# Efficient Nested Loop Pipelining in High Level Synthesis using Polyhedral Bubble Insertion

Antoine Morvan, Steven Derrien, Patrice Quinton

# Efficient Nested Loop Pipelining in High Level Synthesis using Polyhedral Bubble Insertion

Antoine Morvan [1], Steven Derrien [2], Patrice Quinton [1]

[1] *INRIA-IRISA-ENS Cachan*
[2] *INRIA-IRISA-Université de Rennes 1*
*Campus de Beaulieu, Rennes, France*
{amorvan,sderrien,quinton}@irisa.fr

***Abstract*—Loop pipelining is a key transformation in high-level synthesis tools as it helps maximizing both computational throughput and hardware utilization. Nevertheless, it somewhat looses its efficiency when dealing with small trip-count inner loops, as the pipeline latency overhead quickly limits its efficiency. Even if it is possible to overcome this limitation by pipelining the execution of a whole loop nest, the applicability of nested loop pipelining has so far been limited to a very narrow subset of loops, namely perfectly nested loops with constant bounds. In this work we propose to extend the applicability of nested-loop pipelining to imperfectly nested loops with affine dependencies by leveraging on the so-called polyhedral model. We show how such loop nest can be analyzed, and under certain conditions, how one can modify the source code in order to allow nested loop pipeline to be applied using a method called *polyhedral bubble insertion*. We also discuss the implementation of our method in a source-to-source compiler specifically targeted at High-Level Synthesis tools.**

## I. Introduction

After almost two decades of research effort, High-Level Synthesis (HLS) is now about to hold its promises : there now exists a large choice of robust and mature C to hardware tools [1], [2] that are even now used as production tools by world-class chip vendor companies. However, there is still room for improvement, as these tools are far from producing designs with performance comparable to those of expert designers. The reason of this difference lies in the difficulty, for automatic tools, to discover information that may have been lost during the compilation process. We believe that this difficulty can be overcome by tackling the problem directly at the source level, using source-to-source optimizing compilers.

Indeed, even though C to hardware tools dramatically slash design time, their ability to generate efficient accelerators is still limited, and they rely on the designer to expose parallelism and to use appropriate data layout in the source program.

In this paper, our aim is to improve the applicability (and efficiency) of nested loop pipelining (also known as nested software pipeling) in C to hardware tools. Our contributions are described below:

- We propose to solve the problem of nested loop pipelining at the source level using an automatic *loop coalescing* transformation.
- We provide a nested loop pipelining legality check, which indicates (given the pipeline latency) whether the pipelining enforces data-dependencies.
- When this condition is not satisfied, we propose a correction mechanism which 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.

The proposed approach was validated experimentally on a set of representative applications for which we studied the trade-off between performance improvements (thanks to full nested loop pipelining) and area overhead (induced by additional guards in the control code).

Our approach builds on leading edge automatic loop parallelization and transformation techniques based on the *polyhedral model* [3], [4], [5], and it is 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]. This is the reason why we call this method *polyhedral bubble insertion*.

This article is organized as follows, Section II provides an in depth description of the problem we tackle in this work, and emphasizes the shortcomings of existing approaches. Section III aims at summarizing the principles of program transformations and analysis in the polyhedral framework. Sections IV and V present our pipeline legality analysis and our pipeline schedule correction technique and Section VI provides a quantitative analysis of our results. In section VII we present relevant related work, and highlight the novelty of our contribution. Conclusion and future work are described in section VIII.

## II. Motivations

### A. Loop pipelining in HLS tools

The goal of this section is to present and motivate the problem we address in this work, that is *nested loop pipelining*. To help the reader understand our contributions, we will use throughout the remaining of this work a running toy loop-nest example shown in Figure 1; it consists in a double nested

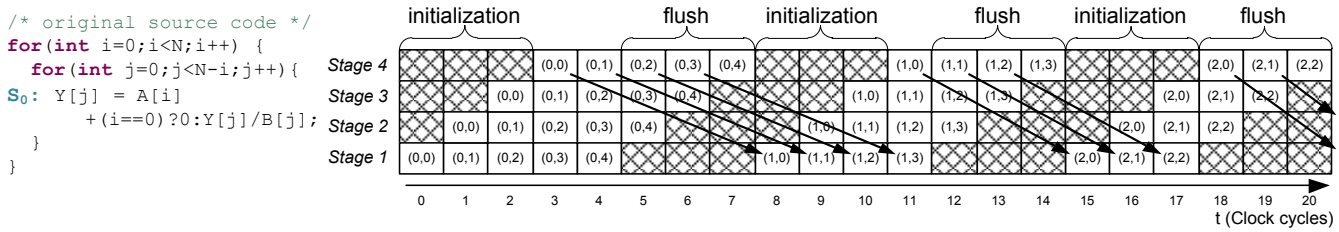


Fig. 2. Motivating example, with a representation of its pipelined execution for $N = 5$, $II = 1$ and $\Delta = 4$. The arrows reprensent dependences between operations.

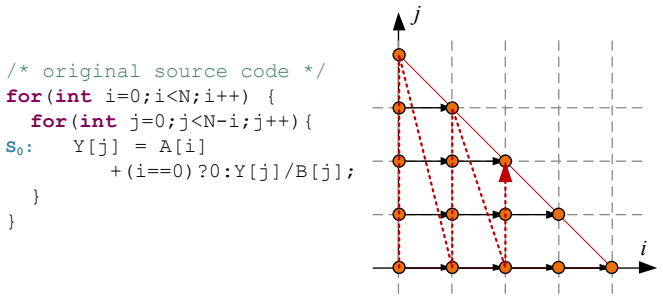


Fig. 1. Motivating example, with its iteration domain and data dependencies (black arrows) for $N = 5$. The red dashed arrow represents the execution order. .

loop operating on a triangular *iteration domain* – the iteration domain of a loop is the set of values taken by its loop indices[1].

The reader can observe that the inner loop (along the $j$ index) exhibits no 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. However, there exists a dependence between $i$ iterations, since `Y[j]` (left-hand side) depends on `Y[j]` (right-hand side) that was modified during the previous $i$ iteration. Therefore, overlapping the execution of two successive $i$ iterations has to be done with care, in order to respect this dependency.

*Loop pipelining* is characterized by two important parameters:

- The *initiation interval* (denoted $II$ in the following), that corresponds to the number of clock cycles separating the execution of two loop iterations.
- The *latency* (denoted $\Delta$) that gives the number of clock cycles required to completely execute one iteration of the loop.

As an illustration, Figure 2 depicts the pipelined execution of the example of Figure 1 with an initiation interval $II = 1$ and a latency of $\Delta = 4$. In practice the value of $II$ 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.

Because it helps maximizing the computation throughput and because it improves hardware utilization, loop pipelining is a key transformation in High-Level Synthesis tools. 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 $II = 1$) is a very common practice. The use of very deep pipeline is even more common when targeting FPGAs devices, as it is often a way to compensate for their relative lower clock speed compared to ASICs. Besides, because the register-cost overhead of pipelining can be easily absorbed by the large amount of flip-flop available on most devices, deeply pipelining FPGA datapath is a very profitable optimization.

However, the performance improvements obtained through pipelining are often hindered by the fact that these tools rely on very basic data-dependency analysis algorithms, and hence they may fail to detect when such a pipelined execution is possible, especially when the inner loop involves complex memory access patterns.

To help designers cope with these limitations, most tools hence offer the ability to bypass part of this conservative dependency analysis through the use of 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 pipelined schedule and then an incorrect circuit, and hence puts the burden to the designer.

### B. The Pipeline Latency Overhead

For loops with large iteration count – we call *loop iteration count* 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 $\Delta$, one may observe a very significant performance degradation, as the pipeline flushing phases dominate the execution time. This is the case of our example in Figure 2. For a value of $N = 5$ and $\Delta = 4$, we obtain a hardware

[1]This toy loop is actually a simplified excerpt from the QR factorization algorithm.

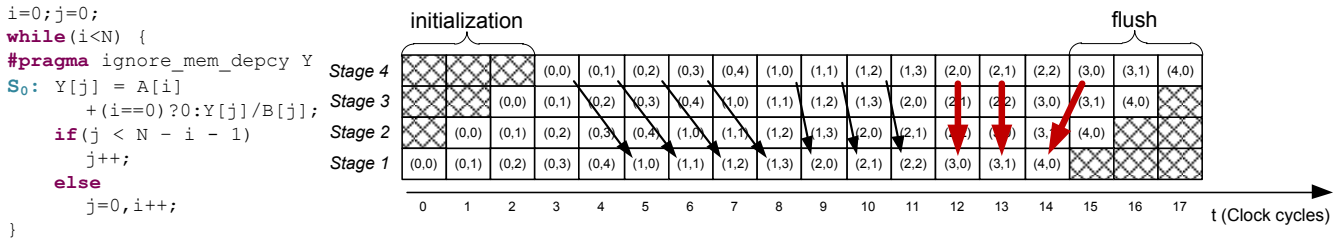


Fig. 3. Illegal nested loop pipelining for $N = 5$, $II = 1$ and $\Delta = 4$. Bold arrows show broken dependences.

utilization rate of only $50\%$. 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 pipeline.

Returning to our example, had it to be mapped to custom hardware by experienced designers, it would have certainly reached a hardware utilization close to $100\%$ thanks to a handcrafted schedule, in which the execution of successive iterations of the $i$ loop would have been carefully overlapped.

### C. Nested loop pipelining & coalescing

It turns out that such an optimization actually corresponds to a *nested loop pipelining* as initially proposed by Doshi et al. [6]. Such *nested loop pipelining* can be realized through a *loop coalescing* transformation, that flattens a loop nest into a single loop that scans the original loop nest domain, and then pipelines the new loop.

It is worth noticing that *nested loop pipelining* was only studied in the scope of a very restrictive subset of loop nests (perfectly nested loop with constant bounds and uniform dependencies) or with relatively imprecise dependency information, which significantly restricts its applicability and/or 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 now more difficult to analyze (they do not depend on loop indices as in Figure 1), we are tempted to bypass some of the dependence analysis through a `#pragma ignore_mem_dpcy Y` directive to enable loop pipelining, as explained in subsection II-A. This directive tells the scheduler to ignore data dependences related to the `Y[j]` array accesses in the statement following the directive. Without such directive, the conservative dependence analysis forbids pipelining.

While this scheduling seems correct at the first glance, it appears that some Read after Write dependencies are violated when $i \geq 3$, as shown in Figure 3. 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 we have evalutated, only one of them (let us call it *Trebuchet-C++*) actually provides the ability to perform such automatic nested loop pipelining. However, its implementation in the tool suffers from severe flaws and generates illegal schedules whenever the domain is not rectangular and/or has non constant loop bounds. From what we understand, even without directives to ignore data dependences, the tool fails for the very same reasons as depicted in Figure 3, that is the tool's analysis is assuming that there are no dependencies carried by the outerloop over the `Y` array.

It can be argued that a simple solution for handling non rectangular loop domains is to resort to a *linearization* of the loop nest prior to pipelining. This linearization consists in padding the iteration domain with *wait states* iterations so as to ensure that the domain scanned by the loop nest is rectangular. This approach turns out to be very inefficient in practice: in our example, the execution overhead (50 %) would be as large as for the non nested pipeline case, beside it does only solve the problem in the case of uniform data dependencies.

### D. Contributions of this work

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

In addition to this legality check, we also propose a technique to *correct* an a priori illegal nested pipeline schedule by inserting *wait states* in the coalesced loop, so as to derive the most efficient *legal* pipelined schedule. These wait states correspond to properly inserted *bubbles* in the pipeline, so the name *polyhedral bubble insertion* of our method.

Finally, to enable experimentation and to remain as vendor independent as possible, we propose an implementation of the polyhedral bubble insertion in the context of a source-to-source compiler that can be used as a preprocessing tool to be used in front of third parties HLS compilers.

## III. Background

In order to do the analysis and the cycle-accurate schedule correction, an iteration-wise dependence analysis as well as a new intermediate representation of loops is necessary. The

polyhedral model is a robust mathematical framework to represent loops; it also comes with a set of techniques to analyze and transform loops and to generate code. In this section, we briefly present the background needed to understand our method.

### A. Structure and Limitations

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 loop and conditional control structures and the only allowed statements are array assignments of arbitrary expressions with array reads (scalar variables are special cases viewed as zero-dimensional arrays). The loop bounds, the conditions and array subscripts have to be affine expressions of loop indexes and parameters.

Each statement $S$ surrounded by $n$ loops in a SCoP has an associated domain $\mathcal{D}_S \subseteq \mathbb{Z}^n$. The domain $\mathcal{D}_S$ represents the set of values the indices of the loops surrounding $S$ can take. Each vector of values in $\mathcal{D}_S$ is called an *iteration vector*, and $\mathcal{D}_S$ is called the *iteration domain* of $S$. $\mathcal{D}_S$ is defined by a set of affine constraints, i.e. the set of loop bounds and conditionals on these indexes. In what follows, we call *operation* a particular statement iteration, 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's constraints for the only statement of Figure 1 are as follows:

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

The polyhedral model is limited to the aforementioned class of programs. This class can be however extended to a larger class of programs at the price of a loss of accuracy in the dependance analysis [10], [11].

### B. Dependences and Scheduling

The real strength of the polyhedral model is its capacity to handle iteration wise dependence analysis on arrays [12]. The goal of dependence analysis is to answer questions of the type *"what is the statement that produced the value being read at current operation, and for what iteration vector?"* For example, in the program of Figure 1, what is the operation that wrote the last value of the right-hand side reference `Y[j]`?

Iterations of a statement in a loop nest can be ordered by the lexicographic order of their iteration vectors. The combination of the lexicographic order and the textual order gives the precedence order (noted $\succ$) of operations, that gives the execution order of operations in a loop nest. When considering sequential loop nests, the precedence order is total.

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

In the polyhedral model, building this precedence order can be done exactly. Therefore, transformations of the loop execution order, also known as *scheduling* transformations, can be constrained to enforce dataflow dependences. This feature may be used to check the legality of a given transformation, but also to automatically compute the space of all possible transformations, in order to find the "best" one. However this is not the topic of this paper, and the reader is referred to Feautrier [13] and Pouchet et al. [5] for more details.

### C. Code generation

Once a loop nest has been scheduled (for example, to incorporate some pipelining), the last step of source-to-source transformation consists in re-generating a sequential code. Two approaches to solve this problem dominate in the litterature. The first one was developed by Quillere and al. [14] and later extended by Bastoul in the context of 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 also tries to regenerate guardless loops, but also provides options to find a trade-off between code size and guards [15].

The second approach, developed by Boulet et al. [16] aims at generating code without loops. The principle is to determine during one iteration the value of the next iteration vector, until all the iteration domain has been visited. Since this second approach behaves like a finite state machine, it si believed to be it is more suited for hardware implementation [17], though there is still very few quantitative evidences to back-up this claim.

## IV. Legality Check

In this section, we propose sufficient conditions for ensuring that a given loop coalescing transformation is legal w.r.t to the data-dependencies of the program.

Consider a sink reference to an array (i.e. a right-hand side array reference in a statement), and let $\vec{y}$ denote its iteration vector. Let $\vec{x}$ be the iteration vector of the source reference for this array reference. Let us write $d$ the function that maps $\vec{y}$ to $\vec{x}$, so that $\vec{x} = d(\vec{y})$.

We define $\Delta$ as the highest latency, in the pipeline datapath, between a read and a write inducing a dependence. We can formulate the conditions under which a loop coalescing is legal w.r.t to this data-dependency as follows: for a pipelined schedule with a latency of $\Delta$, the coalescing will violate data dependencies when 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 less than $\Delta$.

This condition is trivially enforced in one particular case, that is when the loops to be coalesced do not carry any dependences, that is when the loops are parallel. This is possible since one may want to pipeline only the $n-1$ inner loops of the loop nest in which the dependences are only carried by the outermost loop. In such a case, the pipeline is flushed at each step of the outermost loop, hence the latency does not break any dependence.

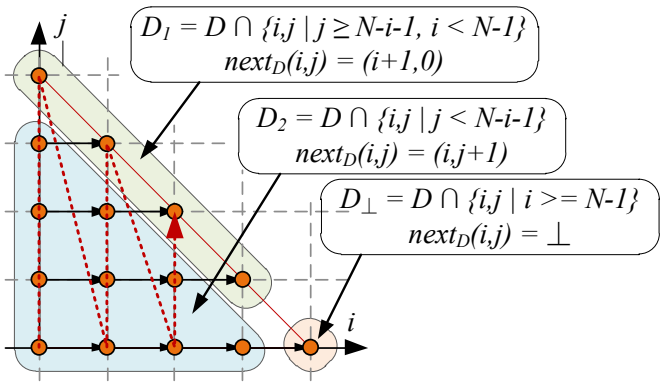


Fig. 4. Sub-domains $D_1$ and $D_2$ have different expressions for their immediate successor.

Let $p$ be the depth of the loop that carries a dependence, the coalescing is ensured to be legal if the loop to be coalesced are at a depth greater than $p$. In practice, the innermost loop is the only depth carrying no dependence, as shown in the example of Figure 1.

Determining if a coalescing is legal then requires a more precise analysis, by computing the number of points between a source iteration $\vec{x}$ and its sink $\vec{y}$. This indeed amounts to count the number of integral points inside a parametric polyhedral domain and corresponds to the *rank* function as proposed by Turjan et al. [18], for which we can obtain a closed form expression using the Barvinok library [19]. However these expressions are in the form of parametric multivariate pseudo-polynomials, and checking whether the value of such polynomials admits a given lower bound is impossible in the general case.

Because of this limitation, we propose another technique which does not involve any polyhedral counting operation. In this approach, we construct a function $next^{\Delta}_{\mathcal{D}}(\vec{x})$ that computes for a given iteration vector $\vec{x}$ its successor $\Delta$ iterations away in the coalesced loop nest's iteration domain $\mathcal{D}$. We then check that all the sink iteration vectors $\vec{y} = d^{-1}(\vec{x})$ of the dependency $d$ are such that $\vec{y} \succeq next^{\Delta}_{\mathcal{D}}(\vec{x})$. In other words, we make sure that the value produced at iteration $\vec{x}$ is used at least $\Delta$ iterations later.

The only difficulty in this legality check lies in the construction of the $next^{\Delta}_{\mathcal{D}}(\vec{x})$ function. This is the problem we address in the following subsection.

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

We will derive the $next^{\Delta}_{\mathcal{D}}$ function by leveraging on a method presented by Boulet et al[16] to compute the immediate successor in $\mathcal{D}$ of an iteration vector $\vec{x}$ according to the lexicographical order. This function is expressed as a solution of a lexicographic minimization problem on a parametric domain made of all successors of $\vec{x}$.

The algorithm works as follows: we start by building the set of points for which the immediate successor belongs to the same innermost loop (say at depth $p$). This set is represented as $D_2$ in the example of Figure 4. We then do the same for the set of points of $\mathcal{D}$ for which no successors were found at previous step, but this time we look for their immediate successors along the loop at depth $p-1$ as shown by the domain $D_1$ in Figure 4. This procedure is then repeated until all dimensions of the domain have been covered by the analysis. At the end, the remaining points are the lexicographic maximum (that is the end) of the domain, and their successor is noted as $\perp$ ($D_{\perp}$ on Figure 4).

The domains involved in this algorithm are parameterized, therefore the approach requires the use of a Parametric Integer Linear Programming solver [20], [16] to obtain a solution which is in the form of a quasi affine mapping function that defines the sequencing relation. Because it is a quasi-affine function[2], and because we only need to look for a constant number of iterations ahead (the latency of the pipeline that we call $\Delta$), we can easily build the $next^{\Delta}_{\mathcal{D}}$ function. This is done by applying the function to itself $\Delta$ times as shown in Equ. (1) :

$$next^{\Delta}_{\mathcal{D}}(\vec{x}) = \overbrace{next_{\mathcal{D}} \bullet next_{\mathcal{D}} \bullet \ldots \bullet next_{\mathcal{D}}}^{\Delta}(\vec{x}) \quad . \tag{1}$$

*Example:* Let us compute the $next_{\mathcal{D}}(\vec{x})$ predicate for the example of Figure 4, where we have $\vec{x} = (i, j)$

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

Note that $\perp$ represents the absence of successor in the loop. Applying the relation four times to itself we then obtain the $next^4_{\mathcal{D}}(i,j)$ predicate, which is given by the mapping below :

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

### B. Building the violated dependency set

As mentioned previously, a given dependency is enforced by the coalesced loop iff we have $\vec{y} \succeq next^{\Delta}_{\mathcal{D}}(\vec{x})$ with $\vec{y} = d^{-1}(\vec{x})$. When $next^{\Delta}_{\mathcal{D}}(\vec{x}) \in \{\perp\}$, that is when the successor $\Delta - 1$ iterations later is out of the iteration domain, the dependence is obviously broken. We can then build $\mathcal{D}^{\dagger}$ the domain containing all the iterations sourcing one of these violated dependencies, using the equation below

$$\mathcal{D}^{\dagger} = \left\{ \vec{x} \in \mathcal{D}_{src} \;\middle|\; \begin{array}{l} d^{-1}(\vec{x}) \prec next^{\Delta}_{\mathcal{D}}(\vec{x}) \\ \text{or } next^{\Delta}_{\mathcal{D}}(\vec{x}) \in \{\perp\} \end{array} \right\} \tag{3}$$

[2]Quasi affine function are affine functions where division (or modulo) by an integer constant are allowed.

where $\mathcal{D}_{src}$ is the set of sources of a dependency in $\mathcal{D}$.

It is important to note that in case of a parameterized domain, the set of these iterations may itself be a parameterized domain. Checking the legality of a nested loop pipelining then sums up to check the emptiness of this parameterized domain, which can easily be done with ISL [21] or Polylib [22].

*a) Example:* In what follows, we make no difference between relations and functions, following the practice used in the ISL tool. In our example, we have the following dependency relation :

$$d(i,j \rightarrow i',j' : i,j \in \mathcal{D} \wedge i \geq 1 \wedge i' = i-1 \wedge j' = j)$$

which can easily be reverted as

$$d^{-1}(i,j \rightarrow i',j' : i',j' \in \mathcal{D} \wedge i \geq 0 \wedge i' = i+1 \wedge j' = j)$$

which corresponds to the data-dependency.

Using the $next^4_{\mathcal{D}}(i,j)$ function obtained in (2), we can then build the domain $\mathcal{D}^\dagger$ of the source iterations violating a data dependency using (3).

In our example, and after resorting to the simplification of this polyhedral domain thanks to a polyhedral library [21], we then obtain :

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

When we substitute $N$ by 5 (the chosen value in our example), we have $\mathcal{D}^\dagger = \{(2,0),(2,1),(3,0)\}$, which is the set of points that causes a dependency violation in Figure 3.

## V. Bubble Insertion

While a legality condition is an important step toward automated nested loop pipelining, it is possible to do better by *correcting* a given schedule to make the coalescing legal. Our idea is to determine *at compile time* an iteration domain where *wait states*, or *bubbles*, are inserted in order to stall the pipeline to make sure that the coalesced loop execution is legal w.r.t data dependencies. Of course we want this set to have the smallest possible impact on performance, both in terms of number of cycles, and in terms of overhead caused by extra guards and housekeeping code.

We already know from the previous subsection the domain $\mathcal{D}^\dagger$ of all iterations whose source violates the dependency relation. To correct the pipeline schedule we can insert additional wait-state iterations in the domain scanned by the coalesced loop. These wait state iterations should be inserted between the source and the sink iterations of the violated dependency. One obvious solution is to add these extra iterations at the end of the inner loop enclosing the source iteration, so that this extra cycle may benefit to all potential source iteration within this innermost loop.

The key question in this problem is to determine how many of such wait states are actually required to fix the schedule, as adding a single wait state in a loop may incidentally fix/correct several violated data-dependency. In the following we propose a simple technique to solve the problem.

```
i=0;j=0;
while(i<N) {
#pragma ignore_mem_depcy Y
   if(j<N-i)
S0:    Y[j] = A[i]
         +(i==0)?0:Y[j]/B[j];
   if((i>N-4&&j<N-i+3&&i<N-1)
      ||j<N-i-1)
       j++;
   else
       j=0,i++;
}
```

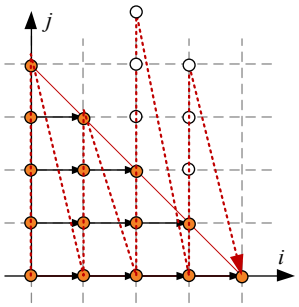


Fig. 5. Corrected pipeline (conservative) for $N = 5$ and $\Delta = 4$. White points correspond to pipeline epilogue inserted only for the iterations that need it.

```
i=0;j=0;
while(i<N) {
#pragma ignore_mem_depcy Y
   if(j<N-i)
S0:    Y[j] = A[i]
         +(i==0)?0:Y[j]/B[j];
   if((i>N-4&&j<3&&i<N-1)
      ||j<N-i-1)
       j++;
   else
       j=0,i++;
}
```

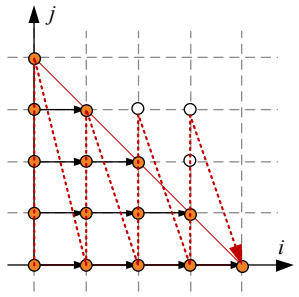


Fig. 6. Corrected pipeline (optimized) for $N = 5$ and $\Delta = 4$. White points correspond to the inserted pipeline bubbles in the iteration domain.

The simplest solution is to pad every inner loop containing an iteration in $\mathcal{D}^\dagger$ with $\Delta - 1$ wait-states. 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. The approach is illustrated in Figure 5, but turns out to be too conservative. For example, the reader will notice that the inner loops for indices $i = 2$ in the example of Figure 1 do not actually need $\Delta - 1 = 3$ additional cycles. In that case only one cycle of wait state is needed, and similarly, for $i = 3$, only two cycles are needed.

This solution is obviously not optimal, as one could easily find a better correction (that is with fewer bubbles), as the one shown in Figure 6. In this particular example, the lower overhead (in terms of wait-state) does not come at the price of an increase in the complexity of the code. This is however not the case in general, and there exist subtle trade-offs between the reduction of bubble count and the complexity of the control logic. We are currently investigating this topic.

## VI. Results and Validation

In this section, we describe how the transformation is being implemented within a compiler framework, and provide quantitative data showing that the approach is practical and lead to significant performance improvements at the price of a moderate increase in area.

### A. The Gecos source to source compiler

GeCoS (Generic Compiler Suite) is an open source-to-source compiler infrastructure [23] integrated within the Eclipse framework and entirely based on Model Driven Software Development tools. GeCoS is specifically targeted to HLS back-ends, and provides (among other features) built-in support for Mentor Algorithmic Data types C++ templates (`ac_int<>`, etc.).

GeCoS also provides a loop transformation framework based on the polyhedral model, that extensively uses third party libraries (ISL for manipulating polyhedral domains [24] and solving parametric integer linear problems, and Cloog [3] for polyhedral code generation). All the transformations presented in this work have been implemented within this framework.

### B. Implementing the loop coalescing transformation

Implementing the loop coalescing transformation amounts in a rewriting of the loop nest structure into a software finite state machine expressed in a while loop. There are two possible approach for implementing this rewriting.

The first approach uses the Control Flow Graph (CFG) corresponding to the loop nest as an input. The problem in this approach is that the automaton is built from an implicit representation of the iteration domain rather than from its formal representation as a polyhedron. As a consequence the resulting automaton contains extra idle states which do not correspond to an actual iteration of the loop nest. On the other hand the main advantage of this approach is its simplicity. From what we understand, this is the approach followed by our reference HLS tool when implementing the *nested loop pipelining* transformation.

The second approach follows the approach of Boulet et al. [16] and consists in building a finite state machine directly out of the loop nest iteration domain[3] using the $next_{\mathcal{D}}(\vec{x})$ mapping introduced in Section IV. Because it is the only way to ensure that the generated code visits the *exact* loop nest iteration domain, it is actually more efficient than the approach based on the CFG. However, the resulting code tends to be more complex and induces area overhead as the number of guards grows as the number of dimension of domain increases and its shape gets more complex.

Because one does generally not want to coalesce the full loop nest, our approach allows the designer to choose (by using a source code directive) how many of the inner most loops should be coalesced, and then use a combination of Cloog with the algorithm of Boulet & Feautrier to regenerate the transformed loop nest structure.

### C. Experimental results

The efficiency of our approach is obviously very sensitive to the trip count in inner loops sizes. As such it is very efficient for improving the performance of non perfectly tiled codes and of non rectangular domains. It is therefore more sensitive to the iteration domains size and domain shapes than to the structure of the application itself.

In this work, we limited ourself to two representative kernels (QR-Cordic, MatMul described below) that provide a good illustration the benefits and drawbacks of the approach.

- The *QR decomposition* is a key building block for wireless MIMO communication systems. The QR kernel operates over (small) non rectangular 3-dimensional iteration domain and is a good candidate for the approach. In the QR, only the most inner loop (index $k$) is parallel, and this loop has a non constant iteration count (Indeed the two innermost loops of the QR kernel resembles a lot our running toy example). As a consequence, direct nested pipelining causes dependence violation, and the kernel must therefore undergo a *bubble insertion step*.
- A *Matrix Multiplication* kernel, in which we performed a loop interchange to enable the pipelining of the 2 innermost loops. In this case the iteration domain is very simple (rectangular), but we allow in some cases the matrix sizes to be parameterized (i.e not known at compile time).

Because our reference HLS tool does not support division nor square root operations, we replaced these operations in the QR algorithm with deeply pipelined multipliers. We insist on the fact that this modification does not impact the relevance of the results given below, since our coalescing transformation only impacts the loop control, the body being left untouched.

For each of these kernels, we used varying fixed and parameterized iteration counts, and also used different fixed point arithmetic wordlength sizes for the loop body operations so as to be able to precisely quantify the trade-off between performance improvement (thanks to nested pipeline) and area overhead (because of extra control cost).

For each application instance, we compared the results obtained when using :

- Simple loop pipelining by our reference HLS tool.
- Nested loop pipelining by our reference HLS tool.
- Nested loop pipelining through loop coalescing and bubble insertion.

For all examples, we derived deeply pipelined datapaths (with $II = 1$ in all cases) and with latency values varying from 4 to 6 in the case of Matrix Multiplication, and from 9 to 12 in the case of the QR factorization depending on the fixed point encoding.

We provide three metrics of comparison: the total accelerator area cost (in LUT and registers), the number of clock cycles required to execute the program, and the clock frequency obtained by the design after place and route. All the results were obtained for an Altera Stratix-IV device with fastest speed-grade, and are given in Table I.

The quantitative evaluation of the area overhead induced by the use of nested pipelining is provided in Figure 7. Our results show that this overhead remains limited and even negligible when large functional units are being used (in the figure, `<a,b>` stands for a `a` bit wide fixed point format with

[3]It is to note that the method can accommodate with imperfectly nested loops thanks to the use of additional (scalar) dimensions which model textual ordering as in the Cloog library [3].

| | | | LUTs | | Registers | | DSPs | | Freq (MHz) | | Clock Cycles | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Benchmark | Latency | Size | HLS | Coal. | HLS | Coal. | HLS | Coal. | HLS | Coal. | HLS | Coal. |
| MM<48,16> | 6 cycles | param | 512 | 579 | 657 | 677 | 10 | 10 | 160 | 164 | n.a. | n.a. |
| | | 128 | 437 | 392 | 542 | 458 | 10 | 10 | 161 | 163 | 2114304 | 2097157 |
| | | 32 | 388 | 351 | 486 | 426 | 10 | 10 | 164 | 160 | 33984 | 32773 |
| | | 8 | 333 | 313 | 429 | 442 | 10 | 10 | 164 | 164 | 624 | 517 |
| MM<24,6> | 4 cycles | 32 | 239 | 200 | 170 | 130 | 4 | 4 | 240 | 250 | 33920 | 32771 |
| QR<48,16> | 12 cycles | param | 999 | 1190 | 1114 | 2169 | 10 | 10 | 166 | 166 | n.a. | n.a. |
| | | 128 | 1944 | 3262 | 7209 | 7891 | 10 | 10 | 162 | 164 | 902208 | 797050 |
| | | 32 | 1229 | 1534 | 2562 | 3230 | 10 | 10 | 167 | 165 | 23312 | 17370 |
| | | 8 | 1018 | 1951 | 1662 | 2297 | 10 | 10 | 166 | 167 | 868 | 746 |
| QR<24,6> | 9 cycles | 32 | 620 | 823 | 1064 | 1240 | 4 | 4 | 231 | 229 | 20336 | 15552 |

TABLE I
PERFORMANCE AND AREA COST FOR OUR NESTED PIPELINE IMPLEMENTATIONS

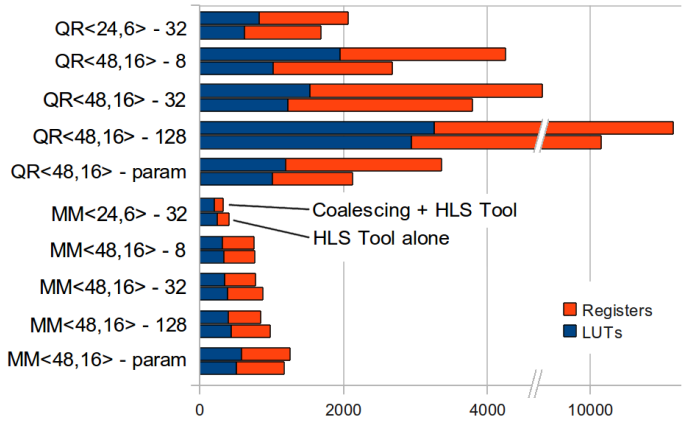


Fig. 7. Area overhead due to the loop coalescing and bubble insertion, this overhead is caused by extra guards on loop indices.

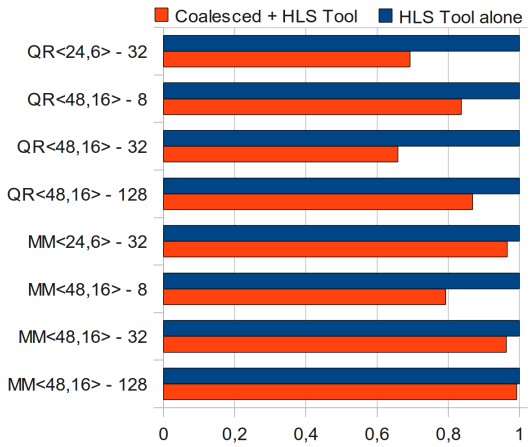


Fig. 8. Normalized execution time (in clock cycles) of the two innermost coalesced pipelined loops (with bubbles for QR) w.r.t to the non-coalesced pipelined loop.

b bit devoted to the integer part). Also, the approach does not significantly impact the clock frequency (less than 5% difference in all cases).

The improvement in execution time due to latency hiding are given in Figure 8. Here one can observe that the efficiency of the approach is highly dependant on the loops iteration count. While the execution time can decrease by up to 34% in some case, the benefits quickly decrease as the domain sizes grow. For larger iteration counts, the performance improvement hardly compensates the area overhead.

One interesting observation is that when no correction is needed (e.g constant size matrix multiplication) our coalescing transformation is more efficient in term of both performance and area than the nested pipeline feature provided with the tool, a result which is easy to explain (see VI-B).

In addition to this quantitative analysis, it is also interesting to point out which examples did cause our reference leading edge commercial HLS tools to either find a good nested loop pipelined schedule or to generate an illegal schedule violating the semantic of the initial program. For the *QR* example, the reference tool would systematically fail to generate a legal nested pipeline schedule for the algorithm. Furthermore it gives an illegal schedule whenever its iteration domain is parameterized.

| Benchmark | $next$ | $next^{15}$ |
|---|---|---|
| ADI Core | 1391 | 71517 |
| Block Based FIR | 697 | 131284 |
| Burg2 | 1166 | 36757 |
| Forward Substitution | 59 | 734 |
| Hybrid Jacobi Gauss Seidel | 15 | 3065 |
| Matrix Product | 187 | 29245 |
| QR Given Decomposition | 72 | 4554 |
| SOR 2D | 90 | 30151 |

TABLE II
$next$ AND $next^{15}$ RUNTIME EVALUATION (IN *ms*)

Last, we also evaluated the runtime needed to perform the $next^k$ operations for several examples. The goal is to demonstrate that the approach is practical in the context of a HLS tool. Results are given in Table II, and show that the runtime (in *ms*) remains acceptable in most cases.

## VII. Related work and discussion

### 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] proposed to use iteration domain partitioning to help combine operation level (pipeline) and loop level parallelism. A somewhat similar problem was addressed by Teich and al.[9] who proposed to combine modulo scheduling with loop-level parallelization techniques. The main limitation of these contributions is that they only support *one-dimensional schedules* [25], which significantly limit their applicability.

A recent work by Alias et al. [26] tackles a problem very similar to ours, as they try to address the problem of generating efficient nested loop pipelined hardware accelerators leveraging custom floating point datapaths. Their approach (also based on the polyhedral model) consists in finding a parallel hyperplane for the loop nest, and then derive a tiling (hyperplanes and tile sizes) which is chosen such that a pipeline of depth $\Delta$ is legal. The approach only targets perfectly nested loop and also requires incomplete tiles to be padded to behave like full-tiles. Besides they restrict themselves to uniform dependencies, so as to guarantee that the reuse distance (i.e the number of points separating a source and a sink) is always constant for a given tile size. In contrast, our framework is more general and supports 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 does not requires padding all incomplete tiles.

The Compaan/Laura [18] toolset takes another view on the problem, 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 which follow the so-called Polyhedral Process Network semantic [27]. 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 [28]. On the other hand, the approach suffers significant area cost overhead as each statement requires its own hardware controller plus possibly complex reordering memory structure. To our opinion, the approach 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. [29] a lot of work has been carried out on the topic (a survey is out of scope of this work). Two directions have mainly been addressed:

- Many contributions have tried to extends software pipelining applicability to wider classes of program structures, by taking control flow into consideration [30].
- The main other research direction has focused on integrating new architectural specificities and/or additional constraints when trying to solve the optimal software pipelining problem [31].

Among these numerous contributions, some of them have been tackling problems very close to ours.

First, Rong et al. [7] have already studied the problem of nested loop software pipelining. Their goal is clearly the same as ours, except that they do restrict themselves to a narrow subset of loop (only constant bound rectangular domains) and do not leverage exact instance-wise dependence information. Besides they do not address the problem from a hardware synthesis point of view. In this work, we tackle the problem for a wider class of programs (known as Static Control Parts), and also we relate the problem to *loop coalescing*.

Another related contribution is the work of Fellahi et al [32], who 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.

### C. Loop coalescing and loop collapsing

Loop coalescing was initially used in the context of parallelizing compilers, for reducing of synchronization overhead [33]. Since synchronization occurs at the end of each innermost loop, coalescing loops reduces the number of synchronization during the program execution. Such an approach is quite similar to ours (indeed, one could see the flushing 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 we can be used to remove unnecessary synchronization steps.

### D. Correcting illegal loop transformations

The idea of applying a correction on a schedule as a post-transformation step is not new, and was introduced by Bastoul et al [34]. Their idea was to first look for interesting combination of loop transformations (be they legal or not), and then try to fix possible illegal schedule instances through the use of loop shifting transformations. Their result was later extended by Vasilache et al. [35], who considered a wider space of correcting transformations.

Our work differs from theirs in that we do not propose to modify the existing schedule, but rather add artifact statements whose goal is to model so called *wait state* operations, which will then make loop coalescing legal w.r.t data dependencies.

## VIII. Conclusion

In this paper, we have proposed a new technique for supporting nested loop software pipelining in C to hardware synthesis

tools. The approach extends previous work by considering a more general class of loops nests. In particular we propose a nested pipeline legality check that can be combined with a compile time bubble insertion mechanism to enforce causality in the pipelined schedule. Our nested loop pipelining technique was implemented as a proof of concept and our preliminary experimental results show promising results for nested loops operating on small iteration domains (up to 30 % execution time reduction in terms of clock cycles, with a limited area overhead).

As a side-note, we believe that our approach can easily be adapted to be used in a more classical optimizing compiler back-ends. Of course, our approach would only makes sense for deeply pipelined VLIW machines with many functional units. In that case we simply need to use the value of the loop body initiation interval as an additional information to determine which dependencies may be violated.

## ACKNOWLEDGEMENT

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 founded by the INRIA-STMicroelectronic Nano2012 project.

## REFERENCES

[1] M. Graphics, "Catapult-c synthesis," http://www.mentor.com.

[2] "Autoesl design technologies," http://www.autoesl.com/.

[3] C. Bastoul, "Code Generation in the Polyhedral Model Is Easier Than You Think," in *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, Juan-les-Pins, France, Sep. 2004, pp. 7–16.

[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 *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*. Tucson, Arizona: ACM Press, June 2008, pp. 90–100.

[6] K. Muthukumar and G. Doshi, "Software pipelining of nested loops," in *Proceedings of the 10th International Conference on Compiler Construction*, ser. CC '01. London, UK: Springer-Verlag, 2001, pp. 165–181. [Online]. Available: http://portal.acm.org/citation.cfm?id=647477.727775

[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. [Online]. Available: http://doi.acm.org/10.1145/1216544.1216550

[8] S. Derrien, S. Rajopadhye, and S. Kolay, "Combined instruction and loop parallelism in array synthesis for fpgas," in *The 14th International Symposium on System Synthesis. Proceedings.*, 2001, pp. 165 – 170.

[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] M. W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul, "The polyhedral model is more widely applicable than you think," in *Compiler Construction*. Springer, 2010, pp. 283–303.

[11] 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*. ACM, 1995, pp. 92–101.

[12] P. Feautrier, "Dataflow analysis of array and scalar references," *International Journal of Parallel Programming*, vol. 20, no. 1, pp. 23–53, 1991.

[13] ——, "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.

[14] 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. [Online]. Available: http://dx.doi.org/10.1023/A:1007554627716

[15] W. Kelly, W. Pugh, and E. Rosser, "Code generation for multiple mappings," pp. 332–341, February 1995.

[16] P. Boulet and P. Feautrier, "Scanning Polyhedra without Do-loops," in *PACT '98: Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*. Washington, DC, USA: IEEE Computer Society, 1998, p. 4.

[17] A.-C. Guillou, P. Quinton, and T. Risset, "Hardware Synthesis for Multi-Dimensional Time," in *ASAP*. IEEE Computer Society, 2003, pp. 40–50.

[18] 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.

[19] 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.

[20] P. Feautrier, "Parametric integer programming," *RAIRO Recherche opérationnelle*, vol. 22, no. 3, pp. 243–268, 1988.

[21] S. Verdoolaege, *Integer Set Library: Manual*, 2010. [Online]. Available: http://www.kotnet.org/ skimo/isl/manual.pdf

[22] D. Wilde, "A library for doing polyhedral operations," IRISA, Tech. Rep., 1993.

[23] The Gecos Source to Source Compiler Infrastructure. [Online]. Available: http://gecos.gforge.inria.fr/

[24] S. Verdoolaege, "ISL: An Integer Set Library for the Polyhedral Model," in *ICMS*, ser. Lecture Notes in Computer Science, K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, Eds., vol. 6327. Springer, 2010, pp. 299–302.

[25] 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.

[26] *Automatic Generation of FPGA-Specific Pipelined Accelerators*, Mars 2011.

[27] S. Verdoolaege, *Handbook of Signal Processing Systems*, 1st ed. Heidelberg, Germany: Springer, 2004, ch. Polyhedral process networks.

[28] C. Zissulescu, B. Kienhuis, and E. F. Deprettere, "Increasing Pipelined IP Core Utilization in Process Networks Using Exploration," in *FPL*, ser. Lecture Notes in Computer Science, J. Becker, M. Platzner, and S. Vernalde, Eds., vol. 3203. Springer, 2004, pp. 690–699.

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

[30] 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, 10.1023/A:1027387028481. [Online]. Available: http://dx.doi.org/10.1023/A:1027387028481

[31] C. Akturan and J. M. F., "Caliber: a software pipelining algorithm for clustered embedded vliw processors," in *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, ser. ICCAD '01. Piscataway, NJ, USA: IEEE Press, 2001, pp. 112–118. [Online]. Available: http://dl.acm.org/citation.cfm?id=603095.603118

[32] M. Fellahi and A. Cohen, "Software Pipelining in Nested Loops with Prolog-Epilog Merging," in *HiPEAC*, ser. Lecture Notes in Computer Science, A. Seznec, J. S. Emer, M. F. P. O'Boyle, M. Martonosi, and T. Ungerer, Eds., vol. 5409. Springer, 2009, pp. 80–94.

[33] 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. [Online]. Available: http://portal.acm.org/citation.cfm?id=628913.629222

[34] C. Bastoul and P. Feautrier, "Adjusting a program transformation for legality," *Parallel processing letters*, vol. 15, no. 1, pp. 3–17, Mar. 2005, classement CORE : U.

[35] N. Vasilache, A. Cohen, and L.-N. Pouchet, "Automatic correction of loop transformations," in *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, ser. PACT '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 292–304. [Online]. Available: http://dx.doi.org/10.1109/PACT.2007.17