

CHiLL: A Framework for Composing High-Level Loop Transformations

Chun Chen, Jacqueline Chame and Mary Hall
{chunchen, jchame, mhall}@isi.edu

June 13, 2008

Abstract

This paper describes a general and robust loop transformation framework that enables compilers to generate efficient code on complex loop nests. Despite two decades of prior research on loop optimization, performance of compiler-generated code often falls short of manually optimized versions, even for some well-studied BLAS kernels. There are two primary reasons for this. First, today's compilers employ fixed transformation strategies, making it difficult to adapt to different optimization requirements for different application codes. Second, code transformations are treated in isolation, not taking into account the interactions between different transformations. This paper addresses such limitations in a unified framework that supports a broad collection of transformations, (permutation, tiling, unroll-and-jam, data copying, iteration space splitting, fusion, distribution and others), which go beyond existing polyhedral transformation models. This framework is a key element of a compiler we are developing which performs empirical optimization to evaluate a collection of alternative optimized variants of a code segment. A script interface to code generation and empirical search permits transformation parameters to be adjusted independently and tested; alternative scripts are used to represent different code variants. By applying this framework to example codes, we show performance results on automatically-generated code for the Pentium M and MIPS R10000 that are comparable to the best hand-tuned codes, and significantly better (up to a 14x speedup) than the native compilers.

1 Introduction

A number of application domains, including scientific computing, signal processing, graphics and games, require high performance and efficiency on matrix-oriented computations. Such applications often achieve efficiencies of less than 10% on

modern architectures, largely due to poor utilization of the memory hierarchy. In spite of decades of research in memory hierarchy optimization [1, 2, 3, 4, 5], there is still a significant and growing gap between performance of hand-tuned and compiler-generated codes. It is commonly believed that there are three main reasons for this: (1) code transformations for the memory hierarchy are somewhat fragile, and not always applicable or effective in the presence of standard code constructs; (2) code transformations are usually studied in isolation, without considering the subtle interactions between a suite of optimizations; and finally, (3) the complex relationship between code properties and application performance limits the effectiveness of purely static models of transformation effects commonly used in compilers.

Our research goal is to address these three limitations by developing a compiler framework that combines a rich set of robust and composable code transformations with *empirical optimization*. In empirical optimization, rather than trying to statically predict performance through analysis, code variants are executed on the target architecture with representative input data sets, so that performance can be measured and compared, addressing limitations (2) and (3) above. While much of the prior work on empirical optimization relies on hand-coded library generators [6, 7, 8], more recent research seeks to apply compiler models to hand-coded generators to limit or avoid search [9, 10, 11], and to develop hybrid model-guided empirical search in a compiler framework [12].

As compared to hand-coded library generators, the clear advantage of a compiler-based framework for empirical optimization is the opportunity to apply optimizations to more general application code, and therefore dramatically accelerate the process of performance tuning. As we will demonstrate in this paper, by combining a rich set of code transformations with empirical search, it is possible for compiler-generated code to achieve performance levels comparable to manually-tuned code for matrix-oriented computation. To address limitation (1) above, we have developed a critical piece of this system, a loop transformation framework and associated interfaces that are specifically designed for evaluating a series of alternative implementations of a computation. The design and construction of this framework, the underlying transformation algorithms and abstractions, and the interface to the rest of the system, are the focus of this paper.

As our framework builds on two decades of prior work in loop transformation, the contribution of the work includes a mix of significant new directions as well as subtle improvements, as follows.

- *Robustness and Applicability*: supports a large set of code transformations in the presence of imperfect loop nests and non-rectangular loop bounds;
- *Systematic*: provides a uniform representation of iteration spaces, statements

and transformations that facilitates composing multiple code transformations, without intermediate code generation;

- *High-quality generated code*: focuses on minimizing overhead in the presence of complex code constructs;
- *High-level transformation script*: used to describe transformation sequence with minimum required parameters, facilitating search of alternative variants and parameter values.

While there exist several transformation frameworks with similar goals [13, 14, 15, 16], ours is the first that is capable of taking a high-level transformation specification and automatically producing high-quality code for LU factorization, a kernel that exhibits the complexities of non-rectangular loop bounds and imperfect loop nests that defeat many transformation algorithms.

By applying this framework to well-known computational kernels that require complex transformations to achieve high performance, we demonstrate that the resulting code quality is quite high. The performance on Matrix Multiply and Triangular Solver on the Pentium M and MIPS R10000 matches the self-tuning library ATLAS and manually-tuned vendor’s libraries, and achieves 1.50x and 1.67x average speedups over the native compilers. For the more complex LU Factorization, we obtain performance that is within 18% of vendor’s libraries and ATLAS, and up to a 16x speedup over the native compilers. These results show that, with a systematic framework, it has now become feasible for compiler-generated code to achieve performance comparable to manually-tuned, even for more complex code constructs than have been previously demonstrated.

The remainder of this paper is organized as follows. Section 2 presents an overview of the approach. Section 3 describes the internal representation used by the framework, and Section 5 presents the high-level transformation script interface. Section 4 describes the implementation of transformations. Section 6 presents an experimental evaluation of the approach, followed by a conclusion.

2 Overview

Our research goal is to construct a compilation system that performs model-guided empirical optimization for the memory hierarchy, and is capable of generating code with performance comparable to manually-tuned code. In this section, we describe the system we are developing to provide context for the completed transformation framework that is the focus of this paper, and subsequently discuss how we must go beyond the prior work in this area to accomplish our goals.

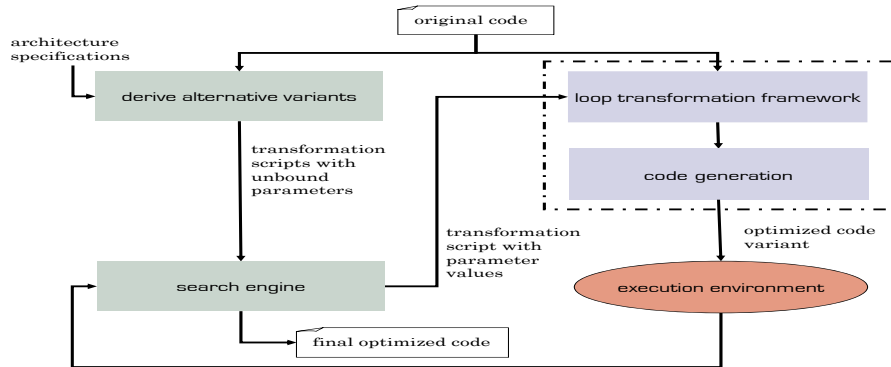


Figure 1: High Level Overview

2.1 Compilation System

A compilation system that performs model-guided empirical optimization generates multiple candidate variants of the original code, and executes these variants, with a possibly wide range of optimization parameters, on the target machine. A code variant is generated by applying a sequence of transformations to the original code, and each code variant may have a large set of possible adjustable optimization parameter values. An empirical-optimization system can be thought of as having the three major components shown as rectangles in Figure 1:

1. *Deriving alternative code variants:*

A decision algorithm such as [5, 2, 12] derives a sequence of code transformations that corresponds to a transformed version of the original code. This sequence of transformations is represented as a *transformation script*, which might contain unbound transformation parameters. A script expresses to the code generator, at a high level, the sequence of transformations to be performed. Scripts provide a precise description used to express alternative transformation strategies, and are presented in Section 5.

2. *Search engine:* The search engine, such as in [12] takes as input a set of scripts, each containing some unbound optimization parameters. In conjunction with representative data sets, the search engine drives the execution of alternative final implementations. For each script, the engine traverses the space of parameter values, producing scripts with bound parameters that are the input of the loop transformation and code generation framework. The final code variants are then executed to determine which script and set of parameter values leads to the best-performing implementation.

3. *Loop transformation and code generation framework*: The loop transformation and code generation framework, described in the remainder of this paper, takes as input the original code and a transformation script with bound parameters, and generates an optimized code version. The loop transformation framework uses a uniform representation for transformations and iteration spaces which allows it to abstract original code constructs and manipulate polyhedra-based iteration spaces instead, facilitating the composition of transformations. Given a transformation script with bound parameters, after applying transformation algorithms, the final transformed code is then generated from the transformed iteration spaces and statements.

2.2 Related Work

Our research goal is to construct the complete system of the previous subsection that is sufficiently robust to be used by our various application partners who already perform these “compiler” optimizations by hand (*e.g.*, in high-end computing, library development and signal processing). Initially, the tool is being used in a semi-automated fashion with manually-written transformation scripts, but we have developed algorithm and are implementing the compiler to generate these scripts automatically using an approach that extends the model-guided empirical optimization strategy from [12] for more general code constructs and architectural features (*e.g.*, SIMD compute engines, software-managed storage and multi-core devices).

In developing this system, we considered the requirements of generating high-quality code for LU factorization as the test of its effectiveness, as it contains imperfect loop nests and non-rectangular loop bounds, which defeat most compiler transformation systems. So far only the work Ahmed et al. [17] and Yi et al. [18] address such challenge in a systematic way. Ahmed et al. try to find an embedding function [17] in the space of Cartesian product of iteration spaces of all statements in a loop nest, which essentially converts the original imperfect loop nest to a perfect loop nest with possible loop interchange effect. Tiling can be applied thereafter. Yi et al. apply the dependence hoisting technique [18], which enables permuting an imperfect loop nest, to tile such loop nest. However, there are no discussions in both works on how to integrate many other transformations into such frameworks.

Overall, there has been an extensive amount of research in memory hierarchy optimizations, with a small subset that have most influenced our work cited here [2, 3, 1, 19, 4, 5]. A large body of work supporting imperfect loop nests has focused on loop fission and fusion, and does not incorporate tiling, permutation and unroll-and-jam of imperfect loop nests. Jiménez et al. [20] developed an approach for unroll-and-jam in the presence of non-rectangular loops based on register tiling

and a new index set splitting algorithm. Most of this prior research has focused on a specific transformation, rather than composing a collection of transformations.

A different way than composing transformations is to decompose the original loop nest into simpler kernels [21], assuming they are much simpler to optimize. The idea to represent multiple transformation versions is also captured by X-language [22], which uses pragmas in the source code to describe sequences of transformations.

As we are developing a framework that supports composition of transformations, the research most closely related to ours is Petit [23], WRaP-IT [15], and the work of Cohen et al. [16]. These frameworks all use a polyhedral representation that is capable of handling non-rectangular loops and imperfect loop nests and support a wide range of transformations. The main difference of our framework with them is that we have developed new algorithms to support one-step transformations for complex loop nests like permuting imperfect loop nest or unroll-and-jam triangular loop nest. Most other systems typically support low-level loop transformations (*e.g.*, shift, extend, contract) for arbitrary loop nest and high-level loop transformations (*e.g.*, unroll-and-jam, permute) for perfect loop nests only. Such limitation requires compiler or higher-level tool to generate non-trivial script composing of low-level loop transformations when complex transformations are required on complex loop nests. As a result, in Cohen et al.’s iterative optimization, they are able to search all 1-dimensional iteration schedules (*i.e.*, a single loop level), but the resulting performance gains are not as significant as ours (*e.g.*, 8-9% gains for LU over Intel’s compiler) [16].

Our framework CHiLL is built upon the Omega Library [24] for polyhedra manipulation and scanning. The functionality of the original Omega Library has been improved, including better simplification of gist function in the presence of multiple integer modular constraints and merging contiguous if-statements in the generated code. Furthermore, through features of our framework described in Section 3, we are able to generate high-quality code for imperfect loop nests as demonstrated in Section 6.

In spite of all this prior work in loop transformations, this paper represents the first integration of all these techniques in a common framework that supports transformation composition, with several new transformation algorithms and the first to our knowledge capable of achieving high performance on automatically-generated code for LU factorization.

```

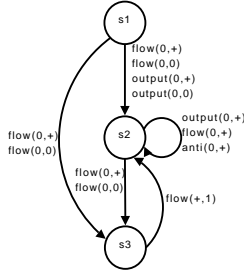
DO I=2, N
s1  SUM(I)=0
    DO J=1, I-1
s2   SUM(I)=SUM(I)+A(J,I)*B(J)
s3   B(I)=B(I)-SUM(I)

```

(a) Original code

$$\begin{aligned}
IS_1 &: \{[i, j] \mid 2 \leq i \leq N \wedge j = 1\} \\
IS_2 &: \{[i, j] \mid 1 \leq j < i \leq N\} \\
IS_3 &: \{[i, j] \mid 2 \leq i \leq N \wedge j = i - 1\}
\end{aligned}$$

(b) Aligned iteration spaces



(c) Dependence graph

$$\begin{aligned}
t_{s_1} &: \{[* , i , * , j , *] \rightarrow [0 , i , 0 , j , 0]\} \\
t_{s_2} &: \{[* , i , * , j , *] \rightarrow [0 , i , 1 , j , 0]\} \\
t_{s_3} &: \{[* , i , * , j , *] \rightarrow [0 , i , 2 , j , 0]\}
\end{aligned}$$

(d) Transformation relations to generate the original loop nest in (a)

Figure 2: Representing loop nest and transformations.

3 Representation

The loop transformation algorithms rely on a polyhedral representation of iteration space for each statement, whose granularity can be adjusted depending on the intended optimization goal. To allow a uniform framework for all loop transformations and support for imperfect loop nests, we introduce two key concepts, *alignment of iteration spaces* and *auxiliary loops* to extend the original Omega representation [13]. We will explain our approach in detail in the rest of this section.

3.1 Aligning Iteration Spaces

For an imperfect loop nest such as in Figure 2(a), our framework would extract the iteration space for each statement as in Figure 2(b). Note that every statement in the loop nest has the same number of dimensions for its iteration space. That is, although s1 and s3 are only surrounded by one loop I, their iteration spaces are still 2-dimensional; more precisely, each represents a line *aligned* in a 2-dimensional iteration space. Once the iteration spaces of all statements are aligned in the same iteration space, we can then design a systematic way to transform imperfect loop nests, and the legality of the transformations can be determined in a way similar to perfect loop nests, i.e., from data dependences prior to transformation.

The alignment process is a generalization of code sinking and loop fusion. Figure 3 shows the algorithm for extracting and aligning the iteration spaces from the respective loops around statements. The derivation of the iteration space is performed in the following way. The algorithm first finds the statement within the

deepest nesting level of the loop nest and sets its iteration space as the reference iteration space for alignment. Then for each statement, the algorithm maps each surrounding loop for the statement to a dimension in the iteration space. For missing dimensions, it sets their iteration space values to upper or lower bounds of iteration spaces of those statements already extracted and aligned. The validity of the extracted iteration space is verified according to the following theorem:

Theorem 1 *Suppose statements $s1$ and $s2$ have common loops l_1, l_2, \dots, l_m and $s1$ precedes $s2$ in loop l_m lexicographically, and k is the last dimension among all l_i 's ($i = 1, 2, \dots, m$) corresponding dimensions in the aligned iteration space. Then an alignment is valid if and only if there is no dependence vector $(\underbrace{0, \dots, 0}_k, d_{k+1}, d_{k+2}, \dots, d_n)$ from $s2$ to $s1$.*

It guarantees that data dependences reflected in the original code are not violated. Furthermore, we want dependence vectors to be as simple as possible. For example, a data dependence $(0, 1, 1)$ is preferred than $(0, +, *)$.¹ The simpler dependence vector not only makes more transformations available to potentially improve performance, but also tends to generate more efficient transformed code with less overhead.

In the example of Figure 2(a), the algorithm first selects statement $s2$. It has loop \mathbb{I} and \mathbb{J} , which determine a 2-dimensional iteration space for the entire loop nest. Statement $s2$'s iteration space is the range extracted from those two loops' lower and upper bounds in that order for the first and second dimension. Next the algorithm selects statement $s1$. It has only one surrounding loop \mathbb{I} , which is already mapped when extracting iteration space for $s2$. So it sets the first dimension for $s1$'s iteration space from this loop \mathbb{I} 's lower and upper bound. For the missing second dimension, it selects $s2$ for alignment. After dependence calculation, it sets the second dimension for $s1$'s iteration space to a point at the lower bound of the second dimension for $s2$'s iteration space, which is loop \mathbb{J} 's lower bound. Now all dimensions for $s1$'s iteration space are set. The same goes for $s3$'s iteration space, with the second dimension set to loop \mathbb{J} 's upper bound instead. The final result is in Figure 2(b) we showed earlier.

Once iteration spaces for all statements are extracted and aligned, the dependence graph among statements is also built at the same time. The polyhedral description of iteration spaces of all statements combined with the data dependences among them give us all the necessary information about the original loop nest to do loop transformations. Thus for the loop nest in Figure 2(a), the loop transforma-

¹ In dependence vectors, we use notations $+$ as range $[1, +\infty)$, $-$ as $(-\infty, -1]$ and $*$ as $(-\infty, +\infty)$.

```

ExtractIterationSpace (loopNest)
Input: loopNest: the original code
Output: Set of iteration spaces for statements
begin
  S := set of statements in loopNest;
  Map each loop to a dimension in the iteration space, initially set as NULL;
  s := FindDeepestNesting (S);
  S := S - {s};
  Let  $l_1, l_2, \dots, l_n$  be s's surrounding loops;
  Map  $l_i, (i = 1, \dots, n)$  to dimension  $i$  in the iteration space;
  Add condition  $LB_i \leq d_i \leq UB_i$  to s's iteration space for loop  $l_i$  iterating from  $LB_i$  to  $UB_i$ , where  $d_i$  represents  $i$ -dimension variable;
  P := {s} /*Set of statements with iteration space extracted*/;
  while S  $\neq \emptyset$  do
    r := FindDeepestNesting (S);
    S := S - {r};
    Let  $l_1, l_2, \dots, l_m, (m \leq n)$  be r's surrounding loops;
    Add the same condition to r's iteration space for mapped loops among  $l_i, 1 \leq i \leq m$ ;
    bestDependenceCost :=  $\infty$ ;
    bestIterationSpace := NULL;
    for each mapping function f from unmapped loops in  $l_i (1 \leq i \leq m)$  to unmapped dimensions in the iteration space do
      Add condition  $LB_i \leq d_{f(i)} \leq UB_i$  to r's iteration space for loop  $l_i$  iterating from  $LB_i$  to  $UB_i$ , where  $d_{f(i)}$  represents  $f(i)$ -dimension variable;
      if  $n > m$  then
        Q := P;
        while Q is not empty do
          t := FindLexicallyClosest(r, Q);
          Q = Q - {t};
          for each remaining unmapped  $(n - m)$  dimension j do
            Set  $d_j$  to the lower bound or upper bound of the same dimension in t's iteration space;
          if current iteration space for r is valid using Theorem 1 then
            cost := CalculateDependenceCost(dependence graph);
            if cost < bestDependenceCost then
              bestDependenceCost := cost;
              bestIterationSpace := current iteration space;
            break;
        else /*  $n = m$  */
          cost := CalculateDependenceCost(dependence graph);
          if cost < bestDependenceCost then
            bestDependenceCost := cost;
            bestIterationSpace := current iteration space;
      if bestIterationSpace is NULL then
        return  $\emptyset$ ;
      else
        Set r's iteration space as bestIterationSpace;
        Set those unmapped loops in  $l_i (1 \leq i \leq m)$  to corresponding dimensions as used in bestIterationSpace;
        P := P  $\cup$  {r};
    end
  end

```

Figure 3: Extracting iteration spaces

tion algorithms described in Section 4 only read Figure 2(b) and (c), without ever checking the original code.

3.2 Auxiliary Loops

We rely on Omega code generation to generate transformed code from iteration spaces. However, the iteration spaces themselves as shown in the previous section do not have enough information to determine the relationship or required execution order among statements or how loops would be organized at a specific loop level. To keep the underlying polyhedral scanning strategy robust and simple, we introduce auxiliary dimensions to the transformed iteration spaces as an abstraction layer above Omega code generation. This is a different strategy from [25], which embeds additional processing logic into code generation phase. We add one *auxiliary loop* to each loop level, with an additional auxiliary loop as the last dimension. These auxiliary loops establish the lexicographical order of loops at each loop level as well as the lexicographical order of statements in the innermost loop, by setting different constant integer values for such loops. So for an n -deep loop nest, we have $(2n + 1)$ -dimension iteration spaces as $[c_1, l_1, c_2, l_2, \dots, c_n, l_n, c_{n+1}]$, where c_i 's are auxiliary loops. Each loop transformation from an n -deep loop nest to a new m -deep loop nest is represented as a set of relations

$$t : \{[c_1, l_1, \dots, c_n, l_n, c_{n+1}] \rightarrow [c'_1, l'_1, \dots, c'_m, l'_m, c'_{m+1}] \dots \}.$$

Figure 2(d) shows the transformation relations to generate the original loop nest, with the initial auxiliary loop values unknown yet. The loop transformation algorithms described in Section 4 will set the appropriate auxiliary loop values to achieve the intended transformations efficiently. Finally, since only constant integer values are allowed in auxiliary loops, there are no actual loops generated for such loop levels.

4 Loop Transformation Algorithms

The transformation algorithms derive transformation relations for each statement that represent the behavior of the transformation. The unimodular and nonunimodular transformations [26, 27, 28, 29, 30], defined by non-singular transformation matrices, on perfect loop nests are largely supported by the underlying Omega Library directly.² However, to support more complex high-level transformations on imperfect loop nests, new algorithms must be developed and they are the subject of this section. Legality of transformations must first be verified by the compiler,

²We improved the Omega Library for this purpose.

but through the use of alignment, such tests follow the standard test [31] for perfect loop nests and are not discussed here. The resulting dependence graph is used not only by the compiler to determine the available legal transformations for optimization, but also by the transformation algorithms to generate correct and efficient code. Subsequently, when applying a transformation the algorithm may modify the dependence graph to reflect the data dependences in the new iteration space. This avoids the need for the compiler to repeatedly build the dependence graph from scratch.

4.1 Permutation

A permutation π of $1, 2, \dots, n$ is represented by creating a transformation relation t ,

$$t = \{[c_1, l_1, c_2, l_2, \dots, c_n, l_n, c_{n+1}] \rightarrow [c'_1, l_{\pi_1}, c'_2, l_{\pi_2}, \dots, c'_n, l_{\pi_n}, c'_{n+1}]\}$$

for each statement, where $c'_1, c'_2, \dots, c'_{n+1}$ are the auxiliary loops after permutation. The constants for auxiliary loops are computed by the algorithm in Figure 4, with S as the set of all statements in the nest and loop level d set to 1.

Figure 5 illustrates how the algorithm assigns auxiliary loop values after permuting loops i and j in Figure 2(a). The transformation relations are shown in Figure 5(b) and (c). The dependence graph after permuting loops i and j is shown in Figure 5(a).

The directed graph corresponding to the first dimension has two nodes, one representing s_1 which is surrounded by a single iteration loop at this level, which means no actual loop required to iterate this single point, and the other representing $\{s_2, s_3\}$ which are surrounded by l_1 with multiple iterations. A single edge represents the dependences from s_1 to s_2 and from s_1 to s_3 respectively. Assigning values to nodes in Figure 5(b) in topological order, $c_1^1 = 0$ in t_{s_1} , $c_1^2 = 1$ in t_{s_2} and $c_1^3 = 1$ in t_{s_3} . The directed graph for the next loop level is shown in Figure 5(c). The edge between s_2 and s_3 corresponds to dependence (0,0), the only dependence between s_2 and s_3 that is not carried by l_1 . Again, constant integer values for auxiliary loops are assigned in topological order, resulting in the transformations in Figure 5(c).

4.2 Unroll-and-Jam

Generating efficient code when applying unroll-and-jam to non-rectangular loop bounds is a non-trivial problem in itself. Furthermore, our framework needs to support unroll-and-jam imperfect loop nests combined with non-rectangular loop bounds as a single high-level loop transformation.

```

SetAuxLoops ( $S, d$ )
Input:  $S$ : set of active statements;  $d$ : current loop level
Output: Set of transformation relations for statements in  $S$ 
begin
  Build a directed graph  $G$  for loop level  $d$  where each node represents a set of statements
  from  $S$  and each edge represents a set of dependences by the following rules:
  

- A statement whose iteration space includes only one iteration of  $l_d$  corresponds to an
    individual node;
- All statements whose iteration spaces include multiple iterations of  $l_d$  are represented by a
    single node;
- Add an edge from nodes  $n_1$  to  $n_2$  if there exists at least one pair of statements  $s_1, s_2$  such
    that  $s_1$  is represented in  $n_1$ 
    and  $s_2$  in  $n_2$ , and there is a data dependence from  $s_1$  to  $s_2$  that is not carried by an outer
    loop  $l_i, 1 \leq i < d$ ;


  /* In the following, we call a statement in a subgraph if this statement is represented by
  a node in the subgraph. */  $order := 0$ ;
  for each strongly connected component (SCC) of graph  $G$  in topological order do
    if SCC has only one statement  $s$  then
      Set  $c_d = order$  and  $c_{d+1} = c_{d+2} = \dots = c_n = 0$  in  $s$ 's transformation
      relation;
    else
      for each statement  $s \in SCC$  do
        Set  $c_d = order$  in  $s$ 's transformation relation;
        SetAuxLoops ( $\{s \mid s \in SCC\}, d + 1$ );
       $order := order + 1$ ;
    end
  end

```

Figure 4: Algorithm for setting auxiliary loop values

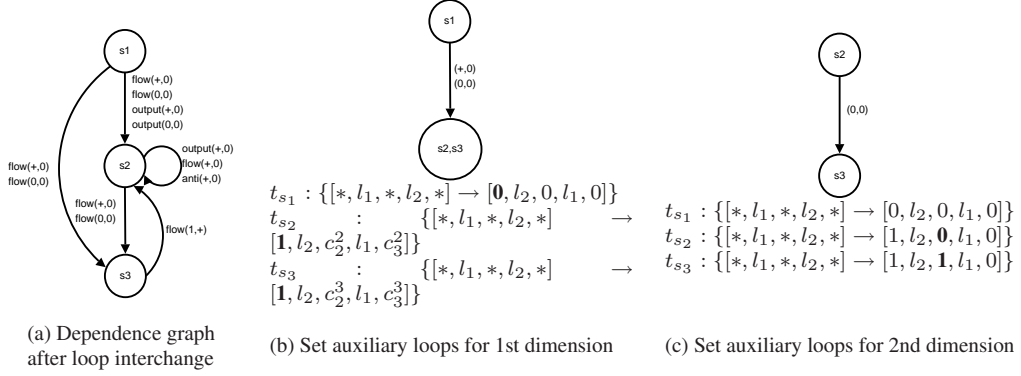


Figure 5: Permute imperfect loop nest in Figure 2 by setting auxiliary loop values

Figure 6 gives an overview of our unroll-and-jam algorithm. The algorithm is based on loop fusion concept. First for each unrolling, a new statement is created with new iteration space which might be different from the original one due to non-rectangular iteration space. By relying on underlying code generation strategy to separate the overlapping and non-overlapping iteration spaces among these statements, we guarantee that unrolled loop body have appropriate loop bounds and correct cleanup code will be generated. One additional issue is that when unroll-and-jam an outer loop with imperfect loop nest inside, data dependence might affect maximal loop bounds allowed for the unrolled loop body. For example in Figure 7, (a) and (b) both have the same iteration space but data dependences among statement s_1 and s_2 are different. Suppose we want to unroll-and-jam outer loop \mathbb{I} by 2, in (a) the unrolled loop body can have \mathbb{J} 's loop bound from 1 to N . In (b), due to data dependence $(1, 0)$ from s_2 to s_1 , the \mathbb{J} 's maximum loop bound allowed is from 1 to $N - 1$. In both cases, we want the correct cleanup code to be generated. To solve this problem, in our algorithm, whenever a new statement is created, its data dependence regarding to other statements is updated accordingly. By using the algorithm in Figure 4, the statements related to unrolling will be reordered to achieve the optimal loop bounds for unrolled loop body.

Figure 8 shows an example of applying our approach to unroll the outer loop of an imperfect loop nest with parallelogramal iteration space by a factor of 2, where (c) is the code generated automatically by our compiler. Since there is no data dependence preventing two unrolled s_1 statements executing before s_2 , the innermost loop for unrolled loop body iterates from $\mathbb{T}2+1$ to $\mathbb{T}2+N$. Correct cleanup code for non-rectangular iteration space is generated inside loop \mathbb{I} and there is also cleanup code after loop \mathbb{I} for when loop \mathbb{I} is not multiple of 2.

```

UnrollAndJam( $l, u$ )
Input:  $l$ : loop to be unrolled;  $u$ : unroll amount
begin
   $N_i := \lceil (UB(l) - LB(l) + 1) / Step(l) \rceil$ ;
  Emit ( $s_o : over_O = N_i \bmod u$ ); /* statement for calculating overflow variable */;
  Apply iteration space splitting (Section 4.4) to  $l$  by condition
   $l \leq UB(l) - over_O * step(l)$ , generating new loops  $l_U$  and  $l_O$  such that:
   $LB(l_U) = LB(l)$ ;  $UB(l_U) = UB(l) - over_O * Step(l)$ ;  $LB(l_O) \geq UB(l_U) + 1$ ;  $UB(l_O) =$ 
   $UB(l)$ ;
  for each statement  $s$  enclosed within  $l_U$  do
    for  $k = 1$  to  $u-1$  do
      Emit ( $s_{u(k)}$  : copy of statement  $s$  with loop variable  $l_U$  replaced by
       $l_U + k * step(l)$ );
      Let  $IS = \{l_1, \dots, l_U, l_{U+1}, \dots, l_n\}$  be  $s$ 's iteration space;
      Partition space  $IS$  on  $l_U$  dimension to two subspaces  $IS_H$  and  $IS_L$ , where
       $IS_H$  includes dimension from  $l_1$  to  $l_U$  and  $IS_L$  from  $l_{U+1}$  to  $l_n$  such that
       $IS = IS_H \cap IS_L$ ;
       $IS_{L'}$  =  $IS_L$  with variable  $l_U$  inside the relation replaced by
       $l_U + k * step(l)$ ;
      Set  $s_{u(k)}$ 's iteration space as  $IS_H \cap IS_{L'}$ ;
      Add modulo constraint  $l_U = LB(l) + u * \alpha, \forall \alpha$  to  $s_{u(k)}$ 's iteration space;
    Add modulo constraint  $l_U = LB(l) + u * \alpha, \forall \alpha$  to  $s$ 's iteration space;
  Update dependence graph for statements enclosed within  $l_U$  /* including new
  statements */;
  SetAuxLoops ( $\{statements\ enclosed\ within\ l_U\}, level(l_{U+1})$ ) /* Figure 4 */;
end

```

Figure 6: Algorithm for unroll-and-jam



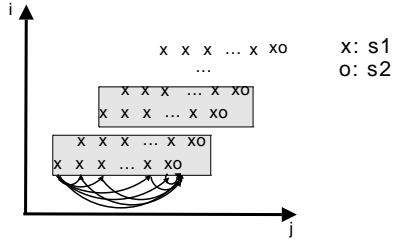
Figure 7: Different unroll-and-jam based on data dependence

```

DO I=0,N
  DO J=I,I+N
s1    F3(I)=F3(I)+F1(I)*W(J-I)
s2    F3(I)=F3(I)*DT

```

(a) Original code



(b) Parallelogramal iteration space

```

OVER1=MOD(N+1,2)
DO T2=0,N-OVER1,2
  F3(T2)=F3(T2)+F1(T2)*W(0)
  DO T4=T2+1,T2+N
    F3(T2)=F3(T2)+F1(T4)*W(T4-T2)
    F3(T2+1)=F3(T2+1)+F1(T4)*W(T4-(T2+1))
  F3(T2+1)=F3(T2+1)+F1(N+T2+1)*W(N)
  F3(T2)=F3(T2)*DT
  F3(T2+1)=F3(T2+1)*DT
IF(OVER1>=1)
  DO T4=N,2*N
    F3(N)=F3(N)+F1(T4)*W(T4-N)
IF(N>=0 .AND. OVER1>=1)
  F3(N)=F3(N)*DT

```

(c) Outer loop I unrolled by 2

Figure 8: Unroll-and-jam imperfect loop nest

4.3 Data Copy

Data copy is an important transformation for locality optimization. It copies the touched data on a given data structure (more precisely, a given uniformly generated set) inside a loop nest into a temporary array. Such copy will change the cache behavior on those data accesses. Data copy can also change the layout of the original array during the copy by changing the leading dimension. Such transformation can enable vectorizing the innermost loop which might not be available before. The new copy statement will be inserted lexicographically before the loop nest in interest. In the discussion below, we assume that array indices are affine. By using polyhedral model, we can precisely calculate the iteration space the array accesses will touch inside the loop nest. The detailed algorithm is shown in Figure 9. For simplicity, we omit the details of uniformly generated array references and index switching in the algorithm.

The algorithm first builds a subspace where each array subscript is represented as a dimension in the space. The range of the each dimension can be calculated from the iteration space the statement is located inside. The copy statement's iteration space is directly deducted from this subspace. For example, for a two-dimensional array, no matter how many loop levels it is enclosed inside, the copy statement outside the entire loop nest only requires two levels of loop nest. However, to generate efficient code as much as possible, in case there is an array subscript only having one value touched inside the loop nest, the algorithm will not create the temporary array's dimension for this array subscript. Subsequently, there is no loop level generated to copy array elements for this dimension.

DataCopy ($d, A[f_1(\vec{l}), f_2(\vec{l}), \dots, f_m(\vec{l})]$)

Input: d : copy statement is outside this loop level; A : array reference with affine subscriptions;

begin

Assume array access A is located in statement s with iteration space $I_s = \{[l_1, l_2, \dots, l_n] \mid \dots\}$;

Set mapping $g := \{[l_1, l_2, \dots, l_{d-1}, l_d, \dots, l_n] \rightarrow [l_1, l_2, \dots, l_{d-1}, i_1, i_2, \dots, i_m] \mid i_1 = f_1(\vec{l}), i_2 = f_2(\vec{l}), \dots, i_m = f_m(\vec{l})\}$;

Set iteration space $I := g(I_s)$;

Let m' be the number of non-single loops among i_1, i_2, \dots, i_m in iteration space I ;

Create declaration for temporary array P with dimensionality m' , where array size for each index is calculated from I /* there is no need to create index with size 1 */;

Create copy statement

$$P[\dots, i_k - LB(i_k) + 1, \dots] = A[\dots, i_k, \dots],$$

where i_k is non-single loop in I and LB is the lower bound. For single loop i_k in I , replace corresponding A 's subscript with $LB(i_k)$;

Set the iteration space for the above copy statement as I with single loops projected away, and with auxiliary loops added to set copy statement lexicographically before statement s 's enclosing loop l_d ;

Replace each reference $A[f_1(\vec{l}), f_2(\vec{l}), \dots, f_m(\vec{l})]$ in statements surrounded by l_d with $P[f_1(\vec{l}) - LB(i_1) + 1, f_2(\vec{l}) - LB(i_2) + 1, \dots, f_m(\vec{l}) - LB(i_m) + 1]$, where there is no index in P for single loop i_k in I ;

end

Figure 9: Algorithm for Data Copy

4.4 Other transformations

Such polyhedral transformation framework can support a wide variety of transformations, some of which are described in [13]. Here we only briefly describe a few transformations that we extend their functionality and generality in light of imperfect loop nests.

Loop Tiling: After aligning iteration spaces and inserting auxiliary loops, applying tiling to imperfect loop nests is straightforward. Suppose loop l_i is to be tiled by size T , and the new tile controlling loop l_i^C is to be placed at level j , $j < i$. We create the following relations:

$$\begin{aligned}
r_1 &= \{[c_1, l_1, \dots, c_j, l_j, \dots, c_i, l_i, \dots, c_n, l_n, c_{n+1}] \rightarrow [\dots, c_{j-1}, l_{j-1}, c_i^C, l_i^C, c'_j, l_j, \dots, c'_i, l_i, \dots] \mid \\
&\quad \exists \alpha (l_i^C = \text{LB} + \alpha T \wedge \alpha \geq 0) \wedge l_i^C \leq l_i < l_i^C + T \wedge c_i^C = 0\}, \\
r_2 &= \{[c_1, l_1, \dots, c_j, l_j, \dots, c_i, l_i, \dots, c_n, l_n, c_{n+1}] \rightarrow [\dots, c_{j-1}, l_{j-1}, c_i^C, l_i^C, c'_j, l_j, \dots, c'_i, l_i, \dots] \mid \\
&\quad l_i^C = \text{LB} \wedge c_i^C = -1\}, \\
r_3 &= \{[c_1, l_1, \dots, c_j, l_j, \dots, c_i, l_i, \dots, c_n, l_n, c_{n+1}] \rightarrow [\dots, c_{j-1}, l_{j-1}, c_i^C, l_i^C, c'_j, l_j, \dots, c'_i, l_i, \dots] \mid \\
&\quad l_i^C = \text{UB} \wedge c_i^C = 1\}
\end{aligned}$$

where LB and UB are the lower and upper bounds of l_i in the union of iteration spaces of all statements within the tile controlling loop l_i^C .

For each statement s and transformation relation t_s : if s is nested within the tile controlling loop l_i^C , set its new transformation relation as $r_1 \circ t_s$; if s is not within l_i^C but executed before the tiled code, set its transformation relation as $r_2 \circ t_s$; otherwise set its transformation relation as $r_3 \circ t_s$.

Iteration Space Splitting: Iteration space splitting decomposes a loop so that each resulting loop executes a disjoint part of the original iteration space. Assume we want to split loop l_d according to inequality condition r :

$$r = \{[l_1, l_2, \dots, l_n] \mid a_0 + a_1 l_1 + a_2 l_2 + \dots + a_k l_k \geq 0\}$$

where a_k is the last non-zero coefficient. Each statement s , with iteration space I_s , is replaced by two copies s_1 and s_2 , with iteration spaces $I \cap r$ and $I \cap \bar{r}$, respectively. Assuming t_s is the transformation relation for s , we now compute the new relations t_{s_1} and t_{s_2} .

If $k < d$, we rely on Omega code generation to separate the disjoint iteration spaces at loop level k so that the auxiliary loop values c_d^1 in t_{s_1} and c_d^2 in t_{s_2} remain the same. If $k = d$, we increment c_d^2 if $a_k < 0$; otherwise we increment c_d^1 . Finally if $k > d$, we rely on the dependence graph to decide which partition executes first. Let D be the function that maps from the loop level in transformation relations to the corresponding dimension in the dependence graph. If there is a dependence vector $(0, \dots, 0, d_{D(d)}, d_{D(d)+1}, \dots)$ such that $\sum_{i=D(d)}^{D(k)} a_{D^{-1}(i)} d_i < 0$ (or > 0),

```

DO I=1, N
  DO J=1, N
    A (J, I)=0.5 * (A (J+1, I-1) +A (J-1, I) )

```

(a) Original code

```

DO T2=1, N-1
  DO T4=1, N-T2
    A (T4, T2)=0.5 * (A (T4+1, T2-1) +A (T4-1, T2) )
  DO T2=1, N
    DO T4=N-T2+1, N
      A (T4, T2)=0.5 * (A (T4+1, T2-1) +A (T4-1, T2) )

```

(b) Split the second loop by $i < N + 1 - j$

```

DO T2=1, N
  DO T4=1, N-T2
    A (T4, T2)=0.5 * (A (T4+1, T2-1) +A (T4-1, T2) )
  DO T4=N-T2+1, N
    A (T4, T2)=0.5 * (A (T4+1, T2-1) +A (T4-1, T2) )

```

(c) Split the first loop by $i < N + 1 - j$

Figure 10: Iteration space splitting at two different loop levels

we increase c_d^2 (or c_d^1) by one. Otherwise the partitions can be executed in arbitrary order, and the auxiliary loop values are set as follows: if $a_k < 0$ increment c_d^2 ; otherwise, increment c_d^1 . Figure 10 shows an example of splitting a double loop nest at different levels according to the same condition.

Loop Fusion: Once the compiler has proved it is safe to perform fusion, expressing fusion using the transformation relations is straightforward. Assume statements s_1 and s_2 with transformation relations t_{s_1} and t_{s_2} are enclosed by distinct loops at level d .

$$\begin{aligned}
t_{s_1} &= \{c_1^1, l_1, \dots, c_d^1, l_d, \dots, c_n^1, l_n, c_{n+1}^1\} \\
t_{s_2} &= \{c_1^2, l_1, \dots, c_d^2, l_d, \dots, c_n^2, l_n, c_{n+1}^2\}
\end{aligned}$$

where $c_d^1 \neq c_d^2$. To fuse the loops enclosing s_1 and s_2 at level d the auxiliary loop value of s_2 is set to the value of the auxiliary loop of s_1 at level d , that is, $c_d^2 = c_d^1$, and auxiliary loop values $c_{d+1}, \dots, c_n, c_{n+1}$ are updated.

Loop Distribution: Assume a set of statements s_1, \dots, s_n are enclosed by the same loop at level d , with transformation relations $t_{s_k}, 1 \leq k \leq n$.

$$t_{s_k} = \{c_1^k, l_1, \dots, c_d^k, l_d, \dots, c_n^k, l_n, c_{n+1}^k\}$$

where $c_d^1 = \dots = c_d^n = c_d$. To distribute statement s_i the algorithm determines new values for the auxiliary loops at level d based on the dependence graph:

- Build a directed graph G , where each node corresponds to a statement enclosed in loop l_d and each edge corresponds to a data dependence not carried by loops l_1, \dots, l_{d-1} .
- Traverse each strongly connected component of G in topological order until s_i is found. Let SCC_i be the strongly connected component including s_i .

- If SCC_i is the last component of G , increment the auxiliary loop values c_d^k of all s_k in SCC_i .
- Otherwise, increment the auxiliary loop values c_d^k of all s_k in G that have not been traversed yet.

5 Transformation Script

The transformation script is a prescription for how to optimize the code. Each line of the script describes a loop transformation to be applied on the existing loop representation. For brevity, we only list most common high-level loop transformations below. As a general rule, each loop transformation affects a set of statements which are enclosed in the same loop of the statement specified by the parameters.

permute($[stmt], order$): the loop order of $stmt$ is permuted to the new $order$, which is represented by a sequence of integers identifying the loops. If permute does not have a $stmt$ parameter, it indicates that the loop order of all statements should be permuted.

tile($stmt, loop, size, [outer-loop]$): Tile loop level $loop$ of $stmt$ with the tile controlling loop at loop level $outer-loop$ (default value 1), with tile size $size$.

unroll($stmt, loop, size$): Unroll $stmt$'s loop level $loop$ by unroll factor $size$. For all unrolled statements, the inner loop bodies below loop level $loop$ are jammed together.

datacopy($stmt, loop, array, [index]$): For the specified $array$ in $stmt$, a temporary array copy construction is introduced for all $array$ accesses touched within loop level $loop$. The $index$ (default value 0) specifies which subscript in $array$ corresponds to the new temporary array's first index (assuming Fortran array layout). The $array$ accesses in $stmt$ are replaced by appropriate temporary array accesses.

split($stmt, loop, condition$): Split $stmt$'s loop level $loop$ into multiple loops according to $condition$. The original $stmt$'s iteration space will satisfy $condition$. The iteration space satisfying the complement of $conditions$ will be split into new statements.

nonsingular($matix$): Transform the perfect loop nest according to nonsingular $matix$. This includes both unimodular and nonunimodular transformations.

6 Experimental Results

We now present performance results for automatically-generated code for three well-known computational kernels, Matrix Multiply, Triangular Solver and LU

Architecture	Code Version	Compiler/Library	Optimization Flags
Pentium M L1: 32KB, 8-way L2: 1MB, 8-way	opt	ifort 9.1.036	-O2 -msse2
	native	ifort 10.0.023	-O3 -xN
	ATLAS library	3.7.14	-
	vendor library	MKL 8.0.2	-
MIPS R10000 L1: 32KB, 2-way L2: 1MB, 8-way	opt	MIPSpro 7.3.0	-O3 -LNO:blocking=off:fusion=0:fission=0: interchange=off:ou_max=1
	native	MIPSpro 7.3.0	-O3
	ATLAS library	3.7.8	-
	vendor library	SCSL 1.4.1.2	-

Table 1: Compiler, optimization flags, and library versions

Decomposition, on two different architectures, Intel Pentium M and SGI MIPS R10000. We then compare the performance of our optimized code with the self-tuning library ATLAS, the vendor-provided libraries and the native compilers. The compilers used for each architecture are listed in Table 1. For our optimized codes (line “opt” in the table), we turn off the native compiler’s loop transformations to prevent them from interfering with our optimizations.

6.1 Methodology

The transformation scripts for each kernel were derived by hand based on extensions to the algorithm in [12] (extended to handle imperfect loop nests with non-rectangular bounds and incorporate iteration space splitting). We used a simple search strategy in the experiments presented in this section. The unbound parameters in the scripts (unroll factors and tile sizes) are constrained by the storage capacity of their associated memory hierarchy levels. In addition, for tile sizes we use a simple heuristic that tries to fit references with temporal reuse into half of the cache, leaving the other half for references with spatial or no reuse.

We first search for the unroll factors of the innermost loops. Since the register file is small we only searched at most nine points in the space, 1x1, 1x2, 2x1, 1x4, 4x1, 2x2, 2x4, 4x2 and 4x4, to find the unroll factors resulting in best performance, and fewer if register capacity was exceeded. For tile dimensions we sample sizes that are powers of 2 that meet the storage constraints and the approximation equations (see Figures 8(c), 9(c) and 10(e)). For example in Figure 11(c), three tile size variables are constrained by two approximation equations, essentially leaving only one free variable to search. Starting at the upper bound of the constraints, and approximation equations and a square tile shape, we sample the tile dimensions that are a factor of 2. Overall, for each transformation script, we searched no more than 20 points to find the parameter values resulting in the performance presented for our optimized codes.

<pre> DO J=1, N DO K=1, N DO I=1, N C(I, J) = C(I, J) + A(I, K) * B(K, J) </pre>	<pre> permute([3,1,2]) tile(0,2,TJ) tile(0,2,TI) tile(0,5,TK) datacopy(0,3,2,[1]) datacopy(0,4,3) unroll(0,4,UI) unroll(0,5,UJ) </pre>	$TK * TI \approx \frac{1}{2}(SIZE_{L2}/8)$ $TK * TJ \approx \frac{1}{2}(SIZE_{L1}/8)$ $UI * UJ < SIZE_R$
(a) Original code	(b) Script	(c) Constraints

Figure 11: Matrix Multiply code and scripts

<pre> DO J=1, N DO K=1, N DO I=K+1, N B(I, J) = B(I, J) - B(K, J) * A(I, K) </pre>	<pre> permute([1,3,2]) tile(0,3,TK) split(0,2,[d3 ≥ d1+TK]) tile(0,3,TI,2) tile(0,3,TJ,2) datacopy(0,3,2) datacopy(0,4,3,[1]) unroll(0,4,UJ1) unroll(0,5,UI1) datacopy(1,2,3,[1]) unroll(1,2,UJ2) unroll(1,3,UI2) </pre>	$TK * TK \leq \frac{1}{2}(SIZE_{L2}/8)$ $TK * TJ \approx \frac{1}{2}(SIZE_{L2}/8)$ $TK * TI \approx \frac{1}{2}(SIZE_{L1}/8)$ $UI_1 * UJ_1 < SIZE_R$ $UI_2 * UJ_2 < SIZE_R$
(a) Original code	(b) Script	(c) Constraints

Figure 12: Triangular Solver code and scripts

```

DO K=1,N-1
DO I=K+1,N
s1  A(I,K)=A(I,K)/A(K,K)
DO I=K+1,N
DO J=K+1,N
s2  A(I,J)=A(I,J)-A(I,K)*A(K,J)

```

$IS_1 : \{[k, i, j] | 1 \leq k \leq N-1 \wedge k+1 \leq i \leq N \wedge j = k+1\}$
 $IS_2 : \{[k, i, j] | 1 \leq k \leq N-1 \wedge k+1 \leq i, j \leq N\}$

(b) Aligned iteration spaces

(a) Original code

```

s1→s2: (0,0,+), (0,0,0)
s2→s1: (+,0,1), (+,+,1)
s2→s2: (+,0,+), (+,0,+), (+,+,0)

```

(c) Dependence vectors

```

permute([1,2,3])
tile(1,3,TJ,1)
split(1,2,[d2≤d1-2])
permute(2,[1,2,4,3])
permute(1,[1,3,4,2])
split(1,2,[d2≥d1-1])
tile(3,2,TI,2)
split(3,3,[d5≤d2-1])
tile(3,5,TK,3)
tile(3,5,TJ,4)
datacopy(3,4,2,[1])
datacopy(3,5,3)
unroll(3,5,UI,1)
unroll(3,6,UJ,1)
datacopy(4,3,2,[1])
tile(1,4,TK,2)
tile(1,3,TI,3)
tile(1,5,TJ,4)
datacopy(1,4,2,[1])
datacopy(1,4,3)
unroll(1,5,UI,2)
unroll(1,6,UJ,2)

```

(d) Transformation script

```

TK1 * TI1 ≈ ½(SIZEL2/8)
TK1 * TJ1 ≈ ½(SIZEL1/8)
TI1 * TI1 ≤ ½(SIZEL2/8)
TK2 * TI2 ≈ ½(SIZEL2/8)
TK2 * TJ2 ≈ ½(SIZEL1/8)
UI1 * UJ1 < SIZER
UI2 * UJ2 < SIZER

```

(e) Constraints on parameters

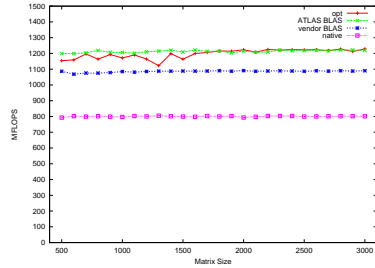
```

DO T2=2,N,64
DO T4=2,T2-64,256
DO T6=1,T4-1,256
DO T8=T6,MIN(T4-1,T6+255)
DO T10=T4,MIN(T2-2,T4+255)
P1(T8-T6+1,T10-T4+1)=A(T10,T8)
DO T8=T2,MIN(T2+56,N),8
DO T10=T8,MIN(N,T8+7)
DO T12=T6,MIN(T6+255,T4-1)
P2(T12-T6+1,T10-T8+1)=A(T12,T10)
DO T10=T4,MIN(T2-2,T4+255)
DO T12=T8,MIN(N,T8+7)
DO T14=T6,MIN(T6+255,T4-1)
A(T10,T12)=A(T10,T12)-P1(T14-T6+1,T10-T4+1)*
P2(T14-T6+1,T12-T8+1)
DO T6=T4,MIN(T4+254,T2-3)
DO T8=T6+1,MIN(T4+255,T2-2)
P3(T6-T4+1,T8-(T4+1)+1)=A(T8,T6)
DO T6=T4+1,MIN(T4+255,T2-2)
DO T8=T2,MIN(N,T2+63)
DO T10=T4,T6-1
A(T6,T8)=A(T6,T8)-P3(T10-T4+1,T6-(T4+1)+1)*A(T10,T8)
DO T4=1,T2-65,256
DO T6=T2-1,N,256
DO T8=T4,MIN(T4+255,T2-2)
DO T10=T6,MIN(T6+255,N)
P4(T8-T4+1,T10-T6+1)=A(T10,T8)
DO T8=T2,MIN(T2+56,N),8
DO T10=T8,MIN(N,T8+7)
DO T12=T4,MIN(T4+255,T2-2)
P5(T12-T4+1,T10-T8+1)=A(T12,T10)
DO T10=T6,MIN(T6+255,N)
DO T12=T8,MIN(N,T8+7)
DO T14=T4,MIN(T2-2,T4+255)
A(T10,T12)=A(T10,T12)-P4(T14-T4+1,T10-T6+1)*
P5(T14-T4+1,T12-T8+1)
DO T4=T2-1,MIN(N-1,T2+62)
DO T8=T4+1,N
A(T8,T4)=A(T8,T4)/A(T4,T4)
DO T6=T4+1,MIN(N,T2+63)
DO T8=T4+1,N
A(T8,T6)=A(T8,T6)-A(T8,T4)*A(T4,T6)

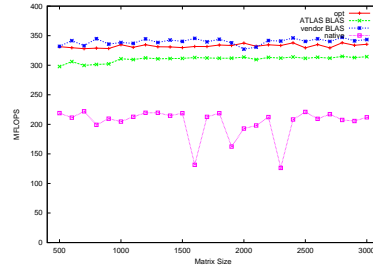
```

(f) Generated code from script with bound parameters in Figure 16(a)

Figure 13: LU Factorization code and scripts



(a) Performance from the script in Figure 11(b) on Pentium M with $TI = 256, TJ = 8, TK = 256, UI = UJ = 1$



(b) Performance from the script in Figure 11(c) on MIPS R10000 with $TI = 512, TJ = 16, TK = 128, UI = UJ = 4$

Figure 14: Performance results of Matrix Multiply

6.2 Matrix Multiply

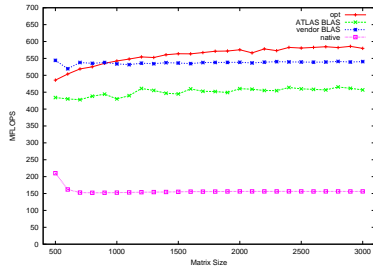
Figure 11 shows the transformation scripts used to optimize Matrix Multiply. Based on the decision algorithm in [12], the optimization strategy consists of exploiting the reuse of $C(I, J)$ in registers, and the reuse of $A(I, K)$ and $B(K, J)$ in caches (A and B have the same amount of temporal reuse, carried by different loops). Figure 11(b) shows the script that applies tiling to keep B in the L1 and A in the L2 cache. Data copying is applied to avoid conflict misses.

To expose SSE optimization opportunities to the Intel compiler, the copying of A transposes the data into the temporary array, which is not required on the SGI machine. Empirical search determines that the best performance for Pentium M is achieved without unrolling, while for R10000 both UI and UJ are set to 4.

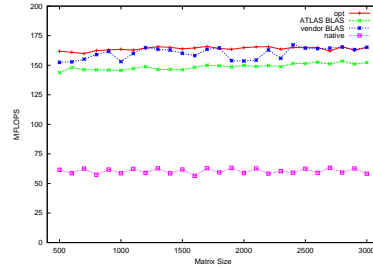
Figure 14 shows the performance of our best code version as compared to ATLAS, the vendor hand-tuned libraries and the native compiler. On the Intel, our generated code achieves an average speedup of 1.50x over the native compiler, 1.10x over the Intel's MKL library and is only 1.2% slower than ATLAS on average. On the SGI, our generated code achieves an average speedup of 1.67x over the native compiler, 1.07x over ATLAS and is only 2.2% slower than the SGI's SCSL library.

6.3 Triangular Solver

The original code and transformation script for Triangular Solver are shown in Figure 12. The same script results in the best code variant for both Pentium and SGI, except transpose in data copying, as shown in optional parameter in datacopy. The two inner loops are permuted to reuse $B(I, J)$ in registers, and loops I and J



(a) Performance from the script in Figure 12(b) on Pentium M with $TI = 8, TJ = 256, TK = 256, UI_1 = UJ_1 = UI_2 = UJ_2 = 1$



(b) Performance from the script in Figure 12(b) on MIPS R10000 with $TI = 16, TJ = 512, TK = 128, UI_1 = UJ_1 = UI_2 = UJ_2 = 4$

Figure 15: Performance results of Triangular Solver

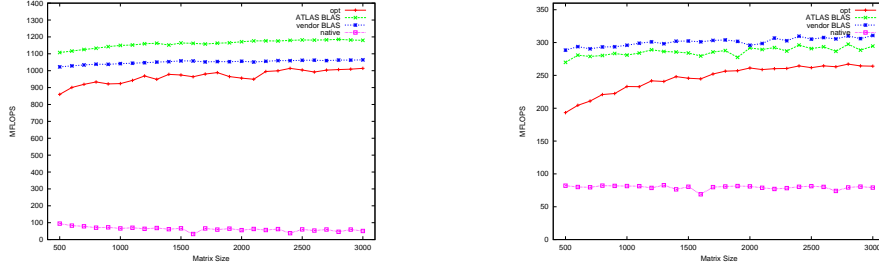
are unrolled (note that this unrolling is specified at the end of the script). For data reuse in cache, loop K is tiled first. The splitting condition is based on the decision to separate read access $B(I, J)$ from write access $B(K, J)$. After splitting, one subloop has non-overlapping read and write accesses and it is then optimized in the same way as Matrix Multiply. The other subloop has only one non-overlapping read access $A(I, K)$, for which data copy is applied.

Figure 15 shows the performance of this code variant. On the Intel this variant, with the best parameter values found by the search, achieves an average speedup of 3.57x over the native compiler, 1.24x over ATLAS, and 1.04x over Intel’s MKL library. On the SGI, this code variant achieves an average speedup of 2.72x over the native compiler, 1.10x over ATLAS, and 1.02x over SGI’s SCSL library.

6.4 LU Factorization

Figure 13 shows the original LU code and its transformation script. The script in Figure 13(d) uses an optimization strategy similar to LAPACK. Loop J is tiled and the loops enclosed by the tile controlling loop are split, separating an imperfect loop nest (a mini-LU) from a perfect loop nest. Splitting the perfect loop nest results in one perfect loop nest with non-overlapping array accesses (a GEMM kernel) with another perfect loop nest with overlapping array accesses (a TRSM kernel). Each kernel is then handled independently and further transformed. Figure 13(f) shows the automatically generated code for this script with the best parameter values found on the Intel.

Figure 16(a) shows the performance of LU on the Intel. Our generated code achieves an average speedup of 16.29x over the native compiler. However, its



(a) Performance from the script in Figure 13(d) on Pentium M with $TJ = 64, TI_1 = TK_1 = TI_2 = TK_2 = 256, TJ_1 = TJ_2 = 8, UI_1 = UI_2 = UJ_1 = UJ_2 = 1$

(b) Performance from the script in Figure 13(e) on MIPS R10000 with $TJ = 64, TI_1 = TK_1 = TI_2 = TK_2 = 256, TJ_1 = TJ_2 = 8, UI_1 = UI_2 = UJ_1 = UJ_2 = 4$

Figure 16: Performance results of LU Factorization

performance is below that of the hand-tuned libraries, achieving 92% of Intel’s MKL library and 83% of ATLAS on average. Figure 16(b) shows the performance on the SGI. Our generated code achieves an average speedup of 3.10x over the native compiler, 82% of SGI’s SCSL library and 86% of ATLAS on average.

7 Conclusion

This paper describes a general and robust framework for composing loop transformations used in memory hierarchy optimization. We demonstrate the effectiveness of this framework on automatically-generated codes for well-known computational kernels that require complex transformations to achieve high performance. The results on Matrix Multiply and Triangular Solver match the performance of ATLAS as well as manually-tuned libraries, within 2.2% and up to 24% faster, and up to an average 3.57x speedup over native compilers. Even for LU, a kernel for which there is no existing compiler that can do end-to-end transformations without user-identified BLAS kernels, we achieve performance within 18% of manually-tuned libraries, and up to a 16x speedup over native compilers. This loop transformation framework represents an important step toward closing the performance gap between compiler-generated and manually-tuned code.

References

- [1] Carr S, Kennedy K. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages*

and Systems Nov 1994; **16**(6):1768–1810.

- [2] Wolf ME, Lam MS. A data locality optimizing algorithm. *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1991.
- [3] Temam O, Granston ED, Jalby W. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. *Proceedings of Supercomputing '93*, 1993.
- [4] Rivera G, Tseng CW. Data transformations for eliminating conflict misses. *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1998.
- [5] McKinley KS, Carr S, Tseng CW. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems* Jul 1996; **18**(4):424–453.
- [6] Bilmes J, Asanović K, Chin CW, Demmel J. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. *Proceedings of the 1997 ACM International Conference on Supercomputing*, 1997.
- [7] Frigo M. A fast Fourier transform compiler. *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1999.
- [8] Whaley RC, Petitet A, Dongarra JJ. Automated empirical optimization of software and the ATLAS project. *Parallel Computing* Jan 2001; **27**(1–2):3–35.
- [9] Yotov K, Li X, Ren G, Cibulskis M, DeJong G, Garzaran M, Padua D, Pingali K, Stodghill P, Wu P. A comparison of empirical and model-driven optimization. *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2003.
- [10] Yotov K, Li X, Ren G, Garzaran M, Padua D, Pingali K, Stodghill P. Is search really necessary to generate high-performance BLAS? *Proceedings of the IEEE: Special Issue on Program Generation, Optimization, and Platform Adaptation* Feb 2005; **93**(2):358–386.
- [11] Yotov K, Pingali K, Stodghill P. Think globally, search locally. *Proceedings of the 2005 ACM International Conference on Supercomputing*, 2005.

- [12] Chen C, Chame J, Hall MW. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. *Proceedings of the International Symposium on Code Generation and Optimization*, 2005.
- [13] Kelly WA. Optimization within a unified transformation framework. PhD Thesis, University of Maryland Dec 1996.
- [14] Lim AW, Lam MS. Maximizing parallelism and minimizing synchronization with affine partitioning. *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, 1997.
- [15] Girbal S, Vasilache N, Bastoul C, Cohen A, Parello D, Sigler M, Temam O. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming* Jun 2006; **34**(3):261–317.
- [16] Pouchet LN, Bastoul C, Cohen A, Vasilache N. Iterative optimization in the polyhedral model: Part I, one-dimensional time. *Proceedings of the International Symposium on Code Generation and Optimization*, 2007.
- [17] Ahmed N, Mateev N, Pingali K. Tiling imperfectly-nested loop nests. *Proceedings of Supercomputing '00*, 2000.
- [18] Yi Q, Kennedy K, Adve V. Transforming complex loop nests for locality. *The Journal of Supercomputing* Mar 2004; **27**(3):219–264.
- [19] Coleman S, McKinley KS. Tile size selection using cache organization and data layout. *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1995.
- [20] Jiménez M, Llabería JM, Fernández A. Register tiling in nonrectangular iteration spaces. *ACM Transactions on Programming Languages and Systems* Jul 2002; **24**(4):409–453.
- [21] Barthou D, Donadio S, Carribault P, Duchateau A. Loop optimization using hierarchical compilation and kernel decomposition. *Proceedings of the International Symposium on Code Generation and Optimization*, 2007.
- [22] Donadio S, Brodman J, Roeder T, Yotov K, Barthou D, Cohen A, Garzarán MJ, Padua D, Pingali K. A language for the compact representation of multiple program versions. *Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing*, 2005.

- [23] Kelly W, Pugh W. A framework for unifying reordering transformations. *Technical Report CS-TR-3193*, Department of Computer Science, University of Maryland 1993.
- [24] Kelly W, Maslov V, Pugh W, Rosser E, Shpeisman T, Wonnacott D. The Omega Library interface guide. *Technical Report CS-TR-3445*, University of Maryland at College Park Mar 1995.
- [25] Bastoul C. Code generation in the polyhedral model is easier than you think. *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2004.
- [26] Banerjee U. Unimodular transformations of double loops. *Proceedings of the 3rd International Workshop on Languages and Compilers for Parallel Computing*, 1990.
- [27] Li W, Pingali K. A singular loop transformation framework based on non-singular matrices. *International Journal of Parallel Programming* 1994; **22**(2):183–205.
- [28] Xue J. Automating non-unimodular loop transformations for massive parallelism. *Parallel Computing* May 1994; **20**(5):711–728.
- [29] Fernández A, Llabería JM, Valero-García M. Loop transformation using nonunimodular matrices. *IEEE Transactions on Parallel and Distributed Systems* Aug 1995; **6**(8):832–840.
- [30] Ramanujam J. Beyond unimodular transformations. *The Journal of Supercomputing* Feb 1995; **9**(4):365–389.
- [31] Allen R, Kennedy K. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers, 2002.