Software Methods for Improvement of Cache Performance on Supercomputer Applications

Allan K. Porterfield

CRPC-TR89009
May 1989

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

Software Methods for Improvement of Cache Performance on Supercomputer Applications

Allan Kennedy Porterfield

Abstract

The performance of many supercomputer programs is limited by the ability to get data from the memory to the processing elements. Increasing bandwidth and decreasing the latency of memory are major concerns of supercomputer architects. Compiler techniques to improve the memory performance of some memory levels have been developed, but no study of the performance and methods to improve the performance of data caches has been reported.

Measuring actual supercomputer cache performance has not been accomplished with previous cache simulation methods. Trace files for programs that take any significant execution time are too large to store. **PFC-Sim** is a program-driven event tracing facility that can simulate data cache performance of very long programs. PFC-Sim simulates cache concurrently with program execution, allowing very long traces to be used, since the trace entries are discarded as quickly as they are generated. Some of these programs have traces in excess of 4 billion entries.

In this work, PFC-Sim is used to measure the cache performance of array references in a benchmark set of supercomputer applications, **RiCEPS**, gathered at Rice University. Data cache hit ratios varied between 44 and 99% (average 70%) for a 16K cache and 48 and 99% (average 91%) for a 256K cache. Some of the programs have very large working sets, yielding poor cache performance even with large caches. PFC-Sim also determines the cache performance for each memory reference in the original program. The hit ratios were clustered around either 100% or 0%.

By locating the references that miss, attempts to improve memory performance can focus on references where improvement is possible. The compiler can estimate the number of loop iterations which can execute without filling the cache, the **overflow iteration**. The overflow iteration combined with the dependence graph can be used to determine at each reference whether execution will result in hits or misses. Using this estimate of dynamic memory performance, two methods for improving performance are examined, **program transformation** and **software prefetching.**

Program transformation can be used to improve cache performance by reordering computation to move references to the same memory location closer together, thereby eliminating cache misses. Many programs have been transformed by hand to improve their memory performance. Using the overflow iteration, the compiler can often do this transformation automatically. Automatic blocking has advantages in correctness and portability. Standard blocking transformations cannot be used on many loop nests that contain transformation preventing data dependences. A new version of blocking transformations is defined, **wavefront blocking**, which allows any loop nest to be blocked, if the components of dependence vectors are bound between the references in the loop nest.

When the cache misses cannot be eliminated, software prefetching can overlap the miss delays with computation. Software prefetching uses a special instruction to preload values into the cache. A cache load resembles a register load in structure, but does not block computation. It only moves the address into cache where a later register load will be required to bring the value into the processor. Using a simple heuristic, the compiler can inform the cache (on average) over 100 cycles before the process will require a load of the value. This allows the cache misses to be serviced in parallel with the computation.

Contents

Abstract ii

Acknowledgments iv

List of Illustrations viii

List of Tables x

1 Introduction 1

1.1 Overview 1

1.1.1 Cache Memories 4

1.2 Dependence Theory 7

1.2.1 Data Dependence Analysis 7

1.2.2 Data Dependence Analysis of Arrays 8

1.2.3 Cache Dependence 10

1.2.4 LC(x) and D(x) 11

1.2.5 Uniformly Generated Dependences 12

1.3 Previous Work 13

1.3.1 Cache Performance Studies 13

1.3.2 Improving Memory Performance 14

2 Data Cache Performance 21

2.1 Introduction 21

2.2 Rice Compiler Evaluation Program Suite 25

2.3 PFC-Sim 27

2.3.1 PFC-Sim Preprocessor 28

2.3.2 Memory Simulation 30

2.3.3 Visualization Tools 34

2.3.4 Potential Uses of PFC-Sim 34

2.4 Experiments 36

2.4.1 Cache Size 36

2.4.2 Set Associativity 41

2.4.3 Write Policy 44
2.4.4 Prefetching 47
2.4.5 Hit Ratios of Individual References 53
2.4.6 Processor Performance 56
2.5 Summary 60

3 Static Model of Memory Performance **62**
3.1 Introduction 62
3.2 Overflow Iteration 63
3.3 Conclusions 74

4 Program Transformation **76**
4.1 Introduction 76
4.2 Loop Fusion 77
4.3 Loop Distribution 83
4.4 Loop Interchange 86
4.4.1 Example 90
4.5 Strip Mine and Unroll 90
4.6 Peel and Jam 92
4.6.1 Safety 93
4.6.2 Example 94
4.7 Loop Skew and Interchange 98
4.8 Blocking Transformations 100
4.8.1 Strip Mine and Interchange 100
4.8.2 Unroll and Jam 104
4.8.3 Wavefront Blocking 108
4.8.4 Choosing the Proper Blocking 111
4.9 Effectiveness 113
4.9.1 Transformable Programs 113
4.9.2 Non-transformable Programs 115
4.9.3 Semi-transformable Programs 117
4.9.4 Automatic Blocking (and Parallelization) 119
4.10 Conclusions 121

5 Software Prefetching **122**

5.1 Introduction 122

5.2 Cache Load Instruction 123

5.3 Insertion of Cache Load Instructions 124

5.4 Effectiveness 126

5.4.1 Hit Ratio 126

5.4.2 Time Between Prefetch and Load 130

5.5 Costs 133

5.5.1 Data Traffic 133

5.5.2 Run-time Overhead 135

5.6 Methods to Reduce Costs 137

5.6.1 Prediction of Misses 137

5.6.2 Register Allocation 139

5.7 Conclusions 141

6 Conclusions **142**

6.1 Overview 142

6.2 Future Work 144

6.2.1 Multiprocessors 144

6.2.2 Multiple Levels of Cache 145

Bibliography **146**

Illustrations

1.1 Types of Data Dependences . . . 7

2.1 PFC-Sim . . . 23
2.2 Matrix Multiply - Before and After PFC-Sim . . . 31
2.3 Effects of Cache Size - Hit Ratios . . . 37
2.4 Effects of Cache Size - Normalized Data Traffic . . . 40
2.5 Effects of Set Associativity - Hit Ratios . . . 42
2.6 Effects of Set Associativity - Normalized Data Traffic . . . 43
2.7 Effects of Various Write Policies - Hit Ratios . . . 45
2.8 Effects of Various Write Policies - Normalized Data Traffic . . . 46
2.9 Effects of Cache Line Lengths - Hit Ratios . . . 48
2.10 Effects of Cache Line Lengths - Normalized Data Traffic . . . 49
2.11 Effects of Hardware Prefetching - Hit Ratios . . . 51
2.12 Effects of Hardware Prefetching - Normalized Data Traffic . . . 52
2.13 Individual Hit Ratio Distribution . . . 54
2.14 Hit Ratios Weighted by Number of Misses . . . 55
2.15 Percentage of Execution Time Spent in Miss Delays . . . 58

3.1 Computing the Overflow Iteration, O . . . 67
3.2 Computing the Misses in a Code Segment . . . 68

4.1 Loop Fusion . . . 77
4.2 Profitability of Loop Fusion . . . 82
4.3 Loop Distribution . . . 83
4.4 Profitability of Loop Distribution . . . 85
4.5 Loop Interchange . . . 87
4.6 Profitability of Loop Interchange . . . 89
4.7 Strip Mining . . . 91

4.8 Unrolling 91
4.9 Peel and Jam 93
4.10 Application of Peel and Jam to WANAL1 96
4.11 Application of Peel and Jam to SIMPLE 97
4.12 Loop Skew 99
4.13 Strip Mine and Interchange 101
4.14 Strip Mine Width Computed during Execution 104
4.15 Unroll and Jam 105
4.16 Unroll and Jam Example 107
4.17 Unroll Factor Computed during Execution 108
4.18 Wavefront Blocking: Strip Mine, Skew and Interchange 109
4.19 Wavefront Blocking: Unroll, Peel and Jam 110
4.20 Profitability of Blocking 112
4.21 LINPACKD — Pivoting Preventing Transformation 116
4.22 FTRVMT — Linearization Preventing Transformation 118

5.1 Insert Prefetch Instructions 125
5.2 PFC-Sim with Prefetching — Matrix Multiplication 127
5.3 PFC-Sim with Prefetching - a portion of SHEAR 127
5.4 PFC-Sim with Prefetching — Index Array 128
5.5 Effect of Software Prefetching - Hit Ratios 129
5.6 Software Prefetching - Time Between Prefetch and Use (Absolute Totals) 131
5.7 Software Prefetching - Time Between Prefetch and Use (Normalized Totals) 132
5.8 Effect of Software Prefetching - Data Traffic 134
5.9 Insert Prefetch Instructions (Estimated Misses Only) 138

Tables

2.1 Data Traffic per Floating Point Operation 60

3.1 Matrix Multiply (100 x 100) — F, Δ, and O 72
3.2 Matrix Multiply — Computed Cache Performance of Dependences . . 72
3.3 Matrix Multiply — Computed and Actual Hit Ratios (32K cache) . . 73

5.1 Useful Prefetches 136
5.2 Execution Time with and without Software Prefetching 136
5.3 Useful Prefetches — After Using Overflow Iteration 138

Acknowledgments

No dissertation is written without a great deal of help and this one is no exception. First and formost, I would like to thank Ken Kennedy for it was his faith in my ability that allowed me to return to school, and his direction that guided this dissertation. David Callahan was the source of many valuable discussions in all areas of compiler optimization and helped immensely throughout the writing of this dissertation. I also want to thank the friends that maintained my sanity throughout graduate school, David Harper, Alan Carle, Jerry Fowler and the entire PFC group. Burton Smith and Tera Computer Company provided the time that was required to finish.

Thank you, Beth and Suzie, for being much better kids than I could ever have been. And finally, thank you Carrie, without whom I could never have gone back to school and maintained the lifestyle to which I had grown accustomed. Thanks also for providing motivation when it was lacking.

Chapter 1

Introduction

1.1 Overview

A major limitation of supercomputers today is the inability to get data to the processing units fast enough to keep the units busy. Vector and parallel processors replicate arithmetic units, dramatically increasing the number of operations that can be performed during any time period. Vector and shared memory multiprocessors, however, still have (at some point in the memory hierarchy) a single memory to support the increase in computational power. This has placed a substantial burden on computer architects to increase bandwidth between the memory and the processing units. Doubling the number of processing units is much easier than doubling the amount of data that main memory can supply during a given time period. As less bandwidth per processor is available, it becomes increasingly important to understand how the bandwidth is used and to define mechanisms to improve its effectiveness.

In von Neumann and data-flow architectures, data must be moved to each processing element before it can be manipulated. Computation may be abstractly modeled as a flow of data between the processor and the memory. In an abstract model, the processor makes requests to a black box, known as the memory, which returns values at some rate (the bandwidth) after a delay (average memory latency). When modeling computation, the boundary between memory and the processor is flexible. Several natural locations to measure the flow of data are between the registers and cache, between cache and main memory, and between main memory and backing store. If the processor includes the registers but not the cache, the memory requests are the loads and stores in the program. When the processor is defined to include the main memory but not backing store, the memory requests are the virtual page misses generated.

Execution time in this model is the time required to do the computation, T_c, plus the time waiting for the data to arrive, T_m. The time spent waiting for data to arrive is the number of memory requests, R, times the average memory latency. If this is broken into components for a three-level memory hierarchy, total execution time is $T_c + T_{m_{cache}} + T_{m_{main}} + T_{m_{backing}}$.

Most compiler optimization research has studied methods to reduce T_c, since it typically encompasses the majority of the total execution time. Register allocation has been the major research area addressing the time spent waiting at any memory level. Register allocation reduces $T_{m_{cache}}$ by reducing the number of requests to cache. Virtual memory systems attempt to reduce $T_{m_{backing}}$ by minimizing the number of virtual page misses that occur across all active programs. Little research has been done on how a compiler can reduce the amount of time spent waiting on requests to either main memory or the backing store [AS79],[Tha81].

The amount of time spent waiting on memory can be reduced in two ways. Either the number of requests can be reduced or the average latency can be reduced. When possible, reducing the number of requests may also reduce the average latency. Memory latency is a function of the rate of requests, $R/(T_c + T_m)$. When the number of requests exceeds the bandwidth of the memory, average latency will increase until the average rate of requests can be satisfied by the bandwidth. Saturation of the memory bandwidth increases T_m until it is a significant fraction of the total execution time. Reducing execution time for a program that saturates the memory bandwidth will require either more memory bandwidth or fewer memory requests.

Previous work in reducing the traffic between a cache and main memory has mainly involved studies of different cache architectures. Natural program locality is high enough to ensure that caches are effective and the cache-main memory interface rarely if ever saturates on non-vector uni-processors. As vector and parallel processors become more common, this will no longer be true. Their multiple processing elements can manipulate much more data in a given time period, greatly increasing the probability that the bandwidth between the cache (or caches) and the main memory becomes saturated. (Note that in the abstract model, multiple computational elements can be represented by lowering the computation time, T_c, allowing a very simple method of simulating the effect of parallel or vector processors.) A second effect of parallel processors comes from parallel programming. For many problems, extracting parallelism decreases locality of reference of each processor. Thus, the total number of references per time unit is rising and processor reference locality is falling, greatly increasing the strain on memory hierarchies.

This dissertation studies the amount of data that flows across the cache-main memory boundary for supercomputer applications. The focus of this work is computationally intensive programs. These programs are the most likely to access large amounts of data, stressing the memory system. Computationally intensive programs are also likely to use highly optimizing compilers to produce the fastest executables possible. Scientific programs that are typically run on supercomputers require a large number of floating point operations. Smaller programs on medium or small computers may benefit from the techniques developed in this thesis, but the focus of this work is on the programs that require Cray-class computers. Methods to reduce the amount of data traffic by increasing the average number of references to each item while it resides in the cache are explored. Increasing the average number of references to each cache item reduces the number of memory requests.

When a memory location is requested, most architectures wait for the value to return before continuing. Certain architectures and operating systems have tried to hide memory latency from the processor. Almost every virtual memory system transfers control to another process during a page miss. This allows the processor to do productive work on other jobs while the backing store produces the page. Some RISC architectures (e.g. IBM's RT) attempt to hide the delay between the registers and the memory by allowing several instructions to execute between a register load and the arrival of data in the register. The Denelcor HEP and HORIZON [KS88] architectures hide main memory latency by switching instruction streams every cycle. If enough instruction streams are available, the processor never delays waiting for memory to return a value. Since the HORIZON is, for the time being, a paper architecture, and few, if any, HEPs are still running, supercomputers today generally wait for the memory on every main memory access.

People seem to believe that the amount of processing lost during these delays, $T_{m_{main}}$, does not significantly slow uni-processors. Otherwise, more work on methods to improve cache performance would exist today. Architects of shared-memory MIMD machines have found memory latency problems to be the principal difficulty in effectively utilizing their machines [KS88]. Cedar[GKLS83] and RP3[PBG+85] compilers go to some effort to localize a processor's memory accesses (both spatially and temporally) to reduce the memory latencies. If memory latency has such an effect on multiprocessors, the assumption that the time spent waiting for memory on uni-processors is insignificant may not be correct.

This dissertation examines that assumption and explores methods to help all programs reduce average memory latency by separating cache and register loads. Separating cache and register loads allows memory latency to be overlapped with execution, reducing the average amount of time spent waiting for each memory request. The next section gives some background on cache memories and dependence analysis. This is followed by a discussion of the previous work in this area.

1.1.1 Cache Memories

Cache memories are high-speed buffers used to hold portions of the main memory believed to be in use. Values in a cache can be accessed in much less time than would be required to access main memory. When a significant fraction of the references are found in the cache, the total memory delay during execution is reduced. The effectiveness of cache has been explained by the "property of locality"[Den72]. Over short periods of time, references are accessed non-uniformly over its address space. References to be used in the near future are likely to be values used in the recent past or those values that are close to the present values.

Cache design has several performance criteria. A cache should maximize the probability that it contains a requested value (the hit ratio). The cache should also be as simple as possible to minimize cache hit access time. Their are other criteria, but these are normally the two most important.

The design of a cache involves a number of decisions. This work examines the benefits (in terms of increasing the hit ratio) and costs (in terms of the total data traffic required) of a number of potential cache structures. The cache parameters discussed in this work are cache size, cache associativity, cache replacement policy, main memory update policy, line size and hardware prefetching.

Cache Size

The first parameter of a cache normally given is its size. The more values that are kept in cache, the higher the probability that the requested value is present. Caches cannot be expanded without limit because of cost, physical size constraints and limited access time. A design may attempt to use the smallest cache that provides a certain level of performance.

Cache Associativity

The user cannot directly address the cache, but rather the cache serves as a buffer for addresses in the main memory. The mapping of addresses into the cache is determined by the associativity of the cache. A fully associative cache can map an address in main memory to any cache block. This requires that the cache maintain a very large associative memory. Each time a block is accessed, the address must be compared against the associative memory to determine if the address is present in the cache. At the other end of the spectrum are direct mapped caches, which map each address to a unique cache block. During a lookup, only one location must be checked to determine if the value is present in the cache. Between the two extremes are set associative caches. A set associative cache can map an address into a limited number of locations. The number of potential cache locations for an address is determined by the degree of set associativity, 2 for 2-way, 4 for 4-way, etc. Most computer caches designed today are set associative.

A tradeoff exists between the number of sets and set size. A direct mapped cache will have four times as many sets that are one-quarter the size of sets in a 4-way associative cache. Poor performance occurs when programs cyclically reference more elements than can be kept in a set. For a direct mapped cache, two elements that are accessed alternately and map to the same slot produce no hits. For larger set sizes, the number of items in the rotation can be larger before performance degrades.

Cache Replacement Policy

When a value is requested from the cache which is not present and the cache is full, some block in the cache must be removed so that the requested block can be brought into the cache. The goal of a replacement policy is to remove values that will not be used in the future.

Main Memory Update Policy

There are two basic methods that memory buffers use to update the backing memory. The buffer either waits as long as possible before writing a value back to the memory hoping to reduce the number of writes, or it passes modified values through immediately, reducing coherency problems and the maximum time to satisfy a miss. Delaying the write to the backing memory is called a *write-back* (or *copy-back*) write policy. Immediately passing the value through to the next memory is a *write-through* policy. The advantage of the write-back method is a much lower volume of memory traffic. For multiprocessors, where the volume of memory traffic will be crucial, the write-through memories have the advantage of maintaining consistency between processors. Write-through memories are further broken into two groups depending on whether it is believed that a value that is written will be used again before it is flushed from the memory. The *write-through no-load on write-miss* policy speculates that it will not be used again so it does not save the block in the cache after storing the value in main memory. While the standard write-through policy loads the block into the cache on a write miss.

Line Size

To improve the probability that a value is in the cache when requested, there are several mechanisms to prefetch memory locations. Prefetching (fetching before the actual request) allows the cache to guess what values will be needed in the near future and bring them into the cache. The cache line is the fixed-sized unit of information transfer between the main memory and the cache. The most common prefetching mechanism is the multi-word cache line. The presence of adjoining items increases the number of hits whenever locality of reference exists in the accesses.

Hardware Prefetching

A second form of prefetching that can be implemented is hardware prefetching. In a hardware prefetching scheme, whenever the first word of a cache line is accessed, the next sequential cache line is fetched from the main memory. For example, if address 100400 were accessed (either loaded from or stored into), then the hardware would request that the block starting at 100404 be brought from main memory to the cache (assuming one word cache lines). Hardware prefetching is very effective when FORTRAN arrays are accessed along columns.

1.2 Dependence Theory

To correctly determine if an address is in the cache when it is accessed, it is necessary to know the last time that it was used. In conjunction with the development of restructuring compilers for vectorization and parallelization, a method for describing the constraints on execution order, called data dependence analysis, has been developed. The constraints identify the potential last accesses of any memory location in the program. A conservative set of **dependences** is built during the analysis. There are four types of dependences: **true**, **anti**, **output** and **input** [Kuc78][KKP+81].

1.2.1 Data Dependence Analysis

The four types of data dependences are shown in Figure 1.1. To have a true dependence between two statements, a value defined in statement S_1 must be used in statement S_2. A true dependence can be viewed as a requirement that a producing statement execute before the consuming statement. A second type of constraint occurs when a statement S_1 uses a value, and a later statement S_2 redefines that value. Statement S_1 must be executed first to ensure that the value produced by S_2 is not used in S_1. This is an anti-dependence. A third ordering constraint occurs when statement S_1 defines a value and a later statement S_2 also defines the value. For the value to be correct upon completion of both statements, S_2 must follow S_1. Dependences of this type are called output dependences. An input dependence is not a constraint on execution order but does indicate a reuse of a memory location. Since both S_1 and S_2 only use the same value, they can execute in any order and produce correct results.

Data dependences do not specify all of the constraints on statement ordering. Control flow through the program affects which statements are actually executed.

S_1	`A = ...`	`... = A`	`A = ...`	`... = A`
S_2	`... = A`	`A = ...`	`A = ...`	`... = A`
	true	anti	output	input

Figure 1.1 Types of Data Dependences

For instance, in the following example, statement S_2 depends on which branch is taken in statement S_1.

```
S1            IF (I.EQ.0) GOTO 10
S2            ANS = ANS / I
S3     10     CONTINUE
```

S_2 is said to be **control dependent**[KKP+81] on S_1. To avoid dealing with control dependences as a separate form of dependence, a process called **IF-Conversion** [AKPW83] has been developed to convert all control dependences into data dependences. IF-Conversion removes all control flow by replacing forward branches with guards on the affected statements. Backward branches are replaced with guards and explicit WHILE loops. IF-Conversion allows us to focus only on data dependences.

Data dependences in the presence of DO loops can be characterized as either **loop carried** or **loop independent**[All83]. A loop carried dependence is simply a dependence that exists between separate iterations of a loop. Each loop carried dependence is associated with a particular loop in the program. A loop independent dependence is a dependence within a single iteration of the loop. Any dependence that is not surrounded by a loop (i.e., exists if all loops are removed) is also loop independent.

1.2.2 Data Dependence Analysis of Arrays

When array references are treated as single scalar variables, very imprecise information is generated. If different elements of an array are being accessed, memory is not being reused and no dependence exists. By determining when the iteration space of array references does not overlap, a much more accurate dependence graph can be built. To make data dependence information on arrays more precise, several tests have been developed to determine if a dependence exists between two array references. For a majority of array references, these tests precisely determine whether dependences exists. For example, in the following loop,

```
(S1)        DO I = 1,30
(S2)           A(I) = A(I)*B(I)
(S3)           C(I) = A(I+1)
(S4)        ENDDO
```

if arrays A, B and C were treated as scalar variables, a loop independent true dependence would exist from S_2 to S_3 on A. Additionally, four loop carried dependences would be identified: (1) a true dependence from S_2 to S_2 on A, (2) an anti-dependence from S_3 to S_2 on A, (3) an output dependence from S_2 to S_2 on A and (4) an output dependence from S_3 to S_3 on C. The multiple cycles in the dependence graph would mark this loop as unparallelizable.

However, if we consider the arrays on an element by element basis, a more precise picture emerges. Since no memory location is modified more than once in the loop, no output dependences actually exist. S_2 and S_3 use the value of A present before the loop starts execution, eliminating both true dependences. The only dependence that remains is the loop carried anti-dependence from S_3 to the next iteration's S_2 on the variable A. With only a single dependence to guarantee, a restructuring compiler can attempt to produce a parallel or vector version of this loop.

A statement in a loop actually represents a large collection of executable statements, one executable statement per iteration. A **direction vector** [Wol82] relates references to the same memory location according to which iterations of the enclosing loops cause the overlap. The direction vector is formed from the **distance vector**, which is the pairwise difference of the overlap of the iteration space that causes the two references to access the same data. The distance vector represents the number of iterations of a loop that a dependence must cross. This will be an important value in determining if a requested value is present in the cache.

Much research has been done to determine when data dependences exist between statements with arrays [Ban76] [Ban79] [AK84] [Kuh80] [Wol82] [All83] [TIF86]. The problem can be stated as, "Given two array references and information about which loops contain them, do they ever refer to the same storage location?"

A data dependence exists between two array references $A(f(\vec{i}_1))$ and $A(g(\vec{i}_2))$ if, and only if, for some vector $\vec{i}_1$ and $\vec{i}_2$

$$f(\vec{i}_1) - g(\vec{i}_2) = 0$$

Bernstein proved that this problem is undecidable for arbitrary functions f and g [Ber66]. Knuth showed that for real programs, the expressions are generally simple [Knu71]. Therefore, placing restrictions on f and g is reasonable. In order to solve the problem, functions have been restricted to affine functions in the loop control variables, i.e.,

$$f(\vec{i}) = a_0 + a_1 \cdot i^1 + \cdots + a_k \cdot i^k$$

where a_j is constant.

Making this restriction, two tests have been developed to determine data dependences, the **gcd** test [Coh73][Ban76] and **Banerjee's Inequality**[Ban76][Wol82] [AK84]. The gcd test determines whether any integer solutions exist for $f - g = 0$. Banerjee's Inequality determines whether there are any solutions to the same equation (integer or real) within the area specified by the loop bounds. Both of the tests are conservative. If either determines that a dependence does not exist, there is no dependence. There are cases where both tests determine that a dependence could exist when there is actually no dependence.

Before applying the gcd or Banerjee's test, the optimizing compiler at Rice University, PFC, uses a **separability** test. The separability test is quick and accurate, but works only on a restricted set of subscript expressions[1]. If a pair of subscripts are of the form $ai + b_1$ and $ai + b_2$, then a dependence exists if either (1) $a = 0$ and $b_1 = b_2$ or (2) $\frac{b_1 - b_2}{a}$ is an integer and is less than or equal to the upper bound of a normalized loop. Separability is much faster than the other tests and works on at least 50% of the expression pairs in a group supercomputer applications. For some of the programs in the group, over 95% of the expression pairs are separable [Ros88].

Kuhn and Triolet [Kuh80][TIF86] describe the region of memory references by a set of linear inequalities. If the sets of linear inequalities overlap, then a dependence exists. This method is more accurate, but potentially much slower.

1.2.3 Cache Dependence

When dealing with cache, there is no distinction between types of references. A load has the same effect as a store — both cause the item touched to become the "most recently used" (ignoring, for the time being write-through no-load on miss). Thus there need be only one type of cache dependence — from reference to reference. An important effect of this observation is that variable uses as well as variable assignment

[1]Randy Allen developed PFC's separability test after studying Lamport's paper on Parallel Execution of Do-Loops [Lam74]

become "kills", where a kill terminates all outstanding dependences. Hence, there may be many fewer cache dependences than data dependences.

To see the difference between the two graphs, observe that the following code fragment has six data dependences and only three cache dependences.

```
                DO I = 1,30
($S_1$)            A = ...
($S_2$)            ...  = A
($S_3$)            ...  = A
                ENDDO
```

The six data dependences in this loop are from S_1 to itself and to S_2 and to S_3, from S_2 to S_3 and S_1, and from S_3 to S_1. At times during the execution of a transforming compiler, the transitive closure of the data dependence graph is required. Many transforming compilers will actually produce all nine dependences from this code fragment. Since transitive cache dependences are not used anywhere, the aggressive use of kill sites is encouraged for cache dependence analysis. Aggressive use of kill sites reduces the cache dependences for this code fragment to three: (1) a loop independent true edge between S_1 and S_2, (2) a loop independent input edge between S_2 and S_3, and (3) a loop carried anti edge between S_3 and S_1. Eliminating half of the dependences can be significant when algorithm running times depend directly on the number of edges in the graph.

1.2.4 LC(x) and D(x)

During computation of the overflow iteration, it will be convenient to blur the distinction between loop carried and loop independent dependences. To avoid having to handle the dependences separately, two functions will be used: *Loop Carried By*, $LC(e)$ and *Distance*, $D(e)$. The values of both of these functions are obvious for loop carried dependences. $LC(e)$ is the loop level that carries the distance, and $D(e)$ is the value of the distance vector at that level. $LC(e)$ for a loop independent dependence edge is the minimum nesting level of any statement between the source and the sink of the edge, including the endpoints. The distance $D(e)$ of a loop independent dependence is always 0.

When trying to determine the amount of memory used by a program segment, a dependence edge that does not have both of its endpoints within the segment should not be considered. For loop carried dependences, the endpoints are contained in the segment if the edge is carried by a loop in the segment. For a loop independent edge, the edge is considered when the endpoints are both within the segment. By defining $LC(x)$ to be the nesting level of the smallest loop that includes both endpoints of a loop independent dependence, $LC(x)$ indicates which loops are affected by an edge x.

The distance $D(x)$ indicates how many iterations of a loop must occur before the dependence can be satisfied. It is correct to set $D(x)$ to 0 for loop independent edges, since the loop control structure will not execute between the source and the sink of the dependence.

1.2.5 Uniformly Generated Dependences

The dependence graph is made up of edges indicating where dependences *may* exist. When we are trying to compute how much memory is used by a loop, the dependences that prevent new data from being used are those that actually exist. *Must* dependences would provide better information about which references require additional data to be used in a program segment. Although transformations need to guarantee that all possible dependences are maintained, when considering memory performance, inaccuracies cause only sub-optimal performance, not incorrect performance. If the *may* edges were included when the overflow iteration was computed, each loop would be predicted to use less memory during execution. If a program was tuned using a low estimate of the amount of data required, the actual usage would cause misses. If a high estimate is used, the actual usage would fit into a smaller cache, just wasting some of the available space. Accurate estimates will allow better tuning, minimizing the misses for a given memory size.

Gannon, Jalby, and Gallivan [GJG87] defined a uniformly generated dependence as a dependence between two references for which a distance vector exists and each element is known (not symbolic).

If two references have a known distance vector between them, then the compiler has determined that the two references will touch the same memory location. A uniformly generated dependence separates two references that *must* access the same location during execution. Uniformly generated dependences fit neatly into the idea of *must* dependences. Many other dependences, such as those due to different symbolic subscripts or index arrays, will rarely or never access the same location. No other well-defined class of dependences definitely uses the same memory location at both ends every time the ends are executed.

May dependences are ignored due to the manner in which the overflow iterations are calculated. For each reference in a loop, the dependence graph is used to determine whether the reference has been used previously in the loop. If a *must* dependence exists (and the execution path between the two statements occurs), then the sink will not use a "new" storage location. The assumption is that *may* dependences will, in actuality, rarely use the same memory locations. If the statements rarely use the same location, then the sink of the dependence will use a "new" storage location. Because the overflow iteration should overestimate the amount of memory used, rather than underestimate, only *must* dependences are considered. For the purposes of computing the overflow iteration of a loop, any dependence that is not uniformly generated is ignored.

1.3 Previous Work

The previous work fits into two basic categories: cache performance studies and attempts to improve memory performance. Previous cache performance studies have used short traces; a method to trace the cache performance of complete programs is presented in this work. The basic difference between the methods to improve memory performance described in this dissertation and previous work, is our attempt to make the best use of a fixed size memory rather than attempting to minimize the set of active locations.

1.3.1 Cache Performance Studies

The best known cache performance study is by Alan Smith [Smi82]. He measured the performance of a combined instruction and data cache on 19 program address traces (3 PDP-11 and 16 IBM 360/370). The traces were split into groups of four. A cache simulation run switched between traces in a group every 10,000 references to model multiprogramming, until each trace had supplied 250,000 references. The traces varied from a text editor written in C to the COBOL compiler and an APL plotting program. Seven of the programs were written in various dialects of FORTRAN. This set of programs was meant to model a general purpose educational workload. The programs were not CPU intensive and did not have large data requirements. A 32K cache produced a 98+% hit ratio for every program measured.

Smith made a number of statements about cache performance. He stated that a hardware prefetching mechanism was effective for all tested programs. Set associativity above 8 produced very little benefit. Optimal line size was between 64 and 256 bytes. LRU was on average 12% more effective than FIFO. There was no clear performance difference between the write-back and write-through policies [Smi82].

In 1985, Smith conducted a detailed examination of hardware prefetching[Smi85b]. In this study, 49 traces from 6 architectures and 7 languages were used. Again, the traces were short (two 500K and the rest 250K or shorter). Only three programs accessed over 100,000 bytes of virtual storage during the traces. It was noticed that the various machines and compilers had different instruction fetch and data write percentages. Smith's report indicated the need to examine a representative workload to determine the effectiveness of a cache for a particular machine.

Lee, Yew and Lawrie [LYL87b] looked at the performance considerations of caches on multiprocessors. The cache performance of 23 scientific library subroutine calls was examined. A compile time cache coherency solution prevented multiple copies of read/write data by marking them as non-cacheable references. The longest trace was about 34 million references, and the average trace was between 5 and 6 million references. They found that the optimal cache line was always less than eight words and in many cases only one or two words.

Chapter 2 discusses a cache performance study using typical supercomputer application programs as the workload. Scientific programs that run on large computers are a different type of workload than a general purpose computing workload. Before attempting to transform programs to improve performance, a study to determine the potential gains is performed. During the performance study, the differences in memory performance due to workload variation are discussed.

1.3.2 Improving Memory Performance

A number of researchers have improved performance at various levels of the memory hierarchy. The work has ranged from trying to lower the number of virtual page faults to eliminating extraneous uses of vector registers. The following sections are brief descriptions of the most relevant work, together with a brief discussion of their influence on and differences with the work in later chapters.

Virtual Memory Locality

Abu-Sufah [AS79] examined the problem of reducing the number of virtual memory pages used by a program. In contrast with earlier work surveyed by Denning [Den70] and Kuck [KL70], he used data dependences as the basis of his solution. Abu-Sufah produced a general mechanism that substantially reduces virtual memory paging. His studies suggest that a transformed program can get the same performance as the original program with one-fifth to one-sixth the number of virtual memory pages.

Abu-Sufah developed a general method for measuring the space-time cost of various programs using a virtual memory page as the unit of allocation. He then examined the effects on the space-time cost of programs, of three transformations individually, and in combination. The three transformations are loop distribution to separate the program into smaller pieces, loop fusion to recombine pieces that are heavily related, and strip mining to access arrays in single page size sections.

Program transformations were very successful for Abu-Sufah in reducing each program's space-time cost. Although the criterion, space-time cost, is not useful for cache memories, the effectiveness of the transformations in tuning programs to the criterion is encouraging. The success demonstrates that transformations can be used to change global memory performance in a predictable manner.

Cache Packing

Kalid Thabit [Tha81] examined several software methods to reduce the effective main memory access time when a cache is present. Two models for compiler control of the cache, the **prompting** and the **explicit** models, were presented. In addition to presenting these models, Thabit showed that memory allocation to make maximum use of long cache lines is equivalent to bin packing and is an NP-complete problem. Also presented is a simple data dependence test to determine whether a blocking transformation (strip mining) suggested by Abu-Sufah is safe.

The prompting model presents the compiler with two cache commands, `Prompt` and `Release`. `Prompt` allocates a memory block in the cache, possibly bringing data in from main memory, and `Release` deallocates the data block. The allocation mechanism is not speculative. If a block is requested, then it will be used. Since Prompts are not speculative, they cannot be moved across control dependences that may affect execution. Allowing speculative prefetching, we can move many of the prefetch commands much further from the actual uses, greatly increasing the likelihood that the prefetch completes before the value is actually required. The prompting model was presented by Thabit with no data on how effective it would be in practice.

The explicit model gives the compiler complete control of cache block allocation, cache block deallocation, and transfer of data to and from main memory. This model reduces the cache management problem to the register allocation problem (on a somewhat larger scale). There are several problems with this model. The most common solution to the register allocation problem in the literature is graph coloring [CAC+81],[Cha82]. The graph coloring solution is intractable and does not seem to be a promising method of cache management. When thousands of cache blocks exist, the coloring could take an unacceptably long time to solve. A second problem is that the control instructions would need to be inserted into the program rather than being handled by the hardware.

Thabit's model for compiler control of cache showed that some simple optimal cache allocation problems are NP-complete. Rather than look for solutions to these problems, compiler writers need to determine what cache behavior is common in actual programs and find solutions that are fast and accurate for those cases.

Cache Locality

Gannon, Jalby and Gallivan have done the only known prior work on maximizing cache locality [GJG87]. They computed the number of active locations in an array. Transformations were then used to reduce this number and to predict a lower bound on the number of cache hits.

The general method is to define a family of "reference" windows for each variable. A reference window defines the current set of elements that must be in the cache (i.e., those that will be used again). A reference window is defined for each dependence and specifies the set of elements used at the source before some time t that are also used at the sink after time t.

To define these windows accurately, several common subcases of data dependences are defined. The most important is when a dependence distance vector can be determined exactly and consists of constants. This is called a **uniformly generated** dependence. It is simple to compute the window size of a uniformly generated dependence. In general, the window will be one larger than the number of iterations necessary to carry the dependence. For example, a loop independent dependence has a distance vector of all zeros and a window size of one. A uniformly generated loop carried dependence with a distance of three has a window size of four elements.

The hit ratio can be calculated by determining a spanning dependence for each variable and then counting the number of references inside the spanning window and the number of different locations accessed by the spanning dependence. The hit ratio is then the number of references minus the number of different locations divided by the number of references. This is a program constant and cannot be changed with safe transformations. Gannon et.al. attempted to minimize the maximum sum of active windows in the program.

A major difference between Gannon's methods and the methods proposed in this dissertation is our emphasis on the loop that does not fit into the cache. Cache management is not an absolute; there is a trade-off between compile time and execution time. Under the approach in this dissertation, it is possible to identify when the program fits into available cache such that there is no need to further improve the program.

This dissertation uses Gannon's definition of uniformly generated dependences to describe dependences that must reuse a memory location. Dependence graphs include edges that either can not be disproved or are known to occur only occasionally. When determining whether a memory location resides in cache when referenced, the only edges of interest are those that actually occur. Uniformly generated dependences capture this idea very cleanly.

Vector Register Allocation

Allen and Kennedy [AK88] studied methods to section vector loops when the number of loop iterations exceeds the vector pipe length. Assuming a vector pipe of 64 elements (CRAY-1, CRAY-XMP) and a 100 element vector operation, the naive solution of executing the first 64 as one operation and the last 36 as a second operation may produce incorrect results. The simple solution of using temporary storage to guarantee that the correct values are always used can use a tremendous amount of storage.

Allen and Kennedy use the data dependence graph to determine when simple sectioning produces potentially incorrect results. A number of program transformations such as, loop reversal, input prefetching, loop splitting, loop interchange, and loop fusion can be used to produce correct results without using temporary storage. The circumstances under which each transformation is safe and effective were described.

Register allocation optimizes the performance of a small explicitly controlled set of locations. The compiler knows the contents of each register at all times. Maintaining this level of information about cache and virtual memories would be very expensive due to the large number of cache locations and the nonlinear complexity of register allocation methods. To improve cache performance, methods that do not track every cache location are needed. Fortunately, cache misses are not catastrophic errors. If a register does not have the expected contents, then the program is incorrect. If the cache does not have the expected value, then the result is a cache miss. Thus, a probabilistic solution can be used.

Allen and Kennedy have shown the effectiveness of program transformations for improving memory access patterns. Vector register allocation deals with specifically allocated memory, but the specific transformations to shorten dependences between references that use the same memory locations could be effective for cache management if applied properly.

Cache Prefetching

Lee, Yew and Lawrie have studied data prefetching for shared-memory multiprocessors [LYL87a], [LYL87b]. Working on the Cedar project with software-enforced cache coherency, they found that data prefetching was effective in addressing the memory access bottleneck. Their view of data prefetching initiates operand fetches several instructions beyond the current processor instruction.

Data prefetching does not address the problem of saturating the memory bandwidth, but it does attempt to overlap the memory latency with execution. They report that prefetching obviates the performance advantage of long cache lines. Since multiprocessor interconnection strategies may have limited-width data paths, using shorter cache lines can reduce the average memory latency. They hypothesized that with data prefetching, shorter cache lines will perform better for multiprocessors.

Lee et. al.[LYL87a] showed that low level prefetching can be effective. In general, prefetches into registers can not be separated from the use of the register by more than a few instructions, since a register must be allocated during the prefetching. If prefetching is separated from register loads, the prefetch can be moved further away from the actual load, allowing a larger overlap of memory latency and computation time. The success of overlapping data fetches at the low level indicates that more aggressive prefetching may be able to almost completely overlap memory latency delays with execution.

Blocking

Two recent papers have reported on computer programs whose performance displayed noticeable sensitivity to problem size when running on vector or parallel processors. Vector performance of an Alliant FX/1 fell by as much as 60% when the vector lengths were greater than the length that could be maintained in the cache [ASM86]. IBM 3090 performance fell by 30% when the arrays of a matrix multiplication were larger than the cache size [LS88]. In both reports, it was noted that performance did not degrade when accesses were blocked into groups smaller than the cache size.

Blocking data accesses to improve the performance of a processor (particularly parallel processors) is a well-known technique. Gallivan, Jalby, Meier and Sameh [GJMS88] discussed the performance of blocked BLAS3 linear algebra routines on an Alliant and found that blocked versions generated twice the MFLOPs for some array sizes. For tridiagonal linear equations, Berry and Sameh [BS88] improved the performance by blocked LU decomposition over the standard LINPACK version[DMBS79] by as much as 800%. On a Cray-2, unrolling the inner loop improved performance by a factor of three, and a blocked assembly program increased performance by a factor of six for large arrays [Cal86].

In these studies, programmers were either blocking the programs by hand, [LS88], or providing library routines that have been blocked by hand (eg. BLAS3 [GJMS88]). Although both of these methods are effective, it is desirable to automate the blocking of data. Any machine dependent function that the compiler can perform simplifies a programmer's job and makes the programs easier to port to new machines. When the compiler performs the blocking, it does not obscure the programmer's view of the program, and the blocking is ensured to be performed correctly. This will eliminate some errors and make others easier to locate. When the programmer is blocking array accesses by hand, some estimate of the potential benefit is weighed against the demands on his time. If the gains are small or difficult to determine, blocking is not done. By spending some compilation time to block memory accesses, any program that benefits from blocking is blocked.

When code is blocked by hand, the program should be retuned every time the memory hierarchy changes. This includes moving between manufacturers, moving between computer models from one company and, in some instances, moving between different versions of the same computer. Automatic blocking can tune code to a particular memory hierarchy by simply recompiling.

The first step towards automatic methods of increasing memory performance is understanding how typical memories perform on real programs. The performance of caches on computationally intensive programs has not been thoroughly studied. Before it is possible to start optimizing programs to run on parallel processors with multiple caches or multiple levels of cache, a firm understanding of how a single cache and processor interact is required. This dissertation attempts to understand how computationally intensive programs run on sequential caches and investigates how to improve their performance. Both software and hardware mechanisms for improving performance are considered.

Chapter 2 describes a tool to measure very long cache traces and uses it to measure the performance of a group of supercomputer application programs. The next chapter presents a static model of the dynamic memory behavior. Program transformations to improve a program's memory performance are discussed in Chapter 4. For those cases where transformations are ineffective at eliminating cache misses, Chapter 5 examines a simple method of overlapping cache misses with other computation.

Chapter 2

Data Cache Performance

2.1 Introduction

This chapter describes an experimental study of a wide variety of programs and caches to determine how various memory structures perform for computationally intensive programs. Hardware designers can use this study to help make design decisions for super and super-mini computers. Chapters 4 and 5 will use the tools built for cache simulation and observations of programs with poor cache performance to examine methods for improving the cache performance during compilation.

Recent studies at the University of Illinois and IBM have shown that memory performance can have a significant impact on the overall performance of a computer. Abu-Sufah and Maloney measured the vector performance of an Alliant FX/8 on increasing vector lengths and found that when the vectors exceeded the cache size, the performance declined by up to 60% [ASM86]. In similar work, Liu and Strother found that a matrix multiply problem that did not fit in the cache ran 30% slower on an IBM 3090 vector processor than when the problem did fit into the cache [LS88]. Both reports recognized the problem as a memory performance problem and suggested methods to improve overall performance by improving the cache hit ratios of the programs.

In both studies, the test programs were quite simple, namely, a program with very long vector operations, and a single matrix multiplication problem. While these are certainly subroutines that will be called by programs running on supercomputers, they will not normally be programs by themselves. However, actual production codes tend to be computationally intensive and thousands of lines long.

In the past, cache performance studies have not examined the class of computationally intensive programs. There are two reasons: (1) there is no commonly acknowledged benchmark, and (2) tracing simulations of long programs requires too much storage. Studies such as those done by Alan Smith [Smi82],[Smi85a] use trace lengths from 10,000 to 333,000 entries. These traces correspond to fractions of a second of execution time for powerful machines. However, traces of this length are used for several reasons. First, the storage required to keep a trace that would correspond to minutes or hours of execution is not available. Second, most cache studies have been directed toward time-sharing systems that will swap the program out to disk after a small time-slice, effectively flushing the cache about every 10,000 accesses. Other studies have used whole programs, but the relationship between these programs and actual supercomputer applications is unknown. Chi and Dietz, [CD89] are looking at the performance of a class of caches. They use full programs such as bubble sort, puzzle, and tower of Hanoi for their study, but the performance of these programs does not necessarily correspond to the performance of supercomputer applications.

The cache studies have focused on cache performance and have paid little attention to the relationship between misses in the cache and the program that is executing. Before the compiler can attempt to improve memory performance, an understanding of the behavior of each reference in the source program is required. To determine the performance of individual references during execution of long programs, we developed a new simulation tool called PFC-Sim.

PFC-Sim is a program-event-driven tracing facility which can be modified to perform a variety of dynamic statistic-gathering tasks. The version used in this dissertation gathers information about memory performance in terms of hits, misses and traffic. Another version counts floating point operations and dynamic vectorization percentage. All versions of PFC-Sim have the same basic structure (see Figure 2.1) including a preprocessor that locates interesting events in the source program and inserts code that invokes a run-time simulator, run-time routines to simulate some functions or just record the events, and visualization tools to make the dynamic statistics easier to understand. For memory simulation, every data reference in the program is preceded by a call to the cache simulator. To simulate execution time, every basic block has a statement that increments the simulated clock by the execution time required for a single execution of the block.

Very long programs can be simulated by PFC-Sim since trace entries produced by a program are discarded as soon as the simulator processes them. In this manner, simulations of any length can be run without concern for disk space. The translation process that marks events in the source gives each event a unique number. By passing

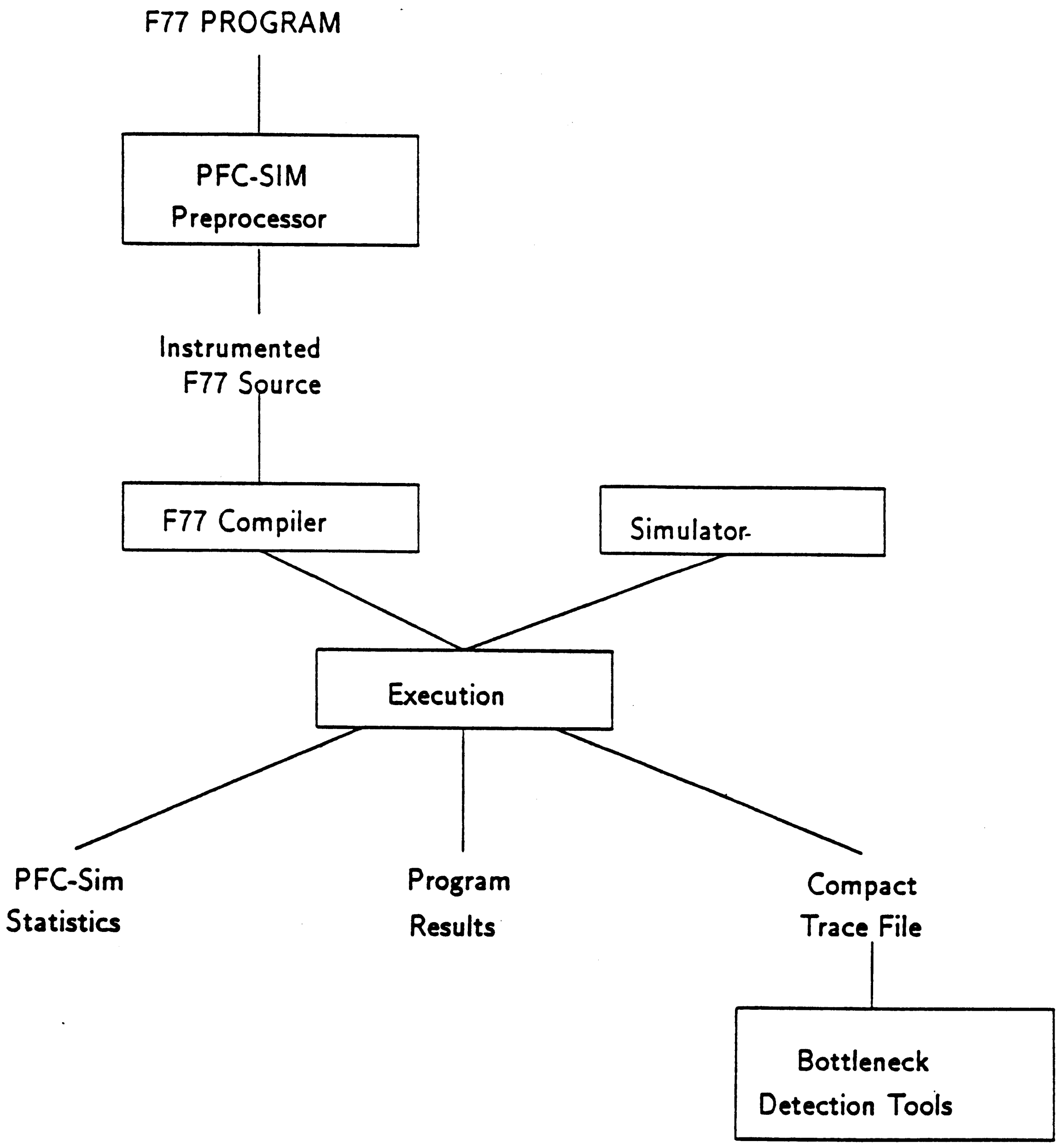

Figure 2.1 PFC-Sim

that number to the simulator, separate statistics for each event are maintained. When execution completes, the statistics about the entire program can be gathered, or subsets (i.e., single loops or individual references) can be examined.

To understand supercomputer program memory performance, a collection of actual production programs is required. Researchers at Rice University are collecting a set of computationally intensive programs known as the Rice Compiler Evaluation Program Suite (RiCEPS). Each program in the suite is written in FORTRAN77 and includes the data required for execution. Each program is either an actual production program or an "abstracted" version, where "abstracting" involves replacing proprietary routines with computationally-similar, non-proprietary versions of those routines. The execution times of the programs vary from less than one minute to several days on an IBM 3081D. As a collection of production supercomputer programs, the benchmark gives a much better idea of the supercomputer run-time performance than programs used in previous studies.

As describe in Chapter 1, a cache can be specified by a number of parameters: cache size, cache line (or block) size, replacement policy, write policy, degree of associativity, and whether it employs a prefetching mechanism. To understand the effects of these parameters, each program in RiCEPS was run with a variety of cache structures. The structures were chosen to allow each parameter to be investigated separately. Each series of simulations was run for each parameter. The series involved changing the parameter while holding all other parameters constant.

The simulations provide two basic types of information about whole programs: the hit ratio and the amount of memory traffic generated. To better understand the results of each design decision, each parameter's effect on these values was determined. Trends among the programs was examined as well as individual programs that do not follow the trends. To reduce the number of misses in a program, the compiler must locate the misses. By counting misses and hits for each reference, the hit ratio for each individual reference was determined.

Cache performance effects on program execution can also be realized from the information produced by PFC-Sim. Using the estimated execution time and the number of misses during the program, the percentage of execution time spent waiting for the memory can be determined. As supercomputers get faster and more distant (in cycles) from their main memories, this can become a pronounced performance problem. The second impact of cache performance is the amount of traffic generated to support the execution. Most modern supercomputers have found it difficult to provide enough bandwidth to keep the floating point units running at peak rates. By investigating the average amount of bandwidth required for each floating point operation, a measure of the peak effective MFLOP rate for a machine was determined.

The following two sections detail the tools used for the data cache performance studies. The programs that presently comprise RiCEPS are described. Then, PFC-Sim's design and implementation is discussed. Using the results of the experimental study, the effects of various cache design decisions on actual program performance are then discussed.

2.2 Rice Compiler Evaluation Program Suite

The Rice Computer Evaluation Program Suite (RiCEPS) is being gathered to address the lack of a publicly accessible set of computationally intensive programs. RiCEPS is a group of programs culled from production supercomputers. Each program was selected to be representative of a group of applications. The programs are substantial in size and require a significant amount of execution time. Where possible, programs that took too long have been shortened. One method used to shorten programs is to reduce the length of time being simulated (from 12 hours to 10 minutes in the case of the BARO test program). In some cases, this has lowered the total execution time on an IBM 3081D from a projected 10 days to about 3 hours. A wave analysis program (WANAL1), which iteratively finds better answers, had the number of iterations reduced from 200 to 3 (shortening the execution time from 67 hours to 80 minutes).

Work on the benchmark set is progressing in parallel with this thesis. The benchmark is expected to grow to about twenty programs by early 1989. When the benchmark programs are considered stable, the intention is to make them available through Argonne National Laboratory's netlib facility. Each program in the benchmark includes, in addition to the FORTRAN program, a brief description of what the code does (what it solves/what algorithms it uses) and at least one set of data to run the program. Some of the programs generate their own data and do not use any explicit data sets.

The experiments reported later in this chapter used a preliminary version of the benchmark containing twelve programs. Many of these programs will not be present in the final benchmark for a variety of reasons. In some cases, we do not have permission to redistribute. In other cases, we may find that a code that arrives later is more representative of programs for that class of computationally intensive problems. The programs' execution times range from one minute (MATRIX) to several hours (SIMPLE, BARO, BOAST). While most programs are 1 to 3 thousand lines, they range in size from 15 lines to over 23,000 lines. All of the programs run under the IBM VS2 FORTRAN compiler on an IBM 3081D.

The benchmark programs used for this work included:

- **MCMB** - a mixed characteristic microbial biodegradation program. Concentration is solved by continuous linears in space. Backward differencing in time is treated by segmented characteristics. Preconditioned conjugate gradient iterations are used to solve matrices for both concentration and pressure.
- **MATRIX** - a simple 100 x 100 matrix multiply.
- **BARO** - weather simulation of an isolated vortex embedded in a mean westerly flow. It uses a shallow water atmospheric model with a fourth order potential entropy conserving advection scheme and leapfrog differencing.
- **SIMPLE** - an abstracted hydrodynamics program produced by Livermore National Laboratory for distribution and tests.
- **EFIE304** - calculates the current distribution on an arbitrary body excited by a plane wave of desired amplitude H-Field. The program manipulates the given data to obtain triangular patches. Current density is calculated at the center of each edge.
- **BOAST** - a black oil reservoir simulator.
- **EULER1** - a solver for one dimensional unsteady Euler equations that utilizes spectral methods: collocation method with Chebyshev series, 1D propagating shock wave, bursting diaphragm flow, and colliding shock wave flows.
- **SHEAR** - three dimensional turbulent fluid dynamics simulation based on spectral techniques.

- **MHD2D** - solves 2D MHD equations with periodic boundary conditions on a 256 x 256 grid using the spectral method and leapfrog time differencing.
- **ONEDIM** - eigenfunctions and eigenvalues of the time independent Schroedinger equation for one dimensional potential.
- **LINPACKD** - benchmark that has been used to determine the floating point performance of the most recent computer/compiler combinations on double precision LINPACK calls. [Don88]
- **WANAL1** - boundary control of wave equations by conjugate gradient method, without initial data smoothing using a perturbed bilinear form without cutting.

2.3 PFC-Sim

PFC-Sim is a program-driven-event tracing facility consisting of three parts: a preprocessor, run-time routines. and visualization tools written for this research (see Figure 2.1). To fulfill the requirements of this research, PFC-Sim needs to run on long programs in a limited amount of space and needs to create a map between actions in the trace and the corresponding events in the source program. To eliminate the need for an enormous trace file, the simulation of memory occurs during program execution. Unique marking of each event in the program source allows a simple mapping between the source program and actions during execution.

A one hour program can access the cache billions of times. Even with very small trace entries, this would easily exceed the available disk storage. In a typical tracing simulation, a single execution of the program is traced, the trace file is compacted in some manner, and then many simulations are run on the resulting compacted trace file. Since PFC-Sim runs on programs where the original trace file is too long to store, the compaction and simulation steps run concurrently with the program being traced, eliminating the need to ever write the trace entries. This requires retracing of the entire program for every different memory simulation, but since at least 90 percent of the execution time is in the simulation routines, the cost of re-executing the program is not prohibitive.

Only computing the hit ratio of a program does not aid a programmer in determining whether a particular section of code is getting adequate memory performance.

To improve a program's memory performance, some means of determining the reference or set of references causing the majority of the misses is needed. If memory accesses are determined by looking at the assembly code produced by the program, it will be very difficult (particularly if the code is optimized) to determine to which source statement a particular load corresponds. By marking loads and stores in the program source, PFC-Sim has several advantages. The mapping between events and source program statements is simple; every event is given a unique number.

The PFC-Sim preprocessor accepts as input a FORTRAN program to be simulated and marks every load or store of an array variable in the program with a call to the simulator. The PFC-Sim simulator consists of three externally-linked FORTRAN calls which can simulate a wide variety of memory structures. The set of visualization tools present the information from the simulation in ways that allow quick identification of important events. Each of the three parts are more fully explained in the following sections.

2.3.1 PFC-Sim Preprocessor

The PFC-Sim preprocessor is embedded inside a powerful vectorizing and parallelizing source-to-source translator developed at Rice University, called the Parallel FORTRAN Converter (PFC). PFC builds and uses a dependence graph to transform a FORTRAN77 program (possibly with vector and parallel extensions) into an equivalent FORTRAN program with vector and/or parallel constructs. The output of PFC-Sim does not use any non-FORTRAN77 constructs that were not present in the original program.

PFC makes a series of passes over an abstract syntax tree. Between any of the passes, the PFC-Sim preprocessor can obtain the tree and mark events of interest. To simulate memory usage on the original input program, PFC-Sim marks the tree immediately after parsing, scanning for two types of events. The first type of event is an access of memory, either a LOAD or a STORE, and the second type of event is the crossing of basic block boundaries. Although this work only describes the memory simulation version of PFC-Sim, any operation that a compiler can detect can be counted dynamically in the same manner. At the present time, a second version of PFC-Sim exists that gathers statistics about vector operations as detected by PFC. The total counts of floating point operations in the RiCEPS programs, used in Section 2.4.6, are from this version of PFC-Sim.

PFC-Sim's approach of adding lines to the program to perform dynamic counts is similar to a technique used by Kumar to measure the total potential parallelism in programs. In his paper [Kum87], Kumar discussed experiments that determined, for each statement, the earliest cycle during which the data used in a statement was available and the earliest cycle that the execution was guaranteed. From these numbers, the earliest cycle that the result of the statement could be available was determined. By measuring the longest chain of statements through the program, Kumar determined the minimum execution time of the program. The maximum average parallelism is then the number of statements divided by the minimum time, and the maximum parallelism is the maximum number of statements executed at any given time.

The PFC-Sim preprocessor uses Kumar's idea, inserting computation to simulate memory usage before every memory access. Every program statement is checked to see what events would occur during its execution. If the event can be simulated in one or two statements then those statements are inserted preceding the actual statement. This technique is used to keep track of actual running time. When more complex simulation is needed, a subroutine call is inserted. The simulator subroutines must then be included when the program is linked. This technique is used for memory simulation.

The PFC-Sim preprocessor inserts two types of calls, namely LOAD and STORE, in front of ordinary statements. Calls to LOAD and STORE are treated by the simulator in the same way that hardware would treat the equivalent instructions. Each call passes the address being accessed, the length of the element, current simulation time and a unique identifier for the particular reference.

Every statement is examined, and two lists are built from the memory references that occur during the statement. The first list contains the references that would require a load during execution. The second list contains the references that are written to memory by the statement and that require a store. A symbol table is used to determine if each reference is an array or a scalar. The default handling of scalars is to ignore the references that they generate. The number of blocks occupied by scalars will normally be very small, and preliminary testing showed that they cause very few additional misses to occur. A second reason to ignore scalars is that many will be kept in registers between uses when a good global register allocator is available. Ignoring scalar references is also practical. The simulator requires three times as long to run when scalars are included.

All statistics about memory performance are kept in the simulator. When program execution completes, the statistics must be saved. Every FORTRAN RETURN or STOP statement in the main routine of a program is preceded by a call to SIMFNL. SIMFNL builds a data set with the information that was gathered on that particular run.

Besides inserting statements for memory references, the preprocessor also divides the program into basic blocks. During execution, profiling data is generated, and simulated execution time is kept. The simulation slows the program down, but event frequency can be determined from the simulated time. An accurate estimate of the execution time of a single basic block is possible in the absence of any memory delays. By incrementing the simulation time at the beginning of any basic block and adding a miss penalty when appropriate, an accurate clock is maintained.

When simulating parallel processors, maintaining a running time of the program will be important for two reasons. First, calculation of cache miss rates requires knowledge of the time, and when parallel memories are modeled, ordering of events can be determined by comparing two execution times. By ordering instruction streams across processors, memory interference between multiple processors can be calculated.

One effect of the preprocessor is to approximately double the size of a program. Figure 2.2 shows matrix multiply before and after the preprocessor. Most of the changes are straightforward.

2.3.2 Memory Simulation

The run-time package for memory simulation consists of five routines. The three mentioned in the previous section and two that are called internally from LOAD and STORE. An initialization routine is called the first time that the memory simulator is activated to set up all of the control structures. The initialization routine reads in the specifications for the requested memory and performs the appropriate actions. The second routine is an automatic prefetch mechanism that can be used to simulate hardware prefetching. Almost any memory structure can be modeled by the package.

```
DO I = 1, N
    DO J = 1, N
        A(I,J) = 0
        DO K = 1, N
            A(I,J) = A(I,J) + B(I,K)*C(K,J)
        ENDDO
    ENDDO
ENDDO
```

becomes

```
DO I = 1, N
    DO J = 1, N
        CALL STORE(A(I,J),4,TIME,1)
        A(I,J) = 0
        DO K = 1, N
            CALL LOAD(A(I,J),4,TIME,2)
            CALL LOAD(B(I,K),4,TIME,3)
            CALL LOAD(C(K,J),4,TIME,4)
            CALL STORE(A(I,J),4,TIME,5)
            A(I,J) = A(I,J) + B(I,K)*C(K,J)
        ENDDO
    ENDDO
ENDDO
```

Figure 2.2 Matrix Multiply - Before and After PFC-Sim

The initialization routine reads in a file that contains seven parameters. The first five specify the type of memory to be modeled, the sixth gives the frequency of trace entries for one of the visualization tools, and the last activates hardware prefetching. The five memory specifications are

- number of memory blocks
- size of each memory block
- associativity
- replacement algorithm
- write policy

The first memory parameter is the number of blocks in the memory. The number of blocks should be divisible by the specified set associativity. In the current implementation, the number of blocks must be no greater than 256K, but this can easily be changed if more are required. Since FORTRAN requires static allocation of arrays, some limit must exist. Without the memory constraints of IBM's VM operating system (without the XA feature), the default maximum number of blocks could be higher.

The second parameter is the block size, or length of a cache line in bytes. The size of each line times the number of blocks defines the size of memory being modeled (SIZE = NUMBER*BLOCKSIZE). The size is specified in bytes and can be any positive integer, although memories normally have sizes that are powers of two.

Associativity is the third parameter. Again, any positive integer is acceptable. When the associativity does not divide the number of blocks evenly, the memory simulator may behave in an unpredictable manner. A direct-mapped memory has a set associativity of one, a fully associative memory has the set size equal to the number of blocks, and an X-way set associativity is requested by setting the third parameter to X. The associative memory is simulated with a linked list, causing fully associative simulations to be substantially slower than the other types.

The fourth parameter is the replacement algorithm to be used. Four different replacement strategies are supported.

- Least Recently Used (LRU) — the most common in practice.

- First In, First Out (FIFO)
- Random (RAND)
- Optimal (OPT) — furthest next use, impossible in practice.

The fifth memory parameter declares the write policy to be used. Three write policies are presently supported.

- Write-back — writes modified blocks only when cache line is being replaced.
- Write-through — writes modified blocks back immediately.
- Write-through, no-load on write- that if a block being written is not present in the cache, it is not placed into the cache.

These options cover most caches and main memories available in present architectures. The OPT replacement algorithm is a useful tool in detecting when transformations could have a noticeable impact on performance. The main problem with implementing the OPT algorithm is that it requires up to one block for every accessed variable in the program. Since PFC-Sim has an imposed limit of 16M of user storage in its present implementation, this has prevented testing of larger programs using an OPT cache.

After the initialization routine has identified the memory to be simulated, the memory control structure is initialized. The initial cache contains no values. After memory initialization, LOAD and STORE calls perform almost identical actions. Both routines emulate the memory lookup function and increment the arrays that count either hits or misses for the unique identifier passed with the call. The difference between the calls is that STORE marks the block as dirty and may require that the block be immediately written out. Besides the arrays to count hits and misses, a traffic array counts the number of bytes that are moved between memories due to each reference.

To provide a means for the programmer to examine memory performance during execution, a compact trace file can be generated. Instead of providing a reference by reference summary, a single entry is generated for every time interval specified. The time interval is the sixth parameter and is given in microseconds (of simulated time).

Trace entires are also generated every 5000 misses, giving more detailed information in code sections with poor hit rates.

The last parameter to the PFC-Sim run-time routines activates hardware prefetching. To reduce the number of memory misses, several people have suggested an automatic prefetching mechanism. In the later sections of this dissertation, the results of testing one particular version are given. The automatic prefetch that is supported places the next sequential memory block into the memory whenever the first byte of a cache line is accessed. This prefetch function assumes that memory is accessed sequentially by most programs.

2.3.3 Visualization Tools

Many different visualization tools can be developed using compact trace and statistics files. Two have been developed. The first tool is a browser (on a color IBM 3279) that allows a user to color every reference in the program according to the hit ratio of that reference. For example, by coloring all of the low hit ratios red, while leaving the remaining references green, references that have the greatest possibility of being improved are easy to locate.

The second tool uses the trace file to graph the miss rate as a function of execution time. The trace file may be viewed at any magnification from the interval between the compact trace file entries to a level which allows the entire program to be viewed. To allow the programmer to locate portions of the code that have poorer memory performance than expected, the tool provides a means of moving from a particular trace file entry to the point in the program which is currently being executed.

2.3.4 Potential Uses of PFC-Sim

PFC-Sim's basic structure is a powerful mechanism for developing a range of tools to examine program behavior. As it was designed for this research, PFC-Sim is both easy to modify and flexible in its possible uses.

The PFC-Sim front-end examines the source of a program, identifies and marks events of interest. During execution, each one of the events can be counted or otherwise noted. As implemented, the basic event is memory references and the dynamic action on each event is to model some type of memory hardware. The structure, however, can be easily modified for other applications. One graduate student, working part-time and having had no previous experience with the compiler (or the operating system environment it lives on), modified PFC-Sim to determine the dynamic vectorization percentage of floating point operations, in about two months. Developing new visualization tools for the back-end required less than a week each. Modifying the output into a form that could be easily accepted by a graphics tool was straightforward.

PFC-Sim can obviously be modified to gather any type of dynamic statistical information desired. More interesting modifications of PFC-Sim are for applications in debugging and performance analysis. By simulating the effects of a machine around the execution of a program, both data and program breakpoints are easy to implement. Other debugging activities can also be implemented. While PFC-Sim would not provide major debugging functionality over present sequential systems, it would provide benefits in parallel debugging. A careful implementation could maintain clocks for each parallel stream and duplicate actual event ordering for specific hardware. Duplication of actual event ordering may simplify debugging of race condition problems. Using PFC-Sim for performance analysis (or performance debugging) is already done to some degree. As a memory simulator, PFC-Sim generates profiling data for the basic blocks in each program. To determine the effectiveness of some transformations in this work, profiling data was used to limit examination of the programs to the most frequently executed basic blocks. Generating more sophisticated tools on top of the available data (and expanding the amount of data collected) would not be difficult.

Modifications to the basic structure of PFC-Sim can be used for a variety of program evaluation and debugging purposes. For some applications, like dynamic statistic gathering, it seems to be exceptionally well-suited to produce information in forms that can be used by other tools. For other applications, like debugging, the PFC-Sim structure is a new approach that could be very useful. While PFC-Sim was developed for this research, this dissertation uses just one of its many possible configurations to examine the performance of data caches on actual supercomputer applications.

2.4 Experiments

The experiments in this section show the effectiveness cache structures for computationally intensive programs. In addition to determining how the hit ratio for the program is affected by the cache structure, the total data traffic between the levels of the memory hierarchy is measured. Every reference in every program will have its individual hit ratio measured dynamically. Compiler transformations can reorder references, increasing or decreasing the distance between consecutive usages of a memory location. If compiler transformations may be successful in reducing cache misses, then it will be useful to examine the effects of transformations on individual references. The eventual goal is to develop compiler predictions of memory performance.

Hundreds of possible cache configurations exist, and testing all possible configurations is impossible with available computer resources. To test the effect of a particular cache parameter, a base cache was defined and a single parameter was varied on each simulation. The selected base cache was a 32K LRU, 4-way set associative cache, with a one-word cache line, using a write-back store policy and doing no prefetching.

2.4.1 Cache Size

When describing a cache, the first characteristic is normally its size. During the design of a computer, the selected cache size is often the largest that can fit in a reasonable fraction of the available physical space and that will not represent an unreasonable fraction of the total machine cost. Within the physical and practical constraints, a larger cache results in a higher hit ratio. The first series of cache simulations examines the effects of increasing cache size from 16K to 256Kbytes.

For every examined cache parameter, two graphs are presented: the generated hit ratio, and the required data traffic. The hit ratio graph shows the absolute hit ratios for each program and the arithmetic average for all programs. The programs in RiCEPS vary greatly in execution time. To graph the amount of data traffic for each cache's structure in a manner that is not dominated by the longer programs, the data traffic graph is normalized. Each data point is a percentage of the minimum data traffic for that program over all graphed caches. (Every line has at least one point at one, and none below one.) This normalizes the graphed points and allows discussion of the relative effect on data transferred between a cache and main memory as a function of cache parameters.

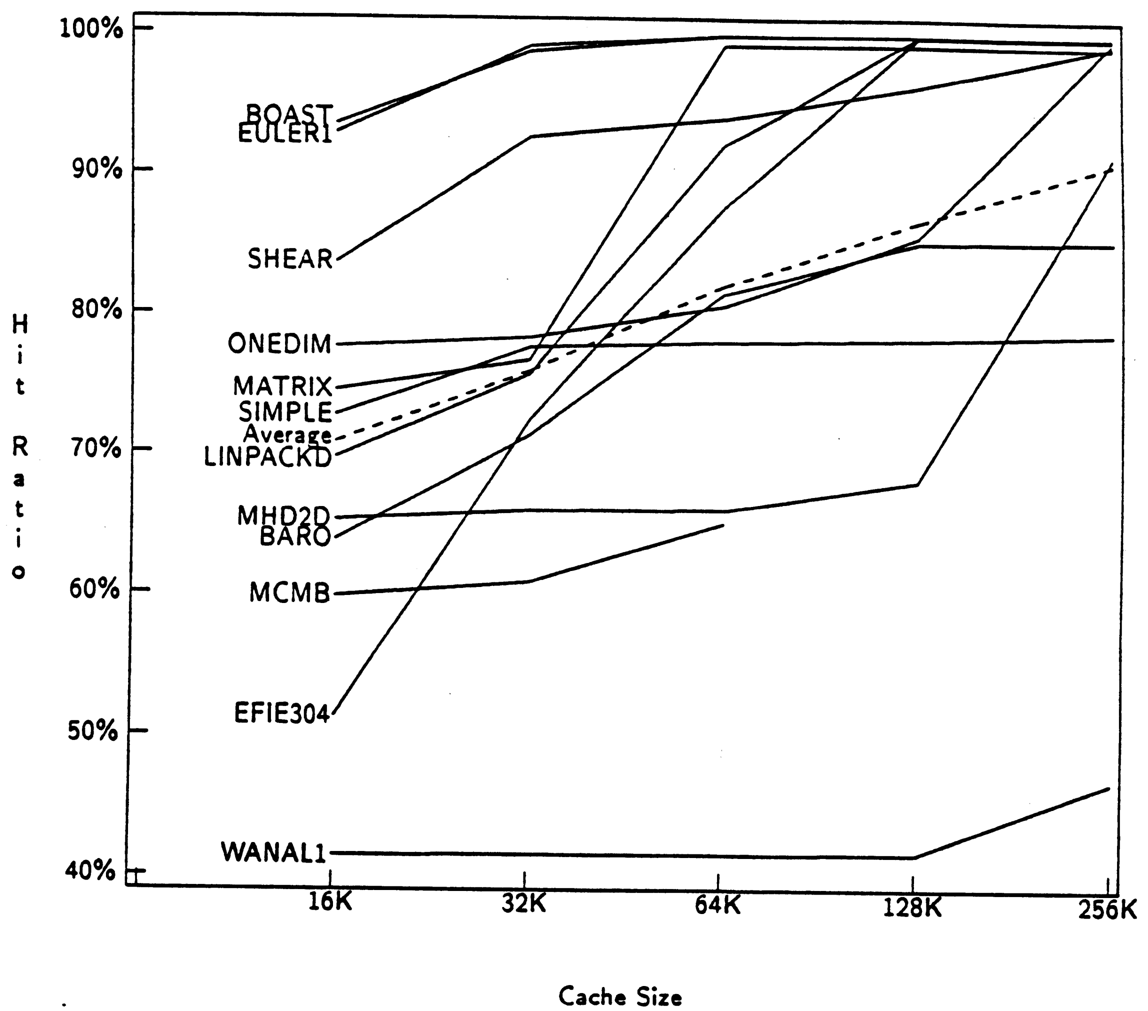

Figure 2.3 Effects of Cache Size - Hit Ratios

As shown in Figure 2.3, all of the programs' hit ratios increased as data cache size increased. An average 16K cache hit 70.8% of the time, a 32K cache hit 75.9% of the time, and the hit ratio continued to rise smoothly to a peak of 90.9% for a 256K cache. The average hit ratio line is nearly straight. Each doubling of the data cache size increased the hit ratio by approximately 5%. The average hit ratios of the larger caches are dominated by a few programs with very large working sets. For 256K caches, only four of the eleven programs[1] had a significant percentage of misses. Over 50% of the misses were generated by a single program WANAL1.

Overall, the average effect of doubling the cache size was to reduce the miss ratio by about 21% for each doubling of cache size. This was lower than the 27% that Alan Smith found with his sample group[Smi82]. The average improvement as cache size increased was lower for this set of programs due to differences in the programs used in the two studies. The larger working sets of the RiCEPS programs increased the probability that a program's working set does not fit into a given cache size. This decreased the effectiveness of each doubling of the cache size.

The programs did not react uniformly to the increasing cache size. EFIE304's hit ratio increased dramatically at both 32 and 64K and very few misses occurred with a 128K data cache. MATRIX's hit ratio increased less than 5% between 16 and 32K, but the misses were almost non-existent with a 64K cache. SIMPLE showed improvement when the cache size increased from 16 to 32K, but few additional hits occurred if the cache was increased to 256K. WANAL1's hit ratio increased less than 0.1% when the cache was increased from 16K to 128K (and only 6% at 256K).

Before a compiler can understand how cache size affects a program's performance, the compiler writer needs to understand why the programs use the larger caches with such differing effectiveness. To understand why programs react so differently, let us consider the performance of MATRIX in detail. The first doubling in cache size had little effect (hit ratio increased by 2%), and the second was very effective (hit ratio increased by 22%). MATRIX only has 5 array accesses, the first when the result element is initialized to zero and the remaining four inside the main loop ($a = A + B * C$). The number of misses produced by four of the references did not change between any of the simulations. The only value that changed is the number

[1]Hit Ratios are not available for MCMB above 64K because the 16Mbyte virtual address space available under the IBM VM Operating system will not contain both the program's data space and the larger cache data structure required.

of misses that occur when loading an element from the C array. When only a 16K cache existed, the load of C missed 100% of the time (1,000,000 misses, 0 hits). Increasing the cache size to 32K caused the hit ratio for the C reference to increase slightly, (8.74% with 912.576 misses. 87,424 hits). Increasing the cache size further to 64k caused the hit ratio for C to jump to 99% (10,000 misses, 990,000 hits) where it remained for 128K and 256K caches. The radical difference can be explained by computing working set sizes.

A 100 x 100 single precision FORTRAN array occupies 40,000 words. In the naive implementation of matrix multiplication that was used, the entire C array is referenced before any element in C is accessed a second time. This causes a perfect LRU mechanism to remove every element before it is ever reused. The only way that an element can still be present in caches smaller than 40,000 bytes is if the LRU replacement is less than perfect. When the number of words in the array exceeds the amount of storage needed for the entire array (and the active column of B), the hit ratio rises very quickly. The hit ratio, graphed as a function of size, is not a smooth function, but a series of steps that occur whenever the working set size of a particular reference is exceeded.

The normalized data traffic graph, illustrated in Figure 2.4, shows the strain that a particular cache/program pair puts on the main memory. The larger caches have potentially smaller fractions of the misses than a small cache. When fewer misses are generated, less data needs to be moved between the main memory and the cache. The programs that require massively more bandwidth at 16K than at 256K are those with working set sizes below 256K yet above 16K. These programs never push values out of large caches, but a small cache will cycle through the values a large number of times before completing execution of the program. Programs with larger working set sizes may cycle through the cache entries even with large caches and demonstrate less disparity between traffic for different memory sizes.

Overall, increasing the cache size is sporadically very effective at reducing cache misses and memory traffic between the cache and the main memory. When the cache size exceeds the working set size for a portion of the program, the increase is very effective; otherwise, the increase has very little effect on the hit ratio. The working set size of a program will be a difficult value to identify during compilation since it can depend both on the program and the input data.

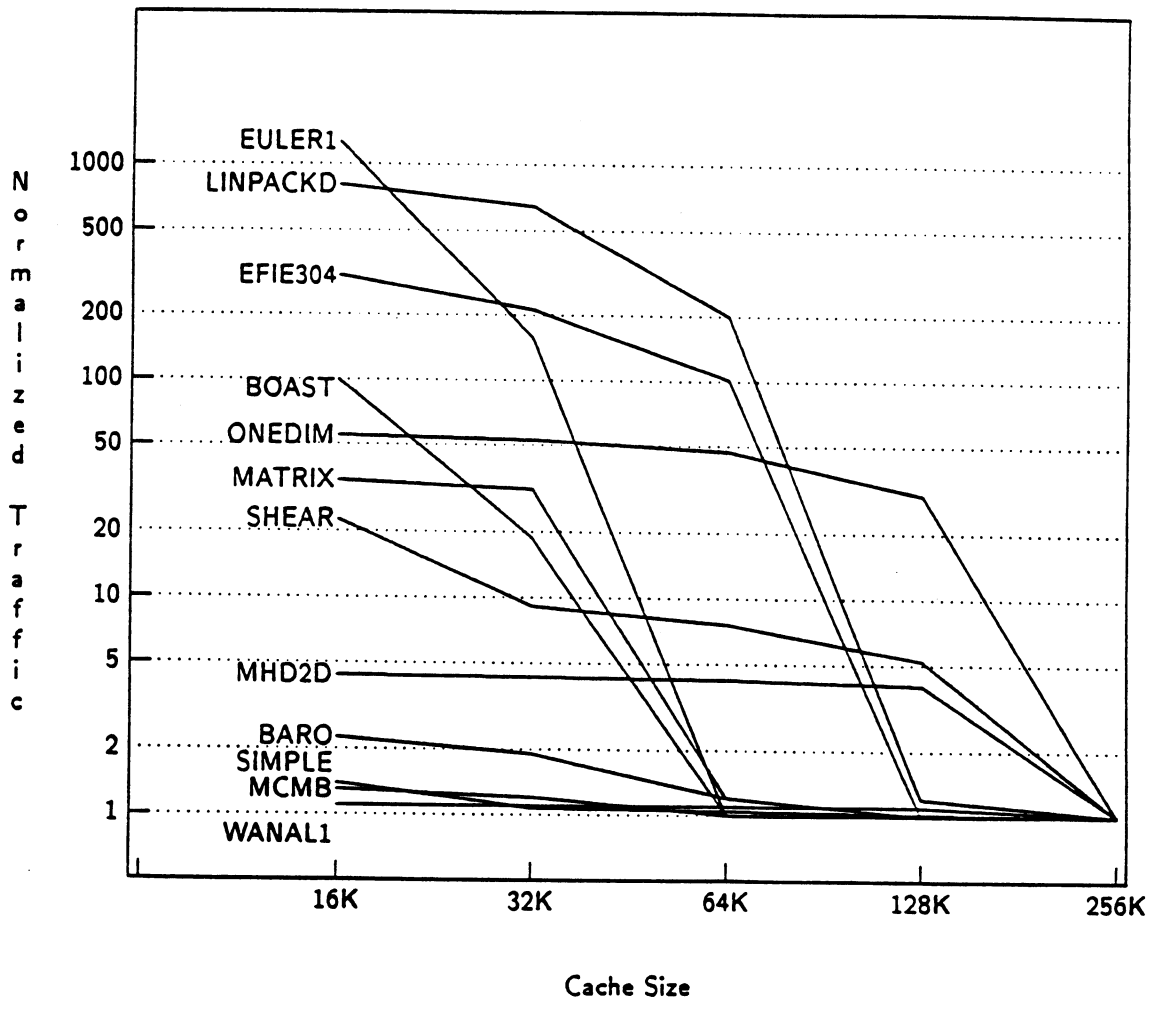

Figure 2.4 Effects of Cache Size - Normalized Data Traffic

2.4.2 Set Associativity

The second cache parameter studied was associativity. In Figure 2.5, hit ratios for the RiCEPS programs are shown under the various set associativities. For the majority of tested programs, the associativities all performed approximately the same. Each of the cache associativities had at least two programs for which it produced fewer misses than the other three tested associativities. The difference in misses was not significant except for four programs: MATRIX, SIMPLE, BOAST and EULER1. MATRIX performed much better with a direct-mapped cache than any set associative cache, while SIMPLE, BOAST and EULER1 had higher hit ratios with an 8-way set associative cache.

It is particularly surprising that MATRIX incurs over twice as many misses with an 8-way set associative cache than with a direct-mapped cache. When examined more closely, it became obvious that MATRIX displayed pathological LRU behavior. The 10,000 elements in the C array are accessed sequentially through a cache with only 8,192 (8K) entries.

To see the behavior, consider sequential accessing of 5 elements with a 4 element cache. With a 4-way (fully) associative LRU cache, the first four values go into the cache (4,3,2,1). The fifth element does not fit and pushes the first value out (5,4,3,2). During the second time through the array, the first element replaces the oldest element (1,5,4,3). This continues and the value requested is never found as it was replaced by the previous access.

A direct-mapped cache only allows each element to enter one location. Normally, sequential values are mapped to different cache locations. If 1 and 5 share a location, the previous accesses now find some locations present in the cache. The first four elements enter the cache. The fifth accessed element replaces the first. To begin the second cycle, the first element replaces the fifth because they share one location. The next three accesses are located in the cache (hits), and then five again replaces one. Each pass through the array after the first pass results in three hits and two misses. With this type of access pattern, MATRIX on a direct-mapped cache produces slightly under half of the misses that resulted when a 4-way associative cache was used.

For BARO and EULER1, the overall hit ratios were very high. Since the programs' working sets fit into the cache, the LRU mechanism prevents values that will be needed again from accidentally being removed from the cache. For these programs,

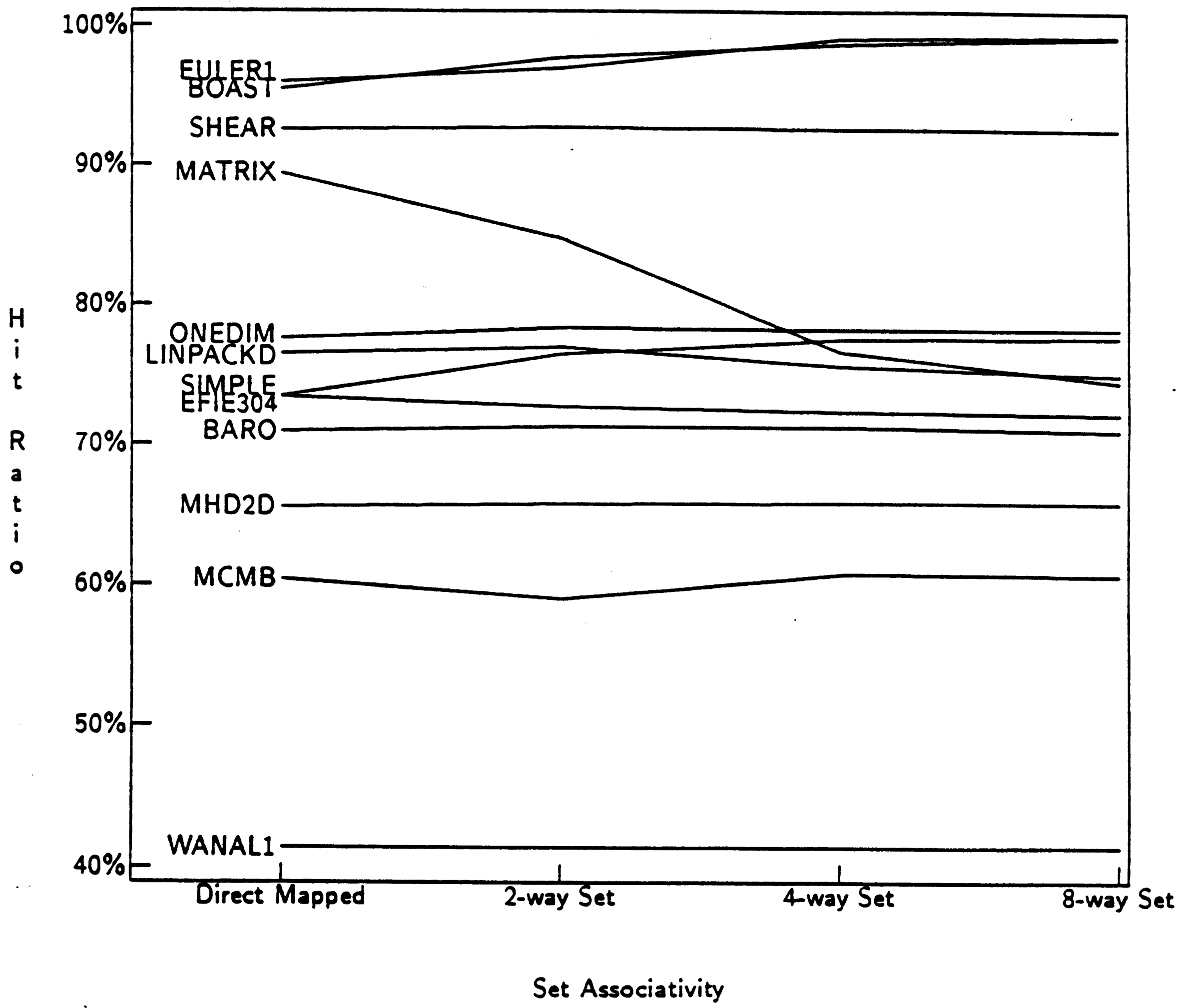

Figure 2.5 Effects of Set Associativity - Hit Ratios

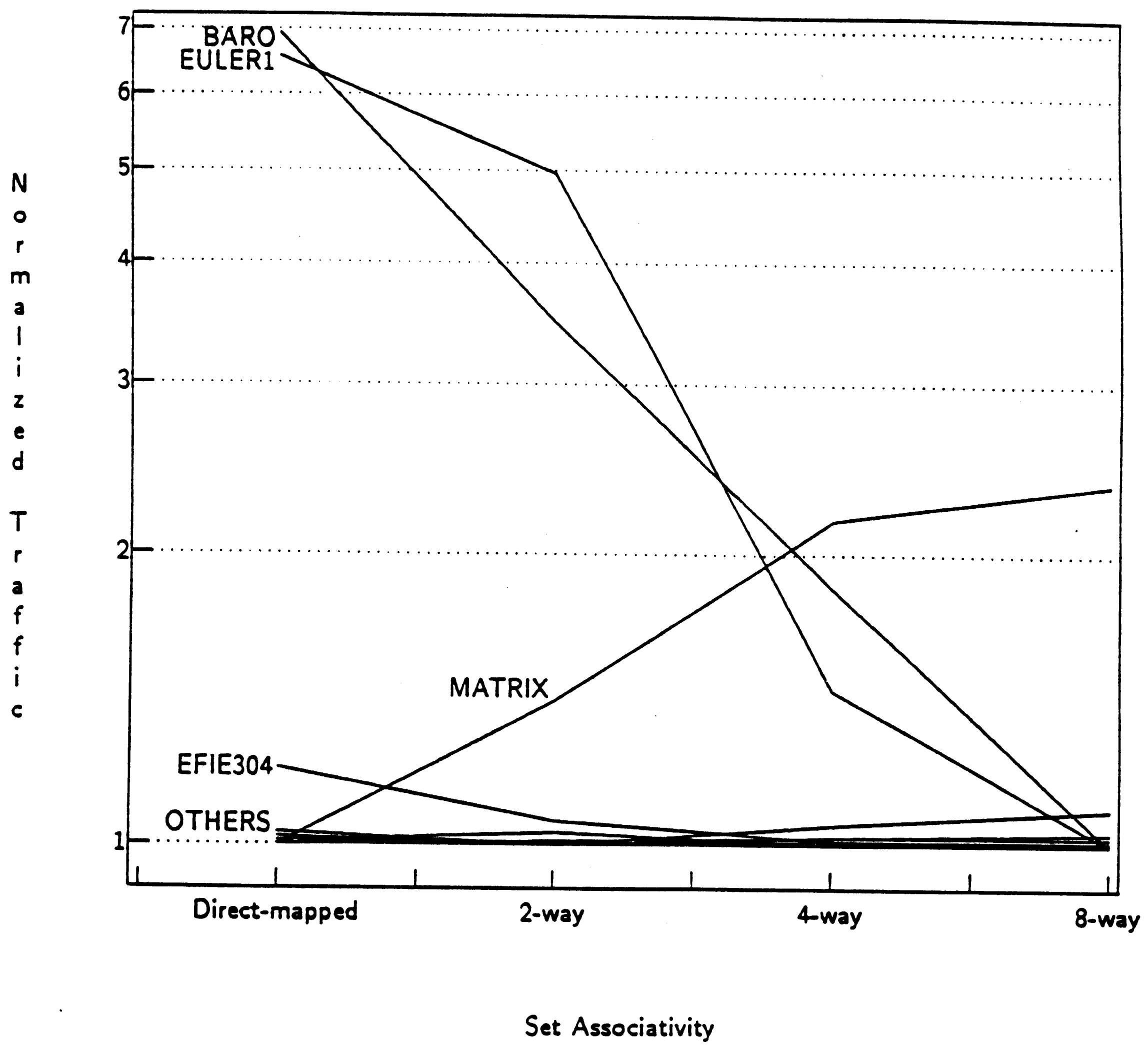

Figure 2.6 Effects of Set Associativity - Normalized Data Traffic

two arrays used blocks which crossed the blocks used by the other, causing a direct mapped cache to discard elements that would be used again. The effect of these premature bumps is very noticeable in Figure 2.6. The values in the cache are used so many times without missing that a small change in the number of misses causes a dramatic increase in traffic between the cache and main memory.

For most programs, the lack of any significant difference in hit ratios supports a belief that pathological situations are uncommon in practice. Data caches with smaller set sizes require less hardware to implement (i.e., fewer comparators), and may perform as well as memories with higher associativities. This will be particularly true when the compiler attempts to keep arrays used during the same iteration from mapping to the same slots (A(1) and B(1) should not use the same slot for any A or B).

2.4.3 Write Policy

Since write-back and write-through both make the same decision on when to load a value into the cache, no difference between the hit ratios of these two policies is possible. For all tested programs, the write-through no-load policy produced a slightly lower hit ratio (within 1%)(see Figure 2.7). Long cache lines would increase this difference since consecutive writes to a single line would cause two misses for the no-load version and only one for the standard version of write-through memory. The difference between hit ratios of write-through and write-through with no-load is small, and could easily be outweighed by other factors.

The extra traffic generated between the cache and main memory by pushing every value through to the main memory on store is shown in Figure 2.8. The write-through caches transferred between 50% and 100% more bytes to and from the main memory for 10 of the 12 programs. The other two programs, BOAST and SHEAR, generated almost 4 times the traffic with a write-through cache as with a write-back cache. The average increase in traffic for the 12 programs was 97%. Both BOAST and SHEAR had high hit ratios. Each repeated calculation of an array value caused additional traffic with a write-through cache that the write-back cache avoided by maintaining the value in cache until all modifications of the array were completed.

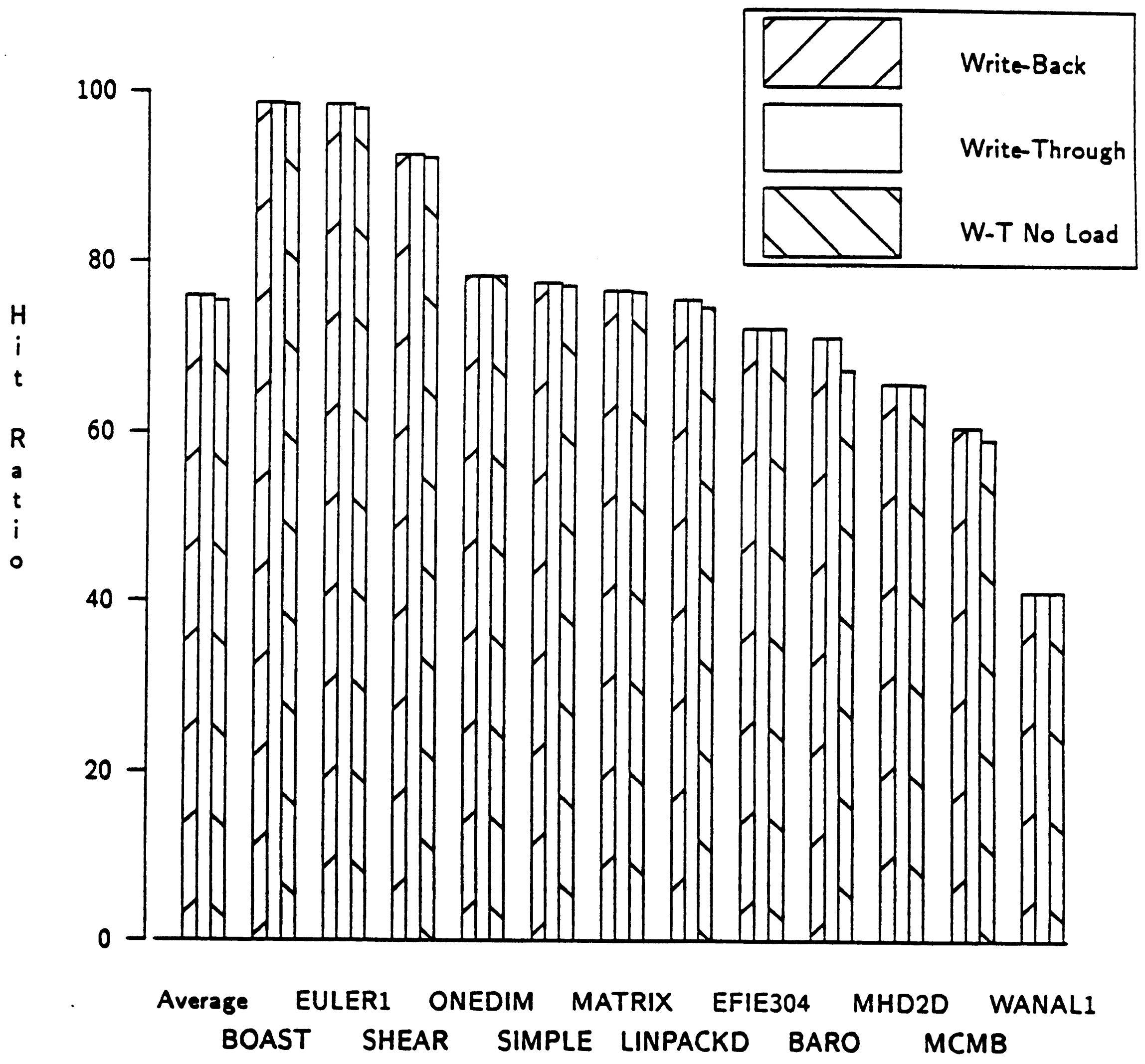

Figure 2.7 Effects of Various Write Policies - Hit Ratios

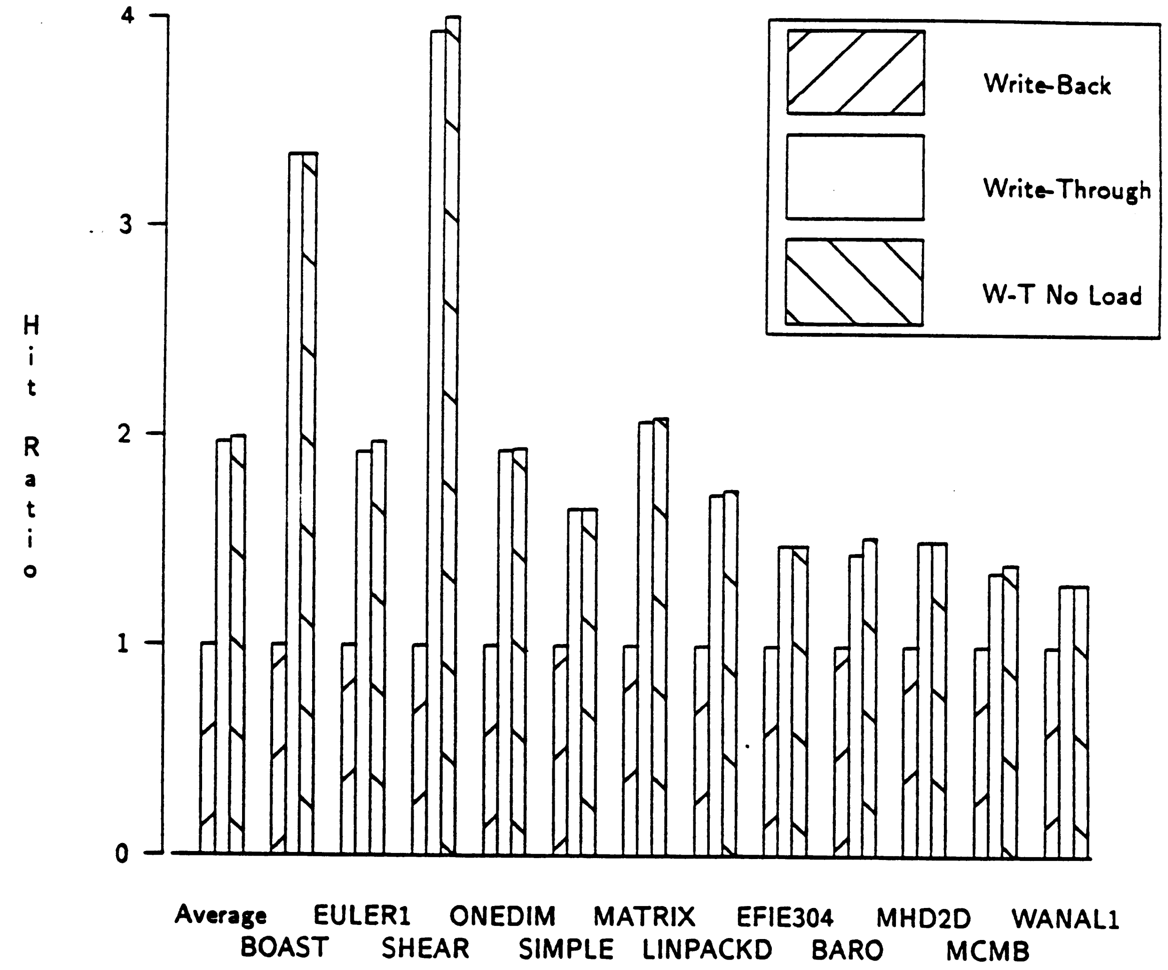

Figure 2.8 Effects of Various Write Policies - Normalized Data Traffic

2.4.4 Prefetching

In the preceding simulations, the cache memory policies attempted to retain data between accesses. This section investigates mechanisms that attempt to predict which data will be used next and bring that data into the cache before it is requested. Two different prefetching methods were tested. The first and most common method was to have cache blocks that are more than one data item long. Also examined was a hardware prefetching mechanism that brings the next sequential block into the cache at every access. Both prefetching strategies work best when memory is accessed sequentially.

Long Cache Lines

To determine the degree of locality among the references and the effects of long cache lines on memory performance, we again examined the hit ratios and the data traffic observed for the various line lengths on the programs in RiCEPS.

Three cache line lengths (4, 64, and 128 bytes) were simulated. In Figure 2.9, we see that the longer cache lines substantially reduced the misses that occur during execution. Several programs, including BARO, showed almost perfect prefetching, (i.e., the number of misses for 4 byte cache lines was 16 times the number of misses for 64 byte lines). Doubling the line length to 128 bytes produced mixed results. Several programs (e.g., BARO) had the number of misses reduced by almost 50%. Others, (e.g., MATRIX), showed little difference in the number of misses between 64 and 128 byte lines. A third group, including EFIE304 and MHD2D, had fewer misses for 64 than 128 byte lines.

The programs fall into two distinct groups upon examining the total amount of transferred data (Figure 2.10). For programs where the prefetching was spectacularly successful, the increase in traffic was minimal (less than 10% for WANAL1 and BARO). The more common case was for traffic to increase from three to six fold for 64 byte lines and five to eight fold for the 128 byte lines.

Overall, prefetching with long cache lines was successful for the computationally intensive programs in the benchmark. The cost of long cache lines showed up in the increased required bandwidth. Many architectures hide this cost by building wide busses and interleaving main memory so that the wider cache lines are supported

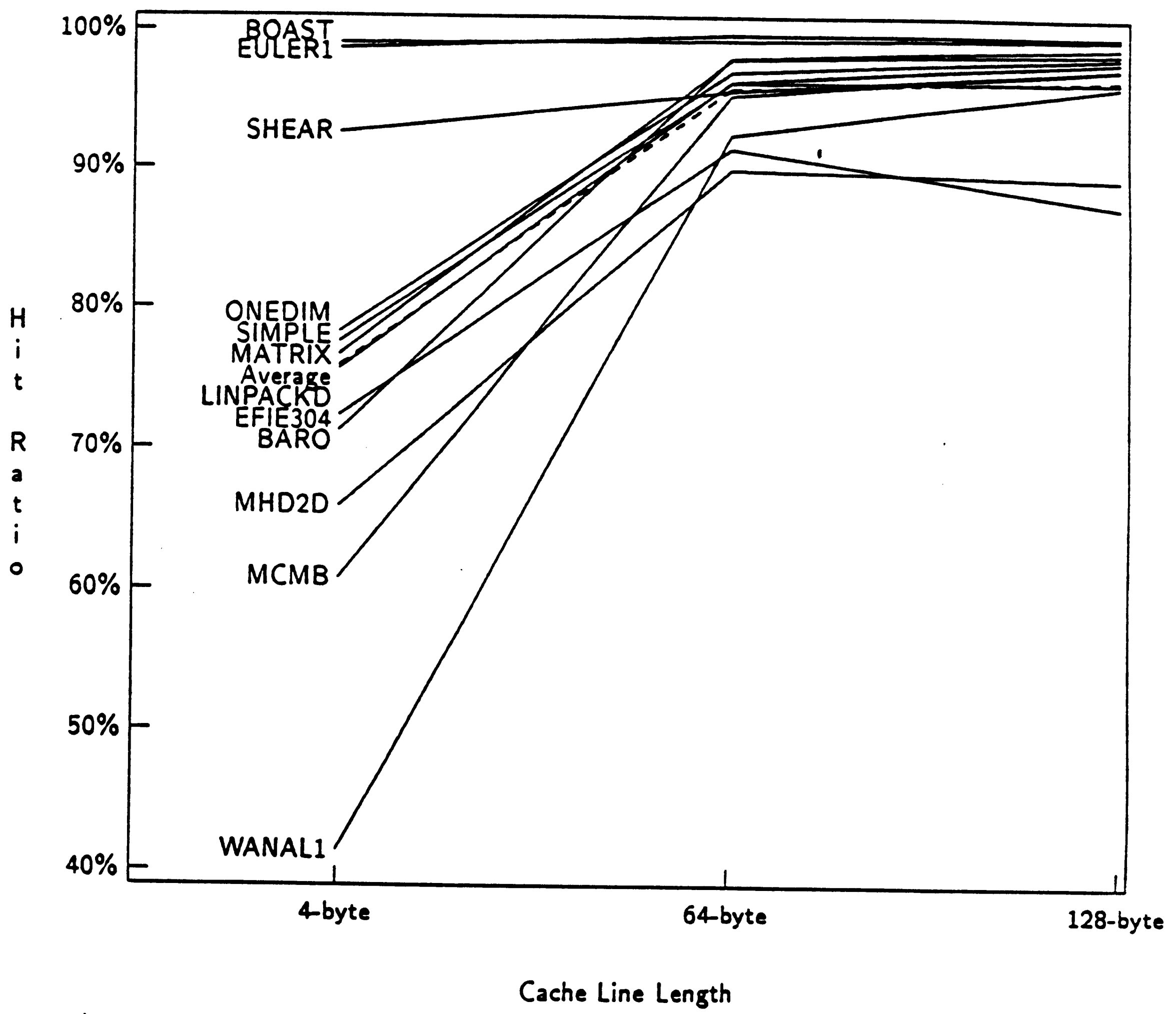

Figure 2.9 Effects of Cache Line Lengths - Hit Ratios

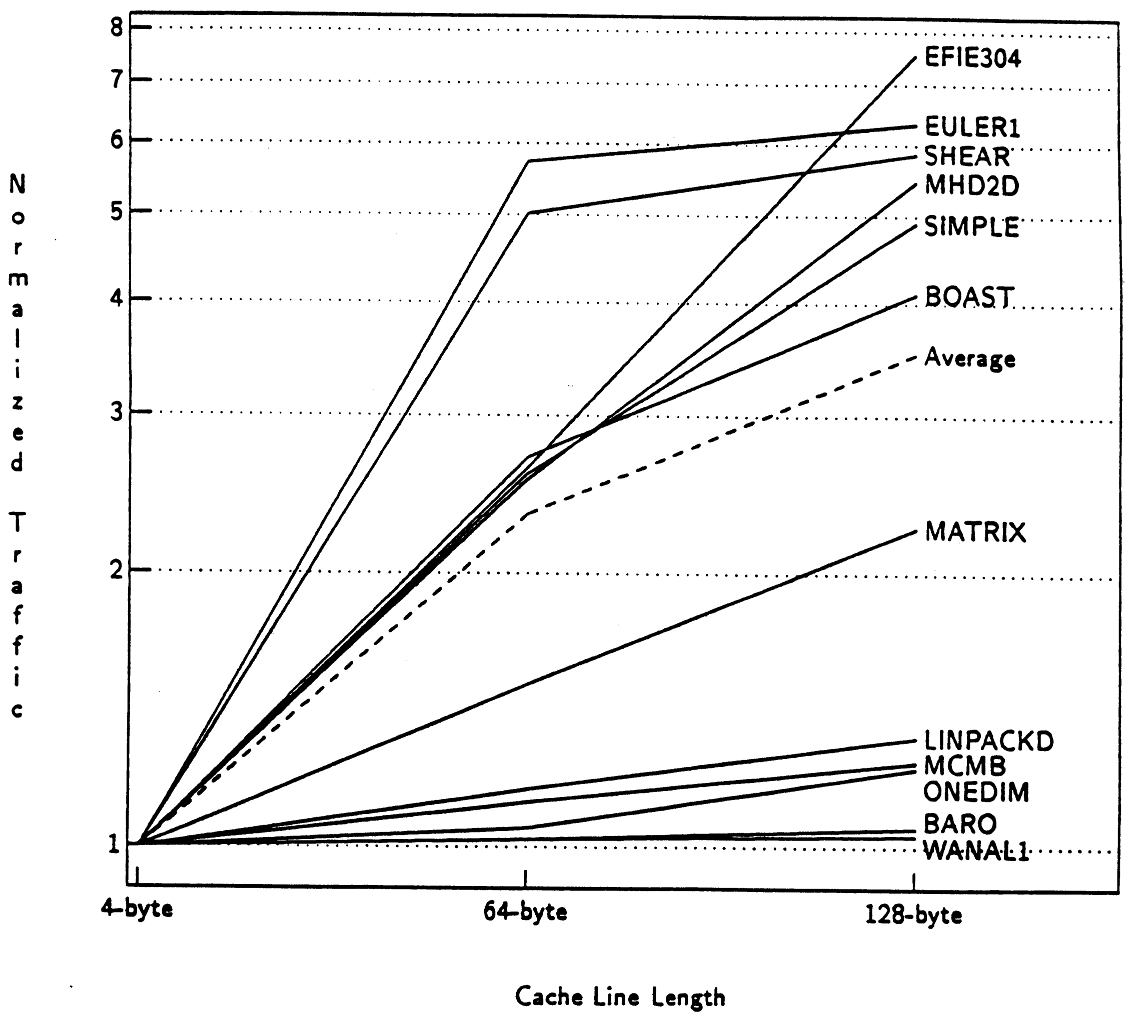

Figure 2.10 Effects of Cache Line Lengths - Normalized Data Traffic

with a single transfer. This allows a higher bandwidth to be supported for wide cache lines than the bandwidth that was supported by single word lines.

The increase in bandwidth from long cache lines and write-through policies will be cumulative. On average, the amount of data moved for a write-through 128 byte cache line will be over 10 times the amount required for a write-back 4-byte cache line.

Hardware Prefetching

When hardware prefetching was enabled, the programs separated into two approximately equal sized groups. Figure 2.11 shows one group of programs where misses were almost completely eliminated. Two of the programs in this group already had very few misses, but three programs (SIMPLE, MATRIX, and BARO) showed stunning improvements in hit ratios. The second group showed almost no improvement using hardware prefetching. BARO appears to have been modified to allow easy vectorization. The accesses to memory are arranged to allow all vectors to have strides of one. With vector strides of one, the prefetching accurately predicted the next required value.

The overhead in data traffic for hardware prefetching was uniformly very low. Only three programs (Figure 2.12) showed more than 1% increase in data traffic. MCMB increased by 3%, SHEAR by 10% and EULER1 by 50%. EULER1's increase in traffic occurs because hardware prefetching actually produced slightly more cache misses than the non-prefetching version. Whenever a prefetch occurs that is not used, it pushes some value out of the cache. Occasionally hardware prefetching pushes out a value that EULER1 would have otherwise reused. The small number of extra loads from this effect are magnified because EULER1 does not otherwise push items out prematurely.

Hardware prefetching is a big win for programs with column-wise accesses and produces very little overhead for any program. This method of hardware prefetching seems to be a profitable feature that any computer designed for computationally intensive programs should consider.

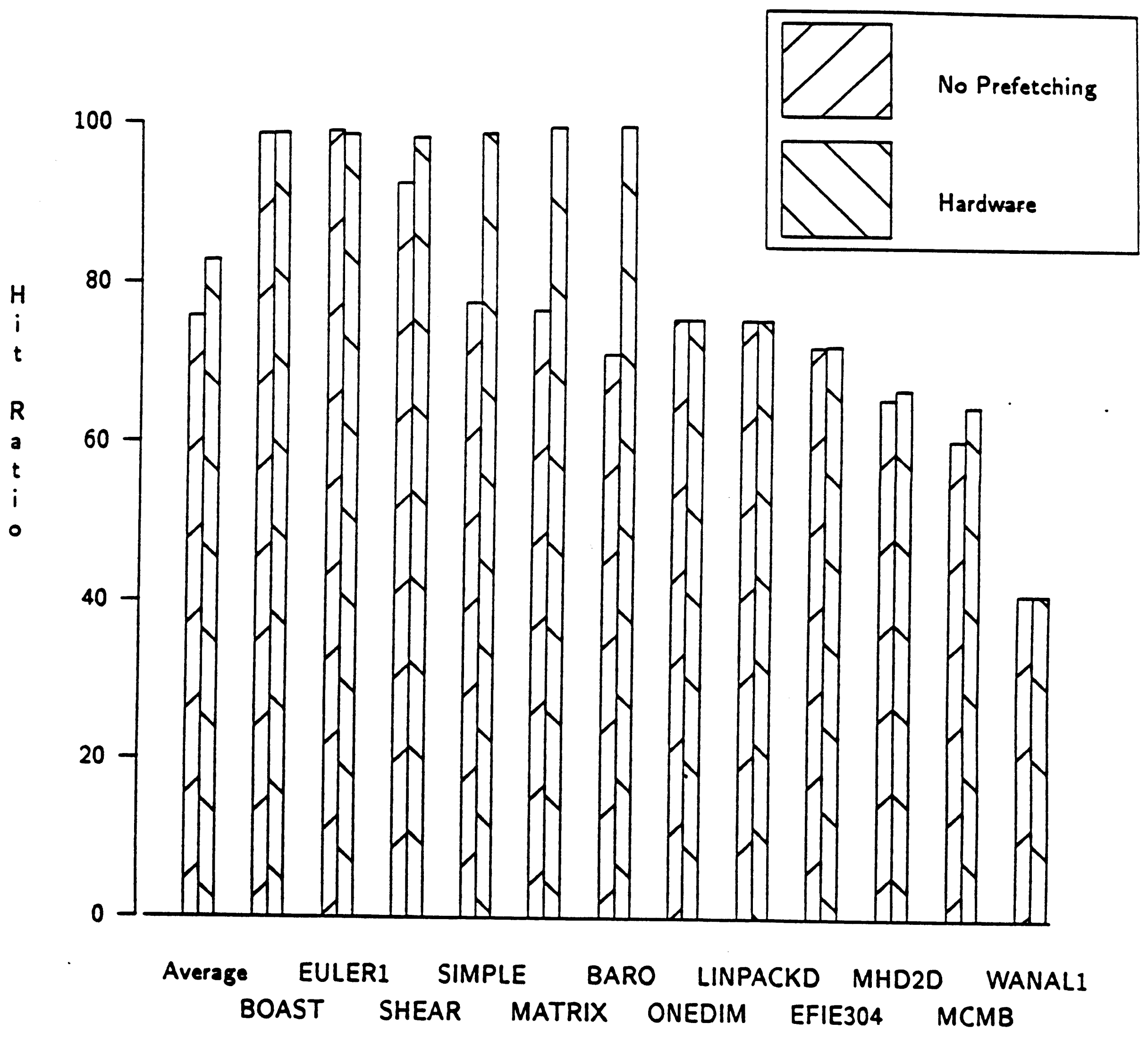

Figure 2.11 Effects of Hardware Prefetching - Hit Ratios

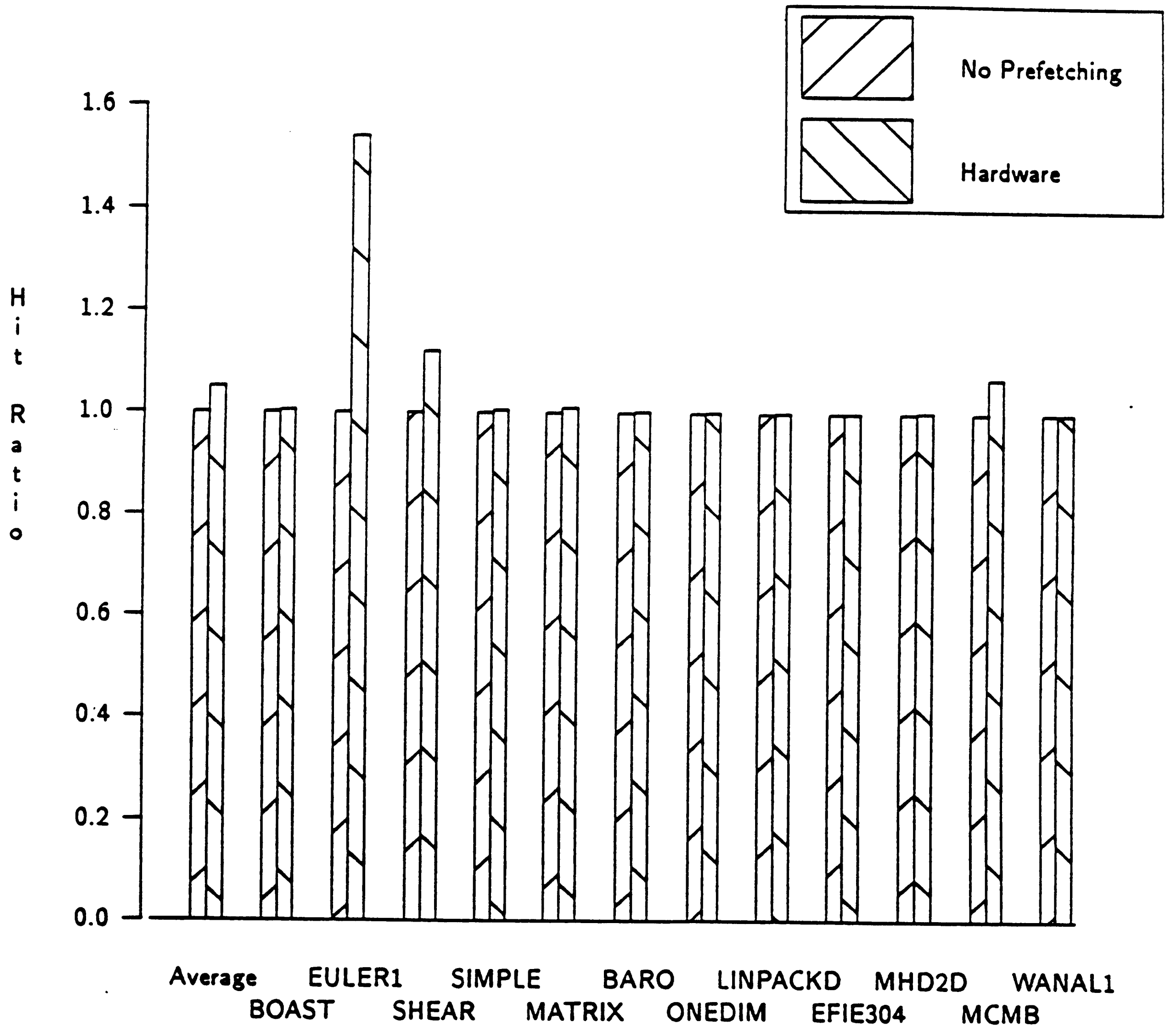

Figure 2.12 Effects of Hardware Prefetching - Normalized Data Traffic

2.4.5 Hit Ratios of Individual References

In Section 2.4.1, the performance of MATRIX was examined in some detail. Three of the references were almost always hits (> 99%), one reference was always a miss and the last reference was a miss when the cache had smaller than 40K. Except for a very small range of cache sizes approximately 40K, every reference in MATRIX could be viewed as either a hit or a miss. If references generally demonstrate this behavior, then a compiler can estimate memory performance by estimating whether each reference will be a hit. Besides estimating current performance, this information can be used to improve the performance. The only references whose hit ratio can be improved are those that generate misses. On average, only considering the references that miss can reduce the number of references that must be considered by 70%.

The cache hit ratio reported by PFC-Sim for complete programs is actually the total of the hits and misses counted for each individual reference in the program. By looking at the hit ratios of individual references (Figure 2.13), references are divided into two groups (hits and misses).

Figure 2.13 graphs the individual hit ratio for every reference in the benchmark programs for a 32K LRU 4-way set associative cache. It clearly demonstrates two preferred ratios. Over 94% of the references had individual hit ratios above 95% or below 5%. No other region had even 1% of the references.

The approximately one-third of the references that miss over 95% of the time account for over 96% of the misses in all of the RiCEPS programs, (see Figure 2.14). Unfortunately, this figure is dominated by the few programs with the most references and therefore the most misses. When individual programs are examined, 9 of the 12 have their misses dominated by references that almost never hit.

The three programs that do not have their misses dominated by references that never hit are LINPACKD, EULER1 and BOAST. LINPACKD's most frequently accessed code occurs in a triangular loop nest (a pair of loops where the induction variable of the outer loop is used as a bound of the inner loop iteration space) using more than 32K of data for the first iterations and less data for the later iterations. If the working set size of this loop nest were computed as if the loop was not triangular, then the references would be marked as never hitting. This would incorrectly calculate the overall hit ratio, but it would identify the references that cause the majority of the cache misses. EULER1 and BOAST both have a small number of

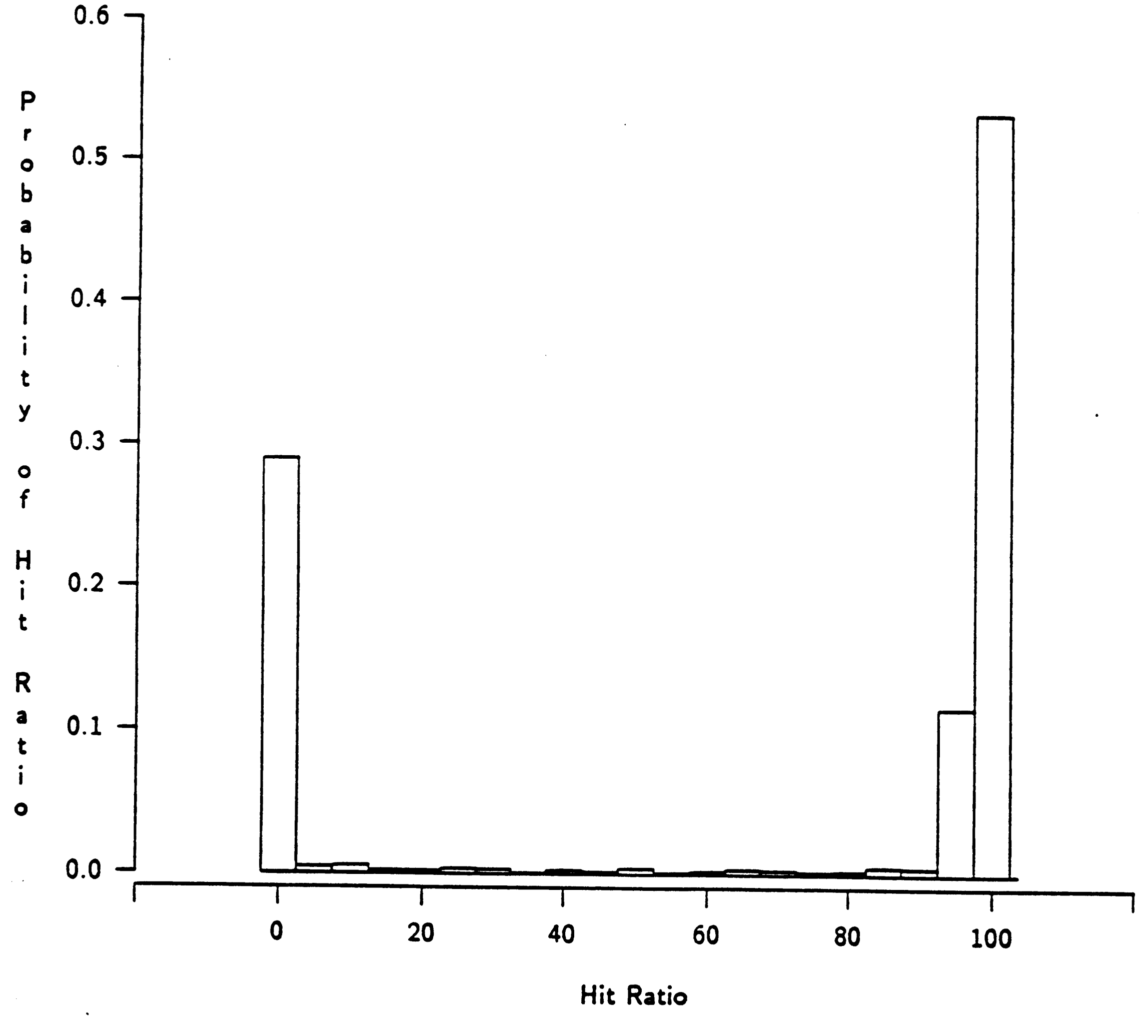

Figure 2.13 Individual Hit Ratio Distribution

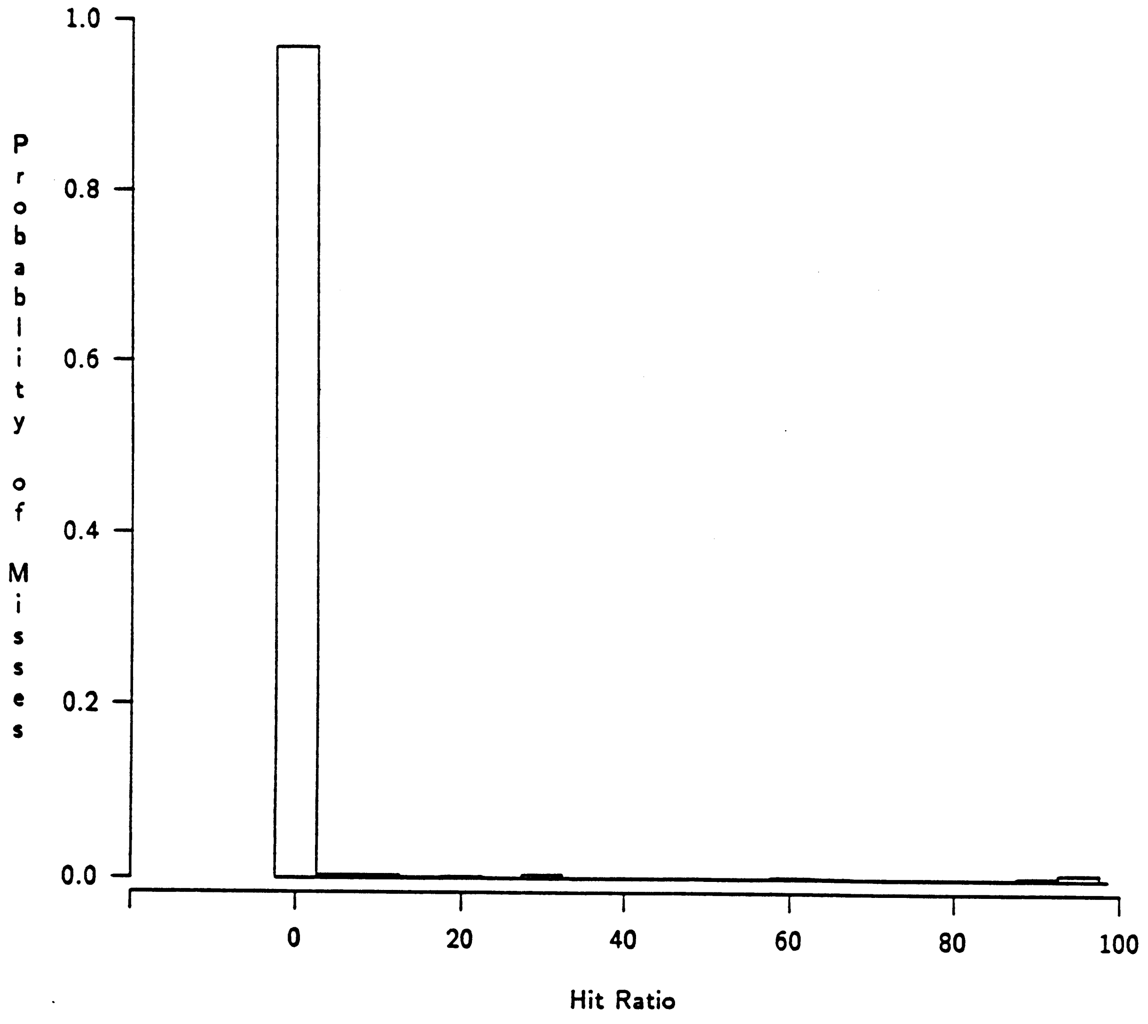

Figure 2.14 Hit Ratios Weighted by Number of Misses

references that miss (about 10%) because of their very high hit ratios (above 98%). EULER1's misses go down with larger set sizes. A large number of small to medium size arrays are being accessed in the loop containing the misses. Arrays that map to the same blocks are the most likely cause of the misses for these references. BOAST may also be affected by arrays mapping onto the same blocks, but increasing the set size to 8 (from 4) does not reduce the number of misses on each reference as it did for EULER1.

The bimodal nature of individual references' hit ratios and the overwhelming percentage of the misses that occur for references that miss on almost all references suggest that, for each program reference, the minimum working set size required to prevent misses is constant between references during execution. If the compiler can make an estimate of the working set size required for each variable, and knows the cache size, then the program references that degrade overall machine performance can be identified and improved. Chapter 3 attempts to compute the working set sizes and Chapter 4 investigates program transformations to improve the match between working set size and the cache size.

2.4.6 Processor Performance

Previous sections have measured the cache performance in absolute numbers. The effectiveness of hiding the CPU from the main memory and the communications costs involved in that hiding were given. More important is the overall impact of the cache on processor performance. The overall performance can be broken into two categories: delays from cache misses and peak performance as dictated by memory bandwidth considerations.

For the test programs, reducing the cache misses can significantly improve performance. As supercomputers have become faster in recent years the speed differences between caches and main memories has increased. Cray now has computers (early versions of the Cray-2 [SW88]) in which accessing the main memory requires 57 cycles. When the delays to memory reach this level, reducing the misses can improve performance by 20 to 30%. In this environment, improving cache performance can be a very profitable optimization.

Miss Delays

PFC-Sim generates profile data counting the number of times that every basic block is executed. Since a basic block has no control flow, it is possible to estimate the execution time (ignoring any memory delays) of each block. An estimate of the total execution time, in cycles, is easily generated during execution. The total execution time for a program would be this estimate plus any delays caused by memory misses.

The estimates of the execution time of a basic block may be parameterized to match many machines. PFC-Sim modifies the code directly after the parser, so no optimizations have occurred to reduce the execution time of basic blocks. In this dissertation, we have assumed that loads, stores, and integer addition instructions, all take 1 cycle, floating point addition takes 10 cycles, and a floating point multiply takes 20 cycles.

Using the estimate for total execution time and the number of misses in each program, the amount of time spent waiting on cache misses can be computed for various distances to main memory. Figure 2.15 graphs the cost of memory performance for each program with a 32K, LRU, 4-byte line, write-back, no prefetching cache. Even when a cache miss takes 50 cycles to return, programs with high hit ratios (BOAST, EULER1) spend less that 1.5% of execution time waiting on the main memory. Any effort to improve the cache performance of these programs will have a minimal (or counterproductive) effect.

Most of the programs fall into a second category. They spend between 2 and 5% of the time waiting on the cache when the main memory takes 5 cycles to return a value. Thus, effort spent on improving cache performance will have little effect on execution time when the memory requires very few cycles to access. As the number of cycles to memory increases, optimization of cache performance could bring noticeable benefits. When the cache is 20 cycles away, elimination of cache delays would result in 7 to 16% faster programs. If main memory is distant (50 cycles), the improvement can be as much as 32%. For this group of programs, attempts to remove cache delays can significantly improve performance when a substantial delay occurs every time main memory is accessed.

One program tested, WANAL1, did not fit into the two groups. WANAL1's cache performance was substantially poorer than any other program tested. Even when main memory was only 5 cycles away, WANAL1 would execute 9.1% faster with no

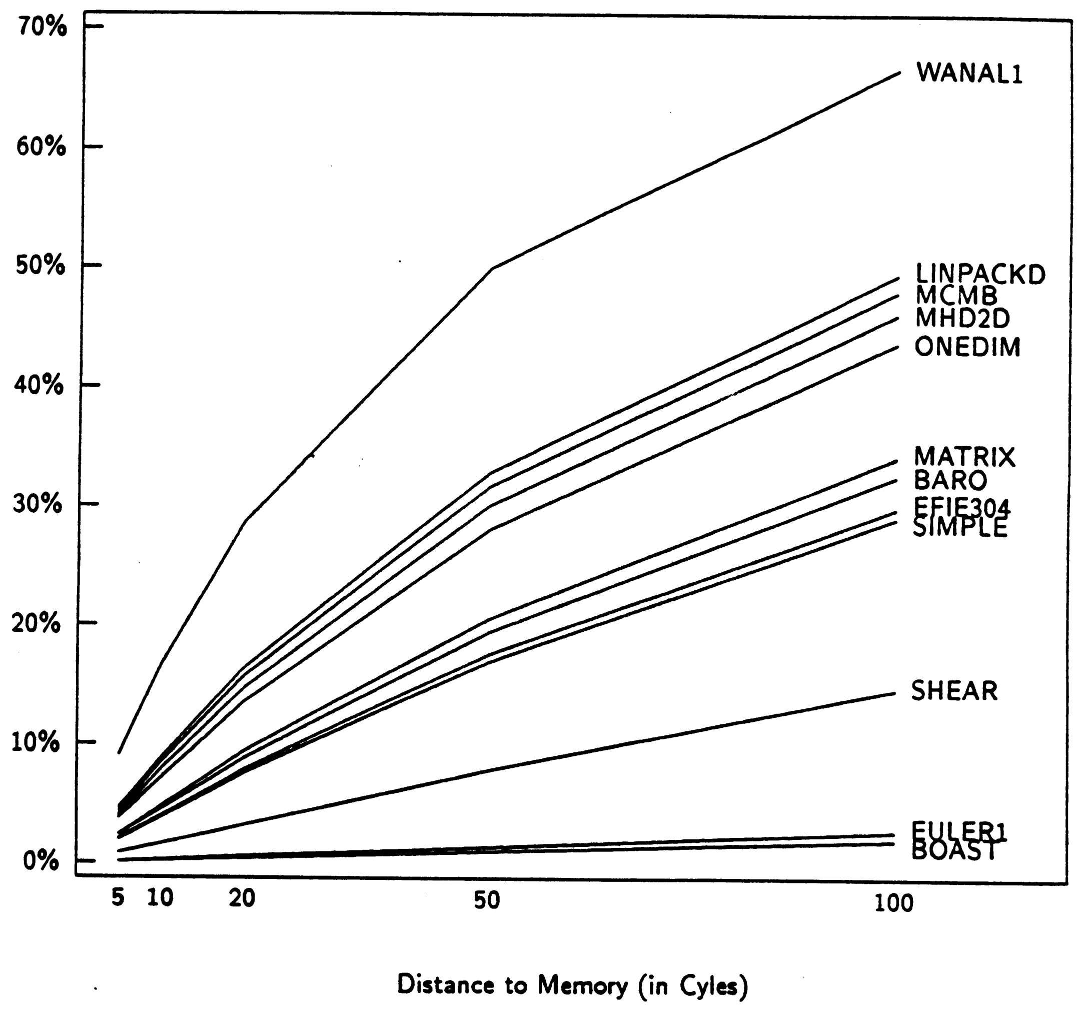

Figure 2.15 Percentage of Execution Time Spent in Miss Delays

cache misses. When main memory was 50 cycles away, half of the execution was spent waiting for memory references.

Data Traffic Requirements

The memory hierarchy can affect execution time of a program, more than the time spent waiting during misses. The hardware bandwidth places an upper bound on the amount of memory that can be moved between memory levels. To prevent saturation of the memory bandwidth, the computer architect must provide enough bandwidth to supply the processing elements. For supercomputers, this can be viewed as providing enough bandwidth so that the floating point units are kept busy.

A second version of PFC-Sim counts the number of floating point operations in a program. Seven of the twelve programs used in this chapter have been studied. (The remaining five uncovered bugs in PFC, that were not present in the parts used for cache simulation.) The amount of data cache traffic generated by these programs for various caches per floating point operation should estimate the maximum sustainable FLOPS for each program on a given architecture.

Table 2.1 shows that an average program running on an architecture with a 32K cache with 4-byte cache lines moved 4.54 bytes from the memory to the cache for every floating point instruction executed. This agrees with a design heuristic that one word of main memory bandwidth per second is required for each floating point operation to be executed per second. On average, an architecture with wide cache lines (64-byte) required over twice as much data traffic (9.68 bytes) as one with shorter cache lines. A write-through cache raised the average required traffic to 8.88 bytes. Thus, unless low-traffic caches are used, the proper estimate for traffic per floating point operation is 2-4 words per floating point operation.

Programs for which long cache lines are not an effective prefetch mechanism require a staggering amount of bandwidth for each floating point operation. SHEAR on a cache with 64-byte lines required almost 28 words per floating point operation. Even programs with data cache hit ratios above 90% (SHEAR - 92.6) can generate data cache traffic per FLOP that will saturate many memory systems. In contrast, WANAL1 had a hit ratio of 41%, but required fewer than 10 words per floating point operation for every tested cache.

RiCEPS Program	Floating Point Operations	Bytes per FLOP (base cache)	Bytes per FLOP (64-byte line)	Bytes per FLOP (write-through)
SIMPLE	509,677,807	3.00	7.77	5.04
EFIE304	3,825,955	4.40	11.57	6.54
WANAL1	1,495,211,989	8.98	9.24	11.71
BARO	69,851,591	2.16	2.21	3.12
LINPACKD	18,558,094	5.80	6.73	10.00
SHEAR	152,106,722	5.56	27.98	21.88
MATRIX	2,000,000	1.88	2.34	3.90
Average	321,604,594	4.54	9.68	8.88

Table 2.1 Data Traffic per Floating Point Operation

These numbers are evidence that hit ratios are poor absolute measures of memory performance. However, hit ratios have been used throughout this work because, at an individual reference level, the hit ratio can still identify potential locations for improvement. Individual references can be divided into two categories according to hit ratio; the vast percentage of the misses occur for those references with poor hit ratios.

Although a high cache hit ratio does not imply that memory bandwidth saturation is avoided, a particular reference with a high hit ratio is not likely to cause saturation. The remainder of this dissertation attempts to find and improve references that could cause memory access problems.

2.5 Summary

This chapter has presented an effective mechanism to study the performance of whole programs. PFC-Sim provides information about how various data caches perform. A number of interesting observations have been made about the performance of the computationally intensive programs in RiCEPS. As motivation for the remainder of this dissertation, we to reemphasize these points.

Eliminating cache misses for supercomputers with distant memories (over 20 cycles away) would be a very effective optimization. The execution time of the tested programs could be reduced by up to 50%, if the cache misses were eliminated.

Two mechanisms for prefetching data have been suggested. Long cache lines are very effective at reducing the number of cache misses in a program. However, the longer lines generate substantially more data traffic between the cache and the main memory. Long cache lines may not be suitable for multiprocessors because cache line sharing between processors increases cache coherence problems. Other mechanisms are needed. Hardware prefetching dramatically improves some programs' memory performance with very little additional data traffic. Unfortunately, hardware prefetching does not improve programs that do not sequentially access arrays. Other prefetching methods are needed.

One attractive solution to the problem of improving memory performance is to transform the program during compilation. This approach is attractive for several reasons. Since the dynamic behavior of source program references tends to be stable (i.e., either a reference is a hit or a miss), the compiler has a reasonable chance of identifying those references responsible for degraded performance. After the references and their dependences are found, a compiler can rearrange the program. By changing the length of dependences, it may be possible to increase the percentage of references that are cache hits during execution. Rearranging the program to reduce misses has an advantage over the prefetching methods of reducing cache misses. When a prefetching mechanism hides a miss delay, the required data traffic stays the same or increases. A cache miss removed by code transformation will require less traffic since the number of times a value is loaded into the cache is decreased.

Chapter 3

Static Model of Memory Performance

3.1 Introduction

The data cache performance study showed that, for a significant number of computationally intensive programs, performance benefits can be obtained by reducing the number of cache misses or overlapping memory latencies with computation. The results further indicate that the cache misses are localized in a relatively small number of references in the program source. By developing tools for the compiler to locate and eliminate the cache misses, faster programs could be produced. In this chapter, we derive a method for the compiler to locate references that are likely to be misses. Later chapters focus on methods to reduce the number of misses and to overlap execution with remaining memory delays.

It is impossible to determine exactly which references are going to produce misses during execution. Two different sets of input data to the same program can easily produce misses at different references. Even if the data set is known, Bernstein proved that determining whether arrays accessed by arbitrary functions access the same memory location is undecidable [Ber66]. If it is impossible to determine if the second reference uses the same memory location as the first, it is certainly impossible to determine if the second location will be found in the cache.

Although one cannot precisely identify every reference that will produce a miss during execution, a close approximation will suffice. If a reference is incorrectly identified as a miss when it is actually a hit, any attempt to move it closer to previous references will not change the result of the program. Only compiler effort is wasted. If a reference is identified as a hit when it is a miss, the compiler will not attempt to improve the reference, leaving the original source potentially sub-optimal but still correct. The estimate of whether a reference is a hit or a miss should also be reasonably fast to generate and easy to maintain during transformations that occur in optimizing compilers. If the estimate is slow to generate or must be regenerated for the entire program every time a transformation is considered, then the cost of locating the references that miss will exceed the acceptable cost in compile time.

By easing the conservative nature of data dependences, a simple concept, the **Overflow Iteration**, can be defined to help determine when a reference will be a miss during execution.

3.2 Overflow Iteration

A dependence edge corresponds to an execution path between two references that access the same memory location. Every time through that execution path, the same statements will be executed. It seems reasonable to assume that each execution causes approximately the same number of different memory blocks to be accessed between the endpoints of a dependence. Since the number of different memory blocks accessed between two references determines whether an access is found in cache, an edge will normally correspond to either a series of hits or a series of misses during execution. This is exactly the behavior that was noted in Section 2.4.5. References were found to be either hits or misses, depending on the size of the cache and the working set size.

To define whether an edge represents a hit or a miss, the number of memory blocks accessed between the endpoints is determined. Dependences are defined in terms of loops (the carrying loop and the distance), so a measure to determine how much memory was touched between endpoints should be defined on the same basis. The **Overflow Iteration**, $O(i)$, for a particular loop is the maximum number of iterations of that loop that can have all of the data accessed by the loop maintained in the memory at the same time. Any edge that requires more iterations of the loop than the overflow iteration will access more different blocks than available and result in a series of misses during execution. Once the overflow iteration is known, we can decide whether a given dependence edge will be a hit and use this knowledge to estimate the hit ratio for every reference.

To compute the overflow iteration, we determine which iteration of the loop will access the M+1^{mboxst} memory block when there are only M available memory blocks. By using the dependence graph with only *must* (uniformly generated) edges, estimating this iteration is possible.

Computing the Overflow Iteration

The first iteration of a loop requires some fixed number of memory blocks to be accessed. Each succeeding iteration uses some of those blocks and some additional memory blocks. We first define $F(i)$ to be the number of memory blocks used by the first iteration of the i loop and $\Delta(i)$ to be the amount of new memory used by a succeeding iteration. The overflow iteration can be defined as

$$O(i) = \begin{cases} (M - F(i))/\Delta(i) + 1 & \text{if } M \geq F(i) \\ 0 & \text{otherwise} \end{cases}$$

where $\Delta(i)$ is constant and M is the number of available memory blocks. The overflow iteration is calculated by determining whether the first iteration fits into the available memory. When it fits, the remaining memory is divided by the amount required for an additional iteration. By adding one iteration to the result of the division, the maximum number of iterations which fit into the available memory is found.

One common coding practice that does not produce a constant $\Delta(i)$ is accessing a triangular portion of an array. Rather than compute $\Delta(i)$ functions that vary with the iteration number, the maximum amount of new memory accessed by any iteration is used as $\Delta(i)$. This underestimates the overflow iteration. The hit or miss prediction for edges carried by outer loops is based on the largest triangular loop. For programs such as LINPACKD, the largest triangular sections do not fit in a 32K cache, so edges carried by outer loops will be marked as misses. For LINPACKD marking the references as misses even though they have approximately 50% hit ratios allows almost every miss in the program to be included in the references that are marked misses.

When computing $F(i)$ and $\Delta(i)$, the dependence graph is trimmed of all non-uniformly generated dependences. With this smaller graph, dependences are used to define when a value will be used for the first time during a loop. The memory used in the first iteration of a loop is the sum of the memory used during the first execution of every reference in the loop, or

$$F(i) = \sum_{A=1}^{\#\text{refs}} f_i(A)$$

where $f_i(A)$ is the amount of memory used by reference A during the first execution of the i^{th} loop.

Similarly, the amount of additional memory used by a later iteration can be defined in terms of $\delta_i(A)$, the additional memory used by reference A during a subsequent iteration of the i^{th} loop.

$$\Delta(i) = \sum_{A=1}^{\#\text{refs}} \delta_i(A)$$

Using the dependence graph, $f_i(A)$ can be estimated. When a reference is a sink of a loop independent dependence whose source is contained in loop i, then that reference will not require any new memory locations to be accessed during its execution. An edge satisfies these conditions if it has $D = 0$ (loop independent) and $LC \geq i$ (carried within this loop). If these conditions are not met, then this reference will be the first to access some memory location or locations in this loop. If the array access is not nested within other loops, it will execute only once, accessing at most one memory location. When no dependences exist, every execution of the reference will access a new location. The number of times a reference is executed is the number of iterations of loops nested within the loop presently being examined. If there exists a loop carried dependence on one of the inner loops, then that loop will not access new data after the first $D(e)$ iterations through that loop. Combining these conditions yields

$$f_i(A) = \begin{cases} 0 & \text{if } \exists \text{ edge } e \text{ s.t. } LC(e) = i,\ D(e) = 0 \\ \prod_{x=i+1}^{\text{maxloopnest}} LB_x & \text{otherwise} \end{cases}$$

where

$$LB_x = \begin{cases} D(e) & \text{if } \exists \text{ edge } e \text{ s.t. } LC(e) = x, D(e) \geq 0 \\ \text{Loop bound of loop } x & \text{otherwise} \end{cases}$$

Note that the check for inner loop independent dependences is performed inside the LB_x calculation.

In many cases, LB_x will not be computable during compilation because some part of the loop bound is symbolic. This mechanism can generate symbolic values for f and δ. As seen in later chapters, some tuning of blocking transformations can be delayed until the symbolic values are known during execution. In other situations, maximum values for the symbolic terms can be determined from the declared array bounds or from information supplied by the programmer.

Computing δ is straightforward after f is known. If a reference in the loop has a loop carried dependence, then it will reuse a value from a previous iteration, otherwise it will need to bring in new values.

$$\delta_i(A) = \begin{cases} 0 & \text{if } \exists \text{ edge } e \text{ s.t. } LC(e) = i, D(e) > 0 \\ f_i(A) & \text{otherwise} \end{cases}$$

These equations provide a means to generate overflow iterations for every loop from a uniformly generated dependence graph, as shown in Figure 3.1. Computing the overflow iteration for every loop in a program will take $O($*maximum loop nesting depth* $*$ *number of dependences in the loop*$)$ time. The innermost loop of the algorithm is building a list of every dependence that has a particular statement as its sink that is carried by the present level. If all of the dependences that have the same node for a sink are linked together (as is done in PFC), then building the list requires every dependence in the graph to be examined at most once for each loop in the deepest loop nest in the program. In general, the number of loops in a loop nest is assumed to be a small constant (5-10), and the number of dependences is a constant times the number of references in a program. The time complexity will normally be $O(\mathit{references})$, although the worst case is $O(\mathit{references}^3)$, where *references* is the number of references in the program.

Using the overflow iteration to estimate the hit ratio of an individual reference only requires a method for estimating the frequency of each of the incoming dependence edges. Several simple methods can be used for these estimates. The simplest method is to determine if the dependence carried by the innermost loop is a hit or miss. If the reference is a miss, then the number of program misses can be incremented by the product of the iteration counts of the surrounding loops. This method ignores all edges except those on the innermost loop, since the outer loops are executed substantially less frequently. This again corresponds to the observed behavior of individual references.

A second, more accurate method is shown in Figure 3.2. Rather than assuming that the reference hit pattern is completely specified by the innermost dependence to the reference, the misses caused by every level are calculated. The misses caused by any level can be estimated using the dependences carried by the level and the iteration count of the appropriate loop level. If no dependences exist or if they are misses, then every iteration of the loop will cause the same number of misses as the

```
forall references A
    for level = nesting of A to 1 by -1

        /* initialize the amount used by the first iteration to 1 if */
        /* innermost loop; otherwise, set to the amount used by all */
        /* iterations of the next inner loop */
        if nesting (A) = level
            then $f_{level}(A) = 1$
            else $f_{level}(A) = LB(level + 1) * f_{level+1}(A)$

        /* find all edges into this reference that are carried by this level */
        get list l of edges such that $LC(e) = level, SINK(e) = A$
        LB(level) = loopbound of level

        /* if any edges exist, then iterations after the first do not */
        /* require more space in the memory */
        if l is empty
            then $\delta_{level}(A) = f_{level}(A)$
            else $\delta_{level}(A) = 0$
        /* check to see if any of the edges are loop independent — if so, */
        /* the memory used by this reference was previously used by the */
        /* source and this statement uses no additional memory */
        for every edge in l
            if $D(e) = 0$
                then $f_{level}(A) = 0$
            /* LB computes the number of iterations that require memory */
            /* access for this loop level — 0 if sink of loop independent */
            /* edge, minimum distance if loop carried edges exist and the */
            /* number of iterations if no dependence edges */
            $LB(level) = min(LB(level), D(e))$
        end
    end
end

/* with $f$ and $\delta$, $F$, $\Delta$ and $O$ are easily computed */
forall loops, l, in the program
    $F(l) = \sum f_l(A)$
    $\Delta(l) = \sum \delta_l(A)$
    if $MS > F(l)$
        then $O(l) = ((MS - F(l))/\Delta(l)) + 1$
        else $O(l) = 0$
end
```

Figure 3.1 Computing the Overflow Iteration, O

```
procedure compute_misses(DoLoop)

    for every reference A in the program

        /*compute the misses arising from this statement */
        miss = 1
        for lev = nest(A) to 1 by -1
            get list, l, of edges such that LC(e) = lev and SINK(e) = A
            if l not empty

                then find d such that d = min(D(e) on l)
                if d < O(lev)
                    /* edges are hits and misses only occur on iterations */
                    /* that occur before the dependences occur */
                    then miss = miss * d
                    /* edges are misses and every iteration must counted */
                    else miss = miss * iteration count of loop lev

            else /* no dependence */
                /* every iteration uses new memory location */
                miss = miss * iteration count of loop lev
        end
    end
end compute_misses
```

Figure 3.2 Computing the Misses in a Code Segment

first iteration. If a loop independent dependence exists (and is a hit), then no access to this variable will be a miss. If only loop carried dependences exist (and are hits), then only the loop iterations that occur before reuse of memory is required cause misses. Memory reuse will start as soon as the number of iterations executed equals the minimum distance of the dependences. This method distinguishes between the references with hit ratios of 100% and those in the 95 to 100% category.

This method considers all dependences carried by a single loop level to be hits or misses. Further improvement may be obtained by examining the dependences carried by a single loop level individually and adding their contribution to the total misses. This method requires more computation in the inner loop of the algorithm, and experience has shown that either all of the dependences carried by a level are misses or they are all hits. Further refinement of the method to compute misses shown in Figure 3.2 will produce little, if any, profit.

Consider what occurs in the computation of Δ and misses when the iteration count of the loops is not known because the loop control variables contain symbolics. This can occur when the size of an array is a run-time constant. Both algorithms would need some mechanism to estimate the number of iterations in the loops with symbolic bounds. This can be done by either looking at the declared bounds of the array or by choosing an arbitrary value.

However, by modifying the algorithms to do symbolic arithmetic, the actual computation of the overflow iteration can be delayed until execution, when the symbolics are known. When the overflow iteration is not exactly computed, it is still possible to perform many of the transformations to improve performance. For some transformations (the blocking transformations, in particular), computation of the optimal parameters for the transformation can be delayed. For other transformations, it is possible to determine that the transformation is profitable for a range of symbolic values. For instance, if fusion is profitable when the loop iterates less than 20,000 times and array bounds are on the order of 100, then performing the fusion will almost certainly improve memory performance.

Example

Cache performance of matrix multiplication, MATRIX (illustrated below), had some interesting data performance results which can be predicted using the overflow iteration.

```
(1) DO I = 1, 100
(2)     DO J = 1, 100
(3)         A(I,J) = 0
(4)         DO K = 1, 100
(5)             A(I,J) = A(I,J) + B(I,K)*C(K,J)
(6)         ENDDO
(7)     ENDDO
(8) ENDDO
```

The first step is to compute F and Δ for each loop, numbering the references (definition of A on line 3 - #1; definition of A on line 5 - #2; use of A on line 5 - #3; definition of B on line 5 - #4; and definition of C on line 5 - #5).
The dependences for matrix multiply are

Dependence Number	Variable	From	To	Type	LC	D
1	A(I,J)	1	3	True	J	0
2	A(I,J)	1	2	Output	J	0
3	A(I,J)	3	2	Anti	K	0
4	A(I,J)	2	3	True	K	1
5	A(I,J)	2	2	Output	K	1
6	A(I,J)	3	3	Input	K	1
7	B(I,K)	4	4	Input	J	1
8	C(K,J)	5	5	Input	I	1

The amount of storage used by each reference on the 1st iteration of the innermost loop is 1.

$f_K(1) = 0$, since the reference is not in the K loop.
$f_K(2) = 0$, sink of dependence 3, $LC(3) =$K and $D(3) = 0$.
$f_K(3) = 1$, sink of no dependence edge, e, with $LC(e) =$ K and $D(e) = 0$.
$f_K(4) = 1$, same as $f_K(3)$.

$f_K(5) = 1$, same as $f_K(3)$.

The total storage for the first iteration loop is the sum of the individual references.

$F(K) = \sum_{x=1}^{5} f_K(x) = 3$.

The amount of additional storage used by each reference on subsequent iterations is:

$\delta_k(1) = 0$, not in loop k.
$\delta_k(2) = 0$, equal to $f_K(2)$.
$\delta_k(3) = 0$, sink of dependences 4 and 6 LC =K and $D = 1$.
$\delta_k(4) = 1$, equal to $f_K(4)$, since it is not a sink of a LC = K dependence.
$\delta_k(5) = 1$, equal to $f_K(5)$, since it is not a sink of a LC = K dependence.

Again, summation produces the total for the entire loop $\Delta(K) = \sum_{x=1}^{5} \delta_K(x) = 2$.
Using the same methods on the middle or J loop, we get

$f_J(1) = 1$, no level LC = J and $D = 0$ dependence edge.
$f_J(2) = 0$, sink of dependence 2, $LC(2)$ =J and $D(2) = 0$.
$f_J(3) = 0$, sink of dependence 1, $LC(1)$ =J and $D(1) = 0$.
$f_J(4) = 100$, LB_K since it is the sink of no LC = K, $D > 0$ dependence.
$f_J(5) = 100$. LB_K since it is the sink of no LC = K, $D > 0$ dependence.

Summing yields $F(J) = \sum_{x=1}^{5} f_J(x) = 201$.
To verify these numbers: $F(K) + 99 * \Delta(K) = 201 = F(J)$. The total used in the inner loop agrees with the total used by the first iteration of the middle loop; exactly as one would expect for matrix multiply.

$\delta_k(1) = 1$, equal to $f_K(1)$.
$\delta_k(2) = 0$, equal to $f_K(2)$.
$\delta_k(3) = 0$, equal to $f_K(3)$.
$\delta_k(4) = 0$, sink of dependence 7, $LC(7)$ = J and $D(7) = 1$.
$\delta_k(5) = 100$, equal to $f_K(5)$.

So $\Delta(J) = \sum_{x=1}^{5} \delta_J(x) = 101$.
Continuing for the next outer loop, generates $F(I) = 10200$ and $\Delta(I) = 200$. Finally, $F(program) = 30000$.

Loop	$F(x)$	$\Delta(x)$	Overflow Iteration (32K memory)	Overflow Iteration (64K memory)
K	3	2	> 100	> 100
J	201	101	80	> 100
I	10200	200	0	31
program	30000	0	0	0

Table 3.1 Matrix Multiply (100 x 100) — F, Δ, and O

Dependence	LC	D	Overflow Iteration (32K cache)	Hit or Miss (32K cache)	Overflow Iteration (64K cache)	Hit or Miss (64K cache)
1	J	0	80	Hit	> 100	Hit
2	J	0	80	Hit	> 100	Hit
3	K	0	> 100	Hit	> 100	Hit
4	K	1	> 100	Hit	> 100	Hit
5	K	1	> 100	Hit	> 100	Hit
6	K	1	> 100	Hit	> 100	Hit
7	J	1	80	Hit	> 100	Hit
8	I	1	0	Miss	31	Hit

Table 3.2 Matrix Multiply — Computed Cache Performance of Dependences

Using these values for F and Δ, $O(i)$ is computed for various memory sizes as in Table 3.1. Looking at the dependence graph, we determine whether an edge will cross more data blocks than are available in the various size caches (see Table 3.2).

From this table, it is obvious that a 64K cache will produce a very high hit ratio. Every edge is a hit. A 32K cache has one dependence that will result in dynamic misses. Since this is the only dependence that references the C array (#5), reference #5 has a predicted hit ratio of 0% for a 32K cache while the rest of the references are predicted to be hits. Table 3.3 shows the actual hit ratios for this program as generated by PFC-Sim. The only difference between the two cache sizes is the misses on reference #4, the B array. If dependence edges are added from the entry point to all unexposed upward uses of variables and are considered misses, then *miss_count* predicts the only reference to the B array (#4) as having a hit ratio of 99%. This exactly matches the measured value.

The overflow iteration does an excellent job of predicting the performance of a small regular numerical program such as matrix multiplication. Its performance on large numerical programs such as those in RiCEPS must still be determined. Complete testing of the overflow iteration can not be done without an implementation. Performing the equations by hand on several of the most frequently executed portions of the RiCEPS programs demonstrated two traits of the method. With the proper information, the references that were predicted to be misses were the same references found to be misses by PFC-Sim. Loss of information at call sites with standard interprocedural information would produce very poor estimates. For overflow iterations to be effectively generated, precise interprocedural information should be

Reference	Computed Hit Ratio	Actual Hit Ratio	Actual Hits	Actual Misses
1	0	0	0	10000
2	100	100	1000000	0
3	100	100	1000000	0
4	100	99	990000	10000
5	0	0	0	1000000

Table 3.3 Matrix Multiply — Computed and Actual Hit Ratios (32K cache)

available. At the present time, restricted regular sections [CK88],[Cal87] with bounds information is being implemented. When completed this should provide information about the precision required to compute an accurate overflow iteration for complex programs.

3.3 Conclusions

The overflow iteration provides a method to determine whether a dependence will be a hit or a miss during the execution of a program. Each reference can have its hit ratio predicted by looking at the dependences of the reference. The effectiveness of the overflow iteration depends on three factors: the accuracy of the overflow iteration in locating misses, the time it takes to compute during compilation and the complexity of its implementation.

On the simple numerical example MATRIX, the overflow iteration is an accurate tool. When the inner loops of programs in RiCEPS were studied by hand, the overflow iteration was accurate whenever no information was lost at call site boundaries. Without an implementation, the accuracy of the overflow iteration is an estimate, but examples computed by hand are encouraging. On triangular loops, such as those that exist in LINPACKD, the references that cause most of the misses (although they actually have about a 50% hit ratio) are marked as misses, allowing techniques in the later chapters of this dissertation to be applied to improve performance.

Computing the overflow iteration should not require a significant amount of compile time. Only a small number of arithmetic operations need to occur for each uniformly generated dependence in the program. The only major cost will be storage to maintain f and δ for every reference. This memory cost could easily be 80 bytes per array reference in the program. In programming environments, this will be a noticeable cost but not the largest memory consumer. In a standard compiler, f and δ should only need to be available for the procedure being compiled. This would limit the total memory required to keep the values to less that 100 Kbytes.

The last requirement for practical use of the overflow iteration is a reasonable implementation. The implementation within PFC, although not complete, seems straightforward. The algorithms are not complicated, if the information that they use is available. PFC can already compute input dependences. It also marks dependences found by the separability test as consistent. Using these as the uniformly generated dependences, all of the information needed to compute $O(i)$ is available. The accuracy of the overflow iteration depends heavily on the accuracy of interprocedural analysis. At the present time, regular section interprocedural analysis is being implemented for PFC. The array sections provided by regular sections should give enough interprocedural accuracy for good overflow iteration estimates.

Chapter 4

Program Transformation

4.1 Introduction

This chapter discusses the use of reordering transformations to maximize the number of dependences with distances less than the overflow iteration of the loop that carries them. If the program can be transformed into a version that makes better use of the cache, then the number of requests to memory can be reduced, improving the execution time. Over the years, researchers have developed optimizing compilers that apply a number of transformations reordering computation without changing the results. A transformation that reduces the amount of memory used between the endpoints of a dependence edge may change a reference from a miss to a hit. Many of the transformations examined in this chapter are already used during vectorization or parallelization of programs in PFC. The remainder could be easily added. These transformations could be added to other parallelizing compilers, when required.

The transformations examined in this work are *loop fusion*, *loop distribution*, *loop interchange*, *strip mine*, *unroll*, two combinations of transformations *peel and jam*, *loop skew and interchange* and three blocking transformations (*strip mine and interchange*, *unroll and jam*, and *wavefront blocking*). All of these transformations are loop based transformations, and can potentially change the order of execution of many statements. Transformations are safe when no dependence is reversed by the transformation [All83], [War84]. A dependence is said to be reversed by transformation when the source and sink of the dependence edge are interchanged. When determining the safety of a transformation, true, anti and output dependence edges, including those that are not uniformly generated must be considered. For each transformation, we specify a mechanism to determine the transformation's effect on memory performance. Since these transformations can change the dependence graph, the quantities f and δ used in determining the overflow iteration may need to be recomputed. For each transformation, we specify a set of conditions that describe when a reference can be a miss before transformation and a hit after transformation. Similar conditions are stated for references that are hits before transformation and become misses after transformation. With a single pass through the statements involved in the transformation, the potential benefit of the transformation can be determined.

4.2 Loop Fusion

A simple template for loop fusion is given in Figure 4.1. During fusion, the loop induction variables, *I* and *J* for these two loops, are replaced with *New*, the loop induction variable for the combined loop. The two loop bodies are merged, retaining their original lexical order. However, fusion intermixes the iterations of the loop bodies. Before fusion, all executions of the first loop body occur before any executions of the second loop body. After fusion, each iteration of the first loop is followed by an iteration of the second loop. Loop fusion is safe when there are no loop independent edges from the first loop body to the second loop body that become loop carried dependences after fusion [Wol82].

Loop fusion is an important transformation in a parallel code generator. To find parallel sections of a program, PFC divides a source program into as many regions as possible, because it is usually easier to prove that a smaller region can run correctly in parallel. Parallel startup costs are high for many processors. Thus, after determining which regions can run in parallel, PFC tries to fuse the regions back together, reducing the number of parallel regions and the parallel startup costs during execution.

Eventually, information about memory requirements of the various regions could be used in deciding which regions are best to merge. When several regions could be merged, a good heuristic is to merge the two regions that require the least amount

```
DO I = 1, N
    loop body #1          DO New = 1, N
ENDDO                         loop body #1
DO J = 1, N                   loop body #2
    loop body #2          ENDDO
ENDDO
```

Figure 4.1 Loop Fusion

of memory after the merge. Memory performance information can also be used to determine the maximum parallel width during execution by estimating the amount of memory traffic produced by each processor. Any processor added after the memory is saturated will not decrease execution time. Thus, accurate estimates of memory performance during loop fusion can become a useful tool in obtaining maximum speedup during parallel execution.

Computing the Effect on Memory Performance

The dependence graph used to determine the profitability of a transformation is the same graph used to compute the overflow iterations. It includes all uniformly generated dependences, including input dependences. Since input edges are included, it is possible to have a dependence that reverses direction when calculating the profitability of fusion. Changing the order of two loads will not affect the program result, so reversing input dependences is allowed during fusion.

Fusion does not change which statements are executed by an iteration of an outer loop, only the order in which they are executed. Thus, the statements use the same memory locations. This implies that the memory used by any surrounding loop will not change during fusion. The values of F, Δ, f, δ and O for the surrounding loops remain unchanged after fusion. Likewise, any loop nested within the loops being fused executes exactly as before, with no change to the overflow values, F, Δ, f, δ, and O. Only the overflow values of the new combined loops must be recomputed.

An initial estimate might be

$$F(new) = F(I) + F(J)$$

$$\Delta(new) = \Delta(I) + \Delta(J)$$

$$O(new) = (M - F(new))/\Delta(new) + 1$$

Although this method is fast, it does not take into account changes in the dependence graph caused by fusion.

By identifying dependence edges that are carried by the new fused loop, accurate values of $F(new)$ and $\Delta(new)$ can be determined. All edges carried individually by the I and J loops will be carried by the combined loop. Edges from a statement in the first original loop to a statement in the second original loop may be carried by the new loop. These edges previously were either loop carried by an outer loop or were loop independent involving a statement in the first loop and a statement in the second loop.

A loop carried dependence before fusion indicates that a value used in one outer loop iteration will be used during a later outer loop iteration. The order of execution of the outer loop does not change during fusion, so the dependences carried by the outer loop are not changed. The statements that cause a loop independent edge from the first loop body to the second loop body will now both be contained in the fused loop. The order of execution of these statements may be different in the new loop. The new edge must be recomputed to determine if it is still loop independent, carried from the original source to the original sink, or reversed.

To compute the amount of memory required by the fused loop, the overflow values of the previous loops are first updated to account for the new dependences generated by the loop independent edges.

A statement which is not a sink for one of the new dependences in the fused loop uses exactly the same amount of memory on each iteration as it did before fusion. The value of $f(A)$ in the loop that did not contain the statement before fusion is 0, so $f_{new}(A)$ is equal to the maximum of the two original values. When a statement is the sink of a new loop independent dependence ($D(e) = 0$ and A is the sink), this reference reuses data from an earlier statement, on the first iteration.

If no loop carried dependence edges terminate at reference A, $\delta_{new}(A)$ is equal to $f_{new}(A)$. If no new edges exist with A as their sink, $\delta_{new}(A)$ is equal to $\delta(A)$ of the loop that originally contained the statement (i.e., $\max(\delta_i(A), \delta_j(A))$). If an old loop carried dependence existed before fusion, the old δ will be zero; otherwise, it will be equal to $f(A)$. If a new loop independent edge is created, then $f_{new}(A)$ and δ_{new} are 0. If a new loop carried edge is created, no new memory will be used on later iterations and $\delta_{new}(A)$ becomes 0. Combining the two conditions, $\delta_{new}(A)$ is 0 when any new edge is created.

$$f_{new}(A) = \begin{cases} \max(f_i(A), f_j(A)) & \text{if } \exists \text{ no new edge } e \text{ s.t. } D(e) = 0, \text{SINK}(e) = \text{A} \\ 0 & \text{otherwise} \end{cases}$$

$$\delta_{new}(A) = \begin{cases} \max(\delta_i(A), \delta_j(A)) & \text{if } \exists \text{ no new edge } e \text{ s.t. SINK}(e) = \text{A} \\ 0 & \text{otherwise} \end{cases}$$

Given the new f and δ values, $F(new)$, $\Delta(new)$, and $O(i)$ are calculated a in the normal way

To find the overall effect of fusion on memory performance, references can be divided into three groups: references for which performance is unchanged; references for which performance is improved and references for which performance is degraded. A memory reference has its performance changed when the dependence edges pointing into it cross the overflow iteration.

Fusion can not degrade memory performance if $O(outer) > 0$, or $O(I) = 0$ and $O(J) = 0$ before fusion. If $O(outer) > 0$, then every edge carried by the inner loop is a hit before and after fusion. Also, every original loop independent edge carried by the outer loop must have been a hit. The amount of memory used by the entire loop nest fits into memory, so fusion does not affect whether references are found in memory. None of the affected edges could cross the overflow iteration. If both $O(I)$ and $O(J)$ are equal to zero, then no edge carried by the inner loop is a hit before fusion and no edge carried by the new loop after fusion will be a hit. Again, none of the affected dependence edges can cross the overflow iteration.

The number of hits that become misses due to fusion must be determined. When an edge that corresponded to a hit becomes a miss, the two references must have moved further apart due to fusion. The only references that move apart are those in statements in the same loop body. After fusion, there is an iteration of the other loop body, where the loop bodies were previously consecutive. This causes misses to occur whenever the overflow iteration of one of the original loops is greater than the distance of the edge ($O(IorJ) > D(e)$) and the overflow iteration of the fused loop is smaller than the distance of the edge ($O(new) < D(e)$). Each iteration of the affected loop uses $f_{IorJ}(A)$ memory, so the number of misses is $f_{IorJ}(A)$ times the number of iterations of the fused loop, N (from Figure 4.1).

When two edges have moved closer together upon fusion, an edge that corresponded to a miss can change to a hit. The only references that move closer are those that are now sinks of the new dependences that replaced the loop independent dependences between the separate loop bodies. These references can be identified by checking which references changed $\delta_{new}(A)$ after initialization. If the original loop independent edge was a miss, then $O(outer) = 0$ and the new edge is a hit. If $O(new) > D(e)$, then the references become hits. The number of new hits is the amount of memory used on each iteration of the new loop, $\delta_{I or J}$, times the number of iterations of the new loop, N.

An algorithm to compute the profitability of loop fusion is given in Figure 4.2. The algorithm is divided into five parts: initializing f and δ, computing changes due to changes in loop independent edges, computing F, Δ and O for the fused loop, finding the references that change from misses to hits, and finding the references that change from hits to misses. Most of this work takes constant time for each reference in the program. Computing the changes in the dependence graph involves checking every loop independent edge between the two original loop bodies. Computing a dependence is a possibly large, but constant time operation. There can be as many as $n * m$ edges to be recomputed, where n is the number of references in the first original loop and m is the number of references in the second original loop. Overall, the number of edges that must be checked is $\| D \|$, the size of the set of affected dependences, D. Thus, the algorithm takes O($\| D \|$) time. In practice, the number of dependences does not actually grow quadratically with the size of the program, but will be a (possibly large) constant. Many compilers, including PFC, precompute fusion information during dependence analysis. If the fusion preventing edges are precomputed, then only the second pass through the dependences is required. The second pass that determines if the performance of a reference changes only examines one edge per reference, resulting in a time bound of O(#references).

Example

The example below demonstrates how fusion can improve memory performance when the two loops being fused use the same data, but one execution of either loop flushes the memory.

```
DO I = 1, 10000
    A = A + B(I)*C(I)       DO New = 1, 10000
ENDDO                            A = A + B(New)*C(New)
DO J = 1, 10000                  AA = AA + B(New)+C(New)
    AA = AA + B(J)+C(J)     ENDDO
ENDDO
```

Both loops in the original program use a total of 30,000 words or 120,000 bytes (single precision) of cache. A cache of less than 128K, such as the Alliant FX/8 and the IBM

```
/* Memory used in the new loop is equal to memory used in the old loop */
/* plus memory involved in dependences in the fused loop */
forall statements A in loop I
        $f_{new}(A) = f_i(A)$
end
forall statements A in loop J
        $f_{new}(A) = f_j(A)$
end

/* Correcting for the new dependences */
forall loop independent dependences between loops I and J
        compute new dependence edge e between endpoints
        if $D(e) = 0$ then $f_{new}(SINK(e)) = 0$
        $\delta_{new}(SINK(e)) = 0$
end

$F(new) = \sum f_{new}$
$\Delta(new) = \sum \delta_{new}$
$O(new) = (M - F(new))/\Delta(new) + 1$

/* determine if reference becomes a hit */
profit = 0
forall references A in loop new
        if ( $\delta_{new}(A) \neq \delta_{IorJ}(A)$ /* being the sink of new edge changes $\delta$ */
            & $O(outer) = 0$      /* edge was a miss */
            & $O(new) > D(e)$ /* $O$ greater than the new edge's distance */
          then profit = profit + $N * \delta_{IorJ}(A)$
end

/* determine if reference becomes a miss */
forall references A in loop new
        if ( $f_{IorJ} \neq 0$ and $\delta_{IorJ}(A) = 0$ /* there exists a loop carried dependence */
            & $O(IorJ) \geq D(e)$ /* edge was a hit */
            & $O(new) < D(e)$ /* now a miss */

/* profit is the net change in memory hits from fusion */
          then profit = profit - $N * f_{(IorJ)}(A)$
end
```

Figure 4.2 Profitability of Loop Fusion

3090, would not have either B or C present when accessed during execution of the second loop. After the loops are fused, the values of B and C are used in both loop bodies before being bumped from cache. The original program produced 40,000 misses and no hits. After fusion, there were 20,000 hits and only 20,000 misses. Thus, loop fusion reduced the number of misses in this example by 50%.

4.3 Loop Distribution

Loop distribution (Figure 4.3) is the inverse of loop fusion. Any pair of loops fused together can be split into the original pair of loops by distribution. Loop distribution is used early in PFC to break a loop into smaller sections that are hopefully easier to vectorize and parallelize. The code generator later attempts to fuse these sections back together.

To determine the memory performance of a pair of loops after distribution, f and δ are determined for the new loops. Loop distribution does not affect f, δ, F, Δ, or O of any loop other than the one being distributed. Building the new values of f and δ involves rebuilding the values for the references that had edges moved to the outer loop. For every dependence between the two new loop bodies, the sink values of f and δ must be recomputed from scratch.

When loop fusion causes edges to be carried by the new larger loop, loop distribution pushes those dependences out to the outer loop. All dependences carried by the original loop with both endpoints in one of the new loops will be carried by the new loop. Any edge between the new loops and carried by the original loop can be replaced by a loop independent edge between the endpoints.

```
                          DO I = 1, N
DO Old = 1, N                 loop body #1
    loop body #1          ENDDO
    loop body #2          DO J = 1, N
ENDDO                         loop body #2
                          ENDDO
```

Figure 4.3 Loop Distribution

The references that must have their overflow values recomputed are exactly the set that could have introduced new misses during distribution (i.e., the set that produced new hits during fusion). If $\delta_{IorJ}(A) \neq \delta_{old}(A)$, then the reference is no longer carried by this loop nest and must be checked to see if distribution creates misses. The new edges are carried by the outer loop and will be misses when $O(outer) = 0$.

After distribution, the edges between iterations of a single loop body span less code than before distribution, so they may become hits. A reference can become a hit when the new distance is greater than the $O(I)$ or $O(J)$ (abbreviated $O(IorJ)$) and less than the original $O(old)$. The only references that can cause this to occur are those with loop carried dependences, or $\delta_{IorJ}(A) = 0$

Figure 4.4 gives the algorithm to determine the profitability of loop distribution. Similar to loop fusion, the algorithm can be divided into five sections: initializing f and δ for the new loop, computing the effect of changed dependences, computing F and Δ, counting the new hits, and counting the new misses.

Computing the effect of the changed dependence graph requires examining every dependence, D, in the loop nest. All other sections each require a single pass over the references in the loop. The work for each reference in each pass is constant, so these sections will take time proportional to the number of references. The worst case time required to compute the effect of distribution is $O(\| D \|)$.

Example

The following loop demonstrates how distribution can improve memory performance when the loop being split uses more data than the size of the cache and at least one of the two new loops fits into the cache.

```
forall statements A in new loop I
        $f_i(A) = f_{old}(A)$
end
forall statements A in new loop J
        $f_j(A) = f_{old}(A)$
end

forall dependences e between loop I and J
        insert dependence edge $f, LC(f) = Outer, D(f) = 0$ to replace e
        recompute $f_{IorJ}(SINK(e))$ and $\delta_{IorJ}(SINK(e))$
end

$F(I) = \sum f_I$
$\Delta(I) = \sum \delta_I$
$O(I) = (M - F(I))/\Delta(I) + 1$

$F(J) = \sum f_J$
$\Delta(J) = \sum \delta_J$
$O(J) = (M - F(J))/\Delta(J) + 1$

/* determine if reference becomes a hit */
profit = 0
forall references A in loop old
        if ( $f_{old} \neq 0$ and $\delta_{old}(A) = 0$ /* there exists a loop carried dependence */
             & $O(IorJ) > D(e)$ /* edge is a hit */
             & $O(old) \leq D(e)$ /* edge was a miss */
           then profit = profit + $N * f_{old}(A)$
end

/* determine if reference becomes a miss*/
forall references A in loop old

        if ( $\delta_{old}(A) \neq \delta_{IorJ}(A)$ /* is the sink of an edge between the loop bodies */
             & $O(outer) = 0$ /* edge is a miss */
             & $O(IorJ) > D(e)$ /* overflow iteration > than old edge's distance */
           then profit = profit -$N * \delta_{IorJ}(A))$
end

/* profit is the net change in memory hits from fusion */
```

Figure 4.4 Profitability of Loop Distribution

```
DO Old = 1, 100
   DO K = 1, 5000
      A(Old,K) = A(Old,K)*B(K)
   ENDDO
   DO K = 1 ,5000
      AA(Old,K) = AA(Old,K)+C(K)
   ENDDO
ENDDO
```

```
DO I = 1, 100
   DO K = 1, 5000
      A(I,K) = A(I,K)*B(K)
   ENDDO
ENDDO
DO J = 1, 100
   DO K = 1, 5000
      AA(J,K) = AA(J,K)+C(K)
   ENDDO
ENDDO
```

Before loop distribution, each K loop uses 10,000 elements (or 40,000 bytes single precision). When both new loops are executed, a total of 80,000 bytes are touched. Since this is greater than a 32K cache, the elements of B and C are pushed out before they can be used a second time. After distribution, only 40,000 bytes are used in each loop, allowing B and C to be retrieved from cache after the first execution. Before distribution, the loops executed 2,000,000 loads (all misses) and 1,000,000 stores (all hits). After distribution, the 1,000,000 stores and 990,000 of the loads are still hits. Thus, loop distribution raised the hit ratio from 33% to 66% for this code segment.

The profitability of fusion on the distributed loops in the example is very negative (i.e., fusion increases the misses by 990,000). Thus, recombining parallel sections, should not be done when the two original loops reuse array values in the cache but the combined loop causes the values to be replaced in the cache before being reused.

4.4 Loop Interchange

As shown in Figure 4.5, loop interchange simply reverses the order of execution of a pair of nested loops. It is used to move parallel loops out to lower levels of nesting (increasing the parallel region size) or move vectorizable loops toward higher levels of nesting based on the target hardware.

Loop interchange is safe when no dependence edges are reversed by the transformation. Edges can be reversed when the direction vector associated with a reference pair has different directions for the I and J loops (for instance $(<,>)$ or $(>,<)$) [All83] in a nest of loops indexed by I and J.

```
DO I = 1, N                   DO J = 1, M
    DO J = 1, M                   DO I = 1, N
        loop body                     loop body
    ENDDO                         ENDO
ENDDO                         ENDDO
```

Figure 4.5 Loop Interchange

Loop interchange does not change the statements executed during either the outer or inner loop. F, Δ, and O remain unchanged for those loops during interchange; however, the values of f and δ can change for the two loops being interchanged. The dependences carried by the two loops do not change, but the amount of memory used by an iteration of each loop changes. Thus, new values of f and δ must be computed for both loops.

Loop independent edges are carried by the innermost loop that contains both endpoints after interchange. Since the innermost loop has changed, they are actually carried by a new loop, but the dependence itself does not change. Thus, $f(I)$ and $f(J)$ are both still equal to 0.

Before interchange, one iteration of the J loop causes its loop body to be executed only one time, while one iteration of the I loop results in M executions of its loop body. After interchange, the I loop requires only one iteration of its loop body, while the J loop now requires N iterations of its loop body.. The new overflow iteration can be computed from the previous values of f and δ, and the number of times each loop executes.

Since one iteration of the I loop now requires only one execution of the loop body as one J loop did before interchange, $f_{i_{new}} = f_{j_{old}}$. If an I loop carried dependence exists, no additional memory is required (if $\delta_{i_{old}} = 0$, then $\delta_{i_{new}} = 0$). If no I loop carried dependence exists, then $\delta_{i_{new}} = f_{i_{new}}$. The new J loop values can be computed by adding the amount of memory used by the iterations of the I loop. The amount used by the first iteration of J is $f_{i_{new}} + (N-1)\delta_{i_{new}}$. The amount used by later iterations of J, $\delta_{j_{new}}$, equals zero, if a J loop carried dependence exists and equals $f_{j_{new}}$, otherwise.

4.4.1 Example

The following loop demonstrates a situation where loop interchange can reduce the number of cache misses.

```
DO I = 1, 100                     DO J = 1, 10000
    DO J = 1, 10000                   DO I = 1, 100
        A(I,J) = B(I)*C(J)                A(I,J) = B(I)*C(J)
    ENDDO                             ENDO
ENDDO                             ENDDO
```

In the example, the innermost loop requires 20,001 words of memory before interchange. Every access to A and C will be a miss (assuming a 32K cache), although each B will be used 10,000 times without being removed from cache. Before interchange there are 999,990 hits and 2,000,010 misses during execution. After interchange, the inner loop only requires 201 words of memory, easily fitting into the cache. Now each value of C is used 100 times, without being replaced in the cache. The 100 values of B are maintained in cache for the entire loop nest, and only A is loaded on every iteration. After loop interchange, the number of hits increases to 1,989,990 while the number of misses decreases to 1,010,010. In this example, loop interchange removes about half of the delays for misses that would have occurred without interchange.

4.5 Strip Mine and Unroll

Strip mining (illustrated in Figure 4.7), groups the iterations of a loop, typically the inner loop, into larger segments. The original loop's step size increases to accommodate the increased work being done within each of its iterations. Unrolling, illustrated in Figure 4.8, also groups the iterations of a loop, but rather than forming a new inner loop, the loop body is replicated some number of times. Again, the loop step of the original loop is increased.

Thus, strip mining and unrolling are two forms of the same transformation. Both transformations take a number of iterations and package them as a single unit that can be manipulated by other transformations. If the new loop produced by strip mining were completely unrolled, the resulting code would be identical to the code after unrolling the original code. Strip mining builds a new loop that contains some number of iterations; unrolling just increases the size of the already existing inner

```
                                  DO I = 1, N, SM
DO I = 1, N                           DO New = 0, SM - 1
    loop body                             loop body
ENDDO                                 ENDDO
                                  ENDDO
```

Figure 4.7 Strip Mining

```
                                  DO I = 1, N, UF
DO I = 1, N                           loop body 1
    loop body                             ...
ENDDO                                 loop body UF
                                  ENDDO
```

Figure 4.8 Unrolling

loop. The loop structure that strip mining introduces slows the resulting code slightly. Unrolling reduces the number of executed loop headers but increases the number of generated instructions.

Strip mine and unroll does not change the order of execution of the modified loop, but it can be used to increase the effectiveness of other transformations. This combination is useful in building chunks of computation that use a specific amount of memory. By varying the size of the chunks, these transformations can be used to tune the results of other transformations. In the next sections, we show that by choosing the correct chunk size, interchange or fusion can be reliably used to improve memory performance of nested loops.

Generating the new dependences and computing f and δ for the loop introduced by strip mining is straightforward. Every loop carried dependence on the original loop will have a matching dependence carried by the new loop with $D(e) = \lceil D(orig)/SM \rceil$ (where SM is the strip width). The memory used by the first iteration of the new loop is equal to the memory used by SM iterations of the original loop.

$$f_{SM}(A) = f_{orig}(A) + \delta_{orig}(A) * (SM - 1)$$
$$\delta_{SM}(A) = \delta_{orig}(A) * SM$$

and

$$F(SM) = F(orig) + \Delta(orig) * (SM - 1)$$
$$\Delta(SM) = \Delta(orig) * SM$$

Unrolling produces no new loop, but it changes the values of f and δ for the original loop. A loop unrolled UF times accesses the same memory as the strip mine loop, where $SM = UF$. So,

$$f_{UF}(A) = f_{orig}(A) + \delta_{orig}(A) * (UF - 1)$$
$$\delta_{UF}(A) = \delta_{orig}(A) * UF$$

and

$$F(UF) = F(orig) + \Delta(orig) * (UF - 1)$$
$$\Delta(UF) = \Delta(orig) * UF$$

Since no changes occur in actual statement order, there is no change in predicted memory performance.

4.6 Peel and Jam

Peel and jam is a transformation that fuses loops with certain kinds of fusion preventing dependence edges. Fusion preventing edges are dependences that would be reversed by fusion. An example is when the first loop defines a value on the second iteration and the second loop uses it in its first iteration. The two loops could not be fused because the value would be used before it was defined. By peeling a number of iterations of the first loop body equal to the distance of the fusion preventing dependence, the correct relationship between the references in the two loop bodies is maintained after fusion.

In peel and jam, the first loop of a pair of loops has a small number of iterations unrolled into a prologue. The majority of the iterations are left in a loop that can be fused with the second loop body (see template in Figure 4.9). Peel and jam can be viewed as aligning the loop nest. The peel step causes a number of iterations of the first loop to be executed prior to execution of the fused loop nest and the fusion step aligns the loop bounds of the two loops.

```
DO I = 1, M+1                 loop body#1(1)
    loop body#1(I)            DO I = 1, M
ENDDO                             loop body#1(I+1)
DO I = 1, M                       loop body#2(I)
    loop body#2(I)            ENDDO
ENDDO
```

Figure 4.9 Peel and Jam

Unlike the other transformations discussed in this chapter, peel and jam is not a well known transformation. As mentioned earlier, peel and jam is very similar to loop alignment [AK88], [ACK86], [Cal87]. In these papers, loop alignment is used to bring all accesses of a memory location to the same iteration of a loop. Callahan [Cal87] states that fusion preventing dependence edges are equivalent to alignment preventing edges. He presents an algorithm that fuses by aligning two loop bodies and placing guards to guarantee that statements only execute on the correct iterations. Peel and jam just attempts to guarantee that the dependences are not reversed when fusion is performed. Peel and jam does not require that the references to a memory location all occur on the same iteration, but does require that all dependences after fusion be from a reference originally in first loop body to a reference originally in the second loop body.

The effect of peel and jam on memory performance is exactly the same as loop fusion. Thus, the profitability of peel and jam can be found using the algorithm shown in Figure 4.2.

4.6.1 Safety

To ensure a safe peel and jam, the compiler must be able to show that no dependence edge is reversed. That is, a definition of fusion preventing dependences is required.

> **Lemma 4.1** Fusion is illegal if there exists a statement I in loop 1 and statement J in loop 2 such that a fused loop carried dependence would exist from J to I after fusion. Any dependence edge that would cause fusion to be illegal is said to be a fusion preventing dependence.

Proof see [Wol82] □

Referring to loop in lemma 4.1, peel and jam attempts to guarantee that if I and J access the same location m, then after transformations J accesses m after I accesses m. If the access by J is after the access by I, then the ordering previous to fusion is maintained.

> **Lemma 4.2** If the number of iterations that are peeled, p, is greater than or equal to the maximum distance, d, of any fusion preventing dependence, then peel and jam is a safe reordering transformation.

Proof After executing p iterations of the first loop body before entering the fused loop, p is added to the iteration number of every reference in the first loop. The distance vectors of the fusion preventing dependences that were previously (d) will now be $d - p$. But $p > d$ so the dependence is no longer from J to I. The dependence is now from I to J and is not fusion preventing. □

The application of safe peel and jam depends on the feasibility of computing the distance vector of the fusion preventing dependences. Uniformly generated dependences can be accurately computed but often represent families of dependences. When the family contains a single dependence, Gannon describes it as uniquely generated [GJG87]. In practice, it is likely that for peel and jam to be a useful transformation, all of the fusion preventing edges must be uniquely generated. This will guarantee that each dependence has a known distance and will normally imply that the distance is less than the loop iteration count (a fact that is probably not true with families of dependences).

4.6.2 Example

There is at least one important group of problems that are amenable to peel and jam. This is the set of grid-based partial differential equation (*pde*) solvers. A typical *pde* code makes numerous passes over a grid, successively smoothing the function. Each individual pass uses the old or new values of adjoining points to compute its new value. Since each iteration uses only nearby values, the dependences have very small maximum distances. Also, since the induction variables tend to be easily recognizable, the compiler can accurately compute the distances.

To obtain a better idea of how peel and jam will work in practice, the following two examples show parts of programs in RiCEPS. The first example illustrated in Figure 4.10 is taken from the one-page inner loop that accounts for well over 99% of WANAL1's execution time. The second example is from the inner loop of SIMPLE. However, SIMPLE's inner loop is much longer than a page, so the total improvement is less than for WANAL1.

In the WANAL1 example in Figure 4.10, three copies of the FI and SI arrays are maintained, namely the current, old, and oldest arrays. Each iteration of the outer loop recomputes the arrays. The first loop nest copies the values of the old array into the oldest array, and the current array into the old array, in preparation for recomputing the current values. The second loop nest computes the new value of FI. The computation of each element of FI involves references to six other elements (its old and oldest values and together the old values of the four adjacent points up, down, left and right).

The two loop nests use the same arrays. Fusing the loop nests increases the locality of reference. Moving the K loop inside the first loop nest is straightforward. The J loop can not be fused because the Jth iteration of the second loop uses the J+1th value of the old array, which has not yet been properly modified. The second loop's access would incorrectly use the oldest value instead of the old value.

The fusion preventing anti-dependence between the two statements is uniquely generated with a distance of one. By peeling the first iteration of the upper loop (J), we can safely fuse the two J loops to improve the memory locality.

Exactly the same method will allow the I loops to be fused. This eliminates the data cache misses for three of the references in the second loop nest. When the resulting program was tested with PFC-Sim, peel and jam eliminated 200 million misses during a single execution of the program.

Performing the transformation on other loops, the total reduction approached 500 million data cache misses. This is well over 20% of the total misses in the program (2.35 billion).

Figure 4.11 is a code segment from the Lawrence Livermore National Laboratory's abstracted hydrodynamic program SIMPLE. The two loop nests use overlapping data but cannot be fused to improve memory performance.

```
      DO 30 N = 1,NT
         DO 35 J = 0, M+1
            DO 35 I = 0, M+1
               DO 35 K = -1, 0
                  FI(I,J,K) = FI(I,J,K+1)
                  SI(I,J,K) = SI(I,J,K+1)
35       CONTINUE
C       Solutions for FI
         DO 40 J = 1, M
            DO 40 I = 1, M
               FI(I,J,1) = 2*FI(I,J,0)-FI(I,J,-1)+P*(FI(I+1,J,0)
                  +FI(I-1,J,0)+FI(I,J+1,0)+FI(I,J-1,0)-4*FI(I,J,0))
40       CONTINUE
30    CONTINUE
```

Figure 4.10 Application of Peel and Jam to WANAL1

Before execution of this segment, LMXP is set equal to LMX + 1 and KMXP equal to KMX + 1. Code motion can move the assignment to variable DTN2 above the 701 loop nest, bringing the two loop nests together. Fusing the 702 and 703 loops is also straightforward. This forms two adjacent loop nests that both use the entire A and B arrays.

The second loop nest uses not only the present location of the A and B arrays but also the three adjacent values. These uses prevent fusion. As in the previous example, the fusion preventing anti-dependences that arise from these accesses are uniquely generated and the distance is easily computed. By peeling off the first iteration of both of the 701 loops, the two loop nests can be fused. This will allow the accesses of A and B to be hits throughout the second loop nest.

The relative gain of peel and jam in this example is less than in the preceding example because there are more references in the SIMPLE loops, which are not affected. In the WANAL1 example, 3 of the 8 references were moved closer to their previous access. The loop nest in SIMPLE has 47 references, of which only 2 are affected. During execution of the largest sample, this resulted in the elimination of 8 million misses and increased the overall hit ratio by 1%.

```
C   Form P+Q in a array A and RHO*AREA in B
     DO 701 L=LMN,LMXP
         DO 465 K=2,KMX
             A(K,L) = P(K,L) + Q(K,L)
             B(K,L) = RHO(K,L) * AJ(K,L)
465      CONTINUE
701  CONTINUE
     DTN2 = DTN + DTN
     DO 450 L = LMN, LMX
         DO 702 K=2,KMX
             V1(K)=A(K,L)*(Z(K,L-1)-Z(K-1,L)) +
                 A(K+1,L)*(Z(K+1,L)-Z(K,L-1)) +
                 A(K+1,L+1)*(Z(K,L+1)-Z(K+1,L)) +
                 A(K,L+1)*(Z(K-1,L)-Z(K,L+1))
             V2(K)=A(K,L)*(R(K,L-1)-R(K-1,L)) +
                 A(K+1,L)*(R(K+1,L)-R(K,L-1)) +
                 A(K+1,L+1)*(R(K,L+1)-R(K+1,L)) +
                 A(K,L+1)*(R(K-1,L)-R(K,L+1))
             V3(K)=DTN2/(B(K,L)+B(K+1,L)+B(K,L+1)+B(K+1,L+1))
             U(K,L)=U(K,L)-V1(K)*V3(K)
             W(K,L)=W(K,L)+V2(K)*V3(K)
             V4(K)=ABS(U(K,L))
             V5(K)=ABS(W(K,L))
702      CONTINUE
C
         DO 703 K=2,KMX
             IF (V4(K).LT.0) U(K,L)=0
             IF (V5(K).LT.0) W(K,L)=0
703      CONTINUE
C
450  CONTINUE
```

Figure 4.11 Application of Peel and Jam to SIMPLE

One interesting aspect of peel and jam is that it works on the two programs that have the lowest cache hit ratios for the largest cache size examined. This is probably not completely coincidental. Each smoothing iteration accesses each element in the array a small number of times. The grids tend to be large to improve the accuracy of the solution. This causes very large working sets and poor data cache performance when the workings sets exceed the data cache size.

4.7 Loop Skew and Interchange

Like the "peel", in "peel and jam", the "loop skew", in "loop skew and interchange" has no actual effect on the computation to be performed. The effect is to change the iteration space in a manner that allows another transformation to be applied safely.

Loop skewing modifies the shape of a DO loop's iteration space. A rectangular array access pattern (N rows of M elements) is turned into a trapezoidal access pattern (i.e., the first M rows have increasing length {1 to M}, the middle row has M elements and the last M rows have decreasing length { M to 1}).

Loop skewing [Wol87] is a derivation of the wavefront method [Lam74] for parallelization. Wolfe uses loop skewing to align a loop nest so that it can be interchanged for parallel execution. In particular, loop skewing (Figure 4.12) eliminates the interchange preventing attribute of direction vectors between various references in the loop nests to be interchanged.

Wolfe states [Wol87] that J can be skewed with respect to I by a scale factor of f by

- replacing the lower bound of the J loop, LBJ, with the expression (LBJ+I*f)
- replacing the upper bound of the J loop, UBJ, with the expression (UBJ+I*f)
- replacing all occurrences of J in the loop with the expression (J-I*f)

Safety

Loop skewing is always safe. Thus, the safety of this combination focuses on the safety of loop interchange after skewing. In peel and jam, the single loop before jamming was peeled to modify the distance vector by one before fusion. The fusion

```
DO I = 1, N                     DO J = 2, M+N
   DO J = 1, M                     DO I = max(1,J-N), min(N,J-1)
      Loop body(I,J)                  Loop body(I,J+I)
   ENDDO                           ENDDO
ENDDO                           ENDDO
```

Figure 4.12 Loop Skew

was then safe if the number of peeled iterations was equal to or greater than the fusion preventing dependence's distance. The scale factor, f, of loop skew performs the same function for allowing loop interchange.

> **Conjecture 4.1** If the skew factor, f, is greater than or equal to the maximum distance, d, of any interchange preventing dependence, then loop skew and interchange is a safe reordering transformation.

Example

Data cache management can benefit from loop skewing, similar to the benefits of peel and jam. A small change to the example used in loop interchange produces the following example, which cannot be interchanged without skewing the loops.

```
DO I = 1, 100                     DO J = 2, 10000+100
   DO J = 1, 10000                   DO I = max(1,J-100), min(100,J-1)
      A(I,J) = A(I+1,J-1) + B(J)       A(I,J-I) = A(I+1,J-I-1) + B(J-I)
   ENDDO                             ENDO
ENDDO                             ENDDO
```

Before the loop skew and interchange, every access in the loop is a miss (for a 32 K cache). Between iterations of the I loop, 10,000 different A's and B's are loaded, easily flushing the cache. Overall, the loop would cause 3 million misses at execution time. After loop interchange, the loop performs much better. During each iteration of the I loop, at most 100 B values are loaded, (normally in the range J−1 to J−100). During the next iteration of the J loop, 99 of the B's are reused and only one is loaded. The total number of references to B that are misses is 10,000 when boundary conditions are taken into account. During each iteration of the loop, the store to A uses a location that was loaded in the previous iteration. This is easy to verify by adding one to I in both subscript positions (I+1 and J−(I+1)) resulting in the subscripts of the right side. Every store to A will be a hit, so the only misses occur when it is loaded. Since there are 1 million elements in A, every iteration must bring in a new value. The total number of misses for this loop nest after loop skew and interchange is 1,010,000, about one-third of the number of misses before transformation.

4.8 Blocking Transformations

Finding a blocking size that allows the inner loops to fit into the memory is crucial to effective blocking. Tuning the block size to fit into memory will allow the minimum number of misses to occur and generate the least traffic between the memory levels. This section describes how the overflow iteration can be used to determine the blocking factor that most effectively improves memory performance. Three blocking transformations (strip mine and interchange, unroll and jam, and wavefront blocking) are examined. Methods to find the maximum sized blocks that still fits into memory are described.

4.8.1 Strip Mine and Interchange

Strip mine and interchange can improve codes such as matrix multiplication where there are references with hit ratios determined by dependences carried by the outer loop, and which are misses in spite of small distances. The goal is to strip the inner loop into pieces such that the new inner loop fits entirely into memory. The newly created middle loop is then moved to the outer loop (Figure 4.13). The original outer loop will now have $F(mid) <$ Memory Size, allowing references that have dependence edges with small distances to be hits.

The new values of f, δ, F, Δ, and O can all be determined by treating strip mine and interchange as two consecutive transformations and using the algorithms in the previous sections. The first step is to compute the values for the new strip mined loop. With these new values, the loop interchange algorithm can be executed to determine the effect of moving the strip mine loop to become the outer loop.

To block most effectively with strip mine and interchange, the compiler needs to calculate the optimal strip size. When the distances of dependences are relatively small, the best strip size will be the largest strip that still allows the outer loop's

```
                                   DO New = 1, N, SM
DO I = 1, M                            DO I = 1, M
    DO J = 1, N                            DO J = 0, SM - 1
        loop body                              loop body
    ENDDO                                  ENDDO
ENDDO                                  ENDDO
                                   ENDDO
```

Figure 4.13 Strip Mine and Interchange

dependences to become hits. This moves the fewest inner loop references to the new strip mined loop. When the outer loop dependence distances are not computable, the optimal value may leave some outer loop dependences as misses. The common assumption is that dependences almost all have very short distances (less than 4 or 5). Under this assumption, the optimal strip mine width will be the width that causes the fewest iterations to be moved out and allows all of the outer loop dependences to become hits.

For strip mine and interchange to be an effective transformation, dependences carried by the inner loop of a perfectly nested loop pair must be hits, and the dependences carried by the outer loop must be misses. If dependence distances are small, this will be approximately equivalent to requiring $O(inner) > 0$ and $O(outer) = 0$. In addition, there must be a reference in the loop body which has no dependences carried by an inner loop. Strip mine and interchange will block the inner loop into sections that fit into the available memory. Once the interchange occurs, the old outer loop (which is now in the middle) will have an overflow iteration greater than zero. Dependences carried by the new middle loop may now be hits.

We must find the strip width that pushes the fewest iterations to the outer loop, and still allows every reference with dependences carried by the middle loop to become a hit. That is, the maximum strip mine width, SM, must have $O(mid) >$ max dependence distance carried by the middle loop, d_i.

$$d_i < O(mid) = \frac{MemorySize - F(mid)}{\Delta(mid)} + 1$$

Both F and Δ depend on the selected SM width. The memory used in the first iteration is the memory used by the first iteration of the inner loop. (i.e., $F(J) +$

SM-1 $*\Delta(J)$ where J is the original inner loop). To calculate $\Delta(mid)$, we calculate the part of $\Delta(mid)$ that does not depend on the strip mine width and the part of $\Delta(mid)$ that does depend on the strip mine length. This involves summing the equivalent of F and Δ for references that do not have dependences carried by the middle loop ($\delta_{mid}(A) = f_{mid}(A)$ when the dependence is not carried by the middle loop).

```
forall statements A in the loop body
        if $\delta_i(A) \neq 0$
                then { BASE += $f_j(A)$
                        INCR += $\delta_j(A)$}
end
```

$$\Delta(mid) = BASE + (SM\text{-}1) * INCR$$

Replacing $F(mid)$ and $\Delta(mid)$ produces

$$d_i < \frac{MemorySize - (F(J) + (SM-1) * \Delta(J))}{BASE + (SM-1) * INCR} + 1$$

Solving for SM generates the largest strip mine width that allows the dependences carried by the middle loop to be hits.

$$SM < \frac{MemorySize - F(J) - d_i * BASE + 1}{INCR + \Delta(J)} + 1$$

The optimal strip mine width is the largest integer that satisfies the above equation. Every reference in the loop is examined to compute BASE and INCR. Finding the maximum effective strip mine width can be done in constant time, after BASE and INCR are available. Overall, computing the strip mine width takes time O(#refs in the loop).

Example

Strip mine and interchange is a profitable transformation when there exists a loop nest where the inner loop requires slightly more memory than is available and references are misses due to outer loop carried dependences. The number of references carried by the inner loop must be less than SM times the number of references carried by the outer loop.

```
                                  DO N = 1,100,50
DO I = 1, 100                       DO I = 1, 100
  DO J = 1, 100                       DO J = 0, 49
    A(I,J) = 0                        A(I,J+N) = 0
    DO K = 1, 100                       DO K = 1, 100
      A(I,J) = A(I,J) +                 A(I,J+N) = A(I,J+N) +
               B(I,K) * C(K,J)                     B(I,K) * C(K,J+N)
    ENDDO                             ENDDO
  ENDDO                             ENDDO
ENDDO                             ENDDO
```

Strip mine and interchange substantially reduces the misses in the classic numerical problem, matrix multiplication illustrated above. When a 32K cache is used, the references to the C array are misses. Combined with the initial loads of the A and B arrays, the original program will miss an 8-way set associative cache 1,020,000 times. In the strip mine and interchange version, the J loop uses 5,000 elements of the C array (instead of 10,000). If the array is single precision, a 32K cache holds the C array between iterations of the I loop. Each iteration of the new N loop requires that values be brought back into the cache. Overall, the new program produces only 40,000 misses. For matrix multiplication, strip mine and interchange can remove up to 96% of the generated misses.

Symbolic Computation

Many programs have loops with iteration counts controlled by a value that is input to the program during execution. The computation of optimal strip mine width does not depend on the iteration counts of the involved loops. It is possible that values of F(J) and $\Delta(J)$ have symbolic terms for loops nested within the J loop. Fortunately, the presence of symbolic values does not prevent the application of strip mine and interchange. The example in Figure 4.14 shows how a matrix multiplication with symbolic loop bounds would be transformed by delaying the computation of the strip mine width until execution.

Delaying the computation of the strip mine width until execution time requires adding a three line (FORTRAN) prologue. The first line of the prologue computes the maximum strip mine width. It is important to remember that integer division in FORTRAN truncates the result. Thus, the first statement takes an implicit floor of

```
                                    INT_TMP1 = (8192 - 2*N3)/(2+N3) + 1
                                    INT_TMP2 = (N2 / INT_TMP1) + 1
                                    INT_TMP3 = N2 / INT_TMP2
                                    DO L = 1,N2,INT_TMP3
DO I = 1, N1                           DO I = 1, N1
   DO J = 1, N2                           DO J = 0, INT_TMP3
      A(I,J) = 0                          A(I,J+L) = 0
      DO K = 1, N3                           DO K = 1, N3
         A(I,J) = A(I,J) +                   A(I,J+L) = A(I,J+L) +
                B(I,K) * C(K,J)                        B(I,K) * C(K,J+L)
      ENDDO                               ENDDO
   ENDDO                               ENDDO
ENDDO                               ENDDO
```

Figure 4.14 Strip Mine Width Computed during Execution

the result, guaranteeing that INT_TMP1 is less than the actual result of the division. In the example, a 32K byte cache and single precision floating point array values are assumed. These values would be known during compilation and are adjusted for the target hardware. Different hardware appears as a different magic number for the number of elements in cache (for example, 32K divided by 4 or 8). The next two instructions compute the maximum integer value that evenly divides the loop bound.

Since even the final INT_TMP3 may not divide the loop bound evenly, the loop body must have any potentially remaining iterations peeled off the top of the loop nest. The number of iterations to be peeled is N3 - (INT_TMP2*INT_TMP3).

Optimal block sizes can be computed whenever all the inner loops execute a number of iterations that can be determined during compilation. This is true even if the loop being strip mined has an unknown number of iterations. When the loop bounds of the inner loops are unknown, the computation of block size can be delayed until execution time when the iteration counts are known.

4.8.2 Unroll and Jam

Strip mine and interchange takes an inner loop, blocks it and moves some of the iterations to outer loops, in an effort to allow dependences carried at the outer level to become hits. Unroll and jam moves blocks inward, Figure 4.15, blocking the outer loop and pushing computation into the inner loop. The goal is to have dependences carried by the outer loop move to the inner loop so they can become hits. This should be done without lowering the inner loop's overflow iteration below the distance of the edges currently carried by the inner loop. The transformation is actually unrolling followed by fusion, but "unroll and jam" is the term created by Callahan, Cocke and Kennedy [CCK87] and better describes the global transformation.

The profitability of unroll and jam can be determined by applying the two separate transformations (unroll followed by loop fusion). As in strip mine and interchange, the blocking factor should be tuned to produce the fewest misses. For unroll and jam, the fewest misses occur when the maximum number of iterations is unrolled without causing the inner loop carried dependences to become misses, or

$$d_j < O(J_{new}) = \frac{MemorySize - F(J_{new})}{\Delta(J_{new})} + 1$$

$F(J_{new})$ and $\Delta(J_{new})$ both depend on the number of unrolled loops. For each reference in the program, we can determine the portions of F and Δ that depend on the number of unrolled loops and the remaining parts of F and Δ. The amount of memory used in the first loop iteration after the transformation is equal to the amount of memory used before the transformation, plus the new memory used by each unrolled iteration. The new memory used by one iteration is equal to that used originally by one iteration, less the amount added by a reference that is the sink of an I loop carried dependence. The memory used by later iterations of the new inner loop is equal to the old total, plus the unroll factor times the amount used by each reference that is not the sink of an I loop carried dependence. That is,

```
DO I = 1, M
    DO J = 1, N
        loop body
    ENDDO
ENDDO
```

```
DO I = 1, M, UF
    DO J = 1, N
        loop body 1
        ...
        loop body UF
    ENDDO
ENDDO
```

Figure 4.15 Unroll and Jam

```
forall statements A in the loop body
        if $\delta_i(A) \neq 0$
                then { $BASE$ += $f_j(A)$
                       $INCR$ += $\delta_j(A)$}
end
$F(J_{new}) = F(J_{old}) + (UF\text{-}1)BASE$
$\Delta(J_{new}) = \Delta(J_{old}) + (UF\text{-}1)INCR$
```

Using this form for F and Δ, the previous equation can be solved for UF.

$$UF < \frac{MemorySize - F(J_{old}) - (d_j - 1) * \Delta(J_{old})}{(d_j - 1) * INCR + BASE} + 1$$

Determining the optimal amount of unrolling before jamming requires the computation of the same BASE and INCR required to determine the optimal strip mine width. Calculating the optimal blocking requires O(#refs in the loop).

Both "strip mine and interchange" and "unroll and jam" block the computation into bounded regions that are sized to allow near optimal use of limited sized memory structures. The difference between the two transformations is the loop that gets blocked. In strip mine and interchange, the inner loop is blocked and moved outward. Unroll and jam blocks the outer loop and moved it inward.

Example

Unroll and jam is profitable when there are references depending on outer loop dependences (with distances smaller than the unroll factor), and the unrolling does not cause inner loop references to become misses. In practice, the code explosion that occurs with large unroll factors limits unroll and jam to cases where the overflow iteration of the inner loop is small.

In the example, Figure 4.16, each execution of the K loop uses almost 8K. The overflow iteration for the J loop will be 4 (32K cache). The use of the A array will be a miss 9,900,000 times. The B array will be a hit except to load each element (99,000 misses). The original program has 9,999,000 misses. Unroll and jam increases the amount of memory used by the J loop to almost 32K, reducing the overflow iteration to 1. The J loop carried dependences have a distance of one and remain hits. The first use of A remains a miss, but the second and third uses become hits because

```
DO I = 1, 99
   DO J = 1, 100
      DO K = 1, 1000
         A(J,K) = A(J,K) + B(I,K)
      ENDDO
   ENDDO
ENDDO
```

```
DO I = 1, 99, 3
   DO J = 1, 100
      DO K = 1, 1000
         A(J,K) = A(J,K) + B(I,K)
      ENDDO
      DO K = 1 ,1000
         A(J,K) = A(J,K) + B(I+1,K)
      ENDDO
      DO K 1,1000
         A(J,K) = A(J,K) + B(I+2,K)
      ENDDO
   ENDDO
ENDDO
```

Figure 4.16 Unroll and Jam Example

the dependence is now loop independent. Only 3,300,000 of the misses on A remain. Combined with the 99,000 misses from loading the B array, the unrolled and jammed version has 3,399,000 misses, just slightly over one-third of the original total.

Symbolic Computation

Like strip mine and interchange, unroll and jam can delay computation of the blocking factor until run-time to handle symbolic bounds. Delaying unroll and jam decreases the effectiveness of the transformation. Since new instructions can not be generated inside the loop during execution, the transformation is slightly changed. The blocked loop must maintain its bounds and is not actually unrolled. A symbolic version of the previous example, as shown in Figure 4.17, demonstrates the differences between the transformation at compile time vs. run time.

The inner loop is re-rolled if the computation of the final blocking factor will be delayed past compile time. This is slightly less efficient for several reasons. There is extra loop overhead, and the compiler loses some opportunities to optimize across iteration boundaries. The computation to determine the proper unrolling almost exactly parallels the computation for strip mine and interchange.

```
DO I = 1, N1
   DO J = 1, N2
      DO K = 1, N3
         A(J,K) = A(J,K) + B(I,K)
      ENDDO
   ENDDO
ENDDO
```

```
INT_TMP1 = (8192 - 2*N3) / (N3) + 1
INT_TMP2 = (N2 / INT_TMP1) + 1
INT_TMP3 = N2 / INT_TMP2
DO I = 1, 99, INT_TMP
   DO J = 1, 100
      DO K = 1, 1000
         DO L = 1, INT_TMP
         A(J,K) = A(J,K) + B(I+L,K)
         ENDDO
      ENDDO
      ENDDO
   ENDDO
ENDDO
```

Figure 4.17 Unroll Factor Computed during Execution

4.8.3 Wavefront Blocking

The previous two blocking techniques both use non-reordering transformations to block data accesses into groups that a second transformation reorders in order to improve the data cache performance of the processor. The resulting memory accesses reuse a square block of data, and hopefully reuse each element before it is pushed from the cache.

Two other combinations of transformations were discussed earlier, namely, peel and jam and loop skew and interchange. Peeling and skewing are non-reordering transformations that allow loops with dependences that would otherwise prevent fusion or interchange to be transformed. By merging the sets of transformations, a very powerful pair of blocking transformations is produced, *strip mine, skew and interchange* and *unroll, peel and jam.* Both of these transforming combinations access the data in a pattern that resembles the hyperplanes (or wavefronts) that Lamport described in [Lam74], for parallelization. For parallelization, every element in the wavefront must have all of its inputs computed before the wavefront is executed (since there is no guaranteed order of computation). For blocking, the wavefronts are executed sequentially. Thus, dependences that lie along the wavefront are allowed.

Due to the similarity with wavefronts, this class of transformations is called *wavefront blocking.*

Figures 4.18 and 4.19 show the two wavefront blocking transformations. The profitability of the two wavefront blocking methods can be found using the same methods as their square blocking counterparts.

By performing a loop skew in the middle of strip mine and interchange, the block being accessed is no longer square. Computing the bounds of the resulting trapezoidal region is more complicated than for blocking transformations. Taking vertical slices out of the trapezoid, results in slices of increasing length until the maximum width is achieved. As various columns are completed the slices will decrease in length. Strip mine, skew and interchange accomplishes the changing slice length by intersecting two regions. The first region is the iteration space of the original outer loop (I in the example). The second region the tilted iteration space of the loop created by the strip mine transformation. Computing the intersection is done by taking the maximum value of the two lower edges and the minimum value of the two upper edges. Figure 4.18 shows a generic example of strip mine, skew and interchange.

Unroll, peel and jam also generates a trapezoidal access pattern. Unrolling the original outer loop (I in the example), produces a number of copies of the inner loop (J). To fuse the first copy with the second copy, one iteration of the first loop must be peeled. To fuse the first loop with the UF copy of the loop, UF - 1 iteration of the first loop must be executed before the fused loop. This produces the large prologue seen in Figure 4.19. Since the loops must all execute the same number of total iterations, an epilogue completing each copies iterations is also generated. The amount of code generated grows rapidly as the unroll factor increases. The prologue and epilogues

```
                              DO New = 2,N+(M/SM)
DO I = 1, N                      DO I = max(1,New-N),min(N,New-1)
   DO J = 1, M                      DO J = 1, SM
      loop body(I,J)                   loop body(I,J+SM*(New-I-1))
   ENDDO                            ENDDO
ENDDO                            ENDDO
                              ENDDO
```

Figure 4.18 Wavefront Blocking: Strip Mine, Skew and Interchange

```
DO I = 1, N
    DO J = 1, M
        loop body
    ENDDO
ENDDO
```

```
DO I = 1, N, UF
    DO J = 1, UF-1
        loopbody 1
    ENDDO
        ⋮
    DO J = UF-1, UF-1
        loopbody UF-1
    ENDDO
    DO J = UF, M
        loopbody 1
        loopbody 2
            ⋮
        loopbody UF
    ENDDO
    DO J = M+1, M+1
        loopbody 2
    ENDDO
        ⋮
    DO J = M+1, M+UF
        loopbody UF
    ENDDO
ENDDO
```

Figure 4.19 Wavefront Blocking: Unroll, Peel and Jam

can be fused into the main loop by using guards to control the execution of the various loop bodies on each iteration. This substantially reduces the code explosion, but adds overhead to each loop body during execution. It may also reduce the opportunities for scalar optimization between the various copies of the loop body.

The wavefront blocking transformations allow us to block any loop nest where the dependence distances can be bounded from below. The transformations are tedious to implement correctly by hand, but can be done during compilation. The problems of implementing these transformations at compile time deal with the accuracy of the dependence graph and the presence of extraneous dependence edges introduced by the programmer. Section 4.9 determines whether the memory performance of the programs in RiCEPS could be improved by these transformations.

4.8.4 Choosing the Proper Blocking

A pair of perfectly nested loops, where one iteration of the inner loop body fits into the memory, but dependences carried by the outer loop are misses, is a good candidate for blocking transformations to improve memory performance. The previous sections derived the optimal blocking factor for strip mine and interchange, as well as unroll and jam. To decide which blocking transformation is most effective, we calculate the profit from each transformation.

Optimal strip mine and interchange causes dependences carried by the outer loop to become hits, but moves a fraction of the inner loop carried dependences outward, where they will be misses. For each reference with an inner loop carried dependence, there will be a dependence carried by the strip mine loop. The number of misses after strip mine and interchange can be computed from $f_{SM}(A)$ for each reference in the program.

The number of misses after optimal unroll and jam can also be determined by looking at each reference. Every dependence carried by the inner loop is unchanged by unroll and jam. The dependences carried by the outer loop that now reside in the same iteration are hits. If a loop is unrolled UF times, then a dependence with distance of one would now be a hit for each iteration except the first. Likewise, a dependence with length of two will miss on the first two iterations, but the third iteration will use the same location as the first, which is still present in the cache.

```
/* Compute BASE and INCR for optimal blocking width calculation */
forall statements A in the loop body
        if $\delta_i(A) = 0$
                then $BASE = f_j(A)$
                     $INCR = \delta_j(A)$
end

/* Calculate the optimal blocking widths UF — for Unroll and Jam */
/* and SM — for Strip Mine and Interchange */

$UF = \lfloor \frac{MS-F(J)-(d_j-1)*\Delta(J)}{(d_j-1)*INCR+BASE} + 1 \rfloor$

$SM = \lfloor \frac{MS-F(J)-d_i*BASE+1}{INCR+\Delta(J)} + 1 \rfloor$

/* For each reference in the program, calculate the number of misses */
/* after blocking, keeping a relative profitability count */

forall statements A in the loop body

if ($\delta_j(A) = 0$ & $f_j(A) \neq 0$) /* misses — Strip mine and Interchange */
        /* there exists a J loop carried dependence that carries data */
        then
                /* if there exist an I loop carried dependence, only the first */
                /* access misses, if not every iteration of the strip mine loop */
                /* misses on every value used by the I loop */

                if ($\delta_i(A) = 0$) then $misses = f_i(A)$
                    else $misses = SM * N * \delta_i(A)$
endif
profit = profit - misses

if ($\delta_i(A) = 0$ & $f_i(A) \neq 0$) /* misses — Unroll and Jam */
        /* there exists an I loop carried dependence that carries data */
        then
                /* The edges that remain misses are the edges */
                /* that span the unrolled loop's edges */

                find distance d of the I loop carried dependence into A
                $misses = \frac{d}{UF} * f_i(A)$
endif
profit = profit + misses
end
if (profit > 0) then return("unroll and jam")
        else return("strip mine and interchange")
```

Figure 4.20 Profitability of Blocking

The total number of misses at each reference is $\frac{d}{UF}$ of the executions of each reference that depend on an outer loop carried dependence.

The algorithm in Figure 4.20 counts the misses that remain after computing the optimal blocking factor for each transformation and returns the blocking factor and type of transformation that should be applied to the loop nest. Identifying the correct blocking transformation to apply to a loop nest requires that the distance of the outer loop carried dependences be determined for every reference. This could be computed and saved from an earlier pass through the dependence graph, but can involve re-examining each dependence edge that terminates in the loop.

4.9 Effectiveness

Using the programs in RiCEPS, a study of the effectiveness of the transformations described in this chapter was conducted. The profiler in PFC-Sim was used to identify the most frequently executed statements, since these basic blocks contain the references that dominate the overall memory performance. An automatic system should not restrict its focus on these basic blocks, but most of the improvement will occur from transformations to these inner loops.

The programs were first examined to see whether an automatic system built on a parallelizing compiler could be expected to noticeably increase the performance of supercomputer applications. For those programs where transformations could not reduce the number of data cache misses, the programs were analyzed to determine if the problem was inherent in the algorithm or if it was a function of the program itself.

In this study, the programs divided into three groups: the transformable, the semi-transformable, and the non-transformable based on the success of automatic transformation in improving cache performance.

4.9.1 Transformable Programs

Transformable programs are programs that a compiler could automatically transform to improve cache performance. This group includes two whole programs (MATRIX and WANAL1) and parts of two other programs (ONEDIM and SIMPLE).

The precision of interprocedural information did not affect the analysis of the tested programs since the transformations affected code segments that included no procedure calls. However, the presence of interprocedural array analysis is important for several of the semi-transformable programs.

The misses in MATRIX can be almost eliminated by strip mine and interchange of the outer two loops, as was shown in the example at the end of the strip mine and interchange section. If the middle loop is split in half and moved to the outside, the number of misses (for a 32K 4-byte line cache) falls from 932,576 to 40,000. Since the loop uses 30,000 different values, proper blocking almost completely eliminates values being pushed from the cache prematurely.

WANAL1 had 2.35 billion misses before transformations. Figure 4.10 shows the innermost loop. The loop interchange, moving the K loop inside the I and J loops, reduces the misses by 336 million. About 250 million more misses were removed by that one application of peel and jam. Another 250 million misses were removed by first distributing the top loop, and peeling and jamming the loop that initializes the SI array with the loop nest that computes the new values of SI. Variable renaming of a summation variable allowed fusion of three loops, eliminating another 200 million misses. Overall, almost half of the misses could be eliminated from the program giving the lowest hit ratio of the tested programs.

Figure 4.11 shows a segment from SIMPLE. Peel and jam eliminated half of the misses that occurred in the loop nests that were transformed. Unfortunately, while almost all of the computation in WANAL1 was in the loops that could be fused, only a small fraction of SIMPLE's computation is contained in the jammed loop nests. Peel and jam still increased the hit ratio by 1%, from 77% to 78%.

Approximately 60% of the misses in ONEDIM occur in three matrix multiply operations. Strip mine and interchange of the three loop nests is identical to that used in the MATRIX program. This should allow the hit ratio (for a 32K cache) to increase from 79% to about 91%. The remaining misses occur in a loop nest that can not be improved and will be discussed in the next section.

Overall, the results of applying transformations to improve memory performance are at best mixed and possibly even disappointing. Only about four programs (out of twelve) could be improved at all, and only three of those were improved significantly. The overall average hit ratio for a 16K cache increased by about 5% to 76%. For a 32K cache, the average hit ratio increased to 81%. Larger caches did not see as much increase (since MATRIX and later ONEDIM fit into the cache), but 64K and 128K caches still had their average hit ratio raised by 3.5%. A 256K increased 3% to 94%. The effect of transformations to increase memory performance is to effectively double the cache size provided by the hardware. The improved programs use a 16K cache almost exactly as well as the original programs used a 32K cache.

Effectively doubling the cache size of any computer is a worthwhile result, but the fact that only one-third of the programs could be modified was disappointing. The following two sections discuss the results of examining each program to determine why transformations were not possible.

4.9.2 Non-transformable Programs

Only two programs (LINPACKD and EFIE304) and part of a third (ONEDIM) had fundamental algorithmic reasons for their inability to be blocked into versions with better memory performance. In all three cases, the problem is pivoting for numerical stability.

Figure 4.21 is an example abstracted from LINPACKD that will serve as an example of pivoting. Some statements that do not affect the ability to apply transformations have been removed. Since L is computed in the loop, there is no effective method to determine how A(L,*) interacts with A(K,*). This causes dependences with distances that can not be computed.

Algorithmically, the various rows can be interchanged at any time before the Gaussian elimination is completed. The compiler is therefore correct in its assessment of the dependences. The values are potentially reused and the distance between the reuses is not computable.

Even with a system that automatically blocks the program, fusion does not seem possible in these programs. Since the dependence distances between the columns can be bounded, a future system may be possible to do some transformations with this program. The variable L is limited in the range from K+1 to N. This allows direction vectors to be accurately computed. By defining the range of dependences to be a box that resembles the letter L, partial pivoting may be divided into loops that access a specific amount of data.

Pivoting is most likely not the only algorithm that demonstrates a dependence nature that will prevent transformations. It is the only one in this initial version of RiCEPS. In Gaussian elimination, the program could replace the pivoting with a QR

```
      DO 60 K = 1,NM1
C         Find L = Pivot element
          L = IDAMAX(N-K+1,A(K,K), 1) + K + 1
C         Interchange if necessary
          IF (L .EQ. K) GOTO 10
              T = A(L,K)
              A(L,K) = A(K,K)
              A(K,K) = T
10        CONTINUE
C         Compute Multipliers
          T = -1.0D0/A(K,K)
          CALL DSCAL(N-K,T,N,A(K+1,K),1)
C         Row Elimination with Column Indexing
          DO 30 J = K+1, N
              T = A(L,J)
              IF (L .EQ. K) GOTO 20
                                          A(L,J) = A(K,J)
                                          A(K,J) = T
20        CONTINUE
              CALL DAXPY(N-K,T,A(K+1,K),1,A(K+1,J),1)
30        CONTINUE
60        CONTINUE
```

Figure 4.21 LINPACKD — Pivoting Preventing Transformation

factorization, which is amenable to blocking, easier to parallelize, and demonstrates better stability. This kind of algorithm revision may be very appropriate for programs that run on supercomputers. Any algorithm that prevents parallelization at outer levels will probably severely limit performance as the number of processors in a supercomputer increases.

4.9.3 Semi-transformable Programs

The largest group of the tested programs are semi-transformable. Automatic techniques to improve memory performance of these programs do not work due to the way that the program was written. The problems fall basically into two groups: code that is too unstructured for a compiler to analysis well, and code that contains a construct that inhibits transformation.

The best example of ugly code is found in a library routine, FTRVMT called by both MHD2D and SHEAR. Figure 4.22 shows a single loop nest taken from the subroutine FTRVMT. In both programs, a large fraction of the total references occur inside this routine. The programmer of the library routine linearized the references to the array DATA. Although linearization is one method suggested for dependence analysis of multiple subscripted arrays [BC86], linearization, in this case, obscures the actual array access pattern. If other parts of the program are examined, it can be determined that neither NSKIP or MSKIP are constants during compilation.

Since the value of MSKIP is unknown, the distances that arise during dependence analysis are symbolic. The presence of symbolic distance values complicates the application of transformations. If the loop were written without linearization, it may be possible to determine that MSKIP is actually accessing a column of the array in a single step manner. With this knowledge, the loops could be blocked.

FTRVMT has been vectorized (the inner loop is preceded by a CRAY vector directive). Although it may perform as well as any other program written for the same machine and compiler, it can not be effectively ported. It is not only machine dependent, but it is also dependent on the intelligence of the compiler vectorization. When this program was first written (pre-1985), the CRAY compiler did not do a very good job of automatic vectorization. As the compiler has improved, and continues to improve, the linearization (and resulting symbolic distance vectors) inhibit any sort of transformation to the surrounding program.

```
      DO 109 JL = 1 ,I2K
         IF (JL-1) 102,102,104
102      EXJ = (1.,0.)
         DO 103 JJ = JL, NPTS,2*I2K
            DO 103 MM = 1,MTRN
               JS = (JJ-1)*NSKIP + (MM-1)*MSKIP + 1
               H = DATA(JS) - DATA(JS+I2K*NSKIP)
               DATA(JS) = DATA(JS) + DATA(JS+I2K*NSKIP)
               DATA(JS+I2K*NSKIP) = H
103      CONTINUE
         /* two more equivalent loops */
109      CONTINUE
```

Figure 4.22 FTRVMT — Linearization Preventing Transformation

BOAST and EULER1 have very high hit ratios (99+ for a 32K cache), making it difficult to determine where misses, other than the initial loading of the cache, occur. The most heavily executed portions of BOAST are two short (5 and 14 lines) subroutines. The subroutines are passed arrays and access a fixed portion of these arrays. Knowing that the access always remains in a column will be critical in determining that a surrounding loop nest could be wavefront blocked. One major loop nest in the program could be wavefront blocked, if a series of writes on each iteration were removed or reordered. Maintaining the order of write statements and guaranteeing that the right values are present at each write inhibits the transformation of several loop nests.

A second program that would require very precise interprocedural analysis to transform is BARO. The work in its inner loop is broken into three pieces. The smallest segment is contained in the procedure with the loops and the other two segments occur in separate subroutines. To block the loops, information about the access pattern of the arrays in the subroutines must be gathered.

Transforming BARO would be a difficult task. Rather than separating the boundary iterations of the grid from the other iterations, the main body of the loop is an intertwined jumble of checks to see if this iteration is on the boundary, followed by jumps to the appropriate code segments. After the boundary iterations are separated (by the compiler or by hand), the program still could not be parallelized. BARO is a partial differential equation program that needs to maintain both old and new values of arrays. Rather than copy an array from a current array into an old array at every iteration, an extra array dimension is declared. Every access to the array that wants an old value uses the variable JQ3, and every access to the current value uses the variable JQ4. When it is time to copy the current value into the old array and compute a new current value, JQ3 is set equal to JQ4 and JQ4 is set equal to 3-JQ3. This effectively copies the array with only two assignments and one subtraction. Compilers will have a difficult job determining that JQ3 is never equal to JQ4. More importantly for blocking, it is difficult to recognize that the values that were loaded into JQ4 on the previous iteration are being accessed on the current iteration using JQ3. BARO appears to use the values within two grid points to determine the next iteration. It appears that wavefront block angled by two iterations would successfully block the access, improving the memory performance significantly.

SIMPLE is another partial differential equation solver that is amenable to a wave-front blocking technique. Again, present compliers would have difficulty automatically performing the transformations. The problem stems from statements that are inside IF statements and execute only when a run-time debugging variable is set. Aggressive scalar expansion would be required in addition to a way to determine the exit branch leaving the infinite loop is in fact implementing a standard inductive loop (e.g., DO I = 1 to NMAX).

The programs in the semi-transformable group cannot be modified by a compiler, or probably even a programming environment, to improve the memory performance. However, they could be rewritten in a style that would make them amenable to transformation. After they are modified, the resulting code could be ported between machines without further modification.

4.9.4 Automatic Blocking (and Parallelization)

One of the goals of automatic vectorization and parallelization has been the transformation of "dusty deck" FORTRAN programs into equivalent vector or parallel versions. A number of vectorizing compilers do a good job of locating loops that can be vectorized. The success of "dusty deck" vectorization has suggested that automatic parallelization can eventually be achieved.

To be most effective at blocking or parallelization, transformations must be performed on large code segments. To manage memory, the compiler attempts to manipulate the entire program in such a way that a value is never pushed from cache until a computation using it has completed. To achieve minimum parallel computation time, the compiler examines whole programs and attempts to find maximal parallel sections. This contrasts with vectorization which normally examines smaller code segments.

Investigating the applicability of automatic blocking of supercomputer programs has uncovered a basic problem with whole program, automatic techniques. The problem is that nearly all large programs (particularly true in RiCEPS, but also in other programs) have some number of constructs that prevent transformation. Some of the constructs are included as performance tricks (like declaring a single array and using pointers to access it, so that only the pointers need to be swapped). Some constructs are attempts to generate fast code with mediocre compilers (linearizing the arrays for vectorization). Some constructs are unpredictable end conditions (programs that check for convergence after every iteration).

Although existing programs may be difficult to transform, the transformations examined in this chapter could be applied to a program that implements the same algorithms without the use of transformation inhibiting constructs. The resulting programs could be automatically optimized for a wide class of processors. The present technique of program optimization for each new class of supercomputers requires the programmer to manually tune the program to the machine. This is a time-consuming process which may be avoidable as present-day state-of-the-art compiler techniques become widespread. Instead of tuning each program for each machine, the program could be modified a single time. The single modification would be aimed at removing the unnecessary constructs that inhibit the compiler from performing transformations. The compiler would then perform the required tuning for each new type of machine.

What does the transformable program look like? These studies have found that better structured programs are more likely to be transformable. Compiler technology has reached a level of sophistication where structured coding practices can be encouraged not only for readability and maintenance, but to reduce the effort involved in porting programs to a new supercomputer.

4.10 Conclusions

This chapter presented some powerful transformations to be exploited during compiler memory management. These transformations include loop fusion, loop distribution, loop interchange, loop skew, strip mine, unroll, strip mine and interchange, and unroll and jam. Additionally, several new transformations, peel and jam and wavefront blocking, have been defined. Their effects on memory performance have been studied.

The new transformations allow otherwise untransformable loop nests to be transformed into loops that are more amenable to increased memory performance. Peel and jam allows two loops to be fused, in spite of the presence of fusion-preventing dependences (whose distances can be computed). A number of applications can benefit from peel and jam, including many grid based partial differential equation solvers. The two wavefront blocking techniques are extensions of the two blocking transformations strip mine and interchange, and unroll and jam. Wavefront blocking includes skewing and peeling with the blocking transformations, allowing the blocking of loop nests that contain interchange or fusion preventing dependences. Wavefront blocking seems to be applicable to a number of the basic algorithms of the programs in RiCEPS.

Compiler memory management of existing programs is hampered by the desire to transform whole programs. Many existing programs contain constructs that prevent the compiler from improving the program. Some of the constructs are derived from random access of memory in the algorithm itself, but a larger number are introduced by the programmer.

Transformations of whole programs will require extensive interprocedural analysis. To block (or parallelize) a program over call sites, accurate information about the behavior of accesses across procedure boundaries is required. With the blocking techniques in this chapter, we believe that state-of-the-art compilers (including programming environments) are sophisticated enough to produce executables from "nice" programs which execute as fast or faster than hand tuned programs. The "nice" programs will remain near optimal after compilation on a different machine (assuming an equivalent compiler).

Chapter 5

Software Prefetching

5.1 Introduction

Software prefetching attempts to increase the hit ratio for all programs while minimizing the additional required data traffic required. In experimental study, caches with very long cache lines obtained significantly higher hit ratios. For programs with data cache hit ratios below 90%, a cache line of 16 words eliminated 65 - 80% of the misses during execution. Most programs had hit ratios over 95% with 16-word cache lines, although the hit ratios of two programs only reached 90%. However, the amount of memory traffic increased substantially with long cache lines. A 16 word cache line, on average, required three times as much data from the main memory as a 1-word cache line. Thus, long cache lines do not seem to be a good choice for multiprocessors. In addition to generating more traffic, cache coherency conflicts are more common with long cache lines. Lee, Yew and Lawrie [LYL87b] found that the optimal multiprocessor cache line was one or two words for most of the routines that they tested.

We found that hardware prefetching was very successful at reducing the number of misses for almost half the programs in RiCEPS. For most of the programs, the amount of additional generated memory traffic was negligible. Only one program produced more than 15% additional memory traffic and most required less than 1%. The prediction strategy is fixed in the hardware and works when the programs access memory in the assumed manner sequentially (for FORTRAN by columns), but a significant number of the programs accessed data in other patterns (by rows, diagonals or randomly).

When examining the programs for which memory performance was not improved by hardware prefetching, we discovered that many had patterns of array access that could be detected during compilation or execution. Every program showed some type of access pattern in the inner loops. Since many of these patterns were detectable at compile time, a mechanism that allows a compiler to manage prefetching is likely to be effective. This chapter studies the cost and effectiveness of a very simple addition to the hardware, a cache load instruction, to allow software management of prefetching. Using this instruction, a straightforward heuristic is used to bring data into the cache before the actual load occurs. The effectiveness of the algorithm for eliminating misses, while positioning the prefetches a substantial distance from actual loads, is studied.

5.2 Cache Load Instruction

For the compiler to assist the processor in prefetching, the compiler must have a mechanism to inform the cache that a memory address will be needed. A cache load instruction is probably the simplest mechanism for both the compiler writer and the hardware designer to use for prefetching. A cache load instruction can be viewed as a no-wait load to a nonexistent register.

A cache load should have all of the address modes of a machine's regular load instruction. The prefetch for an address looks just like a normal load except no register is specified as a destination. To the program, a cache load looks like a NO-OP. The only effect on execution is that one cycle is expended and the instruction counter increments. On machines that allow multiple instructions to be issued during a cycle, the cache load would only use one of the slots.

A no-wait load forces the architect to build a cache that can have multiple outstanding requests. Even if it was acceptable to queue the prefetches so that they were serviced sequentially by the main memory, it would be necessary to allow multiple requests. If the prefetching failed to prevent a miss, the hardware certainly should not wait for a prefetch to complete before issuing the required load.

Caches that allow more than one outstanding request are being designed [SD88] and implemented [GM87]. Scheurich and Dubois present the design for a lockup-free cache for hiding the delays involved in accessing remote locations in a distributed memory multiprocessor. In their paper, one of the methods described for improving processor performance uses a special cache load instruction. RISC architectures, in their attempt to make every instruction be one cycle and make that cycle as short as possible, have already implemented non-blocking load instructions (e.g., IBM RT). Since the nearest memory is more than one machine cycle away, by not blocking on memory, other computations not involving the load can be executed in cycles that would otherwise be dead. Thus, the hardware problems involved in the design and implementation of a no-wait cache load instruction seem to be manageable problems.

A more difficult question for the architect and compiler designer is how to handle run-time faults on the cache load instructions. The hardware could be constructed so that a fault on a cache load would not be reported and the load aborted. It remains to be seen how difficult this is to implement. When the hardware cannot prevent faults, the compiler will have to prevent cache loads from generating either memory protection faults or page faults that would not otherwise occur. Most of the faults can be prevented by attempting to guarantee that prefetching does not occur for iterations that will not execute. If the last iterations of every loop that attempted prefetching were unrolled, most programs would not generate faults on the cache loads that would not occur on later loads.

As defined, a cache load instruction does not represent a major addition to the complexity of a processor. For some processors (like the IBM RT that already allows more than one outstanding load), it may be possible to add the instruction in microcode with no changes to the silicon. A cache load can be used in the compiler like a regular load instruction at any point previous to the actual load. In particular, VLIW and RISC architectures with non-blocking loads and delayed branches may have many NO-OPs that can be replaced with cache loads, making the cache prefetching free.

5.3 Insertion of Cache Load Instructions

We now outline a simple method for identifying data to prefetch. The code that can cause the most accesses during execution are the inner loops. The references with the greatest possibility of generating a large number of misses are array references within the inner loop that use different elements on each iteration. Any array subscript that uses the innermost loop induction variable will be accessing different values on every iteration of the inner loop. These are the likely candidates for prefetching. For prefetching to be effective at reducing miss delays for the processor, the prefetch must precede the actual load by enough time to allow the load from memory to cache to complete, but not so far that the cache flushes the data back out to memory.

```
forall statements S in program
    if S is a DO
     then
        innermost induction variable = loop induction variable
        innermost step = loop step
     end
    forall array references R in statement S
        if innermost induction variable appears in subscript of R
         then
            insert prefetch instruction with reference R incremented
    end
end
```

Figure 5.1 Insert Prefetch Instructions

The algorithm (Figure 5.1) for software prefetching is simple, based on the observation that a single loop iteration should provide enough execution time to allow a cache load to complete, but one iteration is unlikely to access enough data to cause the cache to flush. The heuristic is: if the innermost induction variable is present anywhere in an array subscript, then add the loop step to the innermost induction variable and issue a cache load instruction for the resulting expression.

The preprocessor for PFC-Sim was modified to perform prefetching, using a cache load instruction and the described heuristic algorithm. There are two important details of the implementation. In a single pass over the program, any array subscript that used the induction variable associated with the textually last DO loop is considered to a array access that could profit from prefetching. This causes initialization in an outer loop to generate prefetches (since there is yet no knowledge of the inner loops), but cleanup after an inner loop does not cause prefetches to be generated (since the inner loop is now the most recently defined). An important detail is the placement of the cache load in the loop. Every statement that requires a cache load is immediately followed by its prefetch. This effectively predicts that if loop iteration I follows a certain control flow path through the inner loop, then iteration I + 1 will follow the same path.

Figure 5.2 shows PFC-Sim output with prefetches for matrix multiplication. The only three generated prefetch instructions are in the innermost loop for the next values of the B and C arrays and in the middle loop for the next element of A (the prefetch of A shows a situation where PFC-Sim has yet to detect the inner loop). A second example, Figure 5.3 (from the subroutine FFTB in SHEAR), demonstrates the placement of prefetches in IF clauses. It also has a redundant prefetch. A loop carried input dependence, with distance of two, exists between the accesses of array E. On the third iteration of the loop, the value accessed by the first load will be exactly the value fetched during the first iteration by the second load. Except the first iteration, cache loads issued to prefetch for the first instruction will discover the value to be in the cache already.

A minor modification to the prefetching algorithm allows successful prefetching of values that neither long cache lines or hardware prefetching mechanism can predict. In particular, when induction variables are located in subscripts of nested array accesses, the correct value to prefetch can be calculated. The correct increment to add to the induction variable in the prefetch instruction is the depth of the array. This allows the subscript, which is itself an array reference, to be prefetched one iteration before it is used. One common example of this behavior is index arrays. The code fragment in Figure 5.4 demonstrates prefetching of nested array references.

Software prefetching can prefetch any value for which a pattern of access can be determined. This can allow successful prefetching of values that long cache lines and hardware prefetching would prefetch. Both methods are likely to bring lines into the cache that will never be accessed (unless the mechanism is lucky) when confronted with programs that use one array to determine the location in a second array. As the example shows, software prefetching accurately prefetches even in these cases.

5.4 Effectiveness

To be effective, software prefetching must succeed in two ways. It must eliminate misses, and there must be enough computation between each prefetch and the actual load for the data to arrive from the main memory.

5.4.1 Hit Ratio

The effectiveness of software prefetching was tested on the programs in RiCEPS. Figure 5.5 compares the hit ratios of three caches for each program: a 32K LRU 4-way set associative with 4-byte cache line, the same cache with hardware prefetching,

```
DO I = 1, N
    DO J = 1, N
        CALL STORE(A(I,J),4,TIME,1)
        CALL PREFETCH(A(I,J+1),4,TIME,2)
        A(I,J) = 0
        DO K = 1, N
            CALL LOAD(A(I,J),4,TIME,3)
            CALL LOAD(B(I,K),4,TIME,4)
            CALL PREFET(B(I,K+1),4,TIME,5)
            CALL LOAD(C(K,J),4,TIME,6)
            CALL PREFET(C(K+1,J),4,TIME,7)
            CALL STORE(A(I,J),4,TIME,8)
            A(I,J) = A(I,J) + B(I,K)*C(K,J)
        ENDDO
    ENDDO
ENDDO
```

Figure 5.2 PFC-Sim with Prefetching — Matrix Multiplication

```
DO I = 2, NXP1
    IF (I .LT. NXP1) THEN
        CALL LOAD(E(NVAR,I-1),8,TIME,766)
        CALL PREFET(E(NVAR,I+1-1),8,TIME,765)
        CALL LOAD(E(NVAR,I+1),8,TIME,764)
        CALL PREFET(E(NVAR,I+1+1),8,TIME,763)
        EDIFF = E(NVAR,I+1) - E(NVAR,I-1)
    ENDIF
    IF (I .EQ. NXP1) THEN
        EDIFF = 0.0D0
    ENDIF
ENDDO
```

Figure 5.3 PFC-Sim with Prefetching - a portion of SHEAR

```
DO L = 1, N
    DO I = 1, M
        CALL LOAD(A(INDEX(I),L),8,TIME,364)
        CALL PREFET(A(INDEX(I+1),L),8,TIME,363)
        CALL PREFET(INDEX(I+2),4,TIME,362)
        CALL LOAD(B(I),8,TIME,361)
        CALL PREFET(B(I+1),8,TIME,360)
        B(I) = A(INDEX(I),L)
    ENDDO
ENDDO
```

Figure 5.4 PFC-Sim with Prefetching — Index Array

and the same cache with software prefetching. 10 of the 12 programs have hit ratios of over 98% for caches with software prefetching. Software prefetching does at least as well as hardware prefetching on every program and successfully prefetches on many programs with non-sequential array accesses. Software prefetching even generates higher hit ratios than very long cache lines for many of the programs and is within a small percentage for the other programs.

One program where software prefetching did not work as well was MHD2D. Closer examination of this program indicated that one loop was responsible for a majority of the misses. In this loop, the programmer had performed his own address generation for the array subscripts. Otherwise, the generated address would have been identified as a loop induction variable during loop induction substitution. Once identified as a reference to the loop induction variable, software prefetching would have eliminated almost every miss in the loop, and MHD2D would have had a hit ratio similar to the rest of the programs.

Another program where software prefetching was not completely successful was MCMB. A large number (about three-quarters) of the misses with prefetching were caused by programmer defined auxiliary induction variables. One loop nest increments J every time through the inner loop and uses J to access arrays. Two loops define an auxiliary induction variable to implement a loop that counts down, rather than using a negative induction step. If PFC-Sim had been invoked after PFC's in-

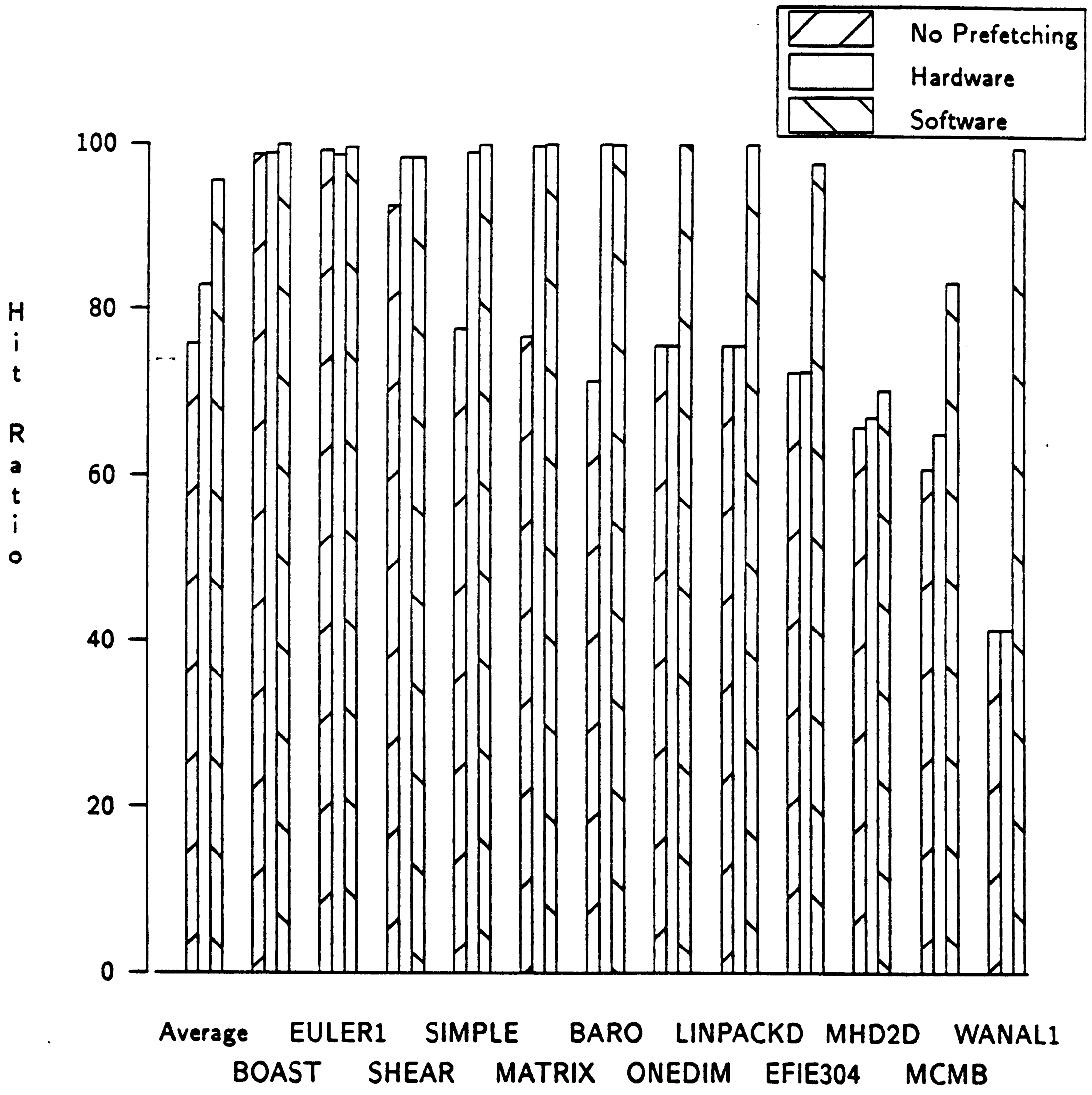

Figure 5.5 Effect of Software Prefetching - Hit Ratios

duction variable substitution, all of these loops would have had prefetch instructions. The number of misses would have been approximately 3%.

Even without catching all of the induction variables, the average miss ratio for a prefetching cache was 95.6%. Excluding the two programs that used alternative induction variables, the average hit ratio would be 99.5%. Thus, prefetching array values one loop before they are needed is very effective for computationally intensive programs, particularly if loop induction variables are identified. Data cache misses can eliminated almost completely.

5.4.2 Time Between Prefetch and Load

If cache loads always immediately precede actual loads, then the actual loads will never cause a memory transfer. However, every load would have to wait for the memory latency of the prefetch. Moving the prefetches one loop iteration away from the actual load provides some amount of execution time to hide the memory latency.

In the run-time routines for PFC-Sim, every prefetch was marked with the program execution time at which it was issued. Whenever a load used a value that had a time field, it recorded how much time had elapsed between the cache load and the actual load. After recording the difference, the time field was zeroed to prevent later accesses from recording their delays.

Figures 5.6 and 5.7 show the number of cycles by which the cache load preceded the register load for all of the programs in RiCEPS. Figure 5.6 shows the percentage of all references in RiCEPS. Several programs have significantly more references than the other programs and dominate this graph. Less than 0.1% of the references were less than 50 cycles away from their prefetch. Miss delays on the references that were successfully prefetched occurred very rarely.

If we normalize each program's results and weigh every program equally, the distribution of time between prefetches and loads changes, as shown in Figure 5.7. This graph has about 1% of the prefetches occurring between 20 and 50 cycles before the load. These prefetches may not complete on some machines before the actual load is reached. But over 95% of the prefetches are over 100 cycles ahead of the first use. Loading values one iteration prior to use successfully hides delays due to cache misses for most of the array accesses in RiCEPS on current memory architectures.

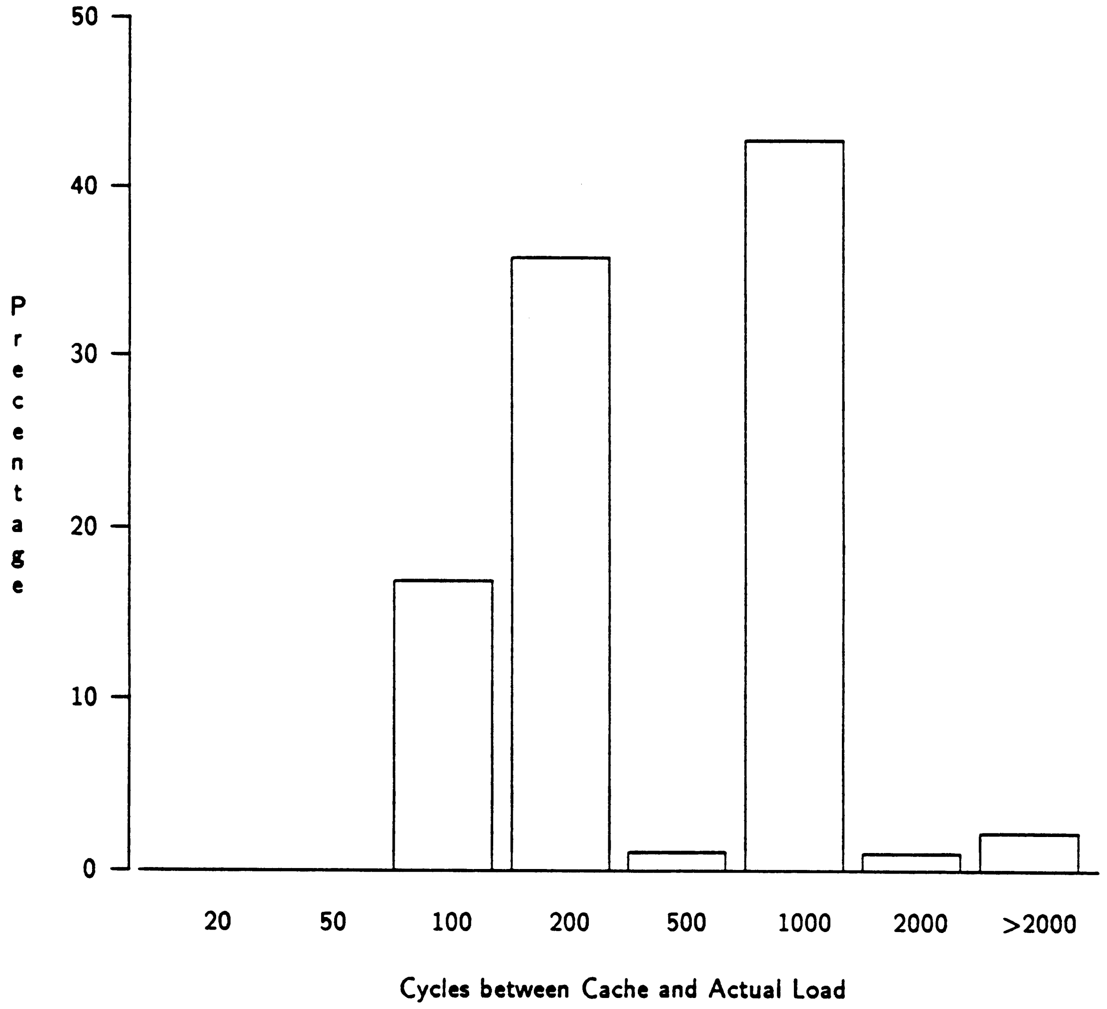

Figure 5.6 Software Prefetching - Time Between Prefetch and Use (Absolute Totals)

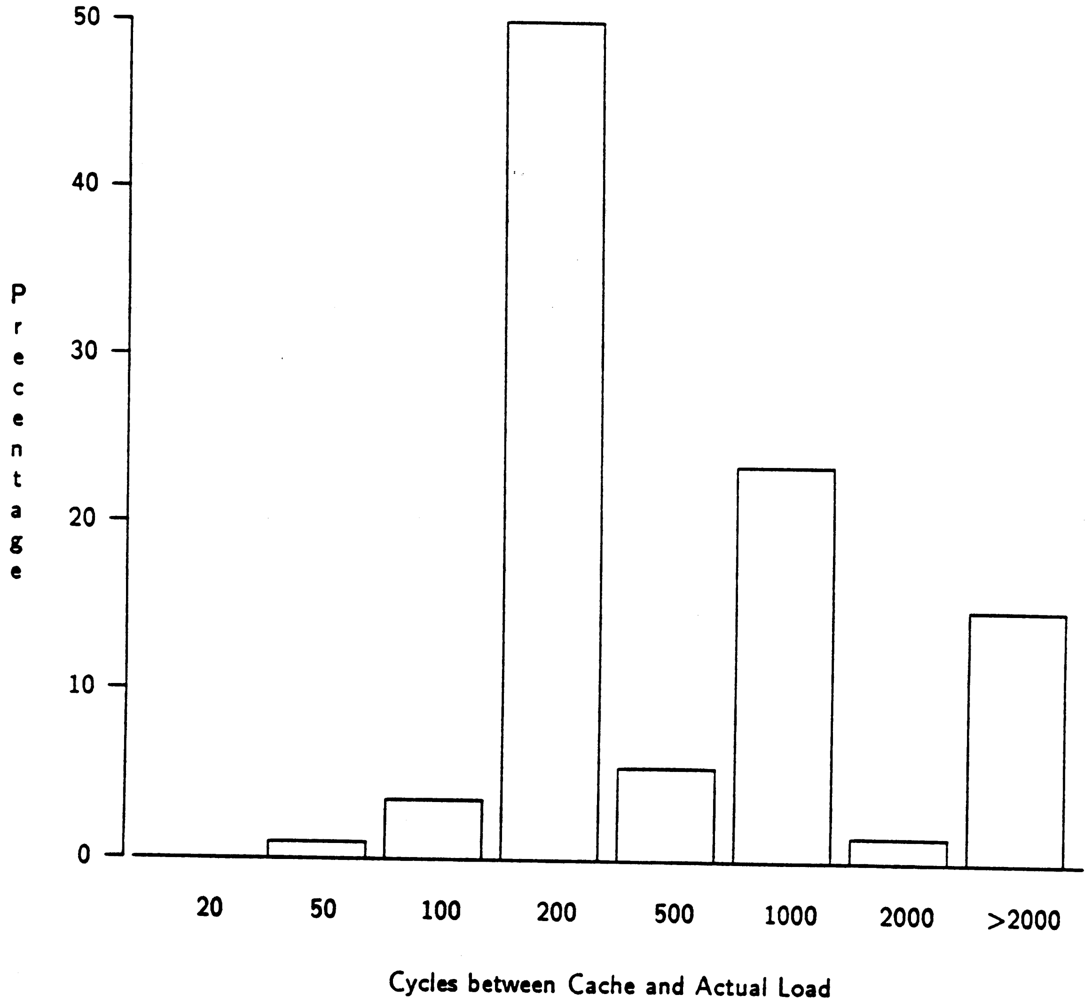

Figure 5.7 Software Prefetching - Time Between Prefetch and Use (Normalized Totals)

We have seen that software prefetching one loop iteration before use is an effective method for eliminating miss delays on sequential processors. For the heuristic to work on multiprocessors, either multiple loop iterations must be scheduled as a unit or the parallelism must be at the outer loop iterations. All parallelizing tools attempt to move parallel loops outward to reduce synchronization overhead. Software prefetching should be equally effective for those parallel programs.

5.5 Costs

Software prefetching would virtually eliminate miss delays for the computationally intensive programs in RiCEPS. If prefetching were free, this could decrease the execution time of programs by up to 50%. For computers with memory approximately 50 cycles away, the average decrease would be over 20%. However, software prefetching is not free. It requires additional traffic and adds instructions to the program.

5.5.1 Data Traffic

One of the reasons for examining the effectiveness of software prefetching was the excellent bandwidth behavior of hardware prefetching. Figure 5.8 compares the data traffic used in software prefetching with that used normally and that used with hardware prefetching.

For most of the programs, hardware prefetching produced between 0 and 2% more memory traffic than a cache with no prefetching. For these programs, software prefetching required slightly more data (fractions of a percent) than hardware prefetching. Software prefetching also requires the same amount of data traffic as hardware prefetching for EULER1, the single program that required 50% more memory accesses. On SHEAR, where hardware prefetching used 15% more data, software prefetching did much better and issued less than 5% more requests to the main memory than when no prefetching occurred.

In summary, software prefetching maintains the good traffic behavior of hardware prefetching and eliminates miss delays. The major cost of software prefetching will be the amount of time required to issue the requests.

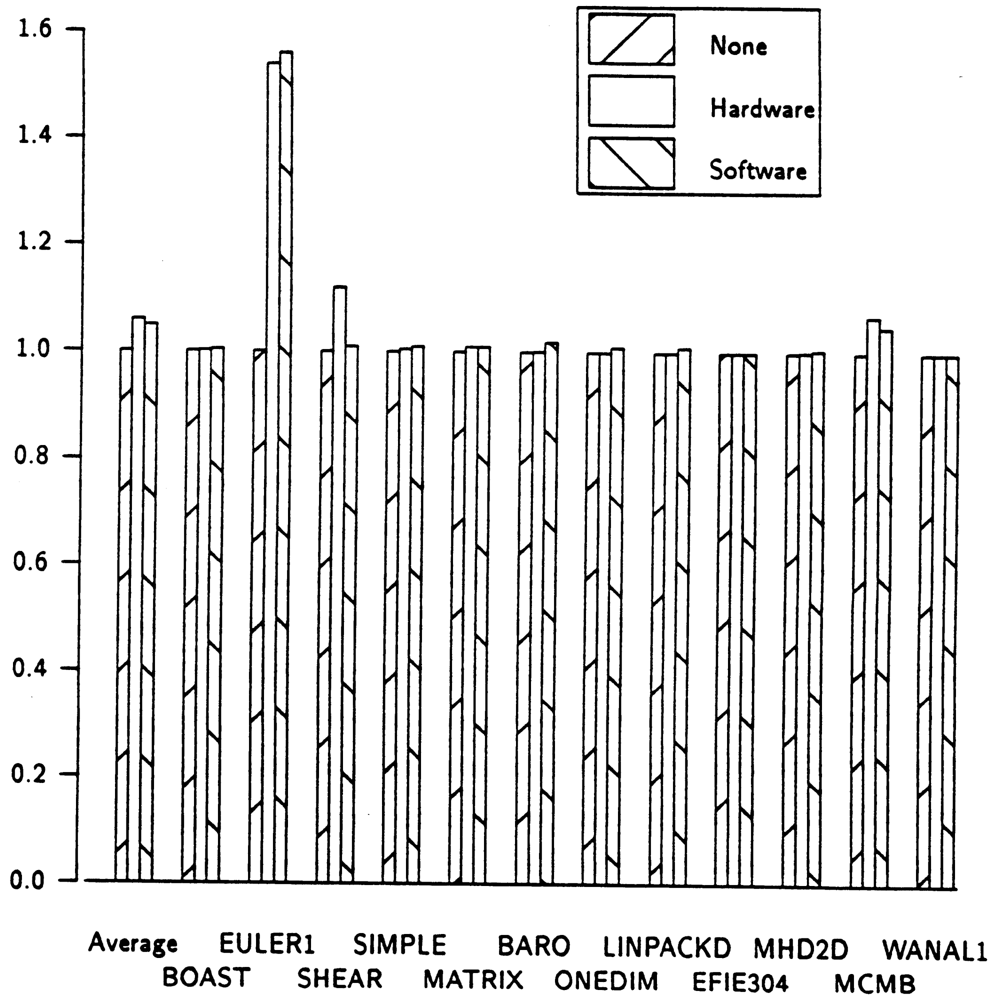

Figure 5.8 Effect of Software Prefetching - Data Traffic

5.5.2 Run-time Overhead

A second overhead associated with software prefetching is the additional execution time to perform the cache loads. There are two types of statements added to programs: cache loads and address generation. Each prefetch requires a single cache load instruction. For each array, there is some amount of address generation that must occur. This can range from a single addition to a complicated arithmetic expression, possibly including more than one integer multiplication.

If every prefetch brought data into the cache, determining the profitability of any individual prefetch would be relatively simple. The prefetch would be profitable whenever the address generation and prefetch take less time than accessing memory. The prefetching mechanism implemented in this work generates many prefetches for memory locations that are already present in the cache (or are already on their way to the cache) or locations that will not be accessed (at least before the location is bumped from the cache). These kinds of prefetches are overhead that successful prefetching must cover.

For the simple prefetching strategy, Table 5.1 shows that slightly less than one-third of the prefetches cause an actual cache miss and force a value into the cache that is used later. For prefetching to be profitable, memory latency should be greater than 3 times the cost of address generation. Unless memory latencies are very high, the address generation overhead will completely overshadow the overlapping of miss delays with execution. It does not decrease execution time to overlap a memory latency with computation that was added to compute the addresses to be prefetched.

The additional execution time required for address computation was estimated with a minor modification to PFC-Sim. As previously stated, the execution time of the program was estimated. By changing the estimated time per execution to include an additional statement for every prefetch instruction, the amount of additional time spent by prefetching is measured. Each prefetch instruction was assumed to require one load (the offset from the previous load), one integer addition, and the prefetch instruction itself. For most programs, the offset is present in registers, reducing the computed overhead values. This estimate is intended to be a reasonable upper bound on the required time.

Table 5.2 shows that the average overhead incurred by a program was 28%. MHD2D had a substantial amount of computation for every prefetched value, result-

Program	Total Prefetches	Useful	Multiple Prefetches	Present	Never Used
LINPACKD	57,425,219	24.6%	23.3%	52.1%	0.0004%
WANAL1	4,004,874,064	58.2%	0.25%	41.1%	0.45%
BOAST	6,946,236	1.7%	4.7%	93.6%	0.01%
MCMB	13,279,100	31.0%	17.5%	51.2%	0.3%
MATRIX	2,010,000	46.4%	0.5%	53.1%	0.01%
SIMPLE	1,161,833,428	22.2%	20.6%	57.1%	0.1%
EFIE304	7,666,945	32.6%	30.8%	36.2%	0.4%
BARO	71,061,469	32.3%	18.1%	49.5%	0.0002%
EULER1	37,154,789	2.3%	2.0%	95.6%	0.02%
SHEAR	372,873,352	26.8%	24.7%	47.9%	0.6%
MHD2D	46,461,504	35.1%	39.8%	24.8%	0.3%
ONEDIM	152,004,856	27.4%	19.6%	53.0%	0.08%
Average		28.4%	16.8%	54.6%	0.19%

Table 5.1 Useful Prefetches

Program	Execution Time No Prefetching	Execution Time Software Prefetching	Software Prefetching Overhead
LINPACKD	1,430,861,530	2,118,18,166	48%
WANAL1	101,160,305,128	165,554,339,949	41%
BOAST	466,316,232	618,891,848	33%
MCMB	1,004,950,150	1,218,560,862	21%
MATRIX	179,561,215	227,801,218	27%
SIMPLE	63,214,137,900	91,090,378,256	44%
EFIE304	562,538,201	666,180,569	18%
BARO	4,794,353,360	6,500,549,985	36%
EULER1	2,417,980,290	2,849,928,149	18%
SHEAR	74,578,049,200	82,369,454,612	10%
MHD2D	14,531,562,400	15,335,199,095	6%
ONEDIM	2,417,980,290	2,849,928,149	34%
Average			28%

Table 5.2 Execution Time with and without Software Prefetching

ing in a very minimal overhead. Other programs, like LINPACKD, performed little computation per reference. Adding prefetching to these programs greatly increased the execution time (up to 48%).

With overhead of 28%, software prefetching is very unlikely to reduce the execution time of programs. For software prefetching to be effective, methods to reduce the costs will be required. Methods to reduce the overhead of software prefetching are discussed in the next section.

5.6 Methods to Reduce Costs

The compiler can reduce the costs of software prefetching in two different ways. The dependence graph and overflow iterations can be used to prevent prefetching of values already present in the cache. Also, the address in a register can be saved between the prefetch and actual load, eliminating the extra address generation. Using these techniques, software prefetching may be profitable for any processor with a cache. The amount of overhead in software prefetching may be reduced to the point that it can be entirely hidden in a VLIW architecture, or any other architecture that tends to have unused instruction slots where cache loads could be placed.

5.6.1 Prediction of Misses

A large percentage of the overhead for the simple software prefetch algorithm is due to generating prefetches for values that are already present in the cache. A better prefetching strategy is to only prefetch references that will be misses (see Figure 5.9).

The overflow iteration was computed by hand for several of the shorter programs in RiCEPS. Table 5.3 compares the hit ratio and the amount of overhead for the two prefetching strategies. For the tested programs, the overflow iteration did a good job of separating the references that should be prefetched from those that should not.[1] The two programs that still had a high percentage of unuseful prefetches were EULER1 and LINPACKD. When the hit ratio was very high, very few prefetches occurred. The overflow iteration version of EULER1 performed about one-third the

[1]When computing the overflow iteration by hand, I have assumed good interprocedural information. If information less exact than regular sections with bounds information is used, the overflow iterations would be less accurate and the number of unnecessary prefetches would be higher.

```
forall statements S in program
        if S is a DO Statement
         then
                innermost induction variable = loop induction variable
                innermost step = loop step
         end
        forall array references R in statement S
                if R is a miss and
                        innermost induction variable appears in subscript of R
                 then
                        insert prefetch instruction
                                        with R's subscripts incremented
        end
end
```

Figure 5.9 Insert Prefetch Instructions (Estimated Misses Only)

Program	Hit Ratio			Useful Prefetches	
	No Prefetch	All Prefetched	Limited Prefetched	All Prefetched	Limited Prefetched
LINPACKD	75.7%	99.9%	99.9%	24.6%	36.9%
WANAL1	41.4%	99.5%	99.4%	58.2%	98.7%
EULER1	99.1%	99.5%	99.7%	2.3%	2.2%
BARO	71.3%	99.8%	99.7%	32.3%	56.2%
MATRIX	76.7%	99.9%	99.7%	46.4%	91.4%
EFIE304	72.4%	97.6%	97.1%	32.6%	70.0%
Average		99.4%	99.3%	32.7%	59.2%

Table 5.3 Useful Prefetches — After Using Overflow Iteration

total number of prefetches, so the total overhead was low, although the percentage of useful prefetches fell only slightly. LINPACKD has triangular loop nests which are hits early and misses after some number of iterations. Prefetching those array references kept the hit ratio for the program high (99.9).

Using the overflow iteration to limit the number of generated prefetches increased the likelihood that any given prefetch actually caused a useful cache load. With every possible value being prefetched, less than one-third of the prefetches caused useful data to be moved into the cache. When the overflow iteration was used to reduce the number of prefetches, almost six out of ten remaining prefetches prevented a later cache miss.

The overflow iteration removed over 54% of the prefetches from the six test programs. This reduced the overhead for the prefetching instructions on the six programs from 31+% to 14+%. If the other programs had shown equivalent reductions, the overhead of software prefetching would be reduced by 12%. This reduction in overhead resulted in a negligible decrease in hit ratio (0.1%). At only 12% overhead, software prefetching begins to be an attractive alternative for computers with memory on the order of 20 cycles from the processor.

Examining several thousand lines of code to determine which references are misses indicated several observations about the calculation of the overflow iteration. In general, it is very easy to look at a small to medium loop and roughly determine the overflow iteration by hand, but it is easy to overlook references. In the first modification of WANAL1, prefetches to two references were incorrectly eliminated. Those two references lowered the total program hit ratio by 6%. Although the algorithm can be implemented by hand, the consistency provided by compiler implementation is preferable.

5.6.2 Register Allocation

Eliminating prefetches for memory locations already in the cache effectively reduces the overhead of software prefetching. The overhead for the remaining prefetches can also be reduced. As defined in the simple prefetching strategy, every prefetch generates its own address, and then the load also generates an address. When the right value is being prefetched, these two addresses will be the same. By keeping the value in a register between the cache load and the actual load, the second address generation is avoided.

By saving the address in a register between the two loads, the cost of a cache load is reduced to a single (possibly hidden) cache load. However saving the address between the two loads greatly increases the lifetime of the address temporary. Increasing the lifetime of temporaries increases the register pressure in the program and can cause more values to be spilled from registers during execution. When the address temporary is spilled, the cost of the cache load is a register store, a register load and the single cache load instruction. To measure the impact on register allocation, several programs were modified and ported to the $\mathcal{R}^n$ environment. The $\mathcal{R}^n$ compiler does graph coloring register allocation [CAC+81], [Cha82] and estimates the amount of spilling that will occur during execution.

Every program ported to $\mathcal{R}^n$ had every prefetchable reference subscript replaced by a temporary variable. The temporary for the next iteration was then calculated in the statement after its use. This caused the address temporary to have a lifetime of one full iteration of the loop, the amount of time that occurs during software prefetching. The overflow iteration had already been used to reduce the number of prefetches that occurred before porting to $\mathcal{R}^n$.

The six programs consisted of a total of 62 routines. The $\mathcal{R}^n$ compiler is still under development, and attempting to compile actual programs uncovered a number of bugs. Only 38 of the routines could successfully pass through register allocation. The original versions of the routines resulted in 223 variables being spilled at an estimated cost of 221,892 cycles. After prefetching address variables were added, the number of spills increased by 58 to 281, and the estimated cost was 708,623 cycles. When all optimizations were activated in the compiler, the number of spills in the original program increased to 387 requiring 353,875 cycles. Prefetching required an additional 74 register spills and 940,813 cycles.

Prefetching increased the number of spills by less than 25% for both the optimized and non-optimized code. The spills tended to occur in more nested loops, increasing the overall cost of spilling by 320% for the non-optimized code and 266% for optimized code. The total cost of spilling is minimal when compared to the number of accesses that actually occur in the programs. WANAL1 alone generates trillions of references.

Unfortunately, the longer routines were less likely pass through the register allocator. For every routine that did pass through the compiler, the estimated spill cost was less than the number of prefetches. Since the cache should maintain any scalar that is used on every iteration, the cost of software prefetching is less than three cache memory operations.

The cost of eliminating duplicate address generation, by saving addresses in registers across iterations is unclear from the experiments. Many routines could not be measured, and the exact correlation between cycles as estimated by $\mathcal{R}^n$ and measured by PFC-Sim is unknown. However, an estimate of the likely effect can be derived. The additional spills from software prefetching are likely to be relatively small (on the order of 20 to 50% more spill sites), but are likely to be in more heavily executed sections of the program (spill costs climbed over 200%).

In summary, using registers to maintain addresses between the prefetch and the actual load may or may not substantially reduce the overhead involved in prefetching. This preliminary study is encouraging, but inconclusive.

5.7 Conclusions

A compiler can almost eliminate cache misses by prefetching the values. A very simple algorithm to perform the prefetching reduced the data cache misses to less than 1% of the register loads in RiCEPS. On machines that do not have empty instruction slots, the cost of prefetching may outweigh the benefits.

In conjunction with an optimizing compiler, the simple method presented here can be modified to produce less than half the previous prefetches with no noticeable performance degradation. This is accomplished by using the overflow iteration to estimate which prefetches are redundant (values already present in the cache) and eliminating them. This reduces the overhead of software prefetching to approximately 12%. The average amount of execution time spent waiting on memory exceeded this when memory was about 20 processor cycles from main memory. The overhead may be further reduced by aggressive register allocation.

Software prefetching appears to be an attractive option on computers where memory is distant and bandwidth is limited. This is an accurate description of most shared memory multiprocessors.

Chapter 6

Conclusions

6.1 Overview

In the introduction, program execution time was divided into two components, T_c, the amount of time required to perform the computation and, T_m, the amount of time spent waiting for data to arrive. The goal of this dissertation was to examine the size of and develop methods to reduce T_m, in particular, $T_{m_{main}}$ the time spent waiting on data cache misses.

In measuring $T_{m_{main}}$, we used several new tools, PFC-Sim and RiCEPS. RiCEPS is a collection of typical supercomputer applications gathered at Rice University to be used for compiler benchmarking. A better understanding of supercomputer performance can be obtained by using a benchmark that resembles actual work loads. To measure the cache performance of the programs in RiCEPS, a new simulation tool, PFC-Sim, was designed and built for this dissertation. PFC-Sim is a program-driven-event tracing facility. By executing concurrently with the program that is being measured, the need to save trace files is eliminated. PFC-Sim was used to examine the cache performance of programs with billions of cache requests. These traces are hundreds of times longer (thousands, in some cases) than any previous cache study has attempted to use.

Since PFC-Sim is program driven, it was possible to examine the behavior of individual references in the original program. When the average hit ratio of the individual references was plotted, a bimodal distribution appeared. The references were either always hits (70% of the references) or always misses (30%). Since a program's misses are concentrated into a small percentage of the actual references, methods to improve cache performance during compilation can be effective.

Before the compiler can attempt to reduce T_m, it must have a mechanism to locate memory requests. Using the dependence graph and the memory size, the overflow iteration was defined to describe the maximum number of iterations of a loop that can fit into memory. The simple method for estimating whether a reference will be a hit or a miss involves a comparison of the dependences that terminate at a reference and the overflow iteration. If the dependences have distances greater than the overflow iteration (or no dependences exist), the reference will be a miss; otherwise, it will be a hit. When computed by hand for a large percentage of the loops in RiCEPS, we found this test to be an accurate estimate of actual performance. The only problem with this method of identifying hits and misses occurs with triangular loops. Triangular loops are the most common example of where individual references do not have hit ratios of either 0% or 100%. The wide part of the triangular section causes data misses, but the narrow section does not. Thus, the proposed estimating method concludes that the reference misses entirely. This causes the reference to be considered by the compiler for potential improvement of memory performance.

The first mechanism to improve memory performance is program transformation. Program transformations that move consecutive uses of a memory location closer together can increase the number of times a value is used before it is replaced. Reasonably fast algorithms to estimate whether a transformation will improve or degrade memory performance have been presented for a large number of transformations. The transformations include peel and jam which is a new mechanism to automatically fuse loops with fusion-preventing dependences that have known distance vectors. Two blocking transformations were extended to increase the number of programs that may be blocked. Strip mine, skew and interchange allows blocking of loops that could not otherwise be interchanged during the blocking process. Similarly, unroll, peel and jam successfully blocks loops that would otherwise fail to fuse back together during blocking.

Using the transformations defined in Chapter 4, it is possible to block most of the algorithms in RiCEPS. Unfortunately, it is also true that present compiler technology will not succeed in automating the blocking process on more than a fraction of the "dusty deck" programs. Current compilers would almost completely block two of the twelve test programs and block a substantial portion of a third. The lack of success is due to the scope of the required transformations. To effectively block a program's data cache, the transformations encompass most of the input programs. This means that if at any point in the program, a construct (which may have been introduced for speed on mediocre compilers, for incremental checkpoints, or just because it was easy to program) inhibits dependence analysis from obtaining an accurate view of the program, blocking cannot be performed. It should be possible to construct a coding guideline, which if followed, would allow most programs to be rewritten, and allow compilers or programming environments to automatically parallelize and otherwise tune the programs for any type of computer. Many programs have been rewritten to improve their performance on vector processors. It should be possible to avoid rewriting programs for every new supercomputer architecture, by removing unnecessary constructs which defeat dependence analysis.

When the number of cache misses cannot be reduced, the only way to reduce T_m is to reduce the average memory latency. One method is to design faster memories. This is expensive and may be impossible. A second method to is to overlap the latency with computation. Software prefetching separates register loads from cache loads. By moving the cache load of a reference that causes cache misses away from the register load, the memory delay involved in moving the value into the cache can be hidden. Simulation has shown that the strategy of retrieving values one iteration before they are actually required can effectively mask any memory delay for over 99% of the array references in a program.

6.2 Future Work

This dissertation has assumed that only one processor is working on a problem and that only one memory exists at a level of the memory hierarchy. Neither of these assumptions is likely to be valid for the next generation of supercomputers. Extending this work will be important to generate the best code for supercomputers.

6.2.1 Multiprocessors

An increasingly popular method of increasing processing power is to supply multiple processors to perform the work. Understanding the memory performance of multiprocessors introduces several important problems not present for uniprocessors.

PFC-Sim is not suited to handle multiple processors. Extensive modifications will be required to accurately simulate the interactions possible between processors. The most important addition will be some method of modeling cache coherency protocols. Correct simulation of memory invalidation traffic will be crucial to correct multiprocessor memory simulation.

Including memory invalidation traffic in the dete::ion of memory references that miss will be a difficult and required task for a compiler trying to minimize multiprocessor memory traffic. Since invalidation traffic may be expensive, methods that detect excessive invalidations and eliminate them will be important.

A third problem to be solved for multiprocessors is whether to transform to improve memory performance or parallelism. The transformations used by parallelization and memory management both work best when applied to the outer loops of a program. Finding rules to guide the spreading of data (to limit traffic between parallel tasks) and the packing of data (to improve cache memory performance). Methods to pack data without reducing parallelism are needed.

6.2.2 Multiple Levels of Cache

Projects like CEDAR are using multiple levels of cache to improve average memory latency. Introducing new levels of the memory hierarchy will require the compiler to tune the program to each level. Iterating of a program source once for each memory level is probably unacceptable. Methods to tune for the entire memory at one time will be desired. Tuning for multiple levels in one pass may be much more efficient than the multiple passes approach.

If two memory levels exist such that the misses of the smaller memory (after tuning) are approximately equal to the misses of the larger memory. The overhead of additional loops may cause the tuning at the outer level to produce slower code. When a single pass is used, it should be possible to detect the marginal improvement and skip tuning of that level.

When multiple levels of cache are present, it is common to make at least one of the caches completely under software control. Local memories require different control methods than described in this work. Using a true cache allowed us to ignore the aliasing problem. If two elements were actually the same, the cache would perform the correct sharing. Local memories do not provide that protection. Allocation into a local memory can not be as aggressive as blocking into a cache.

Bibliography

[ACK86] Randy Allen, David Callahan, and Ken Kennedy. Automatic decomposition of scientific programs for parallel execution. Technical Report Rice COMP TR86-42, Rice University, 1986.

[AK84] Randy Allen and Ken Kennedy. Automatic loop interchange. In *Proceedings of the SIGPLAN 84 Symposium on Compiler Construction*, 1984.

[AK88] Randy Allen and Ken Kennedy. Vector register allocation. Technical Report Rice COMP TR86-45, Rice University, 1988.

[AKPW83] Randy Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of control dependence to data dependence. In *Proceedings of the Tenth ACM Symposium on Principles of Programming Languages*, 1983.

[All83] Randy Allen. *Dependence Analysis for Subscripted Variables and its Application to Program Transformation.* PhD thesis, Rice University, 1983.

[AS79] Walid Abu-Sufah. *Improving the Performance of Virtual Memory Computers.* PhD thesis, University of Illinois at Urbana-Champaign, 1979.

[ASM86] Walid Abu-Sufah and Allen Maloney. Vector processing on the Alliant FX/8 multiprocessor. In *1986 International Conference on Parallel Processing*, 1986.

[Ban76] Uptal Banerjee. Data dependence in ordinary programs. Technical Report 76-837, Department of Computer Science, University of Illinois at Urbana-Champaign, November 1976.

[Ban79] Uptal Banerjee. *Speedup of Ordinary Programs.* PhD thesis, University of Illinois at Urbana-Champaign, 1979.

[BC86] Michael Burke and Ron Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction*, 1986.

[Ber66] A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, 15(5), October 1966.

[BS88] Michael Berry and Ahmed Sameh. Multiprocessor schemes for solving block tridiagonal linear systems. *The International Journal of Supercomputer Applications*, 2(3), Fall 1988.

[CAC+81] Gregory Chaitin, Marc Auslander, Ashok Chandra, John Cocke, Martin Hopkins, and Peter Markstein. Register allocation via coloring. *Computing Languages*, 6, 1981.

[Cal86] D. A. Calahan. Block-oriented, local-memory-based linear equation solution on the Cray-2: Uniprocessor algorithms. In *1986 International Conference on Parallel Processing*, 1986.

[Cal87] David Callahan. *A Global Approach to Detection of Parallelism.* PhD thesis, Rice University, 1987.

[CCK87] David Callahan, John Cocke, and Ken Kennedy. Estimating interlock and improving balance for pipelined architectures. In *1987 International Conference on Parallel Processing*, 1987.

[CD89] Chi-Hung Chi and Henry Dietz. Improving cache performance by selective cache bypass. In *Twenty-second Annual Hawaii International Conference on System Sciences, 1989*, 1989.

[Cha82] Gregory Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the SIGPLAN 82 Symposium on Compiler Construction*, 1982.

[CK88] David Callahan and Ken Kennedy. Analysis of interprocedural side effects in a parallel programming environment. *Journal of Parallel and Distributed Computing*, 5(5), October 1988.

[Coh73] W. L. Cohagen. Vector optimization for the ASC. In *Proceedings of the Seventh Annual Princeton Conferance on Information Sciences and Systems*, 1973.

[Den70] Peter Denning. Virtual memory. *Computer Surveys*, 2(3), September 1970.

[Den72] Peter Denning. On modeling program behavior. In *Proceedings of Spring Joint Computer Conference*, 1972.

[DMBS79] J. Dongarra, C. Moler, J. Bunch, and G. W. Stewart. *LINPACK Users' Guide.* SIAM, Philadelphia, 1979.

[Don88] Jack Dongarra. Performance of various computers using standard linear equations software in a Fortran environment. *Computer Architecture News*, 16(1), March 1988.

[GJG87] Dennis Gannon, William Jalby, and Kyle Gallivan. Strategies for cache and local memory management by global program transformations. In *First International Conference on Supercomputing*, 1987.

[GJMS88] Kyle Gallivan, William Jalby, Ulrike Meier, and Ahmed Sameh. Impact of hierarchical memories on linear algebra algorithm design. *Supercomputer Applications*, 2.1, Spring 1988.

[GKLS83] D. Gajski, David Kuck, David Lawrie, and Ahmed Sameh. CEDAR — a large scale multiprocessor. In *1983 International Conference on Parallel Processing*, 1983.

[GM87] C. E. Gimac and V. M. Milutinovic. A survey of RISC processors and computers of the mid-1980's. *Computer*, 20(9), September 1987.

[KKP+81] David Kuck, Robert Kuhn, David Padua, Bruce Leasure, and Michael Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the Eighth ACM Symposium on Principles of Programming Languages*, 1981.

[KL70] David Kuck and David Lawrie. The use and performance of memory hierarchies: A survey. *Software Engineering*, 1, 1970.

[Knu71] Donald Knuth. An empirical study of FORTRAN programs. *Software Practice and Experience*, 1(12):105–134, 1971.

[KS88] James Kuehn and Burton Smith. The HORIZON supercomputing system: Architecture and software. In *Supercomputing 88*, 1988.

[Kuc78] David Kuck. *The Structure of Computers and Computations*, volume 1. John Wiley and Sons, New York, 1978.

[Kuh80] Robert Kuhn. *Optimization and Interconnection Complexity for: Parallel Processors, Single-Stage Networks, and Decision Trees.* PhD thesis, University of Illinois at Urbana-Champaign, 1980.

[Kum87] Manoj Kumar. Effect of storage allocation/reclamation methods on parallelism and storage requirements. In *The 14th Annual International Symposium on Computer Architecture*, 1987.

[Lam74] Leslie Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2), February 1974.

[LS88] Bowen Liu and Nelson Strother. Programming in VS FORTRAN on the IBM 3090 for maximum vector performance. *Computer*, 21(6), June 1988.

[LYL87a] Roland L. Lee, Pen-Chung Yew, and Duncan H. Lawrie. Data prefetching in shared memory multiprocessors. In *1987 International Conference on Parallel Processing*, 1987.

[LYL87b] Roland L. Lee, Pen-Chung Yew, and Duncan H. Lawrie. Multiprocessor cache considerations. In *The 14th Annual International Symposium on Computer Architecture*, 1987.

[PBG+85] G.F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, K. P. McAuliffe W. J. Kleinfelder, E. A. Melton, V. A. Norton, and J. Weiss. The IBM research parallel processor prototype (RP3) introduction and architecture. In *1985 International Conference on Parallel Processing*, 1985.

[Ros88] Carl Rosene. Private Communication, 1988.

[SD88] C. Scheurich and M. Dubois. Concurrent miss resolution in multiprocessor caches. In *1988 International Conference on Parallel Processing*, 1988.

[Smi82] Alan Smith. Cache memories. *Computer Surveys*, 14(3), September 1982.

[Smi85a] Alan Smith. Cache evaluation and the impact of workload choice. In *The 12th Annual International Symposium on Computer Architecture*, 1985.

[Smi85b] Alan Smith. Disk cache — miss ratio analysis and design considerations. *ACM Transactions on Computer Systems*, 3(3), August 1985.

[SW88] Margaret Simmons and Harvey Wasserman. Performance comparison of the Cray-s and Cray-X-MP/416 supercomputers. In *Supercomputing 88*, 1988.

[Tha81] Khalid Thabit. *Cache Management by the Compiler.* PhD thesis, Rice University, 1981.

[TIF86] Remi Triolet, Francios Irigion, and Paul Feautrier. Direct parallelization of call statements. In *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction*, 1986.

[War84] Joe Warren. A hierarchical basis for reordering transformations. In *Proceedings of the Eleventh ACM Symposium on Principles of Programming Languages*, 1984.

[Wol82] Michael Wolfe. *Optimizing Supercompilers for Supercomputers.* PhD thesis, University of Illinois at Urbana-Champaign, 1982.

[Wol87] Michael Wolfe. Loop skewing: the wavefront method revisited. Technical Report ?, Kuck and Associates, Inc., 1987.