International Journal of Parallel Programming, Vol. ??, No. ?, ??? 2006 (©2006)

Semi-Automatic Composition of Loop Transformations for Deep Parallelism and Memory Hierarchies

Sylvain Girbal,[1] Nicolas Vasilache,[1] Cédric Bastoul,[1] Albert Cohen,[1] David Parello,[2] Marc Sigler,[1] Olivier Temam[1]

[1] ALCHEMY Group, INRIA Futurs and LRI, Paris-Sud 11 University, France.
Email: `first.last@inria.fr`
[2] DALI Group, LP2A, University of Perpignan, France.
Email: `david.parello@univ-perp.fr`

Modern compilers are responsible for translating the idealistic operational semantics of the source program into a form that makes efficient use of a highly complex heterogeneous machine. Since optimization problems are associated with huge and unstructured search spaces, this combinational task is poorly achieved in general, resulting in weak scalability and disappointing sustained performance. We address this challenge by working on the program representation itself, using a semi-automatic optimization approach to demonstrate that current compilers offen suffer from unnecessary constraints and intricacies that can be avoided in a semantically richer transformation framework.

Technically, the purpose of this paper is threefold: (1) to show that syntactic code representations close to the operational semantics lead to rigid phase ordering and cumbersome expression of architecture-aware loop transformations, (2) to illustrate how complex transformation sequences may be needed to achieve significant performance benefits, (3) to facilitate the automatic search for program transformation sequences, improving on classical polyhedral representations to better support operation research strategies in a simpler, structured search space. The proposed framework relies on a unified polyhedral representation of loops and statements, using normalization rules to allow flexible and expressive transformation sequencing. This representation allows to extend the scalability of polyhedral dependence analysis, and to delay the (automatic) legality checks until the end of a transformation sequence. Our work leverages on algorithmic advances in polyhedral code generation and has been implemented in a modern research compiler.

KEY WORDS: Compiler optimization, semi-automatic program transformation, polyhedral model, automatic parallelization.

1. INTRODUCTION

Static compiler optimizations can hardly cope with the complex run-time behavior and hardware components interplay of modern processor architectures. Multiple architectural phenomena occur and interact simultaneously; this requires the optimizer to combine multiple program transformations. Recently, processor architectures have been shifting towards coarser grain on-chip parallelism, to avoid diminishing returns of further extending instruction-level parallelism; and this shift has not curbed the steady complexity increase of memory hierarchies and on-chip communication networks. Both incremental and breakthrough architecture designs for scalable on-chip parallelism build on the low delay and very-high bandwidth of on-chip communications and synchronizations. Current research proposals describe a wealth of fine-grain (instruction-level, vectors), mid-grain (transactions, micro-threads, frames in grid processors) and coarse-grain (threads, distributed agents) paradigms; these come with as many memory models or communication primitives, including inter-cluster registers, local memories (scratch-pads), multi-streaming DMA, networks on a chip, and of course, non-uniform (coherent) cache hierarchies.

Although the associated programming models are often explicitly parallel (threads, agents, data parallelism, vectors), they always rely on advanced compiler technology to relieve the programmer from scheduling and mapping the application, understanding the memory model and communication details. Even provided with enough static information or annotations (OpenMP directives, pointer aliasing, separate compilation assumptions), compilers have a hard time exploring the huge and unstructured search space associated with these lower level mapping and optimization challenges. Indeed, the task of the compiler can hardly been called optimization anymore, in the traditional meaning of lowering the abstraction penalty of a higher-level language. Together with the run-time system (whether implemented in software or hardware), the compiler is responsible for most of the combinatorial code generation decisions to map the simplified and idealistic operational semantics of the source program to the highly complex and heterogeneous machine.

Unfortunately, optimizing compilers have traditionally been limited to systematic and tedious tasks that are either not accessible to the programmer (e.g., instruction selection, register allocation) or that the programmer in a high level language does not want to deal with (e.g., constant propagation, partial redundancy elimination, dead-code elimination, control-flow optimizations). Generating efficient code for deep parallelism and deep memory hierarchies with complex and dynamic hardware components is a completely different story: the compiler (and run-time system) now has to take the burden of much smarter tasks, that only expert programmers would be able to carry. In a sense, it is not clear that these new optimization and parallelization tasks should be called “compilation” anymore. Iterative optimization and machine learning compilation [1–3] are part of the answer to these challenges, building on artificial intelligence and operation research know-how to assist compiler heuristic. Iterative optimization generalizes profile-directed approach to integrate precise feedback from the runtime behavior of the program into optimization algorithms, while machine learning approaches provide an automated framework to build new optimizers from empirical optimization data. However, considering the ability to perform complex transformations, or complex sequences of transformations [4,5], iterative optimization and machine learning compilation will fare no better than existing compilers on top of which they are currently implemented. In addition, any operation research algorithm will be highly sensitive to the structure of the search space it is traversing. E.g., genetic algorithms are known to cope with unstructured spaces but at a higher cost and lower scalability towards larger problems, as opposed to mathematical programming (e.g., semi-definite or linear programming) which benefit from strong regularity and algebraic properties of the optimization search space. Unfortunately, current compilers offer a very unstructured optimization search space. First of all, by imposing phase ordering constraints [6], they lack the ability to perform long sequences of transformations. In addition, compilers embed a large collection of ad-hoc program transformations, but they are *syntactic* transformations, i.e., control structures are regenerated after each program transformation, sometimes making it harder to apply the next transformations, especially when the application of program transformations relies on pattern-matching techniques.

Clearly, there is a need for a compiler infrastructure that can apply complex and possibly long compositions of optimizing or parallelizing transformations, in a rich, structured search space.

We claim that existing compilers are ill-equipped to address these challenges, because of improper program representations and inappropriate conditioning of the search space structure.

This article does (unfortunately) not present any original loop transformation or clever combination of static and dynamic analysis. Instead, it precisely addresses the lack of algebraic structure in traditional loop-nest optimizers, as a small step towards bridging the gap between peak and sustained performance in future and emerging on-chip multiprocessors. We present a framework to facilitate the search for *compositions* of program transformations; this framework relies on a unified representation of loops and statements, and has been introduced in [7]. This framework improves on classical polyhedral representations [8–13] to support a large array of useful and efficient program transformations (loop fusion, tiling, array forward substitution, statement reordering, software pipelining, array padding, etc.), as well as *compositions* of these transformations. Compared to the attempts at expressing a large array of program transformations as matrix operations within the polyhedral model [9, 14, 10], the distinctive asset of our representation lies in the simplicity of the formalism to compose non-unimodular transformations across long, flexible sequences. Existing formalisms have been designed for black-box optimization [8, 11, 12], and applying a classical loop transformation within them — as proposed in [9, 10, 3] — requires a syntactic form of the program to anchor the transformation to existing statements. Up to now, the easy composition of transformations was restricted to unimodular transformations [6], with some extensions to singular transformations [15].

The key to our approach is to clearly separate the four different types of actions performed by program transformations: modification of the iteration domain (loop bounds and strides), modification of the schedule of each individual statement, modification of the access functions (array subscripts), and modification of the data layout (array declarations). This separation makes it possible to provide a matrix representation for each kind of action, enabling the easy and independent composition of the different "actions" induced by program transformations, and as a result, enabling the composition of transformations themselves. Current representations of program transformations do not clearly separate these four types of actions; as a result, the implementation of certain compositions of program transformations can be complicated or even impossible. For instance, current implementations of loop fusion must include loop bounds and array subscript modifications even though they are only byproducts of a schedule-oriented program transformation; after applying loop fusion, target loops are often peeled, increasing code size and making further optimizations more complex. Within our representation, loop fusion is only expressed as a schedule transformation, and the modifications of the iteration domain and access functions are implicitly handled, so that the code complexity is exactly the same before and after fusion. Similarly, an iteration domain-oriented transformation like unrolling should have no impact on the schedule or data layout representations; or a data layout-oriented transformation like padding should have no impact on the schedule or iteration domain representations. Eventually, since all program transformations correspond to a set of matrix operations within our representation, searching for compositions of transformations is often (though not always) equivalent to testing different values of the matrices parameters, further facilitating the search for compositions. Besides, with this framework, it should also be possible to find and evaluate new sequences of transformations for which no static model has yet been developed (e.g., array forward substitution versus loop fusion as a temporal locality optimization).

This article is organized as follows. Section 2 illustrates with a simple example the limitations of syntactic representations for transformation composition, it presents our polyhedral representation and how it can circumvent these limitations. Revisiting classical loop transformations for automatic parallelization and locality enhancement, Section 3 generalizes their definitions in our framework, extending their applicability scope, abstracting away most syntactic limitations to transformation composition, and facilitating the search for compositions of transformations. Using several SPEC benchmarks, Section 4 shows that complex compositions can be necessary to reach high performance and how such compositions are easily implemented using our polyhedral representation. Section 5 describes the implementation of our representation, of the associated transformation tool, and of the code generation technique (in Open64/ORC (16)). Section 6 validates these tools through the evaluation of a dedicated transformation sequence for one benchmark. Section 7 presents related works.

2. A NEW POLYHEDRAL PROGRAM REPRESENTATION

The purpose of Section 2.1 is to illustrate the limitations of the implementation of program transformations in current compilers, using a simple example. Section 2.2 is a gentle introduction to polyhedral representations and transformations. In Section 2.3, we present our polyhedral representation, in Section 2.4 how it alleviates syntactic limitations and Section 2.5 presents normalization rules for the representation.

Generally speaking, the main asset of our polyhedral representation is that it is semantics-based, abstracting away many implementation artifacts of syntax-based representations, and allowing the definition of most loop transformations without reference to any syntactic form of the program.

2.1. Limitations of Syntactic Transformations

In current compilers, after applying a program transformation to a code section, a new version of the code section is generated, using abstract syntax trees, three address code, SSA graphs, etc. We use the term *syntactic* (or syntax-based) to refer to such transformation models. Note that this behavior is also shared by all previous matrix- or polyhedra-based frameworks.

2.1.1. Code size and complexity

As a result, after multiple transformations the code size and complexity can dramatically increase.

```
     for (i=0; i<M; i++)
S1   | Z[i] = 0;
     | for (j=0; j<N; j++)
S2   | | Z[i] += (A[i][j] + B[j][i]) * X[j];
     for (k=0; k<P; k++)
     | for (l=0; l<Q; l++)
S3   | | Z[k] += A[k][l] * Y[l];
```

Fig. 1. Introductory example

	Syntactic (#lines)		Polyhedral (#values)	
Original code	11		78	
Outer loop fusion	44	(×4.0)	78	(×1.0)
Inner loop fusion	132	(×12.0)	78	(×1.0)
Fission	123	(×11.2)	78	(×1.0)
Strip-Mine	350	(×31.8)	122	(×1.5)
Strip-Mine	407	(×37.0)	182	(×2.3)
Interchange	455	(×41.4)	182	(×2.3)

Fig. 2. Code size versus representation size

	Original	KAP	Double Fusion	Full Sequence
Time (s)	26.00	12.68	19.00	7.38

Fig. 3. Execution time

Consider the simple synthetic example of Figure 1, where it is profitable to merge loops i, k (the new loop is named i), and then loops j, l (the new loop is named j), to reduce the locality distance of array `A`, and then to tile loops i and j to exploit the spatial and TLB locality of array `B`, which is accessed column-wise. In order to perform all these transformations, the following actions are necessary: merge loops i, k, then merge loops j, l, then split statement `Z[i]=0` outside the i loop to enable tiling, then strip-mine loop j, then strip-mine loop i and then interchange i and jj (the loop generated from the strip-mining of j).

Because the i and j loops have different bounds, the merging and strip-mining steps will progressively multiply the number of loop nests versions, each with a different guard. After all these transformations, the program contains multiple instances of the code section shown in Figure 4. The number of program statements after each step is indicated in Figure 2.

```
...;
if ((M >= P+1) && (N == Q) && (P >= 63))
| for (ii=0; ii<P-63; ii+=64)
| | for (jj=0; jj<Q; jj+=64)
| | | for (i=ii; i<ii+63; i++)
| | | | for (j=jj; j<min(Q,jj+63); j++)
| | | | | Z[i] += (A[i][j] + B[j][i]) * X[j];
| | | | | Z[i] += A[i][j] * Y[j];
| for (ii=P-62; ii<P; ii+=64)
| | for (jj=0; jj<Q; jj+=64)
| | | for (i=ii; i<P; i++)
| | | | for (j=jj; j<min(Q,jj+63); j++)
| | | | | Z[i] += (A[i][j] + B[j][i]) * X[j];
| | | | | Z[i] += A[i][j] * Y[j];
| | | for (i=P+1; i<min(ii+63,M); i++)
| | | | for (j=jj; j<min(N,jj+63); j++)
| | | | | Z[i] += (A[i][j] + B[j][i]) * X[j];
| for (ii=P+1; ii<M; ii+=64)
| | for (jj=0; jj<N; jj+=64)
| | | for (i=ii; i<min(ii+63,M); i++)
| | | | for (j=jj; j<min(N,jj+63); j++)
| | | | | Z[i] += (A[i][j] + B[j][i]) * X[j];
...;
```

Fig. 4. Versioning after outer loop fusion

In our framework, the final generated code will be similarly complicated, but this complexity does not show until code generation, and thus, it does not hamper program transformations. The polyhedral program representation consists in a fixed number of matrices associated with each statement, and neither its complexity nor its size vary significantly, independently of the number and nature of program transformations. The number of statements remains the same (until the code is generated), only some matrix dimensions may increase slightly, see Figure 2. Note that the more complex the code, the higher the difference: for instance, if the second loop is triangular, i.e., `(j=0; j<i; j++)`, the final number of source lines of the syntactic version is 34153, while the size of the polyhedral representation is unchanged (same number of statements and matrix dimensions).

2.1.2. *Breaking patterns*

Compilers look for transformation opportunities using pattern-matching rules. This approach is fairly fragile, especially in the context of complex compositions, because previous transformations may break target patterns for further ones. Interestingly, this weakness is confirmed by the historical evolution of the SPEC CPU benchmarks themselves, partly driven by the need to avoid pattern-matching attacks from commercial compilers (17).

To illustrate this point we have attempted to perform the above program transformations targeting the Alpha 21364 EV7, using KAP C (V4.1) (18), one of the best production preprocessors available (source to source loop and array transformations). Figure 3 shows the performance achieved by KAP and by the main steps of the above sequence of transformations (fusion of the outer and inner loops, then tiling) on the synthetic example.[3] We found that KAP could perform almost none of the above transformations because pattern-matching rules were often too limited. Even though we did not have access to the KAP source code, we have reverse-engineered these limitations by modifying the example source code until KAP could perform the appropriate transformation; KAP limitations are deduced from the required simplifications. In Section 2.4, we will show how these limitations are overridden by the polyhedral representation.

The first step in the transformation sequence is the fusion of external loops i, k: we found that KAP only attempts to merge perfectly nested loops with matching bounds (i.e., apparently due to additional conditions of KAP's fusion pattern); after changing the loop bounds and splitting out `Z[i]=0`, KAP could merge loops i, k. In the polyhedral representation, fusion is only impeded by semantic limitations, such as dependences; non-matching bounds or non-perfectly nested loops are not an issue, or more exactly, these artificial issues simply disappear, see Section 2.4. After enabling the fusion of external loops i, k, the second step is the fusion of internal loops j, l. Merging loops j, l changes the ordering of assignments to `Z[i]`. KAP refuses to perform this modification (apparently another condition of KAP's fusion pattern); after renaming `Z` in the second loop and later accumulating on both arrays, KAP could perform the second fusion.

Overall, we found out that KAP was unable to perform these two transformations, mostly because of pattern-matching limitations that do not exist in the polyhedral representation. We performed additional experiments on other modern loop restructuring compilers, such as Intel Electron (IA64), Open64/ORC (IA64) and EKOPath (IA32, AMD64, EM64T), and we found similar pattern-matching limitations.

2.1.3. Flexible and complex transformation composition

Compilers come with an ordered set of phases, each phase running some dedicated optimizations and analyses. This phase ordering has a major drawback: it prevents transformations from being applied several times,

[3] Parameters M, N, P and Q are the bounds of the 400MB matrices `A` and `B`.

after some other *enabling* transformation has modified the applicability or adequation of further optimizations. Moreover, optimizers have rather rigid optimization strategies that hamper the exploration of potentially useful transformations.

Consider again the example of Figure 1. As explained above, KAP was unable to split statement `Z[i]` by itself, even though the last step in our optimization sequence — tiling j after fusions — cannot be performed without that preliminary action. KAP's documentation (18) shows that fission and fusion are performed together (and possibly repeatedly) at a given step in KAP's optimization sequence. So while fission could be a potentially enabling transformation for fusion (though it failed in our case for the reasons outlined in the previous paragraph), it is not identified as an enabling transformation for tiling in KAP's strategy, and it would always fail to split to enable tiling.

Moreover, even after splitting `Z[i]` and merging loops i, k and j, l, KAP proved unable to tile loop j; it is probably focusing on scalar promotion and performs unroll-and-jam instead, yielding a peak performance of $12.35s$. However, in our transformation sequence, execution time decreases from $26.00s$ to $19.00s$ with fusion and fission, while it further decreases to $7.38s$ thanks to tiling. Notice that both fusion and tiling are important performance-wise.

So KAP suffers from a too rigid optimization strategy, and this example outlines that, in order to reach high performance, a flexible composition of program transformations is a key element. In Section 4, we will show that, for one loop nest, up to 23 program transformations are necessary to outperform peak SPEC performance.

2.1.4. Limitations of phase ordering

To better understand the interplay of loop peeling, loop fusion, scalar promotion and dead-code elimination, let us now consider the simpler example of Figure 5. The three loops can be fused to improve temporal locality, and assuming `A` is a local array not used outside the code fragment, it can be replaced with a scalar `a`. Figure 6 shows the corresponding optimized code. Both figures also show a graphical representation of the different domains, schedules and access functions for the three statements A, B and C of the original and optimized versions. Notice the middle loop in Figure 5 has a reduced domain. These optimizations mainly consist in loop fusions which only have an impact on scheduling, the last iteration (99) in the domain of A was removed (dead code) and the access function to array `A` disappeared (scalar promotion).

```
  B[1] = 0
  for (i=0; i<100; i++)
A | A[i] = ...;
  for (i=0; i<99; i++)
B | B[i+1] = A[i] ...;
  for (i=0; i<100; i++)
C | C[i] = B[i] ...;
```

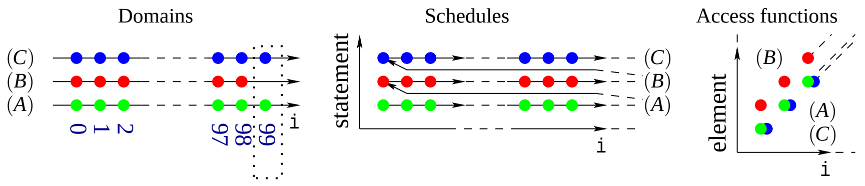


Fig. 5. Original program and graphical view of its polyhedral representation

```
  B[1] = 0
  for (i=0; i<99; i++)
A | a = ...;
B | B[i+1] = a ...;
C | C[i] = ...;
  C[100] = B[100] ...;
```

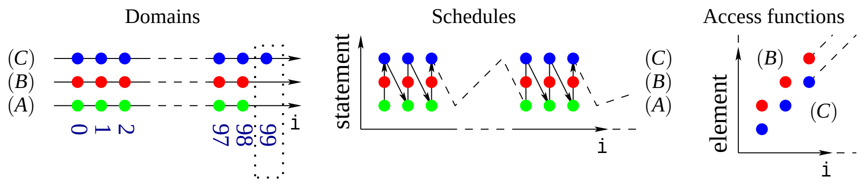


Fig. 6. Target optimized program and graphical view

Again, we tried to optimize this example using KAP, assuming that `A` is a global array, effectively restricting ourselves to peeling and fusion.

The reduced domain of *B* has no impact on our framework, which succeeds in fusing the three loops and yields the code in Figure 7. However, to fuse those loops, syntactic transformation frameworks require some iterations of the first and third loop to be peeled and interleaved between the loops. Traditional compilers are able to peel the last iteration and fuse the first two loops, as shown in Figure 8. Now, because their pattern for loop fusion only matches consecutive loops, peeling prevents fusion with the third loop, as shown in Figure 8; we checked that neither a failed de-

```
B[1] = 0
for (i=0; i<99; i++)
| A[i] = ...;
| B[i+1] = A[i] ...;
| C[i] = B[i] ...;
A[100] = ...;
C[100] = B[100] ...;
```

Fig. 7. Fusion of the three loops

```
B[1] = 0
for (i=0; i<99; i++)
| A[i] = ...;
| B[i+1] = A[i] ...;
A[100] = ...;
for (i=0; i<100; i++)
| C[i] = B[i] ...;
```

Fig. 8. Peeling prevents fusion

```
B[1] = 0
for (i=0; i<99; i++)
| a = ...;
| A[i] = a
| B[i+1] = a ...;
for (i=0; i<100; i++)
| C[i] = B[i] ...;
```

Fig. 9. Dead code before fusion

```
B[1] = 0
for (i=0; i<99; i++)
| a = ...;
| B[i+1] = a ...;
for (i=0; i<100; i++)
| C[i] = B[i] ...;
```

Fig. 10. Fusion before dead code

```
   B[1] = 0
   for (i=0; i<100; i++)
A   | A[i] = A[1] ...;
   for (i=0; i<99; i++)
B   | B[i+1] = A[i] ...;
   for (i=0; i<100; i++)
C   | C[i] = A[i]+B[i] ...;
```

Fig. 11. Advanced example

```
B[1] = 0
for (i=0; i<99; i++)
| A[i] = A[1] ...;
| B[i+1] = A[i] ...;
| C[i] = B[i] ...;
A[100] = A[1] ...;
C[100] = A[100]+B[100] ...;
```

Fig. 12. Fusion of the three loops

```
B[1] = 0
for (i=0; i<99; i++)
| A[i] = A[1] ...;
| B[i+1] = A[i] ...;
A[100] = A[1] ...;
for (i=0; i<100; i++)
| C[i] = A[i]+B[i] ...;
```

Fig. 13. Spurious dependences

pendence test nor an erroneous evaluation in the cost model may have caused the problem. Within our transformation framework, it is possible to fuse loops with different domains without prior peeling transformations because hoisting of control structures is delayed until code generation.

Pattern matching is not the only limitation to transformation composition. Consider the example of Figure 11 which adds two references to the original program, `A[1]` in statement *A* and `A[i]` in statement *C*. These references do not compromise the ability to fuse the three loops, as shown in Figure 12. Optimizers based on more advanced rewriting systems (19) and most non-syntactic representations (10, 20, 3) will still peel an iteration of the first and last loops. However, peeling the last iteration of the first loop introduces two dependences that prevent fusion with the third loop: backward motion — flow dependence on `A[1]` — and forward motion — anti-dependence on `A[i]` — of the peeled iteration is now illegal. KAP yields the partially fused code in Figure 13, whereas our framework may still fuse the three loops as in Figure 12.

To address the composition issue, compilers come with an ordered set of phases. This approach is legitimate but prevents transformations to be applied several times, e.g., after some other transformation has modified the appropriateness of further optimizations. We consider again the example of Figure 5, and we now assume `A` is a local array only used to compute `B`. KAP applies dead-code elimination before fusion: it tries to eliminate `A`, but since it is used to compute `B`, it fails. Then the compiler fuses the two loops, and scalar promotion replaces `A` with a scalar, as shown in Figure 9. It is now obvious that array `A` can be eliminated but dead-code elimination will not be run again. Conversely, if we delayed dead-code elimination until after loop fusion (and peeling), we would still not fuse with the third loop but we would eliminate `A` as well as the peeled iteration, as shown in Figure 10. Clearly, both phase orderings lead to sub-optimal results. However, if we compile the code from Figure 9 with KAP — as if we applied the KAP sequence of transformations twice — array `A` and the peeled iteration are eliminated, allowing the compiler to fuse the three loops, eventually reaching the target optimized program of Figure 6.

These simple examples illustrate the artificial restrictions to transformation composition and the consequences on permuting or repeating transformations in current syntactic compilers. *Beyond parameter tuning, existing compilation infrastructures may not be very appropriate for iterative compilation*. By design, it is hard to modify either phase ordering or selection, and it is even harder to get any transformation pattern to match a significant part of the code after a long sequence of transformations.

2.2. Introduction to the Polyhedral Model

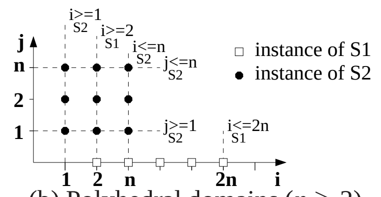
This section is a quick overview of the polyhedral framework; it also presents notations used throughout the paper. A more formal presentation of the model may be found in (14, 8). Polyhedral compilation usually distinguishes three steps: one first has to represent an input program in the formalism, then apply a transformation to this representation, and finally generate the target (syntactic) code.

Consider the polynomial multiplication kernel in Figure 14(a). It only deals with control aspects of the program, and we refer to the two computational statements (array assignments) through their names, S_1 and S_2. To bypass the limitations of syntactic representations, the polyhedral model is closer to the execution itself by considering *statement instances*. For each statement we consider the *iteration domain*, where every statement instance belongs. The domains are described using affine constraints that can be extracted from the program control. For example, the iteration domain of statement S_1, called $\mathcal{D}_{\text{om}}^{S_1}$, is the set of values (i) such that $2 \le i \le n$ as shown in Figure 14(b); a matrix representation is used to represent such constraints: in our example, $\mathcal{D}_{\text{om}}^{S_1}$ is characterized by

$$\begin{bmatrix} 1 & 0 & -2 \\ -1 & 2 & 0 \end{bmatrix} \begin{pmatrix} i \\ n \\ 1 \end{pmatrix} \ge \mathbf{0}.$$

```
     for (i=2; i<=2*n; i++)
S1   | Z[i] = 0;
     for (i=1; i<=n; i++)
     | for (j=1; j<=n; j++)
S2   | | Z[i+j] += X[i] * Y[j];
```

(a) Syntactic form



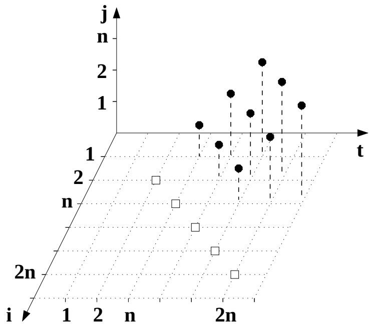
(b) Polyhedral domains ($n \ge 2$)

Fig. 14. A polynomial multiplication kernel and its polyhedral domains

In this framework, a transformation is a set of *affine scheduling functions*. Each statement has its own scheduling function which maps each run-time statement instance to a logical execution date. In our polynomial multiplication example, an optimizer may notice a locality problem and discover a good data reuse potential over array Z, then suggest $\theta^{S_1}(i) = (i)$ and $\theta^{S_2}\begin{pmatrix} i \\ j \end{pmatrix} = (i+j+1)$ to achieve better locality (see e.g., (21) for a method to compute such functions). The intuition behind such transformation is to execute consecutively the instances of S_2 having the same $i+j$ value (thus accessing the same array element of Z) and to ensure that the initialization of each element is executed by S_1 just before the first instance of S_2 referring this element. In the polyhedral model, a transformation is applied following the template formula in Figure 15(a) (22), where $\mathbf{i}$ is the iteration vector, $\mathbf{i}_{gp}$ is the vector of constant parameters, and $\mathbf{t}$ is the *time-vector,* i.e. the vector of the scheduling dimensions. The next section will detail the nature of these vectors and the structure of the Θ and Λ matrices. Notice in this formula, equality constraints capture schedule modifications, and inequality constraints capture iteration domain modifications. The resulting polyhedra for our example are shown in Figure 15(b), with the additional dimension t.

$$\left(\begin{array}{c|c} \mathrm{I} & \Theta \\ \hline 0 & \Lambda \end{array}\right) \cdot \left(\begin{array}{c} -\mathbf{t} \\ \hline \mathbf{i} \\ \mathbf{i}_{gp} \\ 1 \end{array}\right) \begin{array}{c} = \mathbf{0} \\ \hline \geq \mathbf{0} \end{array}$$

(a) Transformation template formula

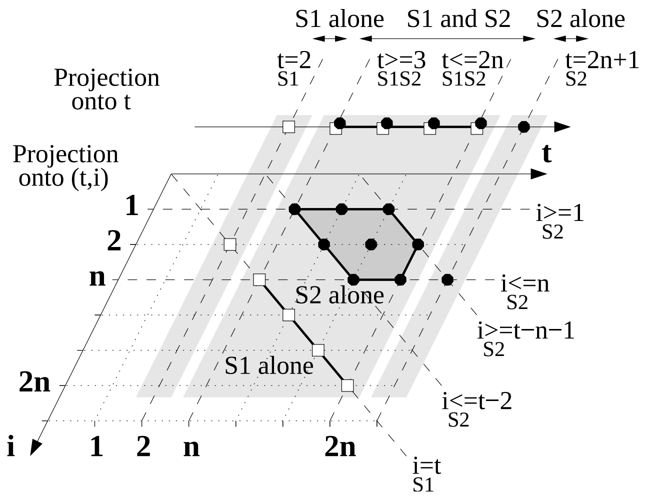


(b) Transformed polyhedra

Fig. 15. Transformation template and its application

Once transformations have been applied in the polyhedral model, one needs to (re)generate the target code. The best syntax tree construction scheme consists in a recursive application of domain projections and separations [23, 22]. The final code is deduced from the set of constraints describing the polyhedra attached to each node in the tree. In our example, the first step is a projection onto the first dimension t, followed by a separation into disjoint polyhedra, as shown on the top of Figure 16(a). This builds the outer loops of the target code (the loops with iterator t in Figure 16(b)). The same process is applied onto the first two dimensions (bottom of Figure 16(a)) to build the second loop level and so on. The final code is shown in Figure 16(b) (the reader may care to verify that this solution maximally exploits temporal reuse of array Z). Note that the separation step for two polyhedra needs three operations: $\mathcal{D}^{S_1}_{\mathrm{om}} - \mathcal{D}^{S_2}_{\mathrm{om}}$, $\mathcal{D}^{S_2}_{\mathrm{om}} - \mathcal{D}^{S_1}_{\mathrm{om}}$ and $\mathcal{D}^{S_2}_{\mathrm{om}} \cap \mathcal{D}^{S_1}_{\mathrm{om}}$, thus for n statements the worst-case complexity is 3^n.

It is interesting to note that the target code, although obtained after only one transformation step, is quite different from the original loop nest. Indeed, multiple classical loop transformations are be necessary to simulate this one-step optimization (among them, software pipelining and skewing). The intuition is that arbitrarily complex compositions of classical transfor-



(a) Projections an separations

```
     t=2; // Such equality is a loop running once
     | i=2;
S1   | | Z[i] = 0;
     for (t=3; t<=2*n; t++)
     | for (i=max(1,t-n-1); i<=min(t-2,n); i++)
     | | j = t-i-1;
S2   | | Z[i+j] += X[i] * Y[j]
     | i=t;
S1   | | Z[i] = 0;
     t=2*n+1;
     | i=n;
     | | j=n;
S2   | | | Z[i+j] += X[i] * Y[j];
```

(b) Target code

Fig. 16. Target code generation

mations can be captured in one single transformation step of the polyhedral model. This was best illustrated by affine scheduling [8, 10] and partitioning [11] algorithms. Yet, because black-box, model-based optimizers fail on modern processors, we propose to step back a little bit *and consider again the benefits of composing classical loop transformations, but using a polyhedral representation*. Indeed, up to now, polyhedral optimization frameworks have only considered the isolated application of one arbitrarily complex affine transformation. The main originality of our work is to address the *composition of program transformations on the polyhedral representation itself*. The next section presents the main ideas allowing to define compositions of affine transformations without intermediate code generation steps.

2.3. Isolating Transformations Effects

Let us now explain how our framework can separately and independently represent the iteration domain, the statements schedule, the data layout and the access functions of array references. At the same time, we will outline why this representation has several benefits for the implementation of program transformations: (1) it is generic and can serve to implement a large array of program transformations, (2) it isolates the root effects of program transformations, (3) it allows generalized versions of classical loop transformations to be defined without reference to any syntactic code, (4) this enables transparent composition of program transformations because applying program transformations has no effect on the representation complexity that makes it less generic or harder to manipulate, (5) and this eventually adds structure (commutativity, confluence, linearity) to the optimization search space.

2.3.1. Principles

The scope of our representation is a sequence of loop nests with constant strides and affine bounds. It includes non-rectangular loops, non-perfectly nested loops, and conditionals with boolean expressions of affine inequalities. Loop nests fulfilling these hypotheses are amenable to a representation in the polyhedral model (24). We call *Static Control Part* (SCoP) any *maximal syntactic program segment* satisfying these constraints (25). In this paper, we only describe analyses and transformations confined within a given SCoP; the reader interested in techniques to extend SCoP coverage (by preliminary transformations) or in partial solutions on how to remove this scoping limitation (procedure abstractions, irregular control structures, etc.) should refer to (26–35).

All variables that are invariant within a SCoP are called *global parameters*; e.g., M, N, P and Q are the global parameters of the introductory example (see Figure 1). For each statement within a SCoP, the representation separates four attributes, characterized by parameter matrices: the iteration domain, the schedule, the data layout and the access functions. Even though transformations can still be applied to loops or full procedures, they are individually applied to each statement.

2.3.2. Iteration domains

Strip-mining and loop unrolling only modify the iteration domain — the number of loops or the loop bounds — but they do not affect the order in which statement instances are executed (the program schedule) or the way arrays are accessed (the memory access functions). To isolate the effect of such transformations, we define a representation of the iteration domain.

Although the introductory example contains 4 loops, i, j, k and l, S_2 and S_3 have a different two-dimensional iteration domain. Let us consider the iteration domain of statement S_2; it is defined as follows: $\{(i,j) \mid 0 \leq i, i \leq M-1, 0 \leq j, j \leq N-1\}$. The iteration domain matrix has one column for each iterator and each global parameter, here respectively i, j and M, N, P, Q. Therefore, the actual matrix representation of statement S_2 is

$$\begin{array}{c} \begin{array}{cc|cccc|c} i & j & M & N & P & Q & 1 \end{array} \\ \left[\begin{array}{cc|cccc|c} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 & 0 & -1 \end{array}\right] \begin{array}{l} 0 \leq i \\ i \leq M-1 \\ 0 \leq j \\ j \leq N-1 \end{array} \end{array}$$

Example: implementing strip-mining All program transformations that only modify the iteration domain can now be expressed as a set of elementary operations on matrices (adding/removing rows/columns, and/or modifying the values of matrix parameters). For instance, let us strip-mine loop j by a factor B (a statically known integer), and let us consider the impact of this operation on the representation of the iteration domain of statement S_2.

Two loop modifications are performed: loop jj is inserted before loop j and has a stride of B. In our representation, loop j can be described by the following iteration domain inequalities: $jj \leq j, j \leq jj+B-1$. For the non-unit stride B of loop jj, we introduce *local variables* to keep a linear representation of the iteration domain. For instance, the strip-mined iteration domain of S_2 is $\{(i,jj,j) \mid 0 \leq j, j \leq N-1, jj \leq j, j \leq jj+B-1, jj \mod B = 0, 0 \leq i, i \leq M-1\}$, and after introducing local variable jj_2 such that $jj = B \times jj_2$, the iteration domain becomes $\{(i,jj,j) \mid \exists jj_2, 0 \leq j, j \leq N-1, jj \leq j, j \leq jj+B-1, jj = B \times jj_2, 0 \leq i, i \leq M-1\}$[4] and its matrix representation is the following (with $B = 64$, and from left to right: columns i, jj, j, jj_2, M, N, P, Q and the affine component):

[4] The equation $jj = B \times jj_2$ is simply represented by two inequalities $jj \geq B \times jj_2$ and $jj \leq B \times jj_2$.

$$\begin{array}{cl}
\begin{array}{rrrrrrrrrr} i & j & jj & jj_2 & M & N & P & Q & 1 \end{array} & \\
\left[\begin{array}{rrr|r|cccc|r}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -1 & 0 & 0 & 1 & 0 & 0 & -1 \\
0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 63 \\
0 & -1 & 0 & 64 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & -64 & 0 & 0 & 0 & 0 & 0
\end{array}\right]
&
\begin{array}{l}
0 \le i \\
i \le M-1 \\
0 \le j \\
j \le N-1 \\
jj \le j \\
j \le jj+63 \\
jj \le 64 \times jj_2 \\
64 \times jj_2 \le jj
\end{array}
\end{array}$$

Notations and formal definition Given a statement S within a SCoP, let d_S be the depth of S, $\mathbf{i}$ the vector of loop indices to which S belongs (the dimension of $\mathbf{i}$ is d_S), $\mathbf{i}_{\text{lv}}$ the vector of d_{lv} local variables added to linearize constraints, $\mathbf{i}_{\text{gp}}$ the vector of d_{gp} global parameters, and Λ^S the matrix of n linear constraints (Λ^S has n rows and $d^S + d^S_{\text{lv}} + d_{\text{gp}} + 1$ columns). The iteration domain of S is defined by

$$\mathcal{D}^S_{\text{om}} = \left\{ \mathbf{i} \mid \exists \mathbf{i}_{\text{lv}}, \Lambda^S \times \left[\mathbf{i}, \mathbf{i}_{\text{lv}}, \mathbf{i}_{\text{gp}}, 1\right]^t \ge \mathbf{0} \right\}.$$

2.3.3. Schedules

Feautrier [(8)], Kelly and Pugh [(10)], proposed an encoding that characterizes the order of execution of each statement instance within code sections with multiple and non-perfectly nested loop nests. We use a similar encoding for SCoPs. The principle is to define a *time stamp* for each statement instance, using the iteration vector of the surrounding loops, e.g., vector (i, j) for statement S_2 in the introductory example, and the static statement order to accommodate loop levels with multiple statements. This statement order is defined for each loop level and starts to 0, e.g., the rank of statement S_2 is 1 at depth 1 (it belongs to loop j which is the second statement at depth 1 in this SCoP), 0 at depth 2 (it is the first statement in loop j). And for each statement, the encoding defines a schedule matrix Θ that characterizes the schedule. E.g., the instance (i, j) of statement S_2 is executed before the instance (k, l) of statement S_3 if and only if

$$\Theta^{S_2} \times \left[i, j, 1\right]^t \ll \Theta^{S_3} \times \left[k, l, 1\right]^t$$

(the last component in the instance vector $(i, j, 1)$ — term 1 — is used for the static statement ordering term). Matrix Θ^{S_2} is shown in Figure 17, where the first two columns correspond to i, j and the last column corresponds to the static statement order. The rows of Θ^{S_2} interleave statement order and iteration order so as to implement the lexicographic order: the first row corresponds to depth 0, the second row to the iteration order of loop i, the third row to the static statement order within loop i, the fourth row to the iteration order of loop j, and the fifth row to the static statement order within loop j. Now, the matrix of statement Θ^{S_3} in Figure 17 corresponds to a different loop nest with different iterators.

$$\Theta^{S_2} = \left[\begin{array}{cc|c} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{array}\right] \qquad \Theta^{S_3} = \left[\begin{array}{cc|c} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{array}\right] \qquad \Theta^{S'_2} = \left[\begin{array}{cc|c} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{array}\right]$$

$$\Theta^{S''_2} = \left[\begin{array}{ccc|c} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right]$$

Fig. 17. Schedule matrix examples

Still, thanks to the lexicographic order, the encoding provides a global ordering, and we can check that $\Theta^{S_2} \times [i, j, 1] \ll \Theta^{S_3} \times [k, l, 1]$; in that case, the order is simply characterized by the static statement order at depth 0.

Because the schedule relies on loop iterators, iteration domain modifications — such as introducing a new loop (e.g., strip-mining) — will change the Θ matrix of all loop statements but not the schedule itself. Moreover, adding/removing local variables has no impact on Θ.

We will later see that this global ordering of all statements enables the transparent application of complex transformations like loop fusion.

Formal definition Let A^S be the matrix operating on iteration vectors, d^S the depth of the statement and β^S the static statement ordering vector. The schedule matrix Θ^s is defined by

$$\Theta^S = \left[\begin{array}{ccc|c} 0 & \cdots & 0 & \beta^S_0 \\ A^S_{1,1} & \cdots & A^S_{1,d^S} & 0 \\ 0 & \cdots & 0 & \beta^S_1 \\ A^S_{2,1} & \cdots & A^S_{2,d^S} & 0 \\ \vdots & \ddots & \vdots & \vdots \\ A^S_{d^S,1} & \cdots & A^S_{d^S,d^S} & 0 \\ 0 & \cdots & 0 & \beta^S_{d^S} \end{array}\right]$$

Example: implementing loop interchange and tiling As for unimodular transformations, applying a schedule-only loop transformation like loop interchange simply consists in swapping two rows of matrix Θ, i.e., really two rows of matrix A. Consider loops i and j the introductory example; the new matrix for S_2 associated with the interchange of i and j is called $\Theta^{S'_2}$ in Figure 17.

Now, tiling is a combination of strip-mining and loop interchange and it involves both an iteration domain and a schedule transformation. In our split representation, tiling loop j by a factor B simply consists in applying the iteration domain transformation in the previous paragraph (see the strip-mining example) and the above schedule transformation on all statements within loops i and j. For statement S_2, the only difference with the above loop interchange example is that strip-mining introduces a new loop iterator jj. The transformed matrix is called $\Theta^{S''_2}$ in Figure 17.

Extending the representation to implement more transformations For some statement-wise transformations like shifting (or pipelining), i.e., loop shifting for one statement in the loop body but not the others (e.g., statements S_2 and S_3, after merging loops i, k and j, l), more complex manipulations of the statement schedule are necessary. In fact, the above schedule representation is a simplified version of the actual schedule which includes a third matrix component called Γ. It adds one column to the Θ matrix for every global parameter (e.g., 4 columns for the running example).

2.3.4. Access functions

Privatization modifies array accesses, i.e., array subscripts. For any array reference, a given point in the iteration domain is mapped to an array element (for scalars, all iteration points naturally map to the same element). In other words, there is a function that maps the iteration domain of any reference to array or scalar elements. A transformation like privatization modifies this function: it affects neither the iteration domains nor the schedules.

Consider array reference `B[j][i]`, in statement S_2 after merging loops i, k and j, l, and strip-mining loop j. The matrix for the corresponding access function is simply (columns are i, jj, j, M, N, P, Q, and the scalar component, from left to right):

$$\left[\begin{array}{ccc|cccc|c} 0&0&1&0&0&0&0&0 \\ 1&0&0&0&0&0&0&0 \end{array}\right].$$

Formal definition For each statement S, we define two sets $\mathcal{L}^S_{\text{hs}}$ and $\mathcal{R}^S_{\text{hs}}$ of $(\mathtt{A}, f)$ pairs, each pair representing a reference to variable `A` in the left or right hand side of the statement; f is the *access function* mapping iterations in $\mathcal{D}^S_{\text{om}}$ to A elements. f is a function of loop iterators, local variables and global parameters. The access function f is defined by a matrix F such that

$$f(\mathbf{i}) = \mathrm{F} \times \left[\mathbf{i}, \mathbf{i}_{\text{lv}}, \mathbf{i}_{\text{gp}}, 1\right]^t.$$

Example: implementing privatization Consider again the example in Figure 1 and assume that, instead of splitting statement `Z[i]=0` to enable tiling, we want to privatize array `Z` over dimension j (as an alternative). Besides modifying the declaration of `Z` (see next section), we need to change the subscripts of references to `Z`, adding a row to each access matrix with a 1 in the column corresponding to the new dimension and zeroes elsewhere. E.g., privatization of $\mathcal{L}^{S_2}_{\text{hs}}$ yields

$$\left\{\left(\mathtt{Z}, \left[\,1\,0 \mid 0\,0\,0\,0 \mid 0\,\right]\right)\right\} \longrightarrow \left\{\left(\mathtt{Z}, \left[\begin{array}{cc|cccc|c} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{array}\right]\right)\right\}.$$

2.3.5. Data layout

Some program transformations, like padding, only modify the array declarations and have no impact on the polyhedral representation of statements. It is critical to define these transformations through a separate representation of the mapping of virtual array elements to physical memory location. This paper does not bring any improvement to the existing solutions to this problem [(20)], which are sufficiently mature already to express complex data layout transformations.

Notice a few program transformations can affect both array declarations and array statements. For instance, array merging (combining several arrays into a single one) affects both the declarations and access functions (subscripts change); this transformation is sometimes used to improve spatial locality. We are working on an extension of the representation to accommodate combined modifications of array declarations and statements, in the light of [(20)]. This extension will revisit the split of the schedule matrix into independent parts with separated concerns, to facilitate the expression and the composition of data layout transformations. A similar split may be applicable to access functions as well.

2.4. Putting it All Together

Our representation allows us to compose transformations without reference to a syntactic form, as opposed to previous polyhedral models where a single-step transformation captures the whole loop nest optimization [(8, 11)] or intermediate code generation steps are needed [(9, 10)].

Let us better illustrate the advantage of expressing loop transformations as “syntax-free” function compositions, considering again the example in Figure 1. The polyhedral representation of the original program is the following; statements are numbered S_1, S_2 and S_3, with global parameters $\mathbf{i}_{\text{gp}} = [M, N, P, Q]^t$.

Statement iteration domains

$$\Lambda^{S_1} = \left[\begin{array}{c|c|c} 1 & 0000 & 0 \\ -1 & 1000 & -1 \end{array}\right] \begin{array}{l} 0 \le i \\ i \le M-1 \end{array}$$

$$\Lambda^{S_2} = \left[\begin{array}{cc|c|c} 1 & 0 & 0000 & 0 \\ -1 & 0 & 1000 & -1 \\ 0 & 1 & 0000 & 0 \\ 0 & -1 & 0100 & -1 \end{array}\right] \begin{array}{l} 0 \le i \\ i \le M-1 \\ 0 \le j \\ j \le N-1 \end{array} \quad \Lambda^{S_3} = \left[\begin{array}{cc|c|c} 1 & 0 & 0000 & 0 \\ -1 & 0 & 0010 & 0 \\ 0 & 1 & 0000 & 0 \\ 0 & -1 & 0001 & 0 \end{array}\right] \begin{array}{l} 0 \le i \\ i \le P \\ 0 \le j \\ j \le Q \end{array}$$

Statement schedules

$$\mathrm{A}^{S_1} = [1] \qquad \mathrm{A}^{S_2} = \begin{bmatrix} 10 \\ 01 \end{bmatrix} \qquad \mathrm{A}^{S_3} = \begin{bmatrix} 10 \\ 01 \end{bmatrix}$$

$$\beta^{S_1} = [00]^t \qquad \beta^{S_2} = [010]^t \qquad \beta^{S_3} = [110]^t$$

$$\Gamma^{S_1} = [0000 \mid 0] \qquad \Gamma^{S_2} = \left[\begin{array}{c|c} 0000 & 0 \\ 0000 & 0 \end{array}\right] \qquad \Gamma^{S_3} = \left[\begin{array}{c|c} 0000 & 0 \\ 0000 & 0 \end{array}\right]$$

$$\text{i.e. } \Theta^{S_1} = \left[\begin{array}{c|c} 0 & 0 \\ 1 & 0 \\ 0 & 0 \end{array}\right] \qquad \text{i.e. } \Theta^{S_2} = \left[\begin{array}{c|c} 00 & 0 \\ 10 & 0 \\ 00 & 1 \\ 01 & 0 \\ 00 & 0 \end{array}\right] \qquad \text{i.e. } \Theta^{S_3} = \left[\begin{array}{c|c} 00 & 1 \\ 10 & 0 \\ 00 & 1 \\ 01 & 0 \\ 00 & 0 \end{array}\right]$$

Statement access functions

$$\mathcal{L}_{\text{hs}}^{S_1} = \left\{ \left(\mathsf{Z}, [1|0000|0] \right) \right\} \qquad \mathcal{R}_{\text{hs}}^{S_1} = \left\{ \quad \right\}$$

$$\mathcal{L}_{\text{hs}}^{S_2} = \left\{ \left(\mathsf{Z}, [10|0000|0] \right) \right\}$$

$$\mathcal{R}_{\text{hs}}^{S_2} = \left\{ \left(\mathsf{Z}, [10|0000|0] \right), \left(\mathsf{A}, \left[\begin{array}{c|c|c} 10 & 0000 & 0 \\ 01 & 0000 & 0 \end{array}\right] \right), \left(\mathsf{B}, \left[\begin{array}{c|c|c} 01 & 0000 & 0 \\ 10 & 0000 & 0 \end{array}\right] \right), \left(\mathsf{X}, [01|0000|0] \right) \right\}$$

$$\mathcal{L}_{\text{hs}}^{S_3} = \left\{ \left(\mathsf{Z}, [10|0000|0] \right) \right\}$$

$$\mathcal{R}_{\text{hs}}^{S_3} = \left\{ \left(\mathsf{Z}, [10|0000|0] \right), \left(\mathsf{A}, \left[\begin{array}{c|c|c} 10 & 0000 & 0 \\ 01 & 0000 & 0 \end{array}\right] \right), \left(\mathsf{X}, [01|0000|0] \right) \right\}$$

Step1: merging loops i and k Within the representation, merging loops i and k only influences the schedule of statement S_3, i.e., Θ^{S_3}. No other part of the polyhedral program representation is affected. After merging, statement S_3 has the same static statement order at depth 0 as S_2, i.e., 0; its statement order at depth 1 becomes 2 instead of 1, i.e., it becomes the third statement of merged loop i.

$$\beta^{S_3} = [0\;2\;0]^t$$

Step2: merging loops j and l Thanks to the normalization rules on the polyhedral representation, performing the previous step does not require the generation of a fresh syntactic form to apply loop fusion again on internal loops j and l. Although Θ^{S_3} has been modified, its internal structure still exhibits all opportunities for further transformations. This is a strong improvement on previous polyhedral representations.

Again, internal fusion of loops j and l only modifies Θ^{S_3}. Its static statement order at depth 2 is now 1 instead of 0, i.e., it is the second statement of merged loop j.

$$\beta^{S_3} = [0\;1\;1]^t$$

Step3: fission The fission of the first loop to split-out statement `Z[i]=0` has an impact on Θ^{S_2} and Θ^{S_3} since their statement order at depth 0 is now 1 instead of 0 (`Z[i]=0` is now the new statement of order 0 at depth 0), while their statement order at depth 1 (loop i) is decreased by 1.

$$\beta^{S_2} = [1\;0\;0]^t \quad \beta^{S_3} = [1\;0\;1]^t$$

Step4: strip-mining j Strip-mining loop j only affects the iteration domains of statements S_2 and S_3: it adds a local variable and an iterator (and thus 2 matrix columns to Λ^{S_2} and Λ^{S_3}) plus 4 rows for the new inequalities. It also affects the structure of matrices Θ^{S_2} and Θ^{S_3} to take into account the new iterator, but it does not change the schedule. Λ^{S_2} is the same as the domain matrix for S_2' in Section 2.3.2, and the other matrices are:

$$\Lambda^{S_3} = \left[\begin{array}{rrr|r|cccc|r} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 0 & 1 & -1 \\ 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 63 \\ 0 & -1 & 0 & 64 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -64 & 0 & 0 & 0 & 0 & 0 \end{array}\right] \begin{array}{l} 0 \le i \\ i \le P-1 \\ 0 \le j \\ j \le Q-1 \\ jj \le j \\ j \le jj+63 \\ jj \le 64jj_2 \\ 64jj_2 \le jj \end{array}$$

$$\mathrm{A}^{S_2} = \begin{bmatrix} 1\,0\,0 \\ 0\,1\,0 \\ 0\,0\,1 \end{bmatrix}, \beta^{S_2} = [1\;0\;0\;0]^t \text{ and } \mathrm{A}^{S_3} = \begin{bmatrix} 1\,0\,0 \\ 0\,1\,0 \\ 0\,0\,1 \end{bmatrix}, \beta^{S_3} = [1\;0\;0\;1]^t$$

Step5: strip-mining i Strip-mining i has exactly the same effect for loop i and modifies the statements S_2 and S_3 accordingly.

Step6: interchanging i and j As explained before, interchanging i and j simply consists in swapping the second and fourth row of matrices Θ^{S_2} and Θ^{S_3}, i.e., the rows of A^{S_2} and A^{S_3}

$$\mathrm{A}^{S_2} = \mathrm{A}^{S_3} = \begin{bmatrix} 1\,0\,0\,0 \\ 0\,0\,1\,0 \\ 0\,1\,0\,0 \\ 0\,0\,0\,1 \end{bmatrix} \longrightarrow \Theta^{S_2} = \left[\begin{array}{c|c} 0\,0\,0\,0 & 1 \\ 1\,0\,0\,0 & 0 \\ 0\,0\,0\,0 & 0 \\ 0\,0\,1\,0 & 0 \\ 0\,0\,0\,0 & 0 \\ 0\,1\,0\,0 & 0 \\ 0\,0\,0\,0 & 0 \\ 0\,0\,0\,1 & 0 \\ 0\,0\,0\,0 & 0 \end{array}\right] \longrightarrow \Theta^{S_3} = \left[\begin{array}{c|c} 0\,0\,0\,0 & 1 \\ 1\,0\,0\,0 & 0 \\ 0\,0\,0\,0 & 0 \\ 0\,0\,1\,0 & 0 \\ 0\,0\,0\,0 & 0 \\ 0\,1\,0\,0 & 0 \\ 0\,0\,0\,0 & 0 \\ 0\,0\,0\,1 & 0 \\ 0\,0\,0\,0 & 1 \end{array}\right]$$

Summary Overall, none of the transformations has increased the number of statements. Only transformations which add new loops and local variables increase the dimension of some statement matrices but they do not make the representation less generic or harder to use for compositions, since they enforce the normalization rules.

2.5. Normalization Rules

The separation between the domain, schedule, data layout and access functions attributes plays a major role in the compositionality of polyhedral transformations. Indeed, actions on different attributes compose in a trivial way, e.g., strip-mining (iteration domain), interchange (schedule) and padding (data layout). Nevertheless, the previous definitions do not, alone, guarantee good compositionality properties. To achieve our goal, we need to define additional normalization rules.

A given program can have multiple polyhedral representations. This is not harmless when the applicability of a transformation relies on the satisfaction of representation prerequisites. For example, it is possible to merge two statements in two loops only if these two statements are consecutive at the loops depth; e.g., assume the statement order of these two statements is respectively 0 and 2 instead of 0 and 1; the statement order (and thus the schedule) is the same but the statements are not consecutive and fusion seems impossible without prior transformations. Even worse, if the two statements have identical β vectors, fusion makes sense only if their schedules span disjoint time iterations, which in turn depends on both their A and Γ components, as well as their iteration domains. Without enforcing strong invariants to the representation, it is hopeless to define a program transformation uniquely from the matrices. *Normalizing* the representation after each transformation step is a critical contribution of our framework. It proceeds as follows.

Schedule matrix structure. Among many encodings, we choose to partition Θ into three components: matrices A (for iteration reordering) and Γ (iteration shifting), and vector β (statement reordering, fusion, fission), capturing different kinds of transformations. This avoids cross-pollution between statement and iteration reordering, removing expressiveness constraints on the combination of loop fusion with unimodular transformations and shifting. It allows to compose schedule transformations without a costly normalization to the Hermite normal form.

Sequentiality. This is the most important idea that structures the whole unified representation design. In brief, distinct statements, or identical statements in distinct iterations, cannot have the same time stamp. Technically, this rule is slightly stronger than that: we require that the A component of the schedule matrix is non-singular, that all statements have a different β vector, and that no β vector may be the prefix of another one.

This invariant brings two strong properties: (1) it suppresses scheduling ambiguities at code generation time, and (2) it guarantees that rule-compliant transformation of the schedule and will preserve sequentiality of the whole SCoP, independently of iteration domains. The first property is required to give the scheduling algorithm full control on the generated code. The second one is a great asset for separating the concerns when defining, applying or checking a transformation; domain and schedule are strictly independent, as much as modifications to A may ignore modifications to β and vice versa.

It is very important to understand that schedule sequentiality is in no way a limitation in the context of deeply and explicitly parallel architectures. First of all, parallel affine schedules are not the only way to express parallelism (in fact, they are mostly practical to describe bulk-synchronous parallelism), and in case they would be used to specify a partial ordering of statement instances, it is always possible to extend the schedule with "spatial" dimensions to make A invertible (36).

Schedule density. Ensure that all statements at the same depth have a consecutive β ordering (no gap).

Domain density. Generation of efficient imperative code when scanning $\mathbb{Z}$-polyhedra (a.k.a. lattice polyhedra or linearly bounded lattices) is known to be a hard problem [37,21]. Although not an absolute requirement, we try to define transformations that do not introduce local variables in the iteration domain. In particular, we will see in the next section that we use a different, less intuitive and more implicit definition of strip-mining to avoid the introduction of a local variable in the constraint matrix.

Domain parameters. Avoid redundant inequalities and try to reduce integer overflows in domain matrices Λ by normalizing each row.

3. REVISITING CLASSICAL TRANSFORMATIONS

The purpose of this section is to review, with more detail, the formal definition of classical transformations in our compositional setting. Let us first define elementary operations called *constructors*. Constructors make no assumption about the representation invariants and may violate them.

Given a vector v and matrix M with $\dim(v)$ columns and at least i rows, $\mathsf{AddRow}(\mathrm{M}, i, v)$ inserts a new row at position i in M and fills it with the value of vector v, $\mathsf{RemRow}(\mathrm{M}, i)$ does the opposite transformation. $\mathsf{AddCol}(\mathrm{M}, j, v)$ and $\mathsf{RemCol}(\mathrm{M}, j)$ play similar roles for columns.

Moving a statement S forward or backward is a common operation: the constructor $\mathsf{Move}(P, Q, o)$ leaves all statements unchanged except those satisfying

$$\forall S \in \mathcal{S}_{\mathrm{cop}}, P \sqsubseteq \beta^S \wedge (Q \ll \beta^S \vee Q \sqsubseteq \beta^S) : \beta^S_{\dim(P)} \leftarrow \beta^S_{\dim(P)} + o,$$

where $u \sqsubseteq v$ denotes that u is a prefix of v, where P and Q are *statement ordering prefixes* s.t. $P \sqsubseteq Q$ defining respectively the context of the move and marking the initial time-stamp of statements to be moved, and where offset o is the value to be added to the component at depth $\dim(P)$ of any statement ordering vector β^S prefixed by P and following Q. If o is positive, $\mathsf{Move}(P, Q, o)$ inserts o free slots before all statements S preceded by the statement ordering prefix Q at the depth of P; respectively, if o is negative, $\mathsf{Move}(P, Q, o)$ deletes $-o$ slots.

3.1. Transformation Primitives

From the earlier constructors, we define transformation *primitives* to serve as building blocks for transformation sequences. These primitives do enforce the normalization rules. Figure 18 lists typical primitives affecting the polyhedral representation of a statement. $\mathbf{1}_k$ denotes the vector filled with zeros but element k set to 1, i.e., $(0, \ldots, 0, 1, 0, \ldots, 0)$; likewise, $\mathbf{1}_{i,j}$ denotes the matrix filled with zeros but element (i, j) set to 1.

Like the Move *constructor, primitives do not directly operate on loops or statements, but target a collection of statements and polyhedra whose statement-ordering vectors share a common prefix* P*. There are no prerequisites on the program representation to the application and composition of primitives.*

We also specified a number of optional *validity prerequisites* that conservatively check for the semantical soundness of the transformation, e.g., there are validity prerequisites to check that no dependence is violated by a unimodular or array contraction transformation. When exploring the space of possible transformation sequences, validity prerequisites avoid wasting time on corrupt transformations.

FUSION and FISSION best illustrate the benefit of designing loop transformations at the abstract semantical level of our unified polyhedral representation. First of all, loop bounds are not an issue since the code generator will handle any overlapping of iteration domains. For the fission primitive, vector (P,o) prefixes all statements concerned by the fission; and parameter b indicates the position where statement delaying should occur. For the fusion primitive, vector $(P,o+1)$ prefixes all statements that should be interleaved with statements prefixed by (P,o). Eventually, notice that fusion followed by fission — with the appropriate value of b — leaves the SCoP unchanged.

The expressive power of the latter two transformations can be generalized through the very expressive MOTION primitive. This transformation can displace a block of statements prefixed by P to a location identified by vector T, preserving the nesting depth of all statements and enforcing normalization rules. This transformation ressembles a polyhedral "cut-and-paste" operation that completely abstracts all details of the programs other than statement ordering in multidimensional time. This primitive uses an additional notation: $\mathrm{pfx}(V,d)$ computes the sub-vector composed of the first d components of V.

UNIMODULAR implements any unimodular transformation, extended to arbitrary iteration domains and loop nesting. U denotes a unimodular matrix. Notice the multiplication operates on both A *and* Γ, effectively updating the parametric shift along with skewing, reversal and interchange transformations, i.e., preserving the relative shift with respect to the time dimensions it was applied upon.

SHIFT implements a kind of hierarchical software pipelining on the source code. It is extended with parametric iteration shifts, e.g., to delay a statement by N iterations of one surrounding loop. Matrix M implements the parameterized shift of the affine schedule of a statement. M must have the same dimension as Γ.

CUTDOM constrains a domain with an additional inequality, in the form of a vector c with the same dimension as a row of matrix Λ.

EXTEND inserts a new intermediate loop level at depth ℓ, initially restricted to a single iteration. This new iterator will be used in following code transformations.

ADDLOCALVAR insert a fresh local variable to the domain and to the access functions. This local variable is typically used by CUTDOM.

PRIVATIZE and CONTRACT implement basic forms of array privatization and contraction, respectively, considering dimension ℓ of the array. Privatization needs an additional parameter s, the size of the additional dimension; s is required to update the array declaration (it cannot be inferred in general, some references may not be affine). These primitives are simple examples updating the data layout and array access functions.

This table is not complete (e.g., it lacks index-set splitting and data-layout transformations), but it demonstrates the expressiveness of the unified representation.

Syntax	Effect
UNIMODULAR(P,U)	$\forall S \in \mathcal{S}_{\text{cop}} \mid P \sqsubseteq \beta^S, \mathrm{A}^S \leftarrow \mathrm{U}.\mathrm{A}^S;\ \Gamma^S \leftarrow \mathrm{U}.\Gamma^S$
SHIFT(P,M)	$\forall S \in \mathcal{S}_{\text{cop}} \mid P \sqsubseteq \beta^S, \Gamma^S \leftarrow \Gamma^S + \mathrm{M}$
CUTDOM(P,c)	$\forall S \in \mathcal{S}_{\text{cop}} \mid P \sqsubseteq \beta^S, \Lambda^S \leftarrow \mathsf{AddRow}\big(\Lambda^S, 0, c/\gcd(c_1,\ldots,c_{d^S+d^S_{\text{lv}}+d_{\text{gp}}+1})\big)$
EXTEND(P,ℓ,c)	$\forall S \in \mathcal{S}_{\text{cop}} \mid P \sqsubseteq \beta^S, \begin{cases} d^S \leftarrow d^S + 1;\ \Lambda^S \leftarrow \mathsf{AddCol}(\Lambda^S, c, 0); \\ \beta^S \leftarrow \mathsf{AddRow}(\beta^S, \ell, 0); \\ \mathrm{A}^S \leftarrow \mathsf{AddRow}(\mathsf{AddCol}(\mathrm{A}^S, c, 0), \ell, \mathbf{1}_\ell); \\ \Gamma^S \leftarrow \mathsf{AddRow}(\Gamma^S, \ell, 0); \\ \forall (\mathsf{A}, \mathrm{F}) \in \mathcal{L}^S_{\text{hs}} \cup \mathcal{R}^S_{\text{hs}}, \mathrm{F} \leftarrow \mathsf{AddRow}(\mathrm{F}, \ell, 0) \end{cases}$
ADDLOCALVAR(P)	$\forall S \in \mathcal{S}_{\text{cop}} \mid P \sqsubseteq \beta^S, d^S_{\text{lv}} \leftarrow d^S_{\text{lv}} + 1;\ \Lambda^S \leftarrow \mathsf{AddCol}(\Lambda^S, d^S + 1, 0);$ $\forall (\mathsf{A}, \mathrm{F}) \in \mathcal{L}^S_{\text{hs}} \cup \mathcal{R}^S_{\text{hs}}, \mathrm{F} \leftarrow \mathsf{AddCol}(\mathrm{F}, d^S + 1, 0)$
PRIVATIZE(A,ℓ)	$\forall S \in \mathcal{S}_{\text{cop}}, \forall (\mathsf{A}, \mathrm{F}) \in \mathcal{L}^S_{\text{hs}} \cup \mathcal{R}^S_{\text{hs}}, \mathrm{F} \leftarrow \mathsf{AddRow}(\mathrm{F}, \ell, \mathbf{1}_\ell)$
CONTRACT(A,ℓ)	$\forall S \in \mathcal{S}_{\text{cop}}, \forall (\mathsf{A}, \mathrm{F}) \in \mathcal{L}^S_{\text{hs}} \cup \mathcal{R}^S_{\text{hs}}, \mathrm{F} \leftarrow \mathsf{RemRow}(\mathrm{F}, \ell)$
FUSION(P,o)	$b = \max\{\beta^S_{\dim(P)+1} \mid (P,o) \sqsubseteq \beta^S\} + 1$ $\mathsf{Move}((P,o+1),(P,o+1),b);\ \mathsf{Move}(P,(P,o+1),-1)$
FISSION(P,o,b)	$\mathsf{Move}(P,(P,o,b),1);\ \mathsf{Move}((P,o+1),(P,o+1),-b)$
MOTION(P,T)	if $\dim(P)+1 = \dim(T)$ then $b = \max\{\beta^S_{\dim(P)} \mid P \sqsubseteq \beta^S\} + 1$ else $b = 1$ $\mathsf{Move}(\mathrm{pfx}(T,\dim(T)-1),T,b)$ $\forall S \in \mathcal{S}_{\text{cop}} \mid P \sqsubseteq \beta^S, \beta^S \leftarrow \beta^S + T - \mathrm{pfx}(P,\dim(T))$ $\mathsf{Move}(P,P,-1)$

Fig. 18. Some classical transformation primitives

Primitives operate on program representation while maintaining the structure of the polyhedral components (the invariants). Despite their familiar names, the primitives' practical outcome on the program representation is widely extended compared to their syntactic counterparts. Indeed, transformation primitives like fusion or interchange apply to sets of statements that may be scattered and duplicated at many different locations in the generated code. In addition, these transformations are not proper *loop* transformations anymore, since they apply to sets of statement iterations that may have completely different domains and relative iteration schedules. For example, one may interchange the loops surrounding one statement in a loop body without modifying the schedule of other statements, and without distributing the loop first. Another example is the fusion of two loops with different domains without peeling any iteration.

Previous encodings of classical transformations in a polyhedral setting — most significantly (9) and (10) — use Presburger arithmetic as an expressive *operating* tool for implementing and validating transformations. In addition to operating on polytopes, our work *generalizes* loop transformations to more abstract *polyhedral domain* transformations, without explicitly relying on a nested loop structure with known bounds and array subscripts to define the transformation.

Instead of anchoring loop transformations to a syntactic form of the program, limitting ourselves to what can be expressed with an imperative semantics, we define higher level transformations on the polyhedral representation itself, abstracting away the overhead (versioning, duplication) and constraints of the code generation process (translation to an imperative semantics).

Naturally, this higher-level framework is beneficial for transformation composition. Figure 19 composes primitives into typical transformations. INTERCHANGE swaps the roles of $\mathbf{i}_o$ and $\mathbf{i}_{o+1}$ in the schedule of the matching statements; it is a fine-grain extension of the classical interchange making no assumption about the shape of the iteration domain. SKEW and REVERSE define two well known unimodular transformations, with respectively the skew factor s with it's coordinates (ℓ, c), and the depth o of the iterator to be reversed. STRIPMINE introduces a new iterator to strip the schedule and iteration domain of all statements at the depth of P into intervals of length k (where k is a *statically known integer*). This transformation is a sequence of primitives and does not resort to the insertion of any additional local variable, see Figure 19. TILE extends the classical loop tiling at of the two nested loops at the depth of P, using $k \times k$ blocks, with arbitrary nesting and iteration domains. Tiling and strip-mining always operate on *time* dimensions, hence the propagation of a line from the schedule matrix (from A and Γ) into the iteration domain constraints; it is possible to tile the surrounding time dimensions of any collection of statements with unrelated iteration domains and schedules.

Syntax	Effect	Comments
$\textsc{Interchange}(P,o)$	$\forall S \in \mathcal{S}_{\text{cop}} \mid P \sqsubseteq \beta^S$, $\begin{cases} \mathrm{U} = \mathrm{I}_{d^S} - \mathbf{1}_{o,o} - \mathbf{1}_{o+1,o+1} + \mathbf{1}_{o,o+1} + \mathbf{1}_{o+1,o}; \\ \textsc{Unimodular}(\beta^S, \mathrm{U}) \end{cases}$	swap rows o and $o+1$
$\textsc{Skew}(P,\ell,c,s)$	$\forall S \in \mathcal{S}_{\text{cop}} \mid P \sqsubseteq \beta^S$, $\begin{cases} \mathrm{U} = \mathrm{I}_{d^S} + s \cdot \mathbf{1}_{\ell,c}; \\ \textsc{Unimodular}(\beta^S, \mathrm{U}) \end{cases}$	add the skew factor
$\textsc{Reverse}(P,o)$	$\forall S \in \mathcal{S}_{\text{cop}} \mid P \sqsubseteq \beta^S$, $\begin{cases} \mathrm{U} = \mathrm{I}_{d^S} - 2 \cdot \mathbf{1}_{o,o}; \\ \textsc{Unimodular}(\beta^S, \mathrm{U}) \end{cases}$	put a -1 in (o,o)
$\textsc{StripMine}(P,k)$	$\forall S \in \mathcal{S}_{\text{cop}} \mid P \sqsubseteq \beta^S$, $\begin{cases} c = \dim(P); \\ \textsc{Extend}(\beta^S, c, c); \\ u = d^S + d^S_{\text{lv}} + d_{\text{gp}} + 1; \\ \textsc{CutDom}(\beta^S, -k \cdot \mathbf{1}_c + (\mathrm{A}^S_{c+1}, \Gamma^S_{c+1})); \\ \textsc{CutDom}(\beta^S, k \cdot \mathbf{1}_c - (\mathrm{A}^S_{c+1}, \Gamma^S_{c+1}) + (k-1)\mathbf{1}_u) \end{cases}$	 insert intermediate loop constant column $k \cdot \mathbf{i}_c \leq \mathbf{i}_{c+1}$ $\mathbf{i}_{c+1} \leq k \cdot \mathbf{i}_c + k - 1$
$\textsc{Tile}(P,o,k_1,k_2)$	$\forall S \in \mathcal{S}_{\text{cop}} \mid (P,o) \sqsubseteq \beta^S$, $\begin{cases} \textsc{StripMine}((P,o), k_2); \\ \textsc{StripMine}(P, k_1); \\ \textsc{Interchange}((P,0), \dim(P)) \end{cases}$	 strip outer loop strip inner loop interchange

Fig. 19. Composition of transformation primitives

3.2. Implementing Loop Unrolling

In the context of code optimization, one of the most important transformations is loop unrolling. A naive implementation of unrolling with statement duplications may result in severe complexity overhead for further transformations and for the code generation algorithm (its separation algorithm is exponential in the number of statements, in the worst case). Instead of implementing loop unrolling in the intermediate representation of our framework, we delay it to the code generation phase and perform full loop unrolling in a *lazy* way. This strategy is fully implemented in the code generation phase and is triggered by annotations (carrying depth information) of the statements whose surrounding loops need to be unrolled; unrolling occurs in the separation algorithm of the code generator (22) when all the statements being printed out are marked for unrolling at the current depth.

Practically, in most cases, loop unrolling by a factor b an be implemented as a combination of *strip-mining* (by a factor b) and *full unrolling* (6). Strip-mining itself may be implemented in several ways in a polyhedral setting. Following our earlier work in (7) and calling b the strip-mining factor, we choose to model a strip-mined loop by dividing the iteration span of the outer loop by b instead of leaving the bounds unchanged and inserting a non-unit stride b, see Figure 20.

```
for(i=ℓ(x); i<=u(x); i++)
```

$\xrightarrow{\text{strip-mine }(b)}$

```
for(t1=⌊ℓ(x)/b⌋; t1<=⌊u(x)/b⌋; t1++)
  for(t2=max(ℓ(x),b*t1); t2<=min(u(x),b*t1+b-1); t2++)
```

Fig. 20. Generic strip-mined loop after code generation

This particular design preserves the convexity of the polyhedra representing the transformed code, alleviating the need for specific stride recognition mechanisms (based, e.g., on the Hermite normal form).

In Figure 21(b) we can see how strip-mining the original code of Figure 21(a) by a factor of 2 yields an internal loop with non-trivial bounds. It can be very useful to unroll the innermost loop to exhibit register reuse (a.k.a. register tiling), relax scheduling constraints and diminish the impact of control on useful code. However, unrolling requires to cut the domains so that `min` and `max` constraints disappear from loop bounds. Our method is presented in more detail in [38]; it intuitively boils down to finding conditionals (lower bound and upper bound) *such that their difference is a non-parametric constant*: the unrolling factor. Hoisting these conditionals actually amounts to splitting the outer strip-mined loop into a kernel part where the inner strip-mined loop will be fully unrolled, and a remainder part (not unrollable) spanning at most as many iterations as the strip-mining factor. In our example, the conditions associated with a constant trip-count (equal to 2) are `t2>=2*t1` and `t2<=2*t1+1` and are associated with the kernel, separated from the prologue where `2*t1<M` and from the epilogue where `2*t1+1>N`. This separation leads to the more desirable form of Figure 21(c).

Finally, instead of implementing loop unrolling in the intermediate representation of our framework, we delay it to the code generation phase and perform full loop unrolling in a lazy way, avoiding the added (exponential) complexity on the separation algorithm. This approach relies on a preliminary strip-mine step that determines the amount of partial unrolling.

3.3. Parallelizing Transformations

Most parallelizing compilers rely on loop transformations to extract and expose parallelism, from vector and instruction-level to thread-level forms of parallelism [39–42, 18, 43–46]. The most common strategy is to compose loop transformations to extract parallel (`doall`) or pipeline (`doacross`) loops [41]. The main transformations include privatization [47, 48, 32] for de-

```
for(t1=M; t1<=N; t1++)
  S1(i = t1);
```

(a) Original code

```
for(t1=M/2; t1<=N/2; t1++)
  for(t2=max(M,2*t1);
      t2<=min(N,2*t1+1); t2++)
    S1(i = t2);
```

(b) Strip-mining of 2

```
if(M%2==1)
  S1(i = M);
for(t1=(M+1)/2; t1<=(N-1)/2; t1++)
  S1(i = 2*t1);
  S1(i = 2*t1+1);
if(N%2==0)
  S1(i = N);
```

(c) Separation & unrolling

Fig. 21. Strip-mining and unrolling transformation

pendence removal and unimodular transformations or node splitting to rearrange dependences [49, 6].

Many academic approaches to automatic parallelization have used the polyhedral model — and partially ordered affine schedules in particular — to describe fine grain vector [50, 8, 51] or systolic [52, 53] parallelism. Affine schedules have also been applied to the extraction and characterization of bulk-synchronous parallelism [11, 36, 13]. Array expansion is a generalization of privatization that leverages on the precision of array dependence analysis in the polyhedral model [54, 55, 33]. Array contraction [6, 13] and its generalization called storage mapping optimization [56–58] allows to control the overhead due to expansion techniques.

Our work does not aim at characterizing parallel execution with partially ordered affine schedules. In this sense, we prefer the more general and decoupled approach followed by traditional parallelizing compilers where parallelism is a separate concern. Loop transformations expressed on the schedule parts of the representation are seen as *enabling* transformations to extract parallel loops or independent instructions in loop bodies. These enabling transformations are associated with a precise dependence analysis to effectively allow to generate code with parallel execution annotations, using e.g., OpenMP.[5]

Recent works are indeed suggesting that parallelism is better expressed as a result of sophisticated analyses and annotations on the program than using rigid partial orders defined by multi-dimensional affine schedules. For example, a modernized version of Polaris has been used to (fully) automatically extract vast amounts of effectively exploitable parallelism in scientific codes, using *hybrid analysis*: a combination of static, dynamic and speculative dependence tests (62). Yet these results used *no prior loop transformation to enhance scalability through additional parallelism extraction or to coarsen its grain*. Although we cannot show any empirical evidence yet, we believe the same reason why our framework improves on single-threaded optimizations (flexibility to express complex transformation sequences) will bring more scalability and robustness to these promising hybrid parallelization techniques.

3.4. Facilitating the Search for Compositions

To conclude this section, we study how our polyhedral representation with normalization rules for compositionality can further facilitate the search for complex transformation sequences.

We have seen that applying a program transformation simply amounts to recomputing the matrices of a few statements. This is a major increase in flexibility, compared to syntactic approaches where the code complexity increases with each transformation. It is still the case for prefetching and strip-mining, where, respectively, a statement is added and matrix dimensions are increased; but the added complexity is fairly moderate, and again the representation is no less generic.

3.4.1. Transformation Space

Commutativity properties are additional benefits of the separation into four representation aspects and the normalization rules. In general, data and control transformations commute, as well as statement reordering and iteration reordering. For example, loop fusion commutes with loop interchange, statement reordering, loop fission and loop fusion itself. In the example detailed in Section 2.4, swapping fusion and fission has no effect on the resulting representation; the first row of β vectors below shows double fusion followed by fission, while the second row shows fission followed by double fusion.

[5] We implemented an exact one when all array accesses are affine (59); graceful degradations exist for the general case (60, 61) but are not supported yet.

$$\begin{tikzcd}
\begin{array}{l}\beta^{S_1} = [0\,0] \\ \beta^{S_2} = [0\,1\,0] \\ \beta^{S_3} = [1\,0\,0]\end{array} \arrow[r] \arrow[d] & \begin{array}{l}\beta^{S_1} = [0\,0] \\ \beta^{S_2} = [0\,1\,0] \\ \beta^{S_3} = [0\,2\,0]\end{array} \arrow[r] & \begin{array}{l}\beta^{S_1} = [0\,0] \\ \beta^{S_2} = [0\,1\,0] \\ \beta^{S_3} = [0\,1\,1]\end{array} \arrow[d] \\
\begin{array}{l}\beta^{S_1} = [0\,0] \\ \beta^{S_2} = [1\,0\,0] \\ \beta^{S_3} = [2\,0\,0]\end{array} \arrow[r] & \begin{array}{l}\beta^{S_1} = [0\,0] \\ \beta^{S_2} = [1\,0\,0] \\ \beta^{S_3} = [1\,1\,0]\end{array} \arrow[r] & \begin{array}{l}\beta^{S_1} = [0\,0] \\ \beta^{S_2} = [0\,1\,0] \\ \beta^{S_3} = [0\,1\,1]\end{array}
\end{tikzcd}$$

Confluence properties are also available: outer loop unrolling and fusion (unroll-and-jam) is strictly equivalent to strip-mining, interchange and full unrolling. The latter sequence is the best way to implement unroll-and-jam in our framework, since it does not require statement duplication in the representation itself but relies on lazy unrolling. In general, strip-mining leads to confluent paths when combined with fusion or fission.

Such properties are useful in the context of iterative searches because they may significantly reduce the search space, and they also improve the understanding of its structure, which in turn enables more efficient search strategies (2).

Strip-mining and shifting do *not* commute. However applying shifting after strip-mining amounts to intra-tile pipelining (the last iteration of a tile stays in that tile), whereas the whole iteration space is pipelined across tiles when applying strip-mining after shifting (the last iteration of a tile being shifted towards the first iteration of the next tile).

3.4.2. When changing a sequence of transformations simply means changing a parameter

Finally, the code representation framework also opens up a new approach for searching compositions of program transformations. Since many program transformations have the only effect of modifying the matrix parameters, an alternative is to *directly search the matrix parameters themselves*. In some cases, changing one or a few parameters is equivalent to performing a sequence of program transformations, making this search much simpler and more systematic.

For instance, consider the Θ^{S_3} matrix of Section 2.3.3 and now assume we want to systematically search schedule-oriented transformations. A straightforward approach is to systematically search the Θ^{S_3} matrix parameters themselves. Let us assume that, during the search we randomly reach the following matrix:

$$\Theta^{S'_3} = \left[\begin{array}{cc|c} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{array}\right]$$

This matrix has 7 differences with the original Θ^{S_3} matrix of Section 2.3.3, and these differences actually correspond to the composition of 3 transformations: loop interchange (loops k and l), outer loop fusion (loops i and k) and inner loop fusion (loops j and l). In other words, searching the matrix parameters is equivalent to searching for compositions of transformations.

Furthermore, assuming that a full polyhedral dependence graph has been computed,[6] it is possible to characterize the *exact set of all schedule, domain and access matrices associated with legal transformation sequences*. This can be used to quickly filter out or correct any violating transformation (63), or even better, using the Farkas lemma as proposed by Feautrier (8), to recast this implicit characterization into an explicit list of domains (of Farkas multipliers) enclosing the very values of all matrix coefficients associated with legal transformations. Searching for a proper transformation within this domain would be amenable to mathematical tools, like linear programming, promising better scalability than genetic algorithms on plain transformation sequences. This idea is derived from the "chunking" transformation for automatic locality optimization (64, 63); it is the subject of active ongoing work.

4. HIGHER PERFORMANCE REQUIRES COMPOSITION

We have already illustrated the need for long sequences of composed transformations and the limitations of syntactic approaches on the synthetic example of Section 2.1. This section provides similar empirical evidence on realistic benchmarks, focusing on single-thread performance and locality optimizations although it also applies to automatic parallelization.

We *manually* optimized 12 SPECfp2000 benchmarks (out of 14) and were able to outperform the peak SPEC performance (65) (obtained in choosing the most appropriate compiler flags) for 9 of them (5).[7] We detail below the composition sequences for 4 of these benchmarks, the associated syntactic limitations and how we override them.

[6] Our tool performs on-demand computation, with lists of polyhedra capturing the (exact) instance-wise dependence information between pairs of references.

[7] The 3 other benchmarks could not be optimized manually in a "reasonable" amount of time, following the empirical methodology presented in (5).

4.1. Manual Optimization Results

Our experiments were conducted on an HP AlphaServer ES45, 1GHz Alpha 21264C EV68 (1 processor enabled) with 8MB L2 cache and 8GB of memory. We will compare our optimized versions with the *base* SPEC performance, i.e., the output of the HP Fortran (V5.4) and C (V6.4) compiler (`-arch ev6 -fast -O5 ONESTEP`) using the KAP Fortran preprocessor (V4.3). We will also compare with the *peak* SPEC performance. Figure 22 summarizes the speedup with respect to the base SPEC performance.

4.1.1. Methodology

Our approach follows an empirical methodology for *whole program* optimization, taking all architecture components into account, using the HP EV68 processor simulator. Even though this optimization process is out of the scope of this article, we briefly describe it in the next paragraphs.

	Peak SPEC	Manual		Peak SPEC	Manual
swim	1.00	1.61	galgel	1.04	1.39
wupwise	1.20	2.90	applu	1.47	2.18
apsi	1.07	1.23	mesa	1.04	1.42
ammp	1.18	1.40	equake	2.65	3.22
mesa	1.12	1.17	mgrid	1.59	1.45
fma3d	1.32	1.09	art	1.22	1.07

Fig. 22. Speedup for 12 SPECfp2000 benchmarks

This methodology is captured in a decision tree: we iterate *dynamic analysis* phases of the program behavior, using HP's cycle-accurate simulator, *decision* phases to choose the next analysis or transformation to perform, and program transformation phases to address a given performance issue. After each transformation, the performance is measured on the real machine to evaluate the actual benefits/losses, then we run a new analysis phase to decide whether it is worth iterating the process and applying a new transformation. Though this optimization process is manual, it is also *systematic* and iterative, the path through the decision tree being guided by increasingly detailed performance metrics. Except for precisely locating target code sections and checking the legality of program transformations, it could almost perform automatically.

From a program transformation point of view, our methodology results in a structured sequence of transformations applied to various code sections. In the examples below, for each program, we focus on one to three code sections where multiple transformations are iteratively applied, i.e., composed. We make the distinction between the *target* transformations identified through dynamic analysis, e.g., loop tiling to reduce TLB misses, and the *enabling* transformations to apply the target transformations themselves, e.g., privatization for dependence removal.

4.1.2. Transformation sequences

In the following, we assume an aggressive inlining of all procedure calls within loops (performed by KAP in most cases). The examples in Figures 26, 24 and 23 show a wide variability in transformation sequences and ordering. Each analysis and transformation phase is depicted as a gray box, showing the time difference when executing the *full benchmark* (in seconds, a negative number is a performance improvement); the base execution time for each benchmark is also indicated in the caption. Each transformation phase, i.e., each gray box, is then broken down into traditional transformations, i.e., white boxes.

All benchmarks benefited from complex compositions of transformations, with up to 23 individual loop and array transformations on the same loop nest for galgel. Notice that some enabling transformations actually degrade performance, like (A2) in galgel.

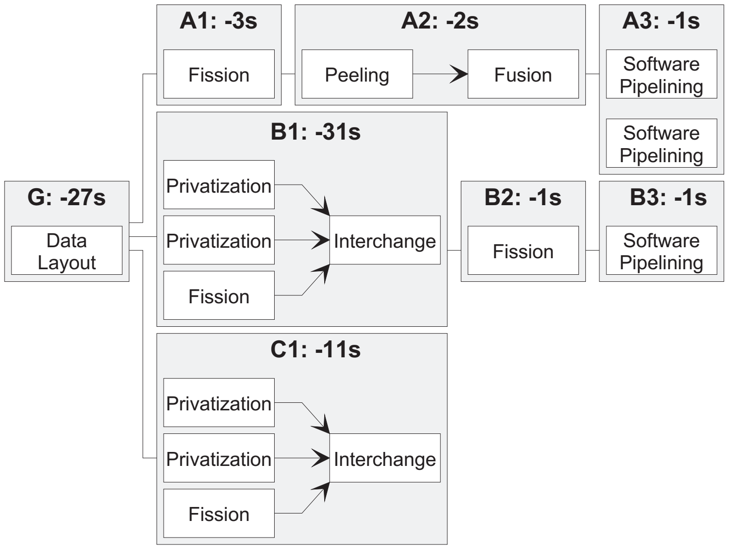


Fig. 23. Optimizing apsi (base 378s)

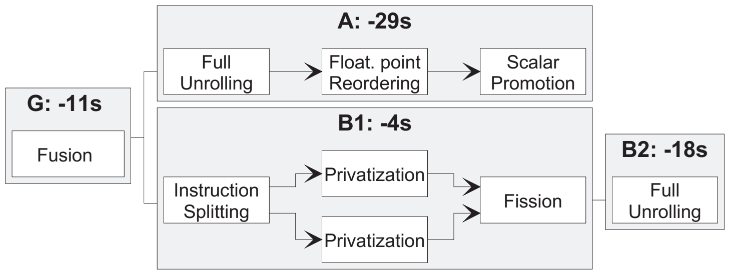


Fig. 24. Optimizing applu (base 214s)

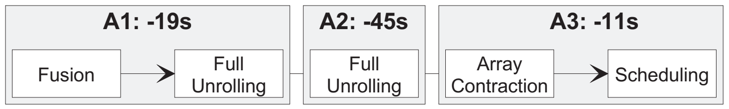


Fig. 25. Optimizing wupwise (base 236s)

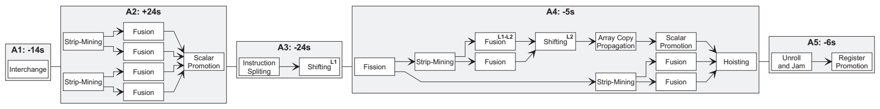


Fig. 26. Optimizing galgel (base 171s)

4.2. Polyhedral vs. Syntactic Representations

Section 2 presented the main assets of our new polyhedral representation. We now revisit these properties on the 4 chosen benchmarks.

4.2.1. Code size and complexity

The manual application of transformation sequences leads to a large code size increase, let aside the effect of function inlining. This is due to code duplication when unrolling loops, but also to iteration peeling and loop versioning when applying loop tiling and strip-mining. Typical cases are phases (A) in applu and (A2) wupwise (unrolling), and (A5) in galgel (unroll-and-jam).

In our framework, none of these transformations perform any statement duplication, only strip-mining has a slight impact on the size of domain matrices, as explained in Section 2.3. In general, the only duplication comes from parameter versioning and from intrinsicly code-bloating schedules resulting from intricate transformation sequences. This “moral” observation allows to blame the transformation sequence rather than the polyhedral transformation infrastructure, yet it does not provide an intuitive characterization of the “good” transformation sequences that do not yield code-bloating schedules; this is left for future work.

Interestingly, it is also possible to control the aggressiveness of the polyhedral code generator, focusing its code-duplicating optimizations to the hottest kernels only, yielding sub-optimal but very compact code in the rest of the program. Again, the design of practical heuristics to drive these technique is left for future work.

4.2.2. Breaking patterns

On the introductory example, we already outlined the difficulty to merge loops with different bounds and tile non-perfectly nested loops. Beyond non-matching loop bounds and non-perfect nests, loop fusion is also inhibited by loop peeling, loop shifting and versioning from previous phases. For example, galgel shows multiple instances of fusion and tiling transformations after peeling and shifting. KAP’s pattern-matching rules fail to recognize any opportunity for fusion or tiling on these examples.

Interestingly, syntactic transformations may also introduce some spurious array dependences that hamper further optimizations. For example, phase (A3) in galgel splits a complex statement with 8 array references, and shifts part of this statement forward by one iteration (software pipelining) of a loop L_1. Then, in one of the fusion boxes of phase (A4), we wish to merge L_1 with a subsequent loop L_2. Without additional care, this fusion would *break dependences*, corrupting the semantics of the code produced after (A3). Indeed, some values flow from the shifted statement in L_1 to iterations of L_2; merging the loops would consume these values before producing them. Syntactic approaches lead to a dead-end in this case; the only way to proceed is to undo the shifting step, increasing execution time by 24 seconds. Thanks to the commutation properties of our model, we can make the dependence between the loops compatible with fusion by shifting the loop L_2 forward by one iteration, before applying the fusion.

4.2.3. Flexible and complex compositions of transformations

The manual benchmark optimizations exhibit wide variations in the composition of control, access and layout transformations. galgel is an extreme case where KAP does not succeed in optimizing the code, even with the best hand-tuned combination of switches, i.e., when directed to apply some transformations with explicit optimization switches (peak SPEC). Nevertheless, our (long) optimization sequence yields a significant speedup while only applying classical transformations. A closer look at the code shows only uniform dependences and constant loop bounds. In addition to the above-mentioned syntactic restrictions and pattern mismatches, our sequence of transformations shows the variability and complexity of enabling transformations. For example, to implement the eight loop fusions in Figure 26, strip-mining must be applied to convert large loops of N^2 iterations into nested loops of N iterations, allowing subsequent fusions with other loops of N iterations.

applu stresses another important flexibility issue. Optimizations on two independent code fragments follow an opposite direction: (G) and (A) target locality improvements: they implement loop fusion and scalar promotion; conversely, (B1) and (B2) follow a parallelism-enhancing strategy based on the opposite transformations: loop fission and privatization. Since the appropriate sequence is not the same in each case, the optimal strategy must be flexible enough to select either option.

Finally, any optimization strategy has an important impact on the order in which transformations are identified and applied. When optimizing applu and apsi, our methodology focused on individual transformations on separate loop nests. Only in the last step, dynamic analysis indicated that, to further improve performance, these loop nests must first be merged before applying performance-enhancing transformations. Of course, this is very much dependent on the strategy driving the optimization process, but an iterative feedback-directed approach is likely to be at least as demanding as a manual methodology, since it can potentially examine much longer transformation sequences.

5. IMPLEMENTATION

The whole infrastructure is implemented as a free (GPL) add-on to the Open64/ORC/EKOPath family of compilers [16,66]. Optimization is performed in two runs of the compiler, with one intermediate run of our tool, using intermediate dumps of the intermediate representation (the `.N` files) as shown in Figure 27. It thus natively supports the generation of IA64 code. The whole infrastructure compiles with GCC3.4 and is compatible with PathScale EKOPath [66] native code generator for AMD64 and IA32. Thanks to third-party tools based on Open64, this framework supports source-to-source optimization, using the robust C unparser of Berkeley UPC [67], and planning a port of the Fortran90 unparser from Open64/SL [68]. It contains 3 main tools in addition to Open64: WRaP-IT which extracts SCoPs and build their polyhedral representation, URUK which performs program transformations in the polyhedral representation, and URGenT the code generator associated with the polyhedral representation[8].

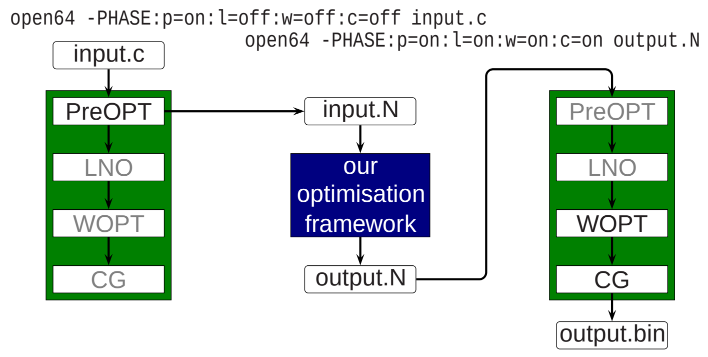

open64 -PHASE:p=on:l=off:w=off:c=off input.c
open64 -PHASE:p=on:l=on:w=on:c=on output.N
input.c
PreOPT
LNO
WOPT
CG
input.N
our optimisation framework
output.N
PreOPT
LNO
WOPT
CG
output.bin

Fig. 27. Optimisation process

5.1. WRaP-IT: WHIRL Represented as Polyhedra — Interface Tool

WRaP-IT is an interface tool built on top of Open64 which converts the WHIRL — the compiler's hierarchical intermediate representation — to an augmented polyhedral representation, maintaining a correspondence between matrices in SCoP descriptions with the symbol table and syntax tree. Although WRaP-IT is still a prototype, it proved to be robust; the whole source-to-polyhedra-to-source conversion (without any intermediate loop transformation) was successfully applied in 34 seconds in average per benchmark on a 512MB 1GHz Pentium III machine.

Implemented within the modern infrastructure of Open64, WRaP-IT benefits from interprocedural analysis and pre-optimization phases such as function inlining, interprocedural constant propagation, loop normalization, integer comparison normalization, dead-code and `goto` elimination, and induction variable substitution. Our tool extracts large and representative SCoPs for SPECfp2000 benchmarks: on average, 88% of the statements belong to a SCoP containing at least one loop (69).

To refine these statistics, Figures 28 and 29 describe the SCoP breakdown for each benchmark with respect to instruction count and maximal loop nesting depth, respectively. These numbers confirm the lexical importance of code that can be represented in our model, and set a well defined scalability target for the (most of the time exponential) polyhedral computations associated with program analyses and transformations.

[8] These tools can be downloaded from `http://www.lri.fr/~girbal/site_wrapit`.

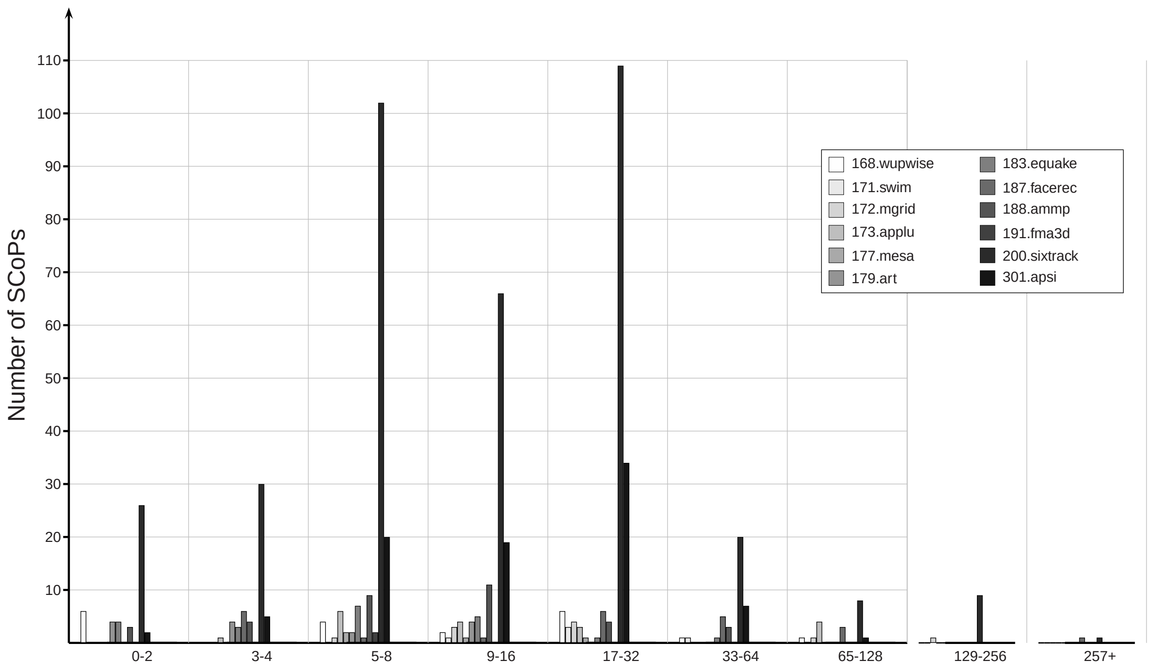


Fig. 28. SCoP size (instructions)

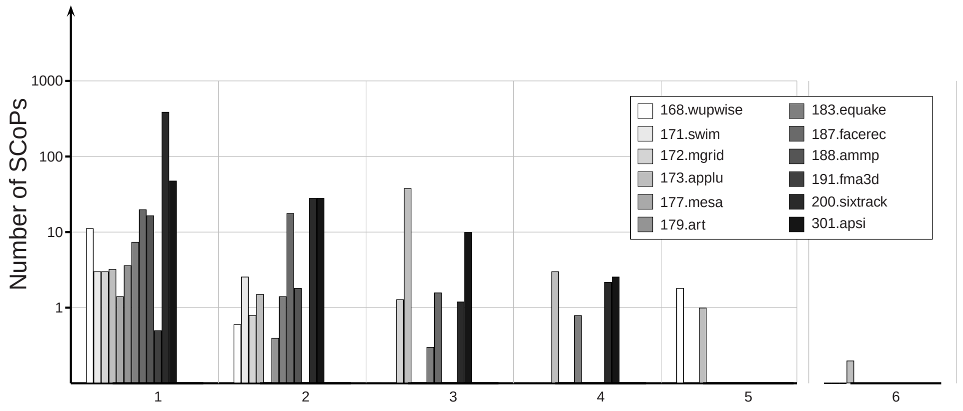


Fig. 29. SCoP depth

To refine this coverage study, we computed the SCoP breakdown with respect to effective execution time. We conducted statistical sampling measurements, using the `oprofile` portable performance monitoring framework. Figure 30 gather the execution time percentage associated with each consecutive block of source statements (over 2.5% execution time). The penultimate column, #SCoPs, gives the number of SCoPs covering this code block: the lower the better. The last column shows the maximal loop nesting depth in those SCoPs and the actual loop nesting depth in the procedure; when the two numbers differ, some enclosing loops are not considered static control. In many cases, *a single full-depth* SCoP is sufficient to cover the whole block of "hot" instructions, showing that polyhedral transformations will be fully applicable to this code block. These results are very encouraging, yet far from sufficient in the context of general-purpose applications. This motivates further research in extending the applicability of polyhedral techniques to "sparsely irregular" code. Inlining was disabled to isolate SCoP coverage in each source code function.[9]

5.2. URUK: Unified Representation Universal Kernel

URUK is the key software component: it performs program transformations within the WRaP (polyhedral) representation. A scripting language, defines transformations and enables the composition of new transformations. Each transformation is built upon a set of elementary actions, the *constructors* (See Section 3).

Figure 31 shows the definition of the Move constructor, and Figure 32 defines the FISSION transformation based on this constructor. This syntax is preprocessed to overloaded C++ code, offering a high-level semantics to manipulate the polyhedral representation. It takes less than one hour for an URUK expert to implement a complex transformation like tiling of imperfectly nested loops with prolog/epilog generation and legality checks, and to have this transformation work on real benchmarks without errors.

Transformation composition is very natural in the URUK syntax. Figure 33 shows how simple it is to implement tiling from the composition of strip-mining and interchange primitives, hiding all the details associated with remainder loop management and legality checking.

5.3. URDeps: URUK Dependence Analysis

An important feature of URUK is the ability to perform transformations *without mandatory intermediate validity checks, and without reference to the syntactic program*. This allows to compute dependence information

[9] We left out 6 SPECfp2000 benchmarks due to the (current) lack of support in our analyzer for function pointers and pointer arithmetic.

	File	Function	Source Lines	%Time	#SCoPs	SCoP Depth / Actual Depth
168.wupwise	zaxpy.f	zaxpy	11–32	20.6%	2	1/1
	zcopy.f	zcopy	11–24	8.3%	1	1/1
	zgemm.f	zgemm	236–271	47.5%	7	3/3
171.swim	swim.f	main	114–119	5.6%	1	2/2
	swim.f	calc1	261–269	26.3%	1	2/2
	swim.f	calc2	315–325	36.8%	1	2/2
	swim.f	calc3	397–405	29.2%	1	2/2
172.mgrid	mgrid.f	psinv	149–166	27.1%	1	3/3
	mgrid.f	resid	189–206	62.1%	1	3/3
	mgrid.f	rprj3	230–250	4.3%	1	3/3
	mgrid.f	interp	270–314	3.4%	1	3/3
173.applu	applu.f	blts	553–624	15.5%	1	6/6
	applu.f	buts	659–735	21.8%	1	6/6
	applu.f	jacld	1669–2013	17.3%	1	3/3
	applu.f	jacu	2088–2336	12.6%	1	3/3
	applu.f	rhs	2610–3068	20.2%	1	4/4
183.equake	quake.c	main	435–478	99%	4	2/3
187.facerec	cfftb.f90	passb4	266–310	35.6%	1	2/2
	gaborRoutines.f90	GaborTrafo	102–132	19.2%	2	2/2
	graphRoutines.f90	LocalMove	392–410	18.7%	2	0/4
	graphRoutines.f90	TopCostFct	451–616	8.23%	1	0/0
200.sixtrack	thin6d.f	thin6d	180–186	15.2%	1	1/3
	thin6d.f	thin6d	216–227	3.7%	1	1/3
	thin6d.f	thin6d	230–244	8.9%	3	1/3
	thin6d.f	thin6d	267–287	8.2%	2	1/3
	thin6d.f	thin6d	465–477	6.3%	1	1/3
	thin6d.f	thin6d	560–588	54.8%	1	2/4
301.apsi	apsi.f	dcdtz	1326–1354	4.3%	1	3/3
	apsi.f	dtdtz	1476–1499	4.3%	1	1/3
	apsi.f	dudtz	1637–1688	4.5%	1	3/3
	apsi.f	dvdtz	1779–1833	4.5%	1	3/3
	apsi.f	wcont	1878–1889	7.5%	1	1/3
	apsi.f	trid	3189–3205	5.9%	1	1/1
	apsi.f	smth	3443–3448	3.7%	1	1/1
	apsi.f	radb4	5295–5321	6.6%	2	2/2
	apsi.f	radbg	5453–5585	9.0%	3	3/3
	apsi.f	radf4	5912–5938	3.2%	2	2/2
	apsi.f	radfg	6189–6287	5.1%	2	3/3
	apsi.f	dkzmh	6407–6510	11.4%	8	1/3

Fig. 30. Execution time breakdown

and to perform validity checks on demand. Our dependence analysis computes an *exact* information whenever possible, i.e., whenever array references are affine (control structures are assumed affine in SCoPs). A list of convex polyhedra is computed for each pair of statements and for each depth, *considering the polyhedral representation only*, i.e., without reference to the initial syntactic program. This allows for *one-time dependence*

```
%transformation move
%param BetaPrefix P, Q
%param Offset o
%prereq P<=Q
%code
{
  foreach S in SCoP
    if (P<=S.Beta && Q<=S.Beta)
      S.Beta(P.dim())+=o;
    else if (P<=S.Beta && Q<<S.Beta)
      S.Beta(P.dim())+=o;
}
```

Fig. 31. Move constructor

```
%transformation fission
%param BetaPrefix P
%param Offset o, b
%code
{
  UrukVector Q=P;
  Q.enqueue(o); Q.enqueue(b);
  UrukVector R=P;
  R.enqueue(o+1);
  UT_move(P,Q,1).apply(SCoP);
  UT_move(R,R,-1).apply(SCoP);
}
```

Fig. 32. FISSION primitive

```
%transformation tile
%param BetaPrefix P
%param Integer k1
%param Integer k2
%prereq k1>0 && k2>0
%code
{
  Q=P.enclose();
  UT_stripmine(P,k2).apply(SCoP);
  UT_stripmine(Q,k1).apply(SCoP);
  UT_interchange(Q).apply(SCoP);
}
```

Fig. 33. TILE primitive

analysis before applying the transformation sequence, and *one-time check* at the very end, before code generation.

Let us briefly explain how this is achieved. Considering two distinct references to the same array in the program, at least one of them being a write, there is a dependence between them if their access functions coincide on some array element. Multiple refinement of this abstraction have been proposed, including dependence directions, distances, vectors and intervals [(6)] to improve the precision about the localization of the actual dependences between run-time statement instances. In the polyhedral model, it is possible to refine this definition further and to compute an *exact* dependence information, as soon as all array references are affine [(59)]. Exact dependences are classically captured by a system of affine inequalities over iteration vectors; when considering a syntactic loop nest, dependences at depth p between access functions F^S and F^T in statements S and T are exactly captured by the following union of polyhedra:

$$\mathcal{D}_{\mathrm{om}}^{S} \times \mathcal{D}_{\mathrm{om}}^{T} \cap \left\{ (\mathbf{i}^{S}, \mathbf{i}^{T}) \mid \mathrm{F}^{S}(\mathbf{i}^{S}) = \mathrm{F}^{T}(\mathbf{i}^{T}) \wedge \mathbf{i}^{S} \ll_{p} \mathbf{i}^{T} \right\},$$

where $\ll_p$ stands for the ordering of iteration vectors at depth p (i.e., equal component-wise up to depth $p-1$ and different at depth p).

Yet this characterization needs to be adapted to programs in a polyhedral representation, where no reference to a syntactic form is available, and where multiple schedule and domain transformations make the definition and tracking of the dependence information difficult. We thus replace the ordering on iteration vectors by the schedule-induced ordering, and split the constraints according to the decomposition of the schedule in our formalism. Two kinds of dependences at depth p can be characterized.

– Loop-carried dependence:

$$\beta^{S}_{0..p-1} = \beta^{T}_{0..p-1} \text{ and } (\mathrm{A}^{S}, \Gamma^{S})\mathbf{i}^{S} \ll_{p} (\mathrm{A}^{T}, \Gamma^{T})\mathbf{i}^{T}.$$

– Intra-loop dependence:

$$\begin{aligned} &\beta^{S}[0..p-1] = \beta^{T}[0..p-1], \\ &\qquad ((\mathrm{A}^{S}, \Gamma^{S})\mathbf{i}^{S})_{0..p-1} = ((\mathrm{A}^{T}, \Gamma^{T})\mathbf{i}^{T})_{0..p-1} \text{ and } \beta^{S}_{p} < \beta^{T}_{p}. \end{aligned}$$

Both kinds lead to a union of polyhedra that is systematically built, before any transformation is applied, for all pairs of references (to the same array) and for all depths (common to these references).

To solve the dependence tracking problem, we keep track of all modifications to the *structure* of the time and domain dimensions. In other words, we record any extension (dimension insertion, to implement, e.g., strip-mining) and any domain restriction (to implement, e.g., index-set splitting) into a work list, and we eventually traverse this list after all transformations have been applied to update dependence polyhedra accordingly. This scheme guarantees that the iteration domains and time dimensions correspond, after transformations, in the precomputed dependence information and in the modified polyhedral program representation.

Dependence checking is implemented by intersecting every dependence polyhedron with the *reversed* schedule of the transformed representation. If any such intersection is non-empty, the resulting polyhedron captures the *exact set of dependence violations*. This step allows to derive the exact set of iteration vector pairs associated with causality constraints violations. Based on this strong property, our implementation reports any dependence violation as a list of polyhedra; this report is very useful for automatic filtering of transformations in an iterative optimization framework, and as an optimization aid for the interactive user of URUK.

Interestingly, our formalism allows both dependence computation and checking to be simplified, relying on scalar comparisons on the β vectors to short-circuit complex polyhedral operations on inner depths. This optimization yields impressive speedups, due to the block-structured nature of most real-world schedules. The next section will explore such a real-world example and show a good scalability of this aggressive analysis.

5.4. URGenT: URUK Generation Tool

After polyhedral transformations, the (re)generation of imperative loop structures is the last step. It has a strong impact on the target code quality: we must ensure that no redundant guard or complex loop bound spoils performance gains achieved thanks to polyhedral transformations. We used the Chunky Loop Generator (CLooG), a recent Quilleré et al. method (23) with some additional improvements to guarantee the absence of duplicated control (22), to generate efficient control for full SPECfp2000 benchmarks and for SCoPs with more than 1700 statements. Polyhedral transformations make code generation particularly difficult because they create a large set of complex overlapping polyhedra that need to be scanned with do-loops (70, 23, 21, 22). Because of the added complexity introduced, we had to design URGenT, a major reengineering of CLooG taking advantage of the normalization rules of our representation to bring exponential improvements to execution time and memory usage. The generated code size and quality greatly improved, making it better than typically hand-tuned code. (38) details how URGenT succeeds in producing efficient code for a realistic optimization case-study in a few seconds only.

6. SEMI-AUTOMATIC OPTIMIZATION

Let us detail the application of our tools to the semi-automatic optimization of the swim benchmark, to show the effectiveness of the approach and the performance of the implementation on a representative benchmark. We target a 32bit and a 64bit architecture: an AMD Athlon XP 2800+ (Barton) at 2.08GHz with 512KB L2 cache and 512MB single-channel DDR SDRAM (running Mandriva Linux 10.1, kernel version 2.6.8), and a AMD Athlon 64 3400+ (ClawHammer) at 2.2GHz zith 1MB L2 cache and single-channel 1GB DDR SDRAM (running Debian GNU/Linux Sid, kernel version 2.6.11). The swim benchmark was chosen because it easily illustrates the benefits of implementing a sequence of transformations in our framework, compared to manual optimization of the program text, and because it presents a reasonably large SCoP to evaluate robustness (after fully inlining the three hot subroutines).

Figure 34 shows the transformation sequence for swim, implemented as a script for URUK. Syntactic compilation frameworks like PathScale EKOPath, Intel ICC and KAP implement a simplified form of this transformation sequence on swim, missing the fusion with the nested loops in subroutine `calc3`, which requires a very complex combination of loop peeling, code motion and three-level shifting. In addition, such a sequence is highly specific to swim and cannot be easily adapted, extended or reordered to handle other programs: due to syntactic restrictions of individual transformations, the sequence has to be considered as a whole since the effect of any of its components can hamper the application and profitability of the entire sequence. Conversely, within our semi-automatic framework, the sequence can be built without concern about the impact of a transformation on the applicability of subsequent ones. We demonstrate this through the dedicated transformation sequence in Figure 34.

This URUK script operates on the `swim.N` file, a persistent store of the compiler's intermediate representation, dumped by EKOPath after interprocedural analysis and pre-optimization. At this step, EKOPath is directed to inline the three dominant functions of the benchmark, `calc1`, `calc2` and `calc3` (passing these function names to the `-INLINE` optimization switch). WRaP-IT processes the resulting file, extracting several SCoPs, the significant one being a section of 421 lines of code — 112 instructions in the polyhedral representation — in consecutive loop nests within the `main` function. Transformations in Figure 34 apply to this SCoP.

Labels of the form C*x*L*y* denote statement *y* of procedure calc*x*. Given a vector v and an integer $r \leq \dim v$, enclose(v,r) returns the prefix of length $\dim v - r$ of vector v (r is equal to 1 if absent). The primitives involved are the following: `motion` translates the β component (of a set of statements), `shift` translates the Γ matrix; `peel` splits the domain of a statement according to a given constraint and creates two labels with suffixes `_1` and `_2`; `stripmine` and `interchange` are self-explanatory; and `time`-prefixed primitives mimic the effect of their iteration domain counterparts on time dimensions. Loop fusion is a special case of the `motion` primitive. Tiling is decomposed into double strip-mining and interchange. Loop unrolling (`fullunroll`) is delayed to the code generation phase.

Notice the script is quite concise, although the generated code is much more complex than the original swim benchmark (due to versioning, peeling, strip-mining and unrolling). In particular, loop fusion is straightforward, despite the fused loops domains differ by one or two iterations (due to peeling), and despite the additional multi-level shifting steps.

```
# Avoid spurious versioning
addContext(C1L1,’ITMAX>=9’)
addContext(C1L1,’doloop_ub>=ITMAX’)
addContext(C1L1,’doloop_ub<=ITMAX’)
addContext(C1L1,’N>=500’)             # Peel and shift to enable fusion
addContext(C1L1,’M>=500’)             peel(enclose(C3L1,2),’3’)
addContext(C1L1,’MNMIN>=500’)         peel(enclose(C3L1_2,2),’N-3’)
addContext(C1L1,’MNMIN<=M’)           peel(enclose(C3L1_2_1,1),’3’)
addContext(C1L1,’MNMIN<=N’)           peel(enclose(C3L1_2_1_2,1),’M-3’)
addContext(C1L1,’M<=N’)               peel(enclose(C1L1,2),’2’)
addContext(C1L1,’M>=N’)               peel(enclose(C1L1_2,2),’N-2’)
                                      peel(enclose(C1L1_2_1,1),’2’)
# Move and shift calc3 backwards      peel(enclose(C1L1_2_1_2,1),’M-2’)
shift(enclose(C3L1),{’1’,’0’,’0’})    peel(enclose(C2L1,2),’1’)
shift(enclose(C3L10),{’1’,’0’})       peel(enclose(C2L1_2,2),’N-1’)
shift(enclose(C3L11),{’1’,’0’})       peel(enclose(C2L1_2_1,1),’3’)
shift(C3L12,{’1’})                    peel(enclose(C2L1_2_1_2,1),’M-3’)
shift(C3L13,{’1’})                    shift(enclose(C1L1_2_1_2_1),{’0’,’1’,’1’})
shift(C3L14,{’1’})                    shift(enclose(C2L1_2_1_2_1),{’0’,’2’,’2’})
shift(C3L15,{’1’})
shift(C3L16,{’1’})                    # Double fusion of the three nests
shift(C3L17,{’1’})                    motion(enclose(C2L1_2_1_2_1),TARGET_2_1_2_1)
motion(enclose(C3L1),BLOOP)           motion(enclose(C1L1_2_1_2_1),C2L1_2_1_2_1)
motion(enclose(C3L10),BLOOP)          motion(enclose(C3L1_2_1_2_1),C1L1_2_1_2_1)
motion(enclose(C3L11),BLOOP)
motion(C3L12,BLOOP)                   # Register blocking and unrolling (factor 2)
motion(C3L13,BLOOP)                   stripmine(enclose(C3L1_2_1_2_1,2),2,2)
motion(C3L14,BLOOP)                   stripmine(enclose(C3L1_2_1_2_1,1),4,2)
motion(C3L15,BLOOP)                   interchange(enclose(C3L1_2_1_2_1,2))
motion(C3L16,BLOOP)                   fullunroll(enclose(C3L1_2_1_2_1,2))
motion(C3L17,BLOOP)                   fullunroll(enclose(C3L1_2_1_2_1,1))
```

Fig. 34. URUK script to optimize swim

The application of this script is fully automatic; it produces a significantly larger code of 2267 lines, roughly one third of them being naive scalar copies to map schedule iterators to domain ones, fully eliminated by copy-propagation in the subsequent run of EKOPath or Open64. This is not surprising since most transformations in the script require domain decomposition, either explicitly (peeling) or implicitly (shifting prolog/epilog, at code generation). It takes 39s to apply the whole transformation sequence up to native code generation on a 2.08GHz AthlonXP. Transformation time is dominated by back-end compilation (22s). Polyhedral code generation takes only 4s. Exact polyhedral dependence analysis (computation and checking) is acceptable (12s). Applying the transformation sequence itself is negligible. These execution times are very encouraging, given the complex overlap of peeled polyhedra in the code generation phase, and since the full dependence graph captures the exact dependence information for the 215 array references in the SCoP at every loop depth (maximum 5 after tiling), yielding a total of 441 dependence matrices. The result of this application is a new intermediate representation file, sent to EKOPath or Open64 for further scalar optimizations and native code generation.

Compared to the *peak performance attainable by the best available compiler*, PathScale EKOPath (V2.1) with the best optimization flags,[10] our tool achieves **32% speedup on Athlon XP and 38% speedup on Athlon 64**. Compared to the *base SPEC* performance numbers,[11] our optimization achieves **51% speedup on Athlon XP and 92% speedup on Athlon 64**. We are not aware of any other optimization effort — manual or automatic — that brought swim to this level of performance on x86 processors.[12]

We do not have results on IA64 yet, due to back-end instability issues in Open64 (with large basic blocks). We expect an additional level of tiling and more aggressive unrolling will be profitable (due to the improved TLB management, better load/store bandwitdh and larger register file on Itanium 2 processors).

Additional transformations need to be implemented in URUK to authorize semi-automatic optimization of a larger range of benchmarks. In addition, further work on the iterative optimization driver is being conducted to make this process more automatic and avoid the manual implementation of an URUK script. Yet the tool in its current state is of great use for the optimization expert who wishes to quickly evaluate complex sequences of transformations.

7. RELATED WORK

The most topical works associated with each technical issue and contribution have been discussed in the relevant specific sections. Here, we will only survey the former efforts in designing an advanced loop nest transformation infrastructure and representation framework.

Most loop restructuring compilers introduced syntax-based models and intermediate representations. ParaScope (40) and Polaris (41) are depen-

[10] Athlon XP: `-m32 -Ofast -OPT:ro=2:Olimit=0:div_split=on:alias=typed -LNO:fusion=2:prefetch=2 -fno-math-errno`; Athlon 64 (in 64 bits mode): `-march=athlon64 -LNO:fusion=2:prefetch=2 -m64 -Ofast -msse2 -lmpath`; `pathf90` always outperformed Intel ICC by a small percentage.

[11] With optimization flag `-Ofast`.

[12] Notice we consider the SPEC 2000 version of swim, much harder to optimize through loop fusion than the SPEC 95 version.

dence based, source-to-source parallelizers for Fortran. KAP [18] is closely related to these academic tools.

SUIF [42] is a platform for implementing advanced compiler prototypes. Although some polyhedral works have been built on SUIF [11, 13], they do not address the composition issue and rely on a weaker code generation method. PIPS [71] is one of the most complete loop restructuring compiler, implementing polyhedral analyses and transformations (including affine scheduling) and interprocedural analyses (array regions, alias). It uses a syntax tree extended with polyhedral annotations, but not a unified polyhedral representation.

Closer to our work, the MARS compiler [20] has been applied to iterative optimization [72]; this compiler's goal is to unify classical dependence-based loop transformations with data storage optimizations. However, the MARS intermediate representation only captures part of the loop-specific information (domains and access functions): it lacks the characterization of iteration orderings through multidimensional affine schedules. Recently, a similar unified representation has been applied to the optimization of compute-intensive Java programs, combining machine learning and iterative optimization [3]; again, despite the unification of multiple transformations, the lack of multidimensional affine schedules hampers the ability to perform long sequences of transformations and complicates the characterization and traversal of the search space, ultimately limiting performance improvements.

To date, the most thorough application of the polyhedral representation was the Petit dependence analyzer and loop restructuring tool [10], based on the Omega library [73]. It provides space-time mappings for iteration reordering, and it shares our emphasis on per-statement transformations, but it is intended as a research tool for small kernels only. Our representation — whose foundations were presented in [25] — improves on the polyhedral representation proposed by [10], and this paper explains how and why it is the first one that enables the composition of polyhedral generalizations of classical loop transformations, decoupled from any syntactic form of the program. We show how classical transformations like loop fusion or tiling can be composed in any order and generalized to imperfectly-nested loops with complex domains, without intermediate translation to a syntactic form (which leads to code size explosion). Eventually, we use a code generation technique suitable to a polyhedral representation that is again significantly more robust than the code generation proposed in the Omega library [22].

8. CONCLUSIONS

The ability to perform numerous compositions of program transformations is driven by the development of iterative optimization environments, and motivated through the manual optimization of standard numerical benchmarks. From these experiments, we show that current compilers are challenged by the complexity of aggressive loop optimization sequences. We believe that little improvements can be expected without redesigning the compilation infrastructure for compositionality and richer search space structure.

We presented a polyhedral framework that enables the composition of long sequences of program transformations. Coupled with a robust code generator, our method avoids the typical restrictions and code bloat of long compositions of program transformations. These techniques have been implemented in the Open64/ORC/EKOPath compiler and applied to the swim benchmark automatically. We have also shown that our framework opens up new directions for searching for complex transformation sequences for automatic or semi-automatic optimization or parallelization.

ACKNOWLEDGMENTS

This work is supported by a grant from the French Ministry of Research (RNTL COP) and INRIA, CEA and HP France fellowships. We are also indebted to many people who helped building the tool set, including Saurabh Sharma who initiated the implementation, Roy Ju and Sun Chan from Intel, John Mellor-Crummey and Nathan Tallent from Rice University, Greg Lindahl and Fred Chow from PathScale, and the UPC team at the University of California Berkeley.

REFERENCES

1. T. Kisuki, P. Knijnenburg, M. O'Boyle, and H. Wijshoff, Iterative compilation in program optimization, *Proc. CPC'10 (Compilers for Parallel Computers)*, pp. 35–44 (2000).
2. K. D. Cooper, D. Subramanian, and L. Torczon, Adaptive optimizing compilers for the 21st century, *J. of Supercomputing* (2002).
3. S. Long and M. O'Boyle, Adaptive java optimisation using instance-based learning., *ACM Intl. Conf. on Supercomputing (ICS'04)*, pp. 237–246, St-Malo, France (June 2004).
4. D. Parello, O. Temam, and J.-M. Verdun, On Increasing Architecture Awareness in Program Optimizations to Bridge the Gap between Peak and Sustained Processor Performance? Matrix-Multiply Revisited, *SuperComputing'02*, Baltimore, Maryland (November 2002).
5. D. Parello, O. Temam, A. Cohen, and J.-M. Verdun, Towards a Systematic, Pragmatic and Architecture-Aware Program Optimization Process for Complex Processors, *ACM Supercomputing'04*, p. 15, Pittsburgh, Pennsylvania (November 2004),

6. M. J. Wolfe, *High Performance Compilers for Parallel Computing*, Addison-Wesley (1996).
7. A. Cohen, S. Girbal, D. Parello, M. Sigler, O. Temam, and N. Vasilache, Facilitating the Search for Compositions of Program Transformations, *ACM Intl. Conf. on Supercomputing (ICS'05)*, pp. 151–160, Boston, Massachusetts (June 2005),
8. P. Feautrier, Some Efficient Solutions to the Affine Scheduling Problem, Part II, multidimensional time, *Intl. J. of Parallel Programming*, **21**(6):389–420 (December 1992), see also Part I, one dimensional time, 21(5):315–348.
9. M. E. Wolf, *Improving Locality and Parallelism in Nested Loops*, Ph.D. thesis, Stanford University (August 1992), published as CSL-TR-92-538.
10. W. Kelly, *Optimization within a Unified Transformation Framework*, Technical Report CS-TR-3725, University of Maryland (1996).
11. A. W. Lim and M. S. Lam, Communication-Free Parallelization via Affine Transformations, *24^{th}ACM Symp. on Principles of Programming Languages*, pp. 201–214, Paris, France (jan 1997).
12. N. Ahmed, N. Mateev, and K. Pingali, Synthesizing transformations for locality enhancement of imperfectly-nested loop nests, *ACM Supercomputing'00* (May 2000).
13. A. W. Lim, S.-W. Liao, and M. S. Lam, Blocking and array contraction across arbitrarily nested loops using affine partitioning, *ACM Symp. on Principles and Practice of Parallel Programming (PPoPP'01)*, pp. 102–112 (2001).
14. W. Pugh, Uniform techniques for loop optimization, *ACM Intl. Conf. on Supercomputing (ICS'91)*, pp. 341–352, Cologne, Germany (June 1991).
15. W. Li and K. Pingali, A singular loop transformation framework based on non-singular matrices, *Intl. J. of Parallel Programming*, **22**(2):183–205 (April 1994).
16. Open Research Compiler, `http://ipf-orc.sourceforge.net`.
17. A. Phansalkar, A. Joshi, L. Eeckhout, and L. John, *Four generations of SPEC CPU benchmarks: what has changed and what has not*, Technical Report TR-041026-01-1, University of Texas Austin (2004).
18. KAP C/OpenMP for Tru64 UNIX and KAP DEC Fortran for Digital UNIX, `http://www.hp.com/techsevers/software/kap.html`.
19. E. Visser, Stratego: A Language for Program Transformation based on Rewriting Strategies. System Description of Stratego 0.5, A. Middeldorp (ed.), *Rewriting Techniques and Applications (RTA'01)*, *Lecture Notes in Computer Science*, Vol. 2051, pp. 357–361, Springer-Verlag (May 2001).
20. M. O'Boyle, MARS: a Distributed Memory Approach to Shared Memory Compilation, *Proc. Language, Compilers and Runtime Systems for Scalable Computing*, Springer-Verlag, Pittsburgh (May 1998).
21. C. Bastoul, Efficient code generation for automatic parallelization and optimization, *ISPDC'2 IEEE International Symposium on Parallel and Distributed Computing*, Ljubjana, Slovenia (October 2003).
22. C. Bastoul, Code Generation in the Polyhedral Model Is Easier Than You Think, *Parallel Architectures and Compilation Techniques (PACT'04)*, Antibes, France (September 2004).

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

24. G.-R. Perrin and A. Darte (eds.), *The Data Parallel Programming Model*, number 1132 in LNCS, Springer-Verlag (1996).

25. A. Cohen, S. Girbal, and O. Temam, A Polyhedral Approach to Ease the Composition of Program Transformations, *Euro-Par'04*, number 3149 in LNCS, pp. 292–303, Springer-Verlag, Pisa, Italy (August 2004),

26. R. Triolet, P. Feautrier, and P. Jouvelot, Automatic parallelization of Fortran programs in the presence of procedure calls, *Proc. of the 1st European Symp. on Programming (ESOP'86)*, number 213 in LNCS, pp. 210–222, Springer-Verlag (March 1986).

27. M. Griebl and J.-F. Collard, Generation of Synchronous Code for Automatic Parallelization of `while` Loops, S. Haridi, K. Ali, and P. Magnusson (eds.), *EuroPar'95*, *LNCS*, Vol. 966, pp. 315–326, Springer-Verlag (1995).

28. J.-F. Collard, Automatic parallelization of While-Loops using speculative execution, *Intl. J. of Parallel Programming*, **23**(2):191–219 (April 1995).

29. D. G. Wonnacott, *Constraint-Based Array Dependence Analysis*, Ph.D. thesis, University of Maryland (1995).

30. B. Creusillet, *Array Region Analyses and Applications*, Ph.D. thesis, École Nationale Supérieure des Mines de Paris (ENSMP), France (December 1996).

31. D. Barthou, J.-F. Collard, and P. Feautrier, Fuzzy Array Dataflow Analysis, *J. of Parallel and Distributed Computing*, **40**:210–226 (1997).

32. L. Rauchwerger and D. Padua, The LRPD Test: Speculative Run–Time Parallelization of Loops with Privatization and Reduction Parallelization, *IEEE Transactions on Parallel and Distributed Systems, Special Issue on Compilers and Languages for Parallel and Distributed Computers*, **10**(2):160–180 (1999).

33. D. Barthou, A. Cohen, and J.-F. Collard, Maximal Static Expansion, *Intl. J. of Parallel Programming*, **28**(3):213–243 (June 2000),

34. A. Cohen, *Program Analysis and Transformation: from the Polytope Model to Formal Languages*, PhD Thesis, Université de Versailles, France (December 1999),

35. J.-F. Collard, *Reasoning About Program Transformations*, Springer-Verlag (2002).

36. A. Darte, Y. Robert, and F. Vivien, *Scheduling and Automatic Parallelization*, Birkhaüser, Boston (2000).

37. A. Darte and Y. Robert, Mapping uniform loop nests onto distributed memory architectures, *Parallel Computing*, **20**(5):679–710 (1994).

38. N. Vasilache, C. Bastoul, and A. Cohen, Polyhedral Code Generation in the Real World, *Proceedings of the International Conference on Compiler Construction (ETAPS CC'06)*, LNCS, Springer-Verlag, Vienna, Austria (March 2006), to appear.

39. J. Allen and K. Kennedy, Automatic Translation of Fortran Programs to Vector Form, *ACM Trans. on Programming Languages and Systems*, **9**(4):491–542 (October 1987).

40. K. D. Cooper, M. W. Hall, R. T. Hood, K. Kennedy, K. S. McKinley, J. M. Mellor-Crummey, L. Torczon, and S. K. Warren, The ParaScope Parallel Programming Environment, *Proceedings of the IEEE*, **81**(2):244–263 (1993).

41. W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, W. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford, Parallel Programming with Polaris, *IEEE Computer*, **29**(12):78–82 (December 1996).

42. M. Hall et al., Maximizing Multiprocessor Performance with the SUIF Compiler, *IEEE Computer*, **29**(12):84–89 (December 1996).

43. S. Carr, C. Ding, and P. Sweany, Improving Software Pipelining With Unroll-and-Jam, *Proceedings of the 29th Hawaii Intl. Conf. on System Sciences (HICSS'96) Volume 1: Software Technology and Architecture*, IEEE Computer Society (1996).

44. A. J. C. Bik, M. Girkar, P. M. Grey, and X. Tian, Automatic Intra-Register Vectorization for the Intel Architecture, *Intl. J. of Parallel Programming*, **30**(2):65–98 (2002).

45. D. Naishlos, Autovectorization in GCC, *Proceedings of the 2004 GCC Developers Summit*, pp. 105–118 (2004), `http://www.gccsummit.org/2004`.

46. A. E. Eichenberger, P. Wu, and K. O'Brien, Vectorization for SIMD architectures with alignment constraints, *ACM Symp. on Programming Language Design and Implementation (PLDI '04)*, pp. 82–93 (2004).

47. D. E. Maydan, S. P. Amarasinghe, and M. S. Lam, Array Dataflow Analysis and its Use in Array Privatization, *20th ACM Symp. on Principles of Programming Languages*, pp. 2–15, Charleston, South Carolina (January 1993).

48. P. Tu and D. Padua, Automatic Array Privatization, *6th Workshop on Languages and Compilers for Parallel Computing*, number 768 in LNCS, pp. 500–521, Portland, Oregon (August 1993).

49. U. Banerjee, *Dependence Analysis for Supercomputing*, Kluwer Academic Publishers, Boston (1988).

50. W. Pugh, The Omega test: a fast and practical integer programming algorithm for dependence analysis, *ACM/IEEE Conf. on Supercomputing*, pp. 4–13, Albuquerque (August 1991).

51. J. Xue, Automating non-unimodular loop transformations for massive parallelism, *Parallel Computing*, **20**(5):711–728 (1994).

52. A.-C. Guillou, F. Quilleré, P. Quinton, S. Rajopadhye, and T. Risset, Hardware Design Methodology with the Alpha Language, *FDL'01*, Lyon, France (September 2001).

53. R. Schreiber, S. Aditya, B. Rau, V. Kathail, S. Mahlke, S. Abraham, and G. Snider, *High-level synthesis of nonprogrammable hardware accelerators*, Technical report, Hewlett-Packard (May 2000).

54. P. Feautrier, Array Expansion, *ACM Intl. Conf. on Supercomputing*, pp. 429–441, St. Malo, France (July 1988).

55. D. Barthou, A. Cohen, and J.-F. Collard, Maximal Static Expansion, *25th ACM Symp. on Principles of Programming Languages (PoPL'98)*, pp. 98–106, San Diego, California (January 1998),

56. V. Lefebvre and P. Feautrier, Automatic Storage Management for Parallel Programs, *Parallel Computing*, **24**(3):649–671 (1998).

57. M. M. Strout, L. Carter, J. Ferrante, and B. Simon, Schedule-Independant Storage Mapping for Loops, *ACM Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'98)*, 8 (1998).

58. F. Quilleré and S. Rajopadhye, *Optimizing Memory Usage in the Polyhedral Model*, Technical Report 1228, Institut de Recherche en Informatique et Systèmes Aléatoires, Université de Rennes, France (January 1999).

59. P. Feautrier, Dataflow Analysis of Scalar and Array References, *Intl. J. of Parallel Programming*, **20**(1):23–53 (February 1991).

60. J.-F. Collard, D. Barthou, and P. Feautrier, Fuzzy array dataflow analysis, *ACM Symp. on Principles and Practice of Parallel Programming*, pp. 92–102, Santa Barbara, CA (July 1995).

61. D. Wonnacott and W. Pugh, Nonlinear array dependence analysis, *Proc. Third Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers* (1995), troy, New York.

62. S. Rus, D. Zhang, and L. Rauchwerger, The Value Evolution Graph and its Use in Memory Reference Analysis, *Parallel Architectures and Compilation Techniques (PACT'04)*, IEEE Computer Society, Antibes, France (2004).

63. C. Bastoul and P. Feautrier, More Legal Transformations for Locality, *Euro-Par'10*, number 3149 in LNCS, pp. 272–283, Pisa (August 2004).

64. C. Bastoul and P. Feautrier, Improving data locality by chunking, *CC Intl. Conf. on Compiler Construction*, number 2622 in LNCS, pp. 320–335, Warsaw, Poland (april 2003).

65. Standard Performance Evaluation Corp., `http://www.spec.org`.

66. F. Chow, Maximizing application performance through interprocedural optimization with the PathScale EKO compiler suite, `http://www.pathscale.com/whitepapers.html` (August 2004).

67. C. Bell, W.-Y. Chen, D. Bonachea, and K. Yelick, Evaluating Support for Global Address Space Languages on the Cray X1, *ACM Intl. Conf. on Supercomputing (ICS'04)*, St-Malo, France (June 2004).

68. C. Coarfa, F. Zhao, N. Tallent, J. Mellor-Crummey, and Y. Dotsenko, Open-source Compiler Technology for Source-to-Source Optimization, `http://www.cs.rice.edu/~johnmc/research.html` (project page).

69. C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam, Putting Polyhedral Loop Transformations to Work, *Workshop on Languages and Compilers for Parallel Computing (LCPC'03)*, LNCS, pp. 23–30, Springer-Verlag, College Station, Texas (October 2003),

70. C. Ancourt and F. Irigoin, Scanning Polyhedra with DO Loop, *ACM Symp. on Principles and Practice of Parallel Programming (PPoPP'91)*, pp. 39–50 (June 1991).

71. F. Irigoin, P. Jouvelot, and R. Triolet, Semantical Interprocedural Parallelization: An Overview of the PIPS Project, *ACM Intl. Conf. on Supercomputing (ICS'91)*, Cologne, Germany (June 1991).

72. T. Kisuki, P. Knijnenburg, K. Gallivan, and M. O'Boyle, The Effect of Cache Models on Iterative Compilation for Combined Tiling and Unrolling, *Parallel Architectures and Compilation Techniques (PACT'00)*, IEEE Computer Society (October 2001).

73. W. Kelly, W. Pugh, and E. Rosser, Code generation for multiple mappings, *Frontiers'95 Symp. on the frontiers of massively parallel computation*, McLean (1995).