UMIACS-TR-93-67 December, 1993
CS-TR-3108

Determining Schedules based on Performance Estimation

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

Univ. of Maryland, College Park, MD 20742

Abstract

In previous work, we presented a framework for unifying iteration reordering transformations such as loop interchange, loop distribution, loop skewing and statement reordering. The framework provides a uniform way to represent and reason about transformations. However, it does not provide a way to decide which transformation(s) should be applied to a given program. This paper describes a way to make such decisions within the context of the framework. The framework is based on the idea that a transformation can be represented as an affine mapping from the original iteration space to a new iteration space. We show how we can estimate the performance of a program by considering only the mapping from which it was produced. We also show how to produce an lower bound on performance given only a partially specified mapping. Our ability to estimate performance directly from mappings and to do so even for partially specified mappings allows us to efficiently find mappings which will produce good code.

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

Selecting Affine Mappings based on Performance Estimation

Wayne Kelly William Pugh

wak@cs.umd.edu, (301)-405-2726 pugh@cs.umd.edu, (301)-405-2705

Department of Computer Science

University of Maryland, College Park, MD 20742

December 9, 1993

Abstract

In [KP93] we presented a framework for unifying iteration reordering transformations such as loop interchange, loop distribution, loop skewing and statement reordering. The framework provides a uniform way to represent and reason about transformations. However, it does not provide a way to decide which transformation(s) should be applied to a given program. This paper describes a way to make such decisions within the context of the framework. The framework is based on the idea that a transformation can be represented as an affine mapping from the original iteration space to a new iteration space. We show how we can estimate the performance of a program by considering only the mapping from which it was produced. We also show how to produce an lower bound on performance given only a partially specified mapping. Our ability to estimate performance directly from mappings and to do so even for partially specified mappings allows us to efficiently find mappings which will produce good code.

1 Introduction

Traditionally, optimizing compilers attempt to improve the performance of programs by applying source to source transformations, such as loop interchange, loop skewing and loop distribution [Wol89]. Each of these transformations has its own special legality checks and transformation rules. These checks and rules make it hard to analyze or predict the effects of compositions of these transformations, without actually performing the transformations and analyzing the resulting code.

To overcome these problems, many researchers have proposed a uniform approach to transforming programs [Ban90, WL91, LMQ91, Fea92, ST92, KP93, ADY92]. The problem of finding the optimal program equivalent to some other program is undecidable in general, since it reduces to the "program equivalence problem"[Tou84]. Consequently existing systems make various simplifying assumptions. Typically they accept only a limited class of source programs, consider applying only a limited set of transformations, and consider only a small number of performance issues.

By representing data dependences as dependence relations we can accurately handle programs with non-uniform dependences. We represent transformations by associating a *multi-dimensional mapping* with each statement. This allows us to represent a large class of transformations, including any combination of loop interchange, loop distribution, skewing, tiling, index set splitting and statement reordering.

Most schedule based systems [LMQ91, Fea92, ADY92] determine which schedule to apply by using linear programming techniques to optimize a linear function representing execution time. Most of these systems assume that parallelism is the only factor that will affect execution time, i.e. they ignore issues such as simplicity and data locality. We believe that creating a linear performance function which takes into account all factors which affect execution time would be prohibitively expensive (if in fact it is even possible).

It may not be possible to represent all performance factors as linear functions, however it is possible to represent them as computable functions. For each performance factor of interest, we have written a program, which given a mapping, produces an estimate of the effect of that factor on execution time. By considering the estimates for each factor and by taking into account the degrees to which these factors affect execution time on a given architecture, we can get a realistic measure of performance. The performance estimators can also take as input, partially specified mappings. The performance estimators were carefully designed so as to satisfy the *monotone property* [Nil80]. A performance estimator P satisfies the monotone property iff:

$$\forall S, S' \quad S \mapsto S' \Rightarrow P(S) \leq P(S')$$

where $S \mapsto S'$ means that mapping S' can be obtained by further specifying the partially specified mapping S. $P(S) \leq P(S')$ means that mapping S is "better" than mapping S', i.e. low performance estimates correspond to low expected execution times.

The performance estimators form the basis of our technique for deciding which transformation to apply. Conceptually we can view the decision process as traversing a search tree. The root of the tree corresponds to the completely unspecified mapping, the leaves in the tree correspond to completely specified mappings and the interior nodes correspond to partially specified mappings. Nodes closer to the leaves correspond to mappings that are more fully specified that those higher in the tree. We use heuristics to limit the branching factor at each node, and use methods described in [KP93] to restrict our search to nodes which correspond to legal mappings.

The monotone property of the performance estimators allows us to use an admissible heuristic search algorithm [Nil80] to find the optimal node(s) in the search tree. Admissibility means that we are guaranteed to find the optimal node(s) in the tree despite the fact that we may not examine all of the nodes. The performance estimates which we obtain for partially specified mappings are used only to eliminate provably bad mappings, they are never used to definitively select potentially good mappings. Therefore it is important that the estimates we obtain for partially specified schedules be lower bounds on the estimates for all extensions of these mappings. This is precisely what the monotone property allows us to do.

In Section 2 we summarize our earlier work [KP93] on developing a framework for unifying reordering transformations. In Section 3 we describe our algorithm for selecting mappings based on performance estimation. In Section 4 we define some terms and concepts used by our performance estimators and in Section 5 we describe the performance estimators we have implemented. In Section 6 we present results based on our implementation and in Section 7 we state our conclusions.

2 The Framework

In [KP93] we presented a framework for unifying iteration reordering transformations. The framework is designed to provide a uniform way to represent and reason about transformations.

2.1 Iteration spaces

A program implicitly specifies an iteration space. An iteration space can be represented using sets of tuples, where each tuple contains the values of the loop variables for a particular execution of a particular statement. For example the program in Figure 1 implicitly specifies the iteration space shown to its right.

2.2 Affine Mappings

The mappings that we use are 1-1 functions. They map each tuple (or iteration) in the original iteration space to a tuple in a new iteration space. For example the mapping in Figure 2 maps iteration $[5, 7]$ in the original iteration space of statement 20 to iteration $[1, 7, 0, 5]$ in a new iteration space. In previous papers, we have referred to these affine mappings as schedules. We have decided to change our terminology since the term "schedule" is used in a

```
      do 30 i = 1, n
10      s(i) = 0
        do 20 j = 1, i-1
20        s(i) = s(i) + a(j,i)*b(j)
30      b(i) = b(i) - s(i)
```

$$\begin{array}{lllll} I_{10}: & \{ & [\,i\,] & \mid 1 \le i \le n \,\} \\ I_{20}: & \{ & [\,i,\ j\,] & \mid 1 \le i \le n \wedge 1 \le j \le i-1 \,\} \\ I_{30}: & \{ & [\,i\,] & \mid 1 \le i \le n \,\} \end{array}$$

Figure 1: Program and associated iteration space

slightly different way by the systolic array community. In their framework, iterations will be executed in parallel iff they are scheduled at the same "time". In our framework, all iterations are mapped to distinct points. However, this doesn't imply that they will be executed sequentially. The decision to run a loop in parallel is a separate decision, which is legal if the loop does not carry any dependences.

The transformed code corresponding to a mapping will in general consist of imperfectly nested loops which execute the iterations in the new iteration space in lexicographic order. For example the program in Figure 1 can be transformed using the mapping in Figure 2 into the program in Figure 2.

$$\begin{array}{llllll} T_{10}: & \{[\,i\,] & \rightarrow [0, & i, & 0, & 0]\,\} \\ T_{20}: & \{[\,i,\ j\,] & \rightarrow [1, & j, & 0, & i]\,\} \\ T_{30}: & \{[\,i\,] & \rightarrow [1, & i-1, & 1, & 0]\,\} \end{array}$$

```
      parallel do 10 i = 1, n
10      s(i) = 0
      do 30 t = 1, n-1
        parallel do 20 i = t+1, n
20        s(i) = s(i) + a(t,i)*b(t)
30      b(t+1) = b(t+1) - s(t+1)
```

Figure 2: Transformed program and associated mapping

2.3 Notation used for mappings

We specify a transformation by specifying a separate mapping for each statement. The mapping associated with statement s_p has the form:

$$T_p : [i_{p1}, \ldots, i_{pm_p}] \rightarrow [f_{p1}, \ldots, f_{pn}]$$

where:

- The iteration variables $i_{p1}, \ldots, i_{pm_p}$ represent the loops nested around the statement s_p.
- The $f_{pj}{}'s$ are affine functions of the iteration variables and symbolic constants.

This mapping represents the fact that iteration $[i_{p1}, \ldots, i_{pm_p}]$ in the original iteration space of statement s_p is mapped to iteration $[f_{p1}, \ldots, f_{pn}]$ in the new iteration space. The f_{pi} expressions are called *mapping components*. For simplicity, but without loss of generality, we require all mappings to have n components. We refer to each of the positions $1, \ldots, n$ as *levels*. A level can also be thought of as the set of mapping components at a particular position. We will use the term mapping to refer to both the mappings of individual statements and the set of mappings associated with all statements.

2.4 Variable and constant parts

We distinguish two parts of a mapping component: the *variable part* and the *constant part*. The variable part is the largest subexpression of the mapping component that is a linear function of the iteration variables. The rest of the expression is called the constant part. For example in the mapping

$$[i, j] \rightarrow [2i + j + n + 1, 0, j]$$

the mapping component at level 1 has variable part $2i + j$ and constant part $n + 1$.

The variable parts of a mapping are of primary importance in determining the parallelism and data locality of the code resulting from a mapping. The framework is designed to leave the important decisions (which variable parts should be used) to the system using the framework, and make the relatively unimportant decisions (which constant parts to use) itself.

2.5 Algorithms provided with the framework

As part of the framework, we provide algorithms to assist in the building of mappings. These algorithms assume that mappings are constructed level by level starting at the first level, (i.e., during step k, $f_{1k}, \ldots, f_{pk}$ are specified).

Given the first $k-1$ levels of a mapping, the *Component legality test* determines whether a given variable part v_{pk} is "legal" at level k of statement s_p. Such a variable part is "legal" if using it as the level k mapping component of statement s_p would not violate any self-dependences[1] on s_p (including any transitive self-dependences). For example in the program in Figure 1, i and j are both legal variable parts for statement 20 at level 1, but $-i$ and $-j$ are illegal. Using legal variable parts is necessary but not sufficient to produce a legal mapping.

Given the first $k-1$ levels of a mapping and a legal variable part v_{pk} for level k of each statement s_p, the *Alignment algorithm* determines a set of constant parts $\{c_1, \ldots, c_m\}$. These c_p's are chosen such that using $v_{pk} + c_p$ as the level k mapping component for statement s_p produces a legal mapping. Mappings obtained from the alignment algorithm are guaranteed to be legal, but they are not guaranteed to have good performance. An algorithm to generate the transformed code corresponding to a mapping is also provided with the framework.

3 The Mapping Selection Algorithm

The framework described in the previous section is meant to be used within some larger system that is responsible for deciding which transformation should be applied. This section overviews our implementation of such a system which uses performance estimation to guide the selection of mappings. Figure 3 gives an outline of our algorithm. This algorithm is closely related to the A^* heuristic search algorithm [Nil80]. We assume that the reader is familiar

```
procedure ExtendMapping(PartialMapping, Level)
    for each stmt
        LegalVariableParts[stmt] = {Profitable legal variable parts}(stmt, Level)
    for each Combination of LegalVariableParts (1 from each stmt)
        Mapping = extend PartialMapping with mapping components
                        from AlignmentAlgorithm(Combination)
        if Mapping is 1-1 then
            if {WorthAccepting} then
                {Modify set of tentatively accepted mappings}
        else
            if {WorthContinuing} then
                ExtendMapping(Mapping, Level+1)

Start by calling: ExtendMapping(UnspecifiedMapping, 1)
```

Figure 3: An algorithm to build mappings level by level

with the basic idea of heuristic search and only discuss the parts of this algorithm (enclosed in curly braces) that are specific to our application. It is important to note however, that in general, this algorithm does involve backtracking.

[1] A self-dependence is a dependence from one iteration of a statement to another iteration of the same statement.

Profitable legal variable parts (p, k) First we use a heuristic to reduce the set of variable parts considered to a finite size. The use of a heuristic at this stage means that we cannot prove that we will find the optimal mapping since it is possible that the optimal variable part is not considered. In practice, we have found that our heuristic works very well. We next apply the component legality test to determine which variable parts are legal. Finally, for each legal variable part, we apply our performance estimator to a mapping whose only specified parts are the mapping components for levels $1 \ldots k-1$ of all statements and the mapping component for level k of statement s_p. A legal variable part is considered profitable if the performance estimate obtained for it is "sufficiently good" compared to the performance estimates of the current tentatively accepted mappings. The definition of "sufficiently good" depends on user specifiable parameters (as will be described in Section 5.4).

WorthContinuing We wish to ensure that the ExtendMapping algorithm will terminate, so at each level we require that progress be made towards making the mappings 1-1. If the progress requirement is met then we apply our performance estimator to the partially specified mapping. The WorthContinuing predicate returns true iff the performance estimate obtained is "sufficiently good" compared to the performance of the current tentatively accepted mappings.

WorthAccepting The WorthAccepting predicate is the same as the WorthContinuing predicate except that the progress requirement is not required.

Modify tentatively accepted mappings We add the current mapping to the list of tentatively accepted mappings and remove any tentatively accepted mappings which have ceased to be "sufficiently good".

4 Information used by the Performance Estimators

In this section we define some terms and concepts which are useful in describing the performance estimators in the next section.

4.1 Rank of mappings and array references

For each statement, s_p we use $V_{pk} = \{v_{p1}, v_{p2}, \ldots, v_{pk}\}$ to represent the set containing the variable parts of the first k mapping components. For each array reference r, we use L_r to represent the set containing the variable parts of the subscripts of r. We use L'_r to represent the set containing the variable parts of all but the first subscript of r. For example, if r is $a(j, i+5, i+j, i+\mathtt{n})$, where i and j are index variables and $\mathtt{n}$ is a symbolic constant, then $L_r = \{j, i, i+j\}$ and $L'_r = \{i, i+j\}$.

The rank of a set of variable parts is the rank of a matrix containing the coefficients of the variable parts (the rows corresponds to variable parts and the columns corresponds to index variables). For example,

$$\text{Rank}(\{i-j, j-k, i-k\}) = \text{Rank}\begin{pmatrix} 1 & -1 & 0 \\ 0 & 1 & -1 \\ 1 & 0 & -1 \end{pmatrix} = 2$$

Given two sets of variables parts S and T, we use $S \lhd T$ to indicate that all of the terms in S can be expressed as linear combinations of the terms in T. This can be determined by checking whether $rank(S \cup T) = rank(T)$.

We define $increases\_rank(s_p, k)$ as $V_{pk} \not\lhd V_{pk-1}$. If $increases\_rank(s_p, k)$, we know that for given values of $f_{p1}, \ldots, f_{pk-1}$, there may be more than one value for f_{pk}. So when we generate code we will have to create a loop to iterate over the range of possible values of f_{pk}.

4.2 A group of statements

Two statements s_p and s_q are said to be in the same *group* at level k unless there exists a level $j(< k)$ such that f_{pj} is a constant, f_{qj} is a constant and $f_{pj} \neq f_{qj}$. If two statements are in different groups at a given level then their execution periods will not overlap at that level.

4.3 Dependences carried at a level

Let d_{pq} be the relation representing all data dependences from statement s_p to statement s_q in the original program. We define T_p^k as:

$$T_p^k : [i_{p1}, \ldots, i_{pm_p}] \rightarrow [f_{p1}, \ldots, f_{pk}]$$

, i.e. the first k levels of T_p. A dependence $i \rightarrow j \in d_{pq}$ is satisfied at level k if $T_p^k(i) \prec T_q^k(j)$ (where $\prec$ denotes lexicographically precedes). If there exists a dependence that is not satisfied before level k but is satisfied at level k, then level k is said to carry a dependence:

$$carries(p, q, k) \quad \equiv \quad \exists\ i \rightarrow j \in d_{pq} \text{ s.t. } T_p^{k-1}(i) = T_q^{k-1}(j) \wedge T_p^k(i) \prec T_q^k(j)$$

Note that this information is also used by the alignment algorithm, so it has already been calculated by the time we do performance estimation.

5 Example Performance Estimators

In this section we describe three performance estimators that we have implemented. For each performance estimate we first describe a calculation based on complete mappings and then explain how we can generate a lower bound on this quantity given only partially specified mappings.

5.1 Simplicity

All other things being equal, we would like to produce code which the user can easily understand. The complexity of code produced is closely related to the complexity of the mapping components. So the first component of our simplicity estimate is:

$$S_1 = \sum_{p \in Stmts} \sum_{1 \leq k \leq n} \left(\sum_{1 \leq j \leq m_p} {t_{pkj}}^2 + neg(t_{pkj}) \right)^2$$

where t_{pkj} is the coefficient of iteration variable i_{pj} in mapping component f_{pk} and neg(x) is 1 if x is negative and 0 otherwise. Note that there are a large number of other functions which would also be acceptable characterizations of simplicity, and that the above choice is somewhat arbitrary.

When a program contains more than a single statement, the complexity of the code produced will depend on how the execution periods of the statements overlap. If two statements are in different groups at a given level, their periods of execution will not overlap at that level, so we can generate simple sequential code. If two statements are in the same group at a given level, their periods of execution may overlap at that level. This will generally lead to complex code unless their periods of execution exactly coincide. So the second component of our simplicity estimate is:

$$S_2 = \left| \{p, q, k \mid p, q \in Stmts \wedge 1 \leq k \leq n \wedge group(p, k) = group(q, k) \wedge f_{pk} \neq f_{qk}\} \right|$$

Our simplicity estimate is $S_1 + S_2$.

We now consider how we can generate lower bounds on these quantities given only the first k levels of a mapping. We can easily determine the part of S_1 contributed by the first k levels. We use $nesting\_level(s_p) - rank(V_{pk})$ as our lower bound on the part of S_1 contributed by levels $k+1, \ldots, n$ (where $nest(s_p)$ is the number of loops nested around statement s_p in the original program). We can easily determine the part of S_2 which is contributed by the first k levels. For a lower bound on the part of S_2 contributed by levels $k+1, \ldots, n$, the best we that we can come up with is zero.

5.2 Parallelism Granularity

Our parallelism granularity estimate is designed to measure the granularity of the parallelism in a program rather than the total amount of parallelism in a program. For each statement we try to determine its outermost parallel loop. Our assumption is that this loop (if it exists) will always have enough iterations to exploit all of the available parallelism in the machine. If this assumption is correct then the most important factor affecting performance will be the granularity of the parallelism. Our parallelism granularity estimate for a given statement is the total number of iterations executed by sequential loops outside the outermost parallel loop. If all loops inside the outermost parallel loop are parallel, then our definition of parallelism granularity is equivalent to latency[Fea92].

```
      for 10 i = 1, 8 do
        for 10 j = 1, 128 do
          forall 10 k = 1, 100 do
            for 10 m = 1, 10 do
10              ...
```

Figure 4: Program with parallelism granularity estimate of $128 \times 8 = 1024$

This will be a good indicator of the amount of overhead incurred through starting up and synchronizing parallel loops and hence closely related to execution time.

We could calculate symbolically the number of iterations executed by the outermost loops, however such symbolic results would be difficult to compare. We could use profiling to obtain estimates for these symbolic constants, however for the sake of simplicity we instead assume that all loops have $N = 100$ iterations.

To determine the outermost parallel loop of a given statement we must determine which levels of its mapping correspond to loops and which of these loops are parallel. Level k of mapping T_p corresponds to a loop if $increases\_rank(T_p, k)$. A loop is parallel if it doesn't carry a dependence between any statements in its body. Two statements can only be in the same loop at a given level if they are in the same group at that level and both statements require a loop at that level. So level k of T_p corresponds to a parallel loop iff:

$$\begin{array}{l} increases\_rank(s_p, k) \wedge \\ \neg \exists q \in Stmts \text{ s.t. } group(p,k) = group(q,k) \wedge increases\_rank(s_q, k) \wedge (carries(p,q,k) \vee carries(q,p,k)) \end{array}$$

If level r is the first level of T_p that corresponds to a parallel loop then the number of outermost sequential loops is $rank(T_p, r) - 1$, so the total number of iterations executed by outermost sequential loops is $N^{rank(T_p,r)-1}$ The parallelism granularity estimate for the entire mapping is the sum of the estimates for each statement.

We now consider how to determine a lower bound on the above quantity given only the first k levels of a mapping. If the first k levels of the mapping contains a parallel loop then the formula given above still applies. Otherwise we assume that level $k+1$ will correspond to a parallel loop giving us an estimate of $N^{rank(T_p,k)}$.

5.3 Locality

Our estimate of locality is an estimate of the number of cache misses that will occur. As before, we assume that each loop has $N = 100$ iterations. We also assume that cache lines contain $C = 8$ array elements, and that array layout is according to FORTRAN conventions (i.e., `a(i,j)` and `a(i+1,j)` occupy adjacent memory locations). To simplify the problem we only take into account self reuse, i.e., reusing a cache line used by a previous iteration of

the same reference. This simplification is justifiable because self reuse can reduce the number of cache misses by a factor of $O(N)$, whereas group reuse can only reduce the number of cache misses by a factor of $O(1)$. If an array reference occurs multiple times in a statement, we only consider one occurrence (only the first occurrence can cause a cache miss).
We say that an array reference r:

- is completely pinned at level k if and only if $L_r \lhd V_{pk}$,
- is partially pinned at level k if and only if $L'_r \lhd V_{pk} \wedge L_r \not\lhd V_{pk}$,
- is made stride 0 after level k if r was not pinned at level $k-1$, and becomes completely pinned at level k.
- is made stride O(1) in level k if r was partially pinned at level $k-1$ and becomes completely pinned at level k.

We assume that all loops will have enough iterations to completely flush the cache. So if a reference r becomes stride 0 after level k, the expected number of cache misses is $N^{\text{rank}(V_{pk})}$. If a reference r becomes stride O(1) in level k, the expected number of cache misses is $N^{\text{rank}(V_{pk})}/C$. We sum the number of cache misses for all array references then divide by the total number of accesses to get the cache miss rate.

We now consider how to generate a lower bound on this quantity given only the first k levels of a mapping. For array references which have already been pinned at level k, we estimate cache misses as we did above. To determine lower bounds for the remaining references we must consider all references in a statement at the same time rather than considering each reference in isolation. For a given statement s_p we use Ref to refer to the set of references in statement s_p that aren't already pinned. For each $i \in \{1, \ldots, |Ref|\}$, we wish to determine a lower bound on the number of additional loops required to pin any i references in Ref.

For each $r \in Ref$, the number of additional loops required to pin r in isolation is $p_r = rank(V_{pk} \cup L_r) - rank(V_{pk})$

Lemma 1 Pinning all references in a set R requires P_R additional loops, where

$$P_R = \text{rank}(V_{pk} \cup \bigcup_{r \in R} L_r) - \text{rank}(V_{pk})$$

Proof Each additional loop can increase the rank of V_{pk} by at most one. □

We could determine the minimum number of additional loops to pin any i references in Ref as $\min\{P_{R'} \mid R' \subseteq Ref \wedge i = |R'|\}$. However, this would require exponential time for some i's. We therefore use approximate methods.

Lemma 2 Pinning any $|R| - 1$ references in a set R requires at least $P_R - \max\{p_r | r \in R\}$ additional loops.

Proof Assume there exists an $R' = R - \{r\}$ that requires $P_{R'} < P_R - p_r$ additional loops to be completely pinned. Then we can pin all the references in R with $P'_R + p_r$ additional loops, a contradiction. □

We use S_R to denote the *sorted* sequence of values $[p_r \mid r \in R]$.

Lemma 3 Pinning any i references in a set R requires at least $D^i_R = \max(S_R[i], P_R - \sum_{j=i+1}^{|R|} S_R[j])$ additional loops.

Proof For all sets $R' \subseteq R$ such that $|R'| = i$, there exists a element $r \in R'$ that in isolation requires $S_R[i]$ additional loops to be pinned. The second term comes from repeated applications of Lemma 2. □

Lemma 4 Pinning any i references in the set $(R_1 \cup R_2)$ requires at least
$B(R_1, R_2, i) = \min_{p,q \text{ s.t. } 0 \le p \wedge 0 \le q \wedge p+q=i} \max(D^p_{R_1}, D^q_{R_2})$ additional loops.

Proof Pinning p references in R_1 and and q references in R_2 separately can't be more difficult than pinning them simultaneously. $\square$

Lemma 5 Pinning any i references in a set $R_1 \cup R_2$ requires at least $\tilde{B}(R_1, R_2, i) = \min_{p,q \text{ s.t. } 0 \leq p,q \wedge p+q=i} \max(D^p_{R_1}, S_{R_2}[q])$ additional loops.

Proof $S_{R_2}[q] \leq D^q_{R_2}$, therefore $\tilde{B}(R_1, R_2, i) \leq B(R_1, R_2, i)$. $\square$

Theorem 1 The following program calculates $\tilde{B}(R_1, R_2, i)$.

```
$p = i$
$q = 0$
$b = P_{R_1} - \sum_{j=p+1}^{|R_1|} S_{R_1}[j]$
while $p \geq 1 \wedge b - S_{R_1}[p] > S_{R_2}[q+1]$ do
    $b = b - S_{R_1}[p]$
    $p = p - 1$
    $q = q + 1$
return $\max(S_{R_1}[k], \min(b, S_{R_2}[q]))$
```

Proof Once we reach a p and q such that the $S_{R_2}[q]$ term dominates, going further is always going to produce values that will be discarded by the minimum calculation of Lemma 5. $\square$

We could use $max_{R_1,R_2 \text{ s.t. } R_1 \cup R_2 = Ref} \tilde{B}(R_1, R_2, i)$ as a lower bound on the number of additional loops required to pin any i references in Ref, but that would be to expensive. We only consider some partitions of Ref into R_1 and R_2 and therefore get a possibly more conservative bound:

$$P^i_{Ref} = max_{R_1,R_2 \text{ s.t. } R_1 \cup R_2 = Ref \wedge \forall r_1 \in R_1, r_2 \in R_2 \; p_{r_1} < p_{r_2}} \tilde{B}(R_1, R_2, i)$$

If b is the number of distinct values of p_r for $r \in Ref$, then we will compute $O(b)$ rank computations, and execute the algorithm in Theorem 1 $O(b|Ref|)$ times. Each execution might take $O(|Ref|)$ time, but in practice will rarely take more than a few steps.

We now consider partially pinning the references in Ref. In order for it to be possible to partially pin r before r is completely pinned, it must be the case that $V_{pk} \cup L_r \not\trianglelefteq V_{pk} \cup L'_r$. Let $\widehat{Ref}$ denote this set of references (where the first subscript has been removed from each reference) and calculate $P^i_{\widehat{Ref}}$ as above.

We use the P^i_{Ref}'s and $P^i_{\widehat{Ref}}$'s to calculate m, a lower bound on the total number of cache misses:

```
$r = \text{rank}(V_{pk})$
$m = 0$
$a = |\widehat{Ref}|$                          /* first available partial pin */
for $i = |Ref|$ downto 1 do
    while $a \geq 1 \wedge P^a_{\widehat{Ref}} \geq P^i_{Ref}$ do      /* can't partial pin $a$ before pinning $i$, */
        $a = a - 1$                        /* throw away $a^{\text{th}}$ partial pin */
    if $a \geq 1$
        then
            $m = m + N^{r+P^i_{Ref}}/C$        /* $i^{\text{th}}$ pin is stride $O(1)$ in level $r + P^i_{Ref}$ */
            $a = a - 1$                    /*Use up $a^{\text{th}}$ partial pin */
        else
            $m = m + N^{r+P^i_{Ref}}$          /*$i^{\text{th}}$ pin is stride 0 after level $r + P^i_{Ref}$ */
```

Example

Assume the statement we are analyzing is `c(i,j) += a(i,k) * b(k,j)`. Below we give p_r, P_{Ref} and $P_{\widehat{Ref}}$ for each of several examples of partially specified mappings (in each case, only the first level is specified).

v_1	$p_{\texttt{c(i,j)}}$	$p_{\texttt{a(i,k)}}$	$p_{\texttt{b(k,j)}}$	P_{Ref}	$P_{\widehat{Ref}}$	m	%
j	1	2	1	$[1,2,2]$	$[0,0,1]$	$N^2/C + N^3/C + N^3/C$	8.38%
k	2	1	1	$[1,2,2]$	$[0,1]$	$N^2 + N^3/C + N^3/C$	8.67%
i	1	1	2	$[1,2,2]$	$[1]$	$N^2 + N^3 + N^3/C$	37.83%
$i+j$	1	2	2	$[1,2,2]$	$[1,2]$	$N^2 + N^3 + N^3/C$	37.83%

Except for the $i+j$ mapping, all of the bounds we determine in this example are tight, i.e. there actually exist extensions of these mappings with the specified locality estimates. For the $i+j$ mapping, we predict that we could have one statement stride 0 after level 2 and two statements stride $O(1)$ in level 3. In fact, the best we can do, if the variable part of the outermost loop is $i+j$, is one statement stride 0 after level 2, one statement stride $O(1)$ in level 3 and one statement stride 0 after level 3.

In their paper on access normalization [LP92], Li and Pingali analyze the following code fragment:

```
for i = 1 to n do
    for j = i to min(i+2*b-2,n) do
        for k = max(i-b+1,j-b+1,1) to min(i+b-1,j+b-1,n) do
            C(i,j-i+1) += $\alpha$*A(k,i-k+b)*B(k,j-k+b) + $\alpha$*A(k,j-k+b)*B(k,i-k+b)
```

When we apply our methods to this example, we correctly predict that using certain skewed mapping such as $i-k$ or $j-i$ will produce the best efficiency. Some predictions made by our algorithms are:

v_1	$p_{\texttt{c(i,j-i+1)}}$	$p_{\texttt{(a\|b)(k,i-k+b)}}$	$p_{\texttt{(a\|b)(k,j-k+b)}}$	P_{Ref}	$P_{\widehat{Ref}}$	m	%
$i-k$	2	1	2	$[1,1,2,2,2]$	$[0,0,1,1,1]$	$2N^2/C + 3N^3/C$	7.55%
$j-i$	1	2	2	$[1,2,2,2,2]$	$[0,1,1,1,1]$	$N^2/C + 4N^3/C$	10.3%
k	2	1	1	$[1,1,1,2,2]$	$[1]$	$3N^2 + N^3 + N^3/C$	23.1%
i	1	1	2	$[1,1,2,2,2]$	$[1,1]$	$2N^2 + N^3 + 2N^3/C$	25.4%
j	1	2	1	$[1,1,2,2,2]$	$[1,1]$	$2N^2 + N^3 + 2N^3/C$	25.4%
$i+j$	2	2	2	$[2,2,2,2,2]$	$[1,1,1,2]$	$2N^3 + 3N^3/C$	47.5%

5.4 Trade-off mechanism

Simplicity, parallelism granuarity and locality are three independent properties of programs, each having an affect on execution time to various degrees depending on the architecture. Unfortunately, a program that is optimal with respect to one of these properties is often less than optimal with respect to one or more of the other properties. So in trying to select a transformation we are usually faced with a trade-off between these properties.

We provide two complementary mechanisms for the user to specify how they would like to handle these trade-offs. For each property p the user specifies two percentages: MAND$_p$ and DESIR$_p$. We use p_a to denote the estimate of property p for mapping a.

A mapping b is *sufficiently good* iff:

$$\forall a, p \ \ \frac{p_b - p_a}{p_a} \leq \mathrm{MAND}_p \ \ \wedge \ \ \exists a, p \text{ s.t. } \frac{p_b - p_a}{p_a} \leq \mathrm{DESIR}_p$$

When a new mapping is added to the list of tentatively accepted mappings, it may cause one or more mappings in the list to be removed. Mapping b causes mapping a to be removed from the list iff:

$$\exists p \ \ \frac{p_b - p_a}{p_a} > \mathrm{MAND}_p \ \ \vee \ \ \forall p \text{ s.t. } \frac{p_b - p_a}{p_a} > \mathrm{DESIR}_p$$

6 Results

The framework described in [KP93] and the system described in this paper, have been implemented in our extension of Michael Wolfe's `tiny` tool. Our extension of `tiny` is available via anonymous ftp from `ftp.cs.umd.edu` in the directory `pub/omega`. In this section we give examples and results based on our implementation.

In Figure 5, we show part of the search tree associated with the following matrix multiply program, Gray nodes were rejected based on the user specified trade-off parameters shown in the table.

```
      do 10 i = 1,n
        do 10 j = 1,p
          do 10 k = 1,m
10          c(i,j) += a(i,k) * b(k,j)
```

	simplicity	granularity	locality
MAND	50%	∞%	50%
DESIR	$-\infty$%	5 %	5%

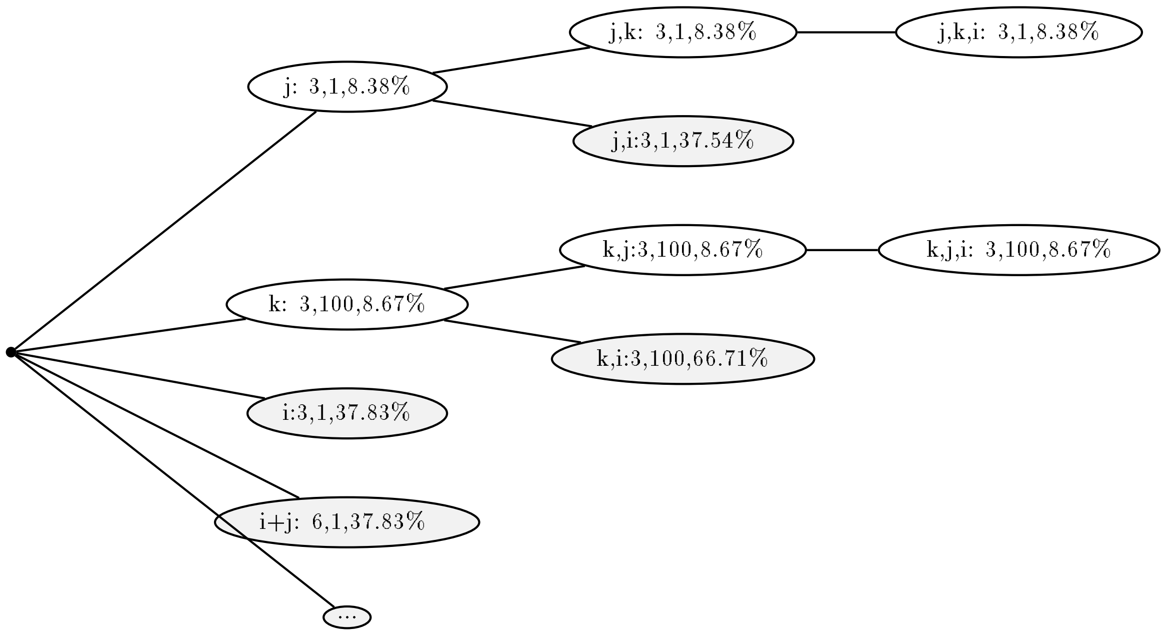

Figure 5: Mapping Search Tree

The two mappings selected are:

$T_{10} : [i, j, k] \quad \rightarrow [j, k, i]$ and $T_{10} : [i, j, k] \quad \rightarrow [k, j, i]$

```
forall j = 1,p do
  for k = 1,m do
    forall i = 1,n do
      c(i,j) = c(i,j)+a(i,k)*b(k,j)
    endfor
  endfor
endfor
```

```
for k = 1,m do
  forall j = 1,p do
    forall i = 1,n do
      c(i,j) = c(i,j)+a(i,k)*b(k,j)
    endfor
  endfor
endfor
```

Many researchers [WL91, KM92] have determined experimentally that the best non-blocked[2] mapping for matrix

[2] We can represent blocking transformations as mappings, however the ExtendMapping algorithm currently doesn't produce them.

multiply is the JKI mapping, closely followed by the KJI mapping.

The choice of trade-off parameters can have a large affect on the set of mappings accepted. For example, if we change $\text{MAND}_{\text{granularity}}$ to 50% then only the JKI mapping is selected. If in addition to changing $\text{MAND}_{\text{granularity}}$ to 50% we also changed $\text{MAND}_{\text{locality}}$ to ∞% then we would get four mappings: JKI, JIK, IJK and IKJ.

7 Conclusions

We have presented a new approach to determining which program transformation(s) to apply. Our decision to represent transformations as mappings, reduces the problem to that of finding an good mapping. We have shown how to estimate the performance of a program given only a mapping for that program. We have also shown how to determine an lower bound on performance given only a partially specified mapping. These abilities can be used to guide the search for good mappings in a search space which otherwise would have been infeasibly large.

Experience with our implementation has shown us that very good results can be obtained in a feasible amount of time for kernel sized programs. For larger sections of code, further refinements will be required before our techniques become feasible. However, we still believe that the basic idea of estimating performance based on partially specified mappings will remain at the heart of our future systems.

References

[ADY92] L. Khachiyan A. Darte and Y.Roberts. Linear scheduling is close to optimality. In *International Conference on Application Specific Array Processors*, pages 37–46, 1992.

[Ban90] 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.

[Fea92] Paul Feautrier. Some efficient solutions to the affine scheduling problem, Part I, One-dimensional time. *Int. J. of Parallel Programming*, 21(5), Oct 1992. Postscript available as `pub.ibp.fr:ibp/reports/masi.92/78.ps.Z`.

[KM92] K. Kennedy and K.S. McKinley. Optimizing for parallelism and data locality. In *International Conference on Supercomputing*, pages 323–334, July 1992.

[KP93] 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.

[LMQ91] 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.

[LP92] 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.

[Nil80] Nils J. Nilsson. *Principles of Artificial Intelligence.* Morgan Kaufmann Publishers, 1980.

[ST92] Vivek Sarkar and Radhika Thekkath. A general framework for iteration-reordering loop transformations. In *ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 175–187, San Francisco, California, Jun 1992.

[Tou84] George J. Tourlakis. *Computability.* Reston Publishing Company, 1984.

[WL91] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, 1991.

[Wol89] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, Cambridge, MA, 1989.