Polyhedral Bubble Insertion: A Method to Improve Nested Loop Pipelining for High-Level Synthesis

Antoine Morvan, Steven Derrien, and Patrice Quinton

***Abstract*—High-Level Synthesis (HLS) allows hardware to be directly produced from behavioral description in C/C++, thus accelerating the design process. Loop pipelining is a key transformation of HLS, as it improves the throughput of the design at the price of a small hardware overhead. However, for small loops, its use often results in a poor hardware utilization due to the pipeline latency overhead. Overlapping the iterations of the whole loop nest instead of the innermost loop only is a way to overcome this difficulty, but current available techniques are restricted to perfectly nested loops with constant bounds, involving uniform dependencies only. Using the polyhedral model, we extend the applicability of the nested loop pipelining transformation by proposing a new legality check and a new loop correction technique, called polyhedral bubble insertion. This method was implemented in a source-to-source compiler targeting High-Level Synthesis, and results on benchmark kernels shows that polyhedral bubble insertion is effective in practice on a much larger class of loop nests.**

***Index Terms*—High-Level Synthesis, Source-To-Source Transformations, Nested Loop Pipelining, Polyhedral Model, Loop Coalescing**

I. Introduction

REDUCING hardware design time is more than ever a priority for chip vendors. To this end, designers are shifting away from register transfer level descriptions in favor of design flows that operate at higher levels of abstractions. High-Level Synthesis (HLS) addresses this need by enabling the hardware components to be directly designed from behavioral specifications in C or C++. There now exist several mature and robust commercial tools [1], [2] that are used for production by major chip makers.

However, designs generated by HLS are far from delivering performance comparable to those produced by experts. This is mainly due to the difficulty for HLS to extract from the source code the information needed to enable some loop transformations. This lack of performance can be overcome by letting the designer either manually drive the HLS tool, or manually expose appropriate structures (data and/or algorithms) directly in the source code. It is our belief that such processes can be automated within a source-to-source optimizing compiler.

The goal of the work reported here is to improve the applicability and the efficiency of *nested loop pipelining* – also known as *nested software pipelining*, – in C-to-hardware translation tools. The contributions of this research are as follows:

- A fast approximation along with an accurate legality check is described. Given a pipeline latency, it indicates whether pipelining a loop nest enforces the data-dependencies of a program.
- When the legality check fails, a loop correction algorithm is proposed. It consists in adding, at compile time, so-called *wait-states* instructions, also known as *pipeline bubbles*, to make sure that the aforementioned pipelining becomes legal.
- In order to make the loop nest amenable to pipelining, the loop nest is flattened at the source level using an automatic *loop coalescing* transformation.

These techniques leverage on the well-known *polyhedral model* [3], [4], [5]. Using the high-level representation of loops of this model, these methods are applicable to a much wider class of programs – namely imperfectly nested loops with affine bounds and index functions – than previously published works [6], [7], [8], [9].

Thanks to tools available in the polyhedral model community, these new methods were implemented within a source-to-source compiler. Their applicability was validated on a set of representative kernels, and the trade-offs between the performance improvements provided by the full nested loop pipelining transformation on the one hand, and the area overhead induced by guards that are added to the control code on the other hand are discussed.

This article is organized as follows. Section II provides an in-depth description of the problem addressed by this research, and mentions existing approaches. Section III summarizes the principles of program analysis and transformations in the polyhedral framework that are needed to understand this work. The new pipeline legality analysis and the loop correction technique are presented in sections IV and V. Section VI describes their implementation and provides a quantitative and qualitative analysis of their performance. In section VII, relevant related work is presented, and the novelty of this contribution is highlighted. Conclusion and future work are described in section VIII.

II. Motivations

The goal of this section is to present and motivate the problem addressed in this research, that is *nested loop pipelining*. To help the reader understand the contributions of this work, a running toy loop-nest example shown in Figure 1 is used throughout the remaining of this article. This example is a simplified excerpt from the QR decomposition algorithm. It consists in a double nested loop operating on a triangular *iteration domain* – the iteration domain of a loop is the set of values taken by its loop indices.

A. Loop pipelining in HLS

Loop pipelining consists in executing the body of a loop using several *pipeline stages*. The effectiveness of this trans-

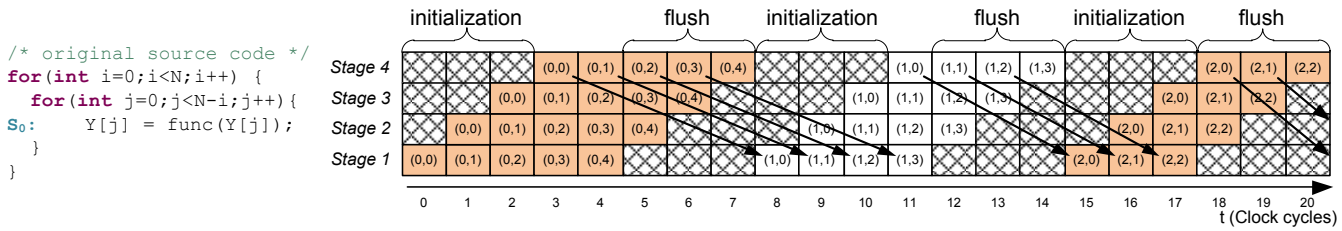
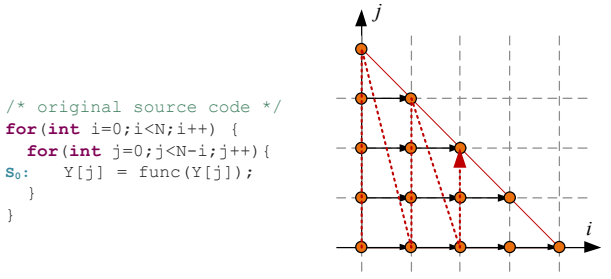


Fig. 2: Representation of the pipelined execution of the simplified QR decomposition loop of Fig. 1, for size parameter $N = 5$, initiation interval $\Phi = 1$ and pipeline latency $\Delta = 4$. Arrows represent dependencies between operations.



(a) Simplified QR loop (b) Graphical representation of (a)

Fig. 1: A simplified QR decomposition loop (a) and the representation of its iteration domain and of the data-dependencies of the Y array (solid arrows) for $N = 5$ (b). The red dashed arrow shows the execution order of the loop.

formation comes from the fact that several loop iterations can be executed simultaneously by the different stages. To produce an equivalent loop, one must however make sure that the executions of successive iterations are independent. *Loop pipelining* is characterized by two important parameters:

- The *initiation interval* (denoted by Φ in the following) is the number of clock cycles separating the execution of two successive loop iterations.
- The *pipeline latency* (denoted by Δ) gives the number of clock cycles required to completely execute one iteration of the loop. The latency usually corresponds to the number of stages of the pipeline.

In the example of Figure 1, the reader can observe that the inner loop (along the j index) exhibits no data-dependencies between calculations done at different iterations (also called *loop carried dependencies*). As a consequence, one can pipeline the execution of this loop by overlapping the execution of several iterations of its inner loop.

As an illustration, Figure 2 depicts the pipelined execution of the example of Figure 1 with an initiation interval $\Phi = 1$ and a latency $\Delta = 4$. In practice the value of the initiation interval Φ is constrained by two factors:

- the presence of loop carried dependencies, which prevents loop iterations to be completely overlapped;
- resource constraints on the available hardware since for a complete pipelined execution, each operation executed in the loop has to be mapped on its own hardware functional unit.

In this example, notice that between two iterations of the external i loop, there is a *flush phase* which is needed to prevent dependencies between these iterations to be violated. We shall see in this paper how this flush phase can be avoided, leading to more efficient pipelined implementations.

Because it helps maximize the computation throughput and because it improves hardware utilization, loop pipelining is a key transformation of high-level synthesis. Besides, as designers generally seek to get the best performance from their implementation, fully pipelining the loop, that is initiating a new inner loop iteration every cycle by choosing $\Phi = 1$, is a very common practice.

However, the performance improvements obtained through pipelining are often hindered by the fact that these tools rely on very imprecise data-dependency analysis algorithms and hence they may fail to detect when a pipelined execution is possible, especially when the inner loop involves complex memory access patterns. To help designers cope with these limitations, most tools offer the ability to bypass part of the dependency analysis using compiler directives (generally in the form of `#pragma`). These directives force the tool to ignore user-specified memory references in its dependency analysis. Of course, this possibility comes at the risk of generating an illegal pipeline and then an incorrect circuit, and hence it puts on the designer the burden to decide whether the transformation is legal or not.

B. The Pipeline Latency Overhead

For loops with large iteration counts – *loop iteration count* is the number of iterations executed by a loop –, the impact of the pipeline latency on performance can be neglected, and the hardware is then almost 100% utilized. However, whenever the iteration count of the loop becomes comparable to its latency Δ, one may observe very significant performance degradation, as the pipeline flush phases dominate the execution time. This is the case in the example of Figure 2. For values $N = 5$ and $\Delta = 4$, the hardware utilization rate is only 50%.

On this example, experienced designers would have certainly reached a hardware utilization close to 100% using a handcrafted schedule in which the execution of successive iterations of the i loop would have been overlapped.

C. Nested loop pipelining

Initially proposed by Doshi et al. [6], *nested loop pipelining* is a means of improving the pipelined execution of a loop. It

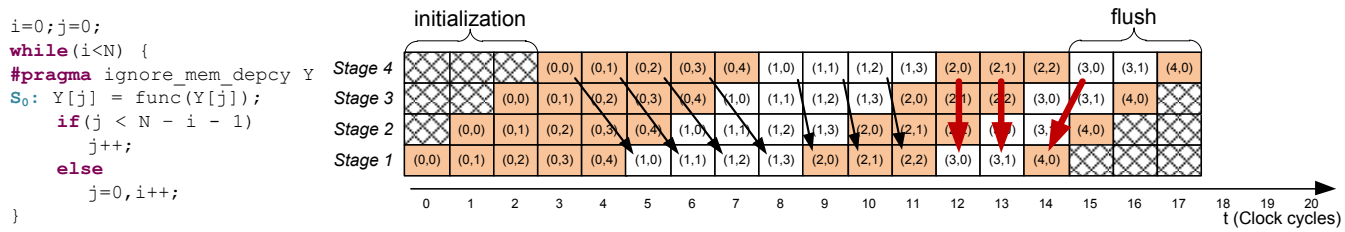


Fig. 3: Illustration of an illegal nested loop pipelining. The example shown is a coalesced version of the simplified QR decomposition loop, for $N = 5$, $\Phi = 1$ and $\Delta = 4$. Thick red arrows show violated dependencies.

is the method considered in this article, and as done in other works [10], it was chosen here to apply it in two steps:

- first rewrite the loop nest to be pipelined so that it becomes a single level loop. This is called *loop coalescing*;
- then pipeline the coalesced loop.

The goal of loop coalescing (also known as *loop flattening*) is to transform the control of the loop so that a single loop scans the original loop nest domain. Different versions of this transformation are discussed in section V-C. Loop coalescing can be done independently of the pipeline transformation.

It is worth noticing that until now, *nested loop pipelining* was only studied for perfectly nested loop with constant bounds and uniform dependencies – a very restrictive subset of loop nests –, or with relatively imprecise dependency information, and this significantly restricts its applicability and its efficiency. While these restrictions may seem over-precautious, it happens that implementing nested loop pipelining (and more particularly enforcing its correctness) is far from trivial and requires a lot of attention.

As an example, Figure 3 shows a coalesced version of the loop nest of Figure 1. Here, because the array accesses in the coalesced version are difficult to analyze (they do not depend on loop indices as in Figure 1), one would be tempted to bypass some of the dependency analysis through a compiler directive (`#pragma ignore_mem_depcy Y`) to force loop pipelining, as explained in subsection II-A. Without such a directive, the conservative dependence analysis would forbid pipelining.

While at first glance this scheduling seems correct, it appears that some Read after Write dependencies are violated when $i \geq 3$, as shown in Figure 3. Indeed, the dependency between two successive i iterations prevents the end of the inner loop pipeline to be overlapped with the beginning of the next one. For example, the memory read operation on `Y[0]` of $(i = 3, j = 0)$ scheduled at $t = 12$ happens before `Y[0]` is updated by the write operation of $(i = 2, j = 0)$ also scheduled at $t = 12$ on the last stage.

As an illustration of this difficulty, among the numerous commercial and academic C-to-hardware tools that the authors have evaluated, only one of them actually provides the ability to perform automatic nested loop pipelining. (This tool is called the *reference HLS tool* in the following, RHLS for short.) However, its implementation suffers from severe flaws and generates illegal schedules whenever the domain has non-constant loop bounds. From what the authors understand, even without directives to ignore data dependencies, RHLS fails for the very same reasons as depicted in Figure 3, that is its analysis assumes that the dependencies carried by the outer loop over the `Y` array are never violated.

D. Contributions of this work

In the following sections, a formalization of the conditions under which nested loop pipelining is legal w.r.t data dependencies is provided, in the case of imperfectly nested loops with affine dependencies (so called SCoPs [3]), where exact (i.e. iteration-wise) data-dependency information is available.

In addition to this legality check, a technique to *correct* an a priori illegal nested pipeline schedule by inserting *wait states* in the coalesced loop is proposed, in order to derive the most efficient *legal* pipelined schedule. These wait states correspond to properly inserted *bubbles* in the pipeline, hence the name *polyhedral bubble insertion* of this new method.

Finally, to enable experimentation and to remain as vendor independent as possible, an implementation of the polyhedral bubble insertion in the context of a source-to-source compiler is described. This implementation can be incorporated as a preprocessing tool to be used ahead of third party HLS tools.

III. Background

In order to perform a precise dependence analysis and if needed, to realize a cycle-accurate schedule correction, an accurate representation of loops is necessary. In this respect, the polyhedral model is a robust mathematical framework. It also comes with a set of techniques to analyze and transform loops and to regenerate source code.

Figure 4 illustrates a standard source-to-source flow within the polyhedral model. This section details each one of these steps.

A. Static Control Parts Detection and Extraction

The polyhedral model is a representation of a subset of programs called *Static Control Parts* (SCoPs), or alternatively *Affine Control Loops* (ACLs). Such programs are composed only of loops and conditional control structures, and the only allowed statements are array assignments of arbitrary expressions with array reads. (Scalar variables are viewed as zero-dimensional arrays.) The loop bounds, the conditions

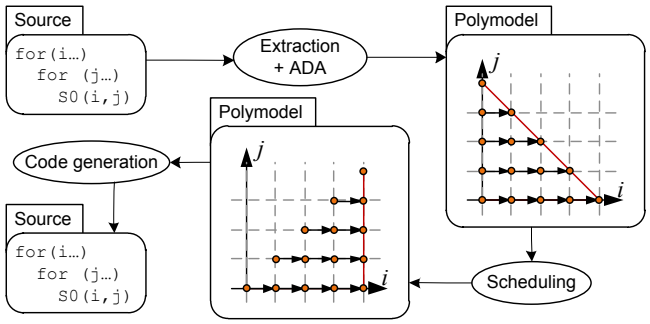


Fig. 4: Overview of a classical source-to-source flow within the polyhedral model. After extracting static control parts (SCoPs) from the source code, the array data-flow analysis (ADA) produces the polyhedral representation of the SCoP. Then scheduling transforms the domain and the execution order of the loop. Finally, code generation produces a loop nest that scans the new domain according to the new execution order.

and array subscripts have to be affine expressions of loop indexes and parameters. Extracting SCoPs is the first step in an automatic polyhedral flow, as depicted in Figure 4.

Each statement S in a SCoP, surrounded by n loops, has an associated domain which represents the set of values the indices of these loops can take. Let $\mathbb{Z}^n$ denote the set of integral coordinate vectors of dimension n. Loop indices are represented by *iteration vectors* of $\mathbb{Z}^n$. The domain of a statement S is called its *iteration domain*, and is denoted by $\mathcal{D}_S \subseteq \mathbb{Z}^n$. In SCoPs, $\mathcal{D}_S$ is defined by a set of affine constraints, i.e. the set of loop bounds and conditionals on its indexes, and it is therefore a *parameterized polyhedron*.

In what follows, we call *operation* a particular iteration of a statement, i.e., a statement with a given iteration vector. Figure 1 shows the graphical representation of such a domain, where each full circle represents an operation. The domain constraints for the only statement S_0 of the loop of Figure 1 are defined by:

$$\mathcal{D}_{S_0} = \{i, j | 0 \leq i < N \wedge 0 \leq j < N - i\} \quad .$$

We shall denote by $S(\vec{v})$ the operation that corresponds to statement S and iteration vector $\vec{v}$.

The polyhedral model is limited to the aforementioned class of programs. This class can be however extended to while loops and data-dependent bounds and indexes, at the price of a loss of accuracy in the dependence analysis [11], [12].

The detection of SCoPs is done by a mere syntactic analysis of the compiler front-end.

B. Array Data-flow Analysis (ADA)

The strength of the polyhedral model is its capacity to allow an iteration-wise dependency analysis on arrays [13] to be performed. The goal of dependency analysis is to answer questions such as *"Q: what operation produced the value being read by the currently executing operation?"* For example, in the program of Figure 1, what operation wrote the last value of the right-hand side reference `Y[j]`? Answering such a question is the second part of the first step in the polyhedral flow presented in Figure 4.

Iterations of *one* statement in a loop nest can be ordered by the lexicographic order of their iteration vectors. Consider two iteration vectors $\vec{a} \in \mathbb{Z}^m$ and $\vec{b} \in \mathbb{Z}^n$. Denote by $\vec{a}_{[q]}$ the q-th component of $\vec{a}$, and by $\vec{a}_{[0..q]}$ the left-most sub-vector $(a_0, \ldots, a_q)$ of $\vec{a}$. Then $\vec{a}$ is said to be lexicographically greater than $\vec{b}$, noted $\vec{a} \gg \vec{b}$, iff either $(\vec{a}_{[0]} > \vec{b}_{[0]})$ or there exists a value $q \in [1..min(m,n)]$ such that $(\vec{a}_{[0..q-1]} = \vec{b}_{[0..q-1]} \wedge \vec{a}_{[q]} > \vec{b}_{[q]})$.

Notice that iterations of several statements in a loop nest can be ordered by combining the lexicographic order of their iteration with their textual order in the loop. This combination defines the *precedence order*, noted $\succ$. When considering sequential loop nests, the precedence order is a total order. To simplify matter, iteration vectors can be extended using the textual rank of the statements in the loop body (see Bastoul [3]), so that the precedence order reduces to the lexicographic order of the iteration vectors, and consequently, this will be assumed in the remaining of this paper. Also in order to simplify our presentation, it will be assumed, without loss of generality, that a statement in a SCoP has at most one array write reference and one array read reference. With this assumption, read array or write array references are uniquely identified by the iteration vector of their operations.

The precedence order allows an exact answer to be given to question *Q*: *"The operation that last modified an array reference in the currently executing operation is just the latest write in the same array reference according to the precedence order."* In the example of Figure 1, the operation that modified the right-hand side reference `Y[j]` in operation $S_0(i,j)$ is the same statement of the loop, when it was executed at previous iteration $S_0(i-1,j)$.

A dependency is represented by a function d that associates to each read the operation that produced the value being read. In our example, $d(S_0(i,j)) = S_0(i-1,j)$. Another way of representing this is to use a relation notation *à la Omega* [14] – functions can be considered as a special case of binary relations. This is also the standard notation of the ISL library [15] that we shall use extensively in this paper:

$$d = \left\{ (i,j) \rightarrow (i',j') \;\middle|\; \begin{array}{c} (i,j) \in \mathcal{D}_{S_0} \wedge (i',j') \in \mathcal{D}_{S_0} \\ \wedge\, (i',j') = (i-1,j) \end{array} \right\} \quad (1)$$

Since they represent all the instances of the dependency in a compact polyhedral relation, dependency functions are called *polyhedral reduced dependencies*. The graph representing all the dependencies for one SCoP is called the *polyhedral reduced dependency graph* (PRDG).

In the remaining of this paper, $sink(d) \subseteq \mathcal{D}$ denotes the domain of the dependence function d, that is to say, the set of array reads on which d can be applied, and $src(d) \subseteq \mathcal{D}$ denotes the range of function d, i.e. the set of array writes that it leads to.

In summary, the second part of the first step of the polyhedral model flow, ADA, is to build the PRDG of a SCoP, which is much more involved as the SCoP detection and extraction (see [13] for details on ADA.)

C. Scheduling

In the polyhedral model, the precedence order is known exactly. Therefore, transformations of the loop execution order, also known as *scheduling* transformations, can be constrained to enforce data-flow dependencies. This is the second step in the polyhedral flow of Figure 4.

A one-dimensional, integral schedule σ enforces a dependency $d : \{\vec{a} \rightarrow \vec{b}\}$ if $\sigma(\vec{a}) > \sigma(\vec{b})$. More generally, a multi-dimensional, integral schedule σ enforces a dependency if $\sigma(\vec{a}) \succ \sigma(\vec{b})$.

To enforce all dependencies, a schedule must meet a set of affine constraints. The intersection of these constraints for all the dependencies in the PRDG gives a polyhedron. Not only some properties can be checked on this polyhedron, for example the legality of a given transformation, but also one can automatically compute the space of all possible transformations, in order to find the "best" one according to some criterion. A considerable amount of work has been done on this topic, and the reader is referred to Feautrier [16] and Pouchet et al. [5] for more details.

As far as loop pipelining is concerned, scheduling is not a central transformation, as one may assume that pipelining is applied to the sequential version of the loop without changing its schedule. However, it is worth mentioning this step, as in some cases, it may be useful to change the schedule of a loop in order to reach a better pipelining transformation.

D. Code Generation

The last step of source-to-source transformation within the polyhedral model is to re-generate a sequential code that scans the new domain, as shown in Figure 4. Two approaches to solve this problem dominate in the literature.

1) Loop Nests Generation: The first one was developed by Quillere and al. [17] and later extended and implemented by Bastoul in the ClooG software [3]. ClooG allows regenerated loops to be guardless, thus avoiding useless iterations at the price of an increase in code size. With the same goal, the code generator in the Omega project [14] tries to regenerate guardless loops, and provides options to find a trade-off between code size and guards.

2) Finite State Machine Generation: The second approach, developed by Boulet et al. [18] aims at generating code without loops. The principle is to determine during one iteration the value of the next iteration vector, until the entire iteration domain has been visited. Rather than generating nested loops, this instead amounts to derive a finite state machine that scans the iteration space.

To do so Boulet et al. introduce a $next_{\mathcal{D}}$ function which, given an iteration $\vec{x} \in \mathcal{D}$, provides its immediate successor $next_{\mathcal{D}}(\vec{x})$ in $\mathcal{D}$ according to the lexicographical order. The construction of this function is detailed in section IV-C. Since this second approach behaves like a finite-state machine, it is believed to be more suitable for hardware implementation [19], though there are still very few quantitative evidences to back-up this claim. We discuss one main aspect of this approach in section V-C, that is its efficiency for coalescing loops.

IV. Legality Check

This section considers the problem of checking that a given nested loop pipelining transformation does not violate dependencies. Section IV-A describes the pipeline model and presents the legality condition. Section IV-B shows how this condition can be checked by computing the reuse distance of dependencies. Another method, based on the computation of the successors of the iteration points is described in section IV-C. Section IV-D explains how to build the set of violated dependencies. Finally, the complete algorithm is described in IV-E.

A. Pipeline Model and Legality Condition

Let Δ be the number of stages of the pipeline, i.e., its latency. In our pipeline model, we consider that all the reads are being executed during the first stage of the pipeline, and all the writes during its last stage (These assumptions are not essential to our method, but they simplify the explanations.)

Let us call *reuse distance* of a dependence the number of points in the iteration domain between a source iteration $\vec{x}$ and its sink $\vec{y}$. Since the execution of the loop follows the lexicographic order on the iteration domain, one can observe that the executions of two successive iteration points are separated by one cycle. Therefore, the number of cycles that separate the execution of the source iteration $\vec{x}$ and that of the sink iteration $\vec{y}$ is equal to their reuse distance. On the other hand, the value produced by the execution of iteration $\vec{x}$ is available Δ cycles after its beginning according to the pipeline model. Therefore, the nested loop pipelining does not violate data dependencies provided the distance (in number of iteration points) between the production of the value (at iteration $\vec{x}$, the source) and its use (at iteration $\vec{y}$, the sink) is equal to or larger than Δ.

This condition is trivially enforced in one particular case, that is when the loop nest to be pipelined does not carry dependencies, i.e. when the loops are parallel. This happens, for example, if the dependencies in the loop nest are carried only by the outermost loop. One can then pipeline the $n-1$ inner loops, and if the pipeline is flushed at each iteration of the outermost loop, the latency does not violate dependencies.

To apply the nested loop pipelining transformation on loops that carry dependencies, or to pipeline a whole loop nest (as shown in the example of Figure 3 for example), a deeper analysis is required, and this is just what our legality condition provides.

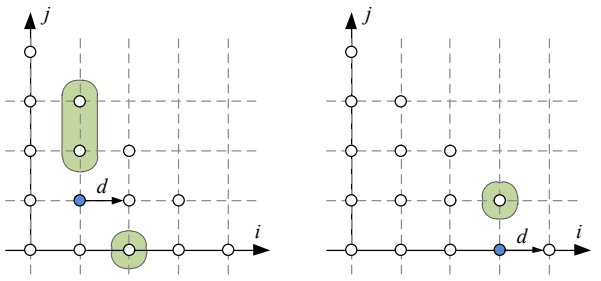
B. Checking the Legality by Estimating the Reuse Distance

Computing the reuse distance between a source and a sink amounts to count the minimum number of iterations that separate the source and the sink in the iteration domain.

Consider the relation R given by

$$R = \{\vec{x} \rightarrow \vec{z} \mid \vec{x} \in src(d) \wedge \vec{x} \prec \vec{z} \prec d^{-1}(\vec{x}) \wedge \vec{z} \in \mathcal{D}\} \quad (2)$$

For a given source point $\vec{x}$, R gives all the iterations points $\vec{z}$ which are lexicographically between $\vec{x}$ and one element of the set $d^{-1}(\vec{x})$, that is, the set of all possible sinks of $\vec{x}$.



(a) R when $(i,j) = (1,1)$ and $N = 5$. (b) R when $(i,j) = (N-1,0)$ (for all $N > 1$).

Fig. 5: Representation of the range of the relation R (enclosed points) over the domain of the example in Figure 1 given the source operation of d.

R is a parameterized polyhedron which can be computed using the ISL software [15]. Given R, the number of points between a source $\vec{x}$ and its closest sink is a parametric multivariate pseudo-polynomial P, which depends on the parameters of the domain and on $\vec{x}$ (see [20]). A closed form of P can be computed using the Barvinok library [21]. Finally, one can compute the minimum Bernstein expansion [22] of P, which gives a lower bound of this expression.

If the resulting bound is greater than $\Delta - 1$, then applying the pipeline is legal.

Example: ■ Starting from the dependency d defined in Equ. (1), we can compute the inverse of d as follows :

$$d^{-1} = \left\{ (i,j) \to (i',j') \;\middle|\; \begin{array}{c} (i,j) \in \mathcal{D}_{S_0} \wedge (i',j') \in \mathcal{D}_{S_0} \\ \wedge\, (i',j') = (i+1,j) \end{array} \right\}$$

Using Equ. (2):

$$R = \left\{ (i,j) \to (i',j') \;\middle|\; \begin{array}{l} (i,j) \in src(d) \wedge (i',j') \in \mathcal{D}_{S_0} \\ \wedge\, (i,j) \prec (i',j') \prec d^{-1}(i,j) \end{array} \right\}$$

After resorting to the simplification of this polyhedral relation thanks to a polyhedral library [15], one obtains:

$$R = \left\{ (i,j) \to (i',j') \;\middle|\; \begin{array}{l} (0 \le i \wedge i' = i \wedge \\ \quad 0 \le j < j' < N - i) \vee \\ (0 \le i \wedge i' = i+1 \wedge \\ \quad 0 \le j' < j < N - i - 1) \end{array} \right\}$$

When $(i,j) = (1,1)$ and $N = 5$, the range of R represents the highlighted set in Figure 5a, that is $\{(i' = 1 \wedge 2 \le j' \le 3) \vee (i' = 2 \wedge j' = 0)\}$. The number of integer points between a source and a sink in $\mathcal{D}_{S_0}$, according to the dependency d, is expressed as follow :

$$P = card(R) = \{(i,j) \to (N - i - 1)\}$$

that is 3 when $(i,j) = (1,1)$ and $N = 5$.

Using the Bernstein expansion, one can compute the minimum value of P over $\mathcal{D}_{S_0}$, for all the possible values of N. As shown in Figure 5b, the minimum is 1, and it is reached for $(i,j) = (N-1,0)$. Thus applying nested loop pipelining with $\Delta = 4$ on this loop nest is not ensured to be legal. ■

The above method is fast, but it does not always give a good estimate of the lower bound. Besides, the result of this analysis does not provide a means to fix the loop, if the legality condition is not met.

C. Constructing the $next^{\Delta}_{\mathcal{D}}(\vec{x})$ function

To avoid the drawbacks of the previous method, one can construct a function $next^{\Delta}_{\mathcal{D}}(\vec{x})$ that gives for a given iteration vector $\vec{x}$ its successor Δ iterations away in domain $\mathcal{D}$. Then by checking that all the sink iteration vectors $\vec{y} \in d^{-1}(\vec{x})$ verify $\vec{y} \succeq next^{\Delta}_{\mathcal{D}}(\vec{x})$, one is sure that the value produced at iteration $\vec{x}$ is used at the earliest Δ iterations later.

The $next^{\Delta}_{\mathcal{D}}(\vec{x})$ function can be derived by leveraging on the $next_{\mathcal{D}}$ function introduced by Boulet et al. [18] in their code generation technique (see Section III-D2). By convention, let $next_{\mathcal{D}}(\vec{x}) = \perp$ when an iteration vector $\vec{x}$ has no successor inside $\mathcal{D}$, and let $next_{\mathcal{D}}(\perp) = \perp$.

Algorithm 1 recalls Boulet et al.'s method, where $dim(\mathcal{D})$ is the number of dimensions of $\mathcal{D}$, $lexmin(succ_i)$ (given by ISL) provides the lexicographic minimum of the relation $succ_i$, and $domain(next_{\mathcal{D}})$ denotes the iteration domain on which $next_{\mathcal{D}}$ is applicable. As expressed here, this algorithm computes the $next_{\mathcal{D}}$ function only on the $depth$ innermost loops, and this feature will be used later on to avoid possible useless computations.

Algorithm 1 Builds the $next_{\mathcal{D}}$ and $next^{\Delta}_{\mathcal{D}}$ function

Require: $1 \le depth \le dim(\mathcal{D})$
  procedure NEXTBOULET($\mathcal{D}$, $depth$)
    $n \leftarrow dim(\mathcal{D})$
    $next_{\mathcal{D}} \leftarrow \emptyset$
    $\mathcal{R} \leftarrow \mathcal{D}$
    for $p = n \to (n - depth)$ **do**
      $lexGT_p \leftarrow \{\vec{x} \to \vec{y} \mid \vec{x}_{[0..p-1]} = \vec{y}_{[0..p-1]} \wedge \vec{x}_{[p]} > \vec{y}_{[p]}\}$
      $succ_p \leftarrow \{\vec{x} \to \vec{y} \mid \vec{x} \in \mathcal{R} \wedge \vec{y} \in \mathcal{D}\} \cap lexGT_p$
      $next_{\mathcal{D}} \leftarrow next_{\mathcal{D}} \cup lexmin(succ_p)$
      $\mathcal{R} \leftarrow \mathcal{D} - domain(next_{\mathcal{D}})$
    end for
    return $next_{\mathcal{D}}$
  end procedure
Require: $1 \le depth \le dim(\mathcal{D})$
  procedure NEXTPOWER($\mathcal{D}$, Δ, $depth$)
    $next_{\mathcal{D}} \leftarrow nextBoulet(\mathcal{D}, depth)$
    return $\overbrace{next_{\mathcal{D}} \bullet next_{\mathcal{D}} \bullet \ldots \bullet next_{\mathcal{D}}}^{\Delta}$
  end procedure

Algorithm 1 can be best explained by following its operation on the example shown in Figure 6. It starts by generating the function giving the immediate successor on the innermost loop at depth p (D_2 and $p = 2$ in the example of Figure 6). When there is no successor on that innermost dimension, that is when the iterators are along the upper bound of the iteration domain, the algorithm looks for a successor on the next outer dimension, at depth $p - 1$ (D_1 and $p = 1$ on Figure 6). This procedure is then repeated until all dimensions of the domain have been scanned by the analysis, or when dimensions $p - depth$ is reached. At termination, the remaining point is the lexicographic maximum of the domain, and its successor is $\perp$ ($D_\perp$ and $p = 0$ on Figure 6).

The domains involved in this algorithm are parameterized polyhedra. Therefore, computing the $next_{\mathcal{D}}$ function can be done using parametric integer linear programming [23], [18]. A solution has then the form of a *piecewise quasi-affine* function. (Quasi-affine functions are affine functions where

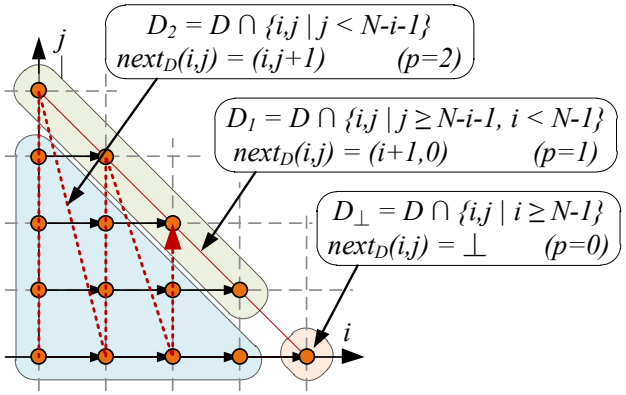
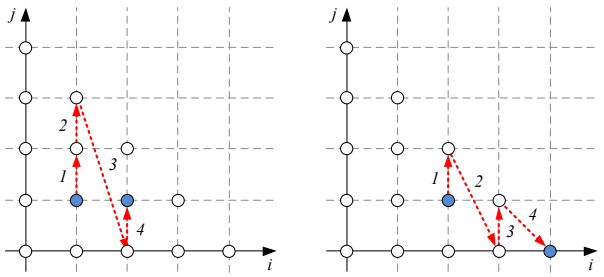


Fig. 6: Expression of the immediate successor (the function $next_{\mathcal{D}}$) for the example of Figure 1. The expression differs according to the current iteration in D_1, D_2 or $D_\perp$.



(a) Using 4th line of equ. (3), $next^4_{D_{S_0}}(1,1) = (2,1)$

(b) Using 6th line of equ. (3), $next^4_{D_{S_0}}(2,1) = (4,0)$

Fig. 7: Representation of the function $next^4_{D_{S_0}}$ on example of Figure 1 when N = 5 for 2 example operations.

division or modulo by an integer constant are allowed.) Since we only need to look for a constant number Δ of iterations ahead, the $next^{\Delta}_{\mathcal{D}}$ function is built by Δ compositions of $next_{\mathcal{D}}$.

Example: ■ When $\vec{x} = (i,j)$, the value of $next_{D_{S_0}}(\vec{x})$ for the example of Figure 6 is as follow:

$$next_{D_{S_0}}(i,j) = \begin{cases} (i,j+1) & \text{if} & j < N-i-1 \\ (i+1,0) & \text{elseif} & i < N-1 \\ \perp & \text{else} & \end{cases}$$

Composing this relation four times, one obtains the $next^4_{D_{S_0}}(i,j)$ function, which is given by :

$$next^4_{D_{S_0}}(i,j) =$$

$$\begin{cases} (i,j+4) & \text{if} & j \leq N-i-5 \\ (i+1,3) & \text{elseif} & i \leq N-5 \wedge j = N-i-1 \\ (i+1,2) & \text{elseif} & i \leq N-4 \wedge j = N-i-2 \\ (i+1,1) & \text{elseif} & i \leq N-3 \wedge j = N-i-3 \\ (i+1,0) & \text{elseif} & i \leq N-4 \wedge j = N-i-4 \\ (N-1,0) & \text{elseif} & i = N-3 \wedge j = 1 \wedge N \geq 3 \\ (N-2,0) & \text{elseif} & i = N-4 \wedge j = 3 \wedge N \geq 4 \\ \perp & \text{else} & \end{cases} \quad (3)$$

For example when $N = 5$ (the chosen parameter) and $(i,j) = (1,1)$, the 4th line of equation (3) is active ($j = N-i-3$ and $i \leq N-3$). Therefore the expression of the successor 4 iterations away is $(i+1,1) = (2,1)$, which can be checked on Figure 7a. ■

D. Building the Violated Dependency Set

As mentioned previously, a given dependency d is enforced by the nested loop pipelining transformation iff, for all $(\vec{x} \rightarrow \vec{y}) \in d$ such that $\vec{y} \in d^{-1}(\vec{x})$, we have $\vec{y} \succeq next^{\Delta}_{\mathcal{D}}(\vec{x})$. A consequence of this condition is that if $next^{\Delta}_{\mathcal{D}}(\vec{x}) = \perp$, that is when the successor Δ iterations later is out of the iteration domain, then the dependency d will be violated by the pipelined execution, because at least one sink of $\vec{x}$ will get the value computed by $\vec{x}$ "too late" due to the pipeline latency.

This observation allows the set $\mathcal{D}^{\dagger}_d$ of all the source iterations violating the dependency d to be built:

$$\mathcal{D}^{\dagger}_d = \left\{ \vec{x} \in src(d) \;\middle|\; \begin{array}{l} d^{-1}(\vec{x}) \prec next^{\Delta}_{\mathcal{D}}(\vec{x}) \\ \text{or } next^{\Delta}_{\mathcal{D}}(\vec{x}) \in \{\perp\} \end{array} \right\} \quad (4)$$

Checking the legality of a nested loop pipelining w.r.t. the dependency d then sums up to check the emptiness of this parameterized domain, which can be done with ISL [15]. Checking the legality condition for a whole SCoP involves checking the emptiness of the set $\mathcal{D}^{\dagger} = \bigcup_{d \in PRDG} \mathcal{D}^{\dagger}_d$.

Example: ■ Using the inverse of d given in section IV-B, and the $next^4_{D_{S_0}}(i,j)$ function of Equ. (3), one can then find the domain $\mathcal{D}^{\dagger}_d$ of the source iterations violating the data-dependency d using Equ. (4).

In our example, and after resorting to the simplification of this polyhedral domain thanks to a polyhedral library [15], one then obtains:

$$\mathcal{D}^{\dagger}_d = \{i,j | (i,j) \in \mathcal{D}_{S_0} \wedge N-4 < i < N-1 \wedge j < N-i-1\}$$

Since d is the only dependency in the loop nest, $\mathcal{D}^{\dagger} = \mathcal{D}^{\dagger}_d$. When one substitutes N by 5 (the chosen value in our example), one gets $\mathcal{D}^{\dagger} = \{(2,0),(2,1),(3,0)\}$, which is the set of points that causes a dependency violation in Figure 3. ■

E. The New Legality Check Algorithm

Algorithm 2 presents the legality check for nested loop pipelining. Argument $depth$ represents the number of inner loops on which the legality check is applied.

A few explanations are in order. The Bernstein expansion is used as a means to avoid some computations of $next^{\Delta}_{\mathcal{D}}$, since they are costly. Function $ehrhart\_card(R)$ is directly provided by the Barvinok library and $bernstein\_bound\_min(P)$ by ISL.

Function $next^{\Delta}_{\mathcal{D}}$ is built according to Algorithm 1. Function $restrict(PRDG, depth)$ removes the dependencies of the PRDG that are not carried by the $depth$ innermost loops, by intersecting the PRDG with the lexicographic equality, only for dimensions that are not within the $depth$ innermost loops.

Finally, notice that this method could be extended to a more general model of pipeline execution, where reads and writes

Algorithm 2 Checks nested loop pipeline legality

procedure BERNSTEINBOUNDING($\mathcal{D}$, d)
$R \leftarrow \left\{ \vec{x} \rightarrow \vec{z} \;\middle|\; \begin{array}{l} \vec{x} \in src(d) \wedge \vec{z} \in \mathcal{D} \\ \wedge\, \vec{x} \prec \vec{z} \prec d^{-1}(\vec{x}) \end{array} \right\}$
$P \leftarrow ehrhart\_card(R)$
return $bernstein\_bound\_min(P)$
end procedure
Require: $1 \leq depth \leq dim(\mathcal{D})$
procedure LEGALITYCHECK($PRDG$, $\mathcal{D}$, Δ, $depth$)
$\mathcal{D}^\dagger \leftarrow \emptyset$
$PRDG = restrict(PRDG, depth)$
for all $d \in PRDG$ **do**
$l \leftarrow bernsteinBounding(\mathcal{D}, d)$
if $l < \Delta - 1$ **then**
$next^{\Delta}_{\mathcal{D}} \leftarrow nextPower(\mathcal{D}, \Delta, depth)$
$\mathcal{D}^\dagger_d \leftarrow \left\{ \vec{x} \in src(d) \;\middle|\; \begin{array}{l} d^{-1}(\vec{x}) \prec next^{\Delta}_{\mathcal{D}}(\vec{x}) \\ \vee\, next^{\Delta}_{\mathcal{D}}(\vec{x}) \in \{\perp\} \end{array} \right\}$
$\mathcal{D}^\dagger \leftarrow \mathcal{D}^\dagger \cup \mathcal{D}^\dagger_d$
end if
end for
return $\mathcal{D}^\dagger = \emptyset$
end procedure

can occur during any stage. Δ would then have to be computed for each pair of read and write, and moreover, write after read and write after write dependencies would have to be taken into consideration.

V. POLYHEDRAL BUBBLE INSERTION

A legality condition is an important step toward automated nested loop pipelining. But it is possible to do better by *correcting* a given loop to make nested loop pipelining legal when the legality check fails. Our idea is to determine *at compile time* an iteration domain where *wait states*, or *bubbles*, are inserted in order to stall the pipeline so that the pipelined execution of the loop becomes legal. These bubbles should be inserted between the sources and the sinks of violated dependencies.

Two constraints are imposed to this correction method. First, bubbles are inserted in the domain scanned by the coalesced loop, not in the original loop nest. The reason is related to the behavior of most HLS tools, whose aggressive optimization techniques are less likely to discard bubbles on the coalesced loop, thus removing their effect. (This technicality could probably be overcome by introducing some kind of NOP instruction that the HLS tool would not optimize.)

Second, this correction mechanism is restricted to loop nests where at least the innermost loop can be pipelined without bubble insertion. In such loop nests, the violated dependencies are not carried by the innermost loop, and one can add the bubbles at the end of the innermost loop, only for iterations that are source of a violated dependency. It turns out that the corrected loop is then quite simple. To the contrary, and although it is perfectly possible to correct other kinds of loops, experience has shown us that the resulting corrected loop contains a large number of new guards, which make an improvement very unlikely.

The key question is to determine how many bubbles are actually required to fix the loop, as adding a single bubble

```
i=0;j=0;
while(i<N) {
#pragma ignore_mem_depcy Y
   if(j<N-i)
S0:   Y[j] = func(Y[j]);
   if((i>N-4&&j<N-i+2&&i<N-1)
      ||j<N-i-1)
       j++;
   else
       j=0,i++;
}
```

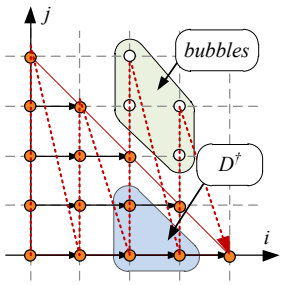


Fig. 8: Illustration of simple padding for $N = 5$ and $\Delta = 4$. White points correspond to inserted bubbles when $\mathcal{D}^\dagger$ is padded with $\Delta - 1 = 3$ bubbles.

in a loop may incidentally fix several violated dependencies. In the following, two solutions to this problem are proposed: simple padding, and optimized padding.

A. Simple Padding

In a previous work [24], the solution proposed was to pad every inner loop containing an iteration in $\mathcal{D}^\dagger$ with $\Delta - 1$ bubbles. As a matter of fact, this amounts to recreate the whole epilogue of the pipelined loop, but only for the outer loops that actually need it. Although simple, this approach turns out to be too conservative. Indeed one can notice that padding $\mathcal{D}^\dagger$ only, instead of the whole inner loops enclosing it, still adds a sufficient (and smaller) number of bubbles. How to build this set of bubbles is described in Algorithm 3, and the result is illustrated in Figure 8.

Algorithm 3 Builds the set of polyhedral bubbles

Require: $size \geq 0$
procedure PAD($\mathcal{D}$, $size$)
$n \leftarrow dim(\mathcal{D})$
$R \leftarrow \left\{ \vec{x} \rightarrow \vec{y} \;\middle|\; \begin{array}{l} \vec{y}_{[0..n-1]} = \vec{x}_{[0..n-1]} \wedge \\ \vec{x}_{[n]} \leq \vec{y}_{[n]} \leq \vec{x}_{[n]} + size \end{array} \right\}$
return $R(\mathcal{D})$
end procedure
Require: $\mathcal{D}^\dagger \subseteq \mathcal{D}$
procedure BUBBLESV1($\mathcal{D}$, $\mathcal{D}^\dagger$, Δ)
$\mathcal{B} \leftarrow \emptyset$
for all $\mathcal{D}^\dagger_d \in \mathcal{D}^\dagger$ **do**
$\mathcal{B} \leftarrow \mathcal{B} \cup pad(\mathcal{D}^\dagger_d, \Delta - 1)$
end for
return $\mathcal{B} - \mathcal{D}$
end procedure

Nevertheless, this method is still too conservative. For example, the reader will notice that the inner loop iterations in $\mathcal{D}^\dagger$ for index $i = 2$ in the example of Figure 8 do not actually need 2 bubbles, but only one.

B. Optimized Padding

The idea behind this second method is to build the set of bubbles while doing the legality check, and to pad every inner loop in $\mathcal{D}$ enclosing $\mathcal{D}^\dagger$ with the exact number of iterations required to preserve the dependency when applying nested loop pipelining. Consider a dependency between a source $\vec{x}$

```
i=0;j=0;
while(i<N) {
#pragma ignore_mem_depcy Y
   if(j<N-i)
S0:    Y[j] = func(Y[j]);
   if((i>N-4&&j<4&&i<N-1)
      ||j<N-i-1)
       j++;
   else
       j=0,i++;
}
```

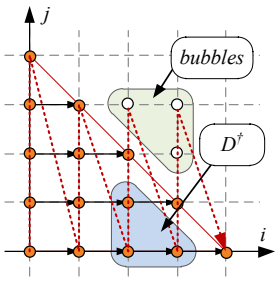


Fig. 9: Optimized padding for $N = 5$ and $\Delta = 4$. White points correspond to the inserted pipeline bubbles in the iteration domain.

and a sink $\vec{y}$, and let r be the reuse distance associated to these iteration points. One can note that if $r < \Delta$, then the dependence is violated, but inserting *at least* $\Delta - r$ bubbles will remove the dependency violation. Thus, the legality check is performed by increasing values of r, $r = 1..\Delta$, and if one computes for each value of r the set of source points $\mathcal{D}^{\dagger}_r$ for which the dependency is violated, it becomes possible to pad the iteration domain with a much more precise number of bubbles $\Delta - r$. This is exactly what Algorithm 4 describes and what is illustrated in Figure 9.

In this example, with $\Delta = 4$, one has $\mathcal{D}^{\dagger}_1 = \emptyset$, $\mathcal{D}^{\dagger}_2 = \{(3,0)\}$ and $\mathcal{D}^{\dagger}_3 = \{(2,0),(2,1)\}$. Therefore padding $i = 2$ by 1 bubble and $i = 3$ by 2 bubbles makes the pipeline legal.

Algorithm 4 Mixed legality check with bubble insertion to build the optimized set of bubbles

Require: $\mathcal{D}_1 \subseteq \mathcal{D}_2$
  procedure ENCLOSING($\mathcal{D}_1$, $\mathcal{D}_2$)
    $n \leftarrow dim(\mathcal{D}_1)$
    return $projectOut(\mathcal{D}_1, n-1) \cap \mathcal{D}2$
  end procedure
Require: $1 \leq depth \leq dim(\mathcal{D})$
  procedure BUBBLESV2($PRDG$, $\mathcal{D}$, Δ, $depth$)
    $PRDG \leftarrow restrict(PRDG, depth)$
    $\mathcal{B} \leftarrow \emptyset$
    for all $d \in PRDG$ **do**
      $l \leftarrow bernsteinBounding(\mathcal{D}, d)$
      if $l < \Delta - 1$ **then**
        for all $r \in [1..\Delta - 1]$ **do**
          $next^r_{\mathcal{D}} \leftarrow nextPower(\mathcal{D}, r, depth)$
          $\mathcal{D}^{\dagger}_{d_r} \leftarrow \{\vec{x} \in src(d) | d^{-1}(\vec{x}) = next^r_{\mathcal{D}}(\vec{x})\}$
          if $\mathcal{D}^{\dagger}_{d_r} \neq \emptyset$ **then**
            $\mathcal{B} \leftarrow \mathcal{B} \cup pad(enclosing(\mathcal{D}^{\dagger}_{d_r}, \mathcal{D}), \Delta - r)$
          end if
        end for
      end if
    end for
    return $\mathcal{B} - \mathcal{D}$
  end procedure

C. The Loop Coalescing Transformation

Once the bubble domain has been computed, the final step consists in regenerating C code for the HLS tool, after coalescing the loop. To do so, the loop nest structure is translated into a software finite state machine expressed as a while loop. Starting from a polyhedral representation of the loop nest, there are two possible approaches for implementing this transformation.

```
for (i = 0; i < 100; i++)
  for (j = 0; j < 2; j++)
    S(i,j);
```

(a) Original sample loop.

```
i = 0, j = 0;
while (i < 100)
  if (j < 2) {
    S(i,j);
    j++;
  } else
    j = 0, i++;
```

(b) CFG-coalesced loop.

```
i = 0, j = 0;
while (i < 100)
  S(i,j);
  if (j < 1)
    j++;
  else
    j = 0, i++;
```

(c) Boulet et al.-coalesced loop.

Fig. 10: Difference between (b) CFG- and (c) Boulet et al. loop coalescing. In CFG loop coalescing, whenever j reaches 2, that is $1/3$ of the iterations, time is spent on control only. Using Boulet et al. approach, j never reaches the value 2.

The first approach relies on the ClooG [3] code generator to produce a loop nest that scans the iteration domains (including bubbles). Coalescing can then be done by rewriting the Control Flow Graph (CFG) of the generated loop nest, as it is done by Ylvisaker et al. [10]. The main advantage of this approach is its simplicity. (From what we understand, this is the approach followed by RHLS when implementing the *nested loop pipelining* transformation.) But it has the drawback that the automaton is built from an implicit representation of the iteration domain rather than from its formal representation as a polyhedron. Consequently the resulting automaton may contain extra idle states that do not correspond to an actual iteration of the loop nest. This situation is shown in Figure 10b. Indeed, whenever j reaches the value 2, the `then` branch of the conditional is not taken, and statement S is not executed. This results in one cycle of outer loop execution spent only for control purpose, that is $1/3$ of the total execution time.

The second approach follows the method of Boulet et al. [18], and consists in building a finite state machine directly from the loop nest iteration domain using the $next_{\mathcal{D}}(\vec{x})$ function introduced in Section IV. The generated code visits the *exact* loop nest iteration domain. (See the example of Figure 10c where j never reaches 2.) The drawback of this second approach is that the resulting code tends to be complex in term of guards, which increases the hardware complexity.

Since the full loop nest needs not be coalesced, we combine both generation methods by letting the designer add directives to specify how many inner loops should be coalesced. The ClooG software is then used to generate the outer loops (using the `stop` option), and Boulet et al.'s method to generate the inner loops.

VI. Experimental results

This section describes how the pipeline legality check and the polyhedral bubble insertion are implemented within a compiler framework, and provides qualitative and quantitative evidence showing that they lead to significant performance improvements at the price of a moderate increase in hardware complexity.

A. The GeCoS source-to-source compiler

GeCoS (Generic Compiler Suite) is an open source-to-source compiler infrastructure [25] integrated within the Eclipse framework and entirely based on Model Driven Software Development tools. GeCoS is specifically targeted to HLS back-ends, and it provides built-in support for Mentor algorithmic data types C++ templates.

GeCoS also contains a loop transformation framework based on the polyhedral model, which extensively uses third party libraries (ISL [15] for manipulating polyhedral domains and solving parametric integer linear problems, and ClooG [3] for polyhedral code generation). All the transformations presented in this work were implemented within this framework.

B. Benchmark Kernels and Experiment Conditions

To evaluate this approach, a set of application kernels representing good candidates for our legality check and for the bubble insertion algorithms was selected. These kernels were chosen to exercise the robustness and correctness of RHLS when confronted to non-trivial cases. When needed, they were modified so that they would allow pipelining for their innermost loop but still expose loop carried dependencies on all their outer loops. As a consequence, nested loop pipelining could not be used blindly, as it might have led to a dependency violation.

The benchmark kernels are as follows:

- Prodmat: a product of 2 matrices where the dependency of the accumulation is moved to the second loop using a loop interchange transformation.
- BBFIR: a block based FIR where the loop nest is skewed to remove the dependencies on the innermost loop. It is the only 2-nested loop kernel.
- Jacobi: a 2D Jacobi stencil with the same transformation as BBFIR.
- FW: an implementation of the Floyd-Warshall algorithm where the loop nest is also skewed.
- QRC: a QR Decomposition using CORDIC operators where the original C implementation already shows an innermost loop without dependency.

The benchmark kernels were submitted to GeCoS, for parsing, polyhedral analysis, and application of pipeline source-to-source transformations. The transformed kernels were then processed by RHLS, which produced VHDL code. The resulting VHDL code was then synthesized using Quartus, to target an Altera Stratix IV FPGA.

Unless otherwise stated, data types of the kernels were 32 bit fixed point numbers. Each kernel was assigned a fixed target pipeline latency Δ in the following way. RHLS was forced, by appropriate directives, to generate a – possibly incorrect – pipeline hardware description of the kernel, targeting a frequency of 100 MHz on the Altera Stratix IV FPGA. Since the pipeline latency is essentially related to the complexity of statement, and much less to the pipeline control, this method provides realistic latency values.

		Run-time (ms)						
		RHLS		PBI V1			PBI V2	
Kernel	Δ	1D	2D	1D	2D	3D	2D	3D
Prodmat	4	ok	illegal	13	230	1671	23	38
BBFIR	4	fail	prevent	17	2125		131	
Jacobi	8	fail	prevent	27	623	2918	153	273
FW	3	fail	prevent	54	69	224	88	183
QRC	13	fail	prevent	30	2449	879	284	718

ok : HLS tool pipelines and it is legal
prevent : HLS tool does not pipeline and it is effectively illegal
fail : HLS tool does not pipeline whereas it is legal
illegal : HLS tool does pipeline whereas it is illegal

TABLE I: Result of RHLS on application kernels, and run-time (Xeon at 2.4GHz) of two versions of the Polyhedral Bubble Insertion (PBI) for the given latency. The analysis for 1 dimension (column 1D) takes exactly the same time for both algorithms, while algorithm 3 (V1) takes more time than algorithm 4 (V2) when bubble insertion is required.

C. Qualitative results

The first experiment was to check if RHLS could pipeline the innermost loop without any directive (1D), and whether it would prevent the second loop to be pipelined (2D). Results for the benchmark kernels are shown in the left part of Table I. RHLS could only pipeline Prodmat, whose array accesses are very simple (see column 1D). It would also allow the two innermost loops of Prodmat to be pipelined, whereas it would lead to a dependency violation when the sizes of the matrices are smaller than the data-path latency (see column 2D). For all other kernels, the dependency analysis of RHLS was too conservative, and it failed at pipelining the innermost loop. It also prevented the second loop to be pipelined.

The rightmost part of Table I presents the run time of our methods (PBI, standing for Polyhedral Bubble Insertion). We run both Algorithms 3 (V1) and 4 (V2) on the kernels. Column 1D gives the time needed to check that there is no dependency on the innermost loop (identical for both algorithms). Columns 2D V1 and 2D V2 give respectively the time needed for algorithm 3 and algorithm 4 to build the set of bubbles for the second loop, and 3D for the whole loop nest.

These run-times depend on the shape of the loop iteration domains, and on the latency Δ, but they are acceptable in the aforementioned context.

One can note that V2 is in general much faster than V1. This is because the linear programming problems solved by V2 are simpler than those of V1.

Generating this $next_{\mathcal{D}}^{\Delta}$ function is very compute intensive. Table II shows that the run-time is still acceptable when the latency Δ and the depth are reasonable. Although the run-time grows exponentially with the latency and depth, the analysis still finishes in extreme cases (last row of the table).

Kernel	$\Delta =$ depth	2	4	8	16	32	64
Prodmat	1	2	2	3	3	3	2
	2	9	11	20	82	657	7350
	3	23	45	222	2089	27250	469139
BBFIR	1	3	4	3	3	3	3
	2	34	74	189	1057	15013	362109
Jacobi	1	1	1	1	1	1	1
	2	7	11	52	442	5156	78068
	3	11	20	80	603	6478	92093
FW	1	1	1	1	1	1	1
	2	6	10	42	358	4335	66184
	3	8	15	58	465	5336	76615
QRC	1	1	1	1	1	1	1
	2	7	9	18	76	615	7019
	3	11	15	48	413	5237	80844

TABLE II: Run-time (Xeon at 2.4GHz) in milliseconds to generate the $next_{\mathcal{D}}^{\Delta}$ function for several kernels, with different Δ and at different depths.

Kernel	Version	Hardware Characteristics			
		ALUT	REG	DSP	Freq. (MHz)
Prodmat	RHLS 1D	489	215	4	272
	PBI 2D V1	629	228	4	235
	PBI 2D V2	553	198	4	246
	PBI 3D V2	559	226	4	231
BBFIR	RHLS 1D	553	152	4	185
	PBI 2D V1	727	231	4	213
	PBI 2D V2	649	241	4	241
FW	RHLS 1D	383	74	0	271
	PBI 2D V1	951	108	0	159
	PBI 2D V2	859	87	0	210
	PBI 3D V2	955	99	0	180
Jacobi	RHLS 1D	1012	845	8	164
	PBI 2D V1	1153	936	8	172
	PBI 2D V2	1226	948	8	168
	PBI 3D V2	1417	975	8	172
QRC	RHLS 1D	5375	2461	24	155
	PBI 2D V1	5792	2755	28	158
	PBI 2D V2	5684	2745	28	155
	PBI 3D V2	5772	2730	28	152

TABLE III: Hardware characteristics for our nested pipeline implementations (PBI 2D V1, PBI 2D V2 and PBI 3D V2) compared to innermost pipeline (RHLS 1D).

To push the methods, algorithm 4 was applied to QRC, with a target frequency of 200 MHz, a 72 bit data-type, and trying to pipeline the 3 loops. For this extreme scenario, RHLS gave a latency of 67 cycles, and algorithm 4 was able to build the set of bubbles, with a run-time of 25 mn. This shows that the method, although costly, is realistic for quite complex examples.

D. Quantitative results

Inserting bubbles in the loop nest makes the control of the loop more complex, since the bubbles domain adds guards to the statements, and constraints to the loop bounds. This extra control code results in additional hardware for the control in the hardware description generated by HLS tools, and it can reduce the maximum frequency achieved by the logic synthesis.

To evaluate the actual impact of these methods, the hardware complexity and the run-time of the algorithms on the benchmark kernels were estimated. Hardware was generated by RHLS using directives to make the tool ignore the dependencies known to be false positives.

Table III provides an evaluation of the hardware complexity and frequency. For each problem size, four versions were considered: RHLS 1D generates hardware using RHLS. PBI 2D V1 and PBI 2D V2 are Algorithms 3 and Algorithm 4 respectively, applied on the second level of the loop nest. PBI 3D V2 is Algorithm 4 applied on the whole loop nest. (Notice that Table III does not contain pipelining results of RHLS, since as explained in subsection VI-C, RHLS failed to detect possible pipelining or would generate illegal pipelines on the kernels.) For each method, Table III displays an estimation of the area cost in terms of registers (REG), adaptive look-up tables (ALUT) and DSP operators (DSP), and it provides an estimation of the maximum frequency of the synthesized design.

The area overhead when inserting bubbles is moderate (less than 25%), except for FW for which the cost doubles. The reasons of this exception are first, that FW only involves additions and comparisons of integer values, which do not cost a lot as compared to the control, hence the huge overhead when control is added; second, the bubble domains of FW contains a number of additional constraints.

The frequency of the generated hardware is smaller when the domain of the bubbles is complex, compared to the original iteration domain (for Prodmat and FW), or is equivalent when the bubble domain is relatively simple (QRC, Jacobi). This was expected, since additional constraints make the control data-path longer, thus increasing the critical path. For unknown reasons, RHLS achieved a much higher frequency when pipelining the 2 loops in the BBFIR kernel, whereas the bubbles domain is complex.

Table IV displays the number of cycles required to execute the pipelined kernels, and it provides the wall clock time w.r.t. the maximum frequency in Table III. For each kernel, two problem sizes were considered in order to evaluate the effect of this parameter.

As shown by the Ratio column, the number of cycles is in general smaller when using nested loop pipelining, because the approach inserts fewer bubbles than the number of cycles required by a full flush at each iteration of the outer loops. However, when the problem size increases, the loop trip count is high, and the flush overhead decreases. Therefore, the reduction of the number of cycles does not compensate for the lower frequency and the area overhead.

Note that algorithm 4 (PBI 2D V2) always insert less bubbles than algorithm 3 (PBI 2D V1), or at worst, the same number (see Prodmat). However, this does not come at the expense of a higher complexity hardware.

For QRC with problem size = $3 \times 3 \times 3$ the gain is relatively small. This is because the loop trip count is small, compared to the latency, thus the set of inserted bubbles is comparable in size to the number of flush cycles of the single loop pipeline.

Results for large problem sizes show that in general, as expected, nested loop pipelining is effective only when the loop count is small compared to the pipeline latency.

Kernel	Problem Size	Performance			
		Version	# Cycles	Time (ns)	Ratio
Prodmat ($\Delta = 4$)	4^3	RHLS 1D	157	577	
		PBI 2D V1	89	378	1.52
		PBI 2D V2	89	361	1.59
		PBI 3D V2	68	294	1.96
	128^3	RHLS 1D	2179456	8012709	
		PBI 2D V1	2097921	8927323	0.89
		PBI 2D V2	2097921	8528134	0.93
		PBI 3D V2	2097156	9078597	0.88
BBFIR ($\Delta = 4$)	256×8	RHLS 1D	3336	18032	
		PBI 2D V1	2552	11981	1.50
		PBI 2D V2	2030	9279	1.94
	1024×32	RHLS 1D	37548	202962	
		PBI 2D V1	34412	161558	1.25
		PBI 2D V2	32282	148050	1.37
FW ($\Delta = 3$)	16^3	RHLS 1D	6625	24446	
		PBI 2D V1	4178	26276	0.93
		PBI 2D V2	4169	19852	1.23
		PBI 3D V2	4107	22816	1.07
	128^3	RHLS 1D	2260737	8342202	
		PBI 2D V1	2097682	13192968	0.63
		PBI 2D V2	2097673	9988919	0.83
		PBI 3D V2	2097163	11650905	0.71
Jacobi ($\Delta = 8$)	30×16	RHLS 1D	13261	80859	
		PBI 2D V1	10981	63843	1.26
		PBI 2D V2	7831	46613	1.73
		PBI 3D V2	7568	44000	1.83
	30×256	RHLS 1D	2072461	12636957	
		PBI 2D V1	1941421	11287331	1.11
		PBI 2D V2	1937431	11532327	1.09
		PBI 3D V2	1937168	11262604	1.12
QRC ($\Delta = 13$)	3^3	RHLS 1D	90	580	
		PBI 2D V1	89	563	1.03
		PBI 2D V2	82	529	1.09
		PBI 3D V2	81	532	1.08
	32^3	RHLS 1D	23406	151006	
		PBI 2D V1	28670	181455	0.83
		PBI 2D V2	17464	112670	1.34
		PBI 3D V2	17610	115855	1.30

TABLE IV: Performance for different problem sizes with architecture characteristics described in Table III.

VII. Related Work and Discussion

This section compares our approach to previous work on loop pipelining and nested loop pipelining.

A. Loop pipelining in hardware synthesis

Earlier work on systolic architectures addressed the problem of fine grain parallelism extraction. Among others, Derrien et al. [8] propose to use iteration domain partitioning to help combine operation-level (pipeline) and loop-level parallelism. A somewhat similar problem is addressed by Teich et al. [9] who propose to combine modulo scheduling with loop-level parallelization techniques. The main limitation of these contributions is that they only support *one-dimensional schedules* [26], which significantly limit their applicability.

Alias et al. [27] address the problem of generating efficient nested loop pipelined hardware accelerators leveraging custom floating-point data-paths. Their approach (also based on the polyhedral model) consists in finding a parallel hyperplane for the loop nest, and then in deriving a tiling (hyperplanes and tile sizes) chosen in such a way that a pipeline of depth Δ is legal. This research only targets perfectly nested loops, and it also requires that incomplete tiles be padded to behave like full tiles. Besides, the authors restrict themselves to uniform dependencies, so as to guarantee that the reuse distance is always constant for a given tile size. In contrast, our methods are more general and support imperfectly nested loops with non-uniform (i.e. affine) dependencies. In addition, in the case of tiled iteration domains, we can provide a more precise correction (in terms of extra bubbles) that would not require padding all incomplete tiles.

The Compaan/Laura [20] tool set takes another point of view, as it does not try to find a global schedule for the program statements. Instead, each statement of the program is mapped on its own process. Dependencies between statements are then materialized as communication buffers, following the so-called *polyhedral process network* semantics [28]. Because the causality of the schedule is enforced by the availability of data on the channel output, there is no need for taking statement execution latency into account in the process schedule [29]. On the other hand, this approach suffers from a significant hardware complexity overhead, as each statement requires its own hardware controller plus possibly complex reordering memory structures. In our opinion, this research is geared toward task level parallelism rather than toward fine grain parallelism/pipeline.

B. Nested loop software pipelining

Software pipelining has proved to be a key optimization for leveraging the instruction level parallelism available in most compute intensive kernels. Since its introduction by Lam et al. [30] a lot of work has been carried out on this topic. Two directions have mainly been addressed:

- Many contributions tried to extend software pipelining applicability to wider classes of program structures, by taking control flow into consideration [31].
- The other main research direction focused on integrating new architectural specificities and/or additional constraints when trying to solve the optimal software pipelining problem [32].

Among these numerous contributions, some of them tackle problems very close to ours.

Rong et al. [7] study software pipelining for nested loops. Their goal is to pipeline a loop that is not innermost by using loop interchange, and to merge the flush and initialization parts of the pipeline to reduce the impact of the latency. This is similar to our research – although they do not target hardware synthesis, – but they restrict themselves to a narrow subset of loops (only constant bound rectangular domains) and they do not leverage exact instance-wise dependency information.

Fellahi et al [33] address the problem of prologue/epilogue merging in sequences of software pipelined loops. Their work is also motivated by the fact that the software pipeline overhead tends to be a severe limitation as many embedded-multimedia algorithms exhibit *low trip count* loops. Again, our approach differs from theirs in the scope of its applicability, as we are able to deal with loop nests (not only sequences of loops), and as we solve the problem in the context of HLS tools at the source level through a loop coalescing transformation. On the contrary, their approach handles the problem at machine code level, which is not possible in our context (source-to-source transformation).

Thanks to the specificities of the Itanium EPIC architecture, Muthukumar et al. [6] are able to control the flush of the pipeline. Their research aims at counting the number of iterations separating the definition of a value from its use. However their approach is only applicable to bounded loops with uniform dependencies. This work also proposes a correction mechanism that partially drains the pipeline when memory dependencies may be violated. Since the same number of bubbles is used for all the iterations of the immediate outer loop, this method implies fewer guards, but also more bubbles than our approach.

C. Loop coalescing

Loop coalescing was initially used in the context of parallelizing compilers in order to reduce the synchronization overhead [34]. Indeed, since synchronization occurs at the end of each innermost loop, coalescing loops reduces the number of synchronizations during the program execution. Such an approach has some similarity to ours (indeed, one could see the flush of the innermost loop pipeline as a kind of synchronization operation). However, in our case we can benefit from an exact timing model of the synchronization overhead, which can be used to remove unnecessary synchronization steps.

D. Correcting illegal loop transformations

The idea of correcting a schedule as a post-transformation step is not new, and it was introduced by Bastoul et al [35]. Their idea was to first look for interesting combinations of loop transformations (be they legal or not), and then to try to fix possible illegal schedule instances with loop shifting transformations. Their result was later extended by Vasilache et al. [36], who considered a wider space of correcting transformations.

Our work differs from theirs in that we do not propose to modify the existing loop schedule, but rather to add artifact statements to improve the behavior of the loop.

E. Generality of the approach

The technique presented in this work can be applied to a subset of imperative programs known as Static Control Parts. Some extensions to this model have been proposed to handle dynamic control structures such as while loop and/or non-affine memory accesses [37]. Proposed approaches suggest approximating non-affine memory index function by a parameterized polyhedral domain (the parameter being used to model the ”fuzziness” introduced by the non-affine array references).

As a matter of fact, the technique presented in this work is able to deal with arbitrary (non-affine) memory access functions, by considering a conservative name based data-dependency analysis whenever non-affine index functions are involved in the program. Extending the approach to program constructs where the iteration space cannot be represented as a parametric polyhedron is however likely to be much more challenging.

VIII. Conclusion

In this paper, a new technique, called polyhedral bubble insertion, was proposed to support nested loop software pipelining in C-to-hardware synthesis tools. A nested pipeline legality check that can be combined with a compile-time bubble insertion mechanism was described. This bubbles insertion allows the causality in the pipelined schedule to be enforced, for a large class of loop nests called SCoPs, thanks to the polyhedral model representation of loops. This technique was implemented as a proof of concept in a source-to-source compiler, and experiments show promising results for nested loops operating on small iteration domains (up to 45% execution time reduction in terms of clock cycles, with a moderate hardware complexity overhead).

This research also demonstrates the potential of source-to-source compilation as a means to overcome the shortcomings of state of the art HLS tools. Especially, source-to-source compilers are very well suited to implementing program transformations using high-level representations such as the polyhedral model.

Future research could go in several directions.

- First, we believe that these methods could be used in more classical optimizing compiler back-ends, for example for deeply pipelined VLIW architectures with many functional units. In that case one simply needs to use the value of the loop body initiation interval as additional information to determine which dependencies may be violated.
- Since all the control within the polyhedral model fits into (quasi-)affine expressions, one possible enhancement would be to apply aggressive strength reduction in order to reduce the overhead induced by the extra guards.
- Another research direction is to investigate the case when dependencies cross several iterations of the outer loop, since our optimized bubble insertion has shown to be suboptimal in this case.
- The value of Δ in the pipeline model is a conservative over-approximation. For example, operations may have different latencies, thus the distance between read and write may differ depending on the operation, resulting in several Δ values (one per read/write pair). More accurate values for Δ could, for example, be obtained by analyzing more precisely the schedule provided by the HLS tool. This might result in less conservative pipeline legality conditions.

Acknowledgments

The authors would like to thanks Sven Verdoolaege, Cedric Bastoul and all the contributors to the wonderful pieces of software that are ISL and ClooG.

This work was funded by the INRIA STMicroelectronics Nano2012-S2S4HLS project.

References

[1] “AutoESL Design Technologies.” http://www.autoesl.com/.
[2] M. Graphics, “Catapult-C Synthesis.” http://www.mentor.com.

[3] C. Bastoul, "Code Generation in the Polyhedral Model Is Easier Than You Think," in *Proceedings of PACT'13*, (Juan-les-Pins, France), pp. 7–16, Sept. 2004.

[4] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "PLuTo: A Practical and Fully Automatic Polyhedral Program Optimization System," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, (Tucson, AZ), ACM, June 2008.

[5] L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos, "Iterative Optimization in the Polyhedral Model: Part II, Multidimensional Time," in *Proceedings of PLDI'08*, (Tucson, Arizona), pp. 90–100, ACM Press, June 2008.

[6] K. Muthukumar and G. Doshi, "Software Pipelining of Nested Loops," in *Proceedings of the 10th Int. Conf. on Compiler Construction*, CC'01, (London, UK), pp. 165–181, Springer-Verlag, 2001.

[7] H. Rong, Z. Tang, R. Govindarajan, A. Douillet, and G. R. Gao, "Single-Dimension Software Pipelining for Multidimensional Loops," *ACM Trans. Archit. Code Optim.*, vol. 4, March 2007.

[8] S. Derrien, S. Rajopadhye, and S. Kolay, "Combined Instruction and Loop Parallelism in Array Synthesis for FPGAs," in *Proceedings of the 14th Int. Symp. on System Synthesis*, pp. 165–170, 2001.

[9] J. Teich, L. Thiele, and L. Z. Zhang, "Partitioning Processor Arrays under Resource Constraints," *VLSI Signal Processing*, vol. 17, no. 1, pp. 5–20, 1997.

[10] B. Ylvisaker, C. Ebeling, and S. Hauck, "Enhanced Loop Flattening for Software Pipelining of Arbitrary Loop Nests," tech. rep., University of Washington, 2010.

[11] M. W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul, "The Polyhedral Model is More Widely Applicable Than You Think," in *Proceedings of Int. Conf. on Compiler Construction*, pp. 283–303, Springer, 2010.

[12] J. F. Collard, D. Barthou, and P. Feautrier, "Fuzzy Array Dataflow Analysis," in *Proceedings of the fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 92–101, ACM, 1995.

[13] P. Feautrier, "Dataflow Analysis of Array and Scalar References," *International Journal of Parallel Programming*, vol. 20, no. 1, pp. 23–53, 1991.

[14] W. Kelly, W. Pugh, and E. Rosser, "Code Generation for Multiple Mappings," *Proceedings of the 5th Symposium on the Frontiers of Massively Parallel Computation*, pp. 332–341, February 1995.

[15] S. Verdoolaege, "ISL: An Integer Set Library for the Polyhedral Model," in *ICMS* (K. Fukuda, J. Van Der Hoeven, M. Joswig, and Y. Takayama, eds.), vol. 6327 of *Lecture Notes in Computer Science*, (Kobe, Japan), pp. 299–302, Springer, Sept. 2010.

[16] P. Feautrier, "Some Efficient Solutions to the Affine Scheduling Problem. Part II. Multidimensional Time," *International Journal of Parallel Programming*, vol. 21, no. 6, pp. 389–420, 1992.

[17] F. Quilleré, S. Rajopadhye, and D. Wilde, "Generation of Efficient Nested Loops from Polyhedra," *International Journal of Parallel Programming*, vol. 28, pp. 469–498, 2000.

[18] P. Boulet and P. Feautrier, "Scanning Polyhedra Without Do-Loops," in *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, (Washington, DC, USA), p. 4, IEEE Computer Society, 1998.

[19] A.-C. Guillou, P. Quinton, and T. Risset, "Hardware Synthesis for Multi-Dimensional Time," in *Proceedings of ASAP 2003*, (The Hague, The Netherlands), pp. 40–50, IEEE Computer Society, June 2003.

[20] A. Turjan, B. Kienhuis, and E. F. Deprettere, "Classifying Interprocess Communication in Process Network Representation of Nested-Loop Programs," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 6, no. 2, 2007.

[21] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe, "Counting Integer Points in Parametric Polytopes Using Barvinok's Rational Functions," *Algorithmica*, vol. 48, no. 1, pp. 37–66, 2007.

[22] P. Clauss and V. Loechner, "Parametric Analysis of Polyhedral Iteration Spaces," *The Journal of VLSI Signal Processing*, vol. 19, pp. 179–194, 1998.

[23] P. Feautrier, "Parametric Integer Programming," *RAIRO Recherche opérationnelle*, vol. 22, no. 3, pp. 243–268, 1988.

[24] A. Morvan, S. Derrien, and P. Quinton, "Efficient Nested Loop Pipelining in High Level Synthesis using Polyhedral Bubble Insertion.," in *Proceedings of Int. Conf. on Field Programmable Technologies* (R. Tessier, ed.), pp. 1–10, IEEE, 2011.

[25] "The GeCoS (Generic Compiler Suite) Source-to-Source Compiler Infrastructure." http://gecos.gforge.inria.fr/.

[26] P. Feautrier, "Some Efficient Solutions to the Affine Scheduling Problem. I. One-Dimensional Time," *International Journal of Parallel Programming*, vol. 21, no. 5, pp. 313–347, 1992.

[27] C. Alias, B. Pasca, and A. Plesco, "Automatic Generation of FPGA-Specific Pipelined Accelerators," in *Proceedings of Int. Symp. on Applied Reconfigurable Computing*, Mars 2011.

[28] S. Verdoolaege, "Polyhedral Process Networks," in *Handbook of Signal Processing Systems* (S. Bhattacharyya, R. Leupers, J. Takala, and E. Deprettere, eds.), Heidelberg, Germany: Springer, first ed., 2004.

[29] C. Zissulescu, B. Kienhuis, and E. F. Deprettere, "Increasing Pipelined IP Core Utilization in Process Networks Using Exploration," in *Proceedings of FPL 2004* (J. Becker, M. Platzner, and S. Vernalde, eds.), vol. 3203 of *Lecture Notes in Computer Science*, (Leuven, Belgium), pp. 690–699, Springer, Aug. 2004.

[30] M. S. Lam, "Software Pipelining: An Effective Scheduling Technique for VLIW Machines," in *Proceedings of PLDI'88*, pp. 318–328, 1988.

[31] H.-S. Yun, J. Kim, and S.-M. Moon, "Time Optimal Software Pipelining of Loops with Control Flows," *International Journal of Parallel Programming*, vol. 31, pp. 339–391, 2003.

[32] C. Akturan and M. F. Jacome, "CALiBeR: a Software Pipelining Algorithm for Clustered Embedded VLIW Processors," in *Proceedings of ICCAD'01*, (Piscataway, NJ, USA), pp. 112–118, IEEE Press, 2001.

[33] M. Fellahi and A. Cohen, "Software Pipelining in Nested Loops with Prolog-Epilog Merging," in *HiPEAC* (André Seznec and Joel S. Emer and Michael F. P. O'Boyle and Margaret Martonosi and Theo Ungerer, ed.), vol. 5409 of *Lecture Notes in Computer Science*, pp. 80–94, Springer, 2009.

[34] M. T. O'Keefe and H. G. Dietz, "Loop Coalescing and Scheduling for Barrier MIMD Architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, pp. 1060–1064, September 1993.

[35] C. Bastoul and P. Feautrier, "Adjusting a Program Transformation for Legality," *Parallel Processing Letters*, vol. 15, pp. 3–17, Mar. 2005.

[36] N. Vasilache, A. Cohen, and L.-N. Pouchet, "Automatic Correction of Loop Transformations," in *Proceedings of the 16th Int. Conf. on Parallel Architecture and Compilation Techniques*, PACT'07, (Washington, DC, USA), pp. 292–304, IEEE Computer Society, 2007.

[37] M. Belaoucha, D. Barthou, A. Eliche, and S.-A.-A. Touati, "FADAlib: an Open Source C++ Library for Fuzzy Array Dataflow Analysis," in *Proceedings of the Seventh International Workshop on Practical Aspects of High-Level Parallel Programming (PAPP 2010)*, 2010.

Antoine Morvan is a PhD student in Computer Science at Ecole Normale Suprieure of Cachan – antenne de Bretagne since 2009. He is also a member of the Cairn research group at IRISA, Rennes. His research interests include High-Level Synthesis, optimizing compilers, and computer aided design.

Steven Derrien obtained his PhD from University of Rennes 1 in 2003, and is now professor at Univesrity of Rennes 1. He is also a member of the Cairn research group at IRISA. His research interests include High-Level Synthesis, loop parallelization, and reconfigurable systems design.

Patrice Quinton obtained a degree of Engineer in Computer Science of ENSIMAG (Grenoble, France), in 1972, and a These d'Etat in Mathematics of the University of Rennes (France) in 1980. He has been Directeur de Recherches of the CNRS, and head of the VLSI Parallel Architectures group of IRISA in Rennes between 1982 and 1997, and since then, he is professor of the University of Rennes 1. Patrice Quinton is currently deputy director of the brittany branch of Ecole Normale Suprieure of Cachan and member of the Cairn research group at IRISA, Rennes. His interests include parallel architectures, VLSI, systolic arrays, computer aided design and sensor networks. Patrice Quinton is co-author of one book, and author and co-author of about one hundred journal papers, international conference communications or book chapters.