UMIACS-TR-94-87.1 November, 1994
CS-TR-3317.1

Code Generation for Multiple Mappings

Wayne Kelly
wak@cs.umd.edu
Dept. of Computer Science

William Pugh
pugh@cs.umd.edu
Institute for Advanced Computer Studies
Dept. of Computer Science

Evan Rosser
ejr@cs.umd.edu
Dept. of Computer Science

Univ. of Maryland, College Park, MD 20742

Abstract

There has been a great amount of recent work toward unifying iteration reordering transformations. Many of these approaches represent transformations as affine mappings from the original iteration space to a new iteration space. These approaches show a great deal of promise, but they all rely on the ability to generate code that iterates over the points in these new iteration spaces in the appropriate order. This problem has been fairly well-studied in the case where all statements use the same mapping. We have developed an algorithm for the less well-studied case where each statement uses a potentially different mapping. Unlike many other approaches, our algorithm can also generate code from mappings corresponding to loop blocking. We address the important trade-off between reducing control overhead and duplicating code.

This work is supported by an NSF PYI grant CCR-9157384 and by a Packard Fellowship.

Code Generation for Multiple Mappings

Wayne Kelly
wak@cs.umd.edu

William Pugh
pugh@cs.umd.edu

Evan Rosser
ejr@cs.umd.edu

Department of Computer Science
University of Maryland, College Park, MD 20742

Abstract

There has been a great amount of recent work toward unifying iteration reordering transformations. Many of these approaches represent transformations as affine mappings from the original iteration space to a new iteration space. These approaches show a great deal of promise, but they all rely on the ability to generate code that iterates over the points in these new iteration spaces in the appropriate order. This problem has been fairly well-studied in the case where all statements use the same mapping. We have developed an algorithm for the less well-studied case where each statement uses a potentially different mapping. Unlike many other approaches, our algorithm can also generate code from mappings corresponding to loop blocking. We address the important trade-off between reducing control overhead and duplicating code.

1 Introduction

Optimizing compilers apply iteration reordering transformations for a variety of reasons. By changing the order of computations in a loop, these transformations can expose parallelism and improve data locality. They can also be used together with other techniques to improve the efficiency of SPMD code, for example, by moving communication statements out of loops, or by restructuring loops to avoid the execution of iterations which do no work.

Traditionally, reordering transformations have been used by applying a sequence of pre-specified transformations such as loop interchange, loop distribution, skewing, tiling, index set splitting and statement reordering [21]. Each of these transformations has its own legality checks and transformation rules. These checks and rules make it hard to analyze or predict the effects of a sequence of these transformations without actually performing the transformations and analyzing the resulting code.

This complexity has inspired a great deal of recent work toward unified systems for iteration reordering transformations[3, 20, 15, 10, 8]. These approaches use a variety of formalisms, but most can be considered as special cases of a formalism we have developed [12]. In our formalism, transformations are represented as one-to-one mappings from the original iteration space to a new iteration space. We allow a potentially different mapping to be used for each atomic statement. We restrict the mappings to be those that can be represented using affine constraints.

Unimodular transformations can be viewed as the special case where there is a single atomic statement (the body of a set of perfectly nested loops) and the mapping is restricted to be linear and onto. The extended unimodular transformations developed by Li and Pingali [15] removes the onto restriction. The schedules produced by Feautrier [10] are not, by themselves, one-to-one, but when they are combined with the space mappings, they become one-to-one. Feautrier allows a potentially different schedule to be used for each *atomic statement*. Schedules are also used by a number of other researchers [14, 8].

We use the following notation to represent the mapping used for statement s_p:

$$T_p : [i_1, \ldots, i_m] \rightarrow [f_1, \ldots, f_n]$$

where:

- $i_1, \ldots, i_m$ are the index variables of the loops nested around statement s_p.
- The f_j's (called *mapping components*) are quasi-affine functions [1] of the iteration variables and symbolic constants.

This mapping represents the fact that iteration $[i_1, \ldots, i_m]$ in the original iteration space of statement s_p is mapped to iteration $[f_1, \ldots, f_n]$ in the new iteration space.

Finding legal mappings that produce efficient code is an important and difficult problem, but is not discussed in this paper. We refer interested readers to our earlier work in that area [12, 11, 13].

This paper deals with the problem of generating transformed code given an original program and a mapping. This involves creating loops and conditionals that iterate over all and only those points in the new iteration space. When each statement uses the same mapping, and that mapping is linear and onto, code generation is relatively simple. If we start with a convex iteration space and apply a one-to-one and onto mapping, then the transformed iteration space will also be convex. The problem of generating perfectly nested loops to iterate over all and only those points in such a convex region has been studied by a number of researchers starting with the seminal work of Ancourt and Irigoin [1].

If the original iteration space is non-convex (as a consequence of non-unit loop steps), or if the mapping applied is not onto, then the transformed iteration space may be non-convex. In these cases it is still possible to generate suitable perfectly nested loops; however, some of the loop steps will be non-unit. Techniques for handling this case are described by Li and Pingali [15].

Our work addresses the case where a potentially different mapping is used for each statement. The corresponding transformed iteration space can be "very" non-convex; that is, there is no set of perfectly nested loops without conditionals, even with non-unit steps, that can scan the space. The simplest code for a non-convex iteration space scans the convex hull of the space, and tests the conditions under which each statement should be executed at the innermost level. This method can incur a high control overhead. We can eliminate control overhead by breaking the convex hull into a sequence of smaller, tighter regions, which eliminates the need for conditionals at the expense of code duplication. Figure 1 shows an example of this. Eliminating control overhead tends to be particularly important for transformations that radically alter the structure of the original program such as when loop blocking is performed or when the transformed iteration spaces of different statements overlap in complex ways.

Our algorithm can be summarized as follows: We first construct an *abstract syntax tree* (AST) that defines an initial structure of the loops and conditions. In determining the initial structure we try to introduce as little control overhead as possible under the restriction that no code duplication is introduced. Section 3 describes the structure of our abstract syntax trees, and Section 4, describes how we determine an initial AST that produces no code duplication. Next, we augment this tree with more detailed information regarding the conditions and loop bounds of the conditionals and loops respectively. This is described in Section 5.

Code generated for the iteration spaces:

$$I_1 : \{[i,j] : 1 \le i \le 10 \land j = 1\}$$
$$I_2 : \{[i,j] : 1 \le i \le 5 \land 1 \le j \le 10\}$$

Code with no duplication:

```
for i = 1 to 10
    for j = 1 to 10
        if (j = 1) s1[i,j]
        if (i < 5) s2[i,j]
```

Code with no avoidable control overhead:

```
for i = 1 to 5
    s1[i,1]
    for j = 1 to 10
        s2[i,j]
for i = 6 to 10
    s1[i,1]
```

Figure 1: Control overhead versus code duplication

Next we consider the problem of removing control overhead. Sources of overhead nested inside the most loops will be executed most frequently and are the most important to remove. But further removing overhead requires code duplication and an increase in code size. This trade-off is controlled by specifying the depths from which overhead will be eliminated. This optimization algorithm is described in Section 7.

Once we have performed this optimization, we generate the actual code using the abstract syntax tree and the information that it contains. Section 6 describes how to generate code from an AST.

Before describing the actual algorithm we first summarize the Omega library, a set of routines that we use to represent and manipulate sets of affine constraints, in Section 2.

2 The Omega library

Many code generation algorithms use linear algebra to represent and manipulate sections of iteration spaces. We use higher-level abstractions called *tuple sets* and *tuple relations*. An integer k-tuple is a point in $\mathcal{Z}^k$. A *tuple relation* is a mapping from tuples to

restrict_domain (R , S)
  $\{ i \rightarrow j \mid i \rightarrow j \in R \wedge i \in S \}$
range (R)
  $\{ j \mid i \rightarrow j \in R \}$
project (S , $1 \ldots r$)
  $\{ [t_1 \ldots t_r] \mid \exists$ integers $t_{r+1} \ldots t_n$ s.t. $[t_1, \ldots, t_r, t_{r+1}, \ldots, t_n] \in S \}$
gist (S , K)
  least constrained set S' s.t. $(S' \wedge K) \Leftrightarrow (S \wedge K)$
convex_hull ($\{S_1, \ldots, S_m\}$)
  TupleSet represented by constraints
  $\{ c \mid c$ part of $S_i \wedge \forall_{j=1 \ldots m} (S_j \wedge c) \Leftrightarrow S_j \}$

Figure 2: Functions provided by the Omega Library

tuples and a *tuple set* is a set of tuples. Tuple relations and sets are represented using the Omega Library [16, 18], which is a set of routines for manipulating affine constraints over integer variables. The relations and sets may involve symbolic constants such as n in the following example: $\{ [i] \rightarrow [i+1] \mid 1 \leq i \leq n \}$. They may also involve existentially quantified variables such as α in the following example: $\{ [i] \mid \exists \alpha \text{ s.t. } i = 2\alpha \wedge 1 \leq i \leq 10 \}$. Relationships between the variables are represented by a disjunction of conjunctions of affine constraints. Figure 2 gives a brief description of the operations on tuple relations and sets that we use in code generation. All operators return their results in the simplest form possible, i.e. redundant constraints are always detected and removed.

3 Code Structure

This section describes the abstract syntax trees (AST) that we use to define the structure of the loops and conditions. An AST can contains three types of nodes:

split nodes - This type of node is labeled with a condition c and has two children named true_child and false_child. A node of this type corresponds to a sequence of two code fragments. The first code fragment will execute iterations that satisfy the condition c and all of the conditions[1] contained in split nodes above the current one. The second code fragment will execute iterations that satisfy the condition $\neg c$ and all of the conditions contained in split nodes above the current one. The true_child defines the structure of the first code fragment and the false_child defines the structure of the second code fragment. The condition must be such that the iterations executed by the true branch are lexicographically less than the iterations executed by the false branch.

loop nodes - This type of node is labeled with an index variable t_k and has only one child. A node of this type corresponds to a `for` loop, possibly surrounded by a conditional statement. The `for nodes` loop iterates over all valid values of the index variable t_k. If there exists an iteration $[i_1, \ldots, i_n]$ of any statement s that satisfies all of the conditions contained in split nodes above the current node then i_k is a valid value. The conditional statement is inserted, if necessary, to enforce constraints contained in split nodes above the current node that don't involve the current index variable and to ensure that at least one iteration of the `for` loop will be executed.

leaf - As its name implies, this type of node has no children. A node of this type corresponds to a sequence of atomic statements. Each atomic statement is surrounded, if necessary, by a conditional statement to ensure that only those iterations that satisfy all of the conditions contained in split nodes above the current node are executed.

Every path from the root to a leaf contains n loop nodes labeled with index variables $t_1, \ldots, t_n$ in that order. The condition in a split node refers only to index variables $t_1, \ldots, t_k$, where t_k is the label of the first loop node below that split node. Figure 4 contains an example of an AST and its corresponding code.

4 Initial code structure

In this section we describe how to construct the AST that defines the initial structure of the loops and conditions. In the initial structure we try to introduce as little control overhead as possible, under the restriction that no code is duplicated. If we were to use this initial structure to generate code, then we would obtain code that is correct but might contain too much overhead. In Section 7 we describe how to modify the initial structure, to decrease overhead at the expense of increased code size. The algorithm to construct the initial AST is given in Figure 4 and explained below.

[1] Whenever we refer to the condition in a split node above some current node, we are actually referring to either that condition or the negation of that condition, depending on which sub-tree of that split node the current node is contained in.

Original code

```
for k = 1 to n
  for i = k+1 to n
    a(i,k) = a(i,k) / a(k,k)
      for j = k+1 to n
        a(i,j) = a(i,j) - a(i,k) * a(k,j)
```

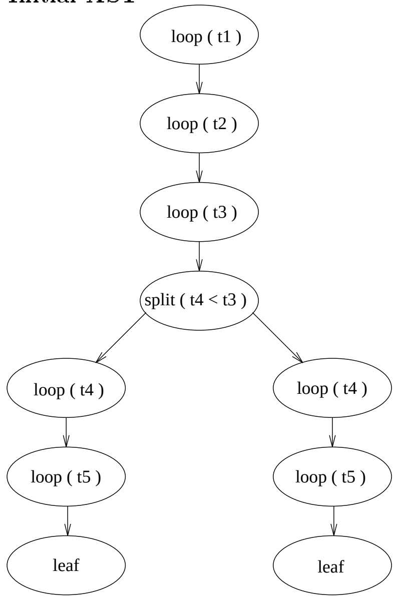
Mapping

$$T_{10} : \{[k,i\ \ ] \rightarrow [64((k-1)\ div\ 64)+1, 64(i\ div\ 64), k, k, i]\}$$
$$T_{20} : \{[k,i,j] \rightarrow [64((k-1)\ div\ 64)+1, 64(i\ div\ 64), j, k, i]\}$$

New Iteration Space:

$$I_1 : \{[t_1,t_2,t_3,t_4,t_5] | \exists \alpha, \beta \text{ s.t. } t_1 = 1 + 64\beta \wedge t_2 = 64\alpha \wedge t_3 - 63, 1 \le t_1 \le t_3 < t_5 \le t_2 + 63, n \wedge t_2 \le t_5 \wedge t_4 = t_3\}$$
$$I_2 : \{[t_1,t_2,t_3,t_4,t_5] | \exists \alpha, \beta \text{ s.t. } t_1 = 1 + 64\alpha \wedge t_2 = 64\beta \wedge t_4 - 63, 1 \le t_1 \le t_4 < t_5 \le t_2 + 63, n \wedge t_4 + 1 \le t_3 \le n \wedge t_2 \le t_5\}$$

Initial AST



Code corresponding to the initial AST

```
if 2<=n then
  for t1 = 1 to n-1 step 64
    for t2 = t1-1 to n step 64
      for t3 = t1 to n
        if t1<t3 then
          for t4 = t1 to min(t2+62,t1+63,t3-1)
            for t5 = max(t2,t4+1) to min(t2+63,n)
              s2[t4,t5,t3]
        if t3<=t2+62 and t3<=t1+63 and t3<=n-1 then
            for t5 = max(t2,t3+1) to min(t2+63,n)
              s1[t3,t5]
```

Figure 3: Initial blocked LU decomposition

Given the original program and a set of mappings, our first step is to compute the new iteration spaces belonging to each of the statements. We restrict the domains of the mappings to the original iteration spaces. The new iteration spaces are the ranges of these restricted mappings. These new iteration spaces may be disjoint or may overlap.

We need to generate code that will execute each statement at all and only those points in its respective iteration space. The new code must also execute the iterations in lexicographical order based on the new coordinate system. Since the new iteration spaces may overlap, the new code may have to interleave the execution of different statements.

At this stage, we calculate only the basic structure of the loop nests, not the loop bounds. We will create nested loops to iterate over all of the points $[t_1, \ldots, t|n]$ in the new iteration spaces. The outermost loop will iterate over the appropriate values of t_1, the next outermost loop iterates over the appropriate values of t_2, and so on.

We build the initial AST in a depth first fashion. At each stage of this construction process, we try to find a suitable condition c on which to split the range of values of the next index variable t_k. The idea is that generating two tighter loops for this index variable, rather than one looser loop, will allow us eliminate some more deeply nested source of control overhead. However, the initial AST must not produce any code duplication, so these split conditions must be chosen such that the new ranges will have disjoint, non-empty sets of active statements. A statement is active if it possesses an iteration that needs to be executed in the branch of the AST currently being constructed. For each active statement, we compute the constraints on t_k for iterations of that statement in terms of $t_1, \ldots, t_{k-1}$. We then check to see if any of these constraints satisfy the above requirements. If such a condition c exists, we generate a split node with c as the condition, and calculate the set active statements for each branch. If necessary, we adjust c so that values of t_k that satisfy it are less than those that do not. We then recursively determine the structure of the child nodes.

If no such c exists, then there is no way to further partition the range of t_k without duplicating code. So, we generate a loop node here for t_k (making t_{k+1} the next index variable). After loops have been generated for each index variable, we generate a leaf node.

5 Evaluating node attributes

In this section we describe how to augment the AST with more detailed information regarding the conditions and loop bounds of the conditionals and loops

```
Generate_Initial_AST ( T, old_IS )
    INPUT:
        T : array [maxStmts] of TupleRelation, T[p] is the mapping associated with statement p.
        old_IS : array [maxStmts] of TupleSet, old_IS[p] is the iteration space of statement p in the original
                code of the statements.
    OUTPUT:
        An AST_node which is the root of the tree which represents the structure of the code.
    ALGORITHM:
        foreach statement s
            new_IS[s] = range ( restrict_domain ( T[s], old_IS[s] ))
            for L = 1 to last_level
                I[s, L] = project ( new_IS[s], 1...L)
        return Partition ( 1, {all stmts} )

Partition ( level, active, I )
    INPUT:
        level : integer Loop levels 1,...,level − 1 have already been generated and should be consided fixed.
        active : set of statement, The statements for which code should be generated
        I : array [maxStmts,maxLevels] of TupleSet, I[s,L] is the new iteration space of statement s projected
                onto symbolic constants and index variables at levels 1...L.
    OUTPUT:
        An AST_node which is the root of the tree which represents the structure of the code.
    ALGORITHM:
        if level = last_level then
            return AST_leaf ( )
        if ∃ constraint c ∈ I[s, level], for some s, s.t. active1 ≠ ∅∧ active2 ≠ ∅∧active1 ∩ active2 = ∅
            where active1 = { s : s ∈ active ∧ {c} ∩ I[s, level] is satisfiable }
              and active2 = { s : s ∈ active ∧ {¬c} ∩ I[s, level] is satisfiable }
        then return AST_split (level, c, Partition (level, active1, I), Partition (level, active2, I))
        else return AST_loop (level, Partition (level+1, active, I))
```

Figure 4: Algorithm to construct the initial AST

respectively. This information is initially used to identify sources of control overhead (see Section 7), and later to generate the actual code (see Section 6). This algorithm is performed on the initial AST and later on the sub-trees of the AST that are modified by the optimization phase.

The algorithm is given in Figure 5 and is explained below. The algorithm performs a depth first traversal of the AST, evaluating attributes of the nodes as it goes. As we move down the tree we maintain two tuple sets: `restrictions` and `known`. The tuple set `restrictions` contains all constraints from split nodes between the current node and nearest loop node above. The tuple set `known` contains all constraints enforced by conditionals and loop bounds above the current node. These tuple sets define the current context; that is, the subsets of the iteration spaces that the code corresponding to the current sub-tree will have to iterate over.

We wish to maintain the property that for every split node, both subtrees represent at least one iteration of some statement. So, when we come to a split node, we check that such iterations exist, and if not, we remove the split node, replacing it with the appropriate child node.

In evaluating a loop node at level l, we compute three things: the statements that should be executed in the loop body, the conditions under which the loop should be executed, and the values of the current index variable t_k the loop should enumerate. We first determine which statements will need to be executed in that loop (those whose iteration spaces intersect `restrictions` ∩ `known`.) Given that set of statements, and the constraints in `known` and `restrictions`, we want to find the strongest conditional and the tightest loop bounds that will not exclude any iterations in those statements' iteration spaces. Any constraints in `restrictions` can be enforced, since any iterations a given constraint excludes will be included in the other subtree of that con-

```
Evaluate ( node, known, restrictions )
  INPUT:
    node : AST_node, the root of the subtree to be evaluated
    known : TupleSet, constraints on index variables and symbolic constants that have been represented
        in above loop nodes
    restrictions : TupleSet, constraints on the current index variable that specify the region whose subtree
        is being evaluated
  OUTPUT:
    This function computes tuple sets that represent the guards and loop bounds for loop nodes and
    guards for the leaf nodes.
  ALGORITHM:
    if (node.type == AST_split)
        if (¬∃ statement s s.t. I[s, node.level] ∩ known ∩ restrictions ∩ node.condition is satisfiable)
            remove node and replace it with node.false_side
            Evaluate (node.false_side, known, restrictions)
        elseif (¬∃ statement s s.t. I[s, node.level] ∩ known ∩ restrictions ∩ ¬ node.condition is satisfiable)
            remove node and replace it with node.true_side
            Evaluate (node.true_side, known, restrictions)
        else
            Evaluate (node.true_side, known, restrictions ∩ node.condition)
            Evaluate (node.false_side, known, restrictions ∩ ¬ node.condition)
    elseif (node.type == AST_loop)
        foreach statement s
            active[s] = (I[s, node.level] ∩ restrictions ∩ known) is satisfiable
        bounds = convex_hull (⋃_{s s.t. active[s]} I[s, node.level]) ∩ known ∩ restrictions ∩
                greatest_common_step (active, node.level, I)
        needsCheck = gist (bounds, known)
        node.guard = project (needsCheck, 1...node.level-1)
        node.loop = gist (needsCheck, guard)
        Evaluate (node.child, bounds, True)
    elseif (node.type == AST_leaf)
        foreach statement s
            node.guard[s] = gist (new_IS[s] ∩ restrictions ∩ known, known)
```

Figure 5: Evaluate Algorithm

straint's split node. We want to add further constraints on t_k so that the loop only iterates over those points for which at least one iteration of some statement is executed. For example, consider the two iteration spaces:

$$I_1 : \{[i] | 1 \leq i \leq 5\}$$
$$I_2 : \{[i] | 1 \leq i \leq 10\}$$

Unless we enforce the constraints $1 \leq i$ and $i \leq 10$, the loop would iterate over points (such as $i = 11$) which do not correspond to an iteration of any statement. However, if we were to add the constraint $i \leq 5$, which is not on the convex hull of the union of the two spaces, we would be erroneously excluding required iterations of statement 2 $(6 \leq i \leq 10)$.

There are also some non-convex constraints that we can enforce. We collect together all stride constraints of the form $\exists \beta$ s.t. $t_k = a_p \beta + b_p$ (where a_p is an integer coefficient and b_p is an affine function of symbolic constants and outer level index variables) that are associated with statements in the loop. We then calculate the *greatest common step* of the loop as follows:

$$\begin{aligned} gcs &= gcd(\{a_p | s_p \text{ is active}\} \cup \\ &\quad \{gcd(b_q - b_p) | s_q \text{ is active} \wedge s_p \text{ is active}\}) \end{aligned}$$

The gcd of an expression is defined to be the gcd of the coefficients in the expression.

We can enforce the constraint

$$\exists \beta \text{ s.t. } t_k = gcs\ \beta + b_p$$

where s_p is an arbitrary statement in the loop, by making gcs the loop step and suitably modifying the lower bound so that it satisfies this constraint. The loop step may not enforce all of the stride constraints on statements in the loop, but we cannot add anything stronger without excluding required iterations. Any remaining stride constraints will be enforced later.

To construct the full set of constraints to be enforced at the loop, we intersect the convex hull of the active statements' iteration spaces with `known`, `restrictions`, and the greatest common step. We use the *gist* operation to remove any constraints that are implied by `known` and thus will be enforced at earlier loop nodes. Once we have determined which constraints can be enforced at this point, we divide them into those that can be enforced in the conditional statement and those that must be enforced by the `for` loop.

To evaluate a leaf node we determine which atomic statements will need to be executed. For each statement that needs to be executed, we calculate the constraints for its iteration space that are not implied by the surrounding loops and conditionals.

6 Generating code from an AST

Generating code in a high-level language is straightforward once we have evaluated the attributes of the AST. The algorithm performs a depth first traversal of the AST, generating the code as it goes.

When we come to a split node, we generate code for the true child and then generate code for the false child. We do not generate a conditional statement at this point; rather, it is the responsibility of the code generated for the child nodes to execute only the appropriate iterations.

When we come to a loop node we generate a `for` loop and possibly a conditional statement around that `for` loop. In the case where we can prove that at most one iteration can execute, we can avoid generating a loop. If the condition attribute of the loop node is not a tautology then an `if` statement is generated to check this condition.

Generating the `for` loop from the bounds attribute of the loop node is slightly more complicated, since the semantics of `for` loops are not defined in terms of (multiple) lower bounds, (multiple) upper bounds and stride constraints. Instead, they are defined in terms of an initial value, a single bound and a step (the bound is either an upper or lower bound, depending on the sign of the step). Since the code we generate enumerates the iteration space in lexicographical order, we need only consider positive steps.

We now need to calculate the initial value: the smallest integer that satisfies both the lower bounds and the stride constraint. Assume that we have a lower bound of the form $L \leq mt_k$, where L is a function of outer level index variables and symbolic constants and m is an integer coefficient. If there is no stride constraint, the initial value implied by this lower bound is $\text{CeilDiv}(L, m)$, where $\text{CeilDiv}(a, b)$ is a function that computes $\lceil L/m \rceil$.

If there is a stride constraint of the form $\exists \beta$ s.t. $t_k = c + s\beta$, the step will be s and the smallest integer that satisfies both the lower bound and the stride constraint is $\text{CeilDiv}(\text{CeilDiv}(L, m) - c, s) * s + c$. In a number of cases, we can generate simpler formulas. For example we can generate simpler formulas when m is 1 or when we can determine that $\text{CeilDiv}(L, m)$ is always a solution to the stride constraint. For space reasons, we do not detail these here.

Finally, we convert multiple upper or lower bounds into a single lower or upper bound using `max` or `min` as appropriate.

When we come to a leaf node we generate code in turn for each active statement. If the guard attribute for a statement is not a tautology then a conditional statement is generated to test this condition. The statements themselves are unchanged from the original program, except that the old index variables are replaced by appropriate functions of the new loop variables. Since the mapping is one-to-one, these functions can be easily determined by simply inverting the mapping.

```
PrintNode (node)
  INPUT:
    node : AST_node, the node for which
      we are generating code.
  OUTPUT:
    for loops and if statements to execute
      the AST
  ALGORITHM:
    if (node.type == AST_split)
      PrintNode (node.left)
      PrintNode (node.right)
    elseif (node.type == AST_loop)
      if (node.guard is not Tautology)
        print_if_then (node.guard)
      print_loop (node.level, node.loop)
      PrintNode (node.child))
    elseif (node.type == AST_leaf)
      foreach statement s
        if (node.guard[s] is Satisfiable)
          if (node.guard[s] is not Tautology)
            print_if_then (node.guard)
          print_statement (s, T[s])
```

Figure 6: Print Code Algorithm

7 Optimizing the code structure

Having generated an initial AST that does not duplicate code, we now consider optimizations that reduce execution time overhead at the cost of an increase in code size. We consider three types of overhead in our implementation: guards around loops, guards around atomic statements, and min's and max's in loop bounds. Zero trip loops are also a potential source of overhead; we generate guards to ensure that loops have at least one iteration, so this problem reduces to the case of guards around loops.

We remove overhead as follows:

1. Find an overhead we wish to remove.
2. Determine a constraint that, if tested, would allow us to eliminate the overhead.
3. Create a new split node that tests that constraint, with the code containing the overhead duplicated under both branches of the split.

Our optimization criteria is the maximum number of loops k_{max}, that we will allow to surround a source of overhead. Roughly speaking, the cost of an overhead nested inside of d loops will be $O(n^d)$, where n is the number of iterations of a loop. We could use more exact methods [17] to evaluate the cost of an overhead. However, the cost of such analysis is probably not worthwhile and would be of questionable benefit (e.g., it is not clear whether it is better to remove a source of overhead executed $mn(n-1)/2$ times or a source executed m^2n times).

If a code fragment executes $O(n^d)$ necessary operations, then executing $O(n^d)$ avoidable overhead is probably unacceptable. Reducing the avoidable overhead to $O(n^{d-1})$ will probably be acceptable in this situation and any further reduction may be unnoticeable. We are able to dynamically control the amount of overhead that we eliminate based on the amount of code explosion seen so far.

Given an AST with loops nested d deep, we first remove overhead nested within d loops. We next remove overhead nested within $d-1$ loops, and so on, until all overhead nested inside k_{max} loops is eliminated.

To remove all overhead nested k_{max} deep, we first traverse down the AST to each loop nested k_{max} deep. Note that in counting nesting depth, we only count loop nodes that require a loop to be generated (i.e., that may contain more than one iteration). Upon reaching a loop nested k_{max} deep, we need to lift out *all* overhead from the body of that loop. We search the body of the loop for a constraint that would eliminate some source of overhead. We then generate a split node on that condition, place copies of the original loop node under both branches of the split node, and attempt to remove overhead from both branches. We only extract one constraint at a time and then reevaluate using the algorithm described in Section 5. Testing one constraint might eliminate the need to test another constraint in one of the two branches (e.g., both $i \leq 10$ and $i \leq 20$ might remove overhead, but $i \leq 20$ would not need to be tested in the true branch of a split on $i \leq 10$).

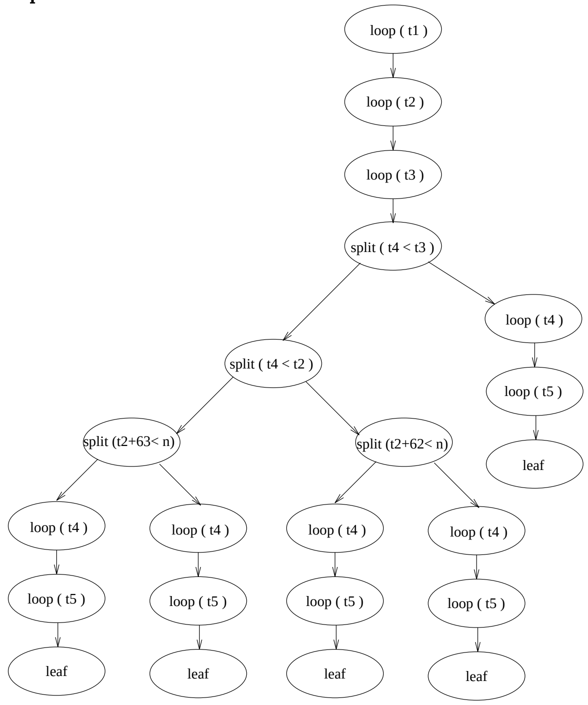
Level of Overhead elim. (k)	Overhead as % of original	Bytes of object code
Naive code	31%	584
Initial code	13%	712
Optimized $k_{max}=4$	13%	712
Optimized $k_{max}=3$	5%	1576
Optimized $k_{max}=2$	4%	5352
Original code	0%	392

Table 1: Results for blocked LU decomposition

Finding constraints that eliminate overheads is fairly straightforward. All guards of atomic statements and loops are sources of overhead and can easily be located by examining the guard attributes of loop and leaf nodes. For loop nodes, we also can check to see if the bounds contain multiple lower bounds or multiple upper bounds. If so, it is straightforward to generate the constraint that eliminates the overhead (e.g., testing $a \leq b$ will eliminate the overhead of computing $\min(a, b)$).

Figure 7 shows the optimized AST and corresponding code for the example introduced in Figure 4, with all overhead inside four or more loops removed. Table 1 shows the overhead (as a percentage of original code time) and code size for different levels of overhead optimization. The blocked version contains more overhead than the original, unblocked code, since it adds extra loops for the blocking. The line marked "Naive" is for comparison only; we do not generate that code. The naive code is a set of loops scanning the convex hull, with all remaining conditions checked as innermost guards. The optimized code for $k_{max}=4$ is identical to the initial case, since are no overheads nested inside 5 loops in the initial code. The results here do not include cache effects, so we can measure the overhead directly. At higher levels of optimization, the code size increases dramatically, with diminishing performance gains. In this example, removing control overhead located inside more than 3 loops ($k_{max}=3$) is probably sufficient.

Optimized AST



Code corresponding to the optimized AST

```
if 2 <= n then
  for t1 = 1 to n-1 step 64
    for t2 = t1-1 to n step 64
      for t3 = t1 to n
        if t1<t3 & 64+t2<=n & 63+t1<=t2 then
          for t4 = t1 to min(t3-1,t1+63,t2-1)
            for t5 = t2 to t2+63
              s2[t4,t5,t3]
        if t1<t3 & 63+t1<=t2 & n<=63+t2 then
          for t4 = t1 to min(t1+63,t3-1,t2-1)
            for t5 = t2 to n
              s2[t4,t5,t3]
        if t1<t3& 63+t2<=n & t2<=63+t1 & t2<t3 then
          for t4=max(t2,t1)to min(t2+62,t1+63,t3-1)
            for t5 = t4+1 to t2+63
              s2[t4,t5,t3]
        if t1<t3& t2<t3 & t2<=63+t1 & n<=62+t2 then
          for t4 = max(t2,t1) to min(t1+63,t3-1)
            for t5 = t4+1 to n
              s2[t4,t5,t3]
        if t3<=62+t2 & t3<=63+t1 & t3<n then
            for t5 = max(t2, t3+1) to min(t2+63,n)
              s1[t3,t5]
```

Figure 7: Optimized blocked LU decomposition

8 Related work

The problem of generating code for a convex region was first addressed by Ancourt and Irigoin [1]. They use Fourier pairwise elimination at each level to provide bounds on each of the index variables. They then form the union of all of these projections to produce a single set of constraints which explicitly contains all of the information necessary to generate code. They propose that fast inexact techniques be used to remove redundancies from this set before it is used to generate code. They consider only the single mapping convex case.

Li and Pingali [15] consider the non-convex case resulting from mappings that are not necessarily onto. They use a linear algebra framework and compute loop bounds and steps using Hermite normal form. They do not consider the multiple mapping case.

Ayguadé and Torres [2] consider a limited case of the multiple mapping case where each statement can have a potentially different mapping but all mappings must have the same linear part (i.e., they only differ in their constant parts).

Chamski [4, 5] generates Nested Loop Structures, which are similar to our AST. He discusses generating code only for the single mapping convex case. He reduces control overhead by generating sequences of loops to remove all min and max expressions in loop bounds. The cost of code duplication may be large when all such overheads are removed. We are able to eliminate control overhead from sources other than min's and max's and we selectively decide which overheads to eliminate by considering both the amount of control overhead and the amount of code duplication that would occur.

Chamski claims [4] that Fourier variable elimination is prohibitively expensive for code generation. We have found it to be a very efficient method, and suspect he used unrealistic examples and/or a poor implementation of Fourier variable elimination. It is well known that Fourier variable eliminate performs poorly on moderate to large systems of constraints where the constraints are dense: each constraint involves many variables. However, the constraints we have seen in both dependence analysis and code generation are quite sparse, and Fourier elimination is quite efficient for sparse constraints [19].

Collard, Feautrier and Risset [7] show how PIP, a parametrized version of the Dual Simplex Method, can be used to solve the simple case. Collard and Feautrier [6] address the multiple mapping case; however, only one dimensional iteration spaces are considered and many guards are generated. They provide some interesting solutions to the situation where statements have incompatible stride constraints (e.g., t_1 is even and t_1 is odd). Stride constraints such as this arise frequently in Feautrier's parallelization framework [9, 10], while we [13, 11] try to avoid them in our framework, since generating good code for them is difficult.

9 Conclusion

We have presented an algorithm to generate transformed code from the original code and a set of one-to-one statement mappings representing the transformation. Unlike most previous systems, our system can generate efficient loop structures even when a potentially different mapping is used with each statement and the resulting union of iteration spaces is non-convex. Our algorithms permit optimizations of the control overhead that results from non-convexity, and allows the user to trade-off lower overhead for code duplication. This approach to generating code and the optimization we describe are particularly important for loop interchange or loop blocking of imperfectly nested loops.

10 Implementation and availability

An implementation of this algorithm is available in the Omega Calculator. The Omega Calculator and copies of our other papers are available from `http://www.cs.umd.edu/projects/omega` and `ftp://ftp.cs.umd.edu/pub/omega`.

References

[1] Corinne Ancourt and Franccois Irigoin. Scanning polyhedra with DO loops. In *Proc. of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 39–50, April 1991.

[2] Eduard Ayguadé and Jordi Torres. Partitioning the statement per iteration space using non-singular matrices. In *International Conference on Supercomputing*, pages 407–415, July 1993.

[3] U. Banerjee. Unimodular transformations of double loops. In *Proc. of the 3rd Workshop on Programming Languages and Compilers for Parallel Computing*, pages 192–219, Irvine, CA, August 1990.

[4] Zbigniew Chamski. Fast and efficient generation of loop bounds. Publication interne 771, Institut de Recherche en Informatique et Systèmes Aléatoires, October 1993.

[5] Zbigniew Chamski. Nested loop sequences: Towards efficient loop structures in automatic parallelization. Publication interne 772, Institut de Recherche en Informatique et Systèmes Aléatoires, October 1993.

[6] J.-F. Collard and P. Feautrier. Automatic generation of data parallel code. In H.J. Sips, editor, *Proceedings of the Fourth International Workshop on Compilers for Parallel Computers*, pages 321–332, Delft, The Netherlands, December 1993.

[7] Jean-Franccois Collard, Paul Feautrier, and Tanguy Risset. Construction of DO loops from systems of affine constraints. Technical Report N° 93-15, Laboratoire de l'Informatique du Parallélisme, Ecolo Normal Supérieure de Lyon, Instiut IMAG, May 1993.

[8] Alain Darte and Yves Robert. Scheduling uniform loop nests. In *Proceedings of ISMN International Conference on Parallel and Distributed Computer Systems*, October 1992.

[9] Paul Feautrier. Some efficient solutions to the affine scheduling problem, Part I, One-dimensional time. *Int. J. of Parallel Programming*, 21(5), Oct 1992.

[10] Paul Feautrier. Some efficient solutions to the affine scheduling problem, Part II, Multidimensional time. *Int. J. of Parallel Programming*, 21(6), Dec 1992.

[11] Wayne Kelly and William Pugh. Determining schedules based on performance estimation. Technical Report CS-TR-3108, Dept. of Computer Science, University of Maryland, College Park, July 1993. to appear in Parallel Processing Letters (1994).

[12] Wayne Kelly and William Pugh. A framework for unifying reordering transformations. Technical Report CS-TR-3193, Dept. of Computer Science, University of Maryland, College Park, April 1993.

[13] Wayne Kelly and William Pugh. Finding legal reordering transformations using mappings. Technical Report CS-TR-3297, Dept. of Computer Science, University of Maryland, College Park, June 1994. also appeared at the 7th annual LCPC workshop.

[14] Herve Leverge, Christophe Mauras, and Patrice Quinton. A language-orientied approach to the design of systolic chips. *Journal of VLSI Signal Processing*, pages 173–182, Mar 1991.

[15] Wei Li and Keshav Pingali. A singular loop transformation framework based on non-singular matrices. In *5th Workshop on Languages and Compilers for Parallel Computing*, pages 249–260, Yale University, August 1992.

[16] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, August 1992.

[17] William Pugh. Counting solutions to presburger formulas: How and why. In *SIGPLAN Conference on Programming Language Design and Implementation*, Orlando, FL, June 1994.

[18] William Pugh and David Wonnacott. Going beyond integer programming with the Omega test to eliminate false data dependences. Technical Report CS-TR-3191, Dept. of Computer Science, University of Maryland, College Park, December 1992.

[19] William Pugh and David Wonnacott. Experiences with constraint-based array dependence analysis. In *Principles and Practice of Constraint Programming Workshop*, Orcas Island, Washington, May 1994.

[20] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, 1991.

[21] Michael Wolfe. *Optimizing Supercompilers for Supercomputers.* Pitman Publishing, London, 1989.