Facilitating the Search for Compositions of Program Transformations

Albert Cohen[1] Sylvain Girbal[1 2] David Parello[1 3]
Marc Sigler[1] Olivier Temam[1] Nicolas Vasilache[1]

[1] ALCHEMY Group, INRIA Futurs and LRI, Paris-Sud University, and HiPEAC network [2] CEA LIST, Saclay [3] HP France

Abstract

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, which requires the optimizer to combine multiple program transformations. Whether these transformations are selected through static analysis and models, runtime feedback, or both, the underlying infrastructure must have the ability to perform long and complex compositions of program transformations in a flexible manner. Existing compilers are ill-equipped to perform that task because of rigid phase ordering, fragile selection rules using pattern matching, and cumbersome expression of loop transformations on syntax trees. Moreover, iterative optimization emerges as a pragmatic and general means to select an optimization strategy via machine learning and operations research. Searching for the composition of dozens of complex, dependent, parameterized transformations is a challenge for iterative approaches.

The purpose of this article is threefold: (1) to facilitate the automatic search for compositions of program transformations, introducing a richer framework which improves on classical polyhedral representations, suitable for iterative optimization on a simpler, structured search space, (2) to illustrate, using several examples, that syntactic code representations close to the operational semantics hamper the composition of transformations, and (3) that complex compositions of transformations can be necessary to achieve significant performance benefits. The proposed framework relies on a unified polyhedral representation of loops and statements. The key is to clearly separate four types of actions associated with program transformations: iteration domain, schedule, data layout and memory access functions modifications. The framework is implemented within the Open64/ORC compiler, aiming for native IA64, AMD64 and IA32 code generation, along with source-to-source optimization of Fortran90, C and C++.

1 Introduction

Both high-performance and embedded architectures include an increasing number of hardware components with complex runtime behavior, e.g., cache hierarchies (including write buffers, TLBs, miss address files, L1 and L2 prefetching…), branch predictors, trace cache, load/store queue speculation, and pipeline replays. Static compiler optimizations have a hard time coping with such hardware components and their complex interactions. The issues are (1) to properly identify the architectural phenomena, and (2) to perform the appropriate and possibly complex sequence of program transformations. For the first issue, iterative optimization [20, 26, 12] is emerging as a promising solution by proposing to assist static analysis with runtime information to guide program transformations. However, for the second issue, iterative optimization environments will fare no better than existing compilers on top of which they are currently implemented. The issue is that multiple architecture phenomena often occur simultaneously and interact together. As a result, multiple carefully combined and crafted program transformations can be necessary to improve performance [29, 28]. Whether these program transformations are found using static analysis or runtime information, the underlying compiler infrastructure must have the ability *to search for* and *to effectively perform* the proper *sequence of program transformations*. Up to now, this fact has been largely overlooked.

As of today, iterative optimization usually consists in choosing a rather small set of transformations, e.g., cache tiling, unrolling or array padding, and focusing on finding the best possible transformation parameters, e.g., tile size or unroll factor [19] using parameter search space techniques. However, complex hardware interplay cannot be solely addressed through the proper selection of transformations parameters. A recent comparative study of model-based versus empirical optimizations [36] indicates that many motivations for iterative optimization are irrelevant when the proper transformations are not available. O'Boyle et al. [19] and Cooper et al. [12] have also outlined that the ability to perform long sequences of composed transformations is key to the emergence of iterative optimization frameworks.

Clearly, there is a need for a compiler infrastructure that can apply complex and possibly long compositions of program transformations. Unfortunately, existing compiler infrastructures are ill-equipped for that task. By imposing phase ordering constraints [35], current compilers 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.

This article introduces a framework to easily search for and perform *compositions* of program transformations; this framework relies on a unified representation of loops and statements, the foundations of which where presented in [10], improving on classical polyhedral representations [13, 34, 17, 22, 1, 23]. Using this representation, 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, can be expressed as a set of simple matrix operations. Compared to the few attempts at expressing a large array of program transformations within the polyhedral model, the distinctive asset of our representation lies in the simplicity of the formalism to compose non-unimodular transformations across long, flexible sequences. Existing formalisms are designed for black-box optimization [13, 22, 1], and applying a classical loop transformation within them — as proposed in [34, 17] — 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 [35], with some extensions to singular transformations [21].

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…

Moreover, 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 new compositions of transformations for which no static model has yet been developed.

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. Using several SPEC benchmarks, Section 3 shows that complex compositions can be necessary to reach high performance, and shows how such compositions are easily implemented using our polyhedral representation. Section 4 briefly describes the implementation of our representation, of the associated transformation tool, and of the code generation technique (in Open64/ORC [27]). Section 5 validates these tools through the evaluation of a dedicated transformation sequence for one benchmark. Section 6 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. In Section 2.2, we present our polyhedral representation, in Section 2.3 how it can alleviate the limitations of the syntactic representation, in Section 2.4 how it can further facilitate the search for compositions of transformations, 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 within the syntactic intermediate representation (abstract syntax tree, three address code, SSA graph...), hence the term *syntactic* (or syntax-based) transformations. Note that this behavior is also shared by all previous matrix- or polyhedra-based frameworks.

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];
```

Figure 1. Introductory example

```
...
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];
...
```

Figure 2. Versioning after outer loop fusion

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 2. The number of program statements after each step is indicated in Figure 3.

The final code generated by the polyhedral representation will be similarly complicated, but this complexity does not show until the 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, so that neither its complexity nor its size vary significantly, whatever the number and nature of program transformations. The number of statements remains the same (until the code is generated), only the dimension of some matrices slightly increases, see Figure 3. 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 same matrix dimensions).

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 [31].

To illustrate this point we have attempted to perform the above program transformations targeting the Alpha 21364 EV7, using KAP C (V4.1) [16], one of the best production preprocessors available (source to source loop and array transformations). For starters, Figure 4 shows the performance achieved by KAP and by the main steps of the abrve tion 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.

2.2.2 *Iteration domains*

Strip-mining and loop unrolling only modify the iteration domain, i.e., the number of loops and iterations 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: $\{(i,j) \mid 0 \le i, i \le M-1, 0 \le j, j \le N-1\}$. 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

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

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: $jj \le j, j \le jj + B - 1$. 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 $\{(i, jj, j) \mid 0 \le j, j \le N-1, jj \le j, j \le jj+B-1, jj \mod B = 0, 0 \le i, i \le M-1\}$, and after introducing local variable jj_2 such that $jj = B \times jj_2$, the iteration domain becomes $\{(i, jj, j) \mid \exists jj_2, 0 \le j, j \le N-1, jj \le j, j \le jj + B - 1, jj = B \times jj_2, 0 \le i, i \le M-1\}$[2] 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):

$$\left[\begin{array}{ccc|c|cccc|c} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 & 0 & 0 & -1 \\ 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 63 \\ 0 & -1 & 0 & 64 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -64 & 0 & 0 & 0 & 0 & 0 \end{array}\right] \quad \begin{array}{l} 0 \le i \\ i \le M-1 \\ 0 \le j \\ j \le N-1 \\ jj \le j \\ j \le jj+63 \\ jj \le 64jj_2 \\ 64jj_2 \le jj \end{array}$$

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 $\mathcal{D}^S_{\text{om}}$ the matrix of n linear constraints ($\mathcal{D}^S_{\text{om}}$ has n rows and $d^S + d^S_{\text{lv}} + d_{\text{gp}} + 1$ columns). The iteration domain of S is defined by

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

2.2.3 *Schedules*

Feautrier [13], Kelly and Pugh [17], 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, 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 5, 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 5 corresponds to a different loop nest with different iterators.

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

Figure 5. 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, one can see that 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 the schedule or Θ.

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

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

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

Example: implementing loop interchange and tiling. As for unimodular transformations, applying a schedule-only program 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 5.

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 5.

Extending the representation to implement more transformations. For some transformations like statement-wise 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,

[2] The equation $jj = B \times jj_2$ is simply represented by two inequalities $jj \ge B \times jj_2$ and $jj \le B \times jj_2$.

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).

2.2.4 Access functions

Privatization modifies array accesses, i.e., array subscripts. For any array reference, a given iteration domain point 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 statement iteration domain nor the statement schedule.

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 affine 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 $\mathcal{L}_{\mathrm{hs}}^{S}$ and $\mathcal{R}_{\mathrm{hs}}^{S}$ 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 $\mathcal{D}_{\mathrm{om}}^{S}$ 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

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

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 $\mathcal{L}_{\mathrm{hs}}^{S_2}$ yields

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

2.2.5 Data layout

Some program transformations, like padding, only modify the array declarations and have no impact on the polyhedral representation of statements. We handle separately such transformations for the moment. Still, 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 [25].

2.3 Putting it All Together

Our representation allows 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 [13, 22] or intermediate code generation steps are needed [34, 17].

To illustrate the advantage of "syntax-free composition", consider 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 $\mathbf{i}_{\mathrm{gp}} = [M, N, P, Q]^t$.

Statements iteration domains.

$$\mathcal{D}_{\mathrm{om}}^{S_1} = \left[\begin{array}{c|cccc|c} 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 & -1 \end{array}\right] \begin{array}{l} 0 \le i \\ i \le M-1 \end{array}$$

$$\mathcal{D}_{\mathrm{om}}^{S_2} = \left[\begin{array}{cc|cccc|c} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 & 0 & -1 \end{array}\right] \begin{array}{l} 0 \le i \\ i \le M-1 \\ 0 \le j \\ j \le N-1 \end{array} \quad \mathcal{D}_{\mathrm{om}}^{S_3} = \left[\begin{array}{cc|cccc|c} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 1 & 0 \end{array}\right] \begin{array}{l} 0 \le i \\ i \le P \\ 0 \le j \\ j \le Q \end{array}$$

Statements schedules.

$$\begin{array}{lll} \mathrm{A}^{S_1} = \left[\,1\,\right] & \mathrm{A}^{S_2} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} & \mathrm{A}^{S_3} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \\ \beta^{S_1} = \left[\,0\ \ 0\,\right]^t & \beta^{S_2} = \left[\,0\ \ 1\ \ 0\,\right]^t & \beta^{S_3} = \left[\,1\ \ 1\ \ 0\,\right]^t \\ \text{i.e. } \Theta^{S_1} = \left[\begin{array}{c|c} 0 & 0 \\ 1 & 0 \\ 0 & 0 \end{array}\right] & \text{i.e. } \Theta^{S_2} = \left[\begin{array}{cc|c} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{array}\right] & \text{i.e. } \Theta^{S_3} = \left[\begin{array}{cc|c} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{array}\right] \end{array}$$

Statement access functions.

$$\mathcal{L}_{\mathrm{hs}}^{S_1} = \left\{\left(\mathtt{Z}, \left[\begin{array}{c|cccc|c} 1 & 0 & 0 & 0 & 0 & 0 \end{array}\right]\right)\right\} \qquad \mathcal{R}_{\mathrm{hs}}^{S_1} = \left\{\quad\right\}$$

$$\mathcal{L}_{\mathrm{hs}}^{S_2} = \left\{\left(\mathtt{Z}, \left[\begin{array}{cc|cccc|c} 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right]\right)\right\}$$

$$\mathcal{R}_{\mathrm{hs}}^{S_2} = \left\{\left(\mathtt{Z}, \left[\begin{array}{cc|cccc|c} 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right]\right), \left(\mathtt{A}, \left[\begin{array}{cc|cccc|c} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{array}\right]\right), \left(\mathtt{B}, \left[\begin{array}{cc|cccc|c} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right]\right), \left(\mathtt{X}, \left[\begin{array}{cc|cccc|c} 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{array}\right]\right)\right\}$$

$$\mathcal{L}_{\mathrm{hs}}^{S_3} = \left\{\left(\mathtt{Z}, \left[\begin{array}{cc|cccc|c} 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right]\right)\right\}$$

$$\mathcal{R}_{\mathrm{hs}}^{S_3} = \left\{\left(\mathtt{Z}, \left[\begin{array}{cc|cccc|c} 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right]\right), \left(\mathtt{A}, \left[\begin{array}{cc|cccc|c} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{array}\right]\right), \left(\mathtt{X}, \left[\begin{array}{cc|cccc|c} 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{array}\right]\right)\right\}$$

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.

$$\beta^{S_3} = \left[\,0\ \ 2\ \ 0\,\right]^t$$

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.

$$\beta^{S_3} = \left[\,0\ \ 1\ \ 1\,\right]^t$$

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.

$$\beta^{S_2} = \left[\,1\ \ 0\ \ 0\,\right]^t \quad \beta^{S_3} = \left[\,1\ \ 0\ \ 1\,\right]^t$$

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 $\mathcal{D}_{\mathrm{om}}^{S_2}$ and $\mathcal{D}_{\mathrm{om}}^{S_3}$) 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. $\mathcal{D}_{\mathrm{om}}^{S_2}$ is the same as the domain matrix for S_2' in Section 2.2.2, and the other matrices are:

$$\mathcal{D}_{\mathrm{om}}^{S_3} = \left[\begin{array}{ccc|c|cccc|c} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 0 & 1 & -1 \\ 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 63 \\ 0 & -1 & 0 & 64 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -64 & 0 & 0 & 0 & 0 & 0 \end{array}\right] \begin{array}{l} 0 \le i \\ i \le P-1 \\ 0 \le j \\ j \le Q-1 \\ jj \le j \\ j \le jj+63 \\ jj \le 64jj_2 \\ 64jj_2 \le jj \end{array}$$

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

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}

$$A^{S_2} = A^{S_3} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Summary. Overall, none of the transformation 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.

2.4 Searching for Compositions

This section investigates how the polyhedral representation can further facilitate the search for complex transformation sequences. Applying a program transformation now simply amounts to recomputing the matrices of a few statements, as illustrated in the previous section. 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.

Commutativity properties. Commutativity properties are additional benefits of the separation between four representation aspects and the normalization actions. In general, data and control transformations commute, as well as inter-statement and inter-iteration reordering. For example, loop fusion commutes with loop interchange, statement reordering, loop fission and loop fusion itself. In the example detailed in Section 2.3, 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.

$$\begin{array}{lclcl}
\beta^{S_1} = [\,0\ 0\,] & & \beta^{S_1} = [\,0\ 0\,] & & \beta^{S_1} = [\,0\ 0\,] \\
\beta^{S_2} = [\,0\ 1\ 0\,] & \rightarrow & \beta^{S_2} = [\,0\ 1\ 0\,] & \rightarrow & \beta^{S_2} = [\,0\ 1\ 0\,] \\
\beta^{S_3} = [\,1\ 0\ 0\,] & & \beta^{S_3} = [\,0\ 2\ 0\,] & & \beta^{S_3} = [\,0\ 1\ 1\,] \\
\qquad\downarrow & & & & \qquad\downarrow \\
\beta^{S_1} = [\,0\ 0\,] & & \beta^{S_1} = [\,0\ 0\,] & & \beta^{S_1} = [\,0\ 0\,] \\
\beta^{S_2} = [\,1\ 0\ 0\,] & \rightarrow & \beta^{S_2} = [\,1\ 0\ 0\,] & \rightarrow & \beta^{S_2} = [\,0\ 1\ 0\,] \\
\beta^{S_3} = [\,2\ 0\ 0\,] & & \beta^{S_3} = [\,1\ 1\ 0\,] & & \beta^{S_3} = [\,0\ 1\ 1\,]
\end{array}$$

Confluence properties are also available: in our framework, outer loop unrolling and fusion (unroll-and-jam) is strictly equivalent to strip-mining, interchange and full unrolling.

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 turns enables more efficient search strategies [12].

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.

Furthermore, assuming that a full polyhedral dependence graph has been computed,[3] 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 [5], or even better, using the Farkas lemma as proposed by Feautrier [13], 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 [4, 5]; it is the subject of active ongoing work.

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

2.5 Normalization Rules

We conclude this section on the polyhedral representation with a technical but important contribution. 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, to reach our goal, we have to guarantee good compositionality properties for actions on the same attribute, e.g., loop fusion with interchange (schedule). We achieve this through additional normalization rules.

With the above framework, a given program can have multiple representations, which in turn can limit the application of transformations. 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. Since prior transformations can affect matrix parameters in many ways, we need to *normalize* the representation after each step.

The normalization rules are the following.

Schedule matrix structure. Among many encodings, we choose to separate 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, e.g., fusion with unimodular transformations and shifting. It allows to compose schedule transformations without a costly normalization to the Hermite form.

Sequentiality. Distinct statements, or identical statements in distinct iterations, cannot have the same time stamp: the A component of the schedule matrix must be non-singular. This suppresses scheduling ambiguities at code generation time.

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

Domain parameters. Avoid redundant inequalities and reduce integer overflows in domain matrices $\mathcal{D}_{\mathrm{om}}$.

3 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. We manually optimized 12 SPECfp2000 benchmarks (out of 14) and were able to outperform the peak SPEC performance [33] (obtained in choosing the most appropriate compiler flags) for 9 of them. We detail below the composition sequences for 4 of these benchmarks, the associated syntactic limitations and how we override them.

3.1 Experimental Results

The experimental platform is 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 6 summarizes the speedup with respect to the base SPEC performance.

Methodology. Our optimization 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

Figure 6. 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.

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 10, 8 and 7 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.

3.2 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.

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 2.2.

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.

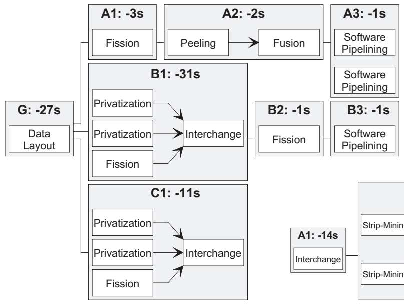
Flexible and complex compositions of transformations. The manual benchmark optimizations exhibit wide variations in the composition of control, access and layout transformations. galgel is an extreme case where KAP does not succeed in optimizing the code, even with the best hand-tuned combination of 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 10, 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.

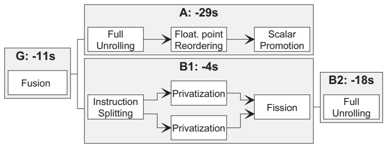
applu stresses another important flexibility issue. Optimizations on two independent code fragments follow an opposite direction: (G) and (A) target locality improvements: they implement loop fusion and scalar promotion; conversely, (B1) and (B2) follow a parallelism-enhancing strategy based on the opposite 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.

4 Implementation

The whole infrastructure is implemented as a free (GPL) plug-in for Open64/ORC [27], allowing generation of native IA64 code. Thanks to third-party tools based on Open64, this framework supports source-to-source optimization, using the robust C unparser of Berkeley UPC [6], and planning a port of the Fortran90 unparser from Open64/SL [9]. The whole infrastructure compiles with





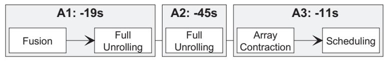


Figure 9. Optimizing wupwise (base 236s)

Figure 8. Optimizing applu (base 214s)

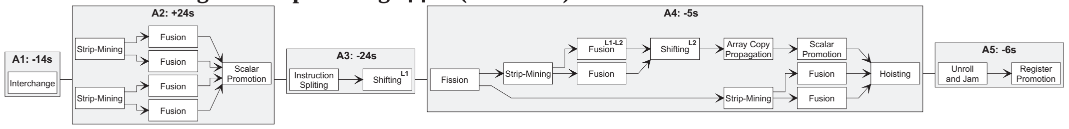


Figure 7. Optimizing apsi (base 378s)

Figure 10. Optimizing galgel (base 171s)

GCC3.4 and is compatible with PathScale EKOPath [8] native code generator for AMD64 and IA32. It contains 3 main tools in addition to ORC: 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. These tools can be downloaded from `http://www.lri.fr/~girbal/site_wrapit`.

Extracting SCoPs. A modern compiler infrastructure like ORC helps the extraction of large SCoPs, thanks to interprocedural analysis and pre-optimization phases: 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: in average, 88% of the statements belong to SCoPs and can be represented in our model.

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.

URUK: Unified Representation Universal Kernel. URUK is the key software component of our framework: 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 called constructors.

Figure 11 shows the definition of the Move constructor, and Figure 12 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 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; }
```

Figure 11. 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); }
```

Figure 12. FISSION primitive

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 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 of the program only*, i.e., without reference to the initial syntactic program. This allows for one-time dependence analysis before applying the transformation sequence, and one-time check at the very end, before code generation. In addition, any dependence violation is reported as a list of polyhedra, capturing the exact set of iterations where causality is lost after the sequence of transformations on a pair of conflicting references. This is very useful for automatic filtering of transformations in an iterative optimization framework, and as an optimization aid for the interactive user of URUK.

URGenT: URUK Generation Tool. After polyhedral transformations, the regeneration 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 with polyhedral transformations. We used the Chunky Loop Generator (CLooG), a recent Quilleré et al. method [32] with some additional improvements to guarantee the absence of duplicated control [3], to generates 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 [2, 32, 3]. Because of the added complexity introduced, we had to design URGenT, a major improvement 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 also improved, often making it better than typically hand-tuned code. The following section shows that URGenT succeeds in producing efficient code for a realistic optimization case-study in a few seconds only.

5 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 choosen 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 13 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 13.

This URUK script operates on the intermediate representation file `swim.N` 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 13 apply to this SCoP.[4] 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')
addContext(C1L1,'M>=500')
addContext(C1L1,'MNMIN>=500')
addContext(C1L1,'MNMIN<=M')
addContext(C1L1,'MNMIN<=N')
addContext(C1L1,'M<=N')
addContext(C1L1,'M>=N')

# Move and shift calc3 backwards
shift(enclose(C3L1),{'1','0','0'})
shift(enclose(C3L10),{'1','0'})
shift(enclose(C3L11),{'1','0'})
shift(C3L12,{'1'})
shift(C3L13,{'1'})
shift(C3L14,{'1'})
shift(C3L15,{'1'})
shift(C3L16,{'1'})
shift(C3L17,{'1'})
motion(enclose(C3L1),BLOOP)
motion(enclose(C3L10),BLOOP)
motion(enclose(C3L11),BLOOP)
motion(C3L12,BLOOP)
motion(C3L13,BLOOP)
motion(C3L14,BLOOP)
motion(C3L15,BLOOP)
motion(C3L16,BLOOP)
motion(C3L17,BLOOP)
```

```
# Peel and shift to enable fusion
peel(enclose(C3L1,2),'3')
peel(enclose(C3L1_2,2),'N-3')
peel(enclose(C3L1_2_1,1),'3')
peel(enclose(C3L1_2_1_2,1),'M-3')
peel(enclose(C1L1,2),'2')
peel(enclose(C1L1_2,2),'N-2')
peel(enclose(C1L1_2_1,1),'2')
peel(enclose(C1L1_2_1_2,1),'M-2')
peel(enclose(C2L1,2),'1')
peel(enclose(C2L1_2,2),'N-1')
peel(enclose(C2L1_2_1,1),'3')
peel(enclose(C2L1_2_1_2,1),'M-3')
shift(enclose(C1L1_2_1_2_1),{'0','1','1'})
shift(enclose(C2L1_2_1_2_1),{'0','2','2'})

# Double fusion of the three nests
motion(enclose(C2L1_2_1_2_1),TARGET_2_1_2_1)
motion(enclose(C1L1_2_1_2_1),C2L1_2_1_2_1)
motion(enclose(C3L1_2_1_2_1),C1L1_2_1_2_1)

# Register blocking and unrolling (factor 2)
time_stripmine(enclose(C3L1_2_1_2_1,2),2,2)
time_stripmine(enclose(C3L1_2_1_2_1,1),4,2)
interchange(enclose(C3L1_2_1_2_1,2))
time_peel(enclose(C3L1_2_1_2_1,3),4,'2')
time_peel(enclose(C3L1_2_1_2_1_2,3),4,'N-2')
time_peel(enclose(C3L1_2_1_2_1_2_1,1),5,'2')
time_peel(enclose(C3L1_2_1_2_1_2_1_2,1),5,'M-2')
fullunroll(enclose(C3L1_2_1_2_1_2_1_2_1,2))
fullunroll(enclose(C3L1_2_1_2_1_2_1_2_1,1))
```

Figure 13. 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 ORC. 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 27s 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 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 ORC 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,[5] our tool achieves **32% speedup on Athlon XP and 38% speedup on Athlon 64**. We are not aware of any other optimization effort — manual or automatic — that reached a similar level of performance on x86 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 needed 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.

6 Related Work

Most loop restructuring compilers introduced syntax-based models and intermediate representations. ParaScope [11] and Polaris [7] are dependence-based source-to-source parallelizers for Fortran and KAP [16] is closely related to these academic tools. They implement syntactic loop transformations.

SUIF [14] is a platform for implementing advanced compiler prototypes. Although some polyhedral works have been built on SUIF [22, 23], they do not address the composition issue and rely on a weaker code generation method. PIPS [15] 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 [25] has been applied to iterative optimization [19]; this compiler aims at the unification of 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 [24]; 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 [17], based on the Omega library [18]. It provides space-time mappings for iteration reordering, and it shares our emphasis

[4] Labels of the form CxLy denote statement y of procedure calc$x$; enclose(v,r) returns the prefix of length $\dim v - r$ of vector v (r is equal to 1 by default). 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.

[5] Athlon XP: `-O3 -m32 -OPT:ro=2:Olimit=0:div_split=on:alias=typed -LNO:fusion=2:prefetch=2 -ipa -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.

on per-statement transformations, but it is intended as a research tool for small kernels only. Our representation — whose foundations were presented in [10] — improves on the polyhedral representation proposed by [17], 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 [3].

7 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 iterative optimization is challenged by the complexity of aggressive loop optimization sequences. We believe it will not bring significant speedups without a specific compilation infrastructure designed for compositionality and 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 code complexity explosion 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 the intrinsic properties of our framework open up new methods for searching for complex transformation sequences.

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 Cédric Bastoul for CLooG, 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 UC Berkeley.

8 References

[1] N. Ahmed, N. Mateev, and K. Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. In *ACM Supercomputing'00*, May 2000.

[2] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loop. In *ACM Symp. on Principles and Practice of Parallel Programming (PPoPP'91)*, pages 39–50, June 1991.

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

[4] C. Bastoul and P. Feautrier. Improving data locality by chunking. In *CC'12 Intl. Conference on Compiler Construction, LNCS 2622*, pages 320–335, Warsaw, Poland, april 2003.

[5] C. Bastoul and P. Feautrier. More legal transformations for locality. In *Euro-Par'10*, number 3149 in LNCS, pages 272–283, Pisa, Aug. 2004.

[6] C. Bell, W.-Y. Chen, D. Bonachea, and K. Yelick. Evaluating support for global address space languages on the cray X1. In *ACM Int. Conf. on Supercomputing (ICS'04)*, St-Malo, France, June 2004.

[7] 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, Dec. 1996.

[8] F. Chow. Maximizing application performance through interprocedural optimization with the pathscale eko compiler suite. `http://www.pathscale.com/whitepapers.html`, Aug. 2004.

[9] 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).

[10] A. Cohen, S. Girbal, and O. Temam. A polyhedral approach to ease the composition of program transformations. In *Euro-Par'04*, number 3149 in LNCS, Pisa, Italy, Aug. 2004. Springer-Verlag.

[11] 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.

[12] K. D. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *J. of Supercomputing*, 2002.

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

[14] M. Hall et al. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, Dec. 1996.

[15] F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: An overview of the pips project. In *ACM Int. Conf. on Supercomputing (ICS'2)*, Cologne, Germany, June 1991.

[16] KAP C/OpenMP for Tru64 UNIX and KAP DEC Fortran for Digital UNIX. `http://www.hp.com/techsevers/software/kap.html`.

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

[18] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *Frontiers'95 Symp. on the frontiers of massively parallel computation*, McLean, 1995.

[19] T. Kisuki, P. Knijnenburg, K. Gallivan, and M. O'Boyle. The effect of cache models on iterative compilation for combined tiling and unrolling. In *Parallel Architectures and Compilation Techniques (PACT'00)*. IEEE Computer Society Press, Oct. 2001.

[20] T. Kisuki, P. Knijnenburg, M. O'Boyle, and H. Wijshoff. Iterative compilation in program optimization. In *Proc. CPC'10 (Compilers for Parallel Computers)*, pages 35–44, 2000.

[21] 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.

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

[23] A. W. Lim, S.-W. Liao, and M. S. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *ACM Symp. on Principles and Practice of Parallel Programming (PPoPP'01)*, pages 102–112, 2001.

[24] S. Long and M. O'Boyle. Adaptive java optimisation using instance-based learning. In *ACM Int. Conf. on Supercomputing (ICS'04)*, pages 237–246, St-Malo, France, June 2004.

[25] M. O'Boyle. MARS: a distributed memory approach to shared memory compilation. In *Proc. Language, Compilers and Runtime Systems for Scalable Computing*, Pittsburgh, May 1998. Springer-Verlag.

[26] M. O'Boyle, P. Knijnenburg, and G. Fursin. Feedback assisted iterative compiplation. In *Proc. LCR*, 2000.

[27] Open research compiler. `http://ipf-orc.sourceforge.net`.

[28] D. Parello, O. Temam, A. Cohen, and J.-M. Verdun. Towards a systematic, pragmatic and architecture-aware program optimization process for complex processors. In *ACM Supercomputing'04*, Pittsburgh, Pennsylvania, Nov. 2004.

[29] 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. In *SuperComputing'02*, Baltimore, Maryland, Nov. 2002.

[30] G.-R. Perrin and A. Darte, editors. *The Data Parallel Programming Model*. Number 1132 in LNCS. Springer-Verlag, 1996.

[31] 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.

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

[33] Standard performance evaluation corp. `http://www.spec.org`.

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

[35] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.

[36] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu. A comparison of empirical and model-driven optimization. In *ACM Symp. on Programming Language Design and Implementation (PLDI'03)*, San Diego, CA, June 2003.