including a mix between common subexpression factorizations and semantic factorizations will be investigated to find an optimal compaction.

Algorithm 1: Build the realization graph $\mathscr{G}_r$ from affine constraints and expressions

```
Input  : Affine constraints $\mathscr{C}$ and affine expressions $\mathscr{E}$
Output: Realization graph $\mathscr{G}_r$
1  Insert initial_node;
2  foreach $c \in \mathscr{C} \cup \mathscr{E}$ do
3      Insert $c$ ;
4      Insert factorization edges
5      foreach $c' \in \mathscr{C} \cup \mathscr{E}$ do          ▷ Factorizations with common subexpressions
6          Insert $\text{CS}(c, c')$ ;
7          Insert factorization edges
8      end
9  end
10 Return the graph $\mathscr{G}_r$
```

Finding an optimal realization An expression factorization edge $u \xrightarrow{\Delta} v$ of the realization graph $\mathscr{G}_r$ means that expression of v (if v is a constraint $e < 0$, the expression is e) may be realized from the expression of u with a cost Δ. For instance, the edge $i+2j+k \xrightarrow{18} 5i+2j+3k$ in Figure 5.9.(a) means that $v = 5i+2j+3k$ might be realized from $u = i+2j+k$ with an expression of cost 18. Note that u and v might be realized directly with cost 22 and 122 respectively (see edges from *initial_node*). u and v might also be realized from $w = 2j$. In that case, we choose edges *initial_node* $\xrightarrow{0} w$, $w \xrightarrow{15} u$ and $w \xrightarrow{117} v$ for a total cost of $0+15+117 = 132$. From this observation, we may conclude that a realization of u and v is a subtree of $\mathscr{G}_r$ rooted at *initial_node* and including the nodes u and v to be realized. Finally, remark that constraint factorization edges $u \to \neg v$ are *always terminal*, since the expression of v is not actually computed. These observations lead to the following definition:

Definition 5.2 (Realization)**.** *Let $\mathscr{G}_r$ be the realization graph of expressions $\mathscr{E}$ and constraints $\mathscr{C}$. A* realization *is a subgraph $\mathscr{T} \subseteq \mathscr{G}_r$ which satisfies the following conditions:*

1. *Each expression/constraint is realized correctly: $\mathscr{T}$ is a tree rooted at initial_node, and $\mathscr{T}$ spans $\mathscr{E} \cup \mathscr{C}$.*
2. *No useless common subexpression is computed: the leaves of $\mathscr{T}$ belong to $\mathscr{E} \cup \mathscr{C}$.*
3. *Negation edges are terminal.*

In other words, a realization is a particular spanning tree of $\mathscr{G}_r$ (condition 1). When common subexpression are involved (white nodes in Figure 5.9.(a)), they must be intermediate results in the final realization (condition 2). Negation edges must be terminal, as their expression is not available for further factorizations (condition 3). The cost of a realization is the sum of the weights Δ on its edges. Hence, finding an efficient realization amounts to compute a *minimal spanning tree* of $\mathscr{G}_r$, under the constraints specified in definition 5.2.

We propose an algorithm in two steps. First, we use Prim's greedy heuristic to find a minimum spanning tree rooted on initial_node among the *expression factorization* edges of $\mathscr{G}_r$. Then, we look for beneficial *constraints factorization* among the *orphan nodes*: nodes u which are neither factorized (spanning tree with *initial_node* $\xrightarrow{\Delta} u$), nor implied in a factorization (u has no successors in the spanning tree). Remark that semantic factorizations require the computation of fresh expressions (expression factorization $u \xrightarrow{\Delta} v$ requires the fresh expression $u - v$, constraint factorization $u \xrightarrow{\Delta}_{\neg} v$ requires the fresh expression $-1 - u - v$). This set $\mathscr{E}_{new}$ of fresh expressions must, in turn, must be compacted. We simply apply recursively our algorithm on $\mathscr{E}_{new}$.

Back to the example On the example, our algorithm finds the spanning tree depicted with red edges on Figure 5.9.(b). The edges chosen for the realization tree are in red. The total cost of the design (72) is greatly improved compared to direct realization (487) (only arcs from initial_node, no factorization at all) and compared to common subexpression factorization (391) (edges from node $2j$ and direct realization of $-5i - 3j - 1$). (c) depicts the DAG finally produced after applying the factorizations selected in (b) by using the construction rules discussed above. In particular, (d) shows the sub-DAG obtained after the recursive compaction of fresh expressions $\mathscr{E}_{new}$: it is a sub-DAG of the final result given in (c).

5.4.5 Experimental results

We present the main result: our compaction method outperforms Intel/Altera synthesis tool by a factor of 30% on the affine control of the polyhedral kernels from the benchmark suite PolyBench/C v3.2 [129].
A detailed study of the impact of semantic factorization on the compaction, in particular the relation between semantic factorizations and strength reduction may be found in [21].

Experimental setup We have applied our algorithm to simplify the affine control generated for the kernels of the benchmark suite PolyBench/C v3.2 [129]. Table 5.3 depicts the kernels and the synthesis results obtained on FPGA. For each kernel, a DPN process network is generated using the DCC tool [20].The execution order is deeply restructured and the affine control per process (control automaton,mux/demux) can be quite complex. Before deriving the DAGs, we simplify the control polyhedra with various heuristics including gist and integer set coalescing [161]. Then, for each process, we collect the affine control and we apply our algorithm to produce a DAG. Table 5.3 presents the sum of the criteria collected for each process. #dags is the total number of DAG produced, $\#\mathscr{C}$ is the total number of affine constraints. $\#\mathscr{E}$ is the total number of affine expressions. All in all, we have produced and analyzed a total of 261 DAGs from 4990 constraints and 2464 expressions.

Synthesis results We have implemented a VHDL generator for our DAGs and a direct generator which puts the affine expressions in VHDL and let the synthesis tool do the optimizations – typically common subexpression elimination and boolean optimizations. This way, we can compare our approach to the optimizations applied by the synthesis tool. The DAGs are generated using the hierarchical approach. Both direct and optimized designs are pipelined at ALM level by adding a sufficient number of registers to the outputs. This way, the synthesis tool will perform low level logic optimizations and retiming to redistribute the registers through the design. The synthesis was performed using Quartus Prime TM 16.1.2 from Intel on the platform on the Arria 10 10AX115S2F4I1SG FPGA with default synthesis options (optimization level - balanced). Intel Quartus Prime is capable of applying highly advanced optimizations automatically including common subexpression factorization and many other advanced boolean optimizations. The DAGs were tested using GHDL simulation tool over uniformly distributed random stimuli.
The synthesis results are presented in the table 5.3. The synthesis results for our DAGs are provided in the column SEM+Quartus (semantic factorizations + quartus). The synthesis results for the direct implementation are given in the column Quartus (quartus only). The gain in ALMs compared to the direct implementation is given in the column Gain. Both implementations run at the maximum FPGA frequency of 645.16 MHz. The frequency is limited by the target MAX delay

Kernel	#dags	$\#\mathcal{C}$	$\#\mathcal{E}$	SEM+Quartus		Quartus		Gain
				ALM	Regs	ALM	Regs	
2mm	15	250	161	1011	612	1430	596	29%
3mm	18	369	206	1775	946	2528	922	30%
atax	12	134	83	628	328	900	321	30%
bicg	11	112	69	500	278	715	278	30%
correlation	27	356	205	1609	1129	2567	909	37%
covariance	16	243	143	1221	708	1872	618	35%
doitgen	9	145	124	393	280	607	268	35%
fdtd-2d	13	502	167	2339	1713	3293	1603	29%
gemm	10	125	93	672	270	851	270	21%
gemver	20	187	137	847	459	1102	438	23%
gesummv	14	95	84	456	245	549	227	17%
heat-3d	8	734	175	3545	1194	5667	2559	37%
jacobi-1d	8	134	64	628	556	912	520	31%
jacobi-2d	8	370	111	1660	1204	2547	1144	35%
lu	7	213	87	1116	666	1469	628	24%
mvt	11	118	70	550	290	758	290	27%
seidel-2d	5	226	63	1161	1464	1758	1291	34%
symm	13	213	116	1011	471	1540	465	34%
syr2k	10	135	90	721	290	944	281	24%
syrk	9	118	81	636	246	828	246	23%
trisolv	9	93	56	474	218	632	213	25%
trmm	8	118	79	549	262	806	253	32%

Table 5.3 – Synthesis results on Polybench/C v3.2

limited by the signal hold timing and other physical constraints. Both versions use a significant amount of 5-6 inputs ALMs, thus achieving higher compression ratio. There is no ALM overhead because of registers, as for all the examples they are entirely packed inside the ALMs containing logic. Experimental results show a significant gain in ALM, compared to the direct implementation optimized by Quartus (common subexpression elimination). The average gain is 30%, with a deviation of 5%. Remark that the kernel `gesummv` shows slightly less gain than the other kernels. The kernel `gesummv` has many simple polyhedra with small bitwidths in the computations. In that case, low-level boolean optimizations are more effective than semantic factorizations: arithmetic operations are merged with boolean $\wedge$ operations from the polyhedra using large (up to 7 bit) LUT. For the DAGs parts involved, this results in 3 to 4 times less ALMs than the optimized dags. The optimized dags can benefit less from these optimizations, as the bitwidth of the operations increases through the computations. Nonetheless, even for that kernel the benefit of semantic factorizations is significant. The synthesis results confirms the validity of our models and approach. Semantic factorization appears to complement nicely the optimization applied by quartus, and may be used profitably as an optimizing preprocessing for affine control.

5.5 Exposing FIFO channels

This section presents our contributions to the compilation of the channels for PPN and DPN process networks. Usually, the compilation of the channels consists into three steps:

1. *Infer the channel type* (FIFO, FIFO with register, addressable channel) by analyzing the producer/consumer communication pattern [155]
2. *Size the channel.* If the channel is not a FIFO, *derive an allocation function* σ. In Chapter 4 we propose a contribution to liveness analysis for array/channel targeting channel sizing and allocation [5]
3. If the channel is not a FIFO, *generate the synchronizations* which enforce the dependences.

Section 5.6 outlines a lightweight synchronization apparatus [19].

This section describes an important channel reorganization algorithm which is able to recover the FIFO communication patterns broken by a loop tiling. Our algorithm is proven to be *complete on DPN*: all the FIFO can be recovered. *This is an important enabling transformation for DPN.*

5.5.1 Communication patterns

A channel $c \in \mathscr{C}$ 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*) [153, 155]. The *in-order* constraint can be written:

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

The unicity constraints can be written:

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

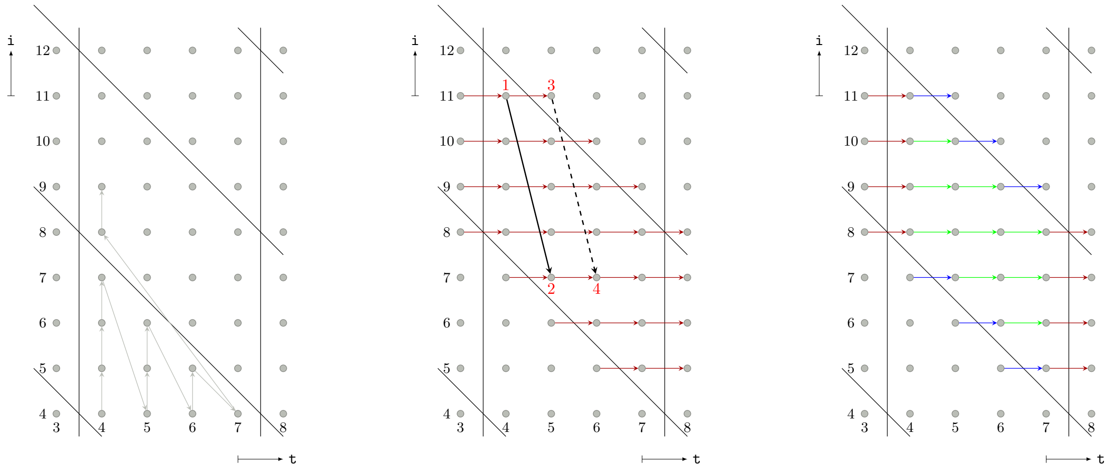
We point out that unicity depends only on the dataflow relation $\rightarrow_c$, it does not dependent on the execution order of the producer process $\prec_P$ and the consumer process $\prec_C$. The negations $\neg$in-order$(\rightarrow_c, \prec_P, \prec_C)$ and $\neg$unicity$(\rightarrow_c)$ amount to check the emptiness of a convex polyhedron, which can be done by most LP solvers. This gives an algorithm to analyze the communication pattern of a channel [155]. In particular, a channel may be implemented by a FIFO iff it verifies

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

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 [153]: 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 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 [20, 154, 157, 177].

5.5.2 Loop tiling breaks the FIFO communication patterns

Consider Figure 5.1.(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 5.10.(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 5.10.(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. We propose an algorithm to reorganize the channels so *most FIFO are recovered on PPN partitioning* (Section 5.5.4). We prove that our method is *complete on DPN partitioning*: all the FIFO are recovered.



(a) Loop tiling (b) Communication pattern (c) Our solution

Figure 5.10 – **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.

5.5.3 Our solution: splitting the channels to recover the FIFO broken

Consider Figure 5.10.(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.

Our algorithm Figure 5.11 depicts the algorithm for partitioning channels given a regular process network $(\mathscr{P}, \mathscr{C}, \theta, \mu)$ (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). $\mathscr{D}_P$ and $\mathscr{D}_C$ are assumed to be tiled with the same number of hyperplanes. P and C are assumed to share the schedule: $\theta(I_1, \dots, I_n, \vec{i}) = (I_1, \dots, I_n, \vec{i})$. In other words, *in a tile*, the execution order is the same as in the original program. In particular, this happens with our tiling scheme for I/O optimization [20]. If not, the next channel $\to_c$ 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 $\to_c^1, \dots, \to_c^n$. Finally, dependences lying in a tile (source and target in the same tile) are added to the partition (line 4): this gives $\to_c^{n+1}$. $\theta_P(x) \approx^n \theta_C(y)$ means that the n first dimensions of $\theta_P(x)$ and $\theta_C(y)$ (tiling coordinates $(I_1, \dots, I_n)$) are the same: x and y belong to the same tile.

5.5.4 Completeness on DPN, limitations on PPN

Completeness on DPN Our splitting algorithm always succeeds to recover all the FIFO on DPN [4]: 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

```
SPLIT($\rightarrow_c$,$\theta_P$,$\theta_C$)
 for $k$ := 1 to $n$
  ADD($\rightarrow_c \cap \{(x,y),\ \theta_P(x) \ll^k \theta_C(y)\}$);
 ADD($\rightarrow_c \cap \{(x,y),\ \theta_P(x) \approx^n \theta_C(y)\}$);

FIFOIZE(($\mathscr{P}$,$\mathscr{C}$,$\theta$,$\mu$))
 for each channel $c$
  $\{\rightarrow_c^1,\dots,\rightarrow_c^{n+1}\}$ := SPLIT($\rightarrow_c$,$\theta_{P_c}$,$\theta_{C_c}$);
  if fifo($\rightarrow_c^k$, $\prec_{\theta_{P_c}}$, $\prec_{\theta_{C_c}}$) $\forall k$
   REMOVE($\rightarrow_c$);
   INSERT($\rightarrow_c^k$) $\forall k$;
```

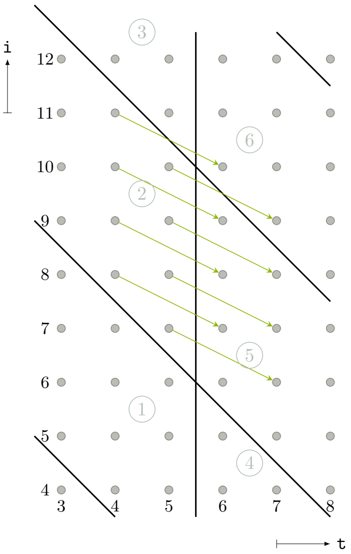
Figure 5.11 – **Our algorithm for partitioning channels** [1]. The algorithm SPLIT produces the dependence partition described on Figure 5.10.(c), for each depth k of the producer schedule θ_P and the consumer schedule θ_C. The dependence partition is kept if each set of the dependence partition has a FIFO pattern.

transformation while keeping FIFO channels, which is necessary (though not sufficient) to map it to hardware. We recall the main result, whose proof may be found in [4]:

Theorem 5.3 (Completeness on DPN)**.** *Consider a PPN* $(\mathscr{P},\mathscr{C},\theta,\omega)$ *transformed to a DPN* $(\mathscr{L},\mathscr{P}',\mathscr{S},\mathscr{C}')$ *w.r.t. a tiled schedule* $\hat{\theta}$ *where each tile is executed with the original execution order (prescribed by* θ*). Then, for each channel* $c' \in \mathscr{C}'$ *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.*

Limitations on PPN

When the RPN follows the PPN partitioning, there is no guarantee that the algorithm will always succeed to recover FIFO. The side figure gives a counter example. We modify the kernel given on Figure 5.1.(a) to observe a single dependence $(t,i) \mapsto (t+2, i-1)$ on the compute domain (•) 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. The next section will, in particular, assess the capabilities of our algorithm to recover FIFO on general PPN.

5.5.5 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 [129]. We have checked the completeness of the algorithm on PPN with DPN partitioning (Table 5.4), study the behavior of the algorithm on general PPN, without DPN partitioning (Table 5.5). Each kernel is tiled to reduce I/O while exposing parallelism [20] 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 5.4 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: $\#\{c \mid \forall c' : \mu(c') = c \Rightarrow c'$ 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: $\#\{c \mid \exists c' : \mu(c') = c \wedge c'$ 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 5.5 shows how the algorithm can recover FIFO on a tiled PPN without the DPN partitioning. The columns have the same meaning as the table 5.4: 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 above, 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 5.4 – Detailed results on PPN with DPN execution scheme. The algorithm is **complete on DPN**: all the FIFO were recovered (%fail = 0)

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%

Impact on storage requirements The side table 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 storage resources are due to our sizing heuristic [5], 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 decomposes a channel into channels of a smaller dimension, for which our sizing heuristic is more precise. In a way, our algorithm allows to find 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.

5.6 Synchronizing non-FIFO communications

When a channel cannot be realized by a FIFO, synchronizations must enforce flow- and anti-dependences between the producer and the consumer:

- The consumer cannot read a value before it is produced (*flow dependence*).
- The producer must wait for a storage slot to be freed before writing a value (*anti dependence*).

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

In the Compaan project, the data flows through a FIFO from the producer process to the consumer process, where it is stored *locally* into a *reordering buffer*, which ensures consumer synchronization and correct retrieval of consumer reads. Several realizations of reordering buffers have been proposed, such as pseudo-polynomial [154] or Content Addressable Memory (CAM) based implementations [157, 177]. The authors report a negative impact on resource usage and on the overall throughput. Also, the *multiplicity* is not handled directly with these implementations: when a data is read several times, it is up to the consumer to remove the data from the buffer after the *last read* [153]. The last read is derived as a subset of the consumer iterations responsible for a last read. It is a union of convex polyhedra, with as many polyhedra as corner cases. However, *this union is huge when the iterations are tiled* (we implemented it), hence none of these implementations are appropriate for our purpose.

Method First, non-FIFO channels are sized and allocated with the array contraction technique described in Chapter 4. Then, each channel is equipped with a *synchronization unit*. Prior to execute an iteration, a process requests a *consumer clearance* from the *synchronization unit* of the read channel. Then it computes a value and it requests a *producer clearance* from the *synchronization unit* of the written channel (see Figure 5.12).
The synchronization unit is equipped with two predicates: Freeze(P) and Freeze(C) with deduces from the requested iterations of the producer $\vec{i}_P$ and the consumer $\vec{i}_C$ if they can be executed.

Consumer freezing predicate The consumer waits for a data to written by the producer before reading it:

$$\text{Freeze(C)} \iff \theta_P(\vec{i}_P) \overset{\ll}{=} \theta_P(\sigma(\vec{i}_C))$$

where σ is the *source* function (giving the producer iteration producing the value to be read). This forces the consumer to wait for the producer (source) to write the value.

Producer freezing predicate The producer waits for a slot to be freed before overwriting it. Since $\vec{i}_C$ cannot be checked easily to be a last read iteration, we use the relaxed predicate:

$$\text{Freeze(P)} \iff \theta_C(\vec{i}_C) \ll^{\leq d} \theta_P(\vec{i}_P)$$

Where d is the *depth* of the dependence resolved by the channel, *w.r.t.* θ_P and θ_C. This forces the producer to be scheduled behind the consumer. This way, the value hold by the previous occurrence of the dependence will not be over written.

Freeze(P) and Freeze(C) may be expressed with a small affine formula. This gives a very *lightweight*, but efficient synchronization apparatus, *which can handle any communication pattern.*

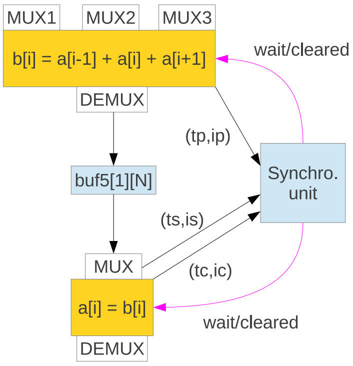


Figure 5.12 – Synchronization of non-FIFO channels

Kernel	Producer efficiency (%)	Consumer efficiency (%)
Vector Sum 1024 (tiles 64)	79%	69%
Vector Sum 1024 (tiles 256)	96%	75%
Matrix Sum 1024x1024 (tiles 64 × 64)	94%	92%
Matrix Multiply 64x64 (tiles 32 × 32)	4%	98%
Jacobi1D 8x512 (tiles 4 × 64)	88%	72%
Cholesky 256x256, (tiles 64 × 64)	90%	98%

Table 5.6 – Average efficiency of synchronization units

Experimental results The synchronizations units have been implemented both on the front-end DCC and in the Xtremlogic back-end (which derives a verilog description of the circuit). In our hardware implementation, the processes *continuously check the status* of the synchronization units by sending read/write requests *until the synchronization unit issues a clearance*. Among all the requests send to a synchronization unit (#cycle_total, one per cycle) some are unsuccessful (#cycles_freeze, the synchronization unit issues a freeze). This way, we can define the *efficiency* as:

$$\text{efficiency} = \frac{\text{cycles\_total} - \text{cycles\_freeze}}{\text{cycles\_total}}$$

Depending on the side where we count the requests and the freezes (from the producer, or from the consumer), we obtain the *producer efficiency* and the *consumer efficiency*.

Table 5.6 depicts the results obtained on 5 kernels with different tile sizes: the sum of two vectors (*Vector Sum*), the sum (resp. multiplication) of two matrices (*Matrix Sum* resp. *Matrix Multiply*), the Jacobi-1D stencil encoded as a perfect loop nest (see Figure 5.1.(a), *Jacobi1D*) and the Cholesky solver from the benchmark suite PolyBench/C v3.2 [129] (*Cholesky*). The kernels were tiled along the lines described in Section 5.3 to amortize the request latency. Then, we measure, through a RTL simulation, the *average efficiency* for *load channels*. For these channels, *the producer is a load process*, and the consumer is a *compute process*. Hence, *the important information is the efficiency on the consumer side*, which impacts directly the overall throughput.

For small tiles and low reuse (*vector sum*, tile 64), the efficiency is degraded because of the pipeline bubbles in the control part of the channel. For the kernels with high data reuse (*Matrix Multiply*), the efficiency of the consumer side is very high, as the tiling enforces data locality. Artificially, the producer efficiency is low because of the data reuse (4%). Indeed, the producer (the *load* process) needs to wait for the data to be completely reused before issuing new data.

5.7 Conclusion

In this chapter, we have presented the data-aware process networks (DPN), a dataflow intermediate representation for HLS which cross fertilizes dataflow parallelism and the data transfer optimization method developed in Chapter 4. Like polyhedral process networks, DPN are encompassed by regular process networks (RPN), a general family of dataflow intermediate representations for HLS, which combines the power of computation and communication partitioning, with a flexible dataflow model of computation. Then, we have presented our contributions to map a DPN to circuit. We have outlined our algorithms to synthesize the process control, enforce FIFO channels after a tiling and compile the synchronizations for non-FIFO channels. Apart from DPN, we have presented a loop scheduling technique to improve the throughput of circuits with pipelined arithmetic operators. These contributions are practice-oriented: we built a complete front-end C → DPN able to produce a DPN from an annotated C kernel. This software is transferred to the XtremLogic startup and is part of the production HLS compiler.

DPN and more generally RPN opens new opportunities for HLS, since the target features (multi-FPGA, memory hierarchy) might be addressed with a relevant partitioning, hence a RPN instance. Also, many challenges must be addressed to map efficiently a DPN to a circuit. In particular, buffers are duplicated to allow parallel reads, processes are produced from statements in the same loop, hence with the same control automaton. Thus, factorization techniques are required.

Also, RPN-level scheduling techniques are required, for instance to keep process pipelines busy. This hits one of the fundamental limits of the polyhedral model, since affine schedules are inherently bounded to fork/join parallelism and cannot express pipelined parallelism. How to extend polyhedral scheduling to prescribe a dataflow execution is still opened today, despite the recent progresses with polynomial scheduling.

6 Conclusion and perspectives

The contributions described in this document addressed the design of compiler algorithms to generate efficient software and hardware. We leverage the polyhedral model to phrase our algorithms, and we operate a cross-fertilization between the polyhedral community, nowadays more software oriented though its systolic roots; and the HLS community, which is progressively acquiring automatic parallelization techniques from the HPC community. We have implemented entirely our research results around HLS as a research compiler, DCC, transferred to the start-up XtremLogic under an Inria License, where it is part of the production HLS compiler.

6.1 Summary of the manuscript

Chapter 2 presents several extensions of loop tiling, a standard loop transformation widely used in compiler optimizations, in particular in all the contributions described in this document. We rephrase the polyhedral formulation of loop tiling to enable any polytopic tile shape, whose size depends on a scaling parameter [25, 26]. The scaling parameter survives the compilation, and may be used at runtime to tune the tile size. Since it is formulated as a polyhedral transformation, and not simply in back-end code generation, it makes possible to reason about the tile size directly in the polyhedral transformation. Also it makes possible to produce parametrized programs, tuneable at runtime, which opens the way to partial polyhedral compilation. We also propose a polyhedral formulation to semantic tiling [23], a transformation increasing the granularity of operations (scalar → matrix). Semantic tiling is used in the numerical community (where it is sometimes referred to as tiling or blocking, though it is not) to realize the tiles with library functions. This task is automated thereafter.

Chapter 3 builds on the ideas developed in my PhD thesis [2] to refactor a program with a performance library [3, 6, 7, 8, 9]. We leverage our semantic tiling transformation to extract the subcomputations to be realized by library functions. We address the equivalence checking of two programs involving reductions [24], this algorithm was finally not used in the framework. We propose a template matching semi-algorithm and we show how to apply it to reengineer a semantic tile as a composition of library calls. The whole system has been validated on 4 medium-sized applications. This work is under publication.

Chapter 4 presents an HLS algorithm to optimize the data transfers between an FPGA circuit and an off-chip memory. In particular, we propose a template of architecture [12, 13] and an algorithm to schedule the data transfers [14, 15] ensuring a maximal communication coalescing, while hiding the communications with computations. We designed our HLS algorithm as a source-to-source transformation in front of a mainstream HLS tool, in charge of producing the circuit. We leverage a loop tiling to improve the data reuse in local memory and to rule the memory prefetch. This approach allows the future exploration of design trade-offs simply by playing on the tile size: local memory size for memory traffic, local memory size for parallelization degree; thereby it gives a simple way to explore the roofline model of the input application in search of *the* optimal operational intensity.

Chapter 5 leverages the ideas behind our I/O system to propose a complete automatic parallelization methodology for HLS. Starting from Polyhedral Process Networks (PPN), we propose a new model of computation, Data-aware Process Networks (DPN), which explicits both parallelism and off-chip memory accesses; then we propose a compiler-oriented generalization, Regular Process Networks (RPN), which encompass many polyhedral parallel intermediate languages/models of computation (PPN, DPN, CRP) and leads to a general automatic parallelization methodology. DPN serves as an intermediate representation in our compiler. In particular, we propose several compilation algorithms which apply to DPN while being general enough to apply to other contexts. We investigate the compilation of the *process control* (control compaction [21], single process scheduling [17, 18]), the compilation of *channels* (typing [1, 4], allocation/sizing [5], synchronizations [19]). This contribution is practice-oriented: we built a complete *front-end C* → DPN able to produce a DPN from an annotated C kernel. This front-end is transferred to the XtremLogic startup under an Inria license and is part of the production HLS compiler.

6.2 Future research topics

My contributions promote the cross-fertilization of compiler techniques for high-performance computing and high-level synthesis techniques. The implementation of HPC applications is a subtle balance between the static part (compiler) and the dynamic part (runtime/OS). Despite the success of polyhedral compilation techniques, compiler-based automatic parallelization is still seen as an unreachable grail. We do believe it is not the case and that time has come to rebalance the compiler part in the design process. The FPGA technology pushed by industry provides a unique opportunity to do so, since most configuration decisions must be made statically.

6.2.1 Compiler support for task-level parallelism

This research project is part of a current Inria team proposal at LIP

So far, the polyhedral model addresses *fine-grain parallelization* at kernel-level. With FPGAs, many configuration decisions must be made at *compile time*. Hence the need to increase the compiler support for exploiting coarse-grain parallelism on multi-FPGA systems. All the aspects of parallelization need to be investigated: decomposition in tasks, intra/inter task scheduling, load balancing, data transfers & synchronization. These aspects are, as always with compilation, completely inter-dependent and strongly constrained by the target architecture.

Semantic tiling gives a possible task splitting. However there is no warranty that it will minimize the data transfer and maximize inner parallelism and data locality. A schedule-driven splitting, which pushes most of the dependences into a task (similarly to *pluto* algorithm [53]) is probably a part of the solution. Scalability issues must also be addressed, as the input application is expected to be larger than a polyhedral kernel (hundreds rather than tens of lines). Furthermore, the frontier between runtime schedule and compile-time scheduling must be redefined in light of FPGA constraints. If the application is completely polyhedral, we may imagine a purely static, compile-time, scheduling. But then we would be bounded by scalability issues. This advocates for mixed static/dynamic approach. How to use properly the FPGA resources must also be investigated. In particular the parallelism of kernels must be tuned dynamically to fit the FPGA resources while ensuring a minimal latency. This raises the question of *partial compilation* addressed partially by monoparametric tiling.

Collaborators on this topic: Laure Gonnord (Univ. of Lyon), Ludovic Henrio (CNRS), Matthieu Moy (Univ. of Lyon).

6.2.2 HLS-specific dataflow optimizations

This research project is addressed in a "Projet Emergence ENS" with Matthieu Moy, which was accepted in December 2018. I am the principal investigator of this project.

We want to address the HLS perspectives opened by Data-aware process networks. Historically, we first addressed a source-to-source HLS approach (Chapter 4) – more suited for academic research – then we moved to a fully integrated approach (Chapter 5) – more viable for a production compiler. Though the latter will clearly give better results, we are bounded by intellectual property issues, which prevents us to work directly on XtremLogic back-end. This pushes us to get back to a source-to-source approach. Hence, the first step is to write a *back-end* DPN → FPGA *via* an existing mainstream HLS tool: DPN → C $\xrightarrow{HLS}$ FPGA. How to feed the HLS tool with a C implementation of a DPN so the main features (dataflow execution, parallelism, FIFO channels, I/O) are conveyed by the HLS process is quite challenging but reachable.
DPN is an intermediate representation, not a circuit. Hence, optimization passes are required before obtaining a reasonable circuit. Some may even be applied by default by the HLS tool itself. The first point is the elimination of redundancies induced by the DPN model itself: buffers are duplicated to allow parallel reads, processes are produced from statements in the same loop and thus with the same control automaton. We plan to study how these constructs can be factorized at C-level and to design the appropriate DPN-to-C translation algorithms.
Also, we want to develop DPN-level analyses and transformations to quantify the optimal reachable throughput and to reach it. We expect DPN-style parallelism to increase the throughput, but in turn it may require an operational intensity beyond the optimal point of the FPGA roofline model. Our scheduling algorithm for perfect loop nests [18] is a good starting point. We will assess the trade-offs, build the cost-models, and the relevant dataflow transformations.

Collaborators on this topic: Matthieu Moy (Univ. Lyon)

6.2.3 Scaling the polyhedral model

This research project is part of an on-going eureka celtic-plus proposal.

One of the biggest challenges with the polyhedral model is the space complexity of intermediate representations. Two operations usually lead in a complexity explosion: subtraction of convex polyhedra and operations between piece-wise affine functions (e.g. $\min, \max$). With DPNs, this translates to a complex control which may jeopardize the compilation. In our DCC compiler, we managed to reduce that complexity with a myriad of tricks. This made it possible to pass all the polybenchs kernels (even heat-3d). Nontheless, the issue arises each time a new polyhedral algorithm is developed. hence the need to find general solutions.
We believe that laziness is the solution! Instead of computing a closed form at compile-time, we let the runtime evaluate it on demand. But then, how to compose with lazy values is an open problem, somehow related to partial compilation. A good starting point is to study how the monoparametric tiling transformation may be expressed in a lazy manner, while staying polyhedral. Another way is to explore how on-the-fly evaluation can reduce the complexity of the control. A good starting point is the control required for the load process (which fetches data from off-chip memory). If we want to avoid multiple load of the same data, the FSM (Finite State Machine) of the load process might be very complex. We believe that dynamic construction of the load set (set of data to load from the main memory) will use less silicon than an FSM with large piecewise affine functions computed statically.

Collaborators on this topic: David Castell-Rufas (U. of Barcelona)

A Further Contributions

This appendix summarizes some of our contributions not described in the manuscript. Section A.1 gives a brief summary of our contributions to *termination analysis* (basically, checking if a program with *while* loops terminates, then bound the computational complexity). Then, we outline briefly our *post-doc contributions* in Section A.3 and Section A.2.

A.1 Termination analysis [10, 11, 16, 22]

Ranking method In [10], we have proposed a program analysis to check the termination of a program encoded as an integer interpreted automata (See Definition 3.1). To do so, we synthesize a *ranking function*: a reverted scheduling function which decreases in a well founded set each time a transition is fired. If such a function can be derived, then the program terminates.
The technique is simple: we reinterpret the integer interpreted automata as a PRDG (state → statement, transition → dependence), then we derive an *affine schedule* [83] while reverting the causality constraint: if $(\vec{v}_p, \vec{v}_q) \in F_{p,q}$, then $\text{rank}_q(\vec{v}_q) \ll \text{rank}_p(\vec{v}_p)$. The difficulty is to associate an iteration domain (an *invariant*) to each state of the automata. To do so, we use *linear relation analysis* based on *abstract interpretation techniques* [89].
When the invariant is not precise enough, no ranking function is found, and we cannot conclude. We also propose a method to infer the program complexity based on polyhedron counting techniques [62].

Extensions In [22], we extended this technique in a *scalable* and *modular* way. The program to analyse is reduced to the smallest relevant subset through a *termination-specific slicing technique*. Then, the program is divided into pieces of code that are analyzed separately, thanks to an external engine for termination.

▷ **Software:** Rank. We have implemented these techniques in a tool called Rank[a] [11]. Aside these techniques, Rank features a non-termination analysis: in some cases, Rank can infer an input causing the program to loop.

[a] Rank is available at `http://compsys-tools.ens-lyon.fr/rank` and can be tried online

Monotone interpretation method We show in [16] how *monotone interpretations* – a termination analysis technique for term rewriting systems – can be used to assess the *inherent parallelism of recursive programs manipulating inductive data structures*. As a side effect, we show how monotone interpretations specify a parallel execution order, and how our approach extends naturally affine scheduling to recursive programs.

A.2 SSA-form with array regions [28]

Program optimization requires a representation of data dependences. Historically, the SSA-form has been introduced as a representation of program dependences to enable pseudo-code optimization. Then, SSA-form moved to back-end passes, from instruction selection [79] to register allocation where it boils down to optimal algorithms [96] thanks to the properties induced on the interference graph.

A SSA-form called *region array SSA* is proposed, which explicits the data flow across *array variables* by adding array regions in the annotation. The ϕ functions explicit the source array regions and can be view as a relaxed version of array dataflow analysis. The array regions are parametrized by dynamic informations (e.g. `if x>0 then` $\mathscr{D}_1$ `else` $\mathscr{D}_2$), which was intended to ease the construction of runtime parallelization algorithms. We implemented the region array SSA form in the Polaris compiler.

A.3 Automatic vectorization [27]

Modern computer architectures features arithmetic instructions operating on vectors. Often, the load instructions assume an alignment of the vectors in memory. If not, expensive reorganization operations are required to set properly the vector registers.

This property may be enforced by reorganizing the data layout of arrays while rescheduling the computation. But still, there remains to generate the code with vector instructions, which is challenging as the parallelism is sometimes not sufficient to fill out full vectors. We have proposed a polyhedral algorithm to extract the complete – full – vector operations and to produce the final vector code.

B Bibliography

B.1 Personal publications

[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. *Program Optimization by Template Recognition and Replacement*. PhD thesis, Université de Versailles, 2005.

[3] Christophe Alias. Tema : an efficient tool to find high-performance library patterns in source code. In *International Workshop on Patterns in High-Performance Computing (PatHPC'05)*, 2005.

[4] Christophe Alias. FIFO recovery by depth-partitioning is complete on data-aware process networks. Technical Report RR 9187, INRIA, June 2018.

[5] 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 SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'07)*, 2007.

[6] Christophe Alias and Denis Barthou. Algorithm recognition based on demand-driven data-flow analysis. In *IEEE Working Conference on Reverse Engineering (WCRE'03)*, 2003.

[7] Christophe Alias and Denis Barthou. On the recognition of algorithm templates. In *International Workshop on Compiler Optimization meets Compiler Verification (COCV'03)*, 2003.

[8] Christophe Alias and Denis Barthou. Deciding where to call performance libraries. In *European Conference on Parallel Processing (Euro-Par'05)*, 2005.

[9] Christophe Alias and Denis Barthou. On domain specific languages re-engineering. In *IEEE/ACM International Conference on Generative Programming and Component Engineering (GPCE'05)*, 2005.

[10] 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.

[11] Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. Rank: A tool to check program termination and computational complexity. In *International Workshop on Constraints in Software Testing Verification and Analysis (CSTVA'13)*, page 238, Luxembourg, March 2013.

[12] Christophe Alias, Alain Darte, and Alexandru Plesco. Optimizing DDR-SDRAM communications at c-level for automatically-generated hardware accelerators. an experience with the altera C2H HLS tool. In *IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP'10)*, 2010.

[13] Christophe Alias, Alain Darte, and Alexandru Plesco. Optimizing remote accesses for offloaded kernels: Application to high-level synthesis for FPGA. In *17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)*, 2012.

[14] Christophe Alias, Alain Darte, and Alexandru Plesco. Optimizing remote accesses for offloaded kernels: Application to high-level synthesis for FPGA. In *2nd International Workshop on Polyhedral Compilation Techniques (IMPACT'12)*, 2012.

[15] 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.

[16] Christophe Alias, Carsten Fuhs, and Laure Gonnord. Estimation of Parallel Complexity with Rewriting Techniques. In *15th International Workshop on Termination (WST'16)*, Obergurgl, Austria, September 2016.

[17] Christophe Alias, Bogdan Pasca, and Alexandru Plesco. Automatic generation of FPGA-specific pipelined accelerators. In *International Symposium on Applied Reconfigurable Computing (ARC'11)*, 2011.

[18] Christophe Alias, Bogdan Pasca, and Alexandru Plesco. FPGA-specific synthesis of loop-nests with pipeline computational cores. *Microprocessors and Microsystems*, 36(8):606–619, November 2012.

[19] Christophe Alias and Alexandru Plesco. Method of automatic synthesis of circuits, device and computer program associated therewith. Patent FR1453308, April 2014.

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

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

[22] Guillaume Andrieu, Christophe Alias, and Laure Gonnord. SToP: Scalable termination analysis of (C) programs (tool presentation). In *International Workshop on Tools for Automatic Program Analysis (TAPAS'12)*, Deauville, France, September 2012.

[23] Guillame Iooss, Sanjay Rajopadhye, and Christophe Alias. Semantic tiling. In *Workshop on Leveraging Abstractions and Semantics in High-performance Computing (LASH-C'13)*, Shenzhen, China, February 2013.

[24] Guillaume Iooss, Christophe Alias, and Sanjay Rajopadhye. On program equivalence with reductions. In *21st International Static Analysis Symposium (SAS'14)*, Munich, Germany, September 2014.

[25] Guillaume Iooss, Christophe Alias, and Sanjay Rajopadhye. Monoparametric tiling of polyhedral programs. Technical Report RR 9233, INRIA, December 2018.

[26] Guillaume Iooss, Sanjay Rajopadhye, Christophe Alias, and Yun Zou. CART: Constant aspect ratio tiling. In Sanjay Rajopadhye and Sven Verdoolaege, editors, *4th International Workshop on Polyhedral Compilation Techniques (IMPACT'14)*, Vienna, Austria, January 2014.

[27] Qingda Lu, Christophe Alias, Uday Bondhugula, Thomas Henretty, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, P. Sadayappan, Yongjian Chen, Haibo Lin, and Tin fook Ngai. Data layout transformations for enhancing data locality on NUCA chip multiprocessors. In *ACM/IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT'09)*, 2009.

[28] Silvius Rus, Guobin He, Christophe Alias, and Lawrence Rauchwerger. Region array SSA. In *ACM/IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT'06)*, 2006.

B.2 General references

[29] Impulse-C. http://www.impulseaccelerated.com.

[30] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. In *Journal of Physics: Conference Series*, volume 180, page 012037. IOP Publishing, 2009.

[31] Corinne Ancourt and François Irigoin. Scanning polyhedra with DO loops. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP), Williamsburg, Virginia, USA, April 21-24, 1991*, pages 39–50, 1991.

[32] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[33] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[34] Franz Baader and Wayne Snyder. *Unification theory*, chapter 8. Elsevier Science Publisher, B.V., alan robinson and andrei voronkov edition, 2001.

[35] Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. Tiling stencil computations to maximize parallelism. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 1–11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[36] John Baras and George Theodorakopoulos. *Path Problems in Networks*. Morgan & Claypool, 2010.

[37] Denis Barthou, Paul Feautrier, and Xavier Redon. On the Equivalence of Two Systems of Affine Recurrence Equations (Research Note). In *Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, pages 309–313, 2002.

[38] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In *13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08)*, pages 1–10, Salt Lake City, UT, USA, February 2008. ACM.

[39] Muthu Manikandan Baskaran, Albert Hartono, Sanket Tavarageri, Thomas Henretty, J. Ramanujam, and P. Sadayappan. Parameterized tiling revisited. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 200–209, New York, NY, USA, 2010. ACM.

[40] Muthu Manikandan Baskaran, Nicolas Vasilache, Benoit Meister, and Richard Lethin. Automatic communication optimizations through memory reuse strategies. In *ACM SIGPLAN Notices*, volume 47, pages 277–278. ACM, 2012.

[41] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam. Putting polyhedral loop transformations to work. In *International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2003.

[42] 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.

[43] Cedric Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 7–16. IEEE Computer Society, 2004.

[44] Cédric Bastoul, Albert Cohen, Sylvain Girbal, Saurabh Sharma, and Olivier Temam. Putting polyhedral loop transformations to work. In *LCPC*, pages 209–225, 2003.

[45] Samuel Bayliss and George A Constantinides. Optimizing sdram bandwidth for custom fpga loop accelerators. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, pages 195–204. ACM, 2012.

[46] Marcus Bednara and Jürgen Teich. Automatic synthesis of fpga processor arrays from loop algorithms. *The Journal of Supercomputing*, 26(2):149–165, 2003.

[47] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. In *International Conference on Compiler Construction*, pages 283–303. Springer, 2010.

[48] S. Bhansali and J. R. Hagemeister. A pattern-matching approach for reusing software libraries in parallel systems. In *First International Workshop on Knowledgebased Systems for the ReUse of Program Libraries*, 1995.

[49] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology. In *Proceedings of the 11th international conference on Supercomputing*, pages 340–347. ACM, 1997.

[50] Michaela Blott. Reconfigurable future for hpc. In *High Performance Computing & Simulation (HPCS), 2016 International Conference on*, pages 130–131. IEEE, 2016.

[51] Bernard Boigelot. *Symbolic methods for exploring infinite state spaces*. PhD thesis, Université de Liège, Liège, Belgium, 1998.

[52] Uday Bondhugula, Vinayaka Bandishti, and Irshad Pananilath. Diamond tiling: Tiling techniques to maximize parallelism for stencil computations. *IEEE Transactions on Parallel and Distributed Systems*, 2016.

[53] 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.

[54] Pierre Boulet and Paul Feautrier. Scanning polyhedra without Do-loops. In *IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, pages 4–9, 1998.

[55] Jichun Bu, Ed F Deprettere, and P Dewilde. A design methodology for fixed-size systolic arrays. In *Application Specific Array Processors, 1990. Proceedings of the International Conference on*, pages 591–602. IEEE, 1990.

[56] Altera C2H: Nios II C-to-hardware acceleration compiler. http://www.altera.com/products/ip/processors/nios2/tools/c2h/ni2-c2h.html.

[57] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Acm Sigplan Notices*, volume 40, pages 519–538. ACM, 2005.

[58] Daniel Chavarría-Miranda and John Mellor-Crummey. Effective communication coalescing for data-parallel applications. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'05)*, pages 14–25, Chicago, IL, USA, 2005. ACM.

[59] Wei-Yu Chen, Costin Iancu, and Katherine Yelick. Communication optimizations for fine-grained UPC applications. In *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, pages 267–278. IEEE Computer, 2005.

[60] Hamidreza Chitsaz, Raheleh Salari, S. Cenk Sahinalp, and Rolf Backofen. A partition function algorithm for interacting nucleic acid strands. *Bioinformatics*, 25(12):i365–i373, 2009.

[61] Jaeyoung Choi, James Demmel, Inderjiit Dhillon, Jack Dongarra, Susan Ostrouchov, Antoine Petitet, Ken Stanley, David Walker, and R Clinton Whaley. Scalapack: A portable linear algebra library for distributed memory computers—design issues and performance. *Computer Physics Communications*, 97(1-2):1–15, 1996.

[62] Philippe Clauss. Counting solutions to linear and nonlinear constraints through ehrhart polynomials: Applications to analyze and transform scientific programs. In *Proceedings of the 10th international conference on Supercomputing, ICS 1996, Philadelphia, PA, USA, May 25-28, 1996*, pages 278–285, 1996.

[63] Jason Cong, Hui Huang, Chunyue Liu, and Yi Zou. A reuse-aware prefetching scheme for scratchpad memory. In *Proceedings of the 48th annual Design Automation Conference (DAC'11)*, pages 960–965, 2011.

[64] NVidia Corporation. Cuda. https://developer.nvidia.com/cuda-zone.

[65] OpenACC Non-Profit Corporation. The openacc application programming interface version 2.0. http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf, jun 2013.

[66] Philippe Coussy and Adam Morawiec. *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer, 2008.

[67] 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.

[68] Alain Darte. Regular partitioning for synthesizing fixed-size systolic arrays. *INTEGRATION, the VLSI journal*, 12(3):293–304, 1991.

[69] Alain Darte and Alexandre Isoard. Exact and approximated data-reuse optimizations for tiling with parametric sizes. In *CC*, pages 151–170, 2015.

[70] Alain Darte, Rob Schreiber, and Gilles Villard. Lattice-based memory allocation. In *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '03, pages 298–308, New York, NY, USA, 2003. ACM.

[71] Alain Darte, Robert Schreiber, B. Ramakrishna Rau, and Frédéric Vivien. Constructing and exploiting linear schedules with prescribed parallelism. *ACM Transactions on Design Automation of Electronic Systems (ACM TODAES)*, 7(1):159–172, 2002.

[72] Alain Darte, Robert Schreiber, and Gilles Villard. Lattice-based memory allocation. *IEEE Transactions on Computers*, 54(10):1242–1257, October 2005.

[73] Florent de Dinechin and Bogdan Pasca. Designing custom arithmetic data paths with flopoco. 2011.

[74] Florent de Dinechin, Patrice Quinton, and Tanguy Risset. Structuration of the Alpha language. In *Massively Parallel Programming Models*, pages 18–24. IEEE, 1995.

[75] Eddy De Greef, Francky Catthoor, and Hugo De Man. Memory size reduction through storage order optimization for embedded parallel multimedia applications. *Parallel Computing*, 23:1811–1837, 1997.

[76] E. F. Deprettere, E. Rijpkema, P. Lieverse, and B. Kienhuis. Compaan: Deriving process networks from Matlab for embedded signal processing architectures. In *8th International Workshop on Hardware/Software Codesign (CODES'2000)*, San Diego, CA, May 2000.

[77] Steven Derrien and Sanjay Rajopadhye. Loop tiling for reconfigurable accelerators. In *International Conference on Field Programmable Logic and Applications*, pages 398–408. Springer, 2001.

[78] Romain Dolbeau, Stéphane Bihan, and François Bodin. Hmpp: A hybrid multi-core parallel programming environment. In *Workshop on general purpose processing on graphics processing units (GPGPU 2007)*, volume 28, 2007.

[79] Dietmar Ebner, Florian Brandner, Bernhard Scholz, Andreas Krall, Peter Wiedermann, and Albrecht Kadlec. Generalized instruction selection using ssa-graphs. In *ACM Sigplan Notices*, volume 43, pages 31–40. ACM, 2008.

[80] Paul Feautrier. Parametric integer programming. *RAIRO-Operations Research*, 22(3):243–268, 1988.

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

[82] Paul Feautrier. Some efficient solutions to the affine scheduling problem. Part I. one-dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, October 1992.

[83] Paul Feautrier. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, December 1992.

[84] Xiushan Feng and Alan J. Hu. Cutpoints for Formal Equivalence Verification of Embedded Software. In *Proceedings of the 5th ACM International Conference on Embedded Software*, pages 307–316, 2005.

[85] Jeff Fifield, Ronan Keryell, Hervé Ratigner, Henry Styles, and Jim Wu. Optimizing OpenCL applications on xilinx fpga. In *Proceedings of the 4th International Workshop on OpenCL*, page 5. ACM, 2016.

[86] Matteo Frigo and Steven G Johnson. FFTW: An adaptive software architecture for the fft. In *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, volume 3, pages 1381–1384. IEEE, 1998.

[87] Al Geist and Daniel A Reed. A survey of high-performance computing scaling challenges. *International Journal of High Performance Computing Applications*, page 1094342015597083, 2015.

[88] Benny Godlin and Ofer Strichman. Regression Verification. In *Proceedings of the 46th Annual Design Automation Conference*, pages 466–471, 2009.

[89] Laure Gonnord. *Accélération abstraite pour l'amélioration de la précision en Analyse des Relations Linéaires*. PhD thesis, Université Joseph Fourier - Grenoble, 2007.

[90] Mentor Graphics. Mentor CatapultC high-level synthesis. http://www.mentor.com.

[91] Tobias Grosser, Albert Cohen, Justin Holewinski, P. Sadayappan, and Sven Verdoolaege. Hybrid hexagonal/classical tiling for GPUs. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 66–75, New York, NY, USA, 2014. ACM.

[92] Tobias Grosser, Hongbin Zheng, Ragesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. Polly – polyhedral optimization in LLVM. In C. Alias and C. Bastoul, editors, *1st International Workshop on Polyhedral Compilation Techniques (IMPACT)*, pages 1–6, Chamonix, France, 2011.

[93] Armin Größlinger. Precise management of scratchpad memories for localizing array accesses in scientific codes. In O. de Moor and M. Schwartzbach, editors, *International Conference on Compiler Construction (CC'09)*, volume 5501 of *Lecture Notes in Computer Science*, pages 236–250. Springer-Verlag, 2009.

[94] Serge Guelton, François Irigoin, and Ronan Keryell. Compilation for heterogeneous computing: Automating analysis, transformations, and decisions. Technical Report A-450, Ecole des Mines de Paris, 2011.

[95] Gautam Gupta and Sanjay Rajopadhye. The Z-polyhedral model. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 237–248. ACM, 2007.

[96] Sebastian Hack, Daniel Grund, and Gerhard Goos. Register allocation for programs in SSA-form. In *International Conference on Compiler Construction*, pages 247–262. Springer, 2006.

[97] Tony Hoare. The Verifying Compiler: A Grand Challenge for Computing Research. In *Proceedings of the 2003 Joint Modular Languages Conference*, pages 25–35, 2003.

[98] Guillaume Iooss. *Detection of linear algebra operations in polyhedral programs*. PhD thesis, Colorado State University & Ecole Normale Supérieure de Lyon, 2016.

[99] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'88, pages 319–329, January 1988.

[100] Ilya Issenin, Erik Borckmeyer, Miguel Miranda, and Nikil Dutt. DRDU: A data reuse analysis technique for efficient scratch-pad memory management. *ACM Transactions on Design Automation of Electronics Systems (ACM TODAES)*, 12(2), April 2007. Article 15.

[101] JEDEC. Double data rate (DDR) SDRAM specification JESD79F. http://www.jedec.org/download/search/JESD79F.pdf.

[102] Gilles Kahn. The semantics of simple language for parallel programming. In *IFIP Congress 74*, pages 471–475, 1974.

[103] C. Karfa, K. Banerjee, D. Sarkar, and C. Mandal. Verification of loop and arithmetic transformations of array-intensive behaviors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(11):1787–1800, Nov 2013.

[104] Richard M Karp, Raymond E Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–590, 1967.

[105] Wayne Kelly, William Pugh, Evan Rosser, and Tatiana Shpeisman. Transitive closure of infinite graphs and its applications. *International Journal of Parallel Programming*, 24(6):579–598, 1996.

[106] Christoph W. Kessler. Pattern-driven automatic parallelization. *Scientific Programming*, 5(3):251–274, August 1996.

[107] DaeGon Kim and Sanjay Rajopadhye. Efficient tiled loop generation: D-tiling. In *Proceedings of the 22Nd International Conference on Languages and Compilers for Parallel Computing*, LCPC'09, pages 293–307, Berlin, Heidelberg, 2010. Springer-Verlag.

[108] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. *SIGPLAN conference of Programing Language Design and Implementation*, 42(6):235–244, June 2007.

[109] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 211–222, New York, NY, USA, 2007. ACM.

[110] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. Proving Optimizations Correct using Parameterized Program Equivalence. In *Proceedings of the 30th ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 327–337, 2009.

[111] Monica D Lam, Edward E Rothberg, and Michael E Wolf. The cache performance and optimizations of blocked algorithms. In *ACM SIGARCH Computer Architecture News*, volume 19, pages 63–74. ACM, 1991.

[112] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, September 1979.

[113] H. Le Verge, C. Mauras, and P. Quinton. The ALPHA language and its use for the design of systolic arrays. *Journal of VLSI Signal Processing*, 3(3):173–182, September 1991.

[114] Vincent Lefebvre and Paul Feautrier. Automatic storage management for parallel programs. *Parallel Computing*, 24:649–671, 1998.

[115] Allen Leung, Nicolas Vasilache, Benoît Meister, Muthu Manikandan Baskaran, David Wohlford, Cédric Bastoul, and Richard Lethin. A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction. In *3rd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU'10)*, Held with ASPLOS XVI, pages 51–61, Newport Beach, CA, March 2011. ACM.

[116] Eric Martin, Olivier Sentieys, Hélène Dubois, and Jean-Luc Philippe. Gaut: An architectural synthesis tool for dedicated signal processors. In *Design Automation Conference with EURO-VHDL'93 (EURO-DAC)*, 1993.

[117] C. Mauras. *ALPHA: un Langage Équationnel pour la Conception et la Programmation d'Architectures Parallèles Synchrones*. PhD thesis, L'Université de Rennes I, IRISA, Campus de Beaulieu, Rennes, France, December 1989.

[118] Sjoerd Meijer. *Transformations for Polyhedral Process Networks*. PhD thesis, Leiden Institute of Advanced Computer Science (LIACS), and Leiden Embedded Research Center, Faculty of Science, Leiden University, 2010.

[119] R. Metzger and Z. Wen. *Automatic Algorithm Recognition: A New Approach to Program Optimization*. MIT Press, 2000.

[120] M.Kandemir and A. Choudhary. Compiler-directed scratch pad memory hierarchy design and management. In *Proceedings of the 39th annual Design Automation Conference (DAC'02)*, pages 628–633, 2002.

[121] Dan I. Moldovan and Jose A. B. Fortes. Partitioning and mapping algorithms into fixed size systolic arrays. *IEEE transactions on computers*, (1):1–12, 1986.

[122] Aaftab Munshi. The opencl specification. In *Hot Chips 21 Symposium (HCS), 2009 IEEE*, pages 1–314. IEEE, 2009.

[123] George C. Necula. Translation Validation for an Optimizing Compiler. In *Proceedings of the 21st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 83–95, 2000.

[124] Josep M Perez, Rosa M Badia, and Jesus Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Cluster Computing, 2008 IEEE International Conference on*, pages 142–151. IEEE, 2008.

[125] Shlomit S. Pinter and Ron Y. Pinter. Program optimization and parallelization using idioms. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'91, pages 79–92, New York, NY, USA, 1991. ACM.

[126] Alexandru Plesco. *Program Transformations and Memory Architecture Optimizations for High-Level Synthesis of Hardware Accelerators*. PhD thesis, Ecole Normale Supérieure de Lyon, 2010.

[127] A. Pnueli, M. Siegel, and F. Singerman. Translation Validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 151–166, 1998.

[128] Sebastian Pop, Albert Cohen, Cédric Bastoul, Sylvain Girbal, Geogres-André Silber, and Nicolas Vasilache. GRAPHITE: Loop optimizations based on the polyhedral model for GCC. In *Proceedings of the 4th GCC Developper's Summit*, pages 1–18, Ottawa, Ontario, Unknown or Invalid Region, 2006.

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

[130] Louis-Noel Pouchet, Peng Zhang, P. Sadayappan, and Jason Cong. Polyhedral-based data reuse optimization for configurable computing. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '13, pages 29–38, New York, NY, USA, 2013. ACM.

[131] William Pugh and David Wonnacott. Going beyond integer programming with the omega test to eliminate false data dependences. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):204–211, 1995.

[132] Markus Püschel, José MF Moura, Bryan Singer, Jianxin Xiong, Jeremy Johnson, David Padua, Manuela Veloso, and Robert W Johnson. Spiral: A generator for platform-adapted libraries of signal processing algorithms. *The International Journal of High Performance Computing Applications*, 18(1):21–45, 2004.

[133] Fabien Quilleré and Sanjay Rajopadhye. Optimizing memory usage in the polyhedral model. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(5):773–815, 2000.

[134] Fabien Quilleré, Sanjay Rajopadhye, and Doran Wilde. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28(5):469–498, October 2000.

[135] Patrice Quinton and Vincent Van Dongen. The mapping of linear recurrence equations on regular arrays. *Journal of VLSI signal processing systems for signal, image and video technology*, 1(2):95–113, 1989.

[136] Sanjay V Rajopadhye, S Purushothaman, and Richard M Fujimoto. On synthesizing systolic arrays from recurrence equations with linear dependencies. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 488–503. Springer, 1986.

[137] Fabrice Rastello and Thierry Dauxois. Efficient tiling for an ODE discrete integration program: Redundant tasks instead of trapezoidal shaped-tiles. In *16th International Parallel and Distributed Processing Symposium (IPDPS 2002), 15-19 April 2002, Fort Lauderdale, FL, USA, CD-ROM/Abstracts Proceedings*, 2002.

[138] D. A. Reed, L. M. Adams, and M. L. Partick. Stencils and problem partitionings: Their influence on the performance of multiple processor systems. *IEEE Transactions on Computers*, 36(7):845–858, July 1987.

[139] Lakshminarayanan Renganarayanan, DaeGon Kim, Sanjay V. Rajopadhye, and Michelle Mills Strout. Parameterized loop tiling. *ACM Trans. Program. Lang. Syst.*, 34(1):3, 2012.

[140] Edwin Rijpkema, Ed F Deprettere, and Bart Kienhuis. Deriving process networks from nested loop algorithms. *Parallel Processing Letters*, 10(02n03):165–176, 2000.

[141] Robert Schreiber, Shail Aditya, B Ramakrishna Rau, Vinod Kathail, Scott Mahlke, Santosh Abraham, and Greg Snider. High-level synthesis of nonprogrammable hardware accelerators. In *Application-Specific Systems, Architectures, and Processors, 2000. Proceedings. IEEE International Conference on*, pages 113–124. IEEE, 2000.

[142] Robert Schreiber and Jack J Dongarra. Automatic blocking of nested loops. 1990.

[143] Weijia Shang and Jose A. B. Fortes. Independent partitioning of algorithms with uniform dependencies. *IEEE Transactions on Computers*, 41(2):190–206, 1992.

[144] K.C. Shashidhar, Maurice Bruynooghe, Francky Catthoor, and Gerda Janssens. Verification of source code transformations by program equivalence checking. In Rastislav Bodik, editor, *Compiler Construction*, volume 3443 of *Lecture Notes in Computer Science*, pages 221–236. Springer Berlin Heidelberg, 2005.

[145] Michelle Mills Strout, Alan LaMielle, Larry Carter, Jeanne Ferrante, Barbara Kreaseck, and Catherine Olschanowsky. An approach for code generation in the sparse polyhedral framework. *Parallel Computing*, 53:32–57, 2016.

[146] JM Tarela and MV Martinez. Region configurations for realizability of lattice piecewise-linear models. *Mathematical and Computer Modelling*, 30(11-12):17–27, 1999.

[147] Jürgen Teich, Alexandru Tanase, and Frank Hannig. Symbolic parallelization of loop programs for massively parallel processor arrays. In *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on*, pages 1–9. IEEE, 2013.

[148] Jürgen Teich, Alexandru Tanase, and Frank Hannig. Symbolic mapping of loop programs onto processor arrays. *Journal of Signal Processing Systems*, 77(1-2):31–59, 2014.

[149] Jürgen Teich and Lothar Thiele. Partitioning of processor arrays: A piecewise regular approach. *Integration, the VLSI journal*, 14(3):297–332, 1993.

[150] Jürgen Teich, Lothar Thiele, and Lee Z Zhang. Partitioning processor arrays under resource constraints. *Journal of VLSI signal processing systems for signal, image and video technology*, 17(1):5–20, 1997.

[151] William Frederick Thies. *Language and compiler support for stream programs*. PhD thesis, Massachusetts Institute of Technology, 2009.

[152] Konrad Trifunovic and Albert Cohen. Enabling more optimizations in GRAPHITE: ignoring memory-based dependences. In *Proceedings of the 8th GCC Developper's Summit*, Ottawa, Canada, October 2010.

[153] 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.

[154] 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.

[155] 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.

[156] Ugh: User-guided high-level synthesis. http://www-asim.lip6.fr/recherche/disydent/disydent_sect_12.html.

[157] 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.

[158] Anand Venkat, Mary Hall, and Michelle Strout. Loop and data transformations for sparse matrix code. In *ACM SIGPLAN Notices*, volume 50, pages 521–532. ACM, 2015.

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

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

[161] Sven Verdoolaege. Integer set coalescing. In *5th International Workshop on Polyhedral Compilation Techniques (IMPACT'15)*, 2015.

[162] Sven Verdoolaege, Albert Cohen, and Anna Beletska. Transitive closures of affine integer tuple relations and their overapproximations. In *International Static Analysis Symposium*, pages 216–232. Springer, 2011.

[163] Sven Verdoolaege, Gerda Janssens, and Maurice Bruynooghe. Equivalence checking of static affine programs using widening to handle recurrences. *ACM Transactions on Programming Languages and Systems*, 34(3):1–35, November 2012.

[164] Sven Verdoolaege, Hristo Nikolov, and Todor Stefanov. Pn: a tool for improved derivation of process networks. *EURASIP journal on Embedded Systems*, 2007(1):19–19, 2007.

[165] Xilinx Vivado HLS. `http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html`.

[166] Douglas B. West. *Introduction to Graph Theory*. Prentice Hall, 1999.

[167] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, SC '98, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.

[168] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[169] Linda M. Wills. *Automated Program Recognition by Graph Parsing*. PhD thesis, MIT, July 1992.

[170] Michael E Wolf and Monica S Lam. A data locality optimizing algorithm. In *ACM Sigplan Notices*, volume 26, pages 30–44. ACM, 1991.

[171] Michael Wolfe. Iteration space tiling for memory hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–361, Philadelphia, PA, USA, 1989. Society for Industrial and Applied Mathematics.

[172] Jingling Xue. *Loop Tiling for Parallelism*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[173] Asim Yarkhan, Jakub Kurzak, and Jack Dongarra. Quark users' guide. *Electrical Engineering and Computer Science, Innovative Computing Laboratory, University of Tennessee*, 2011.

[174] Tomofumi Yuki, Gautam Gupta, DaeGon Kim, Tanveer Pathan, and Sanjay V. Rajopadhye. Alphaz: A system for design space exploration in the polyhedral model. In *Languages and Compilers for Parallel Computing, 25th International Workshop, LCPC 2012*, pages 17–31, September 2012.

[175] Chen Zhang, Di Wu, Jiayu Sun, Guangyu Sun, Guojie Luo, and Jason Cong. Energy-efficient cnn implementation on a deeply pipelined fpga cluster. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, pages 326–331. ACM, 2016.

[176] Xing Zhou, Jean-Pierre Giacalone, María Jesús Garzarán, Robert H Kuhn, Yang Ni, and David Padua. Hierarchical overlapped tiling. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 207–218. ACM, 2012.

[177] 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.

[178] Lenore Zuck, Amir Pnueli, Benjamin Goldberg, Clark Barrett, Yi Fang, and Ying Hu. Translation and Run-Time Validation of Loop Transformations. *Formal Methods in System Design*, 27(3):335–360, November 2005.

Résumé

Depuis la fin de la loi de Dennard, les performances des superordinateurs sont limitées par leur consommation énergétique. Les accélérateurs matériels ont été introduits pour améliorer les performances sous un budget énergétique limité. Ce faisant, les superordinateurs sont devenus des plateformes hétérogènes où des processeurs multicoeurs cohabitent avec des circuits reconfigurables (FPGA, CGRA) et des accélérateurs graphiques (GPU). Programmer de tels ordinateurs, et notamment configurer les circuits FPGA est un défi qui ne peut être relevé qu'avec des modèles et des algorithmes de compilation adaptés. C'est tout l'enjeu de nos travaux de recherche au cours la dernière décennie, dont ce document présente une synthèse.

Un premier chapitre décrit nos contributions au *tuilage de boucles*, une transformation fondamentale pour la parallélisation automatique, qui découpe le calcul en sous-calculs atomiques appelés *tuiles*. Nous reformulons le tuilage de boucles dans le modèle polyédrique pour permettre n'importe tuile polytopique dont la taille dépend d'un facteur homothétique (tuilage monoparamétrique), et nous décrivons une transformation de tuilage pour des programmes avec des *réductions* – une accumulation selon un opérateur associative et commutatif. Nos résultats ouvrent la voie à des *transformations de programme sémantiques*; qui ne préservent pas le calcul, mais produisent un programme équivalent.

Un second chapitre décrit nos contributions à la *reconnaissance d'algorithmes*. Une optimisation de compilateur ne remplacera jamais un bon algorithme, d'où l'idée de reconnaître les instances d'un algorithme dans un programme et de les substituer par un appel vers une bibliothèque haute-performance, chaque fois que c'est possible et utile. Dans notre thèse, nous avons traité la reconnaissance de *templates* – des fonctions avec des variables d'ordre 1 – dans un programme et son application à l'optimisation de programes. Nous proposons une approche complémentaire qui s'appuie sur notre tuilage monoparamétrique complété par une transformation pour tuiler les réductions. Ceci automatise le *tuilage sémantique*, une nouvelle transformation sémantique qui augmente le grain des opérateurs (scalaire → matrice).

Un troisième chapitre présente nos contributions à la *synthèse des communications avec une mémoire off-chip* dans le contexte de la synthèse de circuits haut-niveau (High-Level Synthesis, HLS). Nous proposons un modèle d'exécution basé sur le tuilage de boucles, une architecture pipelinée et un algorithme de compilation source-à-source qui, connecté à l'outil de HLS C2H d'Altera, produit une configuration de circuit FPGA qui réalise un volume minimal de transferts de données. Notre algorithme est optimal – les données sont chargées le plus tard possible et stockées le plus tôt possible, avec une réutilisation maximale et sans redondances.

Enfin, un quatrième chapitre présente nos contributions pour construire un *modèle de compilation complet pour la synthèse de circuits haut-niveau.* Nous présentons les réseaux de processus DPN (Data-aware Process Networks), une représentation intermédiaire dataflow qui s'appuie sur les idées développées au chapitre 3 pour expliciter les transferts de données entre le circuit et la mémoire *off-chip*. Nous proposons une suite d'algorithmes pour compiler un DPN à partir d'un programme séquentiel et pour synthétiser un DPN en circuit. En particulier, nous présentons nos algorithmes pour compiler le contrôle, les canaux et les synchronisations d'un DPN. Ces résultats sont utilisés dans le compilateur de production de la start-up XtremLogic.

Summary

Since the end of Dennard scaling, power efficiency is the limiting factor for large-scale computing. Hardware accelerators such as reconfigurable circuits (FPGA, CGRA) or Graphics Processing Units (GPUs) were introduced to improve the performance under a limited energy budget, resulting into complex heterogeneous platforms. This document presents a synthetic description of our research activities over the last decade on compilers for high-performance computing and high-level synthesis of circuits (HLS) for FPGA accelerators.

A first chapter describes our contributions to *loop tiling*, a key program transformation for automatic parallelization which splits the computation atomic blocks called *tiles*. We rephrase loop tiling in the polyhedral model to enable any polyhedral tile shape whose size depends on a single parameter (monoparametric tiling), and we present a tiling transformation for programs with *reductions* – accumulations w.r.t. an associative/commutative operator. Our results open the way for *semantic program transformations*; program transformations which does not preserve the computation but still lead to an equivalent program.

A second chapter describes our contributions to *algorithm recognition*. A compiler optimization will never replace a good algorithm, hence the idea to recognize algorithm instances in a program and to substitute them by a call to a performance library. In our PhD thesis, we have addressed the recognition of templates – functions with first-order variables – into programs and its application to program optimization. We propose a complementary algorithm recognition framework which leverages our monoparametric tiling and our reduction tiling transformations. This automates *semantic tiling*, a new semantic program transformation which increases the grain of operators (scalar → matrix).

A third chapter presents our contributions to the *synthesis of communications with an off-chip memory* in the context of high-level circuit synthesis (HLS). We propose an execution model based on loop tiling, a pipelined architecture and a source-level compilation algorithm which, connected to the C2H HLS tool from Altera, ends up to a FPGA configuration with minimized data transfers. Our compilation algorithm is optimal – the data are loaded as late as possible and stored as soon as possible with a maximal reuse.

A fourth chapter presents our contributions towards a *complete compilation model for high-level circuit synthesis* in the polyhedral model. We present the Data-aware Process Networks (DPN), a dataflow intermediate representation which leverages the ideas developed in chapter 3 to explicit the data transfers with an off-chip memory. We propose an algorithm to compile a DPN from a sequential program, and we present our contribution to the synthesis of DPN to a circuit. In particular, we present our algorithms to compile the control, the channels and the synchronizations of a DPN. These results are used in the production compiler of the Xtremlogic start-up.