Polyhedra Scanning Revisited

Chun Chen
University of Utah

Abstract

This paper presents a new polyhedra scanning system called CodeGen+ to address the challenge of generating high-performance code for complex iteration spaces resulting from compiler optimization and autotuning systems. The strength of our approach lies in two new algorithms. First, a loop overhead removal algorithm provides precise control of trade-offs between loop overhead and code size based on actual loop nesting depth. Second, an if-statement simplification algorithm further reduces the number of comparisons in the code. These algorithms combined with the expressive power of Presburger arithmetic enable CodeGen+ to support complex optimization strategies expressed in iteration spaces. We compare with the state-of-the-art polyhedra scanning tool CLooG on five loop nest computations, demonstrating that CodeGen+ generates code that is simpler and up to 1.15x faster.

Categories and Subject Descriptors D.3.4 [*Processors*]: Code generation, Compilers, Optimization

General Terms Performance, Algorithms

Keywords polyhedra scanning, polyhedral transformations

1. Introduction

The polyhedral model has long been considered a powerful tool in loop nest optimization. In contrast, ad-hoc loop transformation techniques typically apply one loop transformation at a time and have to switch back to a loop's syntactic form again for the next loop transformation. Subsequent transformations have to deal with increasingly complex loop structures, severely limiting the optimization strategy a compiler can apply. A polyhedral model represents each statement's execution in the loop nest as a lattice point in the space constrained by loop bounds, known as the iteration space. Then a loop transformation can be simply viewed as mapping from one iteration space to another, and various transformations can be composed. This enables reasoning about complex optimization strategies.

In a polyhedral model, polyhedra scanning is used to generate optimized code. The polyhedra representing the iteration spaces of an optimized loop nest are scanned from the first dimension to the last to generate the corresponding output nested loop structure. The quality of generated code directly affects the transformed code's performance. For example, unnecessary control flow in an innermost loop would most likely severely degrade performance on today's architectures. Expensive arithmetic operations such as modulo that result from polyhedra scanning can also degrade performance.

Polyhedra scanning has been studied by researchers since the early 1990s. Generating code from unimodular and non-unimodular transformations [6, 14, 20, 28–30] can be thought of as a polyhedra scanning problem. Ancourt and Irigoin [1] are among the first researchers to propose an algorithm to systematically scan polyhedra to generate loop code. However this method only deals with a single polyhedron. In a general polyhedral framework, statements may have different iteration spaces from each other. Kelly et al. [12] describe a polyhedra scanning system for a set of polyhedra. Their algorithms remove loop overhead from inner loops iteratively from the most compact code generated. Quillere et al. [18] argue that the iterative algorithms in [12] are inefficient, and instead split overlapping polyhedra at each dimension during the scanning process without iterating. However, the method in [18] still requires backtracking to remove dead code. In practice, removing overhead indiscriminately might unnecessarily cause code explosion with little benefit in performance. This might be a particular concern when generating codes for embedded systems where limiting code size is also important. Sometimes removing overhead from just the innermost loop may be a better choice in balancing code size and performance, as it is typically the structure of the inner loop body that directly affects instruction scheduling.

More recently, CLooG [2, 26] tries to overcome the above mentioned problem in Quillere et al.'s method. Conceptually, its algorithm can be thought of as the reverse of Kelly et al.'s method by reducing code size from a starting point of maximal overhead removal. However, we observe that CLooG's code compacting process does not follow the lexicographical order of iteration spaces. One could argue that this reordering might be beneficial to providing flexibility in overhead removal, but it also might result in incorrect code when there is a data dependence preventing such statement reordering.

CodeGen+ uses the same basic concept of Kelly et al.'s method, but its algorithms are completely reworked. Our goal is to produce high-performance code that is competitive with manually-tuned code, so efficiency of generated code is paramount. As context for this work, CodeGen+ has been used extensively as part of an *autotuning* compiler system, whereby the compiler generates a set of parameterized variants of a computation, and employs empirical techniques to find the best variant and associated set of optimization parameter values [25]. The variants represent different optimization strategies, while the optimization parameters are discrete values that govern code generation such as tile size or un-

**This research was funded in part by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.*

roll factor. Due to its mathematical foundations, the polyhedral model is a powerful underlying technology for an autotuning compiler because it can generate correct code under different parameter values and complex optimization strategies. The code resulting from this system has been shown to achieve performance that is comparable and sometimes better than manually-tuned libraries such as ATLAS, Goto, ACML, Cray Scientific Library, PETSc and CUBLAS [3, 13, 19, 21, 23], and manually-tuned code written by programmers targeting supercomputers [19, 22, 24].

Considering the optimization strategies used in hand-tuned codes such as, for example, BLAS linear algebra libraries, to achieve comparable performance polyhedral frameworks must be able to compose a large collection of transformations, such as tiling, permutation, iteration-space splitting, shifting, and unroll-and-jam. Such complex optimization strategies result in correspondingly complex iteration spaces. Parallel code introduces additional transformations such as strip mining across threads, resulting in non-convex iteration spaces, which further increases complexity. Autotuning additionally introduces non-trivial parameter combinations during the search that lead to significant clean-up code and elaborate loop bound calculations.

To summarize, the requirements for polyhedral compilers targeting modern memory hierarchies, multi-core and heterogeneous systems pose significant challenges for polyhedra scanning algorithms, particularly when the goal is to achieve very high levels of performance. We describe in this paper algorithms in CodeGen+ that play an important role in meeting this goal. The main contributions of this paper are: (1) a new loop overhead removal algorithm that can handle complex constraint relationships from Presburger arithmetic among different polyhedra, and provides precise control of trade-offs between loop overhead and code size based on actual loop nesting depth; and, (2) a new algorithm to further simplify if-statements for the code after trade-offs. Since CLooG has recently been extended to use full Presburger arithmetic by integrating with ISL [27], we will provide direct comparisons on five loop nest computations for identical iteration spaces. These experiments demonstrate that our bottom-up algorithms built with mathematical rigor provide a better control of trade-offs between loop overhead and code size and are capable of generating better quality code.

This paper is organized as follows. Section 2 introduces the polyhedral model and the underlying mathematical model. Section 3 describes our polyhedra scanning algorithms and code generation process. In Section 4, we explain the design features and code quality improvements under the perspective of polyhedral framework. Finally, Section 5 summarizes the related work, and Section 6 concludes the paper.

2. Background

We first give a brief introduction of the polyhedral model for loop transformations. Polyhedra scanning is part of this complete solution, and its various design decisions must be considered in this greater context. As a prerequisite for the polyhedral model, we assume that loop nests have affine loop bounds, and there is no control dependence for statements inside the loop nest. We also explain essential functions from the underlying mathematical system that are critical to understand the polyhedra scanning algorithms described in this paper.

2.1 Polyhedral Model

Statements inside a loop nest can be viewed as lattice points in a polyhedron. Executing the loop nest sequentially is equivalent to enumerating those lattice points in lexicographical order. This conceptualization allows the compiler to reason about loop transformations as manipulating iteration spaces of loop nests. For example, consider the following loop nest:

```
    for (i=0;i<n;i++)
      for (j=0;j<i;j++)
s0:     a[i][j]=b[i][j]
```

Statement `s0`'s iteration space is

$$\{[i,j] : 0 \leq i < n \wedge 0 \leq j < i\}.$$

Dimension i in this iteration space corresponds to outer loop `i` in the above loop nest and dimension j corresponds to inner loop `j`. A reordering of a statement's execution order is valid as long as it preserves all data dependences, i.e., the source is always executed before the sink after transformation for any data dependence. Viewing a loop nest by its iteration spaces give us a powerful abstraction to transform a loop nest without being restricted to the original loop structure. Thus, a loop transformation can be represented by a mapping function from one space to the other and their dimensionality does not need to be the same. A mapping must be invertible to guarantee that the same amount of work is done before and after transformation, i.e., a *reordering transformation*. For example, applying mapping function $\{[i,j] \rightarrow [i',j'] : i' = j \wedge j' = i\}$ to the above iteration space would result in

$$\{[i',j'] : 0 \leq j' < i' < n\},$$

which represents the iteration space of the interchanged loop nest.

Thus, after a loop nest has been transformed in the polyhedral model, the new iteration spaces (set of polyhedra) must be converted back to their syntactic format as nested loops to produce code. This code generation is performed using polyhedra scanning. For a single polyhedron, it simply scans from the first dimension to the last while generating the loop bounds and step size for each dimension. Any other constraints that cannot be enforced by loops are represented by if-statements, e.g., a guard to only execute a statement on a subset of iterations of a loop. For example, the above iteration space after loop interchange generates the following code from polyhedra scanning:

```
    for (t1=0;t1<=n-2;t1++)
      for (t2=t1+1;t2<=n-1;t2++)
s0:     a[t2][t1]=b[t2][t1]
```

Further challenges come from scanning a set of polyhedra and allowing more complex constraints such as modulo constraints to be described in iteration spaces. Different polyhedra may overlap with each other with nontrivial constraint relationships. Polyhedra scanning needs to generate concise loop bounds and optimally place additional guard conditions, taking into account trade-offs between loop overhead and code size.

2.2 Integer Linear Arithmetic

CodeGen+ depends on Omega+ (Presburger arithmetic) to manipulate a system of integer equations and inequalities. Omega+ is an updated Omega library [11] which uses enhanced Fourier-Motzkin elimination as its core algorithm [15–17]. Omega+ further extends the Omega library to better support integer modulo constraints in various functions. This enables CodeGen+ to generate efficient codes when iteration spaces do not have unit stride. In this section, we describe a few high-level functions that provide key underlying support in designing polyhedra scanning algorithms. These are *Project*, *Gist* and *Hull*.

Project To project a variable from a polyhedron, the variable is eliminated from all equations and inequalities from a linear system. For example,

$$\text{Project}(\{1 \leq y \leq x \leq 100\}, x) = \{1 \leq y \leq 100\}.$$

In some cases, Project will generate additional constraints if necessary:

$$\text{Project}(\{1 \le x \le 100 \wedge y = 2x\}, x)$$
$$= \{2 \le y \le 200 \wedge \exists\alpha(y = 2\alpha)\}.$$

Gist Gist is a unique function implemented in the Omega library. It takes two relations and the result must satisfy the following condition:

$$\text{Gist}(A, B) \wedge B = A \wedge B.$$

The function can be interpreted as this: given that we know B, what is the extra knowledge in A that has not been known in B. The following illustrates the behavior of this function.

$$\text{Gist}(\{i > 10 \wedge j > 10\}, \{j > 10\}) = \{i > 10\},$$
$$\text{Gist}(\{1 \le i \le 100\}, \{i > 10\}) = \{i \le 100\}.$$

A new enhancement in Omega+ enables Gist to reduce the strength of modulo constraints. For example,

$$\text{Gist}(\{\exists\alpha(i = 6\alpha)\}, \{\exists\alpha(i = 2\alpha)\}) = \{\exists\alpha(i = 3\alpha)\}.$$

The above result can be proved correct using the Chinese remainder theorem. Intuitively speaking, if we already know i is an even number, the extra information included in the fact that i is a multiple of 6 is that i must be a multiple of 3.

Hull Hull of a set of polyhedra returns a single polyhedron that must include all points in the original set of polyhedra. Algebraically, the result of Hull must be a conjunction of constraints. Although a convex hull is the minimal polyhedron that encloses all input polyhedra, it can be very expensive to compute in some cases. Our experience so far is that absolute tight bounds are not necessary when dealing with real application code. Instead, Hull is an approximation algorithm in finding the enclosing polyhedron. A new enhancement in Omega+ makes Hull handle stride conditions and find the lattice when input polyhedra have different modulo constraints. Below is a simple example illustrating Hull's behavior:

$$\text{Hull}(\{1 \le i, j \le 100 \wedge \exists\alpha(j = i + 4\alpha)\} \cup$$
$$\{1 \le i \le 50 \wedge 1 \le j \le 200 \wedge \exists\alpha(j = i + 6\alpha)\})$$
$$= \{1 \le i \le 100 \wedge 1 \le j \le 200 \wedge \exists\alpha(j = i + 2\alpha)\}.$$

It should be pointed out that different mathematical frameworks other than Omega+ can be used as long as the same functionality is provided. The generated code quality may differ due to different capability implemented in each framework.

3. Polyhedra Scanning and Code Generation

In this section we describe our improved polyhedra scanning algorithms, including the ones for the *loop overhead removal* with code size trade-off control and the subsequent *if-statement simplification*. These optimizations are becoming increasingly important for generating high quality code, as optimization strategies become more sophisticated in response to the growing complexity of microprocessors and many-core architectures.

To simplify the discussion, we assume that each polyhedron to be scanned is the end result of applying a mapping function to the original iteration space. The mapping function only affects the variable substitution during the code generation process, which we assume will be properly handled by the system. Also keep in mind that CodeGen+ treats every dimension in the polyhedra during scanning the same way without exception. This allows a clean and consistent strategy for loop overhead optimization and provides a predictable behavior to higher-level polyhedral transformation frameworks.

```
split: node
  set active;
  (relation, node*) (restriction, body)[];

 loop: node                  leaf: node
   set active;                 set active;
   int level;                  relation known;
   relation known;             relation guards[];
   relation restriction;
   relation bounds;
   relation guard;
   node* body;
```

Figure 1. AST node structure.

3.1 AST Structure

CodeGen+ relies on an abstract syntax tree (AST) to track the breakdown of a conjunction of unordered constraints in iteration spaces into levels of constraints, where constraints at each level can only use loop variables of this level and the levels above. Our AST is the same as [12] with only minor reinterpretation. There are three types of nodes: *split*, *loop* and *leaf*. Figure 1 shows the structures of these node types. Note the common active field which includes statements that will be executed within the node or its subtree. In addition, all the fields with an Omega+ relation type must be a single conjunct, which is guaranteed after disjoint iteration spaces are preprocessed to be split into separate ones. Below are detailed descriptions of each node type.

Split A split node has as its fields an array of restrictions and associated subtrees (called body). The order of subtrees is important as it represents the lexicographical order of the scanning result. For each *body* node, the corresponding *restriction*, as its name implies, restricts the iteration space of any node in the subtree. Unlike the other two node types, a split node does not correspond to any actual code when converting from the AST to actual code. It only helps to separate disjoint iteration spaces at the designated level.

Loop The loop node is the core part of the AST. Each loop node corresponds to one loop level. Its *known* and *restriction* fields are for bookkeeping information calculated from its ancestor nodes. Known refers to what constraints have been enforced by the code enclosing this loop, and restriction indicates the restricted iteration space for this node. They are not necessarily the same since not all constraints in the restricted iteration spaces can be enforced by loop nodes seen so far from the top down. Thus, known must be a subset of restriction, a property maintained throughout AST restructuring. Two other fields, *bounds* and *guard*, correspond to actual code that will be generated. Bounds is the condition that can be represented by a loop structure, namely lower and upper bounds for this loop variable and one stride condition for a constant step size. Guard is a conjunction of those extra constraints that cannot be represented by a loop structure as they are too complicated and must be enforced by a separate if-statement. The logical relationship here is that guard is placed outside the loop structure that enforces bounds, and so constraints in guard cannot reference this loop variable.

Leaf A leaf node represents those statements that will be executed at the iteration space restricted by all its parent split nodes. Each statement has an array *guards* field which represents the remaining constraints in the restricted iteration spaces that cannot be enforced by any parent loop nodes and require additional if-statements to enforce. Unlike the split node, there is no or-

```
initAST($level$, $active$, $restriction$)
input: $level$: current loop level;
        $active$: set of active statements;
        $restriction$: condition to restrict polyhedra;
output: root of AST

if ($level > max\_level$)
  new $leaf\_node$ with $active$;
  return $leaf\_node$;
if (single statement in $active$)
  $child$ = initAST($level + 1$, $active$, $restriction$);
  new $loop\_node$ with $level$ and $body$ = $child$;
  return $loop\_node$;
for (each statement $s \in active$)
  /* In the following statement,
        $IS_s$ is the iteration space of the transformed statement.    */
  /* Approximate is an existing Omega operation to simplify the
        iteration space representation.                               */
  $R_s$ = Approximate($restriction$ ∩ Project($IS_s$,
     $l_{(level+1)} \cdots l_{max\_level}$)); /* no existentials */
for (each statement $s \in active$)
  for (each constraint $c \in R_s$ involving loop variable $l_{level}$)
    if ($c$ splits $R_{s \in active}$ into disjoint sets $active1$ and $active2$)
      WLOG assume $l_{level}$ is smaller in the partition
      constrained by $c$ than the one constrained by $\overline{c}$
      $child1$ = initAST($level$, $active1$, $restriction \cap c$);
      $child2$ = initAST($level$, $active2$, $restriction \cap \overline{c}$);
      new $split\_node$ with ($c$, $child1$) and ($\overline{c}$, $child2$);
      return $split\_node$;
$child$ = initAST($level + 1$, $active$, $restriction$);
new $loop\_node$ with $level$ and $body$ = $child$;
return $loop\_node$;
```

Figure 2. Algorithm to initialize AST.

der requirement for statements active at this location as they all have the same lexicographical order.

3.2 Manipulating the AST

We first build an initial AST from a set of polyhedra as the basis for further optimization. As a preprocessing step, each statement's iteration space is split into a disjoint set of polyhedra. In addition, all polyhedra are extended to the same dimensionality of the maximum one, with constant values for additional dimensions. The extra dimensions are conveniently ignored when generating code, and since they are constant values no extra complexity is introduced. For completeness of discussion, Figure 2 shows the algorithm to build the initial AST, which is essentially the same as the one used by [12]. It is invoked initially with initAST(1, {all stmts}, { $\langle l_1$, *TRUE* $\rangle, \ldots, \langle l_{max\_level}$, *TRUE* $\rangle$ }), which invokes the algorithm on all statements in the loop nest, starting at the outermost loop and with no restrictions on the loop indices. The algorithm uses a simple strategy in that at any dimension, if there is an overlap between polyhedra, they are enclosed in the same loop node. Otherwise, several loop nodes with disjoint spaces to a common parent split node are created. This approach represents minimal code size from polyhedra scanning.

3.2.1 Computing Node Properties

After executing the algorithm in Figure 2, we still must initialize other fields such as bounds and guards for loop nodes and guards for leaf nodes. Figure 3 shows the algorithm to compute these node properties, starting at the root of the AST. The algorithm will also be invoked whenever the AST is updated due to a newly inserted split node. There are several key points in the algorithm that are important to overhead optimization and code generation. First, the bounds selected for a loop node are represented directly in a loop

```
$node$.recompute($parent\_active$, $known$, $restriction$)
input: $parent\_active$: set of active statements in parent node;
        $known$: known condition for this node;
        $restriction$: restriction condition for this node;
output: root of new AST

$active = active \cap parent\_active$;
$node.known = known$;
$node.restrictions = restrictions$;
if ($node$ is split node)
  for (each restriction and child pair $(r, c)$ in $node$)
    $c$ = $c$.recompute($active$, $known$, $restriction \cap r$);
elseif ($node$ is loop node)
  for (every statement $s \in active$)
    $R_s$ = Project($IS_s$, $l_{(level+1)} \cdots l_{max\_level}$) ∩
      $restriction$;
    if ($R_s == \emptyset$)
      $active$ = $active$ - $\{s\}$;
  $hull$ = Hull($R_{s \in active}$);
  let $i$ be the index for the loop at $level$
  if ($i$ is degenerate)
    $bounds$ = $i$'s equality constraint in $hull$;
    $guard$ = TRUE;
  else
    $bounds$ = $i$'s lower and upper bounds and single
      modulo constraint with unit coefficient in $hull$;
    $guard$ = Gist(Project($hull$, $i$), $known \cap bounds$);
  endif
  $body$ = $body$.recompute($active$, $known \cap bounds \cap$
    $guard$, $restriction \cap bounds \cap guard$);
  if ($body$ == NULL)
    delete $node$;
    return NULL;
  else return $node$;
elseif ($node$ is leaf node)
  for (each statement $s \in active$)
    $guards[s]$ = Gist($IS_s \cap restriction$, $known$);
    if ($guards[s]$ == FALSE)
      $active$ = $active$ - $\{s\}$;
  if ($active == \emptyset$)
    delete $node$;
    return NULL;
  else return $node$;
```

Figure 3. Algorithm to compute AST node properties.

structure. This guarantees that overhead optimization decisions described in the next algorithm are indeed based on the actual output loop nesting structure. Second, for a loop level with a single iteration (called *degenerate loops*), its guard condition is always set to true. This strategy postpones the extra constraints that cannot be enforced at this loop level into lower level (inner loop) nodes. Because there is an inherent comparison operation for each loop, by pushing the constraints inward, we avoid redundant checking by the guard at this level and an inner loop nest encountered in the subtree.

3.2.2 Loop Overhead Removal

Figure 4 shows the algorithm to lift overheads out of inner loops by duplicating the code. The level of code duplication is controlled by parameter d which is the loop nesting depth. The loop nesting depth can be calculated from the previous computed AST using a recursive function. The leaf node has a nesting depth zero. The nesting depth for other nodes is based on the maximum nesting depth, *max_depth*, among its children. For non-degenerate loop nodes, nesting depth is *max_depth+1*, and it is *max_depth* for all other node types. The liftOverhead algorithm also takes another parameter *propagate_up* as a flag to tell whether it is currently working on a node that is inside the subloop with a nesting depth $\leq d$.

Loop overheads arise from non-tautology guard conditions in loop and leaf nodes. They correspond to if-statements in a loop nest. In our algorithm, a single overhead condition selected to be lifted out of the current position corresponds to a constraint or a combination of constraints whose complement is a single conjunct. For example,

$$i \leq 5$$

can be one selected overhead condition. And

$$\exists\alpha(5\alpha \leq i \leq 5\alpha + 2)$$

can also be lifted out as a whole since its complement

$$\exists\alpha(5\alpha + 3 \leq i \leq 5\alpha + 4)$$

is a single conjunct. This requirement allows a single split node to partition the space into two disjoint subspaces each constrained by a single conjunct.

The algorithm begins at the root of the AST with *propagate_up* set to *FALSE*, and descends the AST recursively. The main part of the algorithm deals with loop nodes with a nesting depth $\leq d$, and thus it will be explained in more detail. When a suitable overhead to be optimized is located, it is propagated up the tree to the highest level possible. The highest level is the loop node such that one of the following conditions is satisfied: (1) the required nesting depth is reached; (2) for constraint(s) with existential variables, the maximum loop level of variables appearing inside is this loop level minus one; or, (3) for constraints without existential variables, the maximum loop level of variables appearing inside is this loop level. The reason for (1) is straightforward since that is the intended place to lift the overhead. Further up will cause more code duplication than required. The reason for (2) and (3) is that this overhead cannot be propagated higher since otherwise it will cause a referenced variable to be projected away.

At this point, the algorithm has identified the location in the AST for inserting a split node for the overhead condition to be lifted. A new split node is inserted at this location and the original subAST is duplicated and reinserted as two children to this newly created split node, one constrained by satisfying the overhead condition and the other by its complement. The order of these two subtrees follows the lexicographical order. Both subASTs are then recomputed for node properties since they are under different restricted spaces now. The algorithm `liftOverhead` is then recursively invoked on this new split node. This process stops when all candidate overhead conditions have been lifted.

Currently we do not treat min/max bounds in a loop nest as overhead since we have not found the run-time costs of such operations to be significant. Nevertheless, if desired, min/max bounds can be easily added into the algorithm and controlled by a different nesting depth parameter if needed. The constraint type is much simpler than those that would appear in the guard condition, and thus the implementation is straightforward.

3.2.3 If-statement Simplification

Our previous algorithm for lifting loop overhead only deals with trade-offs between loop overhead and code size. It is possible that guard conditions from loop nodes of the same level at different subASTs use the same constraint or they contradict each other. This gives us the opportunity to further reduce the control overhead without any negative impact on code size by constructing if-then-else subtrees (represented by IF nodes in the algorithm) and avoiding unnecessary conditionals. This algorithm is invoked for a set of adjacent nodes at the same nesting depth, with *postponed_guard* set to NULL, and *known_guard* representing the context in which the nodes are nested.

```
$node$.liftOverhead($d$, $propagate\_up$)
input: $d$: loop nesting depth counted from innermost;
       $propagate\_up$: flag whether inside a loop nest of
           nesting depth $\leq d$;
output: (overhead constraint, root of new AST)

if ($node$ is split node)
  for (each restriction and child pair ($r$, $c$) in $node$)
    ($r2$, $c2$) = $c$.liftOverhead($d$, $propagate\_up$);
    ($r$, $c$) = ($r$, $c2$);
    if ($r2 \neq TRUE$)
      return ($r2$, $node$);
  return ($TRUE$, $node$);
elseif ($node$ is loop node)
  if (nestingDepth($node$) $> d$)
    ($r$, $c$) = $body$.liftOverhead($d$, $FALSE$);
    $body$ = $c$;
    return ($TRUE$, $node$);
  else
    if ($propagate\_up$)
      pick one $guard$ $r$ with single conjunct complement;
      if ($r \neq TRUE$)
        return ($r$, $node$);
    if ($propagate\_up$ || $bounds$ is not an equality)
      ($r$, $c$) = $body$.liftOverhead($d$, $TRUE$);
    else
      ($r$, $c$) = $body$.liftOverhead($d$, $FALSE$);
    $body$ = $c$;
    if ($r$ == $TRUE$)
      return ($TRUE$, $node$);
    if ($bounds$ is an equality)
      substitute $l_{level}$ in $r$ using the equation;
    if (!$propagate\_up$ || ($r$ has existential variables and
      the maximum level of loop variables used is $level - 1$) ||
      ($r$ has no existential variables and the maximum level of
      loop variables used is $level$))
      WLOG assume $l_{level}$ is smaller in the partition satisfying $r$;
      new $split\_node$ with ($r$, $node$) and ($\overline{r}$, $node$);
      $split\_node$.recompute($active$, $known$, $restriction$);
      return $split\_node$.liftOverhead($d$, $propagate\_up$);
    else
      return ($r$, $node$);
elseif ($node$ is leaf node)
  for (each statement $s \in active$)
    if ($guards[s] \neq \emptyset$)
      pick one constraint $r$ from $guards[s]$ with single
      conjunct complement;
      if ($r \neq TRUE$)
        return ($r$, $node$);
  return ($TRUE$, $node$);
```

Figure 4. Algorithm to lift loop overhead.

There are two scenarios for if-statement simplification. For loop nodes, only guard conditions from neighboring nodes can be considered for merging since the generated code must follow the lexicographical order. On the other hand, for statements in a leaf node, since they all have the exact same lexicographical order, simplifying if-statements can be done in any order and this provides more flexibility in reducing the number of comparisons. Figure 5 shows the algorithm `mergeIfInOrder` to merge if-conditions in order. Merging if-conditions out-of-order can be done in a similar fashion, and it is omitted here due to space limitations.

The algorithm in Figure 5 takes the input of an array of contiguous loop nodes of the same level. It also takes two additional parameters: one tracks those constraints yet to be enforced, and one tracks constraints already generated. The basic idea of the algorithm is to test if a constraint from the guard condition in the first node matches the constraint appearing in the immediately following ones. For the remaining nodes, the algorithm tests if the guards contradict. This partitions the nodes into three contiguous parts with regard to the if-statement for this particular constraint: the first and second part are the then and else part of the if-statement respectively, and the third part follows this if-statement. Then the algorithm recursively works on each group until all constraints are processed. Note that simply comparing the constraints in isolation is not optimal in reducing the number of comparisons. These constraints have additional known conditions guarding execution of the code at this point, and there is no need to retest these conditions. Instead, the algorithm tests whether they are equivalent and complementary given the known condition using the *Gist* function. For example, as shown later in Figure 8(f), if we already know variable i is an even number from its enclosing loops, then the constraint $\exists\alpha(i = 4\alpha)$ is the exact complement of constraint $\exists\alpha(i = 4\alpha + 2)$. We can then put the statements guarded by $\exists\alpha(i = 4\alpha + 2)$ into the else part of the if-statement with condition $(mod(i, 4) == 0)$.

However, recall that in the algorithm to compute node properties (Figure 3), we postpone the computation of guard conditions for those loop nodes which are assignments (degenerate loops). This will cause a problem here in that now the guard conditions at the different loop levels in the AST might actually be translated to neighboring if-statements in the generated code if intermediate loop nodes are all degenerate. As a preprocessing step, we conceptually propagate guard conditions in the child loop nodes into upper level loop nodes if loop nodes between them are all degenerate, with variable substitutions according to those assignments. Such propagation may involve multiple levels of degenerate loop nodes or multiple split nodes if necessary. Figure 6 illustrates this behavior using one AST snippet as an example. Guard conditions in nodes C, D, E and F are translated to neighboring if-statements in the actual code. Thus in our preprocessing step, the guard condition in node C is propagated to node A, and the guard condition in node F is propagated to node E. Finally, node A, D, E and G with updated guard conditions are the first parameters to the algorithm in Figure 5.

```
mergeIfInOrder($nodes[]$, $postponed\_guard$, $known\_guard$)
input: $nodes[]$: neighboring nodes for their guard conditions;
       $postponed\_guard$: guard conditions yet to be enforced;
       $known\_guard$: guard conditions already considered;
output: if-then-else tree

/* for convenience of illustration, assume parameter $nodes$ always
   indexed from 1 to $n$ */
if ($nodes == \emptyset$)
  return $NULL$;
$r$ = Gist($nodes[1].guard$, $known\_guard$);
if (($nodes[1].known \cap known\_guard$) $\subseteq r$)
  for ($i$ in 2..$n$)
    if (Gist($nodes[i].guard$, $known\_guard$) $\neq$ TRUE)
      break;
  $s1$ = code for $nodes[1..(i-1)]$ without their guard condition;
  $s2$ = mergeIfInOrder($nodes[i..n]$, NULL, $known\_guard$);
  $s3$ = concatenate $s1$ and $s2$ in this order;
  if ($postponed\_guard \neq \emptyset$)
    return new IF($postponed\_guard$, $s3$, NULL);
  else
    return $s3$;
else
  $c$ = select a constraint from $nodes[1].guard$ that maximizes
     the contiguous region of nodes starting from 2 to satisfy it;
  Let $c$ partition $nodes$ into three contiguous regions $nodes1$,
  $nodes2$ and $nodes3$ in this order where $nodes1$ satisfies $c$,
  $nodes2$ satisfies $\bar{c}$ and $nodes3$ remaining nodes in the end;
  if ($nodes2 == \emptyset \wedge nodes3 == \emptyset$)
    return mergeIfInOrder($nodes1$, $postponed\_guard \cap c$,
       $known\_guard \cap c$);
  elseif ($nodes2 == \emptyset$)
    $s1$ = mergeIfInOrder($nodes1$, $c$, $known\_guard \cap c$);
    $s2$ = mergeIfInOrder($nodes3$, NULL, $known\_guard$);
    $s3$ = concatenate $s1$ and $s2$ in this order;
    if ($postponed\_guard \neq \emptyset$)
      return new IF($postponed\_guard$, $s3$, NULL);
    else
      return $s3$;
  else
    $s1$ = mergeIfInOrder($nodes1$, NULL, $known\_guard \cap c$);
    $s2$ = mergeIfInOrder($nodes2$, NULL, $known\_guard \cap \bar{c}$);
    $s3$ = mergeIfInOrder($nodes3$, NULL, $known\_guard$);
    $s4$ = new IF($c$, $s1$, $s2$);
    $s5$ = concatenate $s4$ and $s3$ in this order;
    if ($postponed\_guard \neq \emptyset$)
      return new IF($postponed\_guard$, $s5$, NULL);
    else
      return $s5$;
```

Figure 5. Algorithm to merge neighboring if-conditions in order.

3.3 Code Generation

Once the AST is constructed and subsequently optimized, it corresponds to a compiler's high-level intermediate representation (IR) AST if all split nodes are ignored. Thus code generation is a straightforward process that follows from scanning the AST. Some details of converting constraints to code are presented here.

For a loop node with unit stride, lower and upper bounds come from inequalities in bounds conditions, with min/max added if there are multiple such inequalities. For non-unit stride, CodeGen+ must know whether the lower bounds always satisfy the stride condition. Suppose K is the intersection of the known condition and bounds condition for this loop node. First, we create a relation with a modulo constraint for this loop variable with the same stride but starting at a lower bound and test whether there is new knowledge in this relation given K. If not, we further test whether there is new knowledge in the stride condition when this loop variable is set to the lower bound in K. Both tests are done by the Gist function and if both results are a tautology, we can safely use this lower bound; otherwise, we generate a remainder expression to be added to the lower bound.

For degenerate loop nodes (single iteration loops), we either generate assignment code directly or substitute every appearance of this loop variable in its child AST with a substitution expression. The choice depends on how complex the expression is, and whether it can be adjusted by a parameter. Additional consideration is needed when the coefficient c for this loop variable is not a unit; that is, when $|c| \neq 1$. We must test if the right hand side of the assignment without dividing by the c part is always a multiple of c using the Gist function given the known condition when code reaches here. If not, an if-statement with condition $(\text{mod}(\text{RHS}, |c|) == 0)$ must be created to guard the assignment and the code from the child AST, where RHS is the right hand side of the assignment without dividing by the c part.

Here are a few examples where constraints in the guard conditions have existential variables:

$$\exists\alpha(4i = 5j + 4\alpha) \rightarrow mod(4*i - 5*j), 4) == 0,$$
$$\exists\alpha(4\alpha \leq i \leq 5\alpha) \rightarrow ceil(i, 5) <= floor(i, 4).$$

Our implementation also tries to recognize floor definitions from constraints and generate clean code whenever possible. For exam-

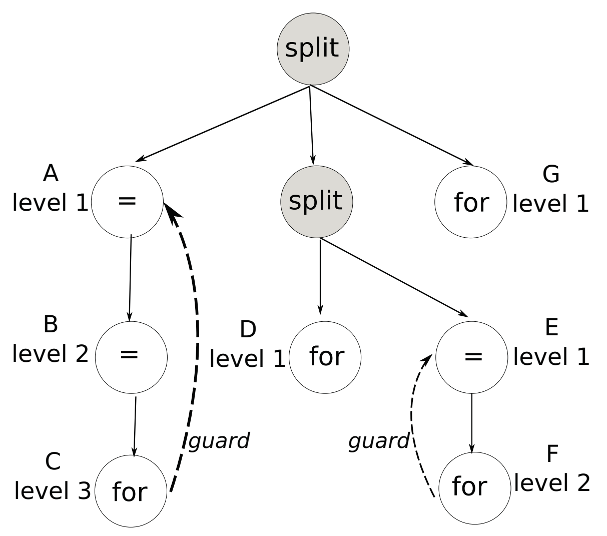

Figure 6. Propagate guard conditions up through degenerate loop nodes (marked by "=").

ple, with α defined by $\lfloor \frac{m}{4} \rfloor$ in the following relation,

$$\{[i] : \exists\alpha(m - 4 < 4\alpha <= m \wedge 4\alpha <= i <= n)\},$$

the generated lower bound for i is $4 * floor(m, 4)$. Thus there is no if-statement needed for the generated loop code from this polyhedron.

4. Experiments

With precise control of trade-offs between loop overhead and code size and reduction of if-conditions, while at the same time preserving the lexicographical order among statements represented in the input iteration spaces, CodeGen+ provides a clean interface and predictable behavior for high-level polyhedral loop transformation frameworks. In this section, we examine the quality of code generated by CodeGen+ after composing a sequence of transformations in a polyhedral framework for a set of loop nest computations. We will compare CodeGen+ with the state-of-the-art polyhedra scanning tool CLooG using examples to illustrate the improvements resulting from our new algorithms.

For the generated codes in this section, we use CLooG 0.16.3 and CodeGen+ 2.2.3. For code compilation and performance, we use gcc 4.6.1 with -O3 flag. For performance measurements, the target architecture is a single core of a 2.8GHz Intel Core i7 930 with 2GBytes memory. We first consider a set of simple code examples shown in Figures 7 and 8 to illustrate the differences resulting from the algorithms in the previous section. Subsequently, we examine a set of kernel computations, shown in Table 1. The code inputs are standard hand written kernels (e.g., a simple 3-deep loop nest for `gemm`) to which transformations are applied using the CHiLL transformation and code generation system [4]. While CHiLL can invoke CodeGen+ internally, we instead capture the iteration spaces of the transformed statements. Then, identical iteration spaces are input to both Codegen+ and to CLooG (via the iscc frontend to ISL in Barvinok) [27]. The output code from both are compared to produce the results in the table.

4.1 Control of Trade-offs

Precise control of trade-offs between loop overhead and code size is important in tailoring the output of polyhedra scanning. For example, in optimizing for high performance, code generation of an innermost loop impacts instruction-level parallelism or SIMD execution in multimedia extensions, which are critical for overall performance. Thus removing loop overhead from innermost loops is usually the best choice in balancing loop overhead and code size. For other situations, a different choice might be selected. As examples, we might not want to lift any overhead to avoid code growth in an embedded system, or we might want to remove as much control flow as possible from loops for the highest performance.

Figure 7(b-d) shows the three variations of generated code from iteration spaces in (a) with different trade-offs of removing loop overhead from subloops of depth 0, 1 and 2, respectively. Note that statement `s0` is enclosed in loop `t1` which itself is a loop nest of depth 2. Thus when removing loop overhead from nesting depth 0 and 1, its enclosing if-condition `(n>=2)` is not moved out of the `t1` loop. Only when removing loop overhead from all subloops of depth 2 is it moved out of the `t1` loop; no more if-conditions remain inside any loop in the generated code (Figure 7(d)). CLooG generates almost identical code to Figure 7(d), which follows the lexicographical order. However, it does not provide such a guarantee when generating codes for other trade-off points using the -f or -l flags, which move conditions based on the first or last loop's nesting depth, respectively.

4.2 If-condition Overhead

The algorithms presented in this paper can reduce the number of unnecessary if-conditions in the generated codes, especially in some difficult situations. Figure 8 shows two examples to compare number of if-conditions in the codes generated from CLooG and CodeGen+. For the iteration space in Figure 8(a), the output of CLooG in Figure 8(b) contains redundant modulo checking in the innermost loop, but this is removed by CodeGen+ as shown in Figure 8(c). Consider the iteration space of Figure 8(d) and the output of CLooG in Figure 8(e). Given that $c1$ is even, based on loop bounds, the condition $((c1 + 2)\%4 == 0)$ can be determined statically to be the complement of condition $(c1\%4 == 0)$. Thus, the mod operation occurs just once in the CodeGen+ code of Figure 8(f), and the outermost condition on variable `n` is not generated since the first loop encountered will check the same condition again as part of its standard bounds checking.

4.3 Application to Code Optimization

We now demonstrate that the improvements in the polyhedra scanning algorithms indeed make a significant impact on generating high-quality codes when complex optimization strategies are applied in polyhedral frameworks. Table 1 shows our comparison result with CLooG using the same iteration spaces generated from CHiLL [4] for three programs generated from scripts in CHiLL's example directory, matrix-vector multiply (`gemv`), matrix-matrix multiply (`gemm`) and LU factorization (`lu`), and two codes `qr` and `swim` taken from CLooG [26]. These examples illustrate the effect of complexity of iteration spaces on generated code. The kernel `gemv` is the simplest, followed by `qr`, `swim` and `gemm`. Iteration spaces for `lu` are by far the most complex.

For `gemv`'s simple unroll-and-jam optimization, both CLooG and CodeGen+ take roughly the same amount of time generating the codes, and the resulting codes have the same performance although the code generated by CLooG is somewhat larger because it generates extra if-conditions corresponding to modulo constraints. The optimization strategy for `qr` involves peeling, shifting and fusion; the resulting code is larger using CLooG which increases code generation and compile time, but performance is comparable for the two systems. For `swim`, the optimization strategy is very close to that of [8], employing peeling, shifting, and fusion. While the transformations are similar to what was used in `qr`, the iteration space complexity grows because there are three loop nests that all must be peeled and shifted different amounts to enable fusion. The

$s0: \{[i] : 1 \le i \le 100 \wedge n > 1\};$
$s1: \{[i,j] : 1 \le i,j \le 100 \wedge n > 1\};$
$s2: \{[i,j] : 1 \le i,j \le 100\};$
(a) iteration spaces

```
for (t1=1;t1<=100;t1++)
  if (n>=2)
    s0(t1);
  for (t2=1;t2<=100;t2++)
    s2(t1,t2);
    if (n>=2)
      s1(t1,t2);
```
(b) no loop overhead removal (depth 0)

```
for (t1=1;t1<=100;t1++)
  if (n>=2)
    s0(t1);
    for (t2=1;t2<=100;t2++)
      s1(t1,t2);
      s2(t1,t2);
  else
    for (t2=1;t2<=100;t2++)
      s2(t1,t2);
```
(c) remove overhead from depth 1 loops (default)

```
if (n>=2)
  for(t1=1;t1<=100;t1++)
    s0(t1);
    for (t2=1;t2<=100;t2++)
      s1(t1,t2);
      s2(t1,t2);
else
  for (t1=1;t1<=100;t1++)
    for (t2=1;t2<=100;t2++)
      s2(t1,t2);
```
(d) remove overhead from depth 2 loops

Figure 7. Different trade-offs between loop overhead and code size in CodeGen+.

$s0: \{[i,j] : 1 \le i \le n \wedge i \le j \le n \wedge \exists \alpha, \beta(i = 1 + 4\alpha \wedge j = i + 3\beta)\};$
(a) single iteration space with complex stride conditions

```
if (n>=1)
  for (i=1;i<=n;i+=4)
    for (j=i;j<=n;j+=3)
      if ((11*i+4*j+9)%12 == 0)
        s0(i,j);
```
(b) CLooG output from (a)

```
for (t1=1;t1<=n;t1+=4)
  for (t2=t1;t2<=n;t2+=3)
    s0(t1,t2);
```
(c) CodeGen+ output from (a)

$s0: \{[i] : 1 \le i \le n \wedge \exists \alpha(i = 4\alpha)\};\ \ s1: \{[i] : 1 \le i \le n \wedge \exists \alpha(i = 4\alpha + 2)\};$
(d) two iteration spaces with complex relationship with each other

```
if (n>=2)
  for (c1=2;c1<=n;c1+=2)
    if (c1%4==0)
      s0(c1);
    if ((c1+2)%4==0)
      s1(c1);
```
(e) CLooG output from (d)

```
for (t1=2;t1<=n;t1+=2)
  if (intMod(t1,4)==0)
    s0(t1);
  else
    s1(t1);
```
(f) CodeGen+ output from (d)

Figure 8. Comparison of number of if-conditions in the generated codes.

code generated by CLooG is more than four times larger than that generated by CodeGen+, and code generation time and compile time are consequently increased. `Swim` achieves CodeGen+'s highest execution-time speedup over CLooG of 1.15x. To consider a very different optimization strategy, both `gemm` and `lu` require multiple levels of tiling and result in many statements generated for the final set of iteration spaces. In addition, `lu` requires extensive splitting of the iteration spaces, for example to separate the computation into a mini-LU, triangular solve and matrix-matrix multiply as in highly-tuned implementations [10]. CodeGen+ generates significantly less lines of code for `gemm` and `lu` as compared with CLooG, a reduction of over 8x and 24x, respectively, while the resulting code performs 1.12x faster than the CLooG output for both. Moreover, code generation time and compilation time have up to an order of magnitude speedup; for example, 25x and 34x reductions, respectively, for `lu`.

Overall, the combined effect of lifting control overhead from innermost loops and if simplification in CodeGen+ results in simpler code with less branching than that generated by CLooG for identical iteration spaces. This simpler code takes less time to generate and compile. For the more complex examples, where tiling or unrolling is used resulting in modulo constraints, or where peeling and shifting causes additional control flow, the code generated by CodeGen+ also performs better, up to 1.15x faster. This performance difference is very significant in the high-performance computing community, where programmers are willing to expend significant extra effort to increase utilization of precious supercomputing resources. As future architectures become more complex, we anticipate the compiler transformation strategies must also increase in complexity, so that this gap in generated code quality is likely to grow.

5. Related Work

Early work on polyhedra scanning focuses on how to generate code for a single polyhedron. Ancourt and Irigoin [1] use Fourier-Motzkin elimination to find loop bounds for unimodular transformations. Their method is improved later by Le Fur; however only transformed iteration spaces without holes are considered [7]. Other research [6, 14, 20, 30] expands the mapping function to allow non-unimodular transformations. Hermite Normal Form is used to find loop strides. Besides using Fourier-Motzkin elimination to extract loop bounds from a system of linear inequalities, different mathematical solutions are pursued. Collard et al. [5] use a dual simplex method. Chernikova's algorithm is another geometric approach to simplify a set of linear inequalities. It is used by Quillere et al. [18] which will be mentioned later in this section.

Kelly et al. [12] are among the first to address the problem of code generation for a set of polyhedra. Moreover, their system is built on a systematic approach to integer linear systems, implemented as part of the Omega library [15–17]. However, loop overhead trade-off decisions are based on static loop levels, without taking into account that modern polyhedral frameworks often rely on additional auxiliary dimensions, some of which are constants, to manage complicated optimization strategies. Moreover, flaws in its code generation logic will cause suboptimal or incorrect code to be generated in some difficult cases.

Griebl et al. [9] also present their code generation solution for multiple polyhedra, which allows non-unimodular transformations.

	lines of generated code			code generation time			gcc compile time			code performance		
	CLooG	CodeGen+	Reduction	CLooG	CodeGen+	Speedup	CLooG	CodeGen+	Speedup	CLooG	CodeGen+	Speedup
gemv	39	24	1.70x	0.151s	0.132s	1.14x	0.095s	0.071s	1.34x	0.091s	0.091s	1.00x
qr	88	35	2.51x	0.316s	0.104s	3.04x	0.115s	0.057s	2.02x	4.255s	4.254s	1.00x
swim	501	107	4.68x	0.622s	0.251s	2.49x	2.767s	0.590s	4.69x	0.599s	0.522s	1.15x
gemm	373	42	8.88x	6.552s	1.082s	6.06x	0.364s	0.086s	4.23x	136.667s	121.736s	1.12x
lu	2635	106	24.86x	106.984s	4.170s	25.66x	7.360s	0.211s	34.88x	49.973s	44.551s	1.12x

Table 1. Comparison of code generation performance using iteration spaces representing real optimization strategies.

Their approach can generate two flavors of code from polyhedra scanning: a run-time solution and a compile-time solution, which loosely correspond to minimal code size and minimal overhead versions, respectively. Their method in reducing overhead would result in $N!$ possibilities in loop bounds for N parameters, which can be handled by Omega code generation [12] with proper min/max bounds with significantly fewer loops. One innovation of Griebl et al.'s approach is that it allows non-invertible mapping by expanding the matrix to an invertible one, filling in unit vectors, during polyhedra scanning. However, this may not be necessary if higher-level tools can select missing dimensions using better knowledge of intended transformations.

Quillere et al. [18] propose that by splitting overlapping iteration spaces at any dimension during the process of scanning polyhedra, they can generate minimum loop overhead code without the iterative method used by [12]. However, this method can easily cause code explosion and provides no control of trade-offs between loop overhead and code size. CLooG [2, 26] uses the same approach of Quillere et al., but it tries to overcome its code explosion limitation. After initial processing, the algorithm will try to reduce the number of polyhedra previously split. However, CLooG does not provide a guarantee of generating code respecting the lexicographical order in iteration spaces during its trade-offs other than the default choice.

6. Conclusion

This paper describes a polyhedra scanning system that can meet the challenge of supporting sophisticated polyhedral frameworks combined with autotuning to generate high-performance code. The algorithms are based on a strict mathematical foundation and are designed to handle complex constraint relationships among a set of polyhedra, with precise control of trade-offs between loop overhead and code size. We compare with the state-of-the-art polyhedra scanning tool CLooG on five loop nest computations, demonstrating that CodeGen+ generates code that is simpler and up to 1.15x faster. We believe that such a system is critical to expand the applicability of polyhedral frameworks and enable compilers to close or significantly narrow the performance gap between compiler-optimized loop nests and hand-tuned code, including domain-specific performance libraries such as BLAS.

Editorial Note. Author Chun Chen passed away shortly after submission of this manuscript. Anand Venkat, Protonu Basu and Mary Hall made minor editing changes to the document to respond to reviewer requests and improve readability, and most significantly, added experimental results for two additional benchmarks, `qr` and `swim`. Questions about the contents of the paper can be directed to {anandv,protonu,mhall}@cs.utah.edu.

Acknowledgments

This research was funded in part by DARPA contract HR0011-10-9-0008, National Science Foundation award CCF-1018881, and Department of Energy Office of Science awards DE-SC0003777 and DE-SC0003520.

References

[1] Corinne Ancourt and François Irigoin. Scanning polyhedra with DO loops. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, April 1991.

[2] Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, October 2004.

[3] C. Chen, J. Shin, S. Kintali, J. Chame, and M. Hall. Model-guided empirical optimization for multimedia extension architecture s: A case study. In *Proceedings of the Workshop on Performance Optimization for High-Level Languages and Libraries (POHLL 2007)*, May 2007.

[4] Chun Chen, Jacqueline Chame, and Mary W. Hall. CHiLL: A framework for composing high-level loop transformations. Technical Report 08-897, University of Southern California, Jun 2008.

[5] Jean-François Collard, Tanguy Risset, and Paul Feautrier. Construction of DO loops from systems of affine constraints. *Parallel Processing Letters*, 5(3):421–436, 1995.

[6] Agustín Fernández, José M. Llabería, and Miguel Valero-García. Loop transformation using nonumimodular matrices. *IEEE Transactions on Parallel and Distributed Systems*, 6(8):832–840, August 1995.

[7] Marc Le Fur. Scanning parameterized polyhedron using Fourier-Motzkin elimination. *Concurrency: Practice and Experience*, 8(6):445–460, 1996.

[8] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming*, 34(3):261–317, June 2006.

[9] Martin Griebl, Christian Lengauer, and Sabine Wetzel. Code generation in the polytope model. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, October 1998.

[10] Mary Hall, Jacqueline Chame, Jaewook Shin, Chun Chen, Gabe Rudy, and Malik Murtaza Khan. Loop transformation recipes for code generation and auto-tuning. In *LCPC*, October, 2009.

[11] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott. The Omega Library interface guide. Technical Report CS-TR-3445, University of Maryland at College Park, March 1995.

[12] Wayne Kelly, William Pugh, and Evan Rosser. Code generation for multiple mappings. In *Proceedings of the 5th Symposium on the Frontiers of Massively Parallel Computation*, February 1995.

[13] Malik Khan. *Autotuning, code generation and optimizing compiler technology for GPUs*. PhD thesis, University of Southern California, May 2012.

[14] Wei Li and Keshav Pingali. A singular loop transformation framework based on non-singular matrices. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, August 1992.

[15] William Pugh. The Omega test: A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

[16] William Pugh and David Wonnacott. An exact method for analysis of value-based array data dependences. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, August 1993.

[17] William Pugh and David Wonnacott. Experiences with constraint-based array dependence analysis. In *Proceedings of the Second International Workshop on Principles and Practice of Constraint Programming*, May 1994.

[18] Fabien Quilleré, Sanjay Rajopadhye, and Doran Wilde. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28(5):469–498, October 2000.

[19] Shreyas Ramalingam, Mary Hall, and Chun Chen. Improving high-performance sparse libraries using compiler-assisted specialization : A PETSc case study. In *Proceedings of the Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2012)*, May 2012.

[20] J. Ramanujam. Beyond unimodular transformations. *The Journal of Supercomputing*, 9(4):365–389, February 1995.

[21] Gabe Rudy, Malik Murtaza Khan, Mary Hall, Chun Chen, and Cha Jacqueline. A programming language interface to describe transformations and code generation. In *Proceedings of the 23rd international conference on Languages and compilers for parallel computing*, pages 136–150. Springer-Verlag, 2011.

[22] Jaewook Shin, Mary W. Hall, Jacqueline Chame, Chun Chen, Paul F. Fischer, and Paul D. Hovland. Speeding up nek5000 with autotuning and specialization. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 253–262, 2010.

[23] Jaewook Shin, Mary W. Hall, Jacqueline Chame, Chun Chen, and Paul D. Hovland. Transformation recipes for code generation and auto-tuning. In *The Fourth International Workshop on Automatic Performance Tuning*, October 2009.

[24] A. Tiwari, J. K. Hollingsworth, C. Chen, M. Hall, C. Liao, D. J. Quinlan, and J. Chame. Auto-tuning full applications: A case study. *International Journal of High Performance Computing Applications*, pages 286–294, August 2011.

[25] Ananta Tiwari, Chun Chen, Jacqueline Chame, Mary Hall, and Jeffrey K. Hollingsworth. A scalable autotuning framework for compiler optimization. In *IPDPS*, Rome, Italy, May 2009.

[26] Nicolas Vasilache, Cédric Bastoul, and Albert Cohen. Polyhedral code generation in the real world. In *Proceedings of the International Conference on Compiler Construction*, March 2006.

[27] Sven Verdoolaege. isl: An integer set library for the polyhedral model. In *International Congress on Mathematical Software*, September 2010.

[28] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1991.

[29] Michael E. Wolf and Monica S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.

[30] Jingling Xue. Automating non-unimodular loop transformations for massive parallelism. *Parallel Computing*, 20(5):711–728, May 1994.