Improving the Performance of Cache Memories Without Increasing Size or Associativity

by

Nicholas P. Carter

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the degrees of

Bachelor of Science

and

Master of Science

at the Massachusetts Institute of Technology

June 1991

© Nicholas Carter, 1991

The author hereby grants to MIT permission to reproduce and
to distribute copies of this thesis document in whole or in part.

Author ______________________________

Department of Electrical Engineering and Computer Science
May 10, 1991

Certified by ______________________________

Thomas Knight
Thesis Supervisor

Certified by ______________________________

Daniel Prener
Thesis Supervisor

Accepted by ______________________________

Arthur C. Smith
Chair, Department Committee on Graduate Students

Improving the Performance of Cache Memories Without Increasing Size or Associativity

by

Nicholas P. Carter

Submitted to the
Department of Electrical Engineering and Computer Science

May 10, 1991

In Partial Fulfillment of the Requirements for the degrees of
Bachelor of Science
and
Master of Science

Abstract

The cache memories now used in many computers are sufficiently large that increasing their size or associativity is no longer a cost-effective means of increasing the performance of the machines containing those caches. Several alternative methods of increasing the performance of caches are examined, focusing on the impact of these methods on the performance of the IBM RISC System/6000 line of workstations. This is done through simulation of the RISC System/6000, using instruction traces taken from the SPEC benchmark suite.

A prefetching scheme is examined which prefetches data based on the likely behavior of load and store with update instructions. This scheme is found to degrade the performance of the RISC System/6000 slightly.

Load interruption and the use of a history table are studied. These methods attempt to reduce execution time by reducing the amount of data that is brought into the cache and never used. Both of these methods are found to be effective, with load interruption producing an average speed improvement of 2.9% over the base machine, while the use of a load history table produces speed improvements of 4.1–4.3%, depending on the size of the history table.

The final strategy which is examined is the use of a victim cache to improve the performance of a direct-mapped cache, allowing the substitution of a direct-mapped cache with a smaller access time for a set-associative cache of the same size, which allows a faster clock speed for the system. The use of a 64 kilobyte direct-mapped cache along with a victim cache of 2–32 entries is found to produce a .5–1.1% speed improvement over a system with the same clock speed containing a four-way, set-associative cache of the same size, suggesting that the use of such a cache is desirable if the cycle time or the hardware complexity of the machine can be reduced by doing so.

Keywords: cache, victim cache, prefetch, history table, load interruption

Thesis Supervisor: Thomas Knight
Title: Thesis Supervisor (Academic)

Thesis Supervisor: Daniel Prener
Title: Company Supervisor (IBM Research Division)

Contents

Chapter 1

Introduction

Throughout much of the history of computers, one of the major limiting factors on the performance of machines has been the speed with which data can be moved between the processor and the memory system. The need for a memory system that provides both large amounts of storage and fast access to memory at a reasonable cost has led to the design of multi-level memory systems, which combine different memory technologies to produce a system with storage as large as the largest element of the system, and speed approaching that of the fastest element of the system.

Multi-level memory systems can provide this combination of speed and size by taking advantage of the fact that the average time required for a memory reference in a multi-level storage system is $T_{avg} = P_{hit} * T_{hit} + P_{miss} * T_{miss}$, i.e. that the average time to access the memory is equal to the probability that the needed datum will be in the fastest memory multiplied by the time required to access that memory, plus the probability that the datum being accessed will not be in the fastest memory times the time required to access the datum if it is not in the fastest memory. While this formula only covers two-level memory systems, it is easily extensible to memory systems with more levels of storage. Examination of this formula makes it clear that, so long as the probability that the data being referenced is in the fastest memory is sufficiently high, the average access time of the memory system will be fairly close to the access time of the fastest memory.

Most modern computers use a three-level memory hierarchy. This usually consists of a cache memory, constructed of static random-access memory (SRAM), a main store, constructed of dynamic random-access memory (DRAM), and a third-level store, which is typically recorded on magnetic media. This allows large amounts of total system memory, since currently available magnetic disks have storage capacities in excess of 100 megabytes, combined with a fairly small average access time, as current cache memories have access times of under 50 nanoseconds.

Even with the use of multi-level storage, modern processors require higher bandwidth to memory than current memory systems can economically provide, making memory access a source of significant delay in many computer systems. Based on the formula for the average memory access time, there are two obvious ways in which the delay caused by the memory system could be reduced. These are to decrease the time needed to access one or more levels of the memory system, or to increase the probability that the desired data is contained in the faster levels of the memory system. Increasing the speeds of various memory systems is the subject of much ongoing research, but suffers from the fact that the designers of memory systems have two goals which often conflict with each other: the desire for increased memory capacity and the desire for increased memory speed. Designers of processors, on the other hand, are only concerned with speed, which tends to lead to processors increasing in speed faster than memory systems. Increasing the hit rate of each level of the memory system has been the subject of a great deal of research, especially in the case of cache memories, and this research has produced a number of methods that have been remarkably effective in doing so in the past. However, modern cache memories have reached the point where these methods are becoming less effective, making alternative methods of improving performance attractive.

1.1 Motivation

In the past, the most common technique for improving the performance of a cache was to increase the amount of data that could be stored in the cache, or to increase the *associativity* of the cache.

The associativity of a memory system refers to the number of places within the system that a given datum can be placed into. A *fully-associative* cache allows any datum to be placed anywhere in the memory; it must search all locations in the cache to determine if a specified datum is located in the cache. While this is the most versatile configuration, and provides the highest hit rate for a given amount of memory, it requires a large amount of hardware to examine each location in the cache in parallel if it is to be possible to access the cache quickly.

A *direct-mapped* cache restricts each datum to a single place in the cache, which is usually determined by examining the address of the datum. Direct-mapped caches are the simplest type of cache to implement, but usually have lower hit rates than fully associative caches of the same size. They can often be built with lower access times than caches with higher associativity, as restricting each datum to one location in the cache means that there is only one place to look for that datum. In addition, because direct-mapped caches do not need to perform as many comparisons as associative caches to determine if a datum is in the cache, they are simpler to design and build, and can fit more memory into a given area on a silicon chip. However, because direct-mapped caches only provide one possible location for each datum, their performance degrades significantly if two data that are used frequently conflict for the same cache line. This can cause direct-mapped caches to have very poor performance in some cases, even when the size of the data being used is much less than the size of the cache.

Set-associative caches represent a compromise between fully-associative caches and direct-mapped caches. They provide a small number of cache lines into which a given datum can be placed. The set of lines into which a given datum can be placed is called a *congruence class*. For example, a four-way, set-associative cache provides four different lines into which a given datum can be stored. Typically, the address of the datum is used to select the congruence class into which a given datum is to be stored, and then some other method is used to select which line in the congruence class the datum will be stored into.

An important characteristic of a cache with associativity greater than one is the *replacement policy* of that cache. The replacement policy of a cache determines which line of a congruence class is overwritten when other data is loaded into that class. The RISC System/6000, along with many other computers, uses a least-recently-used (LRU) replacement policy, in which the line that was least recently referenced is selected to be replaced. Other replacement policies which are sometimes used are FIFO, or first-in-first-out, in which the line that has been in the cache for the longest time is replaced, or random replacement policies, in which a line is selected at random from the congruence class to be replaced.

Set-associative caches usually have better hit rates than equally large direct-mapped caches, as larger numbers of data have to conflict for a given congruence class before interference occurs, but lower hit rates than fully-associative caches, because it is possible for interference to occur between data before the cache is completely full. Their access times are also usually between those of direct-mapped and fully-associative caches, as there are fewer possible locations for a datum than in a fully-associative cache, but more than are provided by a direct-mapped cache. Many contemporary cache designs are set-associative, as that design represents a good compromise between the desires for speed and hit rate. A few are direct-mapped; this cache design is typically used in cases where access time is more important than hit rate. Almost no modern data caches are fully-associative, as the extra hardware required to implement full associativity quickly is typically more expensive than enlarging a less associative design to a size that provides better performance.

Increasing the size or associativity of a cache design tends to improve the performance of that cache, but in an asymptotically decreasing manner; the larger or more associative a cache is, the less improvement in hit rate increasing the size or associativity of that cache brings. The more frequently a datum is used, the more benefit is gained by caching that datum, because the amount of memory delay that is eliminated by having the datum in the cache memory is proportional to the number of times the datum is referenced. Those data which are referenced most frequently are kept in even a small cache; increasing the size or associativity of a cache allows keeping less frequently used data in the cache. Therefore, the larger and more associative a cache is, the less benefit can be gained from increasing either the associativity or the size of the cache, because the most frequently used data will be stored in the smaller cache and the additional size or associativity will only allow the retention of less frequently used data, with a corresponding lesser amount of memory delay that can be eliminated by keeping that data in the cache.

Since increasing the size or associativity of a cache becomes less and less effective as caches grow in size and associativity, and increasing the size or associativity of a cache requires substantial amounts of hardware, it makes sense to look at other methods of improving the performance of caches.

1.2 The Proposed improvements

This thesis examines four methods of improving cache performance, three of which are aimed at increasing the performance of any cache, and the fourth of which is designed to increase the performance of caches with small associativities, allowing designers to take advantage of the shorter access times of those caches by reducing the delays caused by conflict over cache lines.

1.2.1 Prefetching

Caches take advantage of the locality of reference displayed by most computer programs by storing data that has been recently referenced in the cache under the assumption that that data, or data located near it in the memory, will be referenced in the near future. While this is usually an effective method of keeping the most heavily used data in the cache, it would also be desirable to have some method of predicting which data were about to be used, so that those data could be brought into the cache before they were needed, thus eliminating the need to wait for a datum to be brought into the cache from main memory the first time it is used.

This technique, called prefetching, suffers from one major drawback: its effectiveness is limited by the ability of the computer to predict which data will be needed in the near future. Every time the computer prefetches a line into the cache that is not used, it not only wastes the time that was required to bring that line into the cache, but it replaces a line that was in the cache and might have been used again with a line that is not used. For this reason, most simple prefetching schemes, such as prefetching the line that follows the line currently being accessed in the memory, tend to slow the computer down by "polluting" the cache with data that is never used.

Prefetching might be effective, however, if a more accurate method of predicting which data was going to be used in the near future could be found. This thesis explores the possibility of using specific instruction types as indicators of which data are likely to be used soon. The RISC System/6000, and many other machines, have instructions called *load/store with update* instructions. These instructions generate the memory address that they reference by taking the contents of a register, and adding some value to it. The result of this sum is then stored back into the original register. This instruction is usually used in cases where the program is stepping through data in memory, and therefore the probability that the data at the address contained in the register after the execution of this instruction will be used in the near future is very high. For this reason, it seems likely that prefetching the data at that location might improve the performance of the memory system.

Prefetching data in this manner would require some amount of extra hardware, to handle checking to see if the data to be prefetched was already in the cache, and to initiate a memory load to get it from the main store if it were not. However, the amount of hardware required to do this would be small compared to the amount of hardware required to increase the size or associativity of the cache, which would make prefetching attractive if it produced performance improvements comparable to those produced by enlarging the cache.

1.2.2 Interrupting Loads

Prefetching attempts to increase the performance of a cache by locating data which will be used in the near future and bringing it into the cache before it is needed. Another way to improve the performance of a cache would be to reduce the amount of data which is brought into the cache and never used. This would reduce the average time required to access the main store, and thus reduce the average access time of the memory system. A simple way of doing this would be to provide a mechanism by which the process of loading one cache line from the main store into the cache could be aborted to allow another memory access to begin. Since it is certain that the datum which caused the second memory access by being referenced is needed by the program, this method allows the computer to stop bringing data that might be used into the cache in order to bring in data that definitely will be used. This is made possible by the fact that the RISC System/6000 fetches data into cache lines in a circular manner. The first datum accessed is the one which caused the cache miss. The cache line is then read in sequential order until the end of the line is reached, at which time the computer "wraps around" to the beginning of the cache line if necessary to bring the rest of the line into the cache.

While this method of memory access was developed in order to allow the datum which caused a cache miss to be returned to the processor more quickly, it also ensures that the first datum returned from the main store is the only one which it is absolutely necessary to retrieve from the main store. Because of this, a cache line load can be interrupted at any time after the first datum has been returned from the main store, so long as some method is provided to determine which sections of a cache line are valid and which are not. On a machine which always read cache lines from the main store in the same order this would not be as effective, as it would be necessary to wait until the datum that caused the cache line miss was reached to interrupt the memory access, instead of being able to interrupt as soon as the first datum has reached the cache.

This modification would require implementing some method by which it would be possible to detect which portions of a cache line were resident in the memory, since any cache line load which was interrupted would result in a cache line which contained some valid data and some invalid data. One method of doing this would be to put "valid bits" in each cache line. These valid bits would be used to record whether various sections of a cache line had been read from the main store or not. When a cache line was referenced, the valid bit for the section of the line that contained the desired datum would be checked to determine if the datum contained in the cache was valid. If so, a cache hit would occur and be handled in the same manner as in the unmodified system. If not, a cache miss would occur, and a memory access initiated to read the desired data from the main store. The experiments described in this thesis assume this implementation, with each valid bit describing the state of an amount of data equal to the amount of data that can be brought into the cache in one transfer from the main memory.

Implementing this idea would require adding valid bits to each cache line, modifying the memory access hardware to handle loading lines from the main store into a cache line which already contains some valid data, and writing partially valid lines back to the main store if they are replaced in the cache. In particular, it would be necessary to provide some mechanism by which data could be brought into a line which had been partially loaded into the cache and then modified without corrupting the modified data. This could be done by adding "dirty bits" to each cache line, which would record whether or not each section of the cache line had been modified since it had been brought into the cache. The main memory access hardware would also have to be modified to allow one memory access to replace another.

Given that dirty bits would have to be added to each cache line to implement this modification, a possible extension of this idea would be to increase the resolution of the dirty bits such that there was one dirty bit for each section of the cache line that could be modified. On the RISC System/6000, it is not possible to modify memory in amounts smaller than one byte, so this idea would require the use of one dirty bit per byte. It would then be possible to allow store instructions which caused cache misses to simply place their data in a cache line without reading the cache line that their data was part of into the cache, since it would be possible to determine exactly which parts of a cache line had been modified. This would eliminate the delay caused by reading the cache line into memory for those cache lines which are written to and in which data other than that which was written was never read.

Since several of the SPEC benchmark programs modify a large number of lines that they never read, and it seems likely that a reasonable number of lines are written to and then only read in the areas that were written, it is possible that this could produce a significant speed improvement. Due to time restrictions, this extension was not studied in this research.

1.2.3 Load History Tables

Interrupting memory references attempts to improve performance by allowing the memory system to spend time retrieving data that is known to be needed instead of data which merely has some likelihood of being needed. Load history tables also attempt to reduce the amount of time that the memory spends reading data into the cache that will not be used, but by a different method. Load history tables operate under the assumption that many programs spend large amounts of time in loops which execute the same instructions over and over, and that the memory access patterns of a loop are similar from one iteration to the next. Based on these assumptions, it seems likely that if the cache line referenced by an instruction was not accessed sufficiently to justify bringing the entire line into the cache on one iteration of a loop, the cache line referenced by that instruction on a later iteration is also likely not to be referenced enough to be worth bringing into the cache completely.

In order to decide whether or not an instruction's past behavior warrants fetching the entire cache line the next time that instruction causes a cache miss, a load history table is maintained of the last n load and store instructions, with information as to how much of the cache line referenced by each instruction has been referenced since the instruction has been loaded. On the RISC System/6000, loading an entire cache line into the cache takes approximately twice as long as retrieving two separate data from the main store. Because of this, it makes sense to load the entire cache line if it is believed that enough of the cache line will be used to require two main memory references if the entire line is not loaded. In order to determine this, each entry in the table contains two bits of information in addition to the address of the instruction that the entry refers to: one bit which records whether or not the cache line that was loaded the last time that instruction causes a cache miss is still resident in the cache, and another bit that records, if the line is not still in the cache, whether or not enough data from that line was referenced to justify bringing in the entire cache line the next time that instruction caused a miss. In the experiments which were done to test this idea, it was assumed that the load history table was fully associative, with a least-recently-used replacement policy, as it was believed that the load history table would be fairly small, which would make the implementation of this configuration reasonable. Actual implementations of a load history table might use other configurations.

Whenever an instruction causes a cache miss, the load history table is examined. If an entry exists for the instruction, the residency bit of that entry is examined to see if the line is still resident. If it is, the amount of the line that has been referenced is examined, and the results of that used to decide whether or not loading the entire cache line is justified. If the line is no longer resident, the other bit is examined to make that decision. The memory system then either loads the entire cache line into the cache, or just the datum that is needed to allow the instruction to execute. Note that the method examined here does bring the single datum into the cache: it might be worthwhile to examine the consequences of just sending the required data to the processor instead of caching it.

Implementing a load history table would require several hardware modifications to a standard memory system. It would be necessary to implement the history table itself, and to provide the communications between the cache, the history table, and the processor that would be necessary to maintain the fields of the history table. It would also be necessary to provide valid bits in the cache, to handle the case where only one datum of a line is loaded, and “referenced bits” that would keep track of which parts of a cache line had been referenced to allow the load history table to determine whether or not to load the entire line. Finally, it would be necessary to modify the interface to main memory to allow cache lines to be either fully loaded, or to only have one datum loaded.

1.2.4 Victim Cache

The three ideas outlined above are intended to increase the performance of cache memories by reducing the amount of time spent waiting for main storage. Another way to increase the performance of a system containing a cache memory is to increase the speed of the cache memory, thereby allowing a faster cycle time for the main processor. For this reason, there has recently been a resurgence of interest in building direct-mapped caches, especially by designers of very high-performance machines, because these caches can be built with lower access times than equally large associative caches.

Unfortunately, direct-mapped caches have significantly lower hit rates than equally large associative caches, due to conflicts between data for cache lines. A possible way of overcoming this disadvantage, which was proposed recently by Jouppi in [6], is the *victim cache*. A victim cache is a small buffer, which Jouppi proposed would be made fully-associative, that is attached to the main cache, and into which all lines that are removed from the main cache to make room for new lines are placed. Lines are replaced using a LRU replacement scheme in the victim cache. If a cache miss occurs in the main cache, the victim cache is searched for the desired datum at the same time a memory reference to fetch the desired datum from the main store is begun. If the desired datum is in the victim cache, it is then brought into the main cache, which takes much less time than fetching the datum from the main store.

This proposal would improve the performance of a direct-mapped cache significantly in the worst case: a program which uses two lines of memory that need to be stored in the same cache line heavily. Instead of having to go all the way to the main store to get the needed data, it could be brought into the main cache from the victim cache. A significant amount of hardware would be required: the victim cache would have to be built, as well as the connections between the main cache and the victim cache to allow lines to be moved between the main cache and the victim cache without interfering with the ability of the main cache to get data from memory.

In the studies done for this thesis, it was assumed that the victim cache was fully-associative, that it was possible to move lines from the main cache to the victim cache without increasing the amount of time required to handle a cache miss, and that lines could be brought from the victim cache to the main cache in one cycle, meaning that a cache miss in the main cache that resulted in a cache hit in the victim cache would cause one cycle of delay. All tests that were done simulated the victim cache attached to a direct-mapped cache of the same size as the standard RISC System/6000 cache. An interesting follow-on to this research would be to study the effect of adding a victim cache to a more associative cache.

1.3 Methods

Since it was not practical to build hardware prototypes for each of the proposals listed above, it was necessary to simulate the behavior of the RISC System/6000 in order to determine the impact each of the proposed modifications would have on its performance. Simulators have the advantage of being easily modifiable, allowing changes to be made quickly to the behavior of the system, but suffer from the disadvantage that they are only as accurate as the assumptions used in developing them. Simulators also suffer from a lack of speed. The simulator that was developed for this project takes a few hours to simulate 100,000,000 instructions, which would take only seconds to execute on the actual hardware.

One consequence of the slowness of simulators is that it is necessary to select a comparatively small number of test cases to simulate, and hope that the results gained from the analysis of those test cases apply to most of the programs that will be executed on the actual hardware. Another problem with most simulations of computers, including the one used here, is that they only simulate the execution of a single process. This is unfortunate, as the context switching that many machines do in order to support the execution of multiple processes can have a significant impact on the performance of the machine, especially with regard to the performance of cache memories, as much of the data in the cache is no longer of use when a context switch occurs.

An advantage of simulators over building hardware prototypes is that it is much easier to gather information from a simulator, as the simulated machine state is easily accessible. If hardware prototypes had been built for the improvements studied, about the only information that it would have been possible to gain from them without a great deal of extra hardware to make measurements would be the running times of various programs with and without the improvements. In contrast to this, it was easy to program the simulator to gather statistics on the number and frequency of memory accesses, cache hits and misses, and translation look-aside buffer hits and misses, While this information might not prove especially useful in deciding whether or not these improvements should be implemented, it is useful to have.

While the speed penalty incurred by simulating the a computer over building hardware prototypes is severe, and greatly restricts the number and size of the test cases that can be run, the advantages gained in ease of modification and speed of implementation easily outweigh these disadvantages. By choosing a sufficiently large and diverse set of test cases, it should be possible to predict the behavior of these improvements with some degree of accuracy, at least enough to decide which of the improvements should be considered for implementation in hardware, and which should not.

Chapter 2

Previous Research

Cache memories have been the subject of a great deal of research. The early research on caches focused on showing that the use of a cache memory could increase the performance of a computer, and on determining how the cache memory should be organized. One of the major results of this research was the discovery that it was profitable in many cases to split the cache memory into separate instruction and data caches. This allows the instruction and data caches to be located near the sections of the CPU that use the data they contain. Also, since the instructions and data of a program are often in different portions of the memory, splitting the cache eliminates the interference between the instruction and data references, thus increasing the hit rates of both caches. In addition to these factors, most split instruction/data caches do not provide the ability to write data to the instruction cache, which simplifies the instruction cache greatly, although it makes writing self-modifying code much more difficult.

2.1 Cache Size and Associativity

A great deal of work has been done which attempts to characterize the relationship between cache size, associativity, and hit rate. Increasing the size and/or associativity of a cache has been shown to improve the hit rate of that cache, but in a decreasing manner [4] [1]. Each increase in size or associativity of a cache increases the hit rate of the system containing that cache, but each increase brings less improvement than the first. The studies that have been done have found that the incremental benefit to be gained from increasing cache size decreases substantially once the cache contains 32K-128K of memory, and that the benefits from increasing associativity drop off sharply once the cache is four-way set-associative, at least for single programs.

Analysis of the behavior of caches under multiprogramming conditions is much more difficult than analysis of single-program execution, as it is difficult to generate data on the memory reference patterns of operating systems. Since it is impractical to build hardware prototypes of caches in order to study them, most research is done using trace-driven simulation, in which traces of the memory references that occur during program execution are generated and used to simulate the execution of that program on the target architecture. Many researchers attempt to simulate a multiprogramming environment by interleaving traces from several different programs, to recreate the effect that context switches have on the contents of the cache.

Steven Przybylski, Mark Horowitz, and John Hennessy have done work [9] that quantifies the trade-off between cache size, associativity, and clock speed in determining the performance of a computer. Their work assumes a RISC processor capable of providing 25 MIPS peak performance, and shows how much of an increase in cache size or associativity is required to offset a given clock speed penalty incurred in building such a cache.

2.2 Prefetching

Another common topic in cache research is the use of prefetching to improve the hit rate of a cache by bringing data into the cache before it is needed by the program. In his paper [10], Alan J. Smith considers the impact of using a sequential prefetching scheme which prefetches the page of data immediately following any page which is referenced, and varies the page size from 32 bytes, which is typical of cache line lengths, to 1024 or more bytes, which is typical of the page size in the main stores of many computers. He also studies the impact of prefetching whenever a memory reference occurs versus only prefetching when a memory reference causes a miss.

His results show that prefetching in this manner can improve the hit rate of a memory when the memory is fairly large and the page size is fairly small. He finds that the hit rate improvement gained from prefetching is greater than the improvement gained from doubling the line length for those cases where prefetching results in an improvement. From this, he concludes that prefetching is primarily of use in cache memories, where the line sizes tend to be small.

Unfortunately, no work could be found that examined the effects of prefetching on the execution speed of a machine. This illustrates one of the major failings of a great deal of cache research: the assumption that any increase in hit rate leads to an increase in performance. This is not always the case. One example of this can be seen by examining the line size of a cache. For most caches, increasing the line size will increase the hit rate of the cache by bringing more data into the cache on a miss. However, this often does not increase the performance of a machine containing such a cache, as more time is spent bringing data into the cache that is not used than is saved by the increased hit rate.

2.3 Victim Cache

In his paper [6], Norman Jouppi suggested several methods for improving the performance of direct-mapped caches, focusing on the use of such caches in systems with very small cycle times. In such systems, direct-mapped caches are desirable in spite of the fact that they have lower hit rates than associative caches, because the shorter access times of direct-mapped caches make possible greater performance than the same system would have with a slower associative cache. In order to make up for the lower hit rate of direct-mapped caches, Jouppi proposed the use of a small, fully-associative buffer in combination with the direct-mapped cache.

Jouppi proposed two different configurations for this buffer, which he called *miss cache* and *victim cache*. A miss cache would receive all cache lines that are fetched from the main store at the same time as the main cache, and have a LRU replacement policy. A n-line miss cache would therefore contain the last n lines that were fetched from the main memory. Whenever a memory reference occurred, the main cache and the miss cache would be searched in parallel. If a miss occurred in the main cache that resulted in a hit in the miss cache, the miss cache could return the line to the main cache and the processor with only one cycle of delay over the time taken for a cache hit. This would substantially improve the performance of the memory system in the case where some number of frequently-used data conflict for the same cache line, as these data would be kept in the miss cache, and only one cycle of delay would occur when one of the data that were not in the cache were referenced. Since the miss cache would not be expected to return its data in a single cycle, it would be possible to build it to be fully-associative without increasing the cycle time of the machine.

The victim cache extends this idea by caching lines which are thrown out of the main cache. Because of this, no data is duplicated in the victim cache that is contained in the main cache. This causes a system with a victim cache to have performance that is at least as great as a system containing an equally large miss cache, as all of the data that is contained in the cache with a miss cache is contained in the cache with a victim cache, but there is some amount of space left over in the victim cache, because it does not contain any data that is in the main cache, that can be used to store data that is not in the cache with miss cache.

Both of these methods were found to eliminate a substantial percentage of the cache misses which were caused by conflicts for cache lines. This suggests that a direct-mapped cache with a victim cache might have performance comparable to that of a set-associative cache of the same size, which would make such a cache an attractive replacement for a standard set-associative cache if the cycle time of the machine could be reduced by going to a direct-mapped cache. Since a direct-mapped cache is much simpler to implement and takes up much less chip space than an equally large associative cache, the use of these caches might be desirable even if the cycle time of the machine could not be reduced by doing so, as the time and chip space saved could be used to improve the system in other ways, or to lower its cost.

2.4 Other Ideas for Improving Cache Performance

Chi-Hung Chi and Henry Dietz have studied [2] the idea of using compile-time analysis of the behavior of a program do determine if it is worthwhile to bring a given datum into the cache when it is referenced, bypassing the cache and returning the datum directly to the processor if it is not. They find that bypassing the cache can produce significant speed improvements, especially in the case where there is substantial conflict over one or more cache lines.

Gary Hoffman has explored [5] the use of an adaptive cache, which would decide how much data should be brought into the cache on a cache miss based on the recent behavior of the cache. If the hit rate is low, indicating that cache lines are not being used heavily, his system brings small amounts of data into the cache, making the cache behave as if its line size were very small. If the hit rate is high, larger amounts of data would be brought in, simulating the use of long cache lines. His paper describes the use of a "bang-bang" cache controller, which either loads all of a line or just the datum that causes a miss, but the idea could well be extended to a system that would select the amount of data to be loaded on a cache miss to be anywhere from just the datum being requested by the processor to the entire line length, depending on the current behavior of the program. He suggests that adaptive caching would make possible the use of much longer cache lines, allowing larger caches to be placed on a given chip, as the adaptive cache would only use the entirety of the lines when it was beneficial to do so, and would shorten the lines when short cache lines would be beneficial.

Chapter 3

The RISC System/6000

The IBM RISC System/6000 is a superscalar machine, capable of executing up to four instructions per machine cycle. This allows the RISC System/6000 to achieve performance exceeding that of many machines with shorter clock cycles by executing multiple instructions in parallel. The RISC System/6000 also incorporates a sophisticated branch-handling scheme and short pipelines to minimize the impact of branches on execution speed, a strategy which allows it to outperform some supercomputers on tasks which do not vectorize well.

3.1 Hardware

The CPU of the RISC System/6000 can be divided into four separate sections: the instruction unit, the fixed-point unit, the floating-point unit, and the data unit. The instruction unit fetches instructions from memory and dispatches them to the other units. Branch instructions are resolved in the instruction unit, reducing the delay caused by such instructions. The instruction unit contains an eight-kilobyte instruction cache, from which it can read up to four instructions per cycle, sufficient to keep up with the peak processing rate of the CPU. In addition to the instruction cache, the instruction unit contains a FIFO instruction prefetch buffer, which is used to hold instructions which have been fetched from the instruction cache but not yet dispatched or executed. It also contains a 32-bit condition register that allows the results from eight independent compare operations to be stored. This register allows the result on which a conditional branch will depend to be computed well in advance of the branch instruction, reducing the delay caused by the branch.

Each machine cycle, the instruction unit fetches up to four instructions from the instruction cache into the instruction prefetch buffer, depending on the amount of space in the buffer and the number of instructions available in the instruction cache. The first four instructions in the buffer are then examined, and divided into three classes: branch instructions, instructions which manipulate the condition register (such as logical operations on condition register fields), and instructions which are to be dispatched to the fixed- and floating-point units. If the first four instructions in the buffer are of the correct types, the instruction unit dispatches two instructions each cycle to the other units, and resolves two instructions internally: one condition-register and one branch instruction. If the first four instructions in the buffer are not of the correct types, the instruction unit handles as many instructions as possible without violating the instruction order of the original program.

Resolving branch instructions in the instruction unit allows "zero-cycle" branching: if an unconditional branch or a conditional branch that branches on a condition register that has already been set is encountered, it is possible for the instruction unit to resolve the branch without delays in the instruction streams of the other units. The condition register incorporates a locking feature which is used to mark condition register fields when an instruction that modifies that field is dispatched, preventing another instruction which reads that field from executing until the instruction which locked the condition register field has completed execution. This allows some instructions which reference the condition register to be executed earlier in the pipeline than others, and prevents errors which might be caused by instructions reading the contents of the condition register at incorrect times.

In addition to resolving branches in the instruction unit, the RISC System/6000 dispatches instructions along the "not taken" path for conditional branches which it cannot resolve at the time the branch reaches the instruction unit. Because of this, branch delays are only caused by conditional branches which cannot be resolved at the time they reach the instruction unit, and which take their branch. This substantially reduces the delay incurred by branching as compared to most other systems.

Branch delays are further reduced by the use of the *branch-on-count* instruction. This instruction allows a special register in the instruction unit to be loaded with the number of times that the branch-on-count instruction is to take its branch. Each time a branch-on-count instruction is executed, the branch is taken if the value of this register is not zero, and the contents of the register are decremented by one. If the contents of the register are zero, the branch is not taken. This instruction allows the total elimination of branch delays in loops whose number of iterations can be determined prior to the first execution of the loop.

The fixed- and floating-point execution units both receive the same two instructions from the instruction unit each cycle. These instructions are placed in a four-instruction buffer in each unit; the contents of both buffers are kept the same to allow precise interrupts. These buffers allow instructions to be dispatched in a slightly non-ideal manner without impacting performance. Without these buffers, it would be necessary to dispatch one fixed-point and one floating-point instruction every cycle in order for the instruction unit to dispatch two instructions every cycle. With these buffers, it is not necessary to do so, as instructions can be placed in the buffers if they cannot be executed immediately, although it is still important to try to dispatch equal numbers of fixed- and floating-point instructions for maximum speed.

The fixed-point unit is a fairly straightforward pipelined processor. The execute stage of the pipeline only requires one cycle. Sufficient pipeline bypasses have been implemented that no delay is incurred if dependent fixed-point instructions are dispatched on consecutive cycles, except in the case of load instructions. Even if a cache hit occurs, load instructions require a second cycle to get their data from the cache, so one cycle of delay is incurred if an instruction immediately follows the load instruction that fetches the data it needs. The compiler attempts to avoid this penalty by scheduling an independent fixed-point instruction between a load instruction and the first instruction that uses the results of that load.

The floating-point unit is more complicated. The execute stage of its pipeline requires two cycles and is not bypassed. This means that it is possible to execute floating-point instructions at the rate of one instruction per cycle, so long as none of the instructions depend on the results of the instruction which immediately proceeds them. One cycle of delay occurs if the floating-point processor attempts to execute two dependent instructions on consecutive cycles.

A key feature of the floating-point unit is the multiply-add fused, or MAF instruction. This instruction allows the computation of $(\pm A * \pm B) \pm C$ once per machine cycle, provided that no dependencies exist between consecutive operations. This provides for extremely good performance on matrix operations, on which this instruction can be used heavily, and allows for a peak execution rate of two floating-point operations per machine cycle, giving the RISC System/6000 extremely good floating-point performance for its price.

Floating-point load and store instructions are executed in both the fixed-point and floating-point units. Address calculations for both fixed-point and floating-point memory references are performed in the fixed-point unit, allowing the translation lookaside buffers to be located in that unit. A system of interlocks allows synchronization of the two units in order to make sure that address computations go with the correct memory references. This system of interlocks also allows *precise interrupts*, meaning that the particular instruction which caused an interrupt can be determined, and the processor state adjusted to reflect the execution of all instructions occurring before that one in the instruction stream, and only those instructions. The maintenance of precise interrupts is an important feature of the RISC System/6000, as it makes it much more feasible for the programmer to treat the machine as if it were executing instructions in a purely sequential manner, instead of the parallel manner in which they are really executed, making programming much easier.

The data unit consists of either two or four identical DCU chips, depending on the model of the machine. Each of these chips contains 16 kilobytes of data cache, for a total data cache of either 32K or 64K. The data cache chips contain bypasses which allow the desired data to be returned to the processor on a cache miss before the entire line has been loaded into the cache [3]. This is an important feature, as the RISC System/6000's data cache lines take eight cycles to load after the initial memory latency of eight cycles has expired; bypassing the cache lines in this manner halves the wait time between a cache miss and the use of the required data. Similar buffers exist for store operations that allow the processor to continue execution on the cycle immediately following a store operation, even if a cache miss has occurred.

The superscalar architecture of the RISC System/6000, combined with the methods that have been used to reduce branch delays and increase floating-point performance, produces remarkable performance for many applications. While the RISC System/6000 line cannot match the performance of vector supercomputers for those applications which vectorize well, the RISC System/6000 model 550 has been shown to outperform a Cray-1 on many applications.

3.2 Software

The RISC System/6000 was designed with the assumption that almost all applications for it would be written in high-level languages, allowing the design of a machine that would be extremely difficult to program efficiently in its machine language. Because of this, it was imperative that the compilers for the RISC System/6000 generate extremely efficient code. The design of the RISC System/6000's compilers were done in concert with the design of the architecture; one of the goals of the architecture design was to make it easy for the compilers to produce extremely effective code for the hardware.

The RISC System/6000's compilers perform several optimizations that are specifically tailored to the configuration of the hardware, in addition to more traditional optimizations such as common subexpression elimination and register allocation [8]. Once the preliminary code generation section of the compiler has determined which instructions need to be executed to produce the correct output, the compiler re-orders these instructions to increase the efficiency with which they can be dispatched to the execution units, while ensuring that the re-ordered program will produce the same results as the original.

Another optimization takes advantage of the fact that the only branches that cause a delay to the RISC System/6000 are conditional branches which take their branch. Unconditional branches do not cause a delay, so long as they do not result in a cache miss in the instruction cache, and conditional branches which fall through do not cause a delay, as the RISC System/6000 executes instructions along the fall-through path while it is performing the calculations to determine if the branch is to be taken or not. Because of this, instruction loops can be restructured to operate more efficiently. Traditionally, loops are written by following the body of the loop with a conditional branch back to the start of the loop. On the RISC System/6000, this would cause a delay every time the jump back to the start of the loop was executed, because the computer would not know that it would be jumping back to the start until the conditional branch had completed execution, and would conditionally execute the instructions which follow the loop while waiting for the conditional branch to be evaluated.

This delay is avoided by placing the conditional branch at the start of the loop, and an unconditional branch at the end of the loop which branches back to the conditional branch, because the body of the loop will be conditionally executed while the conditional branch is being evaluated. It is sometimes necessary to put an unconditional branch before the conditional branch to ensure that the body of the loop is executed at least once, but unconditional branches do not cause any delay, so this is not a problem. With this scheme, the conditional branch only causes a delay when the loop terminates, as opposed to causing a delay except when the loop terminates, as the original scheme did.

The effectiveness of the compilers for the RISC System/6000 means that it will hardly ever be necessary to program this machine in its native language, and that programs written for the RISC System/6000 in high-level languages will run efficiently. This is important, as other superscalar processors have suffered greatly from the difficulty of programming such a complex architecture.

Chapter 4

The Simulator

In order to study the impact that modifications to the memory system would have on the speed of the RISC System/6000, a simulator was needed that would accurately predict the execution time of programs when executed on the current architecture, and which could easily be modified to reflect the proposed changes to the memory system. After study of the simulators available within IBM at the start of this project, it was decided that writing a simulator from scratch would be the most effective way to guarantee both the availability of the necessary tool, and the accuracy of its measurements. The pre-existing simulators that were studied either did not model the memory system sufficiently accurately for this research or were in the process of being implemented, and it was undesirable for the completion this research to be contingent on the work of another person, especially one who was not under the time constraints imposed by having to write a thesis.

4.1 Design

The simulator that was implemented is what is called a "timer" in IBM parlance, meaning that the structure of the simulator mimics the organization of the machine being simulated, and predicts the execution time of a program by tracking the flow of instructions through a pipeline designed to emulate the pipeline of the actual CPU. Initially, it had been planned to write a simulator which would mimic the organization of the RISC System/6000 to a much smaller degree, computing the execution time of each instruction individually and using that information to calculate the execution time of the entire program. This design was eventually scrapped in favor of the timer-style simulator because the other design required so much state information to accurately compute execution times that it was easier to encode the state information in a model of the processor than to keep track of all the factors that influence execution time explicitly. In addition to this, a timer-style simulator is easier to modify to reflect changes in the machine organization, as the structure of the program mimics the structure of the machine. Simulating changes to the machine organization then involves changing the organization of the program so that it reflects the proposed machine, which is much easier than modifying a program whose structure does not mimic the machine being modeled.

In order to simplify the simulator and increase its speed, it was decided that the simulator would take an instruction trace of the execution of a program as its input, instead of attempting to simulate the execution of actual programs. The generation of traces was not a problem, as a tracing program, written by Ju-Ho Tang at IBM Research Division, was available. The use of instruction traces simplified the simulator greatly. Since it was not necessary to know the results of executing each instruction, only the time taken by each, it was possible to treat instructions with similar characteristics as the same, reducing the complexity of the simulator. It was also not necessary to keep track of the contents of memory and registers, as these would only have been needed to compute branch targets and memory reference addresses, which were available in the instruction trace.

The operation of the RISC System/6000 is simulated on a cycle-by-cycle basis. Each cycle, the main procedure calls several other procedures, which simulate the operation of the stages of the RISC System/6000 pipeline. A data structure, called *mach_state*, is passed from procedure to procedure to record the changes in the state of the machine. The simulator simulates the fetching of instructions from the instruction cache each cycle, calling a procedure to read data from the instruction trace and parse it into a more useful format when necessary. The progress of instructions through the pipeline is simulated, along with the various interlock signals necessary to control a pipelined superscalar processor.

In order to allow modifications to the memory system to be simulated more easily, references to memory were handled by a small number of procedures. This controlled interface to the memory system allowed variations to the memory to be written very easily and selected through conditional compilation, as the memory reference procedures could be modified to reflect the modification without having to modify the rest of the program. The only time that it was necessary to modify other parts of the program to simulate a modification was when prefetching on load/store with update instructions was simulated. In that case, it was easier to modify the code that simulated those instructions to handle the prefetching, rather than having the memory reference procedures detect whether or not the instruction that caused the reference was a load or store with update instruction, and prefetch if necessary.

In addition to the number of cycles taken to execute the instructions in a trace, the simulator generates data on the frequency of memory references, cache misses, and TLB misses. Data is also gathered on the length of residency of a cache line, the length of time a line remains resident after its last reference, the number of times a line is referenced before it is removed from the cache, and the amount of data that is referenced within each line.

4.2 Implementation

The implementation of the simulator was complicated by the complexity of the RISC System/6000's processor. Because the RISC System/6000 is a superscalar machine, it was necessary to make the simulator, which simulates the various stages of the pipeline in serial, act like the entire pipeline was being executed in parallel. This task was made much easier by the fact that that the various stages of the pipeline could be ordered based on the order in which an instruction traverses the pipeline, and that, once an instruction is past a given stage of the pipeline, that stage of the pipeline is no longer able to affect the execution of that instruction. Thus, communication between stages of the pipeline only occurs in the direction opposite to the flow of instructions. This made the simulation of the fixed-point and floating-point pipelines much easier. Unfortunately, however, it is possible for the two pipelines to influence each other in an inherently parallel manner. This communication proved fairly difficult to implement, as it required keeping track of some of the interlock signals for more than one cycle, because signals generated by one processor on a given cycle should not influence the other processor until the next cycle.

Another area that proved difficult to model was the communication between the execution units and the instruction unit. The instruction unit reads instructions from the instruction cache, and dispatches instructions to the fixed-point and floating-point units. It is possible for one of the instruction units to encounter a situation which halts its pipeline while the other unit is free to continue execution. Making sure that all instructions were executed exactly once proved to be a challenge, requiring some relatively complex interlock coding to prevent the instruction unit from dispatching new instructions before both of the execution units were ready, while also preventing the execution units from reading the dispatched instructions more than once. The final solution was a handshaking scheme where each processor sets a flag when it is ready to receive new instructions, and then waits until that flag has been cleared to read the dispatched instructions. The instruction unit only dispatches instructions when both flags have been set, and clears them when it has done so.

Debugging the simulator proved to be extremely difficult. Due to the nature of the program, it was not possible to develop an exhaustive test suite that could be guaranteed to exercise all of its features. In addition, the manner in which data flows through the simulator makes it difficult to guarantee that all memory being used is freed when no longer needed. Since the simulator creates a data structure for every instruction which is not of use once that instruction has passed through the simulator, it was essential that all of the memory that was used be freed when no longer needed, as there was not enough memory available to store the data structures for all of the instructions in a trace.

Another major difficulty in debugging the simulator was the fact that errors in the program often did not manifest themselves until significantly after the error had occurred. One of the most common errors was duplication of a pointer in the pipeline in a manner which was not intended. This tended to result in the memory pointed to by that pointer being freed while it was still in use, and would usually cause the simulator to halt before it was supposed to. Fixing this kind of error required the use of data-gathering statements to determine where the pointer duplication had occurred. Similarly, the most effective method of tracking down memory leaks was to insert statements that would print out the address of every memory allocation and release, making it possible to determine which sections of the program were allocating memory that was not freed.

For most of the ideas being studied, it was possible to write the code that simulated each modification in a manner which closely resembled the hardware that would have to be built to implement the idea. For example, in simulating the load history table, a table was implemented in software which contained a number of instruction addresses and the requisite information to be associated with each instruction, which very strongly resembled the manner in which a hardware implementation of a load history table would be done. The one exception to this was the section of the program which simulated prefetching based on the load/store with update instruction. These instructions come in two forms: one in which the offset to be added to the contents of a register to generate the effective address for that instruction is a constant which is encoded in the bit pattern of the instruction, and one in which the offset is taken from a register.

This posed a problem, as the simulator being used did not keep track of the contents of registers. Since the simulator was trace-driven, it was not necessary to know the results of any of the instructions in order to determine the path taken through the instructions by the program, as this was recorded in the trace being used. This meant, however, that it was not possible to easily determine the address to prefetch from for load/store with update instructions that took their offset from the contents of a register. For those instructions which encode their offset as part of the instruction word, it was quite simple to determine the location to be prefetched from by simply adding the offset to the address that was referenced by the instruction, but this was not possible if the offset was taken from the contents of a register.

The solution to this problem was to create a table of load/store with update instructions, which contained the address of all of the load/store with update instructions which took their offset from a register that the simulator had encountered on that run. Contained with each instruction was the address that that instruction had referenced the last time it was executed. This allowed a prediction to be made about which data should be prefetched on the second and subsequent times that instruction was encountered, by subtracting the address referenced on the previous execution of the instruction from the address referenced on this execution of the instruction to get the offset, and adding that offset to the address currently being referenced to get the predicted target address of that instruction the next time it was executed. While this method does not resemble the way in which an actual hardware implementation of this idea would work very strongly, it should give reasonably accurate results, except in the case where a load/store with update instruction is contained within a loop which is executed a small number of times, as the first time that an instruction is encountered would be a large fraction of the total number of times it would be encountered. This should not significantly affect the overall results, however, as an instruction which is only executed a small number of times makes only a small contribution to the running time of a program, and, if prefetching proved to be beneficial, an actual hardware implementation could be assumed to be at least as effective as the software simulation, as the hardware would have more opportunities to prefetch than the software. In addition to this, load and store with update instructions that took their offset from a register were very rare in the traces used for this work, so the lack of accuracy in the simulation of these instructions should have almost no impact on the overall results.

The relatively simple structure of the simulator made it extremely fast. Over 4,000 instructions could be simulated per second, making it possible to run extremely large simulations in a reasonable amount of time. The speed of this simulator, combined with the relatively large number of computers available at IBM Research Division, allowed a total suite of approximately 17,000,000,000 instructions to be executed in gathering data.

4.3 The Instruction Traces

Since the SPEC benchmark programs are commonly used as a tool for designing and comparing computers, it was decided to take the instruction traces that would be used in this research from these programs. Because the SPEC programs were chosen to be representative of a typical scientific or engineering workload, it was felt that results gained from experiments done with these traces would be valid in predicting the effectiveness of the proposed modifications on the execution times of real-world applications.

Approximately 100,000,000 instructions were traced from each of the ten SPEC programs. This gave a total test suite of about 1 billion instructions, which was hoped to be a large enough sample that the results gained from experiments would be useful in the design of future computers. It was possible to run an experiment through this test suite in 1-2 days, using five RISC System/6000 computers.

Chapter 5

Results

The program traces used in gathering the data reported here did not contain the same number of instructions. In order to reduce the impact that the varying trace length has on the results, all results have been normalized to the performance of a base machine, and expressed as a percentage change in the performance of this base machine. The base machine used here was the current RISC System/6000 cache, which contains 64K of memory, has 128-byte cache lines, and is four-way set-associative, with a few minor improvements to the cache which was actually built.

These minor improvements consist mainly of making the data stored in the cache accessible while the cache is fetching data from the main store to satisfy a cache miss. This facility was included in the original design of the RISC System/6000's cache [3], but was not implemented. The current RISC System/6000 is only able to access data in the line containing the datum which caused the cache miss while a miss is being handled. Another minor improvement, which again reflects a discrepancy between the design and implementation, is the addition of the ability to have both one load miss and one store miss outstanding without halting the machine. The current RISC System/6000 stops and waits for the first cache miss to be satisfied whenever a reference is made to any line other than the one containing the datum which caused the miss. The machine as designed would contain sufficient buffers that it would be possible to continue execution during the resolution of both a load instruction that caused a cache miss and a store instruction that caused a cache miss, so long as none of the data in the lines which caused misses was needed. It would only be necessary to stop and wait if a second load or store miss was encountered while a first miss of that type was being handled, or if an instruction which depended on data which was not yet available was encountered. Simulations run during this research indicate that these modifications would result in approximately a 3% improvement in performance over the current machine, so it is reasonable to assume that the hardware will be brought into line with the

Doubling Cache Size from 64K to 128K	.2%
Doubling Cache Associativity from 4-way to 8-way	.05%
Making the Cache Direct-Mapped	-8.3%

Table 5.1: Improvement in Performance Achieved by Changing the Configuration of the Cache

design before any more significant modifications to the cache are made.

Unless otherwise indicated, the average impact of a modification was generated by normalizing the number of cycles taken to execute each trace on a machine containing the given modification to the number of cycles needed to execute the same trace on the base machine, adding all of the normalized execution times together, and dividing by the number of traces used. Also, whenever a percentage change is given, it represents a percentage of the time taken by the base machine. This will hopefully produce an average value which is meaningful, in that no one trace either dominates the average or has no impact on the average. Also, percent changes should be comparable, since they are all percentages of the execution time of the base machine. Since the traces were taken from the SPEC benchmark programs, which are felt by some to be a reasonably representative workload for workstations like the RISC System/6000, the average change in performance will hopefully be a reasonable indicator of how each of the modifications would impact the actual performance of a workstation which implemented the modification.

5.1 Changing the Configuration of the Cache

In order to provide reference points for some of the proposed extensions to standard cache design, three sets of simulations were run with slight modifications to the configuration of the cache in the RISC System/6000. Simulations were run with the cache size doubled from 64 kilobytes of memory to 128 kilobytes, holding the line size and associativity constant. Another set of simulations was run with the cache associativity doubled from four-way to eight-way set-associative, holding the amount of memory in the cache and the line size constant. These two sets of simulations were run to provide an indicator of how the proposed modifications to cache design compare to traditional methods of increasing cache performance.

A third set of simulations was run using a direct-mapped cache of the same size as the current RISC System/6000's cache, holding the line length and amount of memory in the cache constant. This was done to provide an indicator of the effectiveness of the victim cache in reducing the performance impact of using a direct-mapped cache instead of a set-associative cache.

Less than 1% average improvement in execution time was seen when the cache size or associativity was doubled. Table 5.1 shows the improvement gained through increasing the size or associativity of the RISC System/6000's cache. The size of the improvement was remarkably consistent across the ten program traces that were used. The largest improvement in performance seen by doubling the size of the cache was 1.2 percent, and the largest performance seen by doubling the associativity of the cache was .2 percent, so it does not seem that increasing the size or associativity of the current cache is worthwhile in a single-program environment. Increasing the size of the data cache might still be worthwhile, as most computers like the RISC System/6000 are used in a multi-programmed mode, where multiple programs are executed in rotation to produce the illusion that the machine is executing multiple programs at one time. In this mode of operation, increasing the cache size might be more effective than these results would indicate, as it might be possible to keep the working sets for more than one program in the cache at one time and thus reduce the delay introduced by context switches, as it would not be necessary to load the working set for each program back into the cache when that program's turn came to be executed.

These results agree with those reported by previous researchers, which indicated that the benefit to be gained from increasing the size of a cache drops off significantly once the cache has reached the 32K-128K size range, although the degree to which this occurs depends on the ratio between the time required to satisfy a cache hit and the time required to satisfy a cache miss. Similarly, previous research predicts that increasing the associativity of a four-way set-associative cache tends to have very little impact on the performance of that cache, so it is not surprising that very little benefit was gained by increasing the associativity of the cache.

Changing the cache from four-way set-associative to direct-mapped introduces a performance penalty of 8.3%. This is reasonable, as set-associative caches, even those with very small associativities, tend to be much more effective than direct-mapped caches of the same size. It is interesting to note that two of the traces suffered much greater penalties than the others when going from a set-associative to a direct-mapped cache. These traces were the ones which were taken from the "dnasa7" and "tomcatv" programs in the SPEC benchmark suite. This is not unexpected, as it is known that the pattern in which programs access data has a great deal to do with how much associativity affects the running time of that program, as programs which access data in a way which does not cause conflicts for cache lines are unaffected by changes in associativity, while programs which do cause such conflicts see a much greater change in performance when the associativity of the cache is changed.

5.2 Prefetching on Load/Store With Update

As was predicted, using the target addresses and offsets of load and store with update instructions to predict which data should be prefetched does a very good job of prefetching data that is eventually used by the program. These instructions tend to occur in loops, so predicting that the address generated by adding the offset to the current data address will be referenced on the next iteration of the loop results in prefetching useful data on all but the last iteration of the loop. Oddly enough, however, the simulations that were run indicated that the average running time of the traces was increased by this modification. The average running time for all the traces was increased by 3.9% when prefetching was applied.

The simulation results were very unbalanced, however. The trace taken from the "matrix300" program had its running time increased by 39.6%, while most of the other traces had their running times changed by less than 1%. In fact, the average running time of the traces other than the one taken from the "matrix300" program decreased slightly, although the difference was less than one percent.

The size of the increase in the running time of the matrix300 trace can be explained by considering the type of program it is taken from. The matrix300 program is a relatively naive matrix manipulation algorithm that operates on a 300x300 matrix, which is too large to fit in the data cache of the RISC System/6000. Because of this, it is reasonable to assume that any lines that are prefetched into the cache would be thrown out to make room for other data before they are used, and thus any time spent getting that line would be wasted.

Given the high accuracy rate of the predictions made by this method, it is strange that the execution time of the other nine traces is not improved by a larger amount. In fact, only a small number of the traces ran faster when this prefetching scheme was used. While cache pollution could cause this effect, it seems unlikely, as this prefetching scheme seems to be extremely accurate in its predictions; the data that is prefetched has an extremely good chance of being used.

A possible explanation for this could come from the fact that prefetching can increase the delay period between the time that a cache miss occurs and the time that the next cache miss can be satisfied by causing one instruction to access two lines of the main store. If a given instruction causes cache misses on both its initial reference and the prefetch, it takes a total of 32 machine cycles to complete the two memory references caused by that instruction. This means that another reference to the main store cannot begin until that time has elapsed. If cache misses occur reasonably frequently, this effect could increase the amount of time that the CPU spends waiting for the memory system, by transforming some pairs of cache misses into single cache misses that take longer to handle. If several cycles would have elapsed between each of the original cache misses and the next cache miss, this could waste time by causing the cache miss which follows the prefetching miss to wait for both of the misses caused by that instruction to complete before it can be resolved.

Prefetching data into the cache before it is needed seems like a good way to increase the speed of a computer, but these simulations suggest that it is not effective on the RISC System/6000, even when an extremely accurate method of determining which data should be prefetched is used. Perhaps the solution would be to implement a prefetch instruction, which would bring a line into the cache, but would not load any data into a register. This would allow the programmer or compiler to determine when prefetching was desirable and only prefetch in those cases. It might also be useful to combine prefetching with some of the other modifications studied in this work, which attempt to reduce the amount of useless data brought into the cache. Load interruption, in particular, might improve the performance of this scheme by allowing succeeding cache misses to interrupt data being prefetched, thus reducing the delay caused by combining pairs of cache misses into a single miss.

5.3 Load Interruption

The results of the simulations indicate that allowing cache misses to be interrupted in order to satisfy other cache misses produces an average speed improvement of 2.9% over the base machine. All but one of the traces showed a performance improvement of at least one percent, and the largest improvement was 7.6%. One trace, taken from the "dnasa7" program, took longer to run when this modification was made, taking 3.6% longer to execute than when executed on the base machine.

This illustrates a vulnerability of this modification. The idea behind load interruption is that, because of locality of reference and the fact that the RISC System/6000 loads the datum that caused the miss first, it makes sense to allow cache misses to interrupt previous misses once the datum that caused the miss has been brought into the cache. Since it is assumed that the data which are brought into the cache last are the ones which are least likely to be used in the near future, it is likely that the execution time of the program will be reduced by allowing another cache miss, which will retrieve data which is known to be needed by the program, to take precedence over bringing data into the cache which may or may not ever be needed. Unfortunately, there are times when this belief is incorrect, and allowing memory references to be interrupted makes it necessary to go back to the main store for the data which was not brought into the cache, which causes delay because of the large start-up time for memory references.

Clearly the idea of determining which data in a cache line are not going to be used by the program and avoiding bringing those data into the cache has merit, as shown by the speed improvement in all but one of the traces. However, there is also evidence that there is room to improve the accuracy with

Table Size	Performance Improvement
256 entries	4.3%
64 entries	4.3%
16 entries	4.1%
4 entries	4.3%

Table 5.2: Performance Improvements Achieved Through the Use of a History Table

which this determination is made, and thus improve performance.

5.4 Load History Tables

A load history table has the same goal as allowing loads to be interrupted: it attempts to reduce the amount of data that is brought into the cache which is unused, thereby increasing the speed of the machine containing the cache. A load history table uses information about the past behavior of the instruction at a given address to make decisions about how to handle that instruction in the future, in the belief that this will allow better determination of which data are not worth bringing into the cache than load interruption.

Four sets of simulations were run to test the effectiveness of the use of a history table in reducing the amount of time wasted loading data which is never used into the data cache. As table 5.4 shows, the average performance improvement achieved by this modification is fairly constant regardless of the size of the history table being used. This is probably due to the fact that the history table only has significant impact on the execution time of a program when the program contains memory references that are contained within loops, as the history table has no information about an instruction the first time an instruction at a given address is encountered, so it may not be particularly useful to have an extremely large history table.

Another interesting result is the fact that the performance of the history table modification is not monotonically increasing with the size of the history table. This seems very strange, as the larger history table would seem to have more accurate information, not less. The history table makes the decision about whether or not to bring an entire line into the cache based on how the line brought in the last time that instruction caused a cache miss has been used, so it would seem that the infrequently executed instructions, which remain in the history table the longest, would be the ones about which the history table could make the most accurate judgments. Apparently, this effect is not significant, which is not that hard to understand, as traces of the length used in these experiments are unlikely to contain loops which are both long enough to contain large numbers of memory reference instructions

Size of Victim Cache	Improvement Over Direct-Mapped	Improvement Over Reference Machine
No Victim Cache	0	-8.3%
2 Line Victim Cache	8.8%	.5%
8 Line Victim Cache	9.2%	.9%
32 Line Victim Cache	9.4%	1.1%

Table 5.3: Performance of the Victim Cache

and are executed a sufficient number of times that the history table could have a major impact on the execution time of these traces.

The use of a load history table seems to be an effective way of increasing the performance of a machine by reducing the amount of time spent bringing data that is not going to be used into the cache. This method is more effective than the use of load interruption, which leads to the conclusion that the load history table's explicit prediction of which data are not likely to be used by examining the past behavior of instructions is more effective than the implicit prediction done by load interruption.

5.5 Victim Cache

As reported by Jouppi [6], the addition of a victim cache to a direct-mapped cache is an effective way of improving the performance of a system containing such a cache. In Jouppi's work, the addition of a victim cache was found to remove a sizable fraction of the misses which occurred because of conflicts for cache lines. No attempt was made to compare the performance of a victim cache-equipped direct-mapped cache to an associative cache, as his paper focused on the design of machines with very low cycle times, in which it was not desirable to use a set-associative cache, as the increased access time of a set-associative cache would require an unacceptable increase in the cycle time of the machine.

For this work, the performance of a direct-mapped cache with a victim cache was compared to the performance of a four-way set-associative cache of the same size as the direct-mapped cache (64K). Simulations indicate, as shown in table 5.5, that the addition of even a very small victim cache to a direct-mapped cache the size of the one in the RISC System/6000 allows slightly better performance than a four-way set-associative design, indicating that the use of a direct-mapped cache with a victim cache should be considered if it is possible to reduce the cycle time of the machine by using a direct-mapped cache instead of a set-associative one. While the simulations run showed that the use of a direct-mapped cache with a victim cache performed slightly better than a four-way set-associative cache of the same size, the addition of a victim cache to a set-associative cache should result in a cache with performance at least as good as the direct-mapped cache with victim cache.

The fact that the addition of even a two-entry victim cache to a direct-mapped cache results in better performance than the original four-way set-associative design was mildly surprising. However, in a cache as large as the one in the RISC System/6000, even if direct-mapped, it would be expected that the number of conflicts over cache lines at any one time would be very small, due to the large number of cache lines available (512, in the case of a 64K cache with 128-byte lines). Therefore, it is not surprising that even a very small victim cache can contain most of the lines that are thrown out of the cache because of conflicts.

Explaining the fact that the use of a direct-mapped cache with a victim cache results in a speed improvement is slightly more difficult. In order for this to happen, the victim cache must be able to reduce the effects of some conflicts for cache lines which occur in the set-associative cache. This is apparently what happens. Even in the set-associative cache, there will still be some conflicts over cache lines. When the cache configuration is changed to be direct-mapped, additional conflicts for cache lines are added, and the conflicts which occurred in the set-associative cache remain. Assuming that, because of the large size of the cache, the number of conflicts occurring at any one time is small, it is possible for a victim cache to reduce the impact of not only those conflicts which occur only in the direct-mapped cache, but those which occur in the set-associative cache as well. This explains why the direct-mapped cache with victim cache was shown to achieve slightly better performance than the set-associative cache.

Chapter 6

Conclusions

Previous work suggests, and the simulations run during the course of this research confirm, that there is little benefit to be gained by increasing the size or associativity of caches beyond the 64K, four-way set-associative level when the computer containing this cache is operating in a single-program mode. Few programs access more than this much data, and most of those that do, such as matrix manipulation programs, can often be modified so that their performance is not significantly degraded by being unable to fit all of their data into the cache at one time. In their paper[7], Bowen Liu and Nelson Strother describe how this can be done for matrix-oriented algorithms. Increasing the size of a cache beyond the 32K-128K range may be of benefit in a multiprogrammed environment, as it might be possible for a sufficiently large cache to hold the working sets of several programs, which would allow the computer to switch back and forth between these programs with very little delay. In order for this benefit to be achieved, however, it will probably be necessary for cache sizes to be much larger than is currently feasible, as most modern computers have several tasks being executed at any one time, and it would probably be necessary for the cache to contain a large fraction of the working sets of all of the tasks before substantial benefit was seen.

As traditional cache designs seem to have reached a point of diminishing returns, it seems logical to look for other methods of increasing the performance of cache memories. Since cache memories were introduced, prefetching has been considered as a possible way to make cache memories more effective, by bringing data into the cache before it is needed, thus avoiding the delay that is caused if data is brought into the cache from the main store the first time a given datum is needed. Unfortunately, most prefetching methods do not improve the performance of the machine they are implemented on, either because the predictions of which data should be prefetched are not sufficiently accurate, or because the data that is prefetched replaces other data in the cache that is needed before the prefetched data.

The prefetching scheme studied here, which uses load and store with update instructions to predict which data to prefetch, does not improve the simulated performance of a RISC System/6000. Since load/store with update instructions are typically used in situations where a large amount of data, regularly spaced in the main store, is needed by the program, the predictions made by this method about which data should be prefetched are very accurate. This indicates that either the cache pollution caused by those data which are incorrectly prefetched is a more significant factor than the improvement gained by prefetching data that is used, or that the added delay that can be introduced by causing cache misses to occur closer together in time overwhelms the benefit gained by prefetching data.

Another reason why prefetching is not usually a good idea is the additional processor-memory traffic that occurs when prefetching is in use. Unless all of the predictions made by a prefetching algorithm result in bringing data that will be used into the cache and the prefetching data never causes a datum that is used before the prefetched data to be thrown out of the cache, some additional memory references will be generated in a system that incorporates prefetching that will not be generated in a system that does not incorporate prefetching. These references will either be to prefetch data into the cache that will not be used, or to retrieve data that was replaced in the cache by prefetched data, but which was needed before the prefetched data.

Almost all modern computers have more than one subsystem that accesses the main memory, although these subsystems are rarely modeled in simulations, due to the difficulty in predicting their behavior. Many computers contain a direct connection from external storage, such as magnetic media, to the main store, to allow transfers of data to/from external storage to occur while the processor does something else. Typically, main storage systems can only be accessed by one subsystem at a time, so adding extra memory traffic to a system such as this could negate some of the performance benefit achieved from prefetching by increasing the demands on the main store to the point where other subsystems have to wait for unnecessary memory accesses to complete before they can access the main store. This effect is especially significant in some types of parallel computers, which contain several processors that need to access the same main store. The limiting factor on performance in such machines is often the speed of the connection to main memory, so adding unnecessary memory references is especially detrimental to such systems.

Taking all of these factors into account, it would appear that this prefetching scheme should not be considered for implementation in actual hardware, as it does not improve the performance of the machine as simulated, and there is reason to believe that an actual implementation of this modification would be slower than the simulation. A more profitable tactic for increasing the effectiveness of cache memories is to attempt to reduce the amount of data that is brought into the cache that is never used. Cache memories typically bring some amount of data that is located near the datum that is needed into the cache whenever a cache miss occurs. The locality of reference displayed by most computer programs makes this a good idea in general, but some data that is not needed is brought into the cache by this method, which takes time that could be spent on more useful things if this data could be identified and not brought into the cache.

Load interruption attempts to reduce the amount of unused data that is brought into the cache by allowing one cache miss to interrupt another, so long as the first cache miss has been able to bring the specific datum that was needed into the cache prior to the interruption. The data in the first cache line which is prevented from reaching the cache by the interruption has only a probability of being used, which decreases with distance from the datum that caused the cache miss. The data which have the greatest chance of not reaching the cache due to an interruption are those data which are loaded into the cache last; these data have the smallest probability of being used by the program. Since it is known that the datum which caused the second cache miss will be needed by the program, allowing this datum to take precedence over the data in the first cache line should improve performance by allowing data that is needed by the program to be brought into the cache more quickly, albeit at the expense of causing some data that might be needed by the program not to be brought into the cache.

Simulations indicate that this is effective, producing a 2.9 % average improvement in speed on the traces used, a much larger improvement than was generated by doubling either the size or associativity of the data cache. One of the traces showed a performance decrease of 3.6 %, which is certainly possible if too much data that is needed by the program does not reach the cache because of load interruption. All of the other traces showed speed improvements, so it is reasonable to conclude that implementing load interruption on an actual machine would increase the speed of that machine. Since the amount of hardware required to implement load interruption would be fairly small, load interruption is an attractive candidate for implementation in an actual machine.

The addition of a load history table to a cache also attempts to reduce the amount of unused data that is brought into the cache, but by a different method. The load history table uses the past behavior of the instruction located at a specific address in the memory to predict how to handle future executions of that instruction. If the cache line which was brought into memory the last time that instruction caused a cache miss was used sufficiently that it was worth bringing the entire cache line into memory, the history table tells the cache to bring the entire cache line into memory the next time that instruction causes a cache miss. If the line was not sufficiently used, the history table tells the cache to fetch only the datum that caused the cache miss. Since it is to be expected that each execution of the instruction located at a specific address will result in a similar use of the data fetched by that instruction, this method should very accurately predict which data should be brought into the cache, and which should not.

This method is more effective than load interruption at reducing the running time of a program. Average performance improvements of 4.1-4.3 % were found through simulation. Another benefit of this method is that none of the traces used had their execution time increased when a load history table was added to the reference machine, which suggests that this method would improve the performance of a larger number of programs than load interruption.

It is important to note that both the use of a load history table and load interruption can increase the number of memory references made by a program, because more than one memory reference per line is required if one of these methods causes data that is needed not to be brought into the cache on the first reference to a cache line. However, these methods should reduce the total amount of time spent accessing the main store, by reducing the amount of data that is brought in from the main store. For this reason, these methods would be reasonable additions to a system in which the main store was used heavily, making reducing the amount of time spent accessing the main store important, but might be detrimental in the case of a machine where reducing the number of references to main store was important, such as a machine which took a long time to change the subsystem that was accessing the main store.

While reducing the amount of time wasted by the cache is one way to improve the performance of a machine, another alternative is to change the cache design to allow the use of a shorter clock cycle. The use of a victim cache along with a direct-mapped cache attempts to increase the performance of a computer containing such a cache by reducing the access time of the cache, which would allow the cycle time of the computer to be reduced in many cases, thus improving the performance of the computer.

Because they do not require as many comparisons to determine if a desired datum is in the cache as set-associative caches, direct-mapped caches can be constructed to have lower access times than set-associative caches which are constructed out of the same technology. However, direct-mapped caches have significantly lower hit rates than set-associative caches of the same size. Simulation has shown that a RISC System/6000 with a direct-mapped cache is 8.3% slower than the same machine with a four-way set-associative cache in executing the traces used in this work. In order for the replacement of the set-associative cache in the current RISC System/6000 with a direct-mapped cache of the same size to be attractive, it would have to be possible to reduce the cycle time by enough to counteract this penalty by going to a direct-mapped cache, which is unlikely. However, in very high-performance machines, such as the ones described by Jouppi in his paper where he examines the victim cache [6], the cycle time reduction allowed by the use of direct-mapped caches is enough that they are attractive.

The addition of a small, fully-associative buffer to a direct-mapped cache, which caches the lines which were most recently thrown out of the main cache, and can return those lines to the main cache with only a small delay can improve the performance of a machine which contains a direct-mapped cache to the level enjoyed by a machine containing a set-associative cache of the same size. In the simulations which were run, the replacement of the four-way set-associative cache in the RISC System/6000 with an equally large direct-mapped cache and victim cache resulted in performance improvements of .5-1.1% over the set-associative cache, depending on the size of the victim cache. In these simulations, the victim cache was assumed to take one cycle longer than the main cache to return data to the processor in the event that the desired datum was in the victim cache but not in the main cache. This would suggest that it would be desirable to replace a set-associative cache with a direct-mapped cache whenever it was possible to reduce the cycle time of the computer by doing so. It might even be advisable to replace the cache in the current RISC System/6000 with a direct-mapped cache with victim cache if no cycle time improvement could be made, due to the greater simplicity of the direct-mapped cache.

It has been shown that several of the modifications explored in this work result in better performance than traditional methods of increasing cache performance when applied to caches such as the one in the RISC System/6000. This agrees with work that has been done previously, which found that the benefit to be gained from increasing the size or associativity of a cache drops off substantially once the cache reaches the 32-128K and four-way set-associative level. Prefetching data, using load/store with update instructions to select which data should be prefetched, has been found not to improve the performance of a computer. The use of load interruption or a load history table to reduce the amount of unused data brought into the cache has been shown to produce performance improvements in excess of those achieved through increasing the size or associativity of the RISC System/6000's cache. The addition of a victim cache to a direct-mapped cache increases the performance of a system containing a direct-mapped cache to a level comparable to that of a system containing an equally large set-associative cache. Since all of these methods require less hardware than increasing the size or associativity of a cache the size of the one in the RISC System/6000, and many of them result in better performance, they appear to be attractive alternatives to increasing the size or associativity of a cache in the quest for higher levels of performance.

Bibliography

[1] Anant Agarwal. *Analysis of Cache Performance for Operating Systems and Multiprogramming.* Kluwer Academic Publishers, 1989.

[2] Chi-Hung Chi and Henry Dietz. Improving cache performance by selective cache bypass. In *Proceedings of the 22nd Hawaii International Conference on System Science*, volume 1, pages 277–85, 1903.

[3] William R. Hardell, Jr., Dwain A. Hicks, Lawrence C. Howell, Jr., Warren E. Maule, Robert Montoye, and David P. Tuttle. Data cache and storage control units. In *IBM RISC System/6000 Technology*, pages 44–50. IBM Corporation, 1990.

[4] Mark D. Hill and Alan Jay Smith. Evaluating associativity in cpu caches. *IEEE Transactions on Computing*, 38(12):1612–1630, December 1989.

[5] Gary A. Hoffman. Adaptive cache management. In *Proceedings of the 1987 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 129–32, 1987.

[6] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 364–372, 1990.

[7] Bowen Liu and Nelson Strother. Programming in VS Fortran on the IBM 3090 for maximum vector performance. *IEEE Computer*, pages 65–76, June 1988.

[8] Kevin O'Brien, Bill Hay, Joanne Minish, Hartmann Schaffer, Bob Schloss, Arvin Shepherd, and Matthew Zaleski. Advanced compiler technology for the RISC System/6000 architecture. In *IBM RISC System/6000 Technology*, pages 154–161. IBM Corporation, 1990.

[9] Steven Przybylski, Mark Horowitz, and John Hennessy. Performance tradeoffs in cache design. In *Proceedings of the 15th Annual Symposium on Computer Architecture*, pages 290–8, 1988.

[10] Alan Jay Smith. Sequential program prefetching in memory hierarchies. *IEEE Computer*, pages 7–21, December 1978.

Appendix A

Complete Simulation Results

All results listed are normalized to the reference machine.

Trace	Double Size	Double Associativity	Direct-Mapped	Interrupting	Prefetching
Average	.998	.9995	1.083	.971	1.039
Doduc	.991	.998	1.108	.967	1.001
Eqntott	.988	.9996	1.011	.924	1.007
Espresso	.9998	.9999	1.008	.954	1.00001
Fpppp	.998	.9996	1.010	.965	1.0000
Gcc	.993	.999	1.036	.956	1.0005
Li	.9998	.9999	1.011	.989	.99999
Matrix300	.9995	.9995	1.068	.960	1.396
Nasa7	.9999	1.0000	1.310	1.036	.985
Spice	.9997	1.0000	1.027	.977	1.0000
Tomcatv	.9998	.9999	1.274	.983	1.005

Table A.1: Changes to Cache Configuration, Load Interruption, and Prefetching

Trace	32–entry	8–entry	2–entry
Average	.989	.991	.995
Doduc	1.008	1.01	1.03
Eqntott	.996	.997	.998
Espresso	1.0003	1.001	1.002
Fpppp	1.0008	1.003	1.006
Gcc	.997	1.002	1.008
Li	1.0003	1.001	1.002
Matrix300	1.013	1.014	1.014
Nasa7	.873	.873	.873
Spice	1.003	1.003	1.004
Tomcatv	1.001	1.002	1.014

Table A.2: Victim Cache

Trace	256–entry	64–entry	16–entry	4–entry
Average	.957	.957	.959	.957
Doduc	.968	.962	.962	.963
Eqntott	.924	.924	.938	.931
Espresso	.955	.955	.955	.955
Fpppp	.9482	.949	.950	.9489
Gcc	.950	.950	.949	.948
Li	.990	.990	.990	.990
Matrix300	.960	.960	.960	.960
Nasa7	.935	.935	.935	.930
Spice	.963	.963	.972	.960
Tomcatv	.979	.982	.982	.983

Table A.3: Load History Table