Automatic Correction of Loop Transformations

Nicolas Vasilache Albert Cohen Louis-Noël Pouchet
ALCHEMY Group, INRIA Futurs and Paris-Sud 11 University
firstname.lastname@inria.fr

Abstract

Loop nest optimization is a combinatorial problem. Due to the growing complexity of modern architectures, it involves two increasingly difficult tasks: (1) analyzing the profitability of sequences of transformations to enhance parallelism, locality, and resource usage, which amounts to a hard problem on a non-linear objective function; (2) the construction and exploration of search space of legal transformation sequences. Practical optimizing and parallelizing compilers decouple these tasks, resorting to a predefined set of enabling transformations to eliminate all sorts of optimization-limiting semantical constraints. State-of-the-art optimization heuristics face a hard decision problem on the selection of enabling transformations only remotely related to performance.

We propose a new design where optimization heuristics first address the main performance anomalies, then correct potentially illegal loop transformations a posteriori, attempting to minimize the performance impact of the necessary adjustments. We propose a general method to correct any sequence of loop transformations through a combination of loop shifting, code motion and index-set splitting. Sequences of transformations are modeled by compositions of geometric transformations on multidimensional affine schedules. We provide experimental evidence of the scalability of the algorithms on real loop optimizations.

1. Introduction

Loop tiling, fusion, distribution, shifting as well as unimodular transformations are pervasive tools to enhance locality and parallelism. In most real-world programs, a blunt use of these transformations violates the causality of the computation. On the bright side, loop optimization designers observed that a sequence of systematic preconditioning steps can often eliminate these violations; these steps are called *enabling* transformations. On the other side, this puts a heavy burden on any locality and parallelism enhancement heuristic [20]: the hard profitability problem it deals with is complicated by a decision problem to identify enabling transformations. Loop shifting or pipelining [6] and loop skewing [27] are such enabling transformations of the schedule, while loop peeling isolates violations on boundary conditions [13] and variable renaming or privatization removes dependences. Unfortunately, except in special cases (e.g., the unimodular transformation framework), one faces the combinatorial decision problem of choosing enabling transformations with no guarantee that they will do any good. This approach does not scale to real programs, involving sequences of tens or hundreds of carefully selected enabling transformations.

Our method makes these enabling transformations obsolete, replacing them by a much more tractable "apply and correct" approach. While selecting these enabling transformations a priori is a complex decision problem involving legality and profitability, our method takes the opposite direction: we compute the transformations needed to make a candidate sequence legal a posteriori. We rely on the *polyhedral model*, which promoted *affine schedules* as an abstract intermediate representation of the program semantics. The main algorithm to exhibit a "good" legal schedule in this model has been proposed by Feautrier [11] and was recently improved [12]. This algorithm, and its main extensions [19, 15, 3], rely on linear optimization models and suffer from multiple sources of combinatorial complexity: (1) the number of polyhedra it considers is exponential in the program size; (2) the dimensionality of the polyhedra is proportional to the program size, which incurs an exponential complexity in the ILP the algorithm relies on.

Radical improvements are needed to scale these algorithms to real-size loop nests (a few hundred statements or more), and to complement linear programming with more pragmatic empirical operation research heuristics. Our contribution reduces the dimensions of this combinatorial problem: it allows operation research heuristics to focus on the linear part of the schedule and perform a simpler scalable correction pass on the constant part of the schedule.

Our first contribution is *an automatic correction algorithm to fix an illegal transformation sequence* with "minimal changes", based on the analysis of *dependence violations*. This correction scheme amounts to translating (a.k.a. shifting) the schedule until dependences are satisfied. We

prove two important properties: (1) the algorithm is sound and *complete*: any multidimensional affine schedule will be corrected as long as dependence violations can be corrected by translation; (2) it applies a *polynomial* number of operations on dependence polyhedra.

Our second contribution is a novel, more practical formulation of *index-set splitting*: it is the most general technique to decompose iteration domains, allowing to build more expressive piecewise affine schedules. Its state-of-the-art formulation is another complex decision problem [13] associated with a non-scalable scheduling algorithm. Our approach replaces it by a simple heuristic, combined with the automatic correction procedure. This is extremely helpful, considering that each decision problem has a combinatorial impact on phase selection, ordering and parametrization. The ability to directly control code size expansion is another advantage of this reformulation.

Our third contribution is a practical implementation of these algorithms in the URUK platform [14]. Building on recent scalability improvements in polyhedral compilation [25], we show the effectiveness of our automatic correction scheme on two full SPEC CPU2000fp benchmarks.

The paper is structured as follows. We discuss related work in Section 2. The polyhedral model and array dependence analysis are briefly introduced in Section 3. We then define the correction problem in Section 4 and a complete algorithm to fix an illegal transformation by means of affine translations. Section 5 revisits and improves index-set splitting in the context of schedule corrections. Section 6 presents experimental results.

2. Related Work

The induced combinatorial of enabling transformation decision problems drive the search for a more compositional approach, where transformation legality is guaranteed by construction, using the Farkas lemma on affine schedules. As stated in the introduction, this is one of our main motivation, but taking all legality constraints into account makes the algorithms not scalable, and suggest staging the selection of a loop nest transformation into an "approximately correct" combinatorial step – addressing the main performance concerns – and a second *correction* step. The idea of applying "after-thought" corrections on affine schedules was first proposed by Bastoul [3]. However, it relies on considering the program and all its dependences at each step of the resolution, and so does not scale with program size.

We follow the idea of correcting illegal schedules, but focus on a particular kind of correction that generalizes loop *shifting*, *fusion* and *distribution* [1]. These three classical (operational) program transformations boil down to the same algebraic translation in multidimensional affine scheduling. These are some of the most important enabling transformations in the context of automatic loop parallelization [1, 6]. They induce less code complexity than, e.g., loop skewing (which may degrade performance due to the complexity of loop bound expressions), and can cope with a variety of cyclic dependence graphs. Yet the maximal parallelism problem, seen as a decision one, has been proved NP-complete by Darte [6]. This results from the inability to represent coarse-grain data-parallelism in loops in a linear fashion (many data-parallel programs cannot be expressed with multidimensional schedules [19]). Although we use the same tools as Darte to correct schedules, we avoid this pitfall by stating our problem as a linear optimization.

Decoupling research on the linear and constant parts of the schedule has been proposed earlier [7, 6, 26]. These techniques all use simplified representation of dependences (i.e., dependence vectors) and rely on finding enabling transformations for their specific purpose, driven by optimization heuristics. Our approach is more general as it applies to any potentially illegal affine schedule. Alternatively, Crop and Wilde [5] and Feautrier [12] attempt to reduce the complexity of affine scheduling with a modular or structural decomposition. These algorithms are effective, but still resort to solving large integer-linear programs. They are complementary to our correction scheme.

Our contribution also helps improving the productivity of domain experts. A typical case is the design of domain-specific program generators [22, 8], a pragmatic solution to the design of adaptive, nearly optimal libraries. Although human-written code is concise (code factoring in loops and functions), optimizing it for modern architectures incurs several code-size increasing steps, including function inlining, specialization, loop versioning, tiling, pipelining and unrolling. Domain-specific program generators (also known as active libraries) rely on feedback-directed and iterative optimization. Our approach focuses the problem on the dominant performance anomalies, eliminating the majority of the transformation steps (the enabling ones).

3. Polyhedral Model

The polyhedral model is a thorough tool to reason about analysis, optimization and parallelization of loop nests. We use the notations of the URUK framework [14], a normalized representation of programs and transformations. Under the normalization invariants of this framework, it is possible to apply any sequence of transformations without worrying about its legality, lifting the tedious constraint of ensuring the legality of a transformation before applying it. Only after applying a complete transformation sequence do we care about its correctness as a whole.

Polyhedral Representation. The target of our optimizations are sequences of loop nests with constant strides and

affine bounds. This includes non-rectangular, non-perfectly nested loops, and conditionals with Boolean expressions of affine inequalities. Loop nests fulfilling these hypotheses are amenable to representation in the polyhedral model. We call *Static Control Part* (SCoP) any *maximal syntactic program segment* satisfying these constraints [14]. Invariant variables within a SCoP are called *global parameters*, and refereed to as $\mathbf{g}$; the dimension of $\mathbf{g}$ is denoted by $d_{\mathbf{g}}$.[1] In classical loop nest optimization frameworks, the basic unit of transformation is the loop; the polyhedral model increases expressiveness, applying transformations individually to each statement. For each statement S within a SCoP, we extract the following information:

- its depth d_S as the number of loops enclosing S;
- its iteration vector $\mathbf{i}^S$ as the vector of dimension d_S scanning execution instances of S;
- its iteration domain D^S as a matrix of $d_S + d_{\mathbf{g}}$ columns, bearing the affine inequalities defining all valid iteration vectors for S; formally, $\{\mathbf{i}^S \mid D^S \cdot (\mathbf{i}^S \mid \mathbf{g})^t \geq \mathbf{0}\}$;
- its schedule Θ^S as a matrix of size $(2d_S+1, d_S+d_{\mathbf{g}}+1)$ assigning a logical execution date to each execution instance $\mathbf{i}^S$ in D^S. By definition, the execution of statement iterations follows the lexicographic order on multidimensional execution dates;
- the set of all its memory references of the form $\langle \mathtt{X}, f^S(\mathbf{i}^S)\rangle$ where $\mathtt{X}$ is an array and f^S is its affine subscript function. We also consider more general non-affine references with conservative approximations.

Each iteration $\mathbf{i}^S$ of a statement S is called an *instance*, and is denoted by $\langle S, \mathbf{i}^S\rangle$.

Polyhedral compilation usually distinguishes between three steps: first, represent an input program in the formalism, then apply a transformation to this representation and finally generate the target (syntactic) code. It is well known that arbitrarily complex sequences of loop transformations can be captured in one single transformation step of the polyhedral model [27, 14]; this includes parallelism extraction and exploitation [12, 19, 15]. Yet to ease the composition of program transformations on the polyhedral representation, we further split the representation of the schedule Θ into smaller, interleaved, matrix and vector parts:

- the A^S part is a square matrix of size (d_S, d_S) and expresses the speed at which different statement instances execute along a given dimension;
- the Γ^S matrix of size (d_S, d_g) allows to perform multi-dimensional shifting with respect to global parameters;
- the β^S vector of size (d_S+1) encodes the syntactical, loop-independent interleaving of statements at every loop depth.

Such encodings with $2d_S+1$ dimensions were previously proposed by Feautrier [11], then by Kelly and Pugh [16]. We refer the reader to the URUK framework for a complete formalization and analysis [14]. For improved readability, we use a small example at depth 1 (i.e., domain dimension $d_{S_1} = d_{S_2} = 1$) and 1 parameter N (i.e., $d_{\mathbf{g}} = 1$).

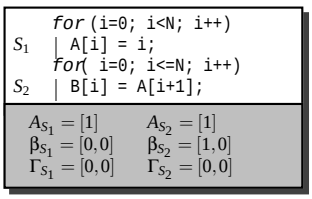

Figure 1. Original

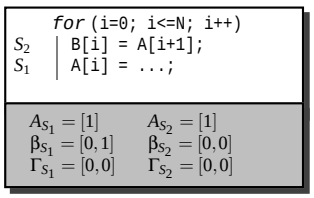

Figure 2. Fusion

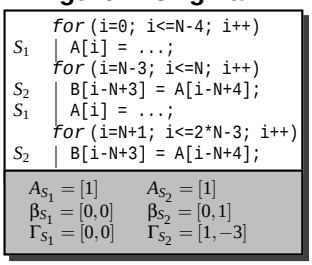

Figure 3. Shift

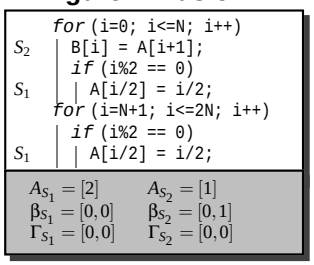

Figure 4. Slow

The original code is shown on Figure 1. Using our framework, it is natural to express transformations like fusion, fission and code motion by simple operations on the β vectors — Figure 2 — even for misaligned loops with different parametric bounds [25]. Shifting by a constant amount or by a parametric amount — Figure 3 — is equally simple with operations on the Γ. We also show the result of slowing S_1 by a factor 2, operating on A with respect to S_2 — Figure 4. It is the duty of the final code generation algorithm [24, 2] to reconstruct loop nests from affine schedules and iteration domains.

An important property is that modifications of β and Γ are commutative and trivially reversible. It is obvious that this does not hold in AST representations of the program. Typically, on Figure 3, one would need to rebuild the original statement S_1 from the prologue and kernel when working with an AST representation. Whereas in the polyhedral model, S_1 is still a single statement until the final regeneration of the code, which alleviates such annoying pattern-matching based reconstructions. This crucial observation is fundamental for defining a complete correction scheme, avoiding the traditional decision problems associated with the selection of enabling transformations.

[1] We insert the non-parametric constant as the last element of $\mathbf{g}$ and will not distinguish it through the remainder of the paper.

Dependence Analysis. Array dependence analysis is the starting point for any polyhedral optimization. It computes non transitively-redundant, iteration vector to iteration vector, directed dependences [9, 25]. In order to correct dependence violations, it is first needed to *compute the exact dependence information between every pair of instances*, i.e., every pair of statement iterations. Considering a pair of statements S and T accessing memory locations where at least one of them is a write, there is a dependence from an instance $\langle S,\mathbf{i}^S\rangle$ of S to an instance $\langle T,\mathbf{i}^T\rangle$ of T (or $\langle T,\mathbf{i}^T\rangle$ depends on $\langle S,\mathbf{i}^S\rangle$) if and only if the following *instance-wise* conditions are met: (1) both instances belong to the corresponding statement iteration domain: $D^S_{\mathbf{i}}\cdot\mathbf{i}^S\geq 0$ and $D^T_{\mathbf{i}}\cdot\mathbf{i}^T\geq 0$, (2) both instances refer to the same memory location: $f^S\cdot\mathbf{i}^S=f^T\cdot\mathbf{i}^T$, and (3) the instance $\langle S,\mathbf{i}^S\rangle$ is executed before $\langle T,\mathbf{i}^T\rangle$ in the original execution: $\Theta^S\cdot\mathbf{i}^S\ll\Theta^T\cdot\mathbf{i}^T$, where $\ll$ is the lexicographic order on vectors.

The multidimensional logical date at which an instance is executed is determined, for statement S, by the $2d_S+1$ vector given by $\Theta^S\cdot\mathbf{i}^S$. The schedule of statement instances is given by the lexicographic order of their schedule vectors and is the core of the code generation algorithm.

The purpose of dependence analysis is to compute a directed dependence multi-graph DG. Unlike traditional reduced dependence graphs, an arc $S\rightarrow T$ in DG is labeled by a polyhedron capturing the pairs of iteration vectors ($\mathbf{i}^S$, $\mathbf{i}^T$) in dependence.

Violated Dependences. After transforming the SCoP, the question arises whether the resulting program still executes correct code. Our approach consists in saving the dependence graph, before applying any transformation [25], then to apply a given transformation sequence, and eventually to run a legality analysis at the very end of the sequence.

We consider a dependence from S to T in the original code and we want to determine if it has been preserved in the transformed program.

Violated dependence analysis [25] efficiently computes the iterations of the Cartesian product space $D^S\times D^T$ that were in a dependence relation in the original program and *whose order has been reversed by the transformation*. These iterations, should they exist, do not preserve the causality of the original program. Let $\delta^{S\rightarrow T}$ denote the dependence polyhedron from S to T; we are looking for the exact set of iterations of $\delta^{S\rightarrow T}$ such that there is a dependence from T to S at transformed depth p. By reasoning in the transformed space, it is straightforward to see that the set of iterations that violate the causality condition is the intersection of a dependence polyhedron with the constraint set $\Theta^S\cdot\mathbf{i}^S\geq\Theta^T\cdot\mathbf{i}^T$.

We denote a violated dependence polyhedron at depth p by $\text{VIO}_p^{S\rightarrow T}$. We also define a slackness polyhedron at depth p which contains the set of points originally in dependence and that are still executed in correct order after transformation. Such a polyhedron will be referred to as $\text{SLA}_p^{S\rightarrow T}$.

4. Correction by Shifting

We propose a greedy algorithm to incrementally correct violated dependences, from the outermost to the innermost nesting depth. This algorithm addresses a similar problem as retiming, in an extended multidimensional case with parametric affine dependences [18, 6]. The basic idea is to correct violations via iterative translation of the schedules; these translations reflect into loop shifting for the case of loop-carried violated dependences and into loop fusion or distribution for the loop-independent case [25].

4.1. Violation and Slackness

At depth p, the question raises whether a subset of those iterations – whose dependence has not yet been resolved up to depth p – are in violation and must be corrected.

When correcting loop-carried violations, we define the following affine functions from $\mathbb{Z}^{d_S+d_T+d_{\mathbf{g}}+1}$ to $\mathbb{Z}^{d_{\mathbf{g}}+1}$:[2]

- $\Delta_{\text{VIO}}\Theta_p^{S\rightarrow T}=A^S_{p,\bullet}-A^T_{p,\bullet}+\Gamma^S_{p,\bullet}-\Gamma^T_{p,\bullet}$ which computes the amount of time units at depth p by which a specific instance of T executes before one of S.
- $\Delta_{\text{SLA}}\Theta_p^{S\rightarrow T}=-A^S_{p,\bullet}+A^T_{p,\bullet}-\Gamma^S_{p,\bullet}+\Gamma^T_{p,\bullet}$ which computes the amount of time units at depth p by which a specific instance of S executes before one of T.

Then, let us define the parametric extremal values of these two functions:

- $\text{SHIFT}^{S\rightarrow T}=\max\limits_{X\in D^S\times D^T}\left\{\Delta_{\text{VIO}}\Theta_p^{S\rightarrow T}\cdot X>0\mid X\in\text{VIO}_p^{S\rightarrow T}\right\}$
- $\text{SLACK}^{S\rightarrow T}=\min\limits_{X\in D^S\times D^T}\left\{\Delta_{\text{SLA}}\Theta_p^{S\rightarrow T}\cdot X\geq 0\mid X\in\text{SLA}_p^{S\rightarrow T}\right\}$

We use the parametric integer linear program solver PIP [10] to perform these computations. The result is a piecewise, quasi-affine function of the parameters (affine with additional parameters to encode modulo operations). It is characterized by a disjoint union of polyhedra where this function is quasi-affine. This piecewise affine function is encoded as a *parametric quasi-affine selection tree* – or *quast* – [10, 9].

The loop-independent case is much simpler. Each violated dependence must satisfy conditions on β:

- if $\text{VIO}_p^{S\rightarrow T}\neq\emptyset\wedge\beta_p^S>\beta_p^T$, $\text{SHIFT}^{S\rightarrow T}=\beta_p^S-\beta_p^T$
- if $\text{VIO}_p^{S\rightarrow T}\neq\emptyset\wedge\beta_p^S\leq\beta_p^T$, $\text{SLACK}^{S\rightarrow T}=\beta_p^S-\beta_p^T$

The correction problem can then be reformulated as finding a solution to a system of differential constraints on parametric quasts. For any statement S, we shall denote by c_S the

[2] We write $A_{p,\bullet}$ to express the p^{th} line and $A_{1..p-1,\bullet}$ for lines 1 to $p-1$.

unknown amount of correction: a piecewise, quasi-affine function; c_S is called the *shift amount* for statement S. The problem we need to solve becomes:

$$\forall (S,T) \in \text{SCoP}, \quad \begin{cases} c_T - c_S & \leq & -\text{SHIFT}^{S\to T} \\ c_T - c_S & \leq & \text{SLACK}^{S\to T} \end{cases}$$

Such a problem can be solved with a variant of the Bellman-Ford algorithm [4], with piecewise quasi-affine functions (quasts) labeling the edges of the graph. Parametric case distinction arises when considering addition and maximization of the shift amounts resulting from different dependence polyhedra. In addition, for correctness proofs of the Bellman-Ford algorithm to hold, triangular inequalities and transitivity of the $\leq$ operator on quasts must also hold. The algorithm in Figure 5 allows to maintain case disjunction while enforcing all the required properties for any configuration of the parameters. At each step of the separation algorithm, two cases are computed and tested for emptiness. The $\text{SHIFT}^{\text{COND}}$ predicate holds the linear constraints on the parameters for a given violation to occur. Step 15 implements the complementary check.[3] As an optimization, it is often possible to extend disjoint conditionals by continuity, which reduces the number of versions (associated with different parameter configurations), hence reduce complexity and the resulting code size. For example:

$$\left.\begin{array}{l}\texttt{if } (M=3) \texttt{ then } 2 \\ \texttt{else if } (M\geq 4) \texttt{ then } M-1\end{array}\right| \equiv \texttt{if } (M\geq 3) \texttt{ then } M-1$$

Experimentally, performing this post processing allows up to 20% less versioning at each correction depth. Given the multiplicative nature of duplications, this can translate into exponentially smaller generated code.

4.2. Constraints graphs

We outline the construction of the constraints graph used in the correction. For depth p, the violated dependence graph VDG_p is a directed multigraph where each node represents a statement in the SCoP. The construction of VDG_p proceeds as follows. For each violated polyhedron $\text{VIO}_p^{S\to T}$, the minimal necessary correction is computed and results in a parametric conditional and a shift amount. We add an edge, between S and T in VDG_p of type $\mathcal{V}$ (violation), decorated with the tuple $(\text{VIO}_p^{S\to T}, \text{SHIFT}^{\text{COND}}, -\text{SHIFT}^{S\to T})$. When $\mathcal{V}$ type arcs are considered, they bear the minimal shifting requirement by which T *must* be shifted for the corrected values of Θ^S and Θ^T to nullify the violated polyhedron $\text{VIO}_p^{S\to T}$. Notice however that shifting T by $\text{SHIFT}^{S\to T}$ amount does not fully solve the dependence problem. It solves it for the subset of points

$\left\{X \in \text{VIO}_p^{S\to T} \mid \Delta_{\text{VIO}}\Theta_p^{S\to T} < \text{SHIFT}^{S\to T}\right\}$. The remaining points - the facet of the polyhedron such that $\left\{X \in \text{VIO}_p^{S\to T} \mid \Delta_{\text{VIO}}\Theta_p^{S\to T} = \text{SHIFT}^{S\to T}\right\}$ - are carried for correction at the next depth and will eventually be solved at the innermost β level, as will be seen shortly.

[3] Unlike the more costly separation algorithm by Quilleré [23] used for code generation, this one only needs intersections (no complement or difference).

```
SeparateMinShifts: Eliminate redundancy in a list of shifts
Input:
  redundantlist: list of redundant shift amounts and conditions
Output: non redundant list of shift amounts and conditions
  resultinglist ← empty list
1 while(redundantlist not empty)
2   if(resultinglist is empty)
3     resultinglist.append(redundantlist.head)
4     redundantlist ← redundantlist.tail
5   else
6     tmplist ← empty list
7     SHIFT_1 ← redundantlist.head
8     redundantlist ← redundantlist.tail
9     while(resultinglist not empty)
10      SHIFT_2 ← resultinglist.head
11      resultinglist ← resultinglist.tail
12      cond1 ← SHIFT_1^COND ∧ SHIFT_2^COND ∧ (SHIFT_2 < SHIFT_1)
13      if(cond1 not empty)
14        tmplist.append(cond1, SHIFT_1)
15      cond2 ← SHIFT_1^COND ∧ SHIFT_2^COND ∧ (SHIFT_2 ≥ SHIFT_1)
16      if(cond2 not empty)
17        tmplist.append(cond2, SHIFT_2)
18      if(cond1 is empty and cond2 is empty)
19        tmplist.append(SHIFT_1^COND, SHIFT_1)
20        tmplist.append(SHIFT_2^COND, SHIFT_2)
21    resultinglist ← tmplist
22 return resultinglist
```

Figure 5. Conditional separation

For a loop-independent VDG_p, the correction is simply done by reordering the β_p values. No special computation is necessary as only the relative values of β_p^S and β_p^T are needed to determine the sequential order. When such a violated, it means the candidate dependence polyhedron $\text{VIO}_p^{S\to T}$ is not empty *and* $\beta_p^S > \beta_p^T$. The correction algorithm forces the synchronization of the loop independent schedules by setting $\beta_p^S = \beta_p^T$ and carries $\text{VIO}_p^{S\to T}$ to be corrected at depth $p+1$.

For each original dependence $\delta^{S\to T}$ in the DG that has not been completely solved up to current depth add an edge between S and T in VDG_p of type $\mathcal{S}$ (slackness). Such an edge is decorated with the tuple $(\text{VIO}_p^{S\to T}, \text{SLACK}^{\text{COND}}, \text{SLACK}^{S\to T})$. When $\mathcal{S}$ type edges are considered, they bear the maximal shifting allowed for S so that the causality relation $S \to T$ is ensured. If at any time a node is shifted by a quantity bigger than one of the maximal allowed outgoing slacks, it will give rise to new outgoing shift edges.

4.3. The Algorithm

The correction algorithm is interleaved with the incremental computation of the VDG at each depth level. The fundamental reason is that corrections at previous depths need to be taken into account when computing the violated dependence polyhedra at the current level. The main idea for depths $p > 0$ is to shift targets of violated dependences by the *minimal shifting amount necessary*. If any of those incoming shifts is bigger than any outgoing slack, the out-

```
CorrectLoopDependentNode: Corrects a node by shifting
Input:
  node: A node in VDG_p
Output: A list of parametric shift amounts and conditionals
for the node
  corrections ← corrections of node already
                          computed in previous passes
1 foreach(edge (S, node, Vi) incoming into node)
2   compute minimal SHIFT^{S→node} and SHIFT^{COND}
3   corrections.append(SHIFT^{S→node}, SHIFT^{COND})
4 if(corrections.size > 1)
5   corrections ← SeparateMinShifts(corrections)
6 foreach(edge (node, T) outgoing from node)
7   foreach(corr in corrections)
8     compute a new shift VIO_p^{node→T} using corr for node
9     if(VIO_p^{node→T} not empty)
10      addedge(node, T, ν, VIO_p^{node→T}) to VDG_p
11    else
12      compute a new slack SLA_p^{node→T} using corr for node
13      addedge(node, T, S, SLA_p^{node→T}) to VDG_p
14  removeedge(edge) from VDG_p
15return corrections
```

Figure 6. Shifting a node for correction

```
CorrectLoopDependent: Corrects a VDG by shifting
Input:
  VDG: A node in VDG_p
Output: A list of parametric shift amounts and conditionals
for the node
  corrections ← empty list
1 for(i = 1 to |V| − 1)
2   nodelist ← nodes(VDG) with incoming edge of type ν
3   foreach(node in nodelist)
4     corrections.append(CorrectLoopDependentNode(node))
5  return corrections
```

Figure 7. Correcting a VDG

```
CorrectSchedules: Corrects an illegal schedule
Input:
  program: A program in URUK form
  dependences: The list of polyhedral dependences of the program
Output: Corrected program in URUK form
1 Build VDG_0
2 correctionList ← CorrectLoopIndependent(VDG_0)
3 commit shifts in correctionList
4 for(p=1; p<=max_{(S ∈ SCoP)}{rank(β_S)})
5   Build VDG_p
6   correctionList ← CorrectLoopDependent(VDG_p)
7   commit shifts in correctionList
8   correctionList ← CorrectLoopIndependent(VDG_p)
9   commit shifts in correctionList
```

Figure 8. Correction algorithm

going slacks turn into new violations that need to be corrected. During the graph traversal, any node is seen at most $|V|-1$ times, where V is the set of vertices. At each traversal, we gather the previously computed corrections along with incoming violations and we apply the separation phase of Figure 5. For loop-carried dependences, the algorithm to correct a node is outlined in Figure 6.

We use an incrementally growing cache to speed up polyhedral computations, as proposed in [25]. Step 8 of Figure 6 uses PIP to compute the minimal incoming shift amount; it may introduce case distinctions, and since we label edges in the VDG with single polyhedra and not quasts, it may result in inserting new outgoing edges. When many incoming edges generate many outgoing edges, Step 11 separates these possibly redundant amounts using the algorithm formerly given in Figure 5. In practice, PIP can also introduce new modulo parameters for a certain class of ill-behaved schedules. These must be treated with special care as they will expand $d_{\mathbf{g}}$ and are generally a hint that the transformation will eventually generate code with many internal modulo conditionals. On the other hand, for loop-independent corrections all quantities are just integer differences, without any case distinction. The much simpler algorithm is just a special case.

The full algorithm recursively computes the current VDG for each depth p, taking into account previously corrected dependences and is outlined in Figure 8. Termination and soundness are straightforward, from those of the Bellman-Ford version [4] applied successively at each depth on the SHIFT amount.

Lemma 1 (Depth-p Completeness) *If shifting amounts satisfying the system of violation and slackness constraints at depth p exist, the correction algorithm removes all violations at depth p.*

The proof derives from the completeness of Bellman-Ford's algorithm. Determining the minimal correcting shift amounts to computing the maximal value of linear multivariate functions ($\Delta_{\mathrm{VIO}}\Theta_p^{S\to T}$ and $\Delta_{\mathrm{SLA}}\Theta_p^{S\to T}$) over bounded parametrized convex polyhedra ($\mathrm{VIO}_p^{S\to T}$ and $\mathrm{SLA}_p^{S\to T}$). This problem is solved in the realm of parametric integer linear programming. The separation algorithm ensures equality or incompatibility of the conditionals enclosing the different amounts. The resulting quasts therefore satisfy the transitivity of the operations of max, min, $+$ and $\leq$. When VDG_p has no negative weight cycle, the correction at depth p succeeds; the proof is the same as for the Bellman-Ford algorithm and can be found in [4].□

As mentioned earlier, the computed shift amounts are minimal such that the schedule at a given depth does not violate any dependence; full resolution of those dependences is carried for correction at the next level. Another solution would be to shift target statements by $\mathrm{SHIFT}^{S\to T}+1$, but this is deemed too intrusive. Indeed, this amounts to adding -1 to all negative edges on the graph, potentially making the correction impossible.

The question arises whether a shift amount chosen at a given depth may interfere with the correction algorithm at a higher depth. The next lemma guarantees it is not the case.

Lemma 2 *Correction at a given depth by the minimal shift amount does not hamper correction at a subsequent depth.*

For $p>1$, if VDG_p contains a negative weight cycle, VDG_{p-1} contains a null weighted slackness cycle traversing the same nodes. By construction, any violated edge at

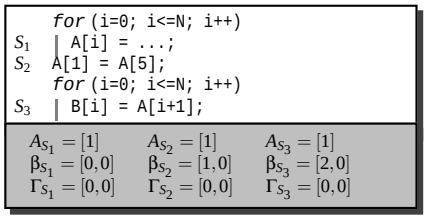

Figure 9. Original code

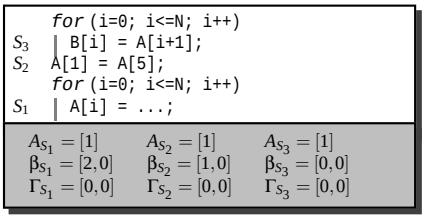

Figure 10. Illegal schedule

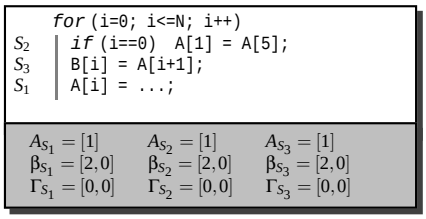

Figure 11. After correcting

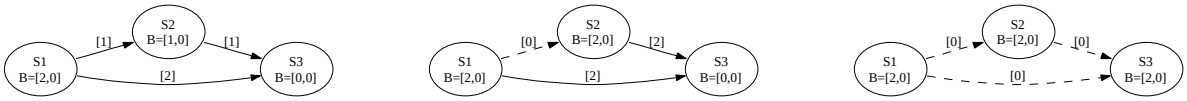

Figure 12. Outline of the correction for $p = 0$

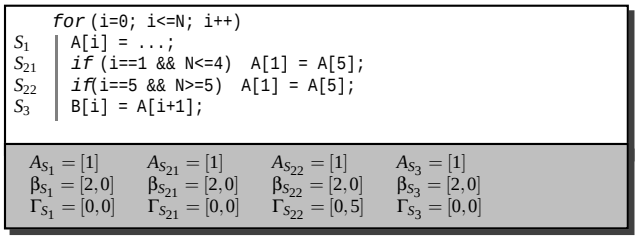

Figure 13. Correcting + versioning S_2

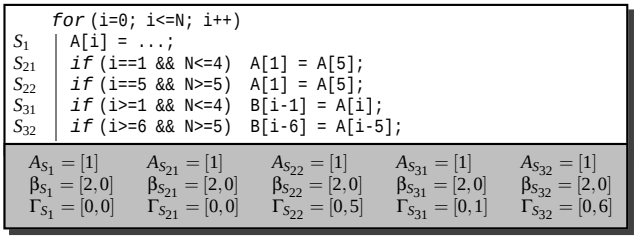

Figure 14. Correcting + versioning S_3

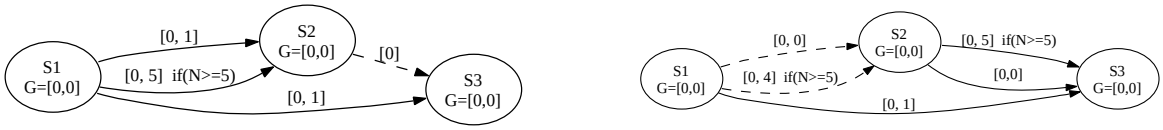

Figure 15. Outline of the correction for $p = 1$

depth p presupposes that the candidate violated polyhedron at previous depth $\text{VIO}_{p-1}^{S \to T}$ is not empty. Hence, for any violated edge at depth p, there exists a 0-slack edge at any previous depths thus ensuring the existence of a 0-slack cycle at any previous depths.□

In other words, the fact that a schedule cannot be corrected at a given depth is an intrinsic property of the schedule. Combined with the previous lemma, we deduce the completeness of our greedy algorithm.

Theorem 1 (Completeness) *If shifting amounts satisfying the system of violation and slackness constraints exist at all depths, the correction algorithm removes all violations.*

Let us outline the algorithm on the example of Figure 9, assuming $N \geq 2$. Nodes represent statements of the program and are labeled with their respective schedules (B for β and G for Γ). Dashed edges represent slackness while plain edges represent violations and are labeled with their constant or parametric amount. Suppose the chosen transformation tries to perform the modifications of the above statements' schedules according to the values in Figure 10.

The first pass of the correction algorithm for depth $p = 0$ detects the following loop independent violations: $S_1 \to S_2$, $S_1 \to S_2$ if $N \geq 5$, $S_1 \to S_3$, $S_2 \to S_3$ if $N \geq 5$. No slack edges are introduced in the initial graph. The violated dependence graph is a DAG and the correction mechanism will first push S_2 at the same β_0 as S_1. Then, after updating outgoing edges, it will do the same for S_3 yielding the code of Figure 11. Figure 12 shows the resulting VDG. So far, statement ordering within a given loop iteration is not fully specified (hence the statements are considered parallel); the code generation phase arbitrarily decides the shape of the final code. In addition, notice S_2 and S_3, initially located in different loop bodies than S_1, have been merged through the loop-independent step of the correction.

The next pass of the correction algorithm for depth $p = 1$ now detects the following loop carried violation: RAW dependence $\langle S_1, 5\rangle \to \langle S_2, 0\rangle$ is violated with the amount $5 - 0 = 5$ if $N \geq 5$, WAW dependence $\langle S_1, 1\rangle \to \langle S_2, 0\rangle$ is violated with the amount $1 - 0 = 1$, RAW dependence $\langle S_1, i+1\rangle \to \langle S_3, i\rangle$ is violated with the amount $i + 1 - i = 1$. There is also a 0-slack edge $S_2 \to S_3$ resulting from S_2 writing `A[1]` and S_3 reading it at iteration $i = 0$. All these information are stored in the violation polyhedron. A maximization step with PIP determines a minimal shift amount

of 5 if $N \geq 5$. Stopping the correction after shifting S_2 and versioning given the values of N would generate the intermediate result of Figure 13. However, since the versioning only happens at commit phases of the algorithm in Figure 8, the graph is given in Figure 15 and no node duplication is performed yet. The algorithm moves forward to correcting the node S_3 and, at step 3, the slack edge $S_2 \rightarrow S_3$ is updated with the new shift for S_2. The result is an incoming shift edge with violation amount 4 if $N \geq 6$; which yields versions of Figure 16 after code generation optimization.

5. Correction by Index-Set Splitting

Index-set splitting has originally been crafted as an enabling transformation. It is usually formulated as a decision problem to express more parallelism by allowing the construction of piecewise affine functions. Yet, the iterative method proposed by Feautrier et al. [13] relies on calls to a costly, non scalable, scheduling algorithm, and aims at exhibiting more parallelism by breaking cycles in the original dependence graph. However, significant portions of numerical programs do not exhibit such cycles, but still suffer from major inefficiencies; this is the case of the simplified excerpts from SPEC CPU2000fp benchmarks `swim` and `mgrid`, see Figures 17–18. Other methods fail to enable important optimizations in the presence of parallelization or fusion preventing dependences, or when loop bounds are not identical. Feautrier's index-set splitting heuristic aims at improving the expressiveness of affine schedules. In the context of schedule corrections, the added expressiveness helps our greedy algorithm to find less intrusive (i.e., deeper) shifts. Nevertheless, since not all schedules may be corrected by a combination of translation and index-set splitting, it is interesting to have a local necessary criterion to rule out impossible solutions.

5.1. Correction Feasibility

When a VDG contains a circuit with negative weight, correction by translation alone becomes infeasible. If no circuit exists in the DG, then no circuit can exist in any VDG, since by construction, edges of the VDG are built from edges in the DG. In this case, whatever A, β and Γ parts are chosen for any statement, a correction is found.

If the DG contains circuits, a transformation modifying β and Γ only is always correctable. Indeed, the reverse translation on β and Γ for all statements is a trivial solution. As a corollary, any combination of loop fusion, loop distribution and loop shifting [1] can be corrected. Thanks to this strong property of our algorithm, we often eliminate the decision problem of finding enabling transformations such as loop bounds alignment, loop shifting and peeling. This observation leads to a local necessary condition for ruling out non admissible correctable schedules.

Lemma 3 *Let $C = S \rightarrow S_1 \rightarrow \ldots \rightarrow S_n \rightarrow S$ be a circuit in the DG. By successive projections onto the image of every dependence polyhedron, we can incrementally construct a polyhedron $\delta^{S \rightarrow^P S_i}$ that contains all the instances of S and S_i that are transitively in dependence along the prefix P of the circuit. If $\delta^{S \rightarrow S}$ is not empty, the function $A^S_{p,\bullet} - A^S_{p,\bullet}$ must be positive for a correction to exist at depth p.*

Without this necessary property, index-set splitting would not enhance the expressiveness of affine schedules enough for a correction to be found (by loop shifting only). This is not sufficient to ensure the schedule can be corrected.

5.2. Index-Set Splitting for Correction

Our splitting heuristic aims at preserving asymptotic locality and ordering properties of original schedule while avoiding code explosion. The heuristic runs along with the shifting-based correction algorithm, by splitting only target nodes of violated dependences. Intuitively, we decompose the target domain when the two following criteria hold: (1) the amount of correction is "too intrusive" with respect to the original (illegal) schedule; (2) it allows a "significant part" of the target domain to be preserved. A correction is intrusive if it is a parametric shift (Γ) or a motion (β).

```
if (N<=4)
| A[0] = ...;
| A[1] = ...;
| A[1] = A[5];
| B[0] = A[1];
| for (i=2; i<=N; i++)
| | A[i] = ...;
| | B[i-1] = A[i];
| B[N-1] = A[N];
```

```
elseif (N>=5)
| for (i=0; i<=4; i++)
| | A[i] = ...;
| A[5] = ...;
| A[1] = A[5];
| for (i=6; i<=N; i++)
| | A[i] = ...;
| | B[i-6] = A[i-5];
| for (i=N+1; i<=N+6; i++)
| | B[i-6] = A[i-5];
```

Figure 16. Versioning after code generation

To assess the second criterion, we consider for every incoming shift edge, the projection of the polyhedron onto every non-parametric iterator dimension. A dimension whose projection is non-parametric and included in an interval smaller than a given constant (3 in our experiments) is called *degenerate*. In the following example, dimension j is degenerate and simplifies into $2i+2 \geq 6j \geq 2i-5$:

$$\left\{\begin{array}{rcrcrcrcl} -2i & + & 3j & + & 4M & + & 5 & \geq & 0 \\ i & + & j & - & M & + & 2 & \geq & 0 \\ i & - & 3j & + & 2M & - & 9 & \geq & 0 \\ -i & & & & & + & M & \geq & 0 \\ i & & & & & & & \geq & 0 \end{array}\right.$$

The separation and commit phases of our algorithm create as many disjoint versions of the correction as needed to enforce minimal shifts. The node duplications allow to express different schedules for different portions of the domain of each statement. To determine if a domain split should be performed, we incrementally remove violated

parts of the target domain corresponding to intrusive corrections until either: we run out of incoming violations, or the remaining part of the domain is degenerate. The intuition is to keep a non-degenerate core of the domain free of any parametric correction, to preserve locality properties of the original schedule. In the end, if the remaining portion of the domain still has the same dimension as the original one, a split is performed that separates the statement into the core that is not in violation and the rest of the domain.

Index set splitting is thus plugged into our correction algorithm as a preconditioning phase before step 4 of Figure 8. Notice that a statement is only split a finite number of times, since each incoming shift edge is split at most once at each depth. To limit the number of duplications, we allow only the core of a domain to be split among successive corrections. If a statement has already been decomposed at a given depth, only its core still exhibits the same locality properties as the original schedule. It is then unnecessary, and even harmful as far as code size is concerned, to further split the out-of-core part of the domain.

Original code in Figure 17 is a simplified version of one of the problems to solve when optimizing `mgrid`. The fusion of the first and third nests is clearly illegal since it would reverse the dependence from $\langle S_1, N-1\rangle$ to $\langle S_2\rangle$, as well as every dependence from $\langle S_1, i\rangle$ to $\langle S_3, i+1\rangle$. To enable this fusion, it is sufficient to shift the schedule of S_2 by $N-1$ iterations and to shift the schedule of S_3 by 1 iteration. Fortunately, only iteration $i=0$ of S_3 (after shifting by 1) is concerned by the violated dependence from S_2: peeling this iteration of S_3 gives rise to S_{3_1} and S_{3_2}. In turn, S_{3_1} is not concerned by the violation from S_2 while S_{3_2} must still be shifted by $N-1$ and eventually pops out of the loop. In the resulting optimized code in Figure 17, the locality benefits of the fusion are preserved and the legality is ensured.

A simplified version of the `swim` benchmark exhibits the need for a more complex split. The original code in Figure 18 features two doubly nested loops separated by an intermediate diagonal assignment loop, with poor temporal locality on array `A`. While allowing to maintain the order of the original schedule for a non-degenerate, triangular portion of S_1 and S_3 instances (i.e., $\{(i,j) \in [1,N] \mid j \neq i\}$); the optimized code in Figure 18 also exhibits much more reuse opportunities and yields better performance.

6. Experimental Results

The whole correction scheme (shifting, versioning and index-set splitting) was implemented in the URUK framework [14]. Our prototype tool was applied to real-world loop optimization problems, providing evidence of the benefits and scalability of our algorithms.

```
    for (i=0; i<N; i++)
S1  | A[i] = ...;
S2  A[0] = A[N-1];
    for (i=1; i<N; i++)
S3  | B[i] = A[i-1];
```

```
S1   A[0] = ...;
     for (i=1; i<N-1; i++)
S1   | A[i] = ...;
S3_1 | B[i+1] = A[i];
S1   A[N-1] = ...;
S2   A[0] = A[N-1];
S3_2 B[1] = A[0];
```

Figure 17. `mgrid`**-like (original and optimized)**

```
    for (i=0; i<N; i++)
    | for (j=0; j<N; j++)
S1  | | A[i][j] = ...;
    for (i=0; i<N; i++)
S2  | A[i][i] = ...;
    for (i=1; i<N; i++)
    | for (j=1; j<N; j++)
S3  | | B[i][j] = A[i][j];
```

```
    for (i=1; i<N; i++)
    | for (j=1; j<i-1; j++)
S1  | | A[i][j] = ...;
S3  | | B[i][j] = A[i][j];
S1  | A[i][i] = ...;
S2  | A[i][i] = ...;
S3  | B[i][i] = ...;
    | for (j=i+1; j<N; j++)
S1  | | A[i][j] = ...;
S3  | | B[i][j] = A[i][j];
```

Figure 18. `swim`**-like (original and optimized)**

Scalability Experiments. Our correction algorithm is applicable under many different scenarios (multidimensional affine schedules and dependence graphs), and we believe it is an important leap towards bridging the gap between the abstraction level of compact loop-based programs and their adaptation to modern architectures. To make its benefits more concrete, we apply it to one of the most important loop transformation for locality: loop fusion. It is often impeded by combinatorial decision problems such as shifting, index-set splitting and loop bounds alignment to remove fusion preventing edges. To give an intuition of the correction effort needed to exhibit unexploited locality, we study the case of aggressive loop fusion on the SPEC CPU2000fp programs `swim` and `mgrid`.

We start from inlined versions of the programs which represent 100% of the execution time for `swim` and 75% for `mgrid`. As a locality-enhancing heuristic, we try to apply loop fusion for all loops and at all loop levels. Since this transformation violates numerous dependences, our correction mechanism is applied on the resulting multidimensional affine schedules. Very simple examples derived from these experiments have been shown in Figures 17–18. Both loops exhibit "hot statements" with important amounts of locality to be exploited after fusion. As indicated in the Figure 19, `mgrid` has 3 hot statements in a 3-dimensional loop, but 12 statements are interleaved with these hot loops and exhibit dependences that prevent fusion at depth 2; `swim` exhibits 13 hot statements with 34 interleaved statements preventing fusion at depths 2 and 3.

Application of our greedy algorithm successfully results in the aggressive fusion of the compute cores, while inducing only small shifts. For `mgrid`, the hot statements once in different loop bodies – separated by distances $(4\times N,0,0)$ and $(8\times N,0,0)$ – are fused towards the innermost level with final translation vectors $(0,0,0)$ for the first, $(2,1,0)$

Program	mgrid	swim
# source statements	31	99
# corrected statements	47	138
# source code size	88	132
# corrected code size	542	447
# hot statements	3	13
# fusion preventing statements	12	34
# peel	12	20
# triangular splits	0	5
# original distance	(4N,0,0) (8N,0,0)	3·(0,3N,0) 6·(0,5N,0)
# final distance	(2,1,0) (3,3,0)	11·(0,0,0) (0,1,0) (0,0,1)

Figure 19. Correction Experiments

Benchmark	St.	Dep.	Dim.	All	Legal	Incomp.
compress-1024	6	56	2	6.2×10^{24}	6480	9
edge-2048	3	30	3	1.7×10^{24}	3.1×10^{7}	1467
latnrm-256	11	75	2	4.1×10^{18}	1.9×10^{9}	678
lmsfir-256	9	112	2	1.2×10^{19}	2.6×10^{9}	19962

Figure 20. Search Space Size

for the second and $(3,3,0)$ for the third one.[4] For swim, the original statements once in separate doubly nested loops have been fused thanks to an intricate combination of triangular index-set splitting, peeling of the boundary iterations, and shifting; 11 statements required no shifting, 1 required $(0,1,0)$ and the other $(0,0,1)$. Peeling and index-set splitting are required as to avoid loop distribution and are quantified in the 7^{th} and 8^{th} rows in the table. Overall, the number of extra statements introduced by index set splitting is about 20–30% which is quite reasonable.

No existing optimizing compiler is capable (up to our knowledge) of discovering the opportunity and applying such aggressive fusions. In addition, existing compilers deal with combinatorial decision problems associated with the selection of enabling transformations. All these decision problems disappear naturally with our correction scheme, in favor of more powerful heuristics that aim at limiting the amount of duplication in the resulting code while enforcing the compute intensive part of the program benefits from locality or parallelization improvements.

Iterative Optimization With Affine Schedules. Recent results by Pouchet et al. show the benefit of affine scheduling to construct and traverse a search space containing only distinct, legal transformations [21]. Despite orders of magnitude of reduction in the size of the search space, compared to state-of-the-art filtering approaches, the method still faces two major drawbacks: first, it is designed for one-dimensional schedules while its natural extension to the case of multidimensional ones leads to further explosion of the size of the search space second, the search space has many dimensions with little impact on performance [21] but exponential impact on search space size.

Finding a correction for a given schedule means computing a value for Γ and β, provided A. Figure 20 summarizes our results, showing dramatic reductions in the size of the search space on four kernels extracted from the UTDSP benchmark suite [17] (with schedule coefficients bounded in interval $[-1,1]$). The table shows: the number of *distinct* affine schedules (All, legal or not), of *distinct legal* affine schedules (Legal), and of *distinct legal incomplete* schedules *where a correction by shifting and fusion/distribution exists* (Incomp.). We also report the number of statements (St.), dependences (Dep.) and the schedule dimension (Dim.).

Our technique dramatically accelerates iterative, feedback directed optimizations by removing degrees of freedom shown to have a lower impact on performance, narrowing the exploration to the most representative subspaces. Overall, the thorough iterative search of all incomplete legal schedules of the above-mentioned UTDSP kernels takes a few minutes. The best automatically corrected transformations achieve speedups from 10% to 368% on an AMD Athlon64 2.4GHz.

Expert-Driven Semi-Automatic Optimization. When an expert performs assisted semi-automatic optimization , long sequences of transformations can be found that dramatically improve the execution speed. For example, to optimize the SPEC CPU2000fp swim benchmark, Girbal et al. [14] crafted a sequence of 40 transformations:

- 3 shifts along the outermost loop, determined by the expert as the only way to enable aggressive fusion on the innermost loops without resorting to complex index-set splitting (Figure 18);
- 2 aggressive nested fusion steps for locality;
- 8 multidimensional shifts aimed at enabling the previous fusions and making them legal;
- 10 statement motion steps with the same purpose;
- 12 loop peeling steps to isolate some non-fusable parts;
- 1 register blocking (or unroll-and-jam) step to improve memory locality (2 stripmine and 1 interchange);
- 2 loop unrolling transformations to improve ILP and help the compiler exhibit register reuse.

The result is a speedup of 38% on AMD Athlon64 w.r.t. peak performance obtained using the best available compiler and optimization flags at that time[5]

On this transformation sequence, only the nested fusion, register tiling and loop unrolling address the performance problem. Our correction algorithms were able to automatically discover the 30 multidimensional shifts and peeling steps necessary to enable the nested fusion and register tiling. Furthermore, when removing the 3 outermost

[4] Intuitively, a multidimensional distance of $(4 \times N, 0, 0)$ is equivalent to $4 \times N.N^2 + 0.N + 0$, assuming every loop has N iterations.

[5] -march=athlon64 -LNO:fusion=2:prefetch=2 -m64 -Ofast -msse2 -lmpath; pathf90 always outperformed Intel ICC by a small percentage.

shifts, our correction scheme triggers a very sophisticated index-set splitting, effectively enabling aggressive fusion, at the cost of some control-flow overhead (triangular iteration spaces). The speed-up drops to 15%, and adding register blocking and unrolling does not help because of increased control-flow overhead. On this example, the running time of the correction algorithm is only a few seconds.

7. Conclusion and Perspectives

We presented a general and complete algorithm to correct dependence violations on multidimensional affine schedules. This algorithm is based on loop shifting, a generalization of software pipelining that may also simulate the effect of loop fusion and distribution. We combined this algorithm with the first index-set splitting technique that operates after affine schedules are computed. The result is a very effective technique to dramatically reduce the complexity of loop nest optimization in the polyhedral model. We demonstrated its scalability on two real-world benchmarks, although previous affine scheduling algorithms would only consider much smaller kernels. Overall, we replaced the combinatorial decision problem of finding a sequence of enabling transformations, by an a posteriori tractable and controllable correction step.

References

[1] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan and Kaufman, 2002.

[2] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *Parallel Architectures and Compilation Techniques (PACT'04)*, Antibes, France, Sept. 2004.

[3] C. Bastoul and P. Feautrier. Adjusting a program transformation for legality. *Parallel processing letters*, 15(1):3–17, March 2005.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1989.

[5] J. B. Crop and D. K. Wilde. Scheduling structured systems. In *EuroPar'99*, LNCS, pages 409–412, Toulouse, France, Sept. 1999. Springer-Verlag.

[6] A. Darte and G. Huard. Loop shifting for loop parallelization. *Intl. J. of Parallel Programming*, 28(5):499–534, 2000.

[7] A. Darte, G.-A. Silber, and F. Vivien. Combining retiming and scheduling techniques for loop parallelization and loop tiling. *Parallel Processing Letters*, 7(4):379–392, 1997.

[8] S. Donadio, J. Brodman, T. Roeder, K. Yotov, D. Barthou, A. Cohen, M. Garzaran, D. Padua, and K. Pingali. A language for the compact representation of multiple program versions. In *Languages and Compilers for Parallel Computing (LCPC'05)*, LNCS, Hawthorne, New York, Oct. 2005. Springer-Verlag. 15 pages.

[9] P. Feautrier. Array expansion. In *ACM Intl. Conf. on Supercomputing*, pages 429–441, St. Malo, France, July 1988.

[10] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22:243–268, Sept. 1988.

[11] P. Feautrier. Some efficient solutions to the affine scheduling problem, part II, multidimensional time. *Intl. J. of Parallel Programming*, 21(6):389–420, Dec. 1992. See also Part I, one dimensional time, 21(5):315–348.

[12] P. Feautrier. Scalable and structured scheduling. *To appear at Intl. J. of Parallel Programming*, 28, 2006.

[13] P. Feautrier, M. Griebl, and C. Lengauer. On index set splitting. In *Parallel Architectures and Compilation Techniques (PACT'99)*, Newport Beach, CA, Oct. 1999. IEEE Computer Society.

[14] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Intl. J. of Parallel Programming*, 2006. Special issue on Microgrids. 57 pages.

[15] M. Griebl. Automatic parallelization of loop programs for distributed memory architectures. Habilitation thesis. Facultät für Mathematik und Informatik, Universität Passau, 2004.

[16] W. Kelly. Optimization within a unified transformation framework. Technical Report CS-TR-3725, University of Maryland, 1996.

[17] C. Lee and M. Stoodley. UTDSP benchmark suite, 1998. http://www.eecg.toronto.edu/ corinna/DSP/.

[18] C. E. Leiserson and J. B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1), 1991.

[19] A. W. Lim and M. S. Lam. Communication-free parallelization via affine transformations. In *24thACM Symp. on Principles of Programming Languages*, pages 201–214, Paris, France, jan 1997.

[20] K. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, july 1996.

[21] L.-N. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache. Iterative optimization in the polyhedral model: Part I, one-dimensional time. In *International Symposium on Code Generation and Optimization*, pages 144–156, San Jose, California, Mar. 2007. IEEE Comp. Soc.

[22] M. Püschel, B. Singer, J. Xiong, J. Moura, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson. SPIRAL: A generator for platform-adapted libraries of signal processing algorithms. *Journal of High Performance Computing and Applications, special issue on Automatic Performance Tuning*, 18(1):21–45, 2004.

[23] F. Quilleré and S. Rajopadhye. Optimizing memory usage in the polyhedral model. Technical Report 1228, IRISA, Université de Rennes, France, Jan. 1999.

[24] F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *Intl. J. of Parallel Programming*, 28(5):469–498, Oct. 2000.

[25] N. Vasilache, A. Cohen, C. Bastoul, and S. Girbal. Violated dependence analysis. In *ACM Intl. Conf. on Supercomputing (ICS'06)*, Cairns, Australia, June 2006.

[26] S. Verdoolaege, M. Bruynooghe, G. Janssens, and F. Catthoor. Multi-dimensional incremental loops fusion for data locality. In *ASAP*, pages 17–27, 2003.

[27] M. E. Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Stanford University, Aug. 1992. Published as CSL-TR-92-538.