Removing Impediments to Loop Fusion Through Code Transformations

Bob Blainey[1], Christopher Barton[2], and José Nelson Amaral[2]

[1] IBM Toronto Software Laboratory, Toronto, Canada
blainey@ca.ibm.com
[2] Department of Computing Science, University of Alberta, Edmonton, Canada
{cbarton, amaral}@cs.ualberta.ca

Abstract. Loop fusion is a common optimization technique that takes several loops and combines them into a single large loop. Most of the existing work on loop fusion concentrates on the heuristics required to optimize an objective function, such as data reuse or creation of instruction level parallelism opportunities. Often, however, the code provided to a compiler has only small sets of loops that are control flow equivalent, normalized, have the same iteration count, are adjacent, and have no fusion-preventing dependences. This paper focuses on code transformations that create more opportunities for loop fusion in the IBM® XL compiler suite that generates code for the IBM family of PowerPC® processors. In this compiler an objective function is used at the loop distributor to decide which portions of a loop should remain in the same loop nest and which portions should be redistributed. Our algorithm focuses on eliminating conditions that prevent loop fusion. By generating maximal fusion our algorithm increases the scope of later transformations. We tested our improved code generator in an IBM pSeries™ 690 machine equipped with a POWER4™ processor using the SPEC CPU2000 benchmark suite. Our improvements to loop fusion resulted in three times as many loops fused in a subset of CFP2000 benchmarks, and four times as many for a subset of CINT2000 benchmarks.

1 Introduction

Modern microprocessors such as the POWER4 have a high degree of available instruction level parallelism and are typically nested within a relatively slow memory subsystem with non-uniform access times. Both of these machine characteristics make the distribution of memory references within a program critical to achieving high performance. In many scientific applications, the structure of loop nests operating on dense data arrays is a primary determinant of overall performance. Compilers with advanced automatic loop restructuring capabilities have emerged to address this performance opportunity [1].

Two important and complementary transformations typically performed in a loop restructuring compiler are loop fusion and loop distribution. Important design decisions when implementing loop optimization include (a) the order in

B. Pugh and C.-W. Tseng (Eds.): LCPC 2002, LNCS 2481, pp. 309–328, 2005.

which these phases should be executed, and (b) whether the *smartness* of the loop optimization algorithm should be placed (i) in loop fusion, (ii) in loop distribution, or (iii) in both.

In this paper we introduce the algorithms used for loop fusion in the IBM XL Fortran and VisualAge®for C++ for AIX compilers. In these compilers maximal loop fusion is performed first and then selective loop distribution takes place, *i.e.,* the smartness is placed in the distribution phase of the loop optimization process. These compilers target the PowerPC architecture and have been in continuous production use since the introduction of the POWER architecture in 1990. In this paper we report performance results for the new IBM processor, the POWER4. The POWER4 processor features two microprocessors running in excess of 1 GHz on a single chip along with a large shared L2 cache and control logic for an even larger off-chip L3 cache and high bandwidth chip-to-chip communication. Each microprocessor features 8 parallel functional units executing instructions in an out-of-order fashion along with dedicated L1 data and instruction caches. As in the POWER3™ processor, the POWER4 data caches include support for automatic prefetching of linear reference streams.

The fusion of small loops to generate larger loops decreases the number of loop branches executed, creates opportunities for data reuse, and offers more instructions for the scheduler to balance the use of functional units. Possible negative effects of loop fusion are increased code size, increased register pressure within a loop, potential overcommiting of hardware resources and the formation of loops with more complex control flow. Increased code size can affect the instruction cache performance. Higher register pressure has the potential of resulting in code with undesirable spilling instructions. Architectures such as the POWER4 achitecture contain hardware support for prefetching linear reference streams. If a loop contains more reference streams than can be prefetched by the hardware, one or more of the reference streams will be plagued by cache misses, causing performance degredations. Loops with complex control flow have a longer instruction path length and can have negative side effects on later optimizations such as software pipelining.

The loop fusion algorithm used in this compiler scans the code to find pairs of normalized loops that can be fused and greedily fuses them. Two loops can be fused if they are control equivalent, have no dependences, and their bounds conform (see Section 4). In order to be fused, there must be no intervening code between the loops. In some situations the code that is between the loops has no data dependences with one of the loops. In this case the code can be moved either before the first loop or after the second loop. In this paper we describe our implementation of this data movement operation. We also implement loop peeling to allow the fusion of loops that originally had non-conforming bounds. Our algorithm processes loops in the same nesting level in a given control flow, moving intervening code, peeling iterations, and fusing loops until no more loops can be fused. We present experimental results comparing the loop fusion algorithm with and without these improvements.

In previous work published on loop fusion, the decision to fuse a set of loops was based on the evaluation of an objective function — usually a measurement of data reuse and/or estimates of resource usage [4, 6, 10]. In our implementation the decision of how the code should be aggregated into a set of loop nests is delayed until loop distribution. Therefore, we can apply maximal loop fusion without regard to resource usage or to the benefits of fusion. For compile time and implementation efficiency we use a greedy algorithm and do not consider cases in which an early fusion might prevent a later, potentially more profitable, fusion.

The main contributions of this paper are:

- A new algorithm that eliminates conditions that prevent loop fusion and increase the scope of later loop restructuring transformations.
- An implementation of the new fusion algorithm in the IBM production compilers for the eServer pSeries, and measured performance on the eServer pSeries 690 that is built around the new POWER4 processor.
- Experimental results that show that the algorithm increases the number of loops fused when compared with the algorithm in the original compiler.

The rest of the paper is organized as follows: Section 2 briefly introduces the POWER4 Architecture, which was used for the performance measurements. Section 3 describes the general loop optimizer that is used in this compiler. Section 4 describes the loop fusion algorithm and Section 5 presents some preliminary experimental results. Section 6 reviews related work.

2 The POWER4 Architecture

The POWER4 is a new microprocessor implementation of the 64-bit PowerPC architecture designed and manufactured by IBM for the UNIX®server market. It features two processor cores running at speeds up to 1.3 GHz placed onto a single die. Four of these dies are placed together to form one multi-chip module (MCM), containing eight processor cores. Each of the two processors on the die has a dedicated 64 KB direct mapped L1 instruction cache, a dedicated 32 KB 2-way set associative L1 data cache and a unified 1 KB 4-way set associative TLB supporting 4 KB and 16 MB page sizes. The two processors share a single 8-way set associative 1.44 MB on-chip combined L2 cache. Each 4 chip (8 processor) MCM has an attached 128 MB L3 cache and dedicated memory controller. For the experiments presented in this paper we used a dual MCM pSeries model 690 server. This machine runs at 1.1 GHz and has 64 GB of main memory[3].

Each L1 instruction cache can support up to 3 outstanding misses and each L1 data cache can support up to 8 outstanding misses. The L1 data cache and the L2 and L3 shared caches include support for automatic prefetching of linear reference streams. Each processor maintains a 12-entry prefetch address filter queue and up to 8 concurrent active prefetch streams. The L2 cache is organized into 3 slices, each 480 KB in size and can offer more than 100 GB/s in bandwidth [12].

3 Overview of Loop Optimizations

In the XL compilers, most optimizing transformations are applied to each loop nest in functions by the iterative application of several specialized passes. Loop fusion enlarges the scope in which later optimizations are applied. Fusion creates opportunities to improve data reuse, to generate coarser grain parallelism, to exploit the use of hardware prefetch streams, to improve the allocation of architected register files, and to improve the scheduling for load/store or floating-point dominated code, or for code that combines both types of operations. The larger scope available for these later optimizations is due to the aggregation of more code into a smaller number of loop nests. In order to reap these benefits we implement maximal fusion first, and later redistribute the code into separate loop nests. The distributor reaggregates code according to a set of constraints and the optimization of an objective function. If the original loop structure is already optimal, the distributor will usually re-create it. Thus the loop fusion phase performs maximal fusion without concern for potential negative effects in the code.

Figure 1 presents the sequence of transformations applied to the code, including loop fusion and loop distribution. Starting on the left of the figure, the early optimizations, aggressive copy propagation and dead store elimination, create opportunities for loop interchanging and loop unroll and jam. Conventional copy propagation algorithms do not move computations into a loop to prevent the enlargement of the dynamic path length of the loop. Our aggressive propagation, however, does move statements into a loop to enable the creation of perfectly nested loops. Figure 2 illustrates the aggressive copy propagation performed in this compiler. The original code is in Figure 2(a). After copy (and expression) propagation the code in Figure 2(b) is obtained, and after the dead store elimination, the code in Figure 2(c) results. Although the multiplication `x*y` now needs to be computed in every iteration of the inner loop, the combination of these two optimizations generates a perfectly nested loop that can be advantageous both for loop permutation and unroll-and-jam. Furthermore, the computation of `x*y` can be moved back out of the loop after the loop optimizer has completed.

Next, maximal loop fusion is performed. The goal is to enhance the scope for optimization in the loop distributor and not necessarily to improve performance on its own. Working with larger portions of the code, the distributor will

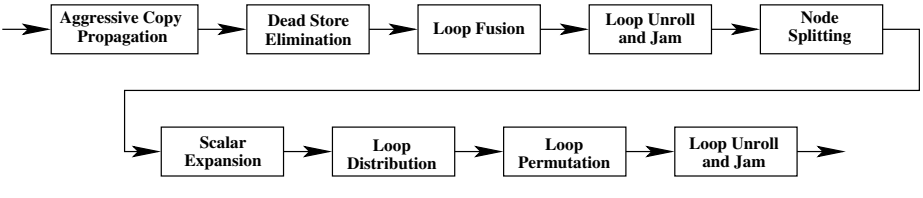


Fig. 1. Loop Optimizations

```
for(i=0 ; i<k ; i++)
 {
   s = x*y;
   for(j=0 ; j<n ; j++)
     V[i][j] = V[i][j] + s;
 }
```

(a)

```
for(i=0 ; i<k ; i++)
  {
   s = x*y;
   for(j=0 ; j<n ; j++)
     V[i][j] = V[i][j] + x*y;
  }
```

(b)

```
for(i=0 ; i<k ; i++)
  {
   for(j=0 ; j<n ; j++)
     V[i][j] = V[i][j] + x*y;
  }
```

(c)

Fig. 2. Example of aggressive copy propagation followed by dead store elimination

encounter more opportunities to explore data reuse, generate coarser grained parallelism, exploit prefetch, improve the use of architected registers, and schedule operations to the fix point, floating point, and load/store units.

After loop fusion, the compiler applies common-subexpression elimination, and *node splitting*. In order to keep the size of the data dependence graphs (DDGs) under control, complex statements are allowed in the code representation at this level. Each one of these statements is a *node* in the DDG. Because these nodes represent complex statements, a node may participate in multiple dependence relations. Node splitting separates a complex statement into two or more simpler statements, each participating in a disjoint dependence relation. In some cases node splitting allows the loop distributor to distribute two portions of a statement into separate loop nests. For instance, such a split is profitable when one part of the statement has self-dependences that prevent parallelization and the other part parallelizable.

The scalar expansion transformation identifies the use of scalars that induce anti or output dependences across loop iterations. In a traditional scalar expansion algorithm, each one of these scalar variables would be expanded into arrays with as many dimensions as required to eliminate the dependences. In this compiler, the expansion is limited to one dimension, and the variables are marked as *expandable* but the actual generation of the arrays is postponed until the code generation phase. At that point the expansion might not be necessary because of code aggregation done by the loop distributor or, if expansion is necessary, the required storage could be overlaid with existing temporary storage.

It is important to strike the right balance between the multiple conflicting goals of the loop distributor. In this compiler suite the loop distributor first identifies the minimal segments of code that must be distributed as a unit. For instance, if an *if* statement is encountered, the test along with the code that appears in both branches, up to but not including the join node, form a unit of code. These code units are called *aggregate* nodes. Aggregate and statement nodes, which form maximal strongly connected components of the DDG are grouped together to form π-nodes, named after the definition by Kuck [13]. Degenerate π-nodes are also formed from the remaining statement and aggregate nodes that are not part of any strongly connected component. A π-node may contain from a single statement to an arbitrarily complex portion of code. π-nodes are the units that the distributor works with.

Some of the characteristics of a π-node that are relevant for the loop distribution algorithm include: register requirements,[1] load/store usage, number of floating point and fixed point operations executed, and the number of prefetchable linear streams.[2] Another important attribute taken into consideration by the distributor is whether the code in a π-node is self-dependent or not. A π-node that is not dependent on itself is parallelizable and should be aggregated only with other non-self-dependent nodes.

Once the π-nodes are formed, the distributor creates an affinity graph that is an undirected weighted graph whose nodes correspond to π-nodes and whose weighted edges represent the affinity between the nodes. Currently the only measure of affinity used in the compiler is the potential for data reuse between the code in the nodes. The compiler uses a greedy algorithm in the distributor: it attempts to aggregate nodes in decreasing order of affinity. The decision about aggregating two π-nodes is based not only on the affinity in the graph, but also on whether aggregation would satisfy data dependences and whether aggregation is desirable based on node attributes. For instance, if the aggregation of two π-nodes would exceed the use of the existing prefetching streams, the nodes are usually not aggregated. Likewise self-dependent (non-parallelizable) nodes are usually not aggregated with non-self-dependent (parallelizable) nodes. Decisions about aggregating nodes are conditioned to the potential increase in data reuse.

After loop distribution, loop permutation and unroll and jam are performed. These transformations are limited in their application to perfectly nested loops and benefit from the loop distributor's efforts to isolate perfect nests.

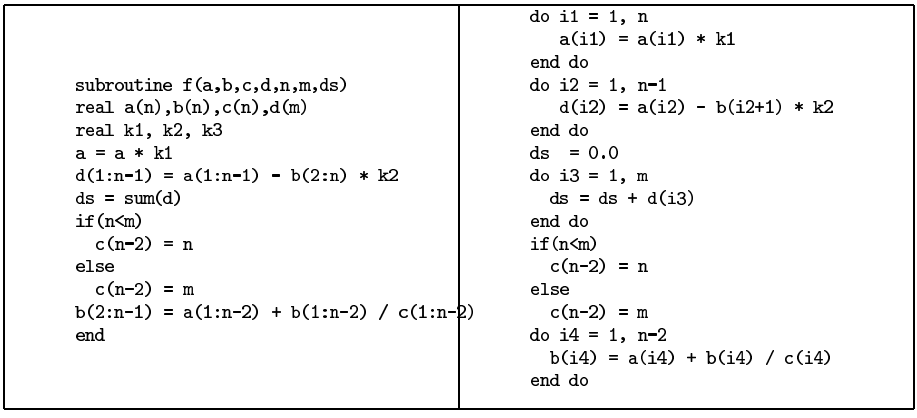
4 Loop Fusion Algorithm

In the XL compiler suite, loop normalization takes place prior to loop fusion. In other words, whenever possible, the loop starting count, its increment, and its direction (always increasing the index) are normalized. We divide loops into two classes: loops that are *eligible* for fusion and loops that are not eligible for fusion. Examples of loops that are *non-eligible* for fusion include loops that were specified to be parallel loops by the programmer (in OpenMP for instance), loops for which normalization fails, non-counted loops, and loops with side entrances and side exits. In order to be fused, two loops that are *eligible* for fusion must satisfy the following conditions:

- they must be conforming,
- they must be control equivalent,
- they must be adjacent, and
- there can be only forward dependences between the loop bodies.

[1] Loop body size is used as an estimator for register pressure.

[2] The number of prefetchable linear streams is an important characteristic for the optimization of code for the Power4 because this architecture has a hardware stream prefetching mechanism that is triggered by regular data accesses.



```
subroutine f(a,b,c,d,n,m,ds)
real a(n),b(n),c(n),d(m)
real k1, k2, k3
a = a * k1
d(1:n-1) = a(1:n-1) - b(2:n) * k2
ds = sum(d)
if(n<m)
  c(n-2) = n
else
  c(n-2) = m
b(2:n-1) = a(1:n-2) + b(1:n-2) / c(1:n-2)
end
```

(a) Fortran 90 Code

```
do i1 = 1, n
   a(i1) = a(i1) * k1
end do
do i2 = 1, n-1
   d(i2) = a(i2) - b(i2+1) * k2
end do
ds  = 0.0
do i3 = 1, m
  ds = ds + d(i3)
end do
if(n<m)
  c(n-2) = n
else
  c(n-2) = m
do i4 = 1, n-2
  b(i4) = a(i4) + b(i4) / c(i4)
end do
```

(b) Fortran 77 Code

Fig. 3. Fortran 90 and Fortran 77 versions of the code for running example

Two normalized loops are *conforming* if they have the same iteration count. A set of loops is *control equivalent* if, whenever one of the loops of the set is executed, all of the other loops must be executed. We say that a loop is executed if its exit test is executed at least once. Two loops are determined to be control equivalent using the dominator and post-dominator properties of the loops. If loop L_j dominates loop L_k and L_k post-dominates L_j then the two loops are control equivalent. Two loops L_j and L_k are *adjacent* if there is no intervening code between them, *i.e.,* in the Control Flow Graph, L_k is the immediate successor of L_j.

We use the contrived running example presented in Fortran 90 and Fortran 77 in Figure 3 to illustrate our loop fusion algorithm. This code example has four loops accessing four different arrays, `a`, `b`, `c`, `d`. We assume that there is no overlap between the memory locations of these arrays, *i.e.,* there is no `i` and `j` such that the address of `x(i)` overlaps with the address of `y(j)`, where `x` and `y` represent the arrays `a`, `b`, `c`, and `d`.

Figure 4 presents the LoopFusion algorithm. The algorithm operates one nest level at a time processing the outermost nesting level first and then moving toward the innermost level (step 1). First the algorithm partitions all the loops that are at the same nest level into sets of loops that are control equivalent. In step 4 all loops that are not eligible for fusion are removed from the set. Since all the loops in a set are control flow equivalent, dominance defines a total order over the set. Therefore, we can use the notion of moving *forward* and moving in *reverse* order through the set. The loop fusion algorithm iterates, alternating forward and reverse passes over the set, until it finds no more loops to be fused. Fusions and code movements that take place during a pass through a set of loops change the control flow graph and the dominance order between the loops. Therefore, before each pass the control flow graphk and the dominance relations are recomputed. The iterations processed in the while loop starting at step 7

```
LoopFusion
1.  foreach NestLevel N_i from outermost to innermost
2.          Gather identically control dependent loops in N_i
            into LoopSets
3.          foreach LoopSet S_i
4.                  Remove loops non-eligible for fusion from
                    S_i
5.                  FusedLoops ← True
6.                  Direction ← Forward
7.                  while FusedLoops = True
8.                          if |S_i| < 2
9.                            break
10.                         endif
11.                         Build Control Flow Graph
12.                         Compute Dominance Relation
13.                         FusedLoops =
                              LoopFusionPass(S_i,Direction)
14.                         if Direction = Forward
15.                           Direction = Reverse
16.                         else
17.                           Direction = Forward
18.                         endif
19.                 endfor
20.         end while
21. endfor
```

Fig. 4. Loop Fusion Algorithm

alternate between forward and reverse passes through the loop set until no loops are fused during a pass. In the code example of Figure 3, all loops are eligible for fusion and control equivalent, thus all four loops are in the same set, and have the following dominance order: `i1` → `i2` → `i3` → `i4` (we will identify the loops in the example by their index variables).

In a forward pass the LoopFusionPass algorithm presented in Figure 5 traverses a set of control flow equivalent loops in dominance order, while during a reverse pass the traversal is in post-dominance order. The function Intervening Code(L_j, L_k) checks whether the two loops are adjacent, *i.e.,* if there is intervening code between them. We use the dominance relation to determine the existence of intervening code. An aggregate node a_x *intervenes* between loops L_j and L_k if and only if L_j properly dominates a_x, $L_j \prec_d a_x$, and L_k properly post-dominates a_x, $L_k \prec_{pd} a_x$. Because the loops L_j and L_k are control equivalent, there cannot be a side entrance or a side exit to the intervening code between the two loops. If we find intervening code between L_j and L_k, we check if the intervening code can be moved either before the first loop or after the second one (step 3). Our algorithm allows for a portion of the intervening code to be moved above the first loop while the remainder of that code is moved

```
LoopFusionPass(S_i, Direction)
1.  FusedLoops = False
2.  foreach pair of loops L_j and L_k in S_i, such that L_j
          dominates L_k, in Direction
3.          if InterveningCode(L_j, L_k) = True and
              IsInterveningCodeMovable(L_j, L_k) = False
4.            continue
5.          endif
6.          σ ← |κ(L_j) − κ(L_k)|
7.          if L_j and L_k are non-conforming and
              σ cannot be determined at compile time
8.            continue
9.          endif
10.         if DependenceDistance(L_j, L_k) < 0
11.           continue
12.         endif
13.         MoveInterveningCode(L_j, L_k, Direction)
14.         if InterveningCode(L_j, L_k) = False
15.           if L_j and L_k are non-conforming
16.             L_m ← FuseWithGuard(L_j, L_k)
17.           else
18.             L_m ← Fuse(L_j, L_k)
19.           endif
20.           S_i ← S_i ∪ L_m − {L_j, L_k}
21.           FusedLoops = True
22.         else
23.           continue
24.         endif
25. endfor
26. return FusedLoops
```

Fig. 5. Loop Fusion Algorithm

after the second loop. This is necessary when a portion of the intervening code cannot be moved down because of dependences with L_k and the remainder of the code cannot be moved up because of dependences with L_j. The algorithm IsInterveningCodeMovable checks for this condition.

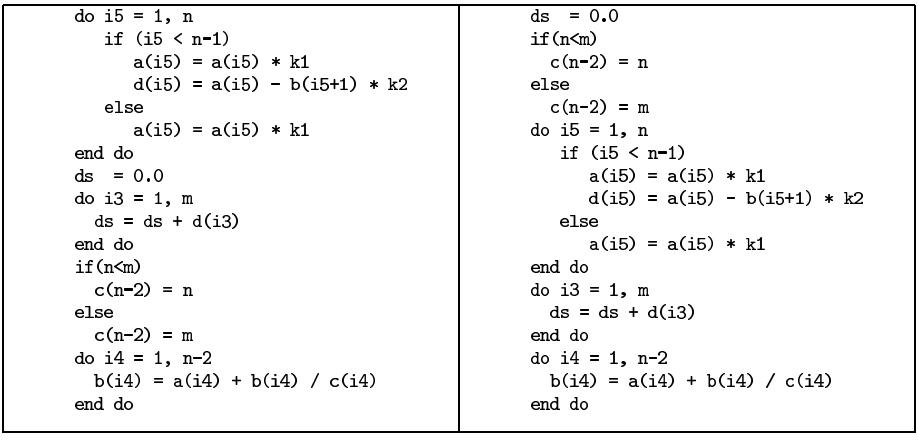
If the two loops do not conform, *i.e.,* if they have different iteration counts, they could be made to conform by guarding iterations of one of the loops. We are only considering loops that were normalized (loops for which normalization failed were eliminated in step 4 of the LoopFusion algorithm). In step 6 we compute the difference between the upper bound of the two loops, $\kappa(L_j)$ and $\kappa(L_k)$ and store the result in σ. Observe that this is a symbolic subtraction as the value of σ may not be known at compile time. In step 7 we abandon our attempt to fuse the loops L_j and L_k if σ cannot be determined at compile time.

On the other hand, if σ is a known constant, a guard is placed in the fused loop to inhibit the extra execution of one of the loop bodies (see step 16).

Figure 9 presents the algorithm FuseWithGuard used to fuse two non-conforming loops L_j and L_k. A new loop, L_m is created with the larger upper bound of the two loops (step 1). A guard branch is then created at the beginning of the loop (step 2) and the bodies of L_j and L_k are included within the guard (steps 3 and 4). The guard branch checks to see if the current iteration count is less than the lower upper bound of the two loops. The bodies of the original loops are then copied into the new loop, preserving the dominance relation between them. An else statement is then inserted to guard the second loop body (step 5). The longer loop is inserted in the else statement (step 6). This guarded fusion creates more code growth than an alternative technique that would simply guard the shorter loop. However, it is preferable in this compiler because it favors a later index set splitting transformation because it will allow the common portions of the fused loop to remain together.

In step 10 we check if the dependence relations between the bodies of loops L_j and L_k prevent fusion. This test is performed last because checking for dependences between loop bodies is the most expensive loop fusion condition that needs to be tested. If there is a negative dependence distance from L_j to L_k, the loops cannot be fused. In the IBM XL compiler suite, data dependences are computed on demand. For our algorithm, this computation is based on the SSA data flow representation within the context of a loop. The information about references to arrays is summarized in matrices of subscripts. These matrices are used along with vectors representing the bounds of surrounding loops to determine the dependence relation between two loop bodies, or between a loop body and intervening code. If there are dependences, the dependence analysis produces a dependence vector consisting of a distance or direction for each loop surrounding the reference pair.

The intervening code between loops L_j and L_k may itself contain loops. These loops are treated as regular code and are moved if dependences allow. During a forward pass, the intervening code is only moved up (step 13). This restriction on the direction of code movement during a pass is a result of an engineering design. A collection of data structures is used to store the control flow graph, the dominator and post-dominator trees, and the SSA data flow graph. We allow these data structures to become inconsistent after the fusion of loops and the movement of intervening code within a pass of the algorithm. These structures are rebuilt at the end of each pass. It would have been possible to modify the interface to these structures to allow them to be updated as fusion progressed, however we do not believe our approach has a noticeable effect on running times and it maintains the original interface. Because code is not moved down (or up) during a forward (or reverse) pass, even if all the intervening code is movable, the part of the code that must move down (or up), because of dependences, is not moved in this step. In this case the two loops do not become adjacent and cannot be fused in the same pass. Therefore, in step 14 we check once more if the loops are adjacent before fusing the two loops in step 18 and updating the



```
do i5 = 1, n
   if (i5 < n-1)
      a(i5) = a(i5) * k1
      d(i5) = a(i5) - b(i5+1) * k2
   else
      a(i5) = a(i5) * k1
end do
ds  = 0.0
do i3 = 1, m
  ds = ds + d(i3)
end do
if(n<m)
  c(n-2) = n
else
  c(n-2) = m
do i4 = 1, n-2
  b(i4) = a(i4) + b(i4) / c(i4)
end do
```

(a) After Fusing `i1` and `i2` into `i5`

```
ds  = 0.0
if(n<m)
  c(n-2) = n
else
  c(n-2) = m
do i5 = 1, n
   if (i5 < n-1)
      a(i5) = a(i5) * k1
      d(i5) = a(i5) - b(i5+1) * k2
   else
      a(i5) = a(i5) * k1
end do
do i3 = 1, m
  ds = ds + d(i3)
end do
do i4 = 1, n-2
  b(i4) = a(i4) + b(i4) / c(i4)
end do
```

(b) After moving intervening code up

Fig. 6. Completing first forward pass in running example

loop set in step 20. When all the intervening code is movable, the movement of the portion of the intervening code that can move up in step 13 prepares the loop set for a potential fusion in the next pass of the algorithm.

In the example of Figure 3(b) the first two loops to be compared are `i1` and `i2`. There are no dependences that prevent their fusion, they are adjacent, but they are non-conforming. The test in step 15 in Figure 5 is true and the two loops are fused using the algorithm in Figure 9. This fusion results in the loop `i5` shown in Figure 6(a).

The next comparison is between loops `i5` and `i3`. There are no dependences preventing fusion, and the loops are non-adjacent but the intervening code (initialization of `ds`) is movable to the point before `i5`. However, the difference between the iteration count of the two loops cannot be determined at compile time (we assume that `n` and `m` are not known until run time), and fusion of `i5` and `i3` fails.

Next `i5` and `i4` are compared, the two loops can be made to conform, there are no dependences preventing fusion, and all the intervening code (which includes loop `i3` and the if-then-else before `i4`) can be moved. Because of the dependence on `d` between `i5` and `i3`, `i3` only can be moved down to the point after `i4`. The dependence on `c(n-2)` requires the aggregate node that contains the if-then-else to be moved up to the point before `i5`. The MoveInterveningCode algorithm moves the intervening code that can be moved up to the point before `i5` resulting in the code shown in Figure 6(b). However, the test on step 14 fails, and the loops cannot be fused in this pass.

The control flow graph is rebuilt and the dominance and post-dominance relations recomputed before a reverse pass starts. In the reverse pass the loops `i4` and `i3` are compared, but they cannot be fused because we cannot determine the difference in their iteration count at compile time. Next, `i4` and `i5` are compared.

```
ds  = 0.0
if(n<m)
  c(n-2) = n
else
  c(n-2) = m
do i5 = 1, n
  if (i5 < n-1)
     a(i5) = a(i5) * k1
     d(i5) = a(i5) - b(i5+1) * k2
  else
     a(i5) = a(i5) * k1
do i4 = 1, n-2
  b(i4) = a(i4) + b(i4) / c(i4)
end do
do i3 = 1, m
  ds = ds + d(i3)
end do
```

(a) After moving intervening code

```
ds  = 0.0
if(n<m)
  c(n-2) = n
else
  c(n-2) = m
do i6 = 1, n
  if (i6 < n-2)
    if (i6 < n-1)
      a(i6) = a(i6) * k1
      d(i6) = a(i6) - b(i6+1) * k2
    else
      a(i6) = a(i6) * k1
    b(i6) = a(i6) + b(i6) / c(i6)
  else
    if (i6 < n-1)
      a(i6) = a(i6) * k1
      d(i6) = a(i6) - b(i6+1) * k2
    else
      a(i6) = a(i6) * k1
end do
do i3 = 1, m
  ds = ds + d(i3)
end do
```

(b) After fusing `i5` and `i4` into `i6`

Fig. 7. Final reverse pass on running example

The only intervening code (loop `i3`) can be moved down below `i4`. The difference in iteration count between `i4` and `i5` is 2 and there are no dependencies that prevent fusion. The intervening code between `i4` and `i5` is moved down (in step 13) resulting in the code shown in Figure 7(a). The two loops are then fused resulting in the code in Figure 7(b) and the reverse pass terminates. The next forward pass will result in no additional fusions and the algorithm will terminate.

As discussed in Section 3, the code is organized into aggregate nodes. An aggregate node is a minimum code segment that must be moved as a unit. Examples of aggregate nodes include a single statement, a nest of loops, or an if-then-else statement with arbitrarily complex code in each branch. The algorithm in Figure 8 checks if all the aggregate nodes in the intervening code found between two loops L_j and L_k can be moved to other places in the program. In step 1 we build the set InterveningCodeSet containing all the aggregated nodes that are *intervening code* between the two loops. An aggregate node a_x is intervening code between two loops L_j and L_k if L_j properly dominates a_x, $L_j \prec_d a_x$ and L_k properly post-dominates a_x, $L_k \prec_{pd} a_x$.

We cannot move aggregate nodes that might have side effects. Instances of code that have side effects include volatile load/store, statements that perform I/O, and unknown functions that might contain such statements. If any of the aggregate nodes in the intervening code between two loops have or may have side effects, the intervening code is *non-movable* (step 2).

When determining the direction in which an aggregate node a_x can move, we need to take into consideration the data dependences between a_x and the

remaining aggregate nodes in the intervening code, as well as the data dependence relations with the loops L_j and L_k. Thus we build a Data Dependence Graph G for the nodes in the aggregate node set (step 4). Then we traverse G in topological order to build the CanMoveUpSet, the set of nodes that can be moved to the point before the loop L_j (steps 5 to 10). A node a_y can move up if

```
IsInterveningCodeMovable(L_j, L_k)
1.  InterveningCodeSet ← {a_x | L_j ≺_d a_x and L_k ≺_pd a_x}
2.  if any node in InterveningCodeSet is non-movable
3.    return False
4.  Build a DDG G of InterveningCodeSet
5.  CanMoveUpSet ← ∅
6.  foreach a_y ∈ G in topological order
7.          if CanMoveUp(Predecessors(a_y)) and L_j ⊀ a_y
8.            CanMoveUpSet ← CanMoveUpSet ∪{a_y}
9.          endif
10. endfor
11. CanMoveDownSet ← ∅
12  foreach a_z ∈ G in reverse topological order
13.         if CanMoveDown(Successors(a_z)) and a_z ⊀ L_k
14.           CanMoveDownSet ← CanMoveDownSet ∪{a_z}
15.         endif
16. endfor
17. if InterveningCodeSet −
      (CanMoveUpSet ∪ CanMoveUpSet) = ∅
18.   return True
19. return False
```

Fig. 8. Algorithm to check if all intervening code can be moved

```
FuseWithGuard(L_j, L_k)
1.  Create L_m with upper bound max(κ(L_j), κ(L_k))
2.  Insert Guard Bound for min(κ(L_j), κ(L_k)) at beginning
    of L_m
3.  Copy body of L_j to L_m, within guard
4.  Copy body of L_k to L_m, after L_j body, within guard
5.  Insert else statement
6.  if (κ(L_j) > κ(L_k))
7.    Copy body of L_j to L_m, after else statement
8.  else
9.    Copy body of L_k to L_m, after else statement
10. endif
```

Fig. 9. Algorithm to fuse loops using a guard statement

there are no data dependences between the preceding loop L_j and a_y, $L_j \not\delta a_y$, and all the predecessors of a_y in G can also move up.

Similarly, in steps 11 to 16 we traverse G in reverse topological order to build the set of nodes that can move down, the CanMoveDownSet. In order to move a node a_z down, there must be no dependences between a_z and the second loop L_k, and all of a_z's successors must be able to move down. The test in step 17 tests if every aggregate node in the InterveningCodeSet can be moved either up or down.

The MoveInterveningCode called in step 13 of the LoopFusionPass uses the sets created by the IsInterveningCodeMovable to move code. If called during a forward pass, it simply traverses the DDG and moves any aggregate node that can move up to the point before the first loop L_j. Likewise, when called during a reverse pass, it moves all nodes that can move down to the point after the second loop L_k.

5 Results

We implemented the algorithms presented in Section 4 in the development version of the IBM XL compiler suite and ran benchmarks compiled with this modified compiler on an IBM eServer pSeries 690 machine built with the POWER4 processor. Figure 10 presents preliminary results for the SPEC2000 and SPEC95

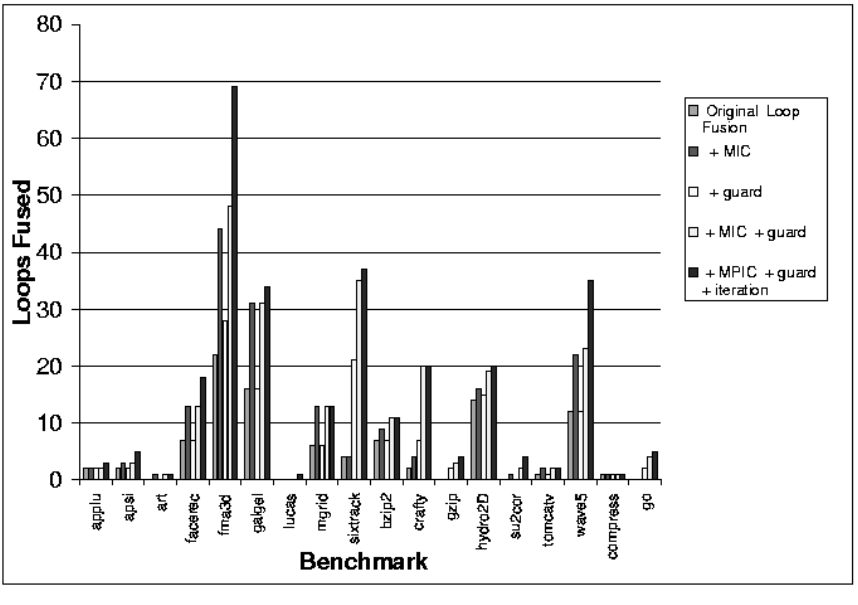


Fig. 10. Number of loops fused with each version of the compiler[3]

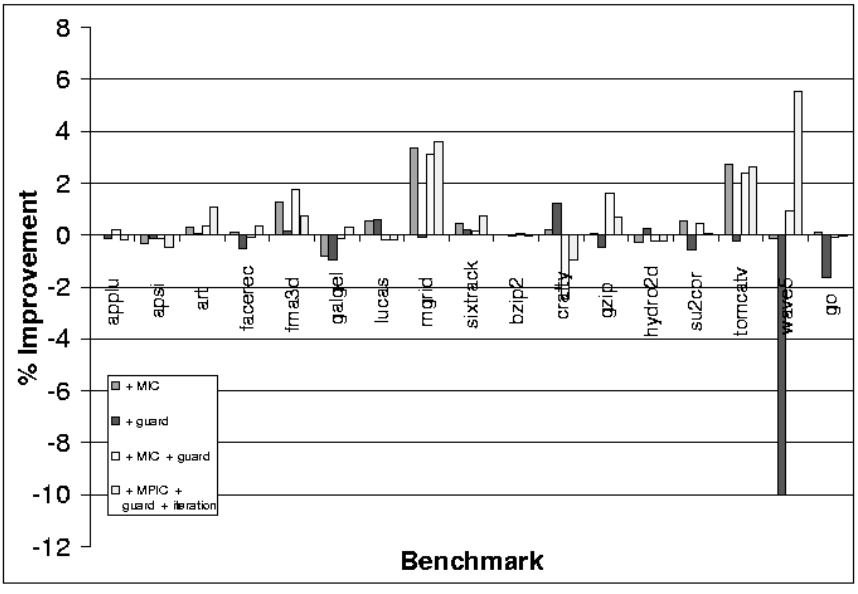


Fig. 11. Execution times for selected SPEC benchmarks with multiple versions of the compiler suite

benchmark suites. We only include in the figures of results the benchmarks in which our loop fusion algorithm affects code transformations, *i.e.,* benchmarks in which more loops are fused as a result of our algorithm. Also, benchmarks from SPEC95 which also occur in the SPEC2000 suite were not repeated.

We compare five versions of our algorithm with an implementation of basic loop fusion. Figure 10 presents the number of loop fusions that occurs in each version of the compiler. The versions of the compiler are:

Original: It is a basic loop fusion algorithm in which no code transformations are performed to try and make loops fusible.

+MIC: Does a single forward pass of the algorithm and moves any intervening code that can be moved up. If all of the intervening code cannot be moved up, fusion fails.

+MPIC: Part of the intervening code is moved up. In order for fusion to benefit from this, the iteration step must be included.

+guard: Non-conforming loops are fused using the guard branch. It does not, however, allow intervening code between two loops to be moved.

+MIC +guard: Combines guarding and simple code motion.

+MIC +guard +iteration: Complete implementation of the iterative algorithm executing as many passes as required for maximal fusion.

The results in Figure 10 indicate that each of the transformations affect different benchmarks. The movement of intervening code (columns *MIC* and *MPIC*) results approximately doubles the number of loops fused in fma3d, galgel,

facerec, and mgrid. The number of loops fused with *MPIC* or without *MIC* partial movement of intervening code is the same. This is to be expected because moving partial intervening code (move some statements up in current pass and the remainder down in the reverse pass) only benefits when the iteration step is added. The more complex *MPIC* pass, however, does not result in performance degredations when compared to the simpler *MIC* pass.

When loops are made to conform through the use of guard branches (*guard*), five times as many loops are fused in sixtrack, three times as many loops are fused in crafty and 27% more loops are fused for fma3d. Both gzip and go had 2 loops fused where none were fused in the original algorithm. However, for all other benchmarks, no extra loops are fused.

Combining the movement of intervening code with the guard branches for loop conformation (*MIC* and *guard*) produces a dramatic increase in the number of fused loops for many benchmarks.

Finally, the addition of the iteration step, in combination with the *MPIC* and *guard* options resulted in even more loops fused in several benchmarks (apsi, facerec, fma3d, galgel and sixtrack). This demonstrates that there are cases in which moving intervening code below the second loop (reverse pass) and splitting intervening code to move part of the intervening code above the first loop and the rest below the second loop can be very beneficial.

The technique to generate more fusion that we report in this paper is an enabling technology for optimizations that take place later in this compiler framework. We are now addressing some of those optimizations and finding ways in which they will benefit from the larger scope provided by our improved fusion. Nonetheless, a paper reporting advancement of compiler technology would not be complete without run times for SPEC benchmarks. Therefore, we present the running times for the different versions of the compiler in Figure 11. In the current version of the compiler, the impact of the increased fusion in the running times is modest. The most significant performance change is in wave5, where tripling the number of loops fused resulted in an improvement of 5.1% in the running time. The other significant performance change is in mgrid, where doubling the number of loops fused resulted in an improvement of 3.6% in the running time. We are in the process of obtaining run-time measurements using hardware counters in the POWER4 (performance of caches, load/stores completed, *etc.,*) to offer better explanations for the performance changes.

When non-conforming loops are fused as a result of adding guard statements, control flow is introduced into the loop body. The insertion of control flow into a loop might inhibit software pipelining. Thus, we would expect to see a degradation in performance in this case. We are currently investigating several of the benchmarks (crafty, fma3d and sixtrack) to determine if there were benefits to loop fusion (*i.e.,* data reuse) which offset the negative effects of introducing control flow. We are also working on a variation of index set splitting that will be able to identify branches within a loop and peel or split the loop to remove control flow splits.

6 Related Work

In this paper we presented improvements to the maximal loop fusion algorithms in the IBM XL Fortran and VisualAge for C compilers. Scant work has been published on maximal loop fusion followed by a loop distributor. In contrast, there has been extensive studies and experimentation with weighted loop fusion. Weighted loop fusion associates non-negative weights with each pair of loop nests. These weights are a measurement of the gains that are expected if the two loops were fused. Examples of gains represented in weighted loop graphs include potential for array contraction, improved data reuse, and improved local register allocation. Given such a weighted graph representing potential fusions, the goal of weighted loop fusion is to group the loop nests into clusters in a way that minimizes the total weight of edges that cross cluster boundaries [14].

In Gao *et al.,* a Loop Dependence Graph (LDG) provides a measure for the number of arrays that can be contracted when two loops are fused. Contracted arrays can be represented by a small number of scalar variables, thus removing memory instructions through the elimination of multiple load/stores of the same array. Their solution for this modified weighted loop fusion problem is based on the max-flow/min-cut algorithm [6]. The LDG based solution for loop fusion focuses on solving the problem of moving data between the cache and the registers, while our approach also takes into consideration the data cache performance.

Kennedy and McKinley used a polynomial reduction of the Multiway Cut problem to prove that solving the weighted loop fusion problem to maximize data reuse is NP-Hard. They also provide a greedy algorithm and a variation of the max-flow/min-cut algorithm to find approximated solutions for loop fusion [10]. Megiddo and Sarkar propose an integer linear programming solution for weighted loop fusion based on the Loop Dependence Graph (LDG) [14].

In [11] Kennedy and McKinley introduce the concept of *loop type*. In their experiments they used two types of loops: *parallel loops* are loops that have no loop-carried dependences, and *sequential loops* are loops that have at least one loop-carried dependence. In order to be fused, two loops must be of the same type, and must be conformable at level k, *i.e.* they are at the same level of perfect nests and all their outer loops are conformable. Two loops are conformable if they have the same iteration count. When performing *Unordered Typed Fusion* they try to produce the fewest loops without giving priority to any loop type. Through a reduction of the Vertex Cover problem they show that the Unordered Typed Fusion problem is NP-Hard. On the other hand, the Ordered Typed Fusion exercises a preference for fusing loops of a given type. For instance, parallel loops should be fused first — and thus potentially prevent some later fusion of sequential loops — when data reuse is not a concern. They propose a greedy algorithm to solve the Ordered Typed Fusion problem.

Loop distribution was introduced in Muraoka's Ph.D. thesis to improve parallelism [15]. Kuck introduced the idea of using a portion of the dependence graphs in the loop distribution algorithm, and defined of a π-block as a strong

[3] Measurements were not done using the official SPEC tools.

connected component of the data dependence graph [13]. Kennedy and McKinley designed a loop distribution algorithm for loops with complex control flow that does not replicate statements or conditions [9]. Hsieh, Hind, and Cytron extend the algorithm to allow the distribution of loops with multiple exits [8]. A comprehensive discussion of loop transformations is found in Bacon *et al.* [2].

In addition to improving data locality and reducing loop overhead, loop fusion can increase the granularity of parallelism and minimize loop synchronization. Some research on loop fusion focuses on multi-processor architectures and programs that can run in parallel. Singhai and McKinley developed a heuristic to fuse loops, taking into account both data locality and parallelism subject to register pressure [16].

Gupta and Bodik introduce a technique for loop transformations, including loop fusion, to be decided at run time instead of compile time [7]. Kennedy and McKinley provided an algorithm for fusing a collection of parallel and sequential loops that minimizes parallel loop synchronizations while maximizing parallelism [10].

7 Final Remarks

Many papers address the problem of optimizing the set of loops that should be fused to increase data reuse through graph partition and related techniques. However, there has been scant documentation of the actual process of combining code motion with fusion to enable maximal loop fusion and allow the redistribution of these loops at a later phase. Thus the description of the maximal loop fusion algorithms in this paper is an important contribution. Our algorithm is fast — there are no noticeable changes in the compile time when the fusion algorithm is implemented — and easy to implement and debug.

Our next line of study will include removing the restriction that all the loops that are fused must be control equivalent. We will investigate techniques similar to the ones described by Chen and Kennedy [5] that allow the critical path of a loop to be increased when there is a potential for benefits due to increased data reuse.

The loop distributor is also being enhanced in light of the new loop fusion implementation. Larger loop nests are being created, which are providing new scenarios for the loop distributor to to evaluate and deal with.

Loop alignment is another well known loop transformation which we are also working on. Performing loop alignment on loops that have a known distance negative dependence will allow even more loops to be fused.

A form of Index Set-Splitting is currently being developed that will analyze the guard branches generated by the FuseWithGuard algorithm and create new loops (through loop peeling or loop splitting) that do not contain the guards. This optimization will eliminate the control flow splits introduced during fusion, which should increase opportunities for later optimizations, such as software pipelining.

We believe this work provides an excellent framework to enhance the number of loops which are fused in a program. This loop fusion enables other optimizations, such as loop distribution, to make better decisions on how to organize loops to increase performance. While the runtime results presented do not indicate this work had any improvement on overall performance, we are confident that it does create more opportunities for other optimizations and work is currently underway to enhance these optimizations to benefit from these fusion results.

Acknowledgements

The work reported in this paper uses the infrastructure built by many hands. We thank the Toronto Portable Optimizer (TPO) and Toronto Optimizing Backend (TOBEY) teams for building this infrastructure. Special thanks to Jim McInnes, Ryan Weedon, and Roch Archambault for extensive and fruitful discussions. This research is supported by an IBM Centre for Advanced Studies (CAS) fellowship and by grants from the National Sciences and Engineering Council of Canada (NSERC), including a grant from the Collaborative Research Development (CRD) Grants program.

Trademarks

References

1. A. W. Lim an S.-W. Liao and M. S. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 103–112, June 2001.
2. D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, 1994.
3. Steve Behling, Ron Bell, Peter Farrell, Holger Holthoff, Frank O'Connell, and Will Weir. The power4 processor introduction and tuning guide. Technical Report SG24-7041-00, IBM, November 2001.
4. C. Ding and K. Kennedy. The memory bandwidth bottleneck and its amelioration by a compiler. In *2000 International Parallel and Distributed Processing Symposium*, pages 181–189, Cancun, Mexico, May 2000.
5. C. Ding and K. Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. In *International Parallel and Distribute Processing Symposium*, San Francisco, CA, April 2001.

6. Guang R. Gao, Russ Olsen, Vivek Sarkar, and Radhika Thekkath. Collective loop fusion for array contraction. In *1992 Workshop on Languages and Compilers for Parallel Computing*, pages 281–295, New Haven, Conn., 1992. Berlin: Springer Verlag.
7. R. Gupta and R. Bodik. Adaptive loop transformations for scientific programs. In *IEEE Symposium on Parallel and Distributed Processing*, pages 368–375, San Antonio, Texas, October 1995.
8. B.-M. Hsieh, M. Hind, and R. Cytron. Loop distribution with multiple exits. In *Proceedings of Supercomputing*, pages 204–213, November 1992.
9. K. Kennedy and K. S. McKinley. Loop distribution with arbitrary control flow. In *Proceedings of Supercomputing*, pages 407–417. IEEE Computer Society Press, November 1990.
10. K. Kennedy and K. S. McKinley. Typed fusion with applications to parallel and sequential code generation. Technical Report CRPC-TR94646, Rice University, Center for Research on Parallel Computation, 1994.
11. Ken Kennedy and Kathryn S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *1993 Workshop on Languages and Compilers for Parallel Computing*, pages 301–320, Portland, Ore., 1993. Berlin: Springer Verlag.
12. Kevin Krewell. Ibm's power4 unveiling continues: New details revealed at microprocessor forum 2000. In *Microprocessor Report: The Insider's Guide to Microprocessor Hardware*, November 2000.
13. D. J. Kuck. A survey of parallel machine organization and programming. *ACM Computing Surveys*, 9(1):29–59, March 1977.
14. Nimrod Megiddo and Vivek Sarkar. Optimal weighted loop fusion for parallel programs. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 282–291, 1997.
15. Y. Muraoka. *Parallelism Exposure and Exploitation in Programs.* PhD thesis, University of Illinois at Urbana Champaign, Dept. of Computer Science, February 1971. Report No. 71-424.
16. S. Singhai and K. McKinley. A parameterized loop fusion algorithm for improving parallelism and cache locality. *The Computer Journal*, 40(6):340–355, 1997.