**University of Alberta**

**Library Release Form**

**Name of Author**: Christopher Mark Barton

**Title of Thesis**: Code Transformations to Augment the Scope of Loop Fusion in a Production Compiler

**Degree**: Master of Science

**Year this Degree Granted**: 2003

Christopher Mark Barton
1145 75 Street
Edmonton, Alberta
Canada, T6K 2S4

**Date**: ____________

**University of Alberta**

# Code Transformations to Augment the Scope of Loop Fusion in a Production Compiler

by

**Christopher Mark Barton**

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

in

Compiler Optimization

Department of Computing Science

Edmonton, Alberta
Spring 2003

**University of Alberta**

**Faculty of Graduate Studies and Research**

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Code Transformations to Augment the Scope of Loop Fusion in a Production Compiler** submitted by Christopher Mark Barton in partial fulfillment of the requirements for the degree of **Master of Science** in *Compiler Optimization.*

José Nelson Amaral (Supervisor)

Duncan Elliot (External)

Jonathan Schaeffer

**Date**:

# Abstract

Loop fusion is a common optimization technique that takes two loops and combines them into a single large loop. Most of the existing loop fusion techniques focus on heuristics to optimize an objective function such as data reuse within the fused loop or parallelizability of the fused loop. Most programs have only a small number of loops that satisfy all of the conditions necessary for loop fusion to occur. This thesis identifies conditions that prevent loop fusion and focuses on code transformations applied to sets of loops to make these conditions satisfiable. It also presents algorithms to fuse loops in the IBM®XL compiler suite that generates code for the IBM family of PowerPC®processors. This compiler uses a heuristic in the loop distributor to ultimately determine which portions of a loop should remain in the same loop nest and which portions should be moved into a different loop nest. Thus, in order for the loop distributor to make the best possible decisions, maximal loop fusion is performed first to make the loop bodies as large as possible. The large loop bodies created by loop fusion can also benefit other loop optimizations which are run after loop fusion and before loop distribution. All of the algorithms have been implemented in the IBM XL compiler framework and tested on an IBM pSeries™ 630 machine equipped with a POWER4™ processor using the SPEC95 and SPEC2000 benchmark suites.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

This thesis introduces new algorithms that extend the scope of applicability of loop fusion in a state-of-the-art, robust, commercial compiler. We also describe our implementation of new techniques to remove loop fusion impeding conditions. Although these techniques are regarded as "known" by compiler practitioners, they are scarcely described in the literature and, to the best of our knowledge, this is the first integrated description of the complete set of algorithms required to remove all of the conditions that prevent loop fusion.

All algorithms were implemented in the IBM XL Compiler Suite which is the set of compilers commercially shipped by IBM.

Several of our improvements to this compiler framework have passed the standard IBM regression tests and will be shipped with the next commercial release of the XL Compiler Suite.

## 1.1 Original Loop Fusion in TPO

The IBM XL compilers contain a language independent, interprocedural analysis component called the Toronto Portable Optimizer (TPO). Before our contribution, TPO contained an efficient but basic and non-aggressive loop fusion algorithm. This algorithm tested for the existence of conditions that made loop fusion illegal in order to ensure correct code generation. However, no code transformations were attempted to remove these conditions and allow fusion to occur.

Due to the importance of loop fusion in TPO's loop optimization framework, the necessity for a more aggressive approach was evident. Our initial approach was to identify common cases that prevented loop fusion and implement code transformations to allow legal fusion in the presence of these cases. The cases initially identified were (*i*) the presence of intervening code between two loops and (*ii*) loops which had different iteration counts. Once we handled these two cases successfully, we addressed the remaining conditions preventing fusion. We implemented instrumentation inside the loop fusion framework to identify situations in which loop fusion was failing. This instrumentation allowed us to identify and prioritize the conditions resulting in failure to fuse loops. The priority was established based on the number of times a specific condition caused loop fusion to fail.

## 1.2 Known Compiler Techniques

Loop fusion is a very popular compiler optimization and is discussed extensively in the compiler literature. It is also implemented in several production compilers including: MIPSpro, Intel's C++ Compiler, the Open Research Compiler and IBM's XL Compiler Suite [35, 6, 3]. As a result, several techniques discussed here are well known within the compiler community. Specifically, using loop peeling to make the loop bounds of two loops conform is a well documented optimization[28][1]. Loop alignment to remove fusion-preventing dependencies from between two loops is also a well established technique[1].

The movement of intervening code from between two non-adjacent loops is a technique that is also well understood and probably implemented in several compilers. However, no formal algorithms have been published that specify how to move intervening code from between two loops. Furthermore, the techniques presented here dealing with movement of intervening code have a unique attribute in the way they reduce the need for rebuilding of internal data structures in TPO, thus reducing compilation time.

## 1.3 Summary of Major Contributions

The major contributions of this thesis are

- Description of four algorithms to enhance loop fusion:
  - IsInterveningCodeMoveable determines if code located between two loops can be moved to an alternative location;
  - FuseWithGuard adjusts the number of times that the two loops iterate;

  - FuseAndAlign eliminates dependencies between two loops through the insertion of conditional code in the loop bodies;
  - LoopFusionPass combines all code transformations to fuse loops;

- Development of the new NonCFELoopFusionPass algorithm that allows two loops that execute under different run time conditions to be fused through code duplication.
- Implementation of these algorithms.
- Integration of algorithms into a production compiler.
- Study of the performance of these new algorithms using an industry-standard set of benchmarks.
- Discussion of further improvements to the loop re-structuring code in a production compiler.

The IsInterveningCodeMoveable algorithm determines if statements located between two loop bodies can be moved. The movement of code from between two loops is likely to be implemented in some production compilers. However, no algorithms to describe the analysis required have been found in the literature.

The FuseWithGuard algorithm causes two loops, which originally would have iterated a different number of times, to iterate the same number of times. Loop peeling is a well-known technique that can be used to make two loops iterate the same number of times [1, 36, 2]. For loop fusion, the disadvantage of loop peeling is that it generates additional statements outside of the loop body. These statements can make subsequent attempts to fuse loops more complicated. Our technique uses a different approach to make the loops iterate the same number of times. It does not generate statements outside the loop thereby making subsequent loop fusion attempts more efficient.

The FuseAndAlign algorithm allows loops that have fusion-preventing dependencies between them to be fused. It is a variation of an optimization known as *loop alignment*. Loop peeling can also be used to perform loop alignment, thus making two loops fusible [1]. Again, loop peeling is unfavorable in this situation because it introduces additional statements outside of the loop bodies that can complicate the fusion of subsequent loops.

The two algorithms implemented in place of loop peeling (FuseWithGuard and FuseAndAlign) solve the same problems that can be solved by loop peeling. However, they improve the performance of the loop fusion algorithm by making subsequent loop fusions easier.

The NonCFELoopFusionPass algorithm allows two loops to be fused even if it cannot be guaranteed that both loop bodies will execute. No known algorithms have been developed to fuse loops in this situation and we have not found any discussions in the literature that address this issue. A detailed evaluation of the benefits/costs of non-CFE loop fusion with a large a large set of benchmarks is left for future research.

Finally, the LoopFusionPass algorithm combines the IsInterveningCodeMoveable, FuseWithGuard, FuseAndAlign, and NonCFELoopFusionPass algorithms listed above. The LoopFusionPass algorithm has two key properties: (*i*) it guarantees that loop fusion can occur before any code transformations are applied and (*ii*) it minimizes the time required to perform loop fusion. Again, there are no known discussions which detail how to combine code transformations to attempt and remove all conditions required for loop fusion. Furthermore, this algorithm is designed to work efficiently within the TPO framework to minimize the compile time overhead of fusing loops.

The algorithms have been implemented in C++ and integrated into the development version of TPO. They can be used to optimize loops written in C, C++ and FORTRAN. Several of the algorithms have passed the standard IBM regression tests. The remaining algorithms are currently undergoing the regression tests and are expected to be released as part of the TPO framework with the next official release of the IBM XL Compile suite.

Chapter 2 provides definitions for terms used throughout this document. Chapter 3 describes TPO in more detail, including a description of the loop optimizations that it performs. Chapter 4 presents the algorithms developed. Specific combinations of the algorithms have been tested with the SPEC95 and SPEC2000 benchmark suites. The results from these tests are presented in Chapter 5. Chapter 6 discusses related work. Chapter 7 discusses conclusions which can be drawn from the algorithms and the experimental results. Finally, the Future Work chapter of this thesis lists suggestions for improving the loop fusion algorithms in order to make loop fusion more profitable. It also includes a discussion of how the loop optimization framework within TPO can be enhanced to further benefit from this more aggressive loop fusion implementation. Appendix A contains results of each optimization applied independently to the SPEC2000 benchmark suite as well as other combinations of the optimizations performed on specific SPEC2000 benchmarks.

# Chapter 2

# Definitions

**Basic Block** A basic block is a sequence of executable statements that has the following attributes:

- There is a single entry point at the first statement in the block
- There is a single exit point at the last statement in the block

Thus, if one statement in a basic block is executed, all statements in the block must be executed.

**Control Flow Graph** A Control Flow Graph (CFG) is a directed multigraph. Nodes in the CFG correspond to Basic Blocks, and edges in the CFG correspond to transfer of control between basic blocks.

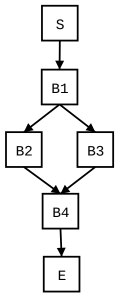


Figure 2.1: Simple Control Flow Graph

Figure 2.1 shows a simple control flow graph. The CFG contains four basic blocks ($B1$, $B2$, $B3$, and $B4$), a distinguished *Entry Node* (S) and a distinguished *Exit Node* (E). Every CFG must have exactly one entry node and exactly one exit node.[1]

[1]If the CFG has multiple entry or multiple exit nodes, a simple graph transformation inserts a new entry and/or exit node along with the necessary new edges.

Block $B1$ has two successors. For a given execution of this graph, either $B2$ or $B3$ is executed, but not both. The last statement in $B1$ is a test (*e.g.,* an *if* statement). Based on the outcome of the test, one of the two paths is selected and the subsequent code executed. This test is called a *decision point.*

Block $B4$ has two predecessors, meaning that the entry point into $B4$ can be from either $B2$ or $B3$. $B4$ is called a *join node* because it joins two (or possibly more) different paths.

**Dominance** A node $N_i$ *dominates* a node $N_j$ if every path from the entry node to $N_j$ goes through $N_i$. When $N_i$ dominates $N_j$, the relationship is described using the the notation $N_i \prec_d N_j$.

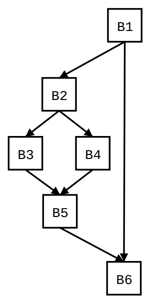
In Figure 2.2(a), $B1$ dominates $B2$, $B3$, $B4$, $B5$, and $B6$ because the only way to get to each of those nodes is through $B1$. On the other hand, $B3$ does not dominate $B5$, because there is a path from the entry node to $B6$ that does not contain $B3$. Similarly, $B4$ does not dominate $B5$.

A node $N_i$ strictly dominates a node $N_j$ if $N_i$ dominates $N_j$ and $N_i \neq N_j$. A node $N_i$ immediately dominates a node $N_j$ if $N_i$ strictly dominates $N_j$ and there is no other node $N_k$ such that $N_i$ dominates $N_k$ and $N_k$ dominates $N_j$.

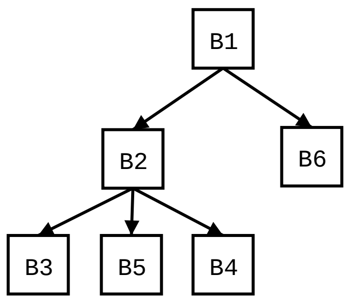
**Dominator Tree** A dominator tree represents dominance relationships. The root of the dominator tree is the entry node. There is an edge from a node $N_i$ to a node $N_j$ in the dominator tree if and only if $N_i$ immediately dominates $N_j$. Every node in the dominator tree only dominates its decedents in the tree.

Figure 2.2(b) presents the dominator tree for the Control Flow Graph of Figure 2.2(a).

We refer to the sub-tree rooted at a node $N_i$ as the dominator tree of node $N_i$. The dominator tree for each CFG node can be efficiently represented using bit vectors. Every node in the CFG has an associated bit in the bit vector. Consider the bit vector that represents the dominator tree for a node $N_i$. The number of the CFG node provides the index for the associated bit in the bit vector. A 0 bit means that the corresponding node in the vector is not dominated by $N_i$. A 1 bit means that the corresponding node is dominated by $N_i$.



(a)



(b)

Figure 2.2: Example Control Flow Graph and Dominator Tree

For example, the dominator tree for $B2$ in Figure 2.2(a) is 011110. The dominator tree for $B6$ is 000001 because $B6$ only dominates itself.

**Postdominance** A node $N_j$ *postdominates* a node $N_i$ if and only if every path from $N_i$ to the exit node goes through $N_j$. When $N_j$ postdominates $N_i$, the relationship is described using the notation $N_j \prec_{pd} N_i$.

In Figure 2.2(a), $B6$ postdominates $B1$, $B2$, $B3$, $B4$ and $B5$ because the only way to get from each of those nodes to the exit node is through $B6$. On the other hand, $B3$ does not postdominate $B2$ because there is a path from $B2$ to $B5$ that does not contain $B3$. Similarly, $B4$ does not postdominate $B2$.

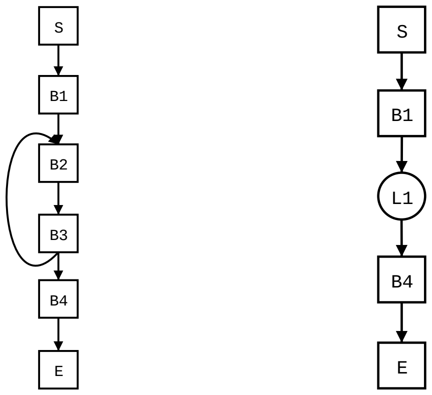
The postdominance relationship can also be represented in a tree structure. The root of the tree is the exit node. Every node in the tree only post dominates its decedents in the tree. Postdominator trees for specific CFG nodes can also be represented using bit vectors. The representation is exactly the same as for dominator trees, except that the postdominance relationship is represented instead.

The postdominator tree for $B6$ in Figure 2.2(a) is 111111. The postdominator tree for $B5$ is 011110. Likewise, the postdominator tree for $B1$ is 100000 because $B1$ only postdominates itself.

**Regions** A region is a collection of basic blocks $B$, in which a header node dominates all the other nodes in the region. A consequence of this definition is that any edge from a node not in the region to a node in the region must be incident in the header.

A *Single Entry, Single Exit*(SESE) region (also known as a *Contiguous* region) is a region in which control can only enter through a single entry node and leave through a single exit node. Thus, when traversing the nodes in a SESE region, the following two conditions must be met: (*i*) from the entry node, all nodes in the region can be reached, (*ii*) during the traversal, no node not in the region is reached before the exit node.

**Extended Control Flow Graph** We define an Extended Control Flow Graph (ECFG) as a directed multigraph. The nodes in the ECFG are either basic blocks or regions. An edge in the ECFG from node $N_i$ to node $N_j$ indicates that control can be transferred from $N_i$ to $N_j$. A node representing a region is itself an ECFG, representing the control flow within the region.



(a) CFG containing a region (b) Resulting ECFG

Figure 2.3: Example ECFG

Figure 2.3 gives an example CFG and the associated ECFG. In Figure 2.3(a), blocks B2 and B3 form a region because of the back-edge from B3 to B2. Figure 2.3(b) shows the region (L1) created from B2 and B3. The contents of L1 is a graph, containing B2 and B3, connected by a directed edge from B2 to B3 as well as the back-edge from B3 to B2. While not shown in this example, the contents of a region can contain arbitrary control flow between basic blocks as well as other regions.

In this text, unless otherwise specified, we assume that nodes in the Extended Control Flow Graph are numbered starting at 0. In an ECFG diagram, square nodes represent basic blocks and circular/oval nodes represent regions.

**Data Dependencies** Two statements $S_i$ and $S_j$ that access the same variable (or memory location) have a *data dependence* between them.

There are four kinds of data dependencies that can occur, based on the type of access that the statements are performing. Figure 2.4 illustrates the four types of data dependencies between two statements.

1. Input dependence occurs when $S_i$ and $S_j$ both read from the same location (Figure 2.4(a)).
2. Output dependence occurs when two statements both write to the same location (Figure 2.4(b)). An output dependence from statement $S_i$ to $S_j$ is written as $S_i \delta^o S_j$.
3. Flow dependence occurs when $S_i$ writes to a location and a subsequent statement $S_j$ reads from the same location (Figure 2.4(c)). A Flow dependence from statement $S_i$ to $S_j$ is written as $S_i \delta^f S_j$.
4. Anti-dependence occurs when $S_i$ reads from a location and a subsequent statement $S_j$ writes to the same location (Figure 2.4(d)). An anti-dependence from statement $S_i$ to $S_j$ is written as $S_i \delta^a S_j$.

```
Si:  =  X      Si: X  =       Si: X  =         Si:    =  X
        .             .               .                  .
        .             .               .                  .
        .             .               .                  .
Sj:  =  X      Sj: X  =        Sj:    =  X     Sj: X  =

   (a)            (b)             (c)              (d)
```

Figure 2.4: Types of Data Dependencies

**Data Dependence Graph** A Data Dependence Graph (DDG) is a directed multigraph that represents data dependencies between statements. A node in a DDG corresponds to either a single statement or a group of statements. Edges in the DDG represent data dependencies between nodes. A directed edge from node $N_i$ to $N_j$ represents a data dependence from $N_i$ to $N_j$. Edges in the DDG can have attributes that represent the type of data dependence the edge represents and a distance/direction vector.

**Normalized Loop** A normalized loop is a countable loop that has been modified to start at a specific lower bound and iterate, by increments of 1, to a specific upper bound.

**Data Dependence in a loop** When data dependencies occur between two statements in a loop, they are generally discussed in terms of the distance of the iteration space containing the statements. That is, when we refer to the distance of a data dependence in a loop, we are referring to the number of loop iterations that the dependence crosses.

**Loop Invariant Expression** A loop invariant expression is an expression whose result is not based on the iteration of a loop. That is, the expression's value does not change throughout the execution of the loop.

**Induction Variable** An induction variable is any scalar variable for which the value of the variable on a given iteration of the loop is a function of loop invariants and the iteration number $(0, \ldots, n)$ [36].

During loop normalization in TPO, two types of induction variables are created: *controlling* induction variable and *derived* induction variable. The controlling induction variable is the variable that controls the execution of the loop. The derived induction variable is any variable whose value is a function of the controlling induction variable.

**Lower bound** The lower bound of a normalized loop is the minimum value that the controlling induction variable can be assigned to during loop execution. If the controlling induction variable is less than the lower bound, the loop body does not execute.

**Upper bound** The upper bound of a normalized loop is the maximum value that the controlling induction variable can be assigned to during the execution of the loop body. Whenever the controlling induction variable is greater than the upper bound, the loop body does not execute.

**Countable loop** A loop is countable when the upper bound and increment of the loop do not depend on values that are computed in the loop body.

**Iteration count** The iteration count of a loop is the number of times that the loop body executes. For instance, a countable loop in the C language, with an upper bound $U$, a lower bound $L$ and an induction variable that is incremented by 1 every iteration has an iteration count of $U - L + 1$.

**Nesting level** The nesting level of a loop is equal to the number of loops that enclose it. Loops are numbered from the outermost to the innermost nesting level, starting at 0.

Given two loops, $L_i$ and $L_j$, if $L_i$ dominates $L_j$, or if $L_j$ postdominates $L_i$, then $L_i$ and $L_j$ must be at the same nesting level. That is, dominance relationships are only reported between loops that are at the same nesting level.

**Perfect loop nest** Loops are perfectly nested if all statements, excluding statements specific to the control of the loop, are located in the innermost loop. Figure 2.5(a) gives an example of a perfect loop nest. Figure 2.5(b) gives an example of a loop nest that is not perfectly nested.

```
for (i=0; i<N; i++)
{
  for (j=0; j<M; j++)
  {
    for (k=0; k<P; k++)
    {
      A[i][j][k] += B[i][j] - C[j][k];
    }
  }
}
```

(a) Perfectly Nested Loops

```
for (i=0; i<N; i++)
{
  B[i] += C[i];

  for (j=0; j<M; j++)
  {
    B[i][j] += C[i][j];

    for (k=0; k<P; k++)
    {
      A[i][j][k] += B[i][j] - C[j][k];
    }
  }
}
```

(b) Imperfectly Nested Loops

Figure 2.5: Nested Loops Example

**Ordering of Loops** Whenever there is a dominance relation between two loops $L_i$ and $L_j$, such that $L_i$ dominates $L_j$, we refer to $L_i$ as the first loop and $L_j$ as the second loop.

# Chapter 3

# Overview of TPO

The Toronto Portable Optimizer (TPO) originated as an interprocedural optimizer for RS/6000 machines. It has evolved since its conception into a compiler component that analyzes and transforms programs at both the procedure and program unit (interprocedural) level. The purpose of TPO is to reduce the execution time and memory requirements of the generated code. TPO is both machine and source language independent through the use of W-code as an intermediate language. W-code is a well established interface between IBM compiler components.

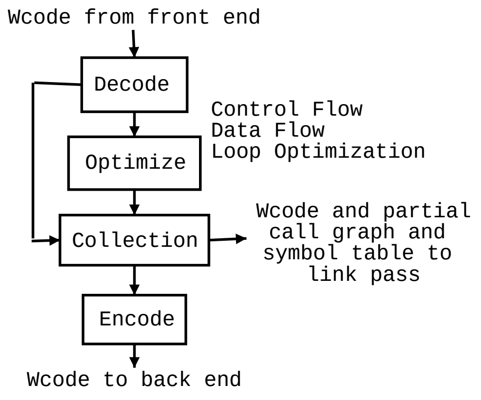


Figure 3.1: High Level Control Flow Through TPO

TPO can perform both intraprocedural and interprocedural analysis. Intraprocedural analysis is performed on individual files at compile time. Interprocedural and intraprocedural analysis is performed on an entire program at link time. Figure 3.1 shows the high level flow of control through TPO. At compile time, input into TPO is provided from the compiler front end. At link time, input into TPO is provided from object files and libraries, some of which may have been compiled by TPO. Object files compiled by TPO are supplemented by W-code and a partial call graph and symbol table.

The three main groups of optimizations performed in TPO are control flow optimizations, data flow optimizations, and loop optimizations. We focus on the loop optimizations. While these optimizations can be performed at both compile time (intraprocedural scope) and link time (interprocedural scope) the focus is on the compile time pass. The future work section discusses applying the loop fusion transformations during the link pass.

## 3.1 Loop Optimizations in TPO

Loops are often a source of performance improvement opportunities because of the repeated execution of statements. Loop optimization in TPO has several objectives including: reduction of path length in the loop body, improvement of memory reference locality to improve the usage of the memory hierarchy (*i.e.*, more efficient use of data caches) restructuring of loops for shared memory parallel execution and the exploitation of vector libraries.

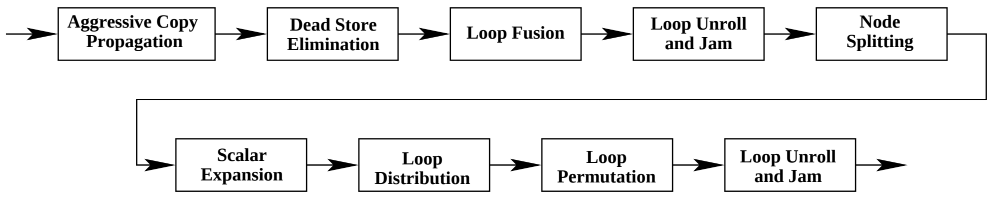


Figure 3.2: Loop Optimizations in TPO

The execution sequence of the major loop optimizations performed in TPO is seen in Figure 3.2. The following sections discuss these optimizations, providing examples to illustrate the code transformations that they perform and, where applicable, how they can improve the performance of loops in a program.

### 3.1.1 Loop Normalization

Loop normalization modifies a countable loop to start at a specific lower bound and iterate, by increments of 1, to a specific upper bound. In TPO, normalized loops have a lower bound of 0. The upper bound is computed based on the original lower bound, upper bound, and increment of the loop to ensure that the loop iterates the correct number of times. Uses of the induction variable in the loop body are modified to maintain consistency with the original loop. Figure 3.3(a) shows an example of an un-normalized loop. The normalized version of the loop is seen in Figure 3.3(b).

```
for i=10,n
  A[i] = A[i+2] * 2
  B[i+1] = A[i-1] + 3
endfor
```

(a) Un-normalized Loop

```
for i=0,n-10
  A[i+10] = A[i+12] * 2
  B[i+11] = A[i-9] + 3
endfor
```

(b) Loop after normalization

Figure 3.3: Loop Normalization Example

Whenever possible, the compiler creates *bump normalized* loops. A loop is bump normalized if its first iteration is 0 and the induction variable is incremented by 1 in every iteration of the loop. Loops whose induction variables increment by 1 every iteration are described as having a *stride* of 1. A loop that is not countable cannot be bump normalized.

### 3.1.2 Aggressive Copy Propagation

Copy propagation identifies statements of the form $a =< expr >$. It then replaces the use of $a$ with $< expr >$ in subsequent statements (as long as $a$ is not redefined). Typical copy propagation algorithms to not move code inside a loop body to prevent enlargement of the dynamic path length of the loop body. Aggressive copy propagation, however, does move statements inside a loop body to create perfectly nested loops. Figure 3.4 demonstrates the aggressive copy propagation performed in TPO. Figure 3.4(a) shows the initial code and Figure 3.4(b) shows the code after aggressive copy propagation. Aggressive copy propagation enables other loop transformations. The computation of `x * y` can be later moved out of the inner loop.

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

(a) Original Code

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

(b) Code after aggressive copy propagation

Figure 3.4: Aggressive Copy Propagation Example

### 3.1.3 Dead Store Elimination

Dead store elimination identifies assignments to variables that are never used and thus not necessary. This optimization is used with aggressive copy propagation to create perfect loop nests. Figure 3.5(a) shows the code created by aggressive copy propagation in the previous section. We see that the assignment to s before the second loop is not necessary because s is no longer used anywhere. Thus, it can be removed, creating the code seen in Figure 3.5(b).

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

(a) Code after aggressive copy propagation

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

(b) Code after dead store elimination

Figure 3.5: Dead Store Elimination Example

Figure 3.5 shows that the combination of aggressive copy propagation followed by dead store elimination has created a perfectly nested loop. While this loop does need to compute the value of `x * y` in every iteration, the perfectly nested loop can be advantageous for both loop interchange and loop unroll and jam. Furthermore, the computation of `x * y` is loop invariant code and thus may be moved back outside the loop after the loop optimizer has completed the loop transformations.

### 3.1.4 Loop Unrolling

Loop unrolling duplicates the body of a loop several times and modifies the loop control so that it executes fewer iterations. The number of copies of the loop body created is called the *unrolling factor*. If the upper bound of the loop is not a multiple of the unrolling factor, a loop residual must be added after the loop. This residual contains the remaining iterations of the loop body.

Loop unrolling is used to reduce the overhead of executing a loop. For example, given a loop $L_j$ that iterates $n$ times, the exit test of the loop (latch condition) must be executed $n$ times. If the loop is unrolled by a factor of 2, the exit test is only executed $n/2$ times. Loop unrolling can also benefit other optimizations, including common-subexpression elimination and instruction scheduling. Common-subexpression elimination identifies repetitive computations of the same expression and replaces the computation with the use of a stored value. Instruction sequencing determines the order of the instructions in memory when the program is being executed. On the other hand, loop unrolling results in more code, which can impact the performance of the instruction cache.

```
for (i=0; i < n; i++)
{
  A[i] = A[i] + B[i];
}
```

(a) Original Loop

```
for (i=0; i < n; i+=4)
{
  A[i] = A[i] + B[i];
  A[i+1] = A[i+1] + B[i+1];
  A[i+2] = A[i+2] + B[i+2];
  A[i+3] = A[i+3] + B[i+3];
}
```

(b) Loop unrolled by a factor of 4

Figure 3.6: Loop Unrolling Example

Figure 3.6(a) shows a simple loop. Figure 3.6(b) shows the loop that is generated by unrolling by a factor of 4. Note that the increment of the induction variable in Figure 3.6 has been modified to increase by 4 for every iteration of the loop.

### 3.1.5 Loop Unroll and Jam

Loop unroll and jam is an optimization performed to increase data reuse within iterations of a loop. It operates on loop nests. The transformation can be visualized in two stages: (*i*) unroll the outer loop, creating multiple copies of the inner loop and (*ii*) fuse the copies of the inner loop. Figure 3.7 shows an example loop before unrolling (a) and after unrolling by 2 (b). Figure 3.8(b) shows the final loop created after the two inner loops are jammed together.

```
for (i=0; i < n; i++)
{
  for (j=0; j < m; j++)
  {
    A[i] = A[i] + B[j];
  }
}
```

(a) Original Loop

```
for (i=0; i < n; i+=2)
{
  for (j=0; j < m; j++)
  {
    A[i] = A[i] + B[j];
  }
  for (j=0; j < m; j++)
  {
    A[i+1] = A[i+1] + B[j];
  }
}
```

(b) After outer loop unrolled by 2

Figure 3.7: Loop Unroll and Jam Example

```
for (i=0; i < n; i+=2;)
{
  for (j=0; j < m; j++)
  {
    A[i] = A[i] + B[j];
  }
  for (j=0; j < m; j++)
  {
    A[i+1] = A[i+1] + B[j];
  }
}
```

```
for (i=0; i < n; i+=2;)
{
  for (j=0; j < m; j++)
  {
    A[i] = A[i] + B[j];
    A[i+1] = A[i+1] + B[j];
  }
}
```

(a) Outer loop unrolled by 2 (b) After jamming inner loops

Figure 3.8: Loop Unroll and Jam Example

### 3.1.6 Node Splitting

TPO allows complex statements in the code representation. This is done to keep the size of the Data Dependence Graphs manageable. Each complex statement is an aggregate node in the DDG. Since it represents complex statements, an aggregate node may participate in multiple dependence relations. Node splitting separates a complex statement into two or more simpler statements. Each of these simpler statements participates in a disjoint dependence relation. In some cases, node splitting allows loop distribution (see Figure 3.2) to distribute two portions of a statement into separate loop nests. For example, if one part of a complex statement has a self-dependence, splitting the statement into two simpler statements allows distribution to put the two statements into different loops, one that is parallelizable and one that is not.

### 3.1.7 Loop Distribution

Loop distribution focuses on splitting a single loop nest into multiple loop nests. In TPO, loop distribution decides which statements should remain together in a loop nest and which statements should be moved to a separate loop. The resulting loops are formed to make optimal use of superscalar execution units, registers, and caches.

The loop distributor begins by building a DDG for a given loop nest. The DDG contains both single statement nodes and aggregate nodes. Aggregate nodes and single statement nodes that form strongly connected components in the DDG are grouped together to form $\pi$-nodes, named after the definition by Kuck [21]. For example, nodes representing loops in the ECFG form $\pi$-nodes in the DDG.

Degenerate $\pi$-nodes are formed from the remaining DDG nodes that are not part of any strongly connected component. Therefore, a $\pi$-node may contain a single statement or an arbitrarily complex portion of the code. $\pi$-nodes are the units that the distributor works with.

There are several characteristics of $\pi$-nodes that are relevant to the loop distribution algorithm. These include: register requirements,[1] load/store usage, number of floating point and fixed point operations executed and the number of prefetchable linear data streams.[2] The distributor also takes into consideration whether the code in a $\pi$-node is self-dependent or not. A $\pi$-node that is not dependent on itself is parallelizable and should be grouped only with other non-self-dependent $\pi$-nodes.

Once the $\pi$-nodes are formed, the distributor creates an affinity graph. This is an undirected weighted graph whose nodes correspond to $\pi$-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 to distribute nodes in the affinity graph. Nodes are gathered in decreasing order of desirability. The decision about grouping two $\pi$-nodes is based not only on the affinity in the graph, but also on whether grouping would satisfy data dependencies and whether grouping is desirable based on node attributes. For example, if the aggregation of two $\pi$-nodes would exceed the capacity of the existing prefetching streams, the nodes are usually not grouped together. Similarly, self-dependent (non-parallelizable) nodes are usually not grouped with non-self-dependent (parallelizable) nodes. Decisions about aggregating nodes are conditioned to the potential increase in data reuse.

Figure 3.9 gives a sample loop nest before distribution (a) and after the loop nest is distributed (b). The loop distributor creates two separate loop nests. The first loop computes all values of a[i]. This computation is moved into a separate loop to create a perfect loop nest which can benefit later loop optimizations. The second loop nest computes the values of c[i][j] and e[i][j]. These two computations remain together because there is data reuse (the value of a[i]) in both computations.

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

[2]The number of prefetchable linear data 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.

```
for (i=0; i<n; i++)
{
  a[i] = b[i] * r;
  for (j=1; j<n; j++)
  {
    c[i][j] = d[i][j-1] / a[i];
    e[i][j] = e[i][j] * a[i];
  }
}
```

(a) Original loop nest

```
for (i=0; i<n; i++)
{
  a[i] = b[i] * r;
}
for (i=0; i<n; i++)
{
  for (j=1; j<n; j++)
  {
    c[i][j] = d[i][j-1] / a[i];
    e[i][j] = e[i][j] * a[i];
  }
}
```

(b) Loops created by distribution

Figure 3.9: Loop Distribution Example

### 3.1.8 Loop Permutation

If a loop nest is perfectly nested, in many cases it is possible to reorder the loops in the nest without affecting program results. This reordering might improve the locality of reference of arrays used in the loop (*i.e.,* reordered memory accesses make better use of the data caches). It can also be used to expose larger units of work to parallel processors.[3]

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

(a) Original loop nest

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

(b) Loop nest after permutation

Figure 3.10: Loop Permutation Example

Figure 3.10 shows an example of loop permutation. In Figure 3.10(a) the inner loop is modifying the induction variable $j$ used to access the first dimension of the arrays `a` and `b`. This access order may lead to poor cache performance because in C arrays are organized in memory in a row first order. In Figure 3.10(b) the loops are reordered so that the inner loop is modifying the induction variable accessing the second dimension of the arrays. When the size of the arrays is large, this loop permutation results in improved cache performance and in a reduction of the execution time of the program.

[3]Increasing parallelism is not currently considered when performing loop permutation in TPO.

### 3.1.9 Index Set Splitting

Loops that contain control flow can be split into multiple loops containing no control flow if the condition controlling the control flow is variant with the loop. Figure 3.11(a) gives an example loop that contains control flow. The condition surrounding the control flow compares the induction variable `i` with the value of `m`. Thus, the condition is variant with the loop. Figure 3.11(b) shows how the original loop is split into two separate loops.

```
for (i=0; i < n; i++)
{
  if (i < m)
    a[i] = a[i] * b[i];
  else
    a[i] = a[i] * a[i];
}
```

(a) Original loop

```
for (i=0; i < m; i++)
{
  a[i] = a[i] * b[i];
}
for (i=m; i<n; i++)
{
  a[i] = a[i] * a[i];
}
```

(b) Loops generated by index set splitting

Figure 3.11: Index Set Splitting Example

The resulting loops are guaranteed to produce the same result as the original loop. The relationship between `n` and `m` does not need to be known at compile time. If $\texttt{n} \leq \texttt{m}$, at most one path of the conditional statement is executed. In this case, the second loop created in Figure 3.11(b) never executes because the initial condition of this loop is not satisfied.

## 3.2 Loop Structure in TPO

The structure of loops in TPO corresponds to the general description of regions in the definitions section. A loop in the source code is transformed into five distinct parts in TPO, each having its own purpose. Figure 3.12 shows an example loop in C source code (a), and the loop representation within TPO (b).

The first branch in the code of Figure 3.12(b), `s1`, protects the initial execution of the loop. In TPO this branch is called the *guard branch* for the loop. The purpose of the guard branch is to ensure that no code between the guard branch and the branch target (s7) is executed if the original loop would not have been executed at least once. If the compiler can ensure that the test condition always evaluate to true (*i.e.,* the initial value of the induction variable is always less than the upper bound) the guard branch can be removed by later optimizations. However, the

```
for (i=k; i < j; i++)
{
  A[i] = A[i] + 2;
  B[i] = B[i-1] * 3;
}
```

(a) Source Loop

```
s1: if (k >= j) goto L1
s2: i = 0
s3: L2:
s4:   A[i+k] = A[i+k] + 2;
s5:   B[i+k] = B[i+k-1] * 3;
s6:   i = i + 1;
s7:   if (i < j-k) goto L2
s8: L1:
```

(b) TPO Representation

Figure 3.12: Example Loop

guard branch is always present while loop optimizations are performed. All loops in TPO are changed into *do* loops in which the exit condition is tested at the end of the iteration, instead of the beginning of the iteration. This transformation is always legal because the initial execution of the loop body is protected by the guard branch.

The initialization of the induction variable (s2) is placed under the guard branch to prevent unnecessary execution of instructions when the loop does not execute.

The body of the loop remains the same; the assignments to the two arrays A and B remain unchanged. The *latch branch* (s7) tests whether the induction variable is within the original bounds. As long as it is, control continues to jump back to the first statement in the loop body. As soon as the induction variable has passed the upper bound, control falls through to the second label.

Figure 3.13 gives a general overview of the parts of a loop structure in TPO. The *guard block* is the block immediately preceding entry into a loop. The guard branch protects the execution of the loop and is the last statement in the guard block. If the guard branch evaluates to true, a jump instruction is executed and control jumps directly to the exit block, skipping the loop entirely. If the guard branch evaluates to false, all blocks representing the loop are executed at least once.

The *loop prolog* is located immediately below the guard branch of the loop, before the loop body. Since it is not included in the loop body, only instructions that are invariant to the loop are located in the loop prolog. The loop prolog normally contains the initialization of the induction variables used in the loop body, as well as any loop invariant code that has been moved out of the loop by the *loop invariant code motion* optimization.

The *loop body* contains all of the loop variant statements that were in the orig-

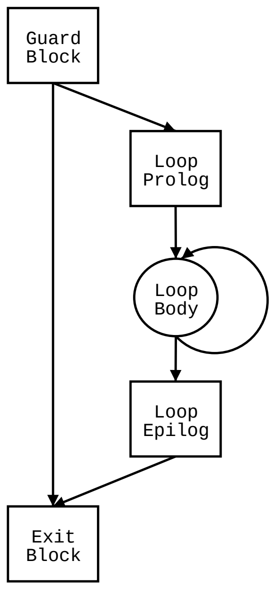


Figure 3.13: Loop Structure in TPO

inal loop plus any additional statements that were moved into the loop body by optimizations.[4] Note, in Figure 3.13, that the loop body can contain an arbitrary number of basic blocks with arbitrary control flow between them. The last statement of the last basic block in the loop body is called the *Latch Branch*. It is a test to determine if the current value of the induction variable is within the bounds set by the original loop. If it is, the back edge branch is taken, resulting in the execution of another iteration of the loop. If the value is not within the bounds, the back edge is not taken and control flows to the loop epilog.

The *loop epilog* block contains statements to be executed after the loop body has executed but only if the loop body was executed. Like the loop prolog, the loop epilog only contains statements that are invariant to the loop body. For instance, any statements that were identified as loop invariant but could not be put into the loop prolog due to data dependencies would be placed in the loop epilog. It is possible that the loop epilog remains empty in which case it is removed by a later optimization.

The beginning of the *exit block* contains the target of the guard branch. This is the first block executed if the guard condition protecting the loop evaluates to true and the jump is taken.

[4]Aggressive copy propagation is one example of an optimization that can move statements into a loop body.

# Chapter 4

# Loop Fusion

## 4.1 Requirements for Loop Fusion

There are two different sets of conditions that need to be checked during loop fusion. First, each loop needs to be checked individually to determine if it can be fused with other loops. Second, pairs of loops are checked to verify that specific conditions are met, or can be met through code transformations. The algorithms presented here focus on making the latter conditions satisfiable.

### 4.1.1 Fusion Candidates

The first step of loop fusion is to create a set of *Fusion Candidates*. This set is created from all of the loops in a given procedure. A series of checks is performed on each loop to verify that it can be fused with other loops. Every fusion candidate must pass all of the following criteria:

1. The loop must be *bump normalized*. If loop normalization was unable to normalize a loop, the loop cannot be fused.

2. The loop must have a single entry point and a single exit point. It must not be possible to execute any statement in the loop body without going through the entry point. Similarly, once inside the loop, it must not be possible to execute any statement not in the loop without first going through the exit point. For example, loops containing break statements are not candidates for fusion. This condition, combined with the first condition ensures that the number of times the loop executes is based solely on the upper bound of the induction variable.

3. Loops marked as parallel (either by the programmer or by the compiler) are not candidates for fusion. This is because the resulting fused loop may not be parallelizable due to loop-carried dependencies introduced during fusion. It is possible to fuse parallel loops with other parallel loops provided the resulting loop does not contain any loop carried dependencies (*i.e.,* the resulting loop is also parallel). This case is not currently considered when performing loop fusion.

4. The loop cannot contain any volatile memory references.

5. The loop cannot contain any statements (including some function calls) that have side effects.

   Side effects are externally visible changes to the environment *e.g.,* Input/Output.

   While it may be possible to move such statements correctly, the compiler treats them conservatively and thus does not move them.

   A statement that invokes a function that cannot be proven to not have side effects is considered to have side effects. In other words, if the compiler cannot prove that the function call does not have side effects, it conservatively assumes that it does.

6. The loop must be guarded.

7. The loop must represent a contiguous region. This is also used to ensure that the loop does not have any side entrances or side exits.

### 4.1.2 Fusion Criteria

In order for a pair of fusion candidates, $L_j$ and $L_k$, that satisfy all the conditions above to be fused, the following four conditions must be met:

1. $L_j$ and $L_k$ must be *adjacent*, *i.e.,* there cannot be a statement $S_i$ such that $S_i$ executes after $L_j$ and before $L_k$.

2. $L_j$ and $L_k$ must be *conforming*, *i.e.,* both $L_j$ and $L_k$ iterate the same number of times.

3. $L_j$ and $L_k$ must be *Control Flow Equivalent*, *i.e.,* when $L_j$ executes, $L_k$ also executes or when $L_k$ executes, $L_j$ also executes.

4. There cannot be any negative distance dependencies between $L_j$ and $L_k$. A negative distance dependence occurs between $L_j$ and $L_k$, where $L_j$ dominates $L_k$, when at iteration $m$, $L_k$ uses a value that is computed by $L_j$ at a future iteration, $m + n$ (where $n > 0$).

These four conditions ensure that the program execution produces the same results after two loops are fused.

The following sections present algorithms that are applied when one of the above conditions is not met. The goal of the algorithms is to manipulate the control flow of the program to make the condition satisfiable.

## 4.2 Adjacency

If there is code between two loops they cannot be fused. However, in many cases this code can be moved to another place to enable loop fusion.

The extended control flow graph is used to determine if two loops are adjacent. Let $L_i$ and $L_j$ be two nodes in the ECFG, each representing a loop, such that $L_i$ dominates $L_j$. One of the conditions for a loop to be a fusion candidate is that it must have only one entry point and one exit point. Thus, the following condition always determines if the two loops are adjacent: if the sole immediate successor of $L_i$ is $L_j$, and the first statement in $L_j$ is the guard branch of $L_j$, the two loops are adjacent.

When two loops are not adjacent, there is *Intervening Code* between them. Movement of the intervening code to other places can be attempted using the following algorithm.

### 4.2.1 Identifying Intervening Code

The dominator and postdominator trees are used to identify the ECFG nodes that intervene between the two fusion candidates. We use one bit vector to record the nodes that are strictly dominated by the first loop and a second bit vector to record the nodes strictly dominated by the second loop. A simple bit vector subtraction produces a record of intervening nodes.[1]

In the ECFG shown in Figure 4.1 two fusion candidates, $A$ and $B$, are separated by intervening code. Each ECFG node is annotated with a unique number for

[1]The bit vector subtraction is performed using an and-with-compliment operation on the two bit vectors.

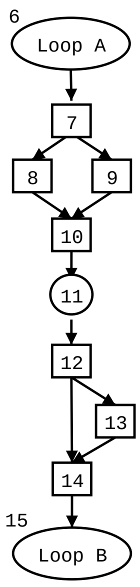


Figure 4.1: Intervening Code Example

identification. Lets assume that this ECFG contains 20 nodes, numbered 0 through 19. Nodes 0 to 5 are located *above* `Loop A` (*i.e.,* `Loop A` postdominates nodes 0 to 5). Nodes 16 to 19 are located *below* `Loop B` (*i.e.,* `Loop B` dominates nodes 16 to 19).

The strict dominator tree for `Loop A`, is represented as the following bit vector:

| 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

The strict dominator tree for `Loop B` is represented as the following bit vector:

| 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

The difference between the two bit vectors results in the following bit vector:

| 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

The resulting bit vector yields nodes 7, 8, 9, 10, 11, 12, 13, 14, and 15 as intervening nodes. Notice that node 15 contains `Loop B`. We include node 15 in the list of intervening nodes because there could be statements in node 15 that must be moved before fusion can take place. These statements in node 15 would appear before the guard branch of `Loop B`.

### 4.2.2 Data Dependence Graph Construction

The algorithm that attempts to move intervening code must analyze (*i*) dependencies between the loop bodies and the intervening code, and (*ii*) dependencies among the statements of the intervening code. The movement of intervening code cannot violate any data dependencies between the loops and the moved statements. In other words, a statement $S$ cannot be moved above `Loop A` if there is a flow, output, or anti-dependence between $S$ and `Loop A`. A similar argument can be made if trying to move $S$ below `Loop B`.

A group of intervening statements can be split by moving some of the statements above the first loop and the remaining statements below the second loop. The splitting of the intervening code cannot violate any data dependencies among the statements moved. For example, if two intervening statements $S_i$ and $S_j$ are originally executed in the order $S_i$ followed by $S_j$ and there is a dependence between $S_i$ and $S_j$, the statements cannot be reordered such that $S_j$ is executed before $S_i$.

Intervening statements that have side effects cannot be moved. If the intervening code contains any statements with side effects, the algorithm cannot make the two loops adjacent and thus the two loops cannot be fused.

Once all of the intervening ECFG nodes between the two loops have been identified, a Data Dependence Graph (DDG) is built using the statements in the ECFG nodes. The DDG is used to determine which statements can be moved from between the two loops while maintaining the necessary relative ordering of the intervening statements. The DDG contains two types of nodes: *single statement nodes* and *aggregate nodes*. Single statement nodes contain a single statement from the list of intervening ECFG nodes. An aggregate node is formed by a minimal SESE region, *i.e.,* the minimum number of nodes required to form a valid SESE region. All of the basic blocks that form an aggregate node must be moved together.

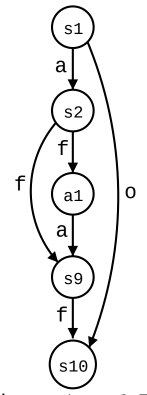
Aggregate nodes are used to make the analysis and the movement of statements easier. An aggregate node represents a collection of statements that, for convenience, must remain together. It would be possible to treat every node (or even every statement) in a region individually. This would allow the region to be split and parts of it to be moved separately. However, it would be necessary to duplicate the conditional branch in every place a member of the region is moved to. Thus, treating the entire region as a single unit results in a simpler analysis and movement of code.

In Figure 4.1, each statement in node 7, except for the branch statement at the end, form a single node in the DDG. The branch statement in node 7, plus all statements in nodes 8 and 9 form an aggregate node. Each statement in node 10 form a single statement node in the DDG. Node 11 is an aggregate node. As with node 7, all statements before the branch statement in node 12 form single nodes. The branch statement in node 12 plus every statement in node 13 form a third aggregate node. Finally, all statements in node 14 form single statement nodes.

Nodes in the DDG are connected using directed edges that represent dependencies between the nodes. An edge from node $N_i$ to $N_j$ in the DDG indicates a data dependency from $N_i$ to $N_j$. Output, flow, and anti-dependencies are monitored because they prevent reordering of statements.

```
s1:  a = b + c;
s2:  c = d + e;
s3:  if k < 10
s4:    f = c * 2;
s5:    g = h + 3;
s6:  else
s7:    g = h + 2;
s8:    f = c * 3;
s9:  h = a + c;
s10: a = h + b;
```

(a) Example code segment



(b) Associated DDG

Figure 4.2: Example Code Fragment and Associated DDG

Figure 4.2(a) gives an example code segment containing 10 statements. The associated DDG is shown in Figure 4.2(b). The nodes in the DDG are labeled with their corresponding statement number, with the exception of `a1` which represents the aggregate node containing `s3`, `s4`, `s5`, `s6`, `s7` and `s8`. The letters `f`, `o` and `a` on the directed edges represent flow, output and anti-dependencies between two nodes in the DDG.

If statements `s3`-`s8` are not treated as a single unit, the conditions `s3` and `s6` must be replicated in every location that the statements `s4`, `s5`, `s7` and `s8` are moved to. For example, if the goal was to move the conditional statement (starting at `s3` and ending at `s8`), Figure 4.3 shows the code generated if the aggregate node `a1` in Figure 4.2(b) is broken. If the aggregate node is not broken, the code cannot be moved.

```
s3:  if k < 10
s5:    g = h + 3;
s6:  else
s7:    g = h + 2;
s1:  a = b + c;
s2:  c = d + e;
s9:  h = a + c;
s10: a = h + b;
s3:  if k < 10
s4:    f = c * 2;
s6:  else
s8:    f = c * 3;
```

Figure 4.3: Splitting and Movement of Aggregate Node

Figure 4.3 demonstrates that while the splitting of the aggregate node allows the code to be moved, it requires the duplication of the test condition. For this reason, aggregate nodes are forced to move as a single unit.

### 4.2.3 Intervening Code Analysis

After the intervening code has been identified, we must determine if the intervening code can be moved from between the two loops. The DDG of the intervening code represents a partial ordering that intervening code statements (or groups of statements) must adhere to. Figure 4.4 provides an algorithm to determine if all intervening statements between two loops, $L_j$ and $L_k$, can be moved from between the loops, thus making them adjacent. The algorithm begins by gathering all statements that are located between $L_j$ and $L_k$ using the technique described in 4.2.1 (Step 1). It then checks to make sure that none of the intervening statements has side effects, and thus is non-movable (Step 2). Step 4 builds a Data Dependence Graph of all of the intervening statements using the methods outlined in Section 4.2.2. A queue data structure is used to collect all of the nodes that can be moved around one of the loops while maintaining the partial ordering of statements provided by the DDG.

The DDG contains at least one node that has no predecessors and at least one node that has no successors. Nodes in the DDG that have no predecessors can be executed before all other nodes in the DDG. Nodes in the DDG that have no successors can be executed after all other nodes in the DDG.

To find the nodes in the DDG that can be moved above the first loop, all of the nodes that have no predecessors are added to the MoveQueue (Step 6). Nodes

```
IsInterveningCodeMovable($L_j, L_k$)
1.  InterveningCodeSet $\leftarrow \{a_x | L_j \prec_d a_x$ and $L_k \prec_{pd} a_x\}$
2.  if any node in InterveningCodeSet is non-movable
3.    return $False$
4.  Build a DDG $G$ of InterveningCodeSet
5.  CanMoveUpSet $\leftarrow \emptyset$
6.  MoveQueue $\leftarrow$ Nodes in $G$ with no predecessors
7.  while MoveQueue $\neq \emptyset$
8.          $N_i \leftarrow$ Head of MoveQueue
9.          if $L_j \not\delta N_i$
10.            CanMoveUpSet $\leftarrow$ CanMoveUpSet $\cup \{N_i\}$
11.            Add each immediate successor of $N_i$ to tail of MoveQueue
12.         endif
13. end while
14. CanMoveDownSet $\leftarrow \emptyset$
15. MoveQueue $\leftarrow$ Nodes in $G$ with no successors
16. while MoveQueue $\neq \emptyset$
17.         $N_i \leftarrow$ Head of MoveQueue
18.         if $N_i \not\delta L_k$
19.            CanMoveDownSet $\leftarrow$ CanMoveDownSet $\cup \{N_i\}$
20.            Add each immediate predecessor of $N_i$ to tail of MoveQueue
21.         endif
22. end while
23. if InterveningCodeSet $-$
      (CanMoveUpSet $\cup$ CanMoveDownSet) $= \emptyset$
24.   return $True$
25. return $False$
```

Figure 4.4: Algorithm to Check if All Intervening Code Can Be Moved.

are then removed from the head of the MoveQueue and data dependence analysis is performed between every statement in the first loop and every statement in the node (Steps 8 and 9). Let's consider the analysis for a node $N_i$. If there are no dependencies between any statement in the first loop and any statement in $N_i$, $N_i \not\delta L_k$, then $N_i$ is added to the CanMoveUpSet and all the immediate successors of $N_i$ in the DDG are added to the MoveQueue (Steps 10 and 11). This continues until the MoveQueue is empty. Once the MoveQueue is empty, all of the DDG nodes that are in the CanMoveUpSet can be moved above the first loop.

To find all of the nodes in the DDG that can be moved below the second loop, all of the nodes that have no successors are added to the MoveQueue (Step 15). Nodes are then removed from the head of the MoveQueue and data dependence analysis is performed between all statements in the node and all statements in the second loop (Steps 17 and 18). Again, consider a node $N_i$ from the head of the MoveDown queue. If there are no data dependencies between any statement in $N_i$, and any statement in second loop, $N_i$ is added to the CanMoveDownSet and all of the immediate predecessors of $N_i$ are added to the MoveQueue (Steps 19 and 20). This repeats until the MoveDown queue is empty. When the MoveQueue is empty, all of the DDG nodes in the CanMoveDownSet can safely be moved below the second loop.

At the end of this analysis, if the collection of nodes in the CanMoveUpSet and the CanMoveDownSet contains all of the nodes that were in the InterveningCodeSet, then all of the intervening code between the two loops can be moved. If all of the code cannot be moved, the two loops cannot be made adjacent and thus cannot be fused.

## 4.3 Conforming Bounds

The bounds of two loops do not conform when the two loops have different iteration counts. This means that the body of one loop executes more times than the body of the other loop. When the difference in iteration counts of two loops cannot be determined at compile time, the compiler assumes that they are different. There are several methods to modify two loops with different iteration counts to make them conform.

### 4.3.1 Loop Peeling

One approach is to use loop peeling to replicate iterations of the loop with the higher execution count. After peeling we have two loops with identical iteration counts and residual code comprised of the peeled iterations. If the iteration count of the first loop is higher than the one of the second loop, peeled iterations are placed before the first loop. If the iteration count of the second loop is higher than the one of the first loop, the peeled iterations are placed after the second loop. This ensures that there is no intervening code placed between the two loops. Figure 4.5 demonstrates loop peeling.

```
for (i=0; i < n; i++)
{
  A[i] = A[i] * 2
}
for (j=0; j < n-2; j++)
{
  A[j] = A[j] + 3
}
```

(a) Original Loops

```
A[0] = A[0] * 2
A[1] = A[1] * 2
for (i=0; i < n-2; i++)
{
  A[i+2] = A[i+2] * 2
}
for (j=0; j < n-2; j++)
{
  A[j] = A[j] + 3
}
```

(b) After Peeling First Loop

Figure 4.5: Loop Peeling Example

Loop peeling has a disadvantage in that peeled iterations of a loop are placed between other loops, thus creating more work for the preparation for fusion. It is always possible to move the peeled iterations around either loop in order to fuse them - if the iterations could not be moved, then the loops could not have been fused.

The peeled iterations may be either laid out sequentially as in Figure 4.5 or grouped together to form another loop. This option allows the amount of code growth to be managed. Consider a case where the upper bounds of two loops differ by a large amount and the loop bodies of the two loops are large. Peeling the iterations would result in the duplication of the loop body many times, resulting in a large amount of code growth. However, putting the peeled statements into a separate loop whose iteration count is the same as the difference in upper bounds minimizes the amount of code growth.

### 4.3.2 Guarding Iterations

An alternative to loop peeling is to guard the body of one of the loops to ensure that it executes the correct number of iterations. Figure 4.6(a) shows a simple example of two loops whose iteration counts differ by $k$, $k \geq 0$. Figure 4.6(b) shows the fused loop created by guarding the iterations.

```
for (i=0; i < n; i++)
{
  A[i] = A[i+2] * 2;
}
for (j=0; j < n-k; j++)
{
  A[j] = A[j] * 3;
}
```

(a) Two loops with different upper bounds

```
for (i=0; i < n; i++)
{
  if (i < n-k)
  {
    A[i] = A[i+2] * 2;
    A[i] = A[i] * 3;
  }
  else
  {
    A[i] = A[i+2] * 2;
  }
}
```

(b) Loops Fused by Iteration Guarding

Figure 4.6: Iteration Guarding Example

The upper bound of the fused loop in Figure 4.6(b) is the maximum of the two original upper bounds. The guard that is placed in the body of the loop determines if the current iteration count is less than the minimum of the original upper bounds. Under the guard is the body of both loops. This guard ensures that the body of one loop does not execute more times in the fused loop than it would have in the original loop. Finally, an else condition is added that contains the remaining iterations of the loop with the larger upper bound. These remaining iterations are called the *residual* of the loop.

The resulting loop is guaranteed to execute the exact number of iterations of each one of the original two loops. Moreover, this method does not introduce intervening code that could create more work for subsequent loop fusion. However, the guard branches introduce control flow into the fused loop body. The addition of control flow to a loop body is generally considered bad practice because it can negatively impact several later optimizations, including software pipelining. However, the Index Set Splitting algorithm, described in Section 3.1.9, removes control flow from a loop body through the creation of several distinct loops. Therefore, the control flow introduced during fusion is eliminated later. Figure 4.7 shows the loops generated after Index Set Splitting for the loop in Figure 4.6(b).

```
for (i=0; i < n; i++)
{
  if (i < n-k)
  {
    A[i] = A[i+2] * 2;
    A[i] = A[i] * 3;
  }
  else
  {
    A[i] = A[i+2] * 2;
  }
}
```

(a) After Iteration Guarding

```
for (i=0; i < n-k; i++)
{
  A[i] = A[i+2] * 2;
  A[i] = A[i] * 3;
}
for (i=n-k; i < n; i++)
{
  A[i] = A[i+2] * 2;
}
```

(b) After Index Set Splitting

Figure 4.7: Index Set Splitting Example

### 4.3.3 Run time Bounds Check

In order for loop peeling and guarding iterations to work, the difference between the upper bounds of the two loops must be known at compile time. It is often the case that this difference cannot be determined at compile time. In this case, it is necessary to add checks at run time to determine which upper bound is greater and to create a fused loop that executes correctly based on these run time values.

Figure 4.8(a) provides an example of two loops whose iteration count differs by an unknown value.[2] The fused loop in Figure 4.8(b) first performs a check to determine the maximum and minimum upper bounds of the two loops. The new loop has an upper bound equivalent to the maximum of the two upper bounds (as it did with guarding iterations in Section 4.3.2). The guard inserted into the loop body has the value of the minimum of the two upper bounds (again, as it did with guarding iterations). Because the relation between the values of `n` and `m` is not known at compile time, an additional check is necessary to determine which loop body should be executed for the residual.

Figure 4.9 shows the algorithm for fusing two loops, $L_j$ and $L_k$ with different bounds. Step 1 of the algorithm creates the upper bound of the new loop, $L_m$ using the maximum of the upper bounds of the two original loops, $\kappa(L_j)$ and $\kappa(L_k)$. Step 2 inserts the first guard branch to limit the iterations of one loop body. If the difference in the upper bounds of $L_j$ and $L_k$ can be determined at compile time an

[2]We assume that the values of `n` and `m` are not known at compile time.

```
for (i=0; i < n; i++)
{
  A[i] = A[i] * 2;
}


for (j=0; j < m; j++)
{
  A[j] = A[j] * 3;
}
```

(a) Two loops with different upper bounds

```
S = max(n,m);
T = min(n,m);
for (i=0; i < S; i++)
{
  if (i < T)
  {
    A[i] = A[i] * 2;
    A[i] = A[i] * 3;
  }
  else
  {
    if (i < n)
    {
      A[i] = A[i] * 2;
    }
    else
    {
      A[i] = A[i] * 3;
    }
  }
}
```

(b) Loops Fused using Run time Bounds Checks

Figure 4.8: Run time Bounds Check Example

```
FuseWithGuard($L_j, L_k$)
1.  Create $L_m$ with upper bound max($\kappa(L_j), \kappa(L_k)$)
2.  Insert Guard Bound for min($\kappa(L_j), \kappa(L_k)$) 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.  if ($\kappa(L_j) > \kappa(L_k)$)
6.    Insert else statement
7.    Copy body of $L_j$ to $L_m$, after else statement
8.  else if ($\kappa(L_j) < \kappa(L_k)$)
9.        Insert else statement
10.       Copy body of $L_k$ to $L_m$, after else statement
11. else
12.     Insert Guard for $\kappa(L_j)$
13.     Copy body of $L_j$ within second guard
14.     Insert else statement
15.     Copy body of $L_k$ within else statement
16. endif
```

Figure 4.9: Algorithm to Fuse Loops Using a Guard Statement

else statement is inserted and the loop body with the higher upper bound is copied under the else statement (Steps 5 to 10). If the difference in upper bounds of $L_j$ and $L_k$ cannot be determined at compile time, two additional guard statements are inserted and the bodies of both loops are copied under the appropriate guard.

```
S = max(n,m);
T = min(n,m);
for (i=0; i < S; i++)
{
  if (i < T)
  {
    A[i] = A[i] * 2;
    A[i] = A[i] * 3;
  }
  else if (i < n)
  {
    A[i] = A[i] * 2;
  }
  else
  {
    A[i] = A[i] * 3;
  }
}
```

(a)

```
S = max(n,m);
T = min(n,m);
for (i=0; i < T; i++)
{
  A[i] = A[i] * 2;
  A[i] = A[i] * 3;
}
for (i=T; i < n; i++)
{
  A[i] = A[i] * 2;
}
for (i=T; i < S; i++)
{
  A[i] = A[i] * 3;
}
```

(b)

Figure 4.10: Run time Bounds Check after Index Set Splitting

The fused loop, shown in Figure 4.8(b), has control flow in it, just as it did in the guarding iterations case. This control flow is later removed by index set splitting. Figure 4.10 shows the loops that are generated by index set splitting for the loop of Figure 4.8(b).

When using run time bounds checks to fuse loops, additional code duplication is required because the residual loop cannot be determined at compile time and thus both loop bodies must be replicated because either could be executed at run time (based on the run time values of the upper bounds). This technique causes a significant amount of code growth (every fused loop will be twice the size of the original two loops). When the two loop bodies are large, the code can quickly grow to sizes that would result in a long compilation time because later optimizations would have to handle a large code analysis. Thus, it is necessary to limit the fusion of loops whose upper bound difference cannot be determined at compile time. This limitation is enforced to prevent an explosion in the size of the code.

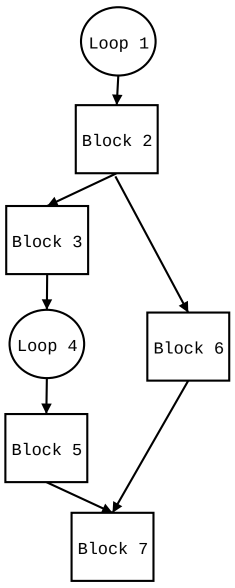
## 4.4 Control Flow Equivalence

Two loops are control flow equivalent when the execution of one loop implies the execution of the other loop. In other words, it is not possible for one of the loops to execute and the other loop not to execute. Control flow equivalence is determined by the dominance relation. Given two loops $L_i$ and $L_j$ in an ECFG, there are four possible dominance relations between $L_i$ and $L_j$, as shown in Figure 4.11.

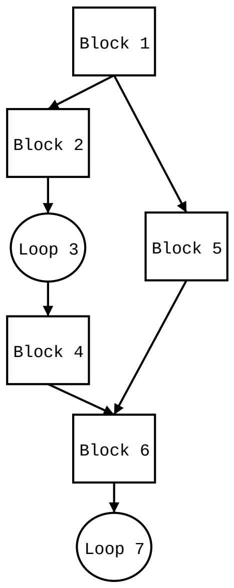
We attempt fusion of two loops if they are control flow equivalent (Figure 4.11(d)), if loop $L_i$ dominates $L_j$ (Figure 4.11(a)), or if loop $L_j$ postdominates loop $L_i$ (Figure 4.11(b)). In all these cases there is a reasonable probability that both loops are executed in the same control path. Loops $L_i$ and $L_j$ may execute in the same control path even when there is no dominance relation between them (Figure 4.11(c)). However, we do not attempt to fuse the loops in this case.

This section presents algorithms for situations depicted in Figure 4.11(a) and 4.11(b). These algorithms copy the loop that always executes into both paths of the conditional statement. This code replication allows the two loops in one path to be fused, while maintaining correct execution for the alternative path.

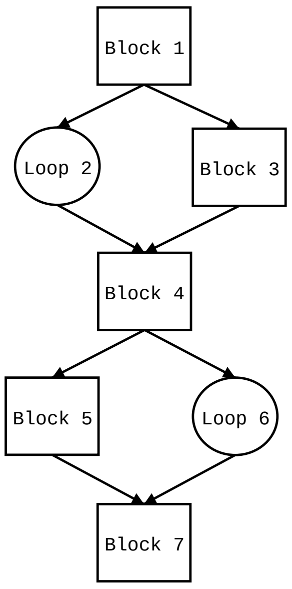
In this paragraph we introduce terminology to refer to the loops and execution paths when a loop $L_i$ dominates a loop $L_j$ (situation depicted in Figure 4.11(a)). Similar terminology can be used when loop $L_j$ postdominates $L_i$ as depicted in Figure 4.11(b). This terminology is used throughout this section. The point at which the control flow path splits into two distinct paths is called the *decision point*. In Figure 4.11(a), the decision point is the last statement in Block 2. The point where two separate control flow paths merge into a single path is the *join point*. In Figure 4.11(a), the join point is the first statement of Block 7. From a decision point, two paths can be taken. The path containing the second loop is the *fusible path*. In Figure 4.11(a), the path on the left, containing `Loop 4` is the fusible path. The path that does not contain the loop is the *non-fusible path*. The path on the right in Figure 4.11(a) is the non-fusible path. The loop that is executed regardless of which path is taken from the decision point is the *unconditionally-executed loop*. In Figure 4.11(a) `Loop 1` is the unconditionally-executed loop. The loop that is executed if the fusible path is taken is the *conditionally-executed loop*. `Loop 4` in Figure 4.11(a) is the conditionally-executed loop.



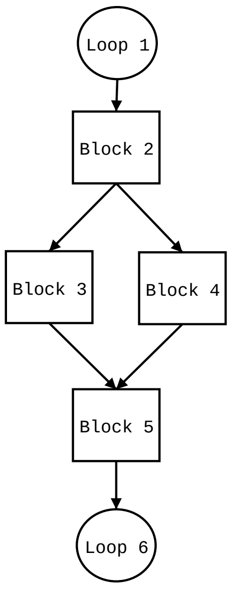
(a) Loop 1 dominates Loop 4



(b) Loop 7 postdominates Loop 3



(c) No dominance relationship between Loop 2 and Loop 6



(d) Loop 1 and Loop 6 are control flow equivalent

Figure 4.11: Possible Control Flow Scenarios

### 4.4.1 Identifying Control Flow Equivalence

The control flow equivalence of two loops is determined using the dominator and the postdominator relationships. Table 4.1 describes the four control flow situations that can occur, in terms of dominance and postdominance. When `Loop A` does not dominate `Loop B` and `Loop B` does not postdominate `Loop A` there is no relationship in the execution of the two loops. An example of this is seen in Figure 4.11(c). While it may be possible for the two loops to reside in the same control flow path, no attempt is made to fuse them. When `Loop A` dominates `Loop B` but `Loop B` does not postdominate `Loop A` the execution of `Loop B` implies that `Loop A` must have executed (Figure 4.11(a)). Similarly, if `Loop B` postdominates `Loop A` but `Loop A` does not dominate `Loop B` then the execution of `Loop A` implies that `Loop B` also executes (Figure 4.11(b)). Finally, if `Loop A` dominates `Loop B` and `Loop B` postdominates `Loop A` then the two loops are control flow equivalent (Figure 4.11(d)).

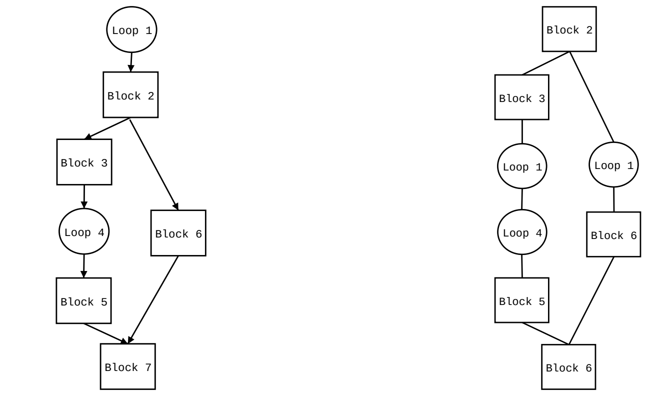
| | A Does Not Dominate B | A Dominates B |
|---|---|---|
| B Does Not Postdominate A | No relation | B $\rightarrow$ A |
| B Postdominates A | A $\rightarrow$ B | A $\equiv$ B |

Table 4.1: Dominance and Postdominance Relationships for Loops

Bit vectors representing the dominator and postdominator trees of the two loops are used to quickly determine the dominance relationship of the two loops. For example, the dominator tree for `Loop 1` in Figure 4.11(a) is represented by the bit vector 1111111. The postdominator tree for `Loop 2` is represented by the bit vector 001100. Thus, `Loop 1` dominates `Loop 4` because the bit vector for `Loop 1` at position 4 has a 1. However, `Loop 4` does not postdominate `Loop 1` because the bit vector for `Loop 4` at position 1 has a 0. Thus, we have the condition where the first loop dominates the second loop but the second loop does not postdominate the first loop.

### 4.4.2 Creating Control Flow Equivalent Loops

Two loops that are not control flow equivalent can be made control flow equivalent by copying the unconditionally-executed loop into the fusible path. This code duplication requires that the unconditionally-executed loop be moved into the non-fusible path to guarantee correct program execution. Consider the two loops `Loop 1` and `Loop 4` in Figure 4.11(a). `Loop 1` is copied immediately above `Loop 4`. It also needs to be copied into the non-fusible path, immediately above Block 6. Note that if there are no nodes in the non-fusible path, `Loop 1` needs to be copied into the non-fusible path, immediately below the decision point and immediately above the join point. Figure 4.12 shows the resulting ECFG after this code transformation.



(a) Original ECFG (b) ECFG After code duplication

Figure 4.12: Creating Control Flow Equivalent Loops

Before the unconditionally executed loop is copied into the fusible path, analysis needs to be performed to ensure that any intervening code between loop $L_j$ and loop $L_k$ can be moved from between the two loops. If loop $L_j$ dominates loop $L_k$ but $L_k$ does not postdominate $L_j$, there can be intervening statements between the unconditionally-executed loop and the decision point (Block 2 in Figure 4.12(a)). There can also be intervening statements on the fusible path between the decision point and the conditionally-executed loop (Block 3 in Figure 4.12(a)). If loop $L_j$ postdominates $L_k$ but $L_k$ does not dominate $L_j$, there can be intervening statements between the conditionally-executed loop and the join point (Block 4 in Figure 4.11(b)). There can also be intervening statements between the join point and the unconditionally-executed loop (Block 6 in Figure 4.11(b)).

Figure 4.13 provides an algorithm to identify the intervening code between two loops $L_j$ and $L_k$ and to determine if the code can be moved. This algorithm requires that loop $L_j$ is the unconditionally-executed loop and loop $L_k$ is the conditionally executed loop. The technique used in Section 4.2.1 cannot be used to gather the intervening nodes because the two loops are not control flow equivalent. Instead, the intervening code is broken up into two parts: (*i*) the unconditionally executed portion and (*ii*) the conditionally executed portion. In order to distinguish between the two parts, the condition node or the join node must be retrieved, depending on the

```
NonCFE_IsInterveningCodeMoveable($L_j, L_k$)
1.  $interveningCode = \emptyset$
2.  if $L_j \prec_d L_k$
3.     $condNode = FindConditionNode(L_j, L_k)$
4.     $interveningCode = interveningCode \cup DominatorTree(L_j) - DominatorTree(condNode)$
5.     $interveningCode = interveningCode \cup DominatorTree(condNode) - DominatorTree(L_k)$
67. else
8.     $joinNode = FindJoinNode(L_j, L_k)$
9.     $interveningCode = interveningCode \cup DominatorTree(L_k) - DominatorTree(joinNode)$
10.    $interveningCode = interveningCode \cup DominatorTree(joinNode) - DominatorTree(L_j)$
11. endif
12. if IsInterveningCodeMovable($interveningCode, movelist, L_j, L_k$)
13.    return $True$
14. else
15.    return $False$
16. endif
```

Figure 4.13: Algorithm to Determine if Intervening Code is Movable

dominance relationship between the two loops. The functions *FindConditionNode* and *FindJoinNode* find the condition node and the join node. Once these nodes have been retrieved, bit vector subtraction is used to identify the two parts of the intervening code. Steps 4 and 10 identify the unconditionally executed intervening code for the two dominance relationships. Steps 5 and 9 identify the conditionally executed intervening code for the two dominance relationships.

At Step 11, *interveningCode* contains a list of statements that reside between loop $L_j$ and loop $L_k$. A modified version of the algorithm in Section 4.2.3 is then called (Step 12). This modified algorithm accepts a list of statements to be moved from between two loops, $L_j$ and $L_k$. If all of the statements in *interveningCode* can be moved from between $L_j$ and $L_k$, the algorithm returns true and the second argument to the algorithm, *moveList* contains all statements that must be moved below $L_k$. If all of the statements cannot be moved, the algorithm returns false and *moveList* contains no statements. If all statements can be moved, the intervening code between loops $L_j$ and $L_k$ does not prevent the loops from being fused.

Figure 4.14 provides the algorithm to move the list of intervening statements that must be moved from between two non-control flow equivalent loops, $L_j$ and $L_k$. This algorithm also requires that loop $L_j$ be the unconditionally executed loop and loop $L_k$ the conditionally executed one. The *moveList* contains all the statements that have a flow, output, or anti-dependence with a statement in loop $L_j$. Any intervening statements for which no dependencies exist, can safely be executed inde-

```
NonCFE_MoveInterveningCode($L_j, L_k, moveList$)
1.  if $L_j \prec_d L_k$
2.     foreach $S_x \in moveList$
3.              Move $S_x$ after $L_k$
4.              if $L_k \not\prec_{pd} S_x$
5.                 Copy $S_x$ into non-fusible path
6.              endif
7      endfor
8.  else if $L_j \prec_{pd} L_k$
9.     foreach $S_x \in moveList$
10.             Move $S_x$ before $L_k$
11              if $L_k \not\prec_d S_x$
12.                Copy $S_x$ into non-fusible path
13.             endif
14.    endfor
15. endif
```

Figure 4.14: Algorithm to Move Intervening Code

pendent of the position of $L_j$. The algorithm begins by identifying which dominance relationship exists between the two loops. The *moveList* may contain statements that are located on the unconditional path as well as the statements located on the fusible path. Movement of these two types of statements must be treated differently. Any statement that is located on the unconditional path (either before the decision point or after the join point depending on the dependence relationship of loops $L_j$ and $L_k$) must be copied into the non-fusible path as well as into the fuseable path. Statements that are located on the fuseable path only need to be moved around $L_k$. If loop $L_j$ dominates loop $L_k$, then any statement $S_x$ that is postdominated by $L_k$, must be located below the decision, *i.e.,* $S_x$ is in the fuseable path. Thus the movement of $S_x$ below $L_k$ does not require any code duplication. A statement $S_x$ that is not postdominated by $L_k$, must be before the decision point, and thus, moving $S_x$ below $L_k$ requires the duplication of $S_x$ into the non-fusible path. The analysis for the case in which loop $L_j$ postdominates loop $L_k$ is similar. Steps 4 to 6 and 11 to 13 identify these scenarios for the different dominance relationships.

When the algorithm is complete, all statements that are dependent on loop $L_j$ have been moved to a point below loop $L_k$. Any statements that remain in the path between $L_j$ and $L_k$ are independent of loop $L_j$. Thus, $L_j$ can safely be copied into the fuseable path above loop $L_k$ and into the beginning of the non-fusible path.

Figure 4.12(b) shows how the unconditionally executed loop is copied into both paths under the decision point.

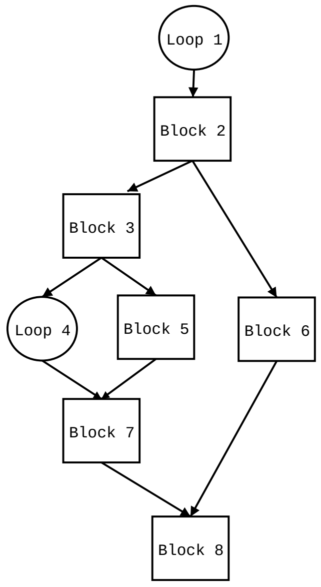


Figure 4.15: Loops Separated by Two Decision Points

The algorithms presented in Figures 4.13 and 4.14 only handle non-control flow equivalent loops that are separated by one decision point. Figure 4.15 shows an example of two loops, `Loop 1` and `Loop 4` that are separated by two decision points. In this example, `Loop 1` must be copied into three control flow paths to preserve program correctness: between `Block 3` and `Loop 4`, between `Block 3` and `Block 5` and between `Block 2` and `Block 6`. Intervening code between `Loop 1` and `Loop 4` also must be copied into multiple paths depending on its location. The analysis required to identify the paths necessary for code duplication (both duplication of the unconditionally executed loop and intervening code) is complex. Furthermore, the duplication of code in all of the necessary paths would increase the code growth and may negatively impact the compilation time for programs. Thus, it was decided to limit the scope of non-control flow equivalent loop fusion to loops separated by a single decision point.

The method to enable the fusion of non-control flow equivalent loops described in this section can result in a large amount of code growth. At least one loop body must be duplicated into two alternative control paths. However, it is likely that additional intervening code must also be duplicated. It would be futile to undertake the rather complex code movement maneuver described here only to find out that the two loops that were made adjacent by it cannot be fused because of one of the other conditions that prevent fusion. Therefore, before any code movement/duplication is performed, we (*i*) verify that each loop is fusible; (*ii*) ensure that we will be able to make the two loops conforming, and (*iii*) analyze the data dependencies between the two loop bodies. If any of these checks fail, the two loops are not made control flow equivalent.

## 4.5 Negative Distance Dependencies

The final condition for loop fusion states that there cannot be any negative distance dependencies between any statement in the two loop bodies. A negative distance dependence exists when the second loop computes data that is read by the first loop in a subsequent iteration. Figure 4.16 shows two loops that have a negative distance dependence between them.

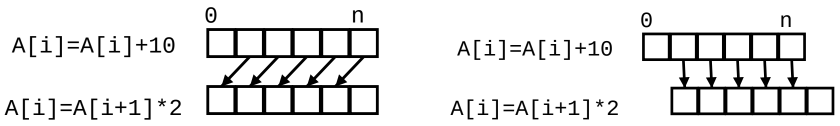
```
for (i=0; i < n; i++)
{
  A[i] = A[i] + 10;
}
for (j=0; j < n; j++)
{
  A[j] = A[j+1] * 2;
}
```

Figure 4.16: Unaligned Loop Example

Figure 4.17(a) shows the array access pattern that results in a negative distance dependence for the loops in Figure 4.16. The top row represents array accesses by the first loop. The bottom row represents array accesses by the second loop. Columns indicate the iteration of the loop. The edges represent data dependencies between the two loop bodies. Since the edges go from a current iteration in the first loop (top row) to a previous iteration in the second loop (bottom row) a negative distance dependence exists. We say that the array accesses are unaligned. A common technique to eliminate negative distance dependencies is called loop alignment. We shift the bottom row over one iteration as shown in Figure 4.17(b). After the shifting, dependencies occur within the same iteration.

### 4.5.1 Loop Alignment

In order to align two loops to remove the negative distance dependencies we must be able to determine the dependence distance at compile time. If there are multiple negative dependence distances between two loop bodies, the maximum of the



(a) Original Array Accesses (b) Array Accesses after alignment

Figure 4.17: Array Access Patterns

absolute values of the distances is used for alignment. For example, given loops $L_j$ and $L_k$ with a maximum negative dependence distance of 2, alignment is achieved by executing two iterations of $L_j$, followed by the remaining iterations of the $L_j$, combined with the iterations of $L_k$. The final 2 iterations of $L_k$ are then executed.

```
for (i=0; i < n; i++)
{
  A[i] = A[i] * 2;
}


for (j=0; j < n; j++)
{
  A[j] = A[j+2] * 3;
}
```

```
for (i=0; i < n+2; i++)
{
  if (i < 2)
  {
    A[i] = A[i] * 2;
  }
  else if (i < n)
  {
    A[i] = A[i] * 2;
    A[i-2] = A[i] * 3;
  }
  else
  {
    A[i-2] = A[i] * 3;
  }
}
```

(a) Original loops (b) Fused loop after alignment

Figure 4.18: Loop Alignment Example

Figure 4.18 shows an example of two non-aligned loops (a) and the resulting alignment combined with fusion (b). In this example, the negative distance dependence between the two loop bodies has a distance of 2 iterations. In other words, the second loop uses a value that is calculated by the first loop 2 iterations in the future. Thus, the aligned loop has a guard that executes the first two iterations of the first loop, to reduce the dependence distance to 0. A second guard is inserted to ensure that the body of the first loop is not executed more times than the original loop. Finally, the body of the second loop is executed to maintain its correct number of iterations.

```
for (i=0; i < n+2; i++)
{
  if (i < 2)
  {
    A[i] = A[i] * 2;
  }
  else if (i < n)
  {
    A[i] = A[i] * 2;
    A[i-2] = A[i] * 3;
  }
  else
  {
    A[i-2] = A[i] * 3;
  }
}
```

(a)

```
for (i=0; i < 2; i++)
{
  A[i] = A[i] * 2;
}
for (i=2; i < n; i++)
{
  A[i] = A[i] * 2;
  A[i-2] = A[i] * 3;
}
for (i=n; i < n+2; i++)
{
  A[i-2] = A[i] * 3;
}
```

(b)

Figure 4.19: Aligned Loops After Index Set Splitting

In this implementation, we again used guard statements to inhibit execution of iterations of one loop body. Index Set Splitting is used to remove the control flow from the loop body and create three separate loops. Figure 4.19 shows the aligned loops after index set splitting has been performed.

There are two scenarios that can arise when attempting to align loops: (*i*) the two loops have the same upper bound or, (*ii*) the two loops have different upper bounds.

When the two loops have the same upper bound, loop alignment produces a loop similar to that seen in Figure 4.18(b). The upper bound of the new loop is equal to the upper bound of the original loops plus the negative dependence distance. An initial guard is inserted to align the two loops. This guard compares the value of the induction variable against the negative dependence distance. Only the body of the first loop is placed under the first guard. A second guard is inserted to test the value of the induction variable against the original upper bound. This second guard protects the first loop from executing too many times. The bodies of both the first loop and the second loop are copied under the second guard. The value of the induction variable used in the body of the second loop is subtracted by the negative dependence distance. Finally, a third guard is added to determine if the value of the induction variable is greater than the original upper bound. The body of the second loop is copied under the third guard guaranteeing that it iterates the correct number of times.

When two loops have different upper bounds, if the upper bound of the second loop is lower than the upper bound of the first the guard branch that is inserted to make the loops conform is used to align the loops. Let's assume the difference in the upper bounds of the two loops is $n$. Let's also assume that the negative dependence between the loop bodies has a distance of $d$. In order to make the two loops conform, a guard branch is inserted to allow the first loop to execute $n$ times more than the second loop. If $n > d$, the guard branch is constructed so that the first $n$ iterations of the first loop are executed first. The remaining iterations include the body of both loops. This results in the two loops being aligned. It is necessary to subtract $n$ from the induction variable in the body of the second loop to ensure the correct data accesses are made. Figure 4.20 shows the algorithm used to fuse two unaligned loops.

If the upper bound of the first loop is lower than the upper bound of the second loop, we need to insert guard branches to make the loops conform. The portion of the fused loop that contains both loop bodies can then be aligned by inserting additional guards as described above. This transformation is not performed because it results in a significant amount of code duplication. Thus, if the upper bound of the first loop is lower than the upper bound of the second loop, the two loops are not aligned and thus cannot be fused.

If the difference in the upper bounds cannot be determined at compile time, a similar approach is necessary to align the loops. This also results in a significant amount of code duplication and thus is not performed.

## 4.6 Ordering Code Transformations

We presented algorithms that can be used to satisfy the four conditions of loop fusion. The order in which these algorithms are applied plays a significant role in the performance of loop fusion. This section focuses on how the algorithms should be applied to maximize the number of loops fused while minimizing the time required to identify loop fusion candidates and to apply the appropriate transformations.

### 4.6.1 Data Structures

There are several data structures that are used extensively during loop fusion. These data structures include the control flow graph, the data flow graph, the domina-

```
FuseAndAlign($L_j, L_k$)
1.  if ($\kappa(L_j) = \kappa(L_k)$)
2       $\sigma \leftarrow min(DependenceDistance(L_j, L_k))$
3       Create $L_m$ with Upper Bound of $\kappa(L_j)$
4       Insert Guard for $\sigma$ at beginning of $L_m$
5       Copy body of $L_j$ to $L_m$ within first guard
6       Insert second guard for $\kappa(L_j)$
7       Copy body of $L_j$ to $L_m$ within second guard
8       Copy body of $L_k$ to $L_m$, after $L_j$ body, within second guard
9       Subtract $\sigma$ from all uses of induction variable in copy of $L_k$ body
10      Insert else statement
11      Copy body of $L_k$ within else statement
12      Subtract $\sigma$ from all uses of induction variable in copy of $L_k$ body
13  else
14      $\sigma \leftarrow |\kappa(L_j) - \kappa(L_k)|$
15      Create $L_m$ with Upper Bound $\kappa(L_k)$
16      Insert Guard for $\sigma$ at beginning of $L_m$
17      Copy body of $L_j$ to $L_m$ within guard
18      Insert else statement
19      Copy body of $L_j$ to $L_m$ within else statement
20      Copy body of $L_k$ to $L_m$, after $L_j$ body, within guard
21      Subtract $\sigma$ from all uses of induction variable in $L_k$ body
22. endif
```

Figure 4.20: Algorithm to Fuse Unaligned Loops Using Guard Statements

tor, and the postdominator trees. The transformations introduced in the previous sections modify the intermediate representation, causing these data structures to become inconsistent. For example, the movement of code from between two loops causes the control flow graph, dominator, and postdominator trees to become stale, no longer representing the current organization.

Incrementally updating the data structures would require very careful crafting of the compiler code and would make the compiler more difficult to maintain. The designers of TPO decided that after statements are moved the control flow graph must be recreated. The creation of these data structures is an expensive operation and should therefore be invoked conservatively. For this reason, we designed the loop fusion algorithm to minimize the number of times the data structures are destroyed and rebuilt. This is achieved by only modifying parts of the data structures where analysis has already been performed. Modifications are never made to a position in the control flow graph where the algorithms have not analyzed. This careful code analysis and update allows us to perform the loop fusion algorithms on an entire set of loops before the data structures are rebuilt. However, care must be taken in the way the loops in the set are ordered and the way the entire set is traversed.

### 4.6.2 Iteration

Control flow equivalent loops that are candidates for fusion are grouped into equivalence classes. The loop fusion algorithm attempts to fuse each loop with every other loop in the same equivalence class. The list must be organized to allow an ordered traversal of the loops, and thus guarantee that the conditions described in Section 4.6.1 are not violated. The list is ordered such that the first time a position in the list is reached for analysis, nothing at that position has been modified. Two passes of loop fusion are necessary in order to complete all steps of the algorithms described in the previous sections. Since all of the loops in the list are control flow equivalent, the dominance relation provides a total order over the list. In the first pass, the loops in the list are ordered in dominance order. Let $L_i$ be the loop at position LoopSet$[i]$ and $L_j$ be the loop at LoopSet$[j]$. For every position $i, j$ $(i < j)$ in the list, $L_i$ dominates $L_j$. Traversal of the list then corresponds to a topological traversal of the ECFG. In this pass, modifications are only made to locations that are located higher in the control flow graph. For example, if there is intervening code between $L_i$ and $L_j$, the intervening code is only moved up, above $L_i$ in this pass. Any code that needs to be moved down is moved during the next pass.

During the second pass of loop fusion, the loops in the list are organized in postdominance order. For every position $m, n$ $(m < n)$ in the list, $L_m$ postdominates $L_n$. Traversal of this list corresponds to a reverse topological traversal of the ECFG. In this pass, modifications are only made to locations that are lower in the control flow graph. For example, intervening code between two loops can only be moved down, below $L_m$.

For a given pass of loop fusion, the direction that the loops are traversed is called the *iteration direction.*

The iteration through sets of loops allows all algorithms to be applied correctly while minimizing the number of times the required data structures must be rebuilt.

### 4.6.3 Loop Fusion Algorithm

The code presented in Figure 4.21 is used as a running example to illustrate the loop fusion algorithm. This example has five loops accessing two different arrays, $A$ and $B$. We assume that there is no overlap between the memory locations of $A$ and $B$, *i.e.,* there is no $i$ and $j$ such that the address of $A[i]$ overlaps with the address of $B[j]$.

```
p = 2;                           /* S1 */
for (i=0; i < n; i++)            /* L1 */
{
  A[i] = A[i] * 2;
}
for (j=0; j < n; j++)            /* L2 */
{
  B[j] = A[j] + A[j+1] * p;
}
p = 4                            /* S2 */
q = 0.0                          /* S3 */
for (k=0; k < m; k++)            /* L3 */
{
  q = q + B[k];
}
if (n < m)                       /* S4 */
{
  for (l=0; l < n; l++)          /* L4 */
  {
    B[l] = B[l] * p;
  }
}
else                             /* S5 */
{
  B[m] = B[n] * 4;               /* S6 */
}
for (p=0; p < n; p++)            /* L5 */
{
  B[p] = A[p] + B[p] / q;
}
```

Figure 4.21: Running Example

Figure 4.22 presents the LoopFusion algorithm. The algorithm operates one nest level at a time, starting at the outermost level and moving towards the innermost level (Step 1). It proceeds by gathering all loops at the current nest level into one set, called the *LoopSet*. In Step 3 loops that are not eligible for fusion are removed from the LoopSet. The algorithm continues by grouping control flow equivalent loops from the LoopSet into lists called *FusionCandidateLists*. Each FusionCandidateList $S_i$ is then processed by the loop fusion algorithm. The algorithm iterates, alternating forward and reverse passes over the list until it finds no more loops to be fused. At the beginning of each pass, the ECFG and dominance relations are re-computed to reflect any changes made in the previous pass.

```
LoopFusion
1.  foreach NestLevel $N_i$ from outermost to innermost
2.      Gather all loops in $N_i$ into a single LoopSet
3.      Remove loops non-eligible for fusion from LoopSet
4.      Gather control flow equivalent loops from LoopSet into FusionCandidateLists
5.      foreach FusionCandidateList $S_i$
6.          FusedLoops $\leftarrow$ $True$
7.          Direction $\leftarrow$ $Forward$
8.          while FusedLoops $= True$
9.              if $|S_i| < 2$
10.                 break
11.             endif
12.             Build Data Structures
13.             Compute Dominance Relation
14.             FusedLoops = CFELoopFusionPass($S_i$,Direction)
15.             if FusedLoops $= False$
16.                 FusedLoops = NonCFELoopFusionPass(LoopSet)
17.             endif
18.             if Direction $= Forward$
19.                 Direction $= Reverse$
20.             else
21.                 Direction $= Forward$
22.             endif
23.         endfor
24.     end while
25. endfor
```

Figure 4.22: Loop Fusion Algorithm.

The while loop starting at Step 8 iterates over the FusionCandidateList until no loops are fused. In every iteration, the CFELoopFusionPass algorithm is called. This algorithm attempts to fuse loops that are control flow equivalent. If the CFELoopFusionPass is unable to fuse any loops, the NonCFELoopFusionPass is called. This algorithm attempts to fuse loops that are not control flow equivalent.

The algorithms are called in this order because it is preferable to fuse control flow equivalent loops over non-control flow equivalent loops to minimize the amount of code growth and the number of times the data structures must be rebuilt.

In the example of Figure 4.21, all loops are at the same nest level and eligible for fusion. Thus, all loops are in the LoopSet. The FusionCandidateList contains `L1`, `L2`, `L3`, and `L5`. `L4` is not included because it is not control flow equivalent with the other loops. The loops in the FusionCandidateList are ordered in dominance order (*e.g.,* `L1`, `L2`, `L3`, `L5`) for the first pass of the loop fusion algorithm.

### 4.6.4 Control Flow Equivalent FusionPass

```
CFELoopFusionPass($S_i$, $Direction$)
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.          $\sigma \leftarrow |\kappa(L_j) - \kappa(L_k)|$
7.          if $L_j$ and $L_k$ are non-conforming and
               $\sigma$ cannot be determined
8.             continue
9.          endif
10.         if $DependenceDistance(L_j, L_k) < 0$ and
               $DependenceDistance(L_j, L_k)$ cannot be determined at compile time
11.            continue
12.         endif
13.         FusedLoops = $True$
14.         MoveInterveningCode($L_j, L_k, Direction$)
15.         if InterveningCode($L_j, L_k$) = $True$
16.            continue
17.         endif
18.         if $DependenceDistance(L_j, L_k) < 0$
19.            $L_m \leftarrow$ FuseAndAlign($L_j, L_k$)
20.         else if $L_j$ and $L_k$ are non-conforming
21.                 $L_m \leftarrow$ FuseWithGuard($L_j, L_k$)
22.         else
23.                 $L_m \leftarrow$ Fuse($L_j, L_k$)
24.         endif
25.         $S_i \leftarrow S_i \cup L_m - \{L_j, L_k\}$
26. endfor
27. return FusedLoops
```

Figure 4.23: Control Flow Equivalent Loop Fusion Algorithm.

Figure 4.23 presents the algorithm used for a single pass of loop fusion on a list of control flow equivalent loops. The algorithm is given a FusionCandidateList $S_i$ containing loops to be fused and the current iteration direction. In a forward pass, the loops contained in the FusionCandidateList $S_i$ are traversed in dominance order while during a reverse pass the traversal is in postdominance order.

For each pair of loops $L_j$ and $L_k$ in $S_i$, the algorithm begins by checking if $L_j$ and $L_k$ are adjacent (Step 3). If they are not adjacent, a check is performed to determine if the intervening code between them can be moved using the algorithm presented in Section 4.2. If all of the intervening code cannot be moved, $L_j$ and $L_k$ cannot be fused and the algorithm continues to the next pair of loops.

In Step 6, the difference between the upper bounds of the two loops $\kappa(L_j)$ and $\kappa(L_k)$ is computed and stored in $\sigma$. Step 7 determines if the two loops do not conform and if the difference in bounds can be determined. If the Compile Time Bounds checking described in Section 4.3.2 is enabled, the compiler must be able to resolve $\sigma$ to a value. If the Run Time Bounds Checking described in Section 4.3.3 is enabled, $\sigma$ can be a symbolic difference (*i.e.*, $n - m$). If the loops cannot be made to conform, the two loops cannot be fused and the algorithm continues to the next pair of loops.

Step 10 determines if the dependence relations between the bodies of the loops $L_j$ and $L_k$ prevent fusion. If there is a negative distance dependence and the distance cannot be determined at compile time, loops $L_j$ and $L_k$ are not fused.

At this point, all four conditions for fusion have been tested and it has been verified that all conditions have already been satisfied or are satisfiable. The algorithm proceeds by performing the necessary code transformations to make all conditions satisfiable. The first transformation is to move any intervening code from between the two loops. This transformation is limited to moving code in one direction based on the current iteration direction. Thus, if it is necessary to move code in both directions, not all code is moved in this iteration.

If all of the intervening code is moved successfully (or no intervening code is present), the algorithm determines which fusion algorithm should be called. If there is a negative distance dependence between the two loops, the FuseAndAlign algorithm from Section 4.5 is used. If the bounds of the two loops do not conform (but no negative distance dependence is present), the FuseWithGuard algorithm from Section 4.3 is used. Note that this algorithm is either the compile time bounds check or the run time bounds check depending on which compiler option is enabled.

In the example of Figure 4.21 the first loops to be compared are `L1` and `L2`. They are adjacent, but they do not conform and there is a negative dependence of distance 1 between the two loop bodies. The test in Step 18 of Figure 4.23 is true and the two loops are fused using the algorithm from Section 4.5. Since the bounds of the two loops did not conform, the loops are aligned using a guard statement. This fusion results in the loop L6 shown in Figure 4.24(a).

```
p = 2;                            /* S1 */
for (i=0; i < n; i++)             /* L6 */
{
  if (i < 1)
  {
    A[i] = A[i] * 2;
  }
  else
  {
    A[i] = A[i] * 2;
    B[i-1] = A[i-1] + A[i] * p;
  }
}
p = 4;                            /* S2 */
q = 0.0;                          /* S3 */
for (k=0; k < m; k++)             /* L3 */
{
  q = q + B[k];
}
if (n < m)                        /* S4 */
{
  for (l=0; l < n; l++)           /* L4 */
  {
    B[l] = B[l] * p;
  }
}
else                              /* S5 */
{
  B[m] = B[n] * 4;                /* S6 */
}
for (p=0; p < n; p++)             /* L5 */
{
  B[p] = A[p] + B[p] / q;
}
```

(a) After L1 and L2 Fused

```
p = 2;                            /* S1 */
q = 0.0;                          /* S3 */
for (i=0; i < n; i++)             /* L6 */
{
  if (i < 1)
  {
    A[i] = A[i] * 2;
  }
  else
  {
    A[i] = A[i] * 2;
    B[i-1] = A[i-1] + A[i] * p;
  }
}
p = 4;                            /* S2 */
for (k=0; k < m; k++)             /* L3 */
{
  q = q + B[k];
}
if (n < m)                        /* S4 */
{
  for (l=0; l < n; l++)           /* L4 */
  {
    B[l] = B[l] * p;
  }
}
else                              /* S5 */
{
  B[m] = B[n] * 4;                /* S6 */
}
for (p=0; p < n; p++)             /* L5 */
{
  B[p] = A[p] + B[p] / q;
}
```

(b) After Intervening Code Moved

Figure 4.24: Running Example

The next comparison is between loops L6 and L3. There are no dependencies preventing fusion but the loops are not adjacent. All of the intervening code can be moved from between the two loops allowing fusion to occur. However, dependencies require that statement S2 be moved down to the point after loop L3 while statement S3 be moved up to the point before L6. Since statements can only be moved up in the forward pass, Step 14 of Figure 4.23 only moves statement S3. Thus, the test for intervening code between loops `L6` and `L3` in Step 15 fails and the loops cannot be fused on this pass. The results of moving statement S3 is shown in Figure 4.24(b).

Loop `L6` cannot be fused with any other loop in the list because statement S2 is intervening code that cannot be moved and thus prevents all fusion attempts.

The next comparison is between loops `L3` and `L5`. The two loops are not adjacent and dependencies prevent the intervening code (`S4`, `L4`, `S6`, `S7`) from being moved from between the two loops. Thus, the test on Step 3 of Figure 4.23 fails and the loops cannot be fused.

The ECFG is rebuilt and the dominance and postdominance relations re-computed before a new pass starts. In the reverse pass, the loops in the FusionCandidateList are ordered in postdominance order (*e.g.,* `L5`, `L3`, `L6`). Loops `L5` and `L3` are compared but cannot be fused because the intervening code between them cannot be moved. `L5` cannot be fused with `L3` or `L6` because the same intervening code prevents fusion.

Next, loops `L3` and `L6` are compared. There is a single intervening statement between them (`S2`) that is moved down to the point below `L3` in Step 14. The two loops are fused, using the run time bounds check algorithm described in Section 4.3.3. The results of this fusion are shown in Figure 4.25(a).

The reverse pass terminates after loops `L3` and `L6` are fused. The next forward pass results in no additional fusions.

### 4.6.5 Non Control Flow Equivalent FusionPass

Figure 4.26 presents the algorithm to fuse non-control flow equivalent loops. The algorithm processes a set of loops, $S_i$, looking for pairs of loops $L_j$ and $L_k$ that satisfy the dominance relations of Figure 4.11(a) and (b). Step 2 guarantees that $L_j$ is the unconditionally executed loop and $L_k$ is the conditionally executed loop. This is required by the algorithms in Section 4.4. Step 3 uses the algorithm from Section 4.4.2 to check for intervening code and determine if the intervening code can be moved. Steps 6 and 7 determine if the upper bounds of loops $L_j$ and $L_k$ conform or can be made to conform. Step 10 determines if there is a negative distance dependence between the two loops and if the loops can be aligned to remove the dependence. In Step 14 the intervening code between the two loops is moved using the algorithm specified in Section 4.4.2. Finally, the two loops are fused using the

```
p = 2;                               /* S1 */
q = 0.0;                             /* S3 */
S8:  S = max(n,m);                   /* S8 */
S9:  T = min(n,m);                   /* S9 */
for (i=0; i < S; i++)                /* L6 */
{
  if (i < T)
  {
    if (i < 1)
    {
      A[i] = A[i] * 2;
    }
    else
    {
      A[i] = A[i] * 2;
      B[i-1] = A[i-1] + A[i] * p;
    }
    q = q + B[i];
  }
  else if (i < n)
  {
    if (i < 1)
    {
      A[i] = A[i] * 2;
    }
    else
    {
      A[i] = A[i] * 2;
      B[i-1] = A[i-1] + A[i] * p;
    }
  }
  else
  {
    q = q + B[i];
  }
}
p = 4;                               /* S2 */
if (n < m)                           /* S4 */
{
  for (l=0; l < n; l++)              /* L4 */
  {
    B[l] = B[l] * p;
  }
}
else                                 /* S5 */
{
  B[m] = B[n] * 4;                   /* S6 */
}
for (p=0; p < n; p++)                /* L5 */
{
  B[p] = A[p] + B[p] / q;
}
```

(a) After L3 and L6 Fused

```
p = 2;                               /* S1 */
q = 0.0;                             /* S3 */
S8:  S = max(n,m);                   /* S8 */
S9:  T = min(n,m);                   /* S9 */
for (i=0; i < S; i++)                /* L6 */
{
  if (i < T)
  {
    if (i < 1)
    {
      A[i] = A[i] * 2;
    }
    else
    {
      A[i] = A[i] * 2;
      B[i-1] = A[i-1] + A[i] * p;
    }
    q = q + B[i];
  }
  else if (i < n)
  {
    if (i < 1)
    {
      A[i] = A[i] * 2;
    }
    else
    {
      A[i] = A[i] * 2;
      B[i-1] = A[i-1] + A[i] * p;
    }
  }
  else
  {
    q = q + B[i];
  }
}
p = 4;                               /* S2 */

if (n < m)                           /* S4 */
{
  for (l=0; l < n; l++)              /* L4 */
  {
    B[l] = B[l] * p;
    B[l] = A[l] + B[l] / q;
  }
}
else                                 /* S5 */
{
  B[m] = B[n] * 4;                   /* S6 */
  for (p=0; p < n; p++)              /* L5 */
  {
    B[p] = A[p] + B[p] / q;
  }
}
```

(b) After L4 and L5 Fused

Figure 4.25: Running Example

```
NonCFELoopFusionPass($S_i$)
1.  FusedLoops = $False$
2.  foreach pair of loops $L_j$ and $L_k$ in $S_i$, such that
            ($L_j$ dominates $L_k$ and $L_k$ does not postdominate $L_j$) or
            ($L_j$ postdominates $L_k$ and $L_k$ does not dominate $L_j$)
3.          if NonCFE_IsInterveningCodeMovable($L_j, L_k$) = $False$
4.             continue
5.          endif
6.          $\sigma \leftarrow |\kappa(L_j) - \kappa(L_k)|$
7.          if $L_j$ and $L_k$ are non-conforming and
               $\sigma$ cannot be determined
8.             continue
9.          endif
10.         if $DependenceDistance(L_j, L_k) < 0$ and
               $DependenceDistance(L_j, L_k)$ cannot be determined at compile time
11.            continue
12.         endif
13.         FusedLoops = $True$
14.         NonCFE_MoveInterveningCode($L_j, L_k$)
15.         if $DependenceDistance(L_j, L_k) < 0$
16.            $L_m \leftarrow$ FuseAndAlign($L_j, L_k$)
17.         else if $L_j$ and $L_k$ are non-conforming
18.                 $L_m \leftarrow$ FuseWithGuard($L_j, L_k$)
19.         else
20.                 $L_m \leftarrow$ Fuse($L_j, L_k$)
21.         endif
22.         Copy $L_j$ into non-fuseable path
23.         $S_i \leftarrow S_i \cup L_m - \{L_j, L_k\}$
24.         Build Data Structures
25. endfor
26. return FusedLoops
```

Figure 4.26: Non Control Flow Equivalent Loop Fusion Algorithm

appropriate fusion algorithm.

Since all of the loops in the LoopSet are not guaranteed to be control flow equivalent, there is no dominance relation between all of the loops in the LoopSet. Thus, it is not possible to use the iterative approach described in Section 4.6.2 to iterate over the LoopSet. For this reason, when two non-control flow equivalent loops are fused, it is necessary to rebuild all of the data structures. This is done in Step 24 of Figure 4.26.

Many of the steps in the NonCFELoopFusionPass algorithm are the same as in the CFELoopFusionPass algorithm of Figure 4.23. Different algorithms to identify and move intervening code are necessary because loops $L_j$ and $L_k$ are not control flow equivalent. The check to determine if all of the intervening code was successfully moved (Step 15 in Figure 4.23) is not necessary because movement of parts of the intervening code is not done when the two loops are not control flow equivalent. If all of the intervening code cannot be moved, the two loops cannot be fused.

In the example of Figure 4.25(a), no additional loops can be fused during a CFELoopFusionPass. The LoopFusion algorithm of Figure 4.22 calls the NonCFELoopFusionPass to attempt to fuse non-control flow equivalent loops. The entire LoopSet, containing `L6`, `L4` and `L5`, is passed to the NonCFELoopFusionPass algorithm.

The first loops to be compared are `L6` and `L4`. There is no intervening code between the two loops that needs to be moved. The upper bounds of loop `L6` and `L4` do not conform, but run time bounds checking can be used to allow fusion of the two loops. However, there is a negative dependence of distance 1 between the accesses of array `B` in loops `L6` and `L4`. This dependence cannot be removed because the upper bounds of loops `L6` and `L4` cannot be determined at compile time. Thus, `L6` and `L4` cannot be fused.

Next, loops `L4` and `L5` are compared. There is intervening code between loops `L4` and `L5`, but the code can be moved allowing the loops to be fused. The bounds of `L4` and `L5` conform and there are no negative distance dependencies between the two loop bodies. Step 14 of Figure 4.26 moves the intervening code from between `L4` and `L5`. The two loops are fused in Step 20 and loop `L5` is copied into the non-fuseable path in Step 22. The results of fusion are shown in Figure 4.25(b).

After successfully fusing two non-control flow equivalent loops, the LoopFusion algorithm of Figure 4.22 calls CFELoopFusionPass and NonCFELoopFusion-Pass once more. No loops will be fused in either pass and the algorithm will terminate.

Chapter 5

# Experimental Results

This chapter presents results collected during both compile time and run time using the SPEC95 and SPEC2000 benchmark suites. The experiments conducted were designed to answer the following three questions:

1. Do the algorithms presented in Chapter 4 enable additional loop fusion?
2. Are there benefits to enabling loop fusion?
3. Do the algorithms prohibitively impact compilation times?
4. What are the run time performance benefits as a result of loop fusion?

## 5.1 Compiler Versions

To answer Question 1, we compare 5 different versions of our algorithm with an implementation of basic loop fusion. The versions of the compiler are:

**Base:** The basic loop fusion algorithm in which no code transformations are performed to try and make loops fusible. The baseline compilation is performed at optimization level O4.

**MIC:** The algorithm iterates, alternating forward and reverse passes, moving intervening code from between two loops to enable fusion.

**GuardCTB:** In addition to moving intervening code, guard branches are used to fuse loops when the difference in upper bounds can be determined at compile time.

**GuardRTB:** In addition to moving intervening code, guard branches are used to fuse non-conforming loops. The compiler does not need to be able to determine the difference in upper bounds at compile time.

**align:** In addition to moving intervening code and fusing non-conforming loops, guard branches are used to align loops that have a negative distance dependence between them.

**nonCFE:** The complete loop fusion algorithm, including movement of intervening code, guard branches to make loops conform and align as well as fusing of non-control flow equivalent loops.

These five versions were chosen because, in our opinion, they create a logical evolution of the techniques presented in Chapter 4 into a complete loop fusion algorithm. Appendix A provides additional results for each of the specific optimizations as well as other combinations of optimizations.

The algorithms were implemented in the development version of the IBM XL compiler suite. Benchmarks compiled with the modified compiler were run on an IBM eServer pSeries 630 machine, built with the POWER4 processor. Measurements from the SPEC benchmark suites were not collected using the official SPEC Tools.

## 5.2 Machine Configuration

The POWER4 is a new microprocessor implementation of the 64-bit PowerPC architecture designed and manufactured by IBM for the UNIX®server market. The pSeries 630 machine is a two- to four-way symmetric multiprocessor machine. Two processor cores running at 1.0 GHz are located on a single die. Each of the 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 1024-entry 4-way set associative TLB supporting 4 KB and 16 MB page sizes. The processors share a single 8-way set associative 1.44 MB on-chip combined L2 cache and a 32 MB L3 cache.

The L1 instruction cache can support up to 3 outstanding misses and the 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 streams of data. 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 [20].

The machine used for testing is a 2-way, POWER4 machine. It contains two processors on a single die running at 1.0 GHz and 2 GB of main memory.

## 5.3 Contrived Example

To answer question 2, we study the performance of the contrived C program is presented in Figure 5.1. The compiler versions introduced in Section 5.1 are used to compile the sample program, fusing different combinations of loops. The resulting executables are run to determine the performance benefit of each compiler version.

The sample C program in Figure 5.1 was compiled with three different compiler optimizations: no loop fusion, baseline, and MIC. With no loop fusion, none of the loops in the program are fused. With baseline loop fusion, loops `L1`, `L2`, `L3` and `L4` are fused, resulting in 7 loops. With MIC loop fusion, loops `L1`, `L2`, `L3` and `L4` are fused into a single loop body, loops `L5`, `L7`, `L9` are fused into a single loop body and loops `L6`, `L8` and `L10` are fused into a single loop body (Figure 5.2). This results in a program with 3 loops, which is the minimum number of loops possible while maintaining correct program execution.

The contrived example was run with two different array sizes (definitions of `N` and `M`), 50000 and 100000. Table 5.1 shows the run times for each of the loop fusion options and each of the array sizes. Each run time is reported in seconds and was computed by averaging the times of three seperate runs, obtained using the UNIX *time* command. The *% Improvement* columns report the improvement in run time of the baseline and MIC compilations compared to the compilation where no loop fusion was performed.

| Array Size | No Loop Fusion | Baseline | % Improvement | MIC | % Improvement |
|---|---|---|---|---|---|
| 50000 | 203.59 | 203.45 | 0.07 | 180.68 | 11.25 |
| 100000 | 773.9 | 773.5 | 0.05 | 724.27 | 6.41 |

Table 5.1: Run Time and Speedups of Contrived Example

Using the baseline compilation option results in 4 loops being fused, a run time improvement of 0.07% for arrays of size 50000 and a run time improvement of

```
int main (int argc, char **argv)
{
  int p, i, j, k, l, s, g, r, a, b, c, d, e, f;
  float q;
  int A[N], B[M], C[N], D[M];
  /* Initialization */
  for (i=0; i < N; i++) {        /* L1 */
    A[i] = (i * 2) + (i - 6);
  }
  for (i=0; i < M; i++) {        /* L2 */
    B[i] = (i * 3) + (6 - i);
  }
  for (i=0; i < M; i++) {        /* L3 */
    C[i] = (i * 4) + (i + 2);
  }
  for (i=0; i < M; i++) {        /* L4 */
    D[i] = (i * 5) + (i + 4);
  }
  j = k = l = r = a = b = c = d = 0;
  for (i=0; i < N; i++) {        /* L5 */
    j += A[i] + 2;
    l += C[i] * 3;
    for (g=0; g < M; g++) {      /* L6 */
      k += B[g] - 4;
      r += D[g] * 2;
    }
  }
  e = 0;
  f = 0;
  for (i=0; i < N; i++) {        /* L7 */
    a += A[i] + 2;
    b += C[i] - 2;
    for (g=0; g < M; g++) {      /* L8 */
      c -= B[g] + 6;
      d -= D[g] / 4;
    }
  }
  p = 4;
  q = 0.0;
  for (i=0; i < M; i++) {        /* L9 */
    q = q + B[i];
    p = p - B[i];
    for (g=0; g < M; g++) {      /* L10 */
      e += B[g] + q;
      f += A[g] - p;
    }
  }
  s = 0;
  p += 5;
  q -= 4;
  printf("%d %d %d %d %d %d %d %d %d %d %d %d\n",
          j, k, l, r, a, b, c, d, e, f, p, q);
}
```

Figure 5.1: Contrived Example

```
int main (int argc, char **argv)
{
  int p, i, j, k, l, s, g, r, a, b, c, d, e, f;
  float q;
  int A[N], B[M], C[N], D[M];
  /* Initialization */
  for (i=0; i < N; i++) {        /* L1 */
    A[i] = (i * 2) + (i - 6);
    B[i] = (i * 3) + (6 - i);
    C[i] = (i * 4) + (i + 2);
    D[i] = (i * 5) + (i + 4);
  }
  j = k = l = r = a = b = c = d = 0;
  e = 0;
  f = 0;
  p = 4;
  q = 0.0;
  for (i=0; i < N; i++) {        /* L5 */
    j += A[i] + 2;
    l += C[i] * 3;
    a += A[i] + 2;
    b += C[i] - 2;
    q = q + B[i];
    p = p - B[i];
    for (g=0; g < M; g++) {      /* L6 */
      k += B[g] - 4;
      r += D[g] * 2;
      c -= B[g] + 6;
      d -= D[g] / 4;
      e += B[g] + q;
      f += A[g] - p;
    }
  }
  s = 0;
  p += 5;
  q -= 4;
  printf("%d %d %d %d %d %d %d %d %d %d %d %d\n",
          j, k, l, r, a, b, c, d, e, f, p, q);
}
```

Figure 5.2: Contrived Example After Loop Fusion with MIC

0.05% for arrays of size of 100000. The loops that are fused (`L1`, `L2`, `L3`, `L4`) are initialization loops and do not contain the majority of the computation performed by the program. Thus, these four loops are considered *cold* loops. This example demonstrates that fusing cold loops does not significantly impact the running time of the program.

Using the MIC compilation option results in 9 loops being fused and a runtime improvement of 11.25% for arrays of size 50000 and a run time improvement of 6.41% for arrays of size 100000. These loop fusions include the initialization loops as well as the loops containing the majority of the computation. This illustrates the benefit of data reuse obtained through loop fusion.

One possible explanation for a smaller performance improvement when using larger arrays is the number of cache misses is not significantly different (*i.e.*, there is the same amount of data reuse with array sizes of 50000 and 100000). If this is the case, the cost of the cache misses is ammortized over a longer running time of the loops (the loops are executing for 100000 iterations as opposed to 50000 iterations) resulting in less performance benefit from data reuse.

## 5.4 Compile Time Results

To answer question 3 we collected the compilation times of the benchmarks using the versions of the algorithm described in Section 5.1.

This section presents results from the loop fusion algorithms gathered at compile time. These results include the number of loops fused and the compilation times for the different configurations of the compiler. Benchmarks that did not have an increase in the number of loops fused with any algorithm are omitted from the table. Also, benchmarks from SPEC95 which also occur in SPEC2000 are not repeated.

Tables 5.2 and 5.3 present the number of loops fused using each algorithm. Counting the number of loops fused is a metric that allows us to analyze how the techniques assist in enabling loop fusion. The results demonstrate that each algorithm was able to increase the number of loops fused.

Overall, the complete loop fusion algorithm, compared to the original loop fusion algorithm, results in 18 times as many loops fused in *sixtrack*, 15 times as many loops fused in *applu* and 8 times as many loops fused in *crafty*. It also enables loop fusion in *art*, *lucas*, *mesa*, *twolf* and *vpr*, where no loops were fused with the original algorithm. These results indicate that the algorithms presented are effective in increasing the number of loops that are fused.

| | Benchmark | Base | MIC | Guard CTB | Guard RTB | align | non CFE |
|---|---|---|---|---|---|---|---|
| SPEC 2000 FP | ammp | 1 | 1 | 1 | 2 | 2 | 2 |
| | applu | 2 | 2 | 3 | 3 | 30 | 30 |
| | apsi | 5 | 5 | 5 | 5 | 6 | 11 |
| | art | 0 | 1 | 1 | 3 | 3 | 3 |
| | facerec | 12 | 18 | 18 | 20 | 20 | 20 |
| | fma3d | 36 | 77 | 82 | 112 | 113 | 116 |
| | galgel | 21 | 35 | 35 | 40 | 40 | 40 |
| | lucas | 0 | 2 | 2 | 2 | 2 | 2 |
| | mesa | 0 | 0 | 0 | 0 | 0 | 1 |
| | mgrid | 12 | 13 | 13 | 13 | 13 | 13 |
| | sixtrack | 4 | 6 | 37 | 44 | 45 | 73 |
| | swim | 4 | 4 | 4 | 4 | 6 | 6 |
| SPEC 2000 INT | bzip2 | 9 | 9 | 12 | 12 | 12 | 12 |
| | crafty | 3 | 12 | 22 | 24 | 25 | 25 |
| | eon | 0 | 0 | 0 | 0 | 0 | 0 |
| | gap | 1 | 1 | 4 | 5 | 5 | 5 |
| | gcc | 5 | 7 | 9 | 9 | 9 | 9 |
| | gzip | 0 | 0 | 4 | 7 | 7 | 7 |
| | twolf | 0 | 1 | 1 | 9 | 9 | 9 |
| | vpr | 0 | 0 | 4 | 4 | 4 | 4 |

Table 5.2: Number of Loops Fused for SPEC2000 Benchmarks

| Benchmark | Base | MIC | Guard CTB | Guard RTB | align | non CFE |
|---|---|---|---|---|---|---|
| tomcatv | 1 | 2 | 2 | 2 | 2 | 2 |
| turb3d | 0 | 0 | 0 | 6 | 6 | 6 |
| compress | 1 | 1 | 1 | 1 | 2 | 2 |

Table 5.3: Number of Loops Fused for SPEC95 benchmarks

Fusing more loops may not always result in better performance. For example, consider four loops, $L_i$, $L_j$, $L_k$, and $L_m$, where loop $L_j$ cannot be fused with $L_k$ or $L_m$ because of fusion preventing dependencies. All other possible fusions are legal. Also, assume that there is data reuse between $L_i$ and $L_j$, but not between $L_i$, $L_k$ and $L_m$. In this example, fusing $L_i$, $L_k$ and $L_m$ would result in the most number of loops fused. However, fusing $L_i$ and $L_j$ may result in better run time performance because of the data reuse between the two loops.[1] There is currently

[1]Note that when $L_i$ is fused with $L_j$, the resulting loop contains the fusion preventing depen-

no mechanism in the loop fusion algorithm to weigh the benefit of fusing two loops. Thus, situations such as this one may not be handled optimally by the loop fusion algorithm. If an attempt was made to fuse $L_i$ and $L_j$ first, they would be fused and data reuse would result. If an attempt was made to fuse $L_i$ with either $L_k$ or $L_m$ first, the fusion would occur and $L_i$ and $L_j$ would not be fused. Work that is currently underway to attempt to identify these situations and deal with them appropriately is described in more detail in Chapter 8.

| | Benchmark | Base | MIC | Guard CTB | Guard RTB | align | non CFE |
|---|---|---|---|---|---|---|---|
| SPEC 2000 FP | ammp | 70.56 | 71.16 | 71.22 | 70.19 | 70.23 | 70.57 |
| | applu | 71.52 | 71.57 | 73.65 | 76.46 | 112.86 | 113.91 |
| | apsi | 70.34 | 70.81 | 70.76 | 70.79 | 70.57 | 71.76 |
| | art | 7.40 | 7.33 | 7.39 | 7.05 | 7.05 | 7.08 |
| | facerec | 46.75 | 47.00 | 46.85 | 47.59 | 47.48 | 48.21 |
| | fma3d | 512.85 | 518.06 | 543.75 | 594.32 | 591.07 | 600.59 |
| | galgel | 110.34 | 109.01 | 109.12 | 109.22 | 109.45 | 110.25 |
| | lucas | 16.47 | 16.85 | 16.98 | 16.82 | 16.88 | 16.83 |
| | mesa | 214.56 | 214.80 | 214.99 | 214.69 | 214.88 | 215.01 |
| | mgrid | 9.31 | 9.16 | 9.23 | 9.19 | 9.18 | 9.27 |
| | sixtrack | 1242.91 | 1246.47 | 1301.94 | 1330.44 | 1330.73 | 1384.83 |
| | swim | 4.48 | 4.5 | 4.54 | 4.5 | 5.03 | 5.01 |
| SPEC 2000 INT | bzip2 | 17.53 | 17.53 | 17.58 | 17.4 | 17.65 | 17.75 |
| | crafty | 78.90 | 79.01 | 357.30 | 358.26 | 360.52 | 359.16 |
| | eon | | 182.47 | 182.27 | 182.66 | 181.9 | 182.64 |
| | gap | 331.94 | 332.44 | 329.56 | 329.78 | 329.66 | 330.92 |
| | gcc | 625.58 | 627.54 | 926.92 | 926.19 | 926.97 | 930.55 |
| | gzip | 18.97 | 19.05 | 18.92 | 19.00 | 19.17 | 19.24 |
| | twolf | 112.37 | 112.82 | 112.63 | 111.73 | 112.17 | 112.59 |
| | vpr | 60.79 | 60.52 | 60.84 | 60.93 | 60.73 | 60.95 |

Table 5.4: Compilation Times for SPEC2000 Benchmarks

| Benchmark | Base | MIC | Guard CTB | Guard RTB | align | non CFE |
|---|---|---|---|---|---|---|
| tomcatv | 2.02 | 2.15 | 2.14 | 2.16 | 2.16 | 2.16 |
| turb3d | 15.57 | 15.78 | 15.6 | 17.66 | 17.75 | 17.66 |
| compress | 2.83 | 2.92 | 2.87 | 2.79 | 2.77 | 2.69 |

Table 5.5: Compile Times for SPEC95 Benchmarks

The compilation times for the benchmarks using each of the algorithms are listed

dencies present in $L_j$ and thus cannot be fused with either $L_k$ or $L_m$.

in Tables 5.4 and 5.5. These results are an average of three compilation runs. The compilation time for the benchmarks indicate the amount of overhead each algorithm adds to the compilation time. There was very little change in compilation time when code was moved to make loops adjacent. When guard branches were inserted to make non-conforming loops conform, the compilation time of *crafty* increased by 4.5 times, the compilation time for *gcc* increased by 48% and the compilation times of *fma3d* and *sixtrack* increased by 4%.

An analysis of the time spent in the actual loop fusion phase of the compiler revealed that it is not contributing significantly to the increase in compilation times. For example, during baseline compilation for *crafty*, the loop fusion phase took 0.165s and contributed 0.21% to the total compilation time. When guard branches were inserted, the loop fusion phase took 1.30s and contributed 0.36% to the total compilation time. Loop distribution, on the other hand, took 0.455s and contributed 0.56% to the total compilation time during baseline compilation. When guard branches were inserted, loop distribution took 256.27s, contributing 72.07% to the total compilation time. Thus, in *crafty*, the increase in compilation time can be attributed to the loop distribution phase, not the loop fusion phase.

Similar results were found for *applu*, whose compilation time increased by 53% when guard branches were inserted to make loops with negative distance dependencies align. During baseline compilation, 1.153s were spent in the loop fusion phase and 15.71s spent in the loop distribution phase. These times contributed 1.62% and 22.12% to the total compilation time respectively. When alignment was performed, 4.97s were spent in the loop fusion phase and 49.39s were spent in the loop distribution phase. These numbers contributed 4.38% and 43.63% to the total compilation time. Again, the increase in time spent in loop distribution is the major cause for the increase in compilation times.

These results indicate that the increase in compilation time is not because of an increased amount of time spent in loop fusion phase. However, this does not mean that loop fusion is not the cause of the increased compilation time. It is possible that other optimizations, such as loop distribution, require more time because of the loops created by loop fusion. For example, it is possible that more time is being spent in the loop distribution phase because loop fusion is fusing loops that should not be fused and the loop distributor requires extra time to determine the best way to split the loops. It is also possible that the size and structure of the loop bodies that are created by loop fusion are difficult for the loop distribution phase to manage, thus causing it to execute for longer. This seems likely in some cases because Index Set Splitting is not currently being used to remove the control flow inserted into the fused loop bodies. A more detailed analysis of the time spent in loop distribution is necessary to fully understand the cause of the compilation time increases.

## 5.5 Run Time Results

To answer question 4 we collected the run times of the benchmarks using the versions of the algorithm described in Section 5.1. These versions did not use the index-set splitting facility described in Chapter 2. Thus, all of the guard branches added into the loop bodies by the algorithms were not removed after loop fusion had completed.

| | Benchmark | Base | MIC | Guard | Guard | align | non | Best |
|---|---|---|---|---|---|---|---|---|
| | | | | CTB | RTB | | CFE | |
| SPEC 2000 FP | ammp | 488.74 | **487.98** | 488.59 | 491.15 | 491.53 | 490.67 | 0.16 % |
| | applu | 319.68 | 320.19 | 321.49 | 321.36 | 307.72 | **307.2** | 3.9 % |
| | apsi | 397.02 | 397.48 | **396.34** | 397.28 | 400.29 | 401.22 | 0.17 |
| | art | 217.60 | 223.82 | 223.1 | 221.71 | **220.73** | 220.95 | -1.43 |
| | facerec | 201.02 | 201.27 | **201.06** | 201.1 | 201.28 | 201.24 | -0.02 |
| | fma3d | 341.14 | **336.77** | 337.08 | 338.30 | 353.13 | 340.35 | 1.28% |
| | galgel | 180.71 | **179.13** | 179.83 | 179.32 | 180.59 | 179.81 | 0.87% |
| | lucas | 266.58 | **264.26** | 265.21 | 265.89 | 265.1 | 264.34 | 0.87% |
| | mesa | 331.30 | 331.65 | **328.21** | 331.90 | | 338.83 | 0.93% |
| | mgrid | 339.76 | **339.54** | 340.78 | 340.49 | 340.59 | 340.26 | 0.06% |
| | sixtrack | 265.32 | **265.54** | 268.15 | 266.42 | 266.81 | 268.1 | 0.08% |
| | swim | 274.03 | 273.86 | 274.04 | **273.82** | 274.07 | 274.41 | 0.08% |
| SPEC 2000 INT | bzip2 | 257.27 | **257.79** | 258.21 | 258.11 | 258.01 | 258.20 | -0.20% |
| | crafty | 186.35 | **189.27** | 192.35 | 198.41 | 195.40 | 195.84 | -1.57% |
| | gap | 246.31 | 242.28 | 236.03 | 236.03 | **235.23** | 241.57 | 4.50% |
| | gzip | 326.83 | **327.13** | 327.61 | 328.72 | 328.68 | 328.16 | -0.09% |
| | twolf | 435.84 | 440.48 | 438.48 | **433.50** | 442.42 | 440.04 | 0.54% |
| | vpr | 260.33 | 262.67 | 257.05 | **256.95** | 258.52 | 258.18 | 1.30% |

Table 5.6: SPEC2000 Run Time Results

Tables 5.6 and 5.7 present the run time results for the SPEC2000 and SPEC95 benchmark suites, compiled using the different algorithms. The results reported are the running times for each benchmark, averaged over three runs. The *Best* column displays the best run time improvement for each benchmark. The best run time for

| Benchmark | Base | MIC | Guard CTB | Guard RTB | align | non CFE | Best |
|---|---|---|---|---|---|---|---|
| tomcatv | 28.55 | **26.76** | 27.23 | 27.26 | 27.23 | 27.23 | 6.27% |
| turbo3d | 96.1 | **96.09** | 96.17 | 96.54 | 96.56 | 96.11 | 0.01% |
| compress | 46.09 | 46.19 | 46.09 | **46.05** | 46.36 | 46.21 | 0.09% |

Table 5.7: SPEC95 Run Time Results

each benchmark is in bold.

As seen in the Tables 5.6 and 5.7, the impact of increased loop fusion on the running times of the benchmarks is modest. The movement of intervening code to make loops adjacent results in a 2.8% decrease in performance for *art*. The use of guards to make two non-conforming loops, whose difference in bounds is known at compile time, conform results in a 4.2% improvement over the baseline in *gap* and a 3.2% decrease in performance over the baseline in *gap*. The use of guards to make two non-conforming loops, whose difference in bounds cannot be determined at compile time, conform results in a 6.5% decrease in performance over the baseline. The use of guard branches to make loops with negative distance dependencies align results in a 4.2% performance improvement over the baseline for *applu* and a 4.4% decrease in performance over the baseline for *fma3d*. The fusion of non-control flow equivalent loops results in a 2.3% decrease in performance over baseline for *mesa*.

There are two explanations for the mediocre impact of loop fusion on the running times of the benchmarks. First, since the index-set splitting optimization has not been integrated with loop fusion into the compiler framework, the guard branches introduced by several of the algorithms are not being removed. The presence of these guard branches inside the loop body could have an impact on the performance of other compiler optimizations, such as software pipelining. It is anticipated that when index-set splitting has been fully integrated with loop fusion and the the guard branches are removed, some of the benchmarks will benefit from increased run time performance.

The second explanation for the small changes in run times is the amount of time that the fused loops contribute to the total running time of the programs. For example, compiling *galgel* with the MIC option resulted in an additional 14 loops fused. However, there was no significant difference in the run time. Since the movement of intervening code does not introduce guard branches, the absence of index-set splitting cannot be attributed to the run time performance (*i.e.,* there are no guard branches introduced during fusion so there is no reason to expect index-set splitting will improve the run time performance). A run time performance analysis of *galgel* determined that 36.3% of the total execution time is spent in five functions: *SysNSN* (12.3%), *DGEMV* (7.8%), *DLARFX* (6.2%), *syshtn* (5.0%) and *dgemm* (5.0%). Only one loop was fused in all five of these functions. Thus, a large part of the execution time is spent in functions in which no loops were fused. This indicates, that the majority of loops fused are infrequently executed.

Similar results are found for *fma3d* when using runtime bounds checks to fuse loops (GuardRTB column). An additional 30 loops were fused, but there was no change in run time performance. A run time profile analysis was performed and revealed that 37.8% of total program execution time is spent in 6 functions: *solve*(8.0%), *platq_stress_integration*(7.5%), *khplq_gradient_operator*(7.1%), *platq_internal_forces*(6.9%), *khplq_divergence_operator*(4.4%) and *material_41_integration*(3.9%). No new loops were fused in any of these functions as a result of the runtime bounds check (*i.e.,* the same loops were fused when using the compile time bounds checks). Thus, none of the additional 30 loops fused were located in the functions responsible for 37.8% of the programs total execution. This also indicates that infrequently executed loops are being fused.

# Chapter 6

# Related Work

This work focuses on maximal loop fusion for the IBM XL compiler suite. Very little work has been published on maximal fusion followed by selective distribution. In contrast, there has been extensive studies and experimentation on loop fusion for parallelism, loop fusion to increase data reuse, loop fusion to minimize memory bandwidth, and weighted loop fusion.

Loop fusion to increase parallelism and increase data reuse as well as weighted loop fusion require that loop bodies be as small as possible. Thus, maximal loop distribution is generally performed prior to performing loop fusion. The formation of smaller loops removes constraints between statements originally located in the same loop body, giving the loop fusion algorithms more freedom to selectively fuse loops (group statements) based on a specific criterion.

## 6.1 Loop Fusion for Parallelism

Loop fusion to increase parallelism focuses on fusing loops such that the resulting loop bodies can be run in parallel. Running a loop body in parallel can have performance benefits if the loop iterates many times, or if the body of the loop is large. Loop fusion is used to increase the size of loop bodies and thus increase the benefits of running the loop in parallel. However, care must be taken while fusing loops to ensure that the fused loop can be run in parallel.

In their conversion of Kennedy and M$^{\mathrm{C}}$Kinley's research work [18] [19] to textbook material, Allen and Kennedy[1] identify the following two *safety constraints* for loop fusion:

1. *Fusion-preventing dependence constraint*: Two loops $L_i$ and $L_j$ cannot be correctly fused if there exists a fusion-preventing dependence between them.

2. *Ordering constraint*: Two loops cannot be correctly fused if there exists a path of loop-independent dependencies between them that contains a loop or statements that are not being fused with $L_i$ and $L_j$.

Constraint 1 corresponds to the fusion preventing dependence criterion and constraint 2 corresponds to the adjacent criterion identified in Section 4.1.2. They mention other constraints such as conforming bounds, but do not discuss how to handle programs where these other constraints are not satisfied.

Allen and Kennedy[1] identify what they call a "parallelism inhibiting" dependence constraint that can effect the profitability of fusing two loops:

**Definition 6.1.1** *An edge between statements in two different loops is said to be parallelism inhibiting if, after fusing the two loops, the dependence would be carried by the combined loop.*

Allen and Kennedy[1] then identify the following two profitability constraints for loop fusion:

1. *Separation constraint*: A sequential loop should not be fused with a parallel loop because the result would necessarily be sequential, reducing the total amount of parallelism.

2. *Parallelism-inhibiting dependence constraint*: Two parallel loops should not be fused if there exists a parallel-inhibiting dependence between them.

### 6.1.1 Typed Fusion

In typed fusion, there is a strict distinction between parallel and sequential loops. Typed fusion attempts to create the largest possible parallel loops using loop fusion. It does not attempt to determine when sequential loops might be run in parallel with other sequential loops or with parallel loops.

Allen and Kennedy define the typed fusion problem in terms of graph $G$, containing nodes $N$ and edges $E$. Nodes in the graph represent loops. Every node has an associated type; the type of a node can be either parallel or sequential. Edges in the graph represent dependencies. Each graph has a set of *bad edges* that contain fusion preventing or parallel-inhibiting edges. That is, a bad edge $e_b$ between two nodes $n_i$ and $n_j$ indicates that there is either a fusion preventing dependence or a parallel-inhibiting dependence between the loops represented by $n_i$ and $n_j$.

They define a solution for the typed fusion problem as a graph $G' = (N', E')$ where $N'$ is obtained by fusing nodes of a specific type, subject to the following constraints:

1. *Bad edge constraint*: no two nodes connected by a bad edge can be fused.
2. *Ordering constraint*: no two nodes joined by a path containing a node that is not of the specified type can be fused.

By specifying the type to be parallel loops, the typed fusion problem formulated above can be used to maximize the number of parallel loops that are created by loop fusion. That is, an optimal solution to the typed fusion problem has the smallest number of parallel loops. Kennedy and M$^{C}$Kinley presented an optimal greedy algorithm to the typed fusion problem[19].

### 6.1.2 Unordered and Ordered Typed Fusion

It is possible to extend the types in typed fusion to represent other constraints of fusion. For example, the iteration counts of loops can be introduced as types - two loops with different iteration counts would have different types. This would not provide a mechanism to make the two loops conform, but would fit into the typed fusion framework for selecting loops to fuse.

In typed fusion, when there are more than two types to consider, the order in which loop types are fused can determine how many loops are fused. Allen and Kennedy[1] give the following example to illustrate the importance of type order during typed fusion with more than two types:

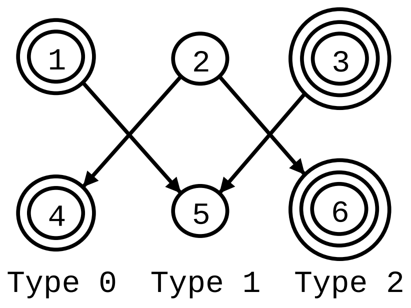
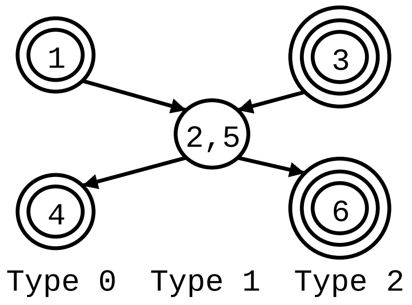
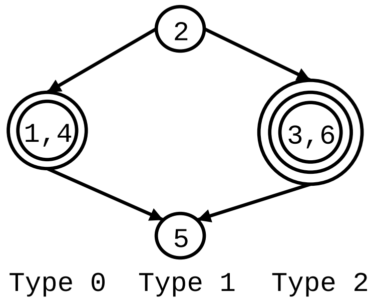


Figure 6.1: Order-sensitive Fusion Problem

In Figure 6.1, there are fusion preventing edges between loops 1 and 5, loops 2 and 4, loops 2 and 6 and loops 3 and 5. Two loops that have a fusion preventing edge between them cannot be fused. When two loops are fused, the resulting loop contains the outgoing and incoming edges of all of the original loops.



(a) Result of fusing type 1 first



(b) Result of fusing type 0 first

Figure 6.2: Order-sensitive Fusion Problem

Figure 6.2(a), demonstrates the loops that are created when loops of type 1 in Figure 6.1 are fused first. The result is the creation of bad paths for both type 0 and type 2, meaning that no more loops can be fused. Thus, if type 1 is fused first, the result contains 5 loops. However, if type 0 is fused first, a bad path is created only for type 1. This allows type 2 to be fused also, resulting in the loops shown in Figure 6.2(b). The result of fusing type 0 first is more successful, containing only 4 loops.

Kennedy and M$^{c}$Kinley used a polynomial reduction of the Vertex Cover problem to show that the unordered type fusion problem is NP-Hard[18].

The *ordered typed fusion* problem places priorities on the types. This allows types of higher priority to be considered for fusion before types of lower priority. This is equivalent to saying that it is more important to minimize the number of nodes of a higher priority than of a lower priority type. Kennedy and M$^{c}$Kinley provide a greedy algorithm to solve the ordered typed fusion problem by treating each type, in order of priority, as a typed fusion problem[18].

## 6.2 Loop Fusion for Data Reuse

Loop fusion to increase data reuse can be divided into two separate areas. Loop fusion has been used to increase the register reuse within a loop body. It has also been used to increase the data cache performance by utilizing data as much as possible before it is expunged from the cache.

On modern, superscalar processors, data must be moved from memory into hardware registers before being used in a computation. While moving data from a cache into a register is a relatively inexpensive operation (compared to moving data from lower levels to higher levels in the memory hierarchy), it still requires time to perform. Loop fusion has been used to try and reduce the number of loads from data cache into hardware registers by placing computations from separate loop bodies that use the same data into a single loop body. This reduces the number of loads from data cache to hardware registers, resulting in a more efficient use of the data.

Moving data between levels in the memory hierarchy can be an expensive operation, depending on which level of the memory hierarchy contains the data. As you move down through the memory hierarchy, transferring data to higher levels becomes increasingly more expensive. Loop fusion has been used to increase data cache performance by utilizing data as much as possible before it is expunged from the cache. This reduces the amount of data that has to be retrieved from a lower level in the memory hierarchy. The cost of retrieving data increases significantly as you move lower in the memory hierarchy, allowing data reuse to have a significant impact on the runtime performance of many programs.

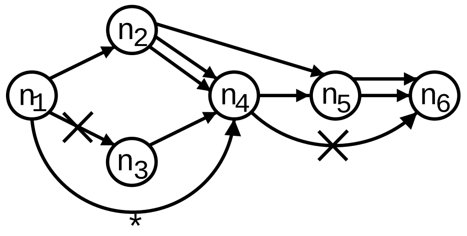
### 6.2.1 Loop Dependence Graph

A Loop Dependence Graph (LDG) is a directed multigraph that represents a single entry single exit region containing $k$ perfect loop nests, introduced by Ferrante *et al.* [12]. Every loop in the LDG must be control flow equivalent and have conformable bounds. A node in the LDG represents a loop nest. A directed edge from a node $n_i$ to $n_j$ represents a loop independent data dependence from node $n_i$ to node $n_j$. Every edge in the LDG is marked as being *fusible* or *non-fusible*. Two nodes $n_i$ and $n_j$ connected by a non-fusible edge cannot be fused because doing so would violate the data dependence test for loop fusion. Fusible edges in the LDG are further classified as being *contractable* or *non-contractable*. A fusible contractable edge between two nodes $n_i$ and $n_j$ signifies there is data reuse between statements in node $n_i$ and node $n_j$ while a fusible non-contractable edge means no data reuse exists. An LDG edge can represent flow, output or anti-dependencies between two nodes, however, only edges representing flow dependencies can be marked as contractable.

Gao *et al.*[14] provide a sample FORTRAN code fragment to illustrate a simple

LDG. This sample code has been translated to C and modified slightly to make it consistent with the other examples presented in this document. Figure 6.3 shows the C code fragment (a) and the resulting LDG (b). Edges in the LDG marked with an X represent nonfusable edges while edges marked with an $*$ represent fusible-noncontractaable edges. All other edges are fusible and contractable.

```
L1: for (i=0; i < N; i++)
    {
      A[i] = E[i];
    }
L2: for (i=0; i < N; i++)
    {
      B[i] = A[i] * 3;
      C[i] = B[i] + 10;
    }
L3: for (i=0; i < N; i++)
    {
      D[i] = A[i+1] * 2;
    }
L4: for (i=0; i < N; i++)
    {
      E[i] = B[i] C[i]*D[i];
    }
L5: for (i=0; i < N; i++)
    {
      F[i] = B[i] * 2;
      G[i] = E[i] - F[i];
    }
L6: for (i=0; i < N; i++)
    {
      H[i] = F[i] + G[i] * E[i+1];
    }
```



(a) A C code fragment (b) The LDG of C code fragment

Figure 6.3: An Example LDG for a Given Code Fragment

In Figure 6.3, the nonfusable edge from node $n_1$ to node $n_3$ is a result of the fusion-preventing dependence between L1 and L3. The nonfusable edge from node $n_4$ to node $n_6$ exists for the same reason. The non-contractable edge from node $n_1$ to $n_4$ is a result of the anti-dependence on array E in loops L1 and L4.

Gao *et al.* use loop fusion to maximize data reuse within a loop. This maximization allows the same register to be used for multiple computations of the same data, saving the cost of loading the same data into a register in multiple loop bodies.

Gao *et al.* proposed an algorithm based on the max-flow/min-cut algorithm [14]. Their algorithm works in two stages. The first stage identifies the minimum number of clusters required for a solution and determines in which cluster each node can legally be placed. The number of required clusters is determined by performing a topological traversal of the LDG, assigning each node to a cluster based on the edges in the graph and the other nodes already in the cluster.

Finally, a reduced graph $G'$ is created by collapsing all clusters into nodes in $G'$. Nodes from the LDG that could only reside in one cluster are contained within the new cluster nodes. Nodes that could reside in multiple clusters are left as regular nodes in $G'$.

Gao *et al.* focus on solving the problem of moving data between caches and registers. Our approach can benefit the data cache performance, but is not driven by the performance benefits that would result. Also, there is less of a focus in our approach on register reuse for data within a loop body.

Allen and Kennedy also discuss loop fusion to improve register reuse [1]. They identify dependences between statements $S_i$ and $S_j$ in loops $L_i$ and $L_j$ as profitable loop fusion for register reuse if $(i)$ statements $S_i$ and $S_j$ remain independent when $L_i$ and $L_j$ and fused, or $(ii)$ statements $S_i$ and $S_j$ become forward loop-carried dependences when $L_i$ and $L_j$ are used.

```
for (i=0; i < n; i++)
{
  A[i] = B[i] + C[i] * 2;
}
for (i=0; i < n; i++)
{
  D[i] = B[i] + C[i] + 10;
}
```

(a) Independent statements

```
for (i=0; i < n; i++)
{
  A[i] = B[i] + C[i] * 2;
  D[i] = B[i] + C[i] + 10;
}
```

(b) After loop fusion

Figure 6.4: Fusion of Loops Containing Independent Statements

Condition $(i)$ may create the potential for register reuse within a loop iteration if loop fusion is performed. Figure 6.4 provides an example of two loops containing a loop independent dependence. In Figure 6.4(a), the values of `B[i]` and `C[i]` are used in both loop bodies. Thus, two values need to be loaded from memory into registers for every iteration of both loop bodies. Figure 6.4(b) shows the loop created after loop fusion. In this loop, it is only necessary to load two values from memory for every iteration of the loop body. Thus, through the reuse of registers in the fused loop, the number of memory loads into registers has been reduced by 2n.

Condition $(ii)$ may create the potential for register reuse across loop iterations

```
for (i=1; i < n; i++)
{
  A[i] = B[i] + 2;
}
for (i=1; i < n; i++)
{
  C[i] = A[i-1] * 3;
}
```

(a) Forward loop carried dependence

```
for (i=1; i < n; i++)
{
  C[i] = A[i-1] * 3;
  A[i] = B[i] + 2;
}
```

(b) Loop fusion and reordering of statements

Figure 6.5: Fusion of Loops Containing Forward Loop-carried Dependencies

if statements $S_i$ and $S_j$ can be reordered. The two original loops in Figure 6.5(a) contain a forward loop-carried dependence to array A. Before fusion, it is necessary to load the values of `A[i]` and `A[i-1]` from memory into registers for every iteration of the first and second loops respectively. After fusion, and reordering of statements, the value of `A[i]` can be used in the next iteration for the value of `A[i-1]`, thus saving a load from memory. It is necessary to reorder the two statements in the fused loop to create the register reuse scenario. However, statement order is not a factor in preserving loop-carried dependencies thus the statements can be safely reordered.

Allen and Kennedy also provide algorithms to fuse loops, using loop alignment and loop peeling to remove fusion preventing dependencies and maximize the amount of register reuse in the fused loop body[1]. The technique of using loop peeling to align loops containing fusion preventing dependencies is very similar to the technique presented in Section 4.5 to align loops by guarding iterations. A fusion preventing dependence from statement $S_i$ to statement $S_j$, indicates that $S_j$ uses a value that is created by $S_i$ in a future iteration. The goal is to align the loops so that the creation of the value in $S_i$ is on the same iteration as the use of the value in $S_j$. This also maximizes the amount of register reuse within the fused loop.

Their algorithms work an a set of loops that have varying upper and lower bounds. The set of loops must be in the order in which the loops should be executed. The result consists of three components. First is a series of pre-loops, consisting of the peeled iterations from the beginning of all loops that require peeling. Second is the fused loop body, which contains the common parts of all loops. Third is a series of post-loop bodies that contain the remaining iterations of all the loop bodies. The key point to this algorithm is that it can take a set of loops with varying lower and upper bounds and create a single fused loop body and a series of pre-loops and post-loops. In our algorithms, all loops considered for fusion have been normalized and thus all have the same lower bounds. Furthermore, the input to Allen and Kennedy's algorithm is a set of loops to be fused, in the order in which they should be fused. The algorithm does not determine if fusing the loops is legal nor attempt to make loops fusible.

## 6.3 Bandwidth-minimal Loop Fusion

The increasing difference between processor and memory hierarchy speeds has become the primary limiting factor in program performance [10]. This observation has motivated research that focuses on reusing data loaded into the data cache as much as possible before it is removed from the cache. By increasing the amount of data reuse from the data cache, the amount of data transferred between levels in the memory hierarchy is reduced resulting in an increase in the data cache performance.

Data cache performance is similar to register reuse in that it attempts to reuse data. However, granularity of the data that is being reused is slightly coarser. In order to increase register reuse, the same data must be used in multiple statements within a loop body. In order to increase data cache performance, data from the same cache line must be used before the data is discarded. For example, if the array accesses in two loop bodies are not aligned, but the loops can still be fused, it may not be possible to reuse registers in the fused loop. However, it is likely that the fused loop will benefit from increased data cache performance because parts of the array will stay in the data cache through several iterations of the loop.

```
for (i=0; i < n; i++)
{
  B[i] = A[i] + 10;
}
for (i=0; i < n; i++)
{
  C[i] = A[i] * 2;
}
```

(a) Original loops

```
for (i=0; i < n; i++)
{
  B[i] = A[i] + 10;
  C[i] = A[i] * 2;
}
```

(b) After loop fusion

Figure 6.6: Loop Fusion to Illustrate Data Cache Performance

Loop fusion will also benefit from data cache performance if the array accessed by the two loops is larger than cache. Figure 6.6 shows an example of two loops, before and after loop fusion. Assume that `n` is larger than the cache. Every iteration of the first loop in Figure 6.6(a) will result in a data cache miss. When the first loop completes, the entire cache will be filled with values from the `A` array, but the values required at the beginning of the second loop will not be present because they have been overwritten by values from later iterations. Thus, every iteration of the second loop will also incur a cache miss. The fused loop in Figure 6.6(b) will incur a cache miss at the beginning of each iteration while loading the value from the `A` array. However, the second use of this value in fused loop will result in a cache hit. Thus, the fused loop has reduced the number of cache misses by a factor of `n`.

Ding and Kennedy proposed a *Bandwidth-minimal loop fusion* to minimize memory transfer of data and provided a polynomial solution to a restricted form of the problem [10]. They also provided a polynomial reduction of the $k$-way cut problem to the Bandwidth-minimal fusion problem to show that the unrestricted problem is NP-hard.

The Bandwidth-minimal loop fusion problem is modeled using a *fusion graph*, based on the Loop Dependence Graph used by Gao *et al.*. Nodes in the graph represent loops, directed edges model data dependencies and undirected edges represent fusion-preventing constraints. Ding and Kennedy define the objective for fusion is stated as follows [10]:

Given a fusion graph, how can we divide the nodes into a sequence of partitions such that

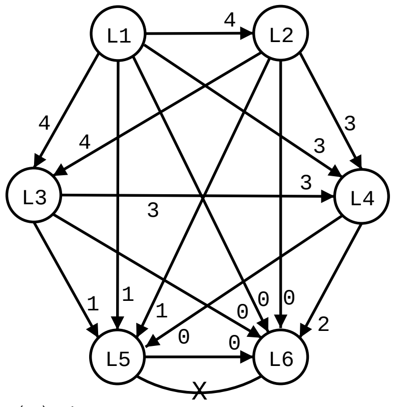
- (Correctness) each node appears in one and only one partition; the nodes in each partition have no fusion-preventing constraint among them; and dependence edges flow only from an earlier partition to a later partition in the sequence.
- (Optimality) the sum of the number of distinct arrays in all partitions is minimal

Ding and Kennedy present the following example to illustrate that the data-reuse problem formulation of Gao *et al.* and Kennedy and M$^{\text{C}}$Kinley is different from their bandwidth-minimal loop fusion[10].

Figure 6.7(a) shows the data accessed by 6 different loops. Letters `A, B, C, D, E` and `F` represent arrays, while sum is a scalar quantity. Figure 6.7(b) shows the

| Loop | Array Accesses |
|---|---|
| 1 | A, D, E, F |
| 2 | A, D, E, F |
| 3 | A, D, E, F |
| 4 | B, C, D, E, F |
| 5 | A, sum |
| 6 | B, C, sum |

(a) Array accesses in loops



(b) Associated Fusion Graph

Figure 6.7: Fusion Graph Example

associated fusion graphs for the loops listed in Figure 6.7(b). The weights on the directed edges represent the number of shared arrays between the two loops. For example, the edge from L1 to L2 has a weight of 4 because the two loops access the same 4 arrays. Similarly, the edge from L1 to L4 has a weight of 3 because they only access three of the same arrays. The undirected edge from L5 to L6 represents a fusion-preventing dependence between L5 and L6, meaning that the two loops cannot be fused together or cannot be located in the same fusion cluster.

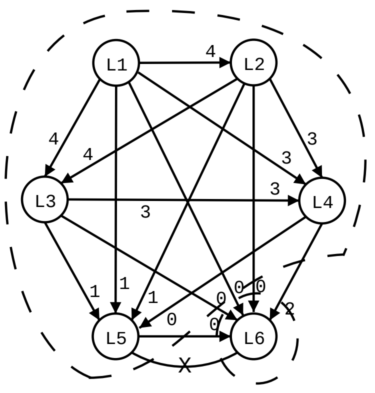


Figure 6.8: Loop Fusion to Maximize Data Reuse

Figure 6.8 shows the partition that would be formed using the algorithms Gao *et al.*and Kennedy and M$^{\text{C}}$Kinley. Their approach partitions the nodes to minimize the total weight of edges between partitions. The resulting partitions group nodes L1, L2, L3, L4 and L5 into a single group, leaving L6 to form the second partition. This results in a total cross-partition edge weight of 2, which is the lowest possible for this graph. However, it requires 8 arrays to be loaded.

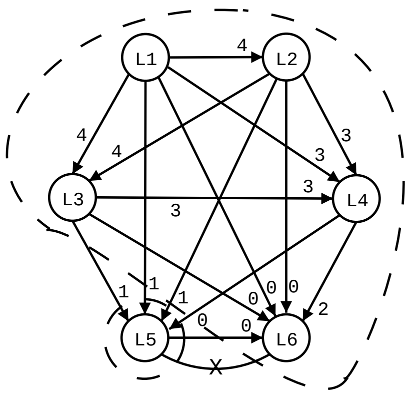


Figure 6.9: Loop Fusion to Maximize Data Reuse

Figure 6.9 shows the partitions that would minimize the bandwidth consumption. This partition only requires 7 arrays to be loaded, but has slightly less data reuse as a result. Also, the cross-partition edge weight total is 3, which is not optimal based on the formulation by Gao *et al.*and Kennedy and M$^{C}$Kinley.

Ding and Kennedy argue that in order to correctly represent data sharing between loops, a *hypergraph* is necessary. Since multiple loops can share the same data, traditional edges in a graph cannot fully represent the data sharing relationship between loops. Instead, a single *hyper-edge* is used to connect multiple nodes in the graph. A graph containing hyper-edges is called a hyper-graph [10].

The Bandwidth-minimal loop fusion problem can be formulated in terms of hypergraphs by adding hyper-edges to a fusion graph. Each hyper-edge connects all loops that access a specific array. The length of the hyper-edge corresponds to the number of partitions the edge is connected to after partitioning. The goal of Bandwidth-minimal loop fusion is to minimize the total length of all hyper-edges.

## 6.4 Weighted Loop Fusion

Weighted loop fusion provides a framework for selecting groups of loops to be fused. This selection is based on profitability of performing loop fusion on a set of loops. Data reuse is the most common measurement of profitability, however other criterion such as parallelism or resulting code size could also be used. The weighted loop fusion problem uses a weighted graph to represent profitability for fusing different pairs of loops. Nodes in the graph represent loops that can be fused. Edges in the graph have two meanings, depending on the value of their weights. An edge from node $N_i$ to node $N_j$ with positive weight (*i.e.,* weight $\geq 0$) represents the potential gain from fusing $N_i$ and $N_j$. An edge from node $N_i$ to node $N_j$ with negative weight (*i.e.,* weight $< 0$) indicates that $N_i$ and $N_j$ cannot be fused (*i.e.,* there is a fusion preventing dependence between $N_i$ and $N_j$). Given a weighted graph representing fusion potentials, the goal of weighted loop fusion is to group the nodes into clusters in a way that minimizes total weight of positive edges that cross cluster boundaries while ensuring that no cluster contains an edge with negative weight.

Kennedy and M$^{\mathrm{C}}$Kinley used a polynomial reduction of the Multiway Cut problem to prove that solving the weighted loop fusion problem is NP-Hard. They also provide a greedy algorithm and a variation of the max-flow/min-cut algorithm to find approximated solutions for the weighted loop fusion problem [18].

Megiddo and Sarkar presented an integer programming formulation for the weighted loop fusion problem. The number of variables and constraints in the formulation is linearly proportional to the size of the weighted loop fusion problem. They also provide a custom branch-and-bound algorithm that can be used to solve their integer programming formulation [27].

# Chapter 7

# Conclusions

While there has been extensive work done on loop fusion for a variety of scenarios, scant work has been done on methods to handle the conditions surrounding loop fusion. This work presented algorithms to test and move intervening code from between two loops, thus making them adjacent. Alternative techniques to make loops with different bounds conform and loops with fusion preventing dependencies align were also presented. These alternative techniques use guard branches placed within the fused loop body to limit the execution of one of the original loop bodies. This technique does not effect future loop fusion attempts and the guard branches can be removed after loop fusion has completed using index set splitting.

An algorithm to fuse loops that are not control flow equivalent was also presented. This algorithms copies the unconditionally executed loop into both control flow paths, allowing the two loops in one control flow path to be fused while still maintaining program correctness should the other control flow path be taken.

Finally, an algorithm to identify and correct the conditions preventing loop fusion was given. This algorithm determines if code transformations can be used to modify a program in order to fuse pairs of loops. No code transformations are performed if all conditions necessary for fusion cannot be satisfied, thus limiting the amount of unnecessary transformations performed. The loop fusion algorithm uses the dominator/postdominator relationships between control flow equivalent loops to enforce an ordered traversal of fusion candidates. This reduces the number of times internal data structures are built and lowers the overall compilation time of the program.

The results presented in Section 5.4 show the effect of the different algorithms on the number of loops fused. The benchmarks *fma3d*, *galgel* and *crafty* all benefitted significantly from the movement of intervening code from between two loops. Additional loops were fused in *fma3d*, *galgel* and *crafty* as a result of adding compile time guards to the loops to make the bounds conform. *Vpr* saw 4 loops fused where none were fused before as a result of the guardCTB algorithm.

The number of successfully fused loops increased in *fma3d*, *galgel*, *sixtrack*, *gzip* and *twolf* as a result of using run time bounds checking to fuse loops. *Applu* saw a dramatic increase in the number of loops fused when guards were added to align loops and remove negative distance dependencies from between loop bodies. *Apsi*, *fma3d*, *sixtrack*, *swim* and *crafty* each had a single additional fused loop as a result of loop alignment. Finally, *apsi* and *sixtrack* both saw a significant increase in the number of loops fused when non-control flow equivalent loops were fused. *Fma3d* and *mesa* also saw a small increase in the number of fused loops when non-control flow equivalent loops were fused.

The effects of each algorithm on the compilation time of the benchmarks was also reported. From this data, we see that in some cases the algorithms do increase the compilation time of the benchmarks. However, the increase in compilation time is not extreme, thus allowing the loop fusion algorithms to be applied in a commercial setting.

The run time results are not as impressive as the compile time results. There are three possible reasons for this. First, it is possible that many of the loops being fused are cold loops. In several benchmarks, cross referencing was performed to associate hot functions with the number of loops fused in those functions. It was determined that in several cases the majority of loops being fused were located in cold functions and thus not executed frequently. Second, the index set splitting optimization is not being used after loop fusion has completed. Thus, the loops fused with the FuseWithGuard and FuseAndAlign algorithms still contain the control flow used to make the loops conform and align. It is hoped that once the interface to allow index set splitting to be used on loops created by loop fusion will be finished soon and that once control flow is removed from the loops a performance increase will be seen.

The third possible explanation for the meager run time results is that the loop distributor may not be dealing with the loop created by loop fusion ideally. The loop fusion mechanism is now fusing significantly more loops and may be creating opportunities that have never been encountered by the distributor. Thus, it is possible that the distributor is not splitting the loops in the best way.

# Chapter 8

# Future Work

## 8.1 Loop Optimization Framework

The loop optimization sequence in TPO is designed to perform maximal loop fusion followed by selective distribution. The algorithms presented here were designed to fuse as many loops as possible, without placing priority on fusing some pairs of loops over other pairs of loops. Instead, all possible pairs of loops are fused. This is done with the knowledge that the loop distributor will "undo" any bad choices that loop fusion made *i.e.,* loop distribution will separate a loop body into different loops if keeping them together has a negative effect on program performance.

As the algorithm for loop fusion evolved, more aggressive loop fusion was performed. Specifically, the fusing of loops whose difference in bounds could not be determined at compile time and fusing non-control flow equivalent loops. Both of these algorithms require the duplication of code to ensure program correctness. It became necessary to limit the code growth resulting from these algorithms to keep the compile time of the program within an acceptable limit. By limiting the amount of code growth, maximal loop fusion could no longer be performed. Thus, it seems prudent to use heuristics to guide loop fusion, especially in the cases where the total number of allowable loop fusions is limited.

These heuristics should include knowledge of loop structures that will benefit subsequent loop optimizations *e.g.,* perfect loop nests. The heuristics should also be aware of the situations that the loop distributor can properly deal with and ensure that loop fusion does not fuse two loops that cannot be distributed by the distributor. This leads to a tighter coupling between the loop fusion algorithm and the loop distribution algorithms. The loop fusion algorithms should be aware of what the loop distributor does and use this knowledge to help select the loops that will be fused.

## 8.2 Interprocedural Loop Fusion

TPO allows the loop optimizations to be run at either compile time or link time. Running loop optimizations at compile time means that the scope of the optimizations are limited to a procedure. Running the loop optimizations at link time means the algorithms can be applied at an interprocedural scope. The loop fusion algorithms presented here are not limited to work on an interprocedural scope. While the results presented in this document were taken from an intraprocuderal scope (*i.e.,* algorithms were run at compile time) they do not require an intraprocuderal scope.

Running the algorithms at link time will create the potential for loops located in separate functions to be fused.

One way to allow loops from different procedures to be fused would be to have the potential for loop fusion to guide the function inlining routine. For example, assume a function $X$ contains a loop $L_j$ and a function $Y$ contains a loop $L_k$. If $Y$ was called from $X$ and if $L_i$ and $L_j$ could be fused, the inlining mechanism would be more inclined to inline $Y$ into $X$ (copy the body of $Y$ into $X$, thereby eliminating the function call) and allow $L_j$ and $L_k$ to be fused.

It would be necessary to add checks that determine if two loops in separate procedures can be fused. This would prevent a function from being inlined only to determine that the two loops could not be fused. It would also be beneficial to have tests that determine the profitability of fusing two loops. This can help the inlining mechanism determine if benefits of inlining the function and fusing the loops outweigh the disadvantages of inlining *i.e.,* data reuse within the fused loop vs the code growth from inlining the function.

# Bibliography

[1] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures.* Morgan Kaufmann Publishers, 2002.

[2] David F. Bacon, Susan L. Graham, and Oliver 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] Bob Blainey, Christopher Barton, and José Nelson Amaral. Removing impediments to loop fusion through code transformations. In *15th Workshop on Languages and Compilers for Parallel Computing*, pages 271–281, College Park, Maryland, 2002.

[5] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. Compiler optimizations for improving data locality. In Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 252–262, San Jose, California, 1994.

[6] Intel Corporation. *Intel C++ Compiler User's Guide.* Intel Corporation, 1996-2002.

[7] Alain Darte. On the complexity of loop fusion. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 149–157, 1999.

[8] K. Diefendorff. Best new technology: POWER4 — IBM's chip multiprocessor is analysts' choice for technology award. In *Microprocessor Report: The Insider's Guide to Microprocessor Hardware*, February 2000.

[9] Chen Ding. *Improving Effective Bandwidth through Compiler Enhancement of Global and Dynamic Cache Reuse.* PhD thesis, Department of Computer Science, Rice University, January 2000.

[10] Chen Ding and Ken 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.

[11] Chen Ding and Ken Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. In *International Parallel and Distributed Processing Symposium (CD-ROM)*, San Francisco, CA, April 2001.

[12] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Language Systems*, 9(3):319–349, 1987.

[13] Amit Ganesh. Fusing loops with backward inter loop data dependence. *ACM SIGPLAN Notices*, 29(12):25–30, 1994.

[14] 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.

[15] Rajiv Gupta and Rastislav Bodik. Adaptive loop transformations for scientific programs. In *IEEE Symposium on Parallel and Distributed Processing*, pages 368–375, San Antonio, Texas, October 1995.

[16] Bor-Ming Hsieh, Michael Hind, and Ron Cytron. Loop distribution with multiple exits. In *Proceedings of Supercomputing*, pages 204–213, November 1992.

[17] Ken Kennedy and Kathryn S. McKinley. Loop distribution with arbitrary control flow. In *Proceedings of Supercomputing*, pages 407–417. IEEE Computer Society Press, November 1990.

[18] 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.

[19] Ken Kennedy and Kathryn 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.

[20] 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.

[21] David J. Kuck. A survey of parallel machine organization and programming. *ACM Computing Surveys*, 9(1):29–59, March 1977.

[22] Chi-Chung Lam, Daniel Cociorva, Gerald Baumgartner, and P. Sadayappan. Optimization of memory usage and communication requirements for a class of loops implementing multi-dimensional integrals. In *12th International Workshop on Languages and Compilers for Parallel Computing*, pages 350–364, La Jolla, San Diego, 1999.

[23] E. Christopher Lewis, Calvin Lin, and Lawrence Snyder. The implementation and evaluation of fusion and contraction in array languages. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 50–59, Montreal, Canada, 1998.

[24] Amy W. Lim, Shih-Wei Liao, and Monica 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.

[25] N. Manjikian and Tarek S. Abdelrahman. Fusion of loops for parallelism and locality. *IEEE Transactions on Parallel and Distributed Systems*, 8(2):193–209, 1997.

[26] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[27] Nimrod Megiddo and Vivek Sarkar. Optimal weighted loop fusion for parallel programs. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 282–291, 1997.

[28] Steven S. Muchnick. *Advanced Compiler Design & Implementation.* Morgan Kaufmann Publishers, 1997.

[29] Gabriel Rivera and Chau-Wen Tseng. Locality optimizations for multi-level caches. In *ACM International Conference on Supercomputing (CDROM)*, 1999.

[30] Gerald Roth and Ken Kennedy. Loop fusion in high performance FORTRAN. In *ACM International Conference on Supercomputing*, pages 125–132, 1998.

[31] Edwin H.-M. Sha, Timothy W. O'Neil, and Nelson L. Passos. Efficient polynomial-time nested loop fusion with full parallelism. *International Journal of Computers and Their Applications*, In Press.

[32] Edwin Hsing-Mean Sha, Chenhua Lang, and Nelson L. Passos. Polynomial-time nested loop fusion with full parallelism. In *International Conference on Parallel Processing - Vol. 3*, pages 9–16, 1996.

[33] Sharad Singhai and Kathryn S. McKinley. A parameterized loop fusion algorithm for improving parallelism and cache locality. *The Computer Journal*, 40(6):340–355, 1997.

[34] Yonghong Song Sun, Rong Xu, Cheng Wang, and Zhiyuan Li. Data locality enhancement by memory reduction. In *ACM International Conference on Supercomputing*, pages 50–64, June 2001.

[35] Michael E. Wolf, Dror E. Maydan, and Ding-Kai Chen. Combining loop transformations considering caches and scheduling. In *IEEE/ACM International Symposium on Microarchitecture*, pages 274–287, Paris, France, December 1996.

[36] Michael Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Publishing Company, 1996.

# Appendix A

# Additional Results

Table A.1 presents additional results collected using the algorithms described in Chapter 4. Each algorithm was applied individually to the benchmarks. The best performance for each benchmark is in bold. The *Best* column gives the best performance improvement for the benchmark.

| | Benchmark | Base | MIC | Guard CTB | Guard RTB | align | non CFE | Best |
|---|---|---|---|---|---|---|---|---|
| SPEC 2000 FP | ammp | 488.74 | 487.98 | 487.42 | 488.01 | **486.55** | 488.95 | 0.45% |
| | applu | 319.68 | 320.19 | 320.39 | 319.81 | 319.82 | **319.74** | -0.02% |
| | apsi | 397.02 | 397.48 | 397.03 | 396.71 | 396.28 | **395.43** | 0.40% |
| | art | 217.60 | 223.82 | 221.46 | 220.29 | **219.81** | 221.29 | -1.02% |
| | facerec | 201.02 | 201.27 | 201.12 | 200.79 | 201.32 | **200.66** | 0.18% |
| | fma3d | 341.14 | 336.77 | **334.16** | 340.64 | 340.63 | 339.88 | 2.05% |
| | galgel | 180.71 | **179.13** | 179.56 | 179.76 | 179.74 | 179.53 | 0.87% |
| | lucas | 266.58 | **264.26** | 265.96 | 265.87 | 266.11 | 265.93 | 0.87% |
| | mesa | 331.30 | 331.65 | **330.04** | 331.21 | 332.18 | 332.41 | 0.38% |
| | mgrid | 339.76 | 339.54 | 339.71 | **339.48** | 340.11 | 339.61 | 0.08% |
| | sixtrack | 265.32 | 265.54 | 268.46 | **265.36** | 265.46 | 267.99 | -0.02% |
| | swim | 274.03 | 273.86 | 274.05 | 273.66 | **273.64** | 274.21 | -0.22% |
| SPEC 2000 INT | bzip2 | 257.27 | 257.79 | 256.55 | **256.34** | 256.54 | 256.45 | 0.36% |
| | crafty | 186.35 | 189.27 | **185.35** | 186.56 | 186.15 | 186.41 | 0.54% |
| | gap | 246.31 | 242.28 | **236.57** | 243.30 | 243.25 | 238.19 | 3.95% |
| | gzip | 326.83 | 327.13 | 328.29 | 326.79 | 326.84 | **326.53** | 0.09% |
| | twolf | 435.84 | 440.48 | 439.82 | 442.26 | 440.43 | **436.86** | -0.23% |
| | vpr | 260.33 | 262.67 | 266.01 | **260.41** | 261.63 | 261.46 | -0.03% |

Table A.1: SPEC2000 Run Time Results

# Appendix B

# Trademarks