1994

A new hardware prefetching scheme based on dynamic interpretation of the instruction stream

James Edward Van Peursem
Iowa State University

Order Number 9503601

A new hardware prefetching scheme based on dynamic interpretation of the instruction stream

Van Peursem, James Edward, Ph.D.

Iowa State University, 1994

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

A new hardware prefetching scheme based on dynamic

interpretation of the instruction stream

by

James Edward Van Peursem

A Dissertation Submitted to the

Graduate Faculty in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Department: Electrical Engineering and Computer Engineering
Major: Computer Engineering

Approved:

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Members of the Committee:

Signature was redacted for privacy.

Signature was redacted for privacy.

Signature was redacted for privacy.

Signature was redacted for privacy.

Iowa State University
Ames, Iowa

1994

To my loving supportive wife Denise.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

I would like to thank my major professor, Dr. James A. Davis for his guidance and constructive criticism of my research. This work would not be as thorough if it were not for his scrutiny. I would also like to thank Dr. Doug W. Jacobson for his patience and help, those many times when I popped my into his office asking for advice.

Thanks go to Jim Larus for the development of SPIM and Anne Rogers for her cycle level enhancements. This research would not have been possible without a CPU simulator such as SPIM.

I also wish to thank my friend Steve VanderWiel who helped me develop this idea in its early stages by getting together for brainstorming sessions over lunch at our favorite establishment. He was also a good sounding board throughout the whole project. Thanks also go to Bjoern Stabell, Ken Ronny Schouten, and Bert Gijsbers for the development of xpilot. Whenever my sanity was starting to falter, I could always count on a good game of xpilot to bring me back in check. :)

Lastly, I wish to give special thanks to my wife Denise, my daughter Morgan, and my family and friends for their love and support.

ABSTRACT

It is well known that memory latency is a major deterrent to achieving the maximum possible performance of a today's high speed RISC processors. Techniques to reduce or tolerate large memory latencies become essential for achieving high processor utilization.

Many methods, ranging from software to hardware solutions, have been studied with varying amounts of success. Most techniques have concentrated on data prefetching. However, our simulations show that the CPU is stalled up to 50% of the time waiting for instructions. The instruction memory latency reduction technique typically used in CPU designs today is the one block look-ahead (OBL) method.

In this thesis, I present a new hardware prefetching scheme based on dynamic interpretation of the instruction stream. This is done by adding a small pipeline to the cache that scans forward in the instruction stream interpreting each instruction and predicting the future execution path. It then prefetches what it predicts the CPU will be executing in the near future.

The pipelined prefetching engine has been shown to be a very effective technique for decreasing the instruction stall cycles in typical on-chip cache memories used today. It performs well, yielding reductions in stall cycles up to 30% or more for both scientific and general purpose programs, and has been shown to reduce the number of instruction stall cycles as compared to the OBL technique as well.

The idea of sub-line prefetching was also studied and presented. It was thought that prefetching full cache lines might present too much overhead in terms of bus bandwidth, so prefetches should only fill partial cache lines instead. However it was determined that prefetching partial cache lines does not show any benefit when dealing with cache lines smaller than 128 bytes.

1. INTRODUCTION

It is well known that memory latency is a major deterrent to achieving the peak execution speed of a today's high speed RISC processors. This chapter will review methods for reducing memory latency.

1.1 Memory latency is a performance bottleneck

Historically, the speed of a computer system was determined mainly by the frequency of the CPU clock. Advances in VLSI technology this past decade have given a consistent speedup in the microprocessor clock frequency at a rate of 50% to 100% each year whereas dynamic memory speeds have increased at a rate of 10% or less each year [Hennessy 91]. This has caused the development of an increasingly wider performance gap between the two.

In today's high-speed computing systems where RISC microprocessor clock rates are at or above 200 MHz [DEC 92], memory bandwidth is quickly becoming the performance bottleneck of the system. The microprocessor speeds are now in the range where it is very expensive or literally impossible to make dense memory chips at the same speed as the CPU. An entire memory system constructed from these high-speed chips is far too costly for most applications.

For example, Digital Equipment Corporation sells workstations with their Alpha 21064 processor running at speeds up to 200 MHz. This is a RISC processor which is superscalar and heavily pipelined, and can theoretically issue two new instructions every clock cycle. Therefore memory needs to supply two instructions and their operands every 5 ns. The fastest semiconductor memory available today in reasonable quantities is static memory with access times in the 10 ns range. In fact, most systems use dynamic memory because it is available in much denser packaging. The fastest available dynamic memories today have speeds in the 50–70 ns range. This difference in speed means that the CPU will have to wait for memory requests before it can continue processing. The amount of time between when the CPU requests a memory address and when the data is presented to the CPU is defined as the memory latency of the system. The bandwidth of a bus system is the amount of information it can transfer per second.

1.2 Cache memory

Cache memories help reduce this memory bandwidth disparity by keeping subsets of main memory in a small high-speed memory, but since the cache is smaller in size, the whole data set for a program cannot usually fit in this small high-speed memory. Therefore, when the CPU requests an item not in the cache, the data has to be retrieved from main memory and the CPU has to stall until main memory can complete the transaction. As the performance gap becomes wider, high-performance processors become more sensitive to stalls caused by cache misses because memory is becoming slower relative to their speed.

Typically cache memory only works in programs that exhibit good locality of reference. This manifests itself in two ways:

- Temporal locality (locality in time)—After an item is referenced, it will likely be referenced again soon.
- Spatial locality (locality in space)—After an item is referenced, items near it will likely be referenced soon.

To take advantage of temporal locality, when an item is referenced, a copy of it is kept in the cache. That way when the item is referenced again, it is available in the high-speed memory.

To take advantage of spatial locality, when an item is referenced, that item plus the items near it, a unit of memory called a cache line or block, are loaded into the cache. This is particularly effective for accesses to arrays and vectors which are frequently accessed sequentially.

Cache memory does not increase the memory speed but instead reduces the number of accesses to main memory. No matter how fast the cache memory is, the problem of CPU stalls still exists when an item is referenced that is not stored in the cache. Cache memory is typically much smaller than main memory, so this can be somewhat frequent. During the time the miss is being resolved, the CPU can do nothing but wait, so it is very important to try to maximize the hit ratio (minimize the miss ratio).

The average time it takes to access a hierarchal memory can easily be calculated. For a two level hierarchy (cache plus main memory) the equation is:

$$T_{eff} = \left(\text{Hit Ratio} \times T_{cache}\right) + \left(\left(1 - \text{Hit Ratio}\right) \times \left(T_{cache} + T_{mem}\right)\right)$$

In a given system, the only variable in this equation is the hit ratio as the other parameters are generally considered fixed by the design or technology. A plot of the average access time verses the hit ratio is shown in Figure 1.1. Note that this equation yields a linear plot where an increase in the hit ratio yields a proportional decrease in the average memory access time.

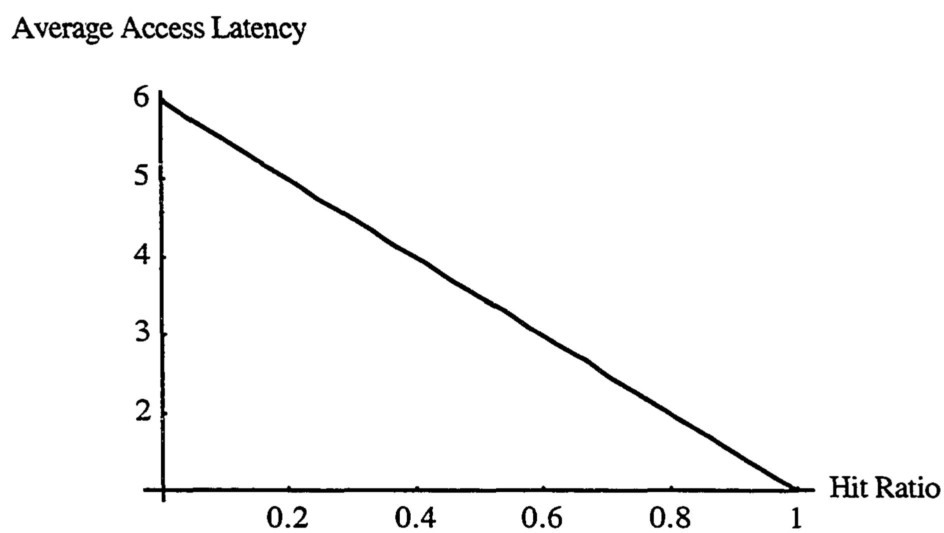

Figure 1.1: Access latency verses hit ratio
(in a system where cache access time = $^1/_5$ main memory access time)

Figure 1.2 shows some typical hit ratios for varying line and cache sizes. In the primary on-chip cache, a hit ratio of 85% – 97% is usually seen. With hit ratios this high, the average access is at or below two times the cache memory speed. As shown in the graph in Figure 1.1 this is a large improvement over the speed of main memory. With hit ratios this high it would seem futile to introduce techniques to increase it farther, but as will be shown, a small increase in the hit ratio can result in a significant execution speedup.

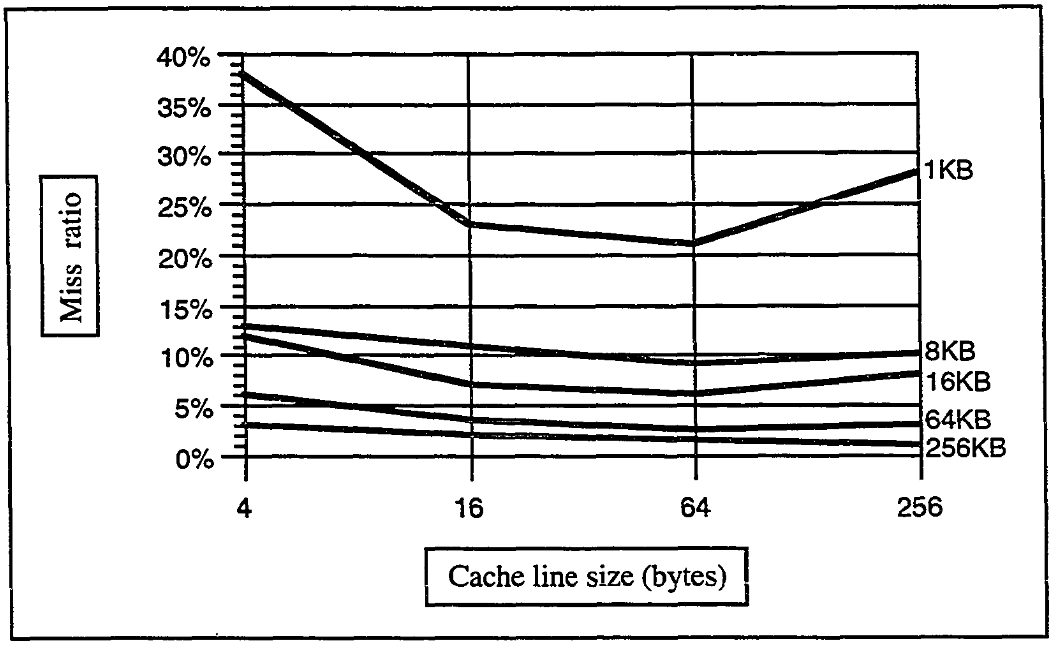

Figure 1.2: Typical miss ratios [Hennessy & Patterson 90]

1.3 Hit ratio analysis

Since the CPU needs to access memory every cycle, any access time greater than one cycle causes the CPU to stall, meaning it can do nothing but wait for this memory transaction to complete. Therefore an average access time of two means that on average, the CPU will have one stall cycle for every access, and is therefore only getting half of the possible performance.

A clearer way to look at the miss ratio penalty is in the amount of speedup attained from the cache. Given a cache memory access time of one cycle, and a main memory access time of five cycles, the cache can give a five times maximum possible speedup. The equation for calculating the speedup verses hit ratio is a variation on Amdahl's law as shown in Figure 1.3. A graph of speedup verses hit ratio of a system where memory access time is five times longer than cache access time is shown in Figure 1.4.

Note that there is a significant increase in speedup above an 80% hit ratio. In fact, the speed of the system can double when going from a 75% to a 100% hit ratio. This implies that large performance improvements can be obtained by small improvements in the hit ratio when the hit ratio is high.

$$\text{Speedup} = \frac{1}{\text{Miss Ratio} + \dfrac{\text{Hit Ratio}}{\text{Cache Speed}}}$$

Figure 1.3: Amdahl's law as applied to caches

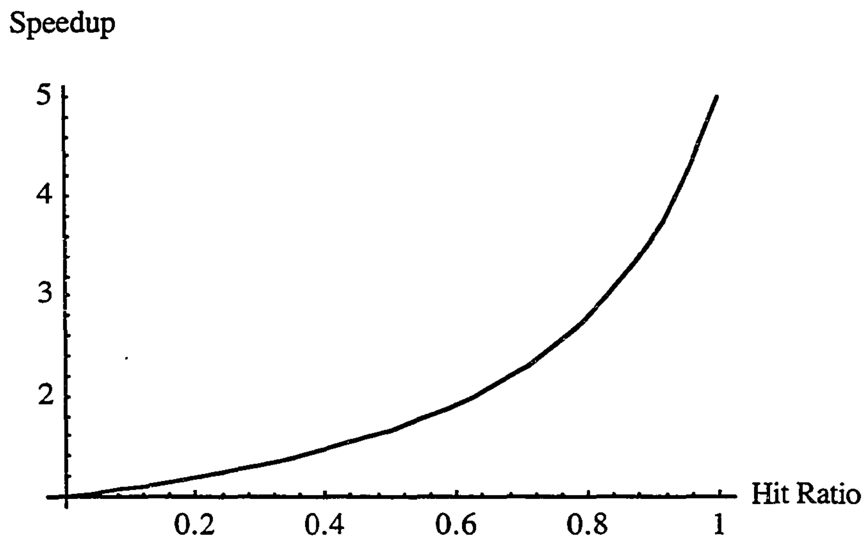

Figure 1.4: Speedup verses hit ratio
(in a system where cache access time = $^1/_5$ main memory access time)

1.4 Overlapping computations with memory accesses

Most memory latency reduction techniques try to decrease memory latency by overlapping computation with memory accesses. In a traditional system design, memory is idle until the CPU makes a memory request. At that point, the CPU stalls until the memory cycle is complete (one latency period). The CPU then acts upon that memory element during which time the memory is idle again. This type of serial access demonstrates a fundamental problem with the existing model. As Figure 1.5 shows, if these two operations can be overlapped, there is great potential for speedup.

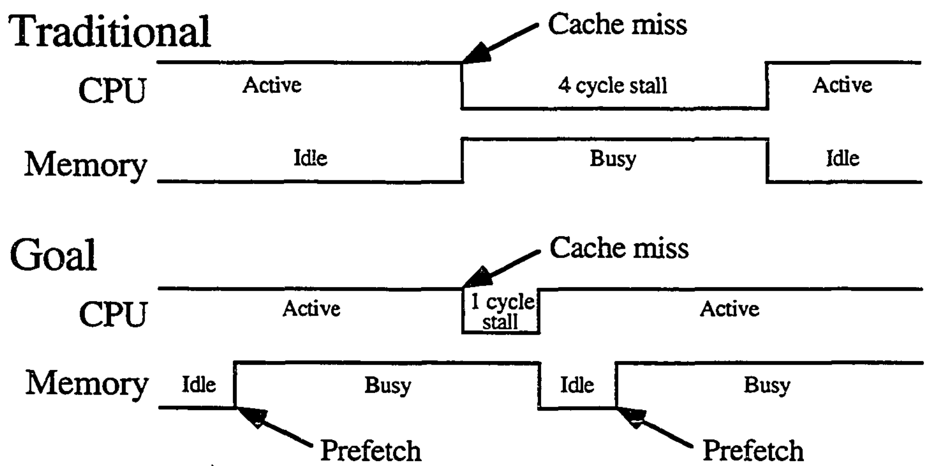

Figure 1.5: Traditional verses overlapped memory access

1.5 Misses in the instruction stream

Most of the existing research in memory latency reduction techniques focus solely on data accesses. This is very important, however it completely ignores a significant portion of the memory bandwidth which is caused by the instruction stream. Simulations performed while working on this project show that the CPU spends up to 50% of its time stalled waiting for instructions from main memory. Table 1.1 shows what percentage of time each of our benchmark programs stalls while waiting for instructions for 32 byte cache lines.

Table 1.1: Percent of CPU time spent stalled due to instruction cache misses.

benchmark	4K	8K	16K	32K
straight	33.81%	33.81%	33.81%	33.81%
dhry21	46.32%	35.79%	20.04%	17.27%
hanoi	0.26%	0.25%	0.24%	0.24%
heapsort	0.95%	0.58%	0.46%	0.28%
gzip	5.18%	5.81%	0.18%	0.21%
cc1	51.37%	45.40%	35.85%	28.11%
xlisp	40.23%	29.03%	26.36%	24.75%

The research presented here concentrates primarily on instruction prefetching. The solution proposed handles both instruction and data prefetching, however the main focus is the instruction stream.

1.6 Introduction to the pipelined prefetch engine

The solution proposed in this dissertation adds a small pipeline to the front of the cache which scans forward in the instruction stream and prefetches instructions that the CPU is about to execute. It decodes these instructions in an effort to predict the execution path. If it can accurately predict how the CPU will execute the instruction stream it will be able to prefetch the correct instructions into the cache before the CPU needs them. That way, when the CPU reaches this point of execution, the instructions will be in the cache and no stall will be necessary.

The prediction of the instruction stream is complicated by conditional branches. At the time the prefetch engine encounters such an instruction the condition will not yet be resolved so it is not possible to know whether the branch will be taken or not. The prefetch engine therefore includes a branch prediction mechanism.

1.7 Related work

This section presents the related work of other researchers. Many techniques have been presented for reducing the memory latency bottleneck. Each technique has its strengths and weaknesses which will be discussed.

1.7.1 General methods and ideas

Most of the traditional methods for increasing the hit ratio involve changing the design parameters of the cache memory. For example, one very obvious way to increase the hit ratio is to make the cache storage larger. If more things fit in the cache, it is more likely that because of temporal locality you will reuse an item that is stored there. However, since the primary cache is usually located on the CPU chip, any area used by the cache is area taken away from the processing functions. Or if the processing area stays the same, the die area must grow, which results in lower yields and increased production costs [Hennessy & Patterson 90].

Another method for increasing the hit ratio is to increase a cache's associativity. Associativity is a measure of how many different places within the cache memory an item may be placed. In a direct mapped cache, there is exactly one place where an item may be placed. In the other extreme, a fully associative cache, an item can be placed anywhere within the cache storage. This allows more flexibility in where to store items, and what to store in the cache so the most often accessed items can remain in the cache. The trade-off here is that adding associativity makes the physical size of the comparison circuits much larger. It also makes the cache slower in general because in a direct mapped cache the address is presented to both the tag memory and the storage memory, whereas in an associative cache the tag comparison needs to be done first.

Other ways of decreasing the memory latency penalty have been investigated. All of the following techniques try to decrease memory latency by increasing the overlap of computations and memory accesses as discussed in Section 1.4.

1.7.2 Write buffers

The simplest technique for reducing memory latency during write cycles is a write buffer which is organized as a simple FIFO queue [Smith 82]. When a write is to occur, the word to be written is entered into a queue. The system then uses spare memory cycles to write buffered words to the next level of the memory hierarchy.

An extension to the write buffer is a write cache [Bray & Flynn 91], organized like a small cache that uses an allocate strategy on write misses and write backs to reduce the number of writes. Unlike the FIFO queue, the write cache allows writes to the same cache line to be combined, thereby reducing the number of write cycles necessary. The disadvantage however is that on the first write to a cache line, the entire line needs to be loaded from main memory to be sure its contents are valid. This technique works well in programs where writes are done to many successive locations, however if the write pattern is more random, this technique has a negative effect.

1.7.3 Non-blocking caches

Another proposed solution is a design that allows the processor to continue execution on unsatisfied memory references through the use of non-blocking caches (also called lockup-free caches) [Kroft 81, Öner & Dubois 1992]. This architecture pipelines cache memory requests and reads the results directly into the appropriate section of the CPU needing the data. The non-blocking caches hide the latency of data access misses by allowing execution to proceed concurrently with cache misses until an instruction that actually needs a value to be returned is reached. Because of consistency requirements and dependencies, it is likely that only a portion of the read latency can be hidden. In [Chen & Baer 92, Gupta & Hennessy 91] comparative studies were done on the performance of non-blocking caches and other techniques. They found that in general non-blocking caches are less effective than prefetching techniques (described later). However, non-blocking caches combine very nicely with other techniques and hybrid systems are proposed that involve non-blocking caches plus prefetching techniques.

1.7.4 Speculative loads

In load/store architectures (e.g., most RISC processors), compiler techniques can be used to minimize the data latency. Through the use of speculative loads [Rogers & Li 92] the compiler can schedule non-blocking data fetch instructions which load items before they are needed by the CPU. These fetch instructions are non-blocking and can asynchronously load data directly into the CPU registers.

Speculative loads require both software and hardware support. The compiler becomes more complex because it needs to move data loads as far ahead of data accesses as possible, even across basic block boundaries. The pipeline control also becomes more complex because the load instructions must be non-blocking, and register accesses can block the pipe. If the data is not yet present in a register when that register is next accessed, the pipeline must stall until the data is present. Extra hardware is also required to asynchronously load memory data directly into the register file and notify the pipeline control when it is valid.

1.7.5 Context switching caches

Another technique is the use of hardware context switching [Smith 78a, Weber & Gupta 89]. If several threads of execution exist in a system, memory latencies can be hidden by quickly switching contexts rather than waiting for the memory access to complete. This requires special dedicated hardware to store the contexts and be able to quickly switch between them. This technique is effective when memory latency times are very large, and many threads of execution exist. For example, the APRIL architecture [Agarwal et al. 90] had a context switching time of 10 cycles, and was shown to give 80% processor utilization in a system with a 55 cycle memory latency.

For systems with naturally lower latencies, context switching is not as effective because the switching times are usually on the same order as the latency times.

1.7.6 Cache prefetching

Another way to reduce memory latency is through cache prefetching, which is the action of bringing data into the cache before it is actually needed. Prefetching is similar to speculative loads mentioned above in the sense that it is non-blocking and is a best guess of what will be needed by the CPU in the near future. The main difference, however, is that in cache prefetching, the data is loaded into the cache rather than directly into CPU registers.

One advantage of loading into the cache rather than CPU registers, is that the penalty for an incorrect guess is lower because the storage area is much larger. An incorrect guess in speculative loads consumes a CPU register, of which there are usually around 32 available. An incorrect guess in cache prefetching consumes one cache line, of which there are usually hundreds to thousands available.

Another advantage is that cache prefetching design is usually simpler and more modular than support hardware for speculative loads. In speculative loads, the controller needs to identify whether a register is valid and conditionally stall the pipe if it is not. It also takes special hardware to asynchronously load the data from memory directly into a register, bypassing the pipeline. In cache prefetching, the support hardware usually exists in the cache and is totally isolated from the CPU design.

Depending on how prefetches are determined and initiated, prefetching can be hardware or software controlled. The hardware approach detects accesses with regular patterns and issues prefetches at run time, whereas the software approach relies on the compiler to analyze programs and to insert explicit prefetch instructions at compile time.

1.7.7 Software prefetching

Software prefetching techniques rely on data access patterns being detected by static program analysis at compile time. The compiler inserts data prefetch (non-blocking) instructions several cycles before their corresponding need by the CPU. The processor has to explicitly execute these prefetch instructions at run-time to initiate the memory fetch. Some modern microprocessors have cache related instructions in their instruction set, such as the MIPS R4000 family. The CACHE instruction [Kane & Heinrich 92] can be used to perform various cache maintenance tasks, including filling a cache block from main memory.

The PowerPC architecture [Motorola 93] contains many cache maintenance instructions like dcbt (Data Cache Block Touch) which is described as a hint that performance will likely be improved if the block containing the address is fetched into the data cache because the program will probably soon load from that address.

Special compiler techniques are necessary to make software prefetching work. Many problems exist in the compiler analysis including static recognition of access patterns. It is not possible to statically determine patterns in accesses to linked lists and pointer references. Another complication is that it is difficult to determine whether an item will already be cached [Mowry et al. 92]. It would be counterproductive to issue a prefetch on an item that is already in the primary cache. In general, software data prefetching has been shown to be effective in scientific code, but not so in general code.

The Stanford DASH project proposed a nonbinding software-controlled prefetching technique [Mowry & Gupta 91]. In this context, the software prefetching was shown to increase the performance of MP3D (a particle-based simulator used in aeronautics) and LU (an LU-decomposition program) applications by 86% and 83% respectively. It is claimed that only 16 and 8 lines of code respectively were added in order to achieve these results. One reason this was so effective in DASH was because of the high stall rate otherwise present in cache coherent multiprocessor systems.

Software schemes have some important disadvantages. First, they add an instruction execution overhead. Depending on how many prefetch instructions need to be executed, the act of merely fetching the prefetch instructions could add significant memory traffic and become a disadvantage. Another subtle but important disadvantage is that the prefetch instructions are also stored in the cache, which can push out other potentially useful instructions. One other disadvantage is that they need to gain all of their information statically at compile time. There is more information available at run-time which hardware prefetching techniques may be able to take advantage of.

1.7.8 Hardware prefetching

Hardware prefetching is different from software prefetching in that the determination of what to prefetch is done at run-time rather than compile time. The simplest hardware prefetching scheme is the one block look-ahead (OBL) policy [Smith 78b, Smith 82]. That is, upon referencing block i (for all i) in memory, the only potential prefetch is to block i + 1. There are three variations of OBL considered. *Prefetch always* issues a prefetch after every reference, so that when an address in block i is accessed (read or write), block i + 1 is always prefetched (if it's not in the cache already). *Prefetch on misses* issues a prefetch of block i + 1 when a memory access causes a miss in block i. This effectively doubles the cache line size because at every miss, two lines are loaded. In *Tagged prefetch*, which was first proposed by [Gindele 77], each block has a tag bit associated with it. When a block is prefetched, its tag bit is reset to zero, and each time a block is used, its tag bit is set to one. When a block undergoes a zero to one transition (i.e., when the line is referenced for the first time after prefetching or is demand fetched), a prefetch is initiated for the next sequential line.

Smith's experiments first involved studying OBL for page prefetching in virtual memory. He found that page prefetching degraded performance with the 4096 byte page sizes typically used, and that it performed better with smaller page sizes. It appeared to work best with page sizes around 32 to 64 bytes. He then commented that these sizes are typical of cache lines, so OBL should work well in cache prefetching.

The simulation method used address trace files of various business and scientific programs. The cache had 64 sets with lines of 32 and 64 bytes. The number of lines in the cache was varied as part of the experiment. He simulated a multiprogramming environment by switching between trace files every 10,000 time units, where cache references took one time unit and main memory accesses took ten. He used the multiprogramming simulation because he felt it was a more true representation of a real work load than a uniprogram simulation. I agree with this, however I feel that to more accurately study the trade-offs of one prefetching method over another, the problem should be as simple and isolated as possible. I therefore simulated only one program running at a time.

Today, cache memory is approximately five to ten times faster than main memory. Also, because direct mapped caches can operate faster than associative caches, on-chip RISC caches are typically direct mapped. Today's memory characteristics are somewhat different than in the systems Smith studied, but the same methods still apply.

Smith's experiments showed that *prefetch always*, when used on both the instructions and data, reduced the combined miss ratio by 50 to 90 percent and that *tagged prefetch* performed almost equally as well. The advantage of *tagged prefetch* was a slight reduction in the number of blocks transferred from memory to cache. This indicates that it is slightly more accurate in prefetching what is needed. *Prefetching on misses* was less than half as good at reducing the miss ratio as the other two methods.

The OBL methods appear to perform well for instruction prefetching because of its sequential nature. It does not fair very well for data prefetching however. Today if a vendor uses prefetching in the instruction cache, the method of choice is usually the *prefetch always* OBL method.

Rather than base the decision for prefetching purely on the current fetch address, the solution proposed in this dissertation adds a small pipeline to the front of the cache which scans forward in the instruction stream and prefetches instructions that the CPU is about to execute. It decodes these instructions in an effort to predict the execution path. If it can accurately predict how the CPU will execute the instruction stream it will be able to prefetch the right instructions into the cache before the CPU needs them. That way, when the CPU reaches this point of execution, the instructions will be in the cache and no stall will be necessary. A comparison of performance is made between this new technique and OBL.

[Przybylski 90] argues against complex prefetching strategies. His observation is that complicated prefetching strategies do not produce the performance improvements indicated by the accompanying reductions in miss ratios because of limited memory bandwidth and a strong temporal clustering of cache misses. It is therefore important to consider performance measures other than the miss ratio. For the environments he simulated, the most effective fetch strategy improved performance by between 1.7% and 4.5% over the simplest strategy. The performance measurement used in this dissertation is the actual number of stall cycles introduced in the CPU by cache misses in the instruction stream. This gives a very accurate picture of how the prefetching technique performs.

[Jouppi 90] presents a cache prefetching architecture that, rather than prefetching into the cache, prefetches into a separate FIFO buffer. In this way the prefetching will not pollute the cache and displace potentially useful items. When a cache miss occurs, the head of the FIFO is checked and can provide the data with a one cycle delay. This would appear to work as an extension to almost any prefetching technique.

A more recent data prefetching technique is based on detecting access patterns at run-time [Baer & Chen 91, Fu & Patel 92]. The architecture contains a reference prediction table (RPT) that contains entries for each load or store instruction executed. The RPT is used to keep track of previous reference addresses and associated strides for load and store instructions. When the program is approaching another load or store instruction that is in the RPT, the prefetch engine adds the previous stride to the previous referenced address and begins a prefetch of this item.

This method was shown to be a very effective technique for scientific code, while not being effective for general code [Baer & Chen 91, Fu & Patel 92]. This is due to the regularity of stride accesses in array and matrix operations that are common in scientific code. More specifically, it reduced the data access penalty by over 90% for the Matrix and Expresso benchmarks. It showed a gain of 35% to 70% for Tomcatv, Nasa, Eqntott, and Xlisp. The method showed less than a 30% improvement for the Spice, Dudoc, Gcc, and Fpppp. It should be noted that this data prefetching method may combine nicely with the instruction prefetching method presented in this dissertation.

Hardware schemes have several important disadvantages. The primary difficulty is detecting a pattern in the accesses and only constant stride accesses can be expected to do well. This type of access is common in scientific code where array and matrix operations are common. However, as Mowry points out [Mowry et al. 92], the compiler can do an effective job of detecting these access patterns too.

Another key disadvantage is that any hardware prefetching technique is put on silicon, where it is very expensive to change. For a commercially available microprocessor targeted for many different applications, this lack of flexibility can be very limiting because of the different memory systems it may be operating in. Compiler-based techniques are much more flexible.

1.8 Using the instruction stream for prefetching

As has been pointed out, memory latency is a prevailing problem in the performance of high-speed computers today so new methods are necessary to reduce this problem. In this dissertation, I present a method for reducing the memory latency of the instruction stream by having a small pipeline in front of the cache. This pipeline scans forward in the instruction stream and tries to predict the execution path of the CPU. It prefetches the instructions that it thinks the CPU will be executing soon.

The one block look-ahead (OBL) method is typically used today for instruction prefetching. My method is different from this in that the candidate for prefetching is not determined solely by its approximation to the current fetch location. Instead the prefetch candidate is determined by the predicted future execution path, which is determined by the interpreting pipeline.

1.9 Organization of this dissertation

Chapter 2 contains a conceptual introduction and general description of the prefetch engine. The simulation method as well as the programs simulated are described in Chapter 3. Chapter 4 contains analysis of the data and compares the results with the results of other techniques. In particular, the simulations were run side-by-side with the OBL technique which is commonly used for instruction prefetching in mainframe computers and some microprocessors today. Chapter 5 gives conclusions and suggestions for future work.

2. DETAILED DESCRIPTION OF THE PIPELINED PREFETCHING ENGINE

In an ideal cache prefetching scheme, the cache would prefetch an item from memory so that it reaches the cache just before the CPU requests that item, thereby avoiding the cache miss and CPU stall. This requires that the prefetching engine accurately predict what the CPU will need in the near future. The goal is clear, but the problem is accurately predicting future references.

As mentioned previously, current research in hardware cache prefetching uses methods that try to identify a memory access pattern so that the cache prefetch engine can use spare memory cycles to fetch memory items forward within that same pattern.

2.1 Conceptual view

A core thesis of this research is that it is possible to more accurately predict the future of execution by employing a technique for having the prefetch engine interpret the instruction stream as compared to trying to identify access patterns. The instruction stream gives a very good indication of what the CPU will be doing. If the prefetch engine reads and decodes the instruction stream, it can use the information to determine what the CPU will be doing and what operands it will need. In fact, in a load/store architecture the only instructions that affect memory are the load and store variants, and the only instructions that affect the path of execution are branch variants. The other instructions are not important to the prediction of future memory usage.

The conceptual structure of this pipelined prefetching system is shown in Figure 2.1. It has the normal components of a typical passive cache, but also includes a prefetching engine which is conceptually located between the primary cache and main memory. The prefetch engine is responsible for interpreting the instruction stream and predicting what the CPU will be doing. It can then prefetch the necessary instructions and data into the primary cache so they will be available when the CPU executes those regions of code.

Because the prefetch engine needs to communicate directly with the CPU, it will likely only work with the primary on-chip cache. In an actual implementation, the CPU, primary cache, and prefetch engine would likely all reside on the same silicon die as shown by the dashed box.

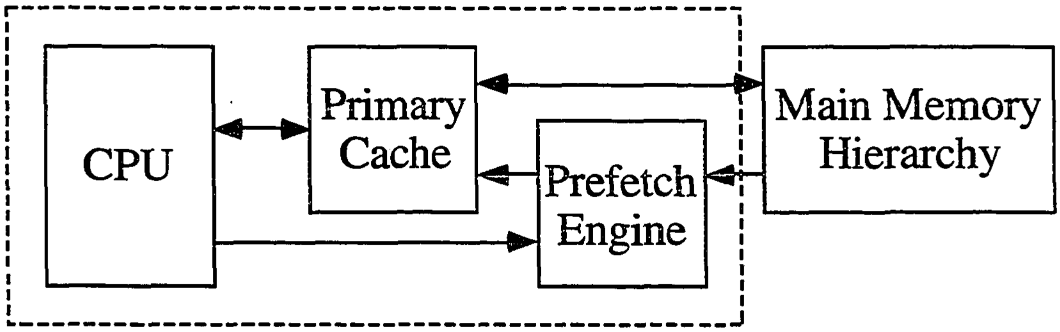

Figure 2.1: Conceptual view

2.2 Block diagram

A detailed block diagram is shown in Figure 2.2. The first thing to note is that the traditional cache is still present, and essentially unchanged. When the CPU initiates a memory fetch, the cache acts in every way the same as a normal cache memory. The only extensions needed in the normal cache portion are extra read and write ports into the cache data and tag memories.

The proposed cache architecture has some additional control structures that implement the prefetching model. The instruction fetch, instruction decode, and operand fetch stages operate very similar to the corresponding pipeline stages in a modern RISC processor. The CPU and prefetch engine do not share these stages; they are totally independently from the CPU pipeline, but act in a similar fashion.

The prefetch engine maintains a prefetch address counter (PAC), which is used as the address to prefetch the instruction from. The instruction fetch stage is responsible for prefetching these instructions from main memory. Once they arrive, they are stored in the cache and a copy is passed on to the instruction decode and branch prediction stages. If the instruction is a load or store variant, the information gained in this stage is passed on to the operand fetch stage, which starts prefetches of the needed operands for that instruction.

The branch prediction stage receives a copy of the prefetched instructions and is responsible for predicting the target address for branch variants. The methods used for these predictions are flexible, as discussed in Section 2.2.2. The address of branch instructions and the calculated target address are sent to the branch correlation stage which enqueues this information into a

FIFO. It's purpose is to make sure that branches were predicted correctly. If a mistake was made, it corrects the Prefetch Address Counter (PAC).

This whole process happens every clock cycle, in a pipeline fashion. The pipeline may better be viewed as in Figure 2.3. The details of how each stage works are presented starting at Section 2.2.1.

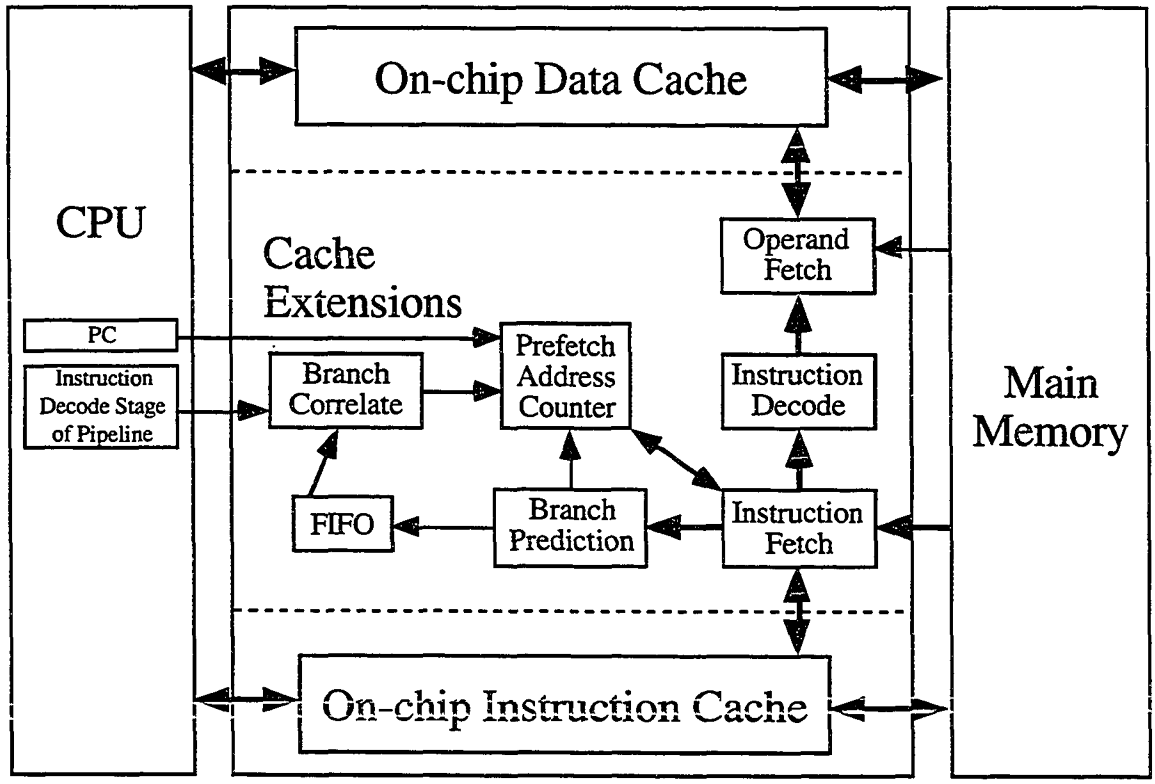

Figure 2.2: Block diagram

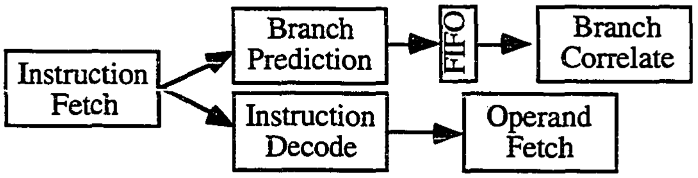

Figure 2.3: Prefetch engine pipeline

The prefetch engine needs to be working far enough ahead in the instruction stream so that each instruction can be loaded from main memory by the time the CPU needs to execute it. Since memory is slower than the CPU, it would appear that this would not be possible. However, programs typically have many loops which branch back to previously executed (and cached) code, so the prefetch engine takes advantage of this by prefetching instructions while the CPU is looping in the cache.

In fact, the higher the hit ratio, the more time the prefetch engine has to prefetch. The more time the prefetch engine has to prefetch, the higher the hit ratio will likely be. It would seem that this positive feedback would lead to large gains, but one foreseen limit is memory bandwidth. There may not be enough spare memory cycles to be able to do enough prefetching. Another limit is the accuracy of the branch predictions. With improper predictions, the unit will prefetch in the wrong direction, which will likely do more harm than good.

2.2.1 Instruction fetch stage

The instruction fetch (IF) stage is responsible for prefetching instructions into the instruction cache. It bases its prefetches on the Prefetch Address Counter (PAC), which is maintained by various pipeline stages within the pipelined prefetch engine. The PAC starts out at the same address as the Program Counter (PC) within the CPU and gradually moves forward, ahead of the PC. Figure 2.4 provides a flow chart describing the IF stage's operation during one clock cycle.

There is a throttling mechanism built into the prefetch engine so it does not get too far ahead of the CPU. The reason for this is that if the PAC were allowed to get too far ahead, it would risk prefetching an instruction that needs to be stored in the same cache line that the CPU is currently executing. Also because branch predictions are not perfect, the farther ahead the PAC gets, the less likely it is prefetching instructions that the CPU will actually execute.

The throttling mechanism is based on the number of outstanding branch instructions rather than the number of instructions it is ahead of the CPU. There are some architectural reasons for this decision, which will be discussed below in the description of the branch prediction and branch correlation stages (Sections 2.2.2 and 2.2.3). Studies were done to determine how the prefetch engine performs with different maximum leads. The results are presented in the Chapter 4.

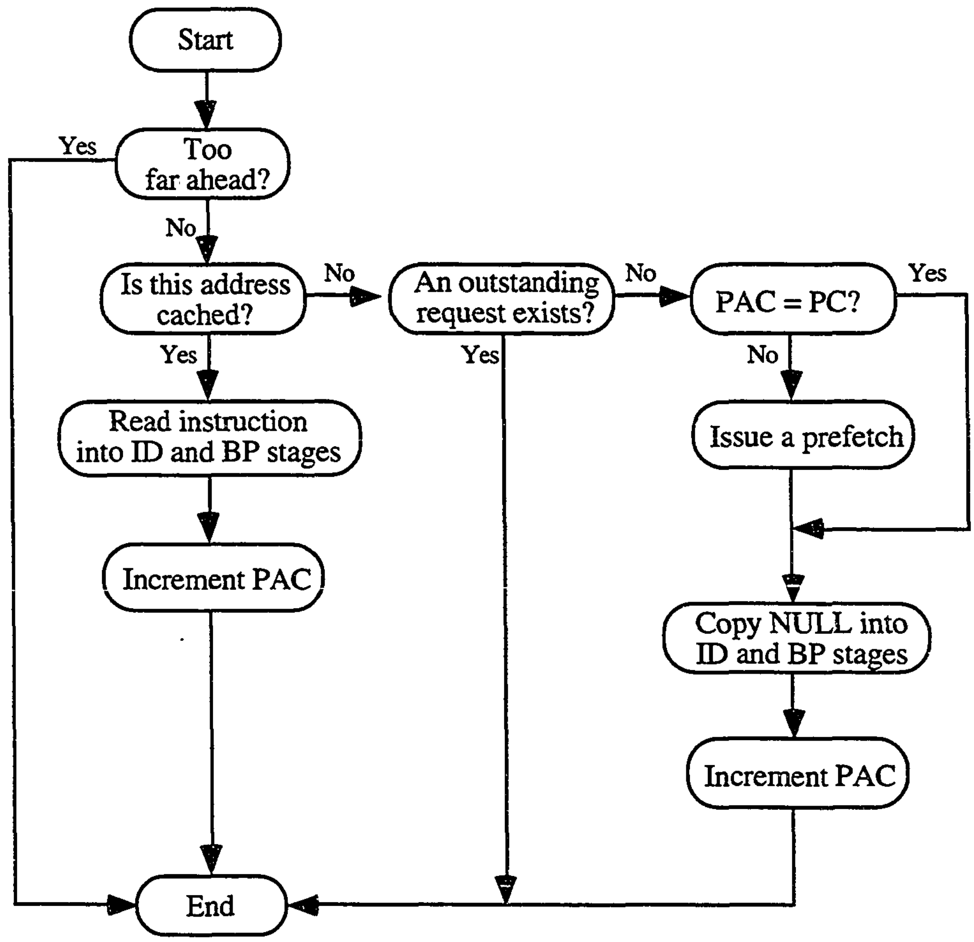

Figure 2.4: Instruction fetch stage flow chart

The first step in the algorithm for the IF stage is to check if the instruction at the PAC address is cached or not. If it is already in the cache, the IF stage reads the instruction from the cache and passes a copy to the instruction decode and branch prediction stages, and then increments the PAC.

If the address was not present in the cache, the IF stage checks to make sure that no other outstanding prefetch request still remains. If none exist, it checks to see if the PAC is pointing to the same address as the PC in the CPU. If it is, there is no need to prefetch this item because the CPU will be fetching it. Otherwise it issues a prefetch of the item and passes a NULL (no information) into the instruction decode and branch prediction stages. The last step is to increment the PAC.

2.2.2 Branch prediction stage

The Branch Prediction (BP) stage is responsible for predicting whether or not a given branch instruction is likely to be taken. It receives a copy of the instruction from the IF stage and decodes the instruction far enough to determine if it is a branch variant. If not, this stage does nothing. That is, no information is passed to the branch correlation stage.

If it is a branch instruction, this stage predicts if the branch will be taken. There are a number of branch prediction methods available (which will be discussed next), each of which works better in different environments. The prefetch engine is not sensitive to any particular branch prediction method. It is suggested that the branch prediction method used be the same that is employed in the CPU. In this way, the compiler optimizations will also apply to the prefetching engine. Also, if the CPU incorrectly predicts a branch, it will start fetching instructions in the wrong direction until the branch is resolved. It is desirable then to have these instructions resident in the cache as well. Even if they won't actually be executed, they will still cause cache misses and CPU stalls so need to be resident.

Four branch prediction methods were implemented and compared. In the first method, predictions are based on the calculated target address of the branch instruction. If it is a backward going branch, it will likely be taken, whereas forward going branches are assumed to not be taken. The basis for this is that in conventional languages such as C or Pascal, loops are frequently used, which implies a backward going branch at the end of the loop bringing the flow of execution back to the beginning. This is also the method employed in the PA-RISC design by Hewlett Packard. They claim that by making this policy, the compiler can better optimize the branches to the CPU's performance [Asprey et al. 93].

The second method is just the inverse of this which means that backward branches are assumed to be not taken, and forward branches are taken. This method was tried simply to provide a comparison with the previous method.

The third method was derived from current research papers on the topic [Yeh & Patt 93], where a two level branch history is used. Two bits are added to the width of the storage for each instruction in the cache. When a new line is loaded, these bits are set to the binary value 10. When a branch is executed, the bit pair for that instruction are either incremented if the branch was taken, or decremented if the branch was not taken. Then in the process of prefetching, if a branch instruction is encountered, these bits are consulted and used to predict if the branch will be taken. If their binary value is greater than or equal to 10, the branch is predicted to be taken.

The fourth method is simply to predict that all branches will be taken, regardless of the other factors. The basis for this is that there is an equal probability of a branch instruction being located in any word of a cache line, so in only a small fraction of the time will one be the last instruction in that line. Therefore if the branch is not taken, there will be instructions available within that same cache line for the CPU to execute before the next miss occurs. Since the branch correlation stage knows that the prefetch engine has predicted a branch incorrectly as soon as the CPU executes it, the PAC will quickly be corrected and prefetching will immediately begin down the correct path. Therefore it is best to predict that the branch is taken.

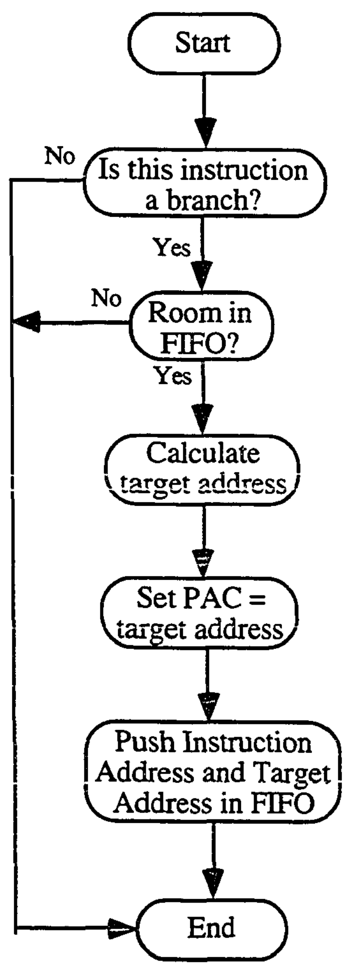

Figure 2.5: Branch prediction stage flow chart

Again, the choice of branch prediction method is based on the CPU that the prefetch unit will be working with. In the simulations I ran, the always branch method was the most accurate. It is also very attractive because it is much simpler and consumes less space in VLSI. As discussed above, it will also likely perform best at run-time, because in case the branch is not taken there are likely to be instructions after it which are in the same cache line.

Figure 2.5 shows a flow chart detailing the BP stage's operation. As can be seen, this is a fairly simple stage. The throttling mechanism described in the IF stage is used in the branch prediction stage as well. In fact, it is the branch prediction stage that sets the internal flags of whether the prefetch engine is too far ahead. The determining factor is if there is room available in the FIFO located between this stage and the branch correlation stage. If there is no room available in the FIFO then the prefetch engine must stall until the first predicted branch is resolved.

If the instruction is not a branch, the stage does nothing. If it is a branch, and there is room in the FIFO, it calculates the target address of the branch using any branch prediction method suitable (as described above). It then sets the Prefetch Address Counter (PAC) to the target address, and enqueues the address of this branch instruction and the predicted branch target address onto the FIFO.

2.2.3 Branch correlation stage

The Branch Correlation (BC) stage is responsible for making sure that the branch predictions were correct. If the predictions were wrong, the BC stage corrects the PAC so that the prefetch engine is actually prefetching what will be used.

This stage is disconnected from the rest of the pipeline and does not execute at every clock cycle. Instead, it fires only when the CPU decodes a branch instruction and calculates its target address. When this happens the BC stage compares the CPU generated instruction and target addresses with the head entry in the FIFO, which is filled by the BP stage. If the instruction address and target address match those which were predicted and entered in the FIFO then a correct branch was taken, so the BC stage simply pops the top entry off the FIFO. If either the instruction addresses or target addresses don't match, then an incorrect branch was encountered or a branch was incorrectly predicted. In either case, the BC stage sets the PAC to the target address calculated by the CPU and clears the FIFO.

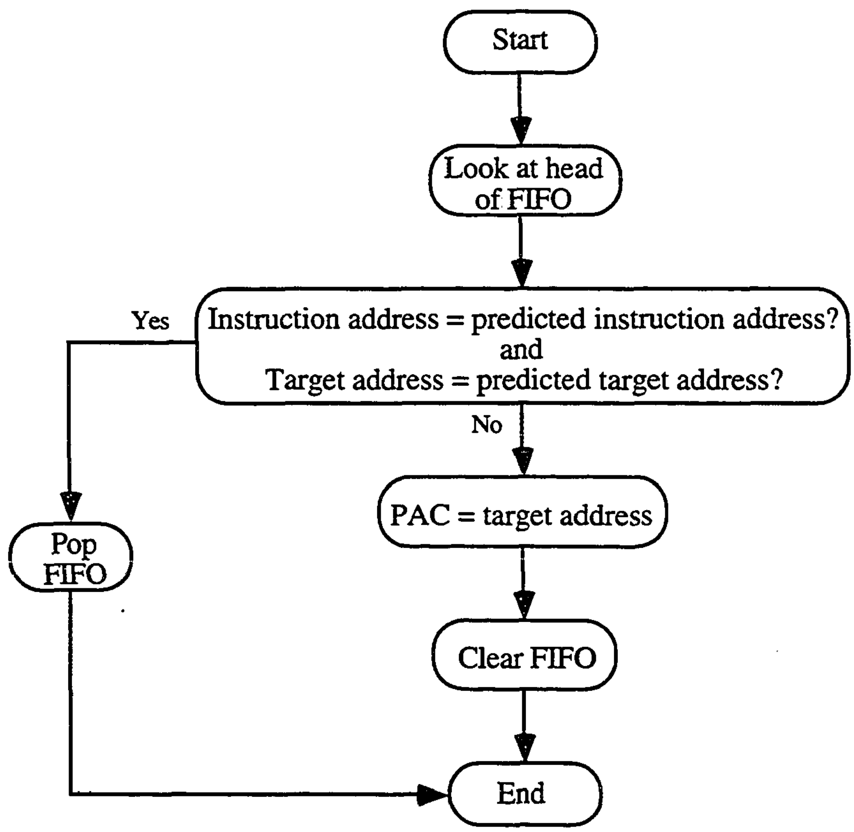

Figure 2.6: Branch correlation stage flow chart

2.2.4 Instruction decode stage

The instruction decode stage (ID) receives the instruction from the instruction fetch stage and decodes it, looking for load and store instruction variations. If the instruction is a load or store variant, and the addressing mode is either immediate or register based, the ID stage will calculate the address of the word to be read. This information is passed on to the operand fetch stage. This stage is only used for data prefetching, which was not examined for this dissertation. Figure 2.7 shows a flow chart describing its operation.

2.2.5 Operand fetch stage

The OF stage receives information from the ID stage on what load or store instruction is about to be executed, and what address the instruction will be working on. If that item is not already in the data cache, it needs to be prefetched. This stage is only used for data prefetching, which was not examined for this dissertation. It seems an obvious extension to the pipelined prefetching engine to have it load the operands for instructions as well, but it may not be possible to determine the address of all operands at run-time.

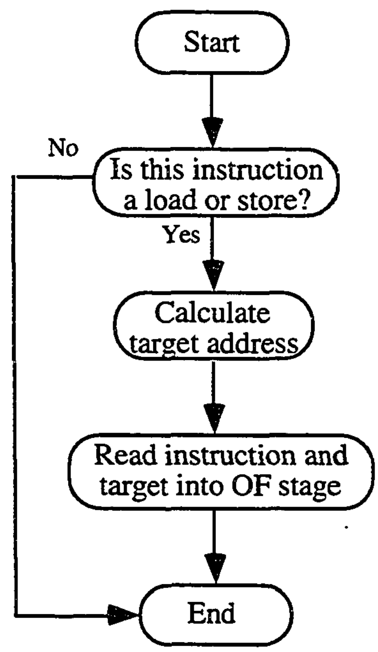

Figure 2.7: Instruction decode stage flow chart

2.3 Handling the branch delay slot

As the control of the prefetch engine is to be implemented as a pipeline, there is a one instruction delay between the instruction fetch, and the calculation of the next address. Therefore if a branch instruction enters the IF stage, the instruction immediately following it will always be fetched. But if the branch was to be taken, this instruction should not normally have been fetched. However, with the advent of the RISC pipeline in the CPU, it is common to have delayed branching, in which the instruction after the branch instruction is always executed. Therefore it is desirable to prefetch this instruction as well. By keeping the branch penalty the same as the branch penalty of the CPU, an incorrect post-branch fetch should never be made (excluding incorrectly predicted branches).

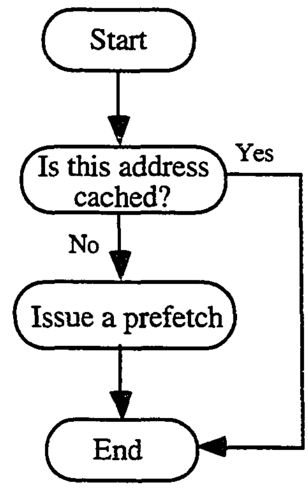

Figure 2.8: Operand fetch stage flow chart

2.4 Bus arbitration

The IF stage issues a prefetch for an instruction by entering its address into a prefetch buffer. There is an extension to the existing bus arbitration logic that actually does the prefetching when off-chip bus bandwidth is available. For example, currently there needs to be arbitration for the bus when the CPU issues any combination of operand read, data write, or instruction fetch, so it is an easy extension to add prefetching to this arbitration equation. It is desirable to make prefetches the lowest priority so that they do not preempt any normal processor activity. When the external memory bus is idle, and no processor requests are pending, the prefetches can take place.

There is also need for a unit which watches the CPU fetches and sees if they match anything in the prefetch buffer. If the CPU performs a fetch on an address that is in this buffer, it no longer needs to be prefetched. If it were prefetched later this would cause a waste of off-chip bus bandwidth.

2.5 Cache line size for prefetching

The off-chip bus bandwidth is a very limited commodity, therefore any prefetches should be done as accurately as possible, loading only what is necessary so that it does not consume too much memory bandwidth. It is also desirable to keep the prefetching activity from impeding normal processor activity, so it is best to make prefetches complete as quickly as possible. Our simulations show that when prefetching a full cache line, there is usually a processor request waiting for the bus by the time the prefetch finishes, which may cause stalling. However, if a portion of the contents of the line being prefetched will be used, it will save some stall cycles later.

It would seem desirable to make the prefetched cache lines smaller so the prefetch would complete quickly. However, making the cache lines smaller makes the hit ratio of demand fetched lines lower because of the reduced use of spacial locality. It is therefore desirable to make the demand fetched cache lines larger. It appears that an ideal solution would be to have each element adjustable independently.

This was implemented by making the cache lines small and having prefetches load a single line at a time, while demand fetches load an even multiple of this prefetched cache line size. For example, when the CPU does a fetch that causes a cache miss, load 64 bytes. But when doing a prefetch, only load 16 bytes. This would be implemented by having a cache line size of 16 bytes, and have the CPU demand fetch cause 4 cache lines to be loaded. When the cache is acting as a spacial reference source for the CPU, a larger cache line is better. But when doing prefetches, a small cache line is better. Experiments were done to determine the best line size multiplier and are presented in Chapter 4.

2.6 Prefetching invalid addresses

Because the prefetch engine will be working ahead of the CPU, it is not always possible to accurately predict the target of a branch. Some branches are register based which means that until the register contents are valid, the target address is not known. Therefore it is possible that the prefetch engine may branch to an address which is not valid, such as a bad page or memory which is not present. It would be bad to let the prefetch unit cause page faults and bus errors. In the same way, it is possible that the prefetch engine will prefetch past the end of a valid page, which would also cause a page fault. The time needed to load a page from disk is very long compared to memory accesses, so incorrectly causing a page to be loaded would be a big time penalty and is too much of a risk. Rather than resolve the page fault in this case, it would be best to simply notify the prefetch unit of the fault. In other words, tell the prefetch engine that the address is invalid, but do not have the memory management unit try to load a page from disk or tell the CPU that there was an error.

There are two cases that need to be considered depending on the organization of the cache. If it is a physically mapped cache, the memory management unit is located between the CPU and the cache. The addresses that the prefetch unit issues are physical addresses so page faults will never occur. However bus errors can occur if a read is done to an address where there is either supervisor memory (when fetched at a user level), or no memory present. In this case the solution is to have the chip select decoding logic recognize the problem and immediately report the bus error. Inside the bus handling state machine, recognize that this error was a result of a prefetch request and simply ignore the error and notify the prefetch engine. At this stage the prefetch engine could stall or start fetching down the other side of the last branch.

The other case is if the cache is virtually addressed, in which case the memory management unit is located between the cache and main memory. It is therefore necessary that the memory management unit be extended slightly so that in the case of a cache prefetch, it does not try to resolve a TLB entry that is in main memory. That is, the TLB only caches a small portion of the page translation table and keeps the rest out in main memory. When a page fault occurs, the TLB may need to load a portion of the table from main memory in order to resolve the fault. In this case it would be equivalent of turning a short prefetch request in to a long memory operation. It is also important that the memory management unit never resolve a page miss caused by a prefetch because loading a page is far too slow of an operation to risk at prefetching time. In both of these cases it is best if the MMU asserts an error line for the prefetch engine telling it that the address is not resolvable. Again here the prefetch engine could either stall until the CPU gets to this point or it could start prefetching down the other side of the last branch instruction. The last branch instruction is the last item in the FIFO between the BP and BC stages. It would be easy to find this element and calculate the other side of the branch.

3. SIMULATION METHOD

This chapter presents the simulation method used for studying the pipelined prefetching engine as well as the machine being simulated.

3.1 The simulator used

The simulator used for this research was SPIM [Larus 90], a MIPS R2000/R3000 CPU simulator written by Jim Larus from the University of Wisconsin. Anne Rogers from Princeton, when developing her speculative load model [Rogers & Li 92] added cycle level extensions to SPIM so that it simulates the MIPS processor at the clock cycle level. The simulator models a MIPS R2000/R3000 CPU with primary on-chip direct-mapped, write-through cache, and an N-way interleaved memory system with split bus cycles possible. The simulator is easy to modify for any memory and cache model.

SPIM can execute standard Unix executables or programs written in MIPS assembly. With the cycle level extensions, it also implements the R2000 pipeline at the clock cycle level. There are four basic reasons for using this simulator instead of a typical cache simulator based on address traces:

- We can easily collect relevant data on any program compiled for the MIPS R2000/R3000 architecture.
- Collecting and interpreting meaningful address traces is difficult due to the large size of the trace data.
- The full source code is available and is easily modified, so any statistic can be reported.
- It simulates a real execution environment including pipeline stalls, etc.

SPIM can simulate the execution of compiled programs, which provides a very dynamic simulation environment, so any set of programs can be used for measurements. Typically cache simulators work with address trace files, which must be collected using other means. It is sometimes difficult to gather accurate address traces. The prefetch engine also needs to decode the actual instructions being executed rather than just seeing memory request addresses, so address trace files do not provide enough information.

The third reason is that it is easy to modify the simulator to report any statistics necessary. Rather than just reporting hit and miss statistics, it can also collect and report CPU stall cycles, total CPU cycles, number of idle memory cycles, number of instruction stall cycles caused by cache misses, cycles per instruction, etc.

The last reason is that the SPIM simulator simulates the MIPS R2000 pipeline, so pipeline hazards, bubbles, flushes, etc. are simulated. This gives a more accurate view of performance than any instruction level or address trace simulations can. This simulator gives a true measure of improvements made, considering all aspects of the CPU.

We modified the SPIM simulator to include the prefetching engine presented in Chapter 2. This involved changes to the existing cache handling and bus arbitration code as well as adding the code for the pipelined prefetching engine.

3.2 The machine being simulated

For the basis of studying how the prefetching engine will perform in a computer system today, the simulator used numbers that would match a typical single CPU architecture. The primary cache can be accessed in one cycle, and main memory (or secondary cache) can be accessed in 5 cycles. These parameters are roughly equivalent to a 100 MHz system with 50 ns RAM, which would represent a typical machine in use today. The system has a simple bus design with no split cycles, and no interleaved memory. These factors would help increase the throughput of any given system, including the prefetching system.

Simulations were run on a wide variety of cache configurations, with cache lines ranging from four bytes to 512 bytes, and the caches having between eight lines and 1024 lines.

The MIPS R2000 is a classical RISC microprocessor with a five stage pipeline and a load/store architecture. A functional block diagram is shown in Figure 3.1. Some of its features include 32 general-purpose 32-bit registers, on-chip translation lookaside buffer (TLB) for fast address translation of virtual-to-physical memory mapping of the 4 Gbyte virtual address space, support for external coprocessors like the R2010 floating point unit (FPU), and a high bandwidth memory interface which handles separate external instruction and data caches ranging in size from 4 Kbytes to 64 Kbytes each.

The instruction set for the R2000/R3000 is based on a load/store architecture and contains 12 load and store instructions, and 12 branch type instructions. The full instruction set is shown in Table 3.1.

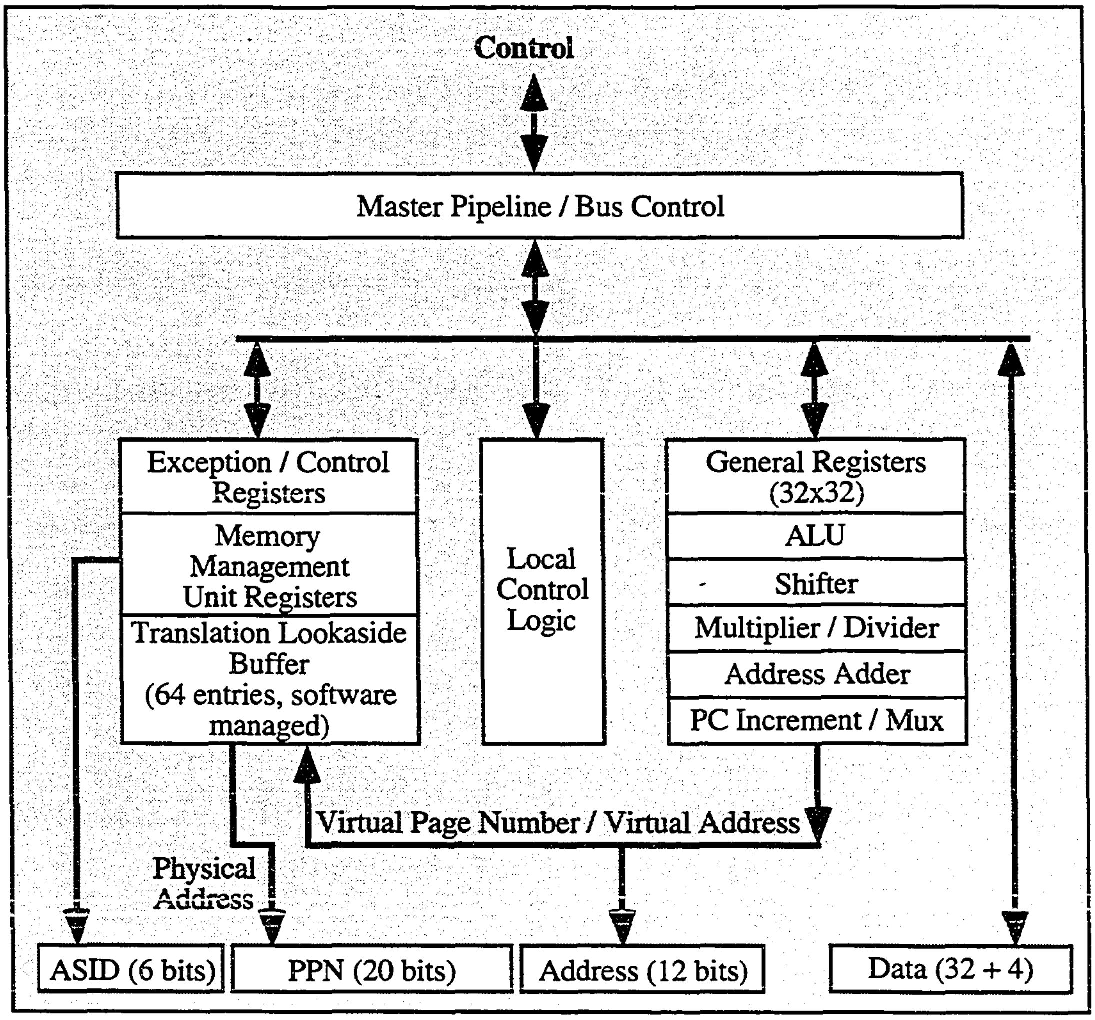

Figure 3.1: R2000/R3000 functional block diagram [Kane & Heinrich 92]

Table 3.1: MIPS R2000/R3000 instruction set [Kane & Heinrich 92]

OP	*Description*	*OP*	*Description*
	Load / Store Instructions		**Multiply / Divide Instructions**
LB	Load Byte	MULT	Multiply
LBU	Load Byte Unsigned	MULTU	Multiply Unsigned
LH	Load Halfword	DIV	Divide
LHU	Load Halfword Unsigned	DIVU	Divide Unsigned
LW	Load Word	MFHI	Move from HI
LWL	Load Word Left	MTHI	Move to HI
LWR	Load Word Right	MFLO	Move from LO
SB	Store Byte	MTLO	Move to LO
SH	Store Halfword		**Jump and Branch Instructions**
SW	Store Word	J	Jump
SWL	Store Word Left	JAL	Jump and Link
SWR	Store Word Right	JR	Jump to Register
	Arithmetic Instructions	JALR	Jump and Link Register
ADD	Add	BEQ	Branch on Equal
ADDI	Add Immediate	BNE	Branch on Not Equal
ADDU	Add Unsigned	BLEZ	Branch on Less than or Equal
ADDIU	Add Immediate Unsigned		to Zero
SUB	Subtract	BGTZ	Branch on Greater Than Zero
SUBU	Subtract Unsigned	BLTZ	Branch on Less Than Zero
LUI	Load Upper Immediate	BGEZ	Branch on Greater than or Equal
SLT	Set on Less Than		to zero
SLTI	Set on Less Than Immediate	BLTZAL	Branch on Less Than Zero
SLTU	Set on Less Than Unsigned		and Link
STLIU	Set on Less Than Immediate	BGEZAL	Branch on Greater than or Equal
	Unsigned		to Zero And Link
AND	AND		**Coprocessor Instructions**
ANDI	AND Immediate	LWCz	Load Word to Coprocessor z
NOR	NOR	SWCz	Store Word from Coprocessor z
OR	OR	MTCz	Move to Coprocessor z
ORI	OR Immediate	MFCz	Move From Coprocessor z
XOR	Exclusive OR	CTCz	Move Control to Coprocessor z
XORI	Exclusive OR Immediate	CFCz	Move Control from Coprocessor z
	Shift Instructions	COPz	Coprocessor z Operation
SLL	Shift Left Logical	BCzT	Branch on Coprocessor z True
SRL	Shift Right Logical	BCzF	Branch on Coprocessor z False
SRA	Shift Right Arithmetic		**Special Instructions**
SLLV	ShiftLeft Logical Variable	SYSCALL	System Call
SRLV	Shift Right Logical Varuable	BREAK	Break
SRAV	Shift Right Arithmetic Variable		

3.3 The benchmark programs

To represent the average job mix run on a workstation, seven benchmark programs were used in these studies, ranging from mathematics to sorting to benchmarks to compilers. The seven programs are:

straight – a simple program that has no branches in its body. There are some branches necessary in the start-up code, but once in the main body, the program is essentially sequential. The program consists of 135 lines of mathematical equations, which when compiled, produces 20,755 machine instructions. When run, 1,042 assembly code instructions were executed. This program will illustrate how the prefetching method works without branches, and when the cache is first filling.

dhry21 – the dhrystone program is a well-known benchmark used to measure CPU performance. This program compiles to 21,556 machine instructions. When run, 62,286 assembly code instructions were executed.

heapsort – a program implementing the heapsort sorting algorithm for various data set sizes. This program is very small so it usually fits in larger caches. The heapsort program compiles to 21,474 machine instructions. When run, 7,049,943 assembly code instructions were executed.

hanoi – a recursive solution to the towers of hanoi problem, which was used to indicate how the idea performs in recursive programs, or ones with many small procedures. This program is also very small and usually fits in larger caches. The hanoi program compiles to 20,890 machine instructions. When run, 1,576,026 assembly code instructions were executed.

gzip – the Free Software Foundation's popular LZW compression program. This is a typical general purpose program which is computationally intensive and should be a good representation of non-scientific code. The gzip program compiles to 36,056 machine instructions. When run, 5,754,372 assembly code instructions were executed.

cc1 – the Free Software Foundation's C compiler that is a part of the gcc package. It was used to compile a small C source file into a small assembly program. This too is a good representation of how the idea works in non-scientific code. The cc1 program compiles to 406,624 machine instructions. When run, 10,738,975 assembly code instructions were executed.

xlisp – a lisp language interpreter and execution environment. In the simulations, xlisp was interpreting a small lisp program. This has the behavior of being a very branchy program that is hard to predict its behavior. This too is a good representation of how the idea works in non-scientific code. The xlisp program compiles to 55,884 machine instructions. When run, 1,385,273 assembly code instructions were executed.

4. RESULTS

The programs described in Chapter 3 were each simulated 64 times, with cache lines ranging from 4 bytes to 512 bytes, and the cache having between 8 lines and 1024 lines. The resulting data was analyzed and the key issues investigated were branch prediction accuracy, sub-line prefetching performance, instruction stall times, increases in bus traffic, and the overall performance of a system with prefetching enabled.

4.1 Branch prediction accuracy

The accuracy of the prediction of the execution is directly dependent on the accuracy of the branch predictions. If the prefetching engine predicts a conditional branch incorrectly, it will prefetch down the wrong path. This has two possible consequences: 1) the correct side of the branch may not be resident in the cache so the CPU may have a cache miss, and 2) the incorrectly prefetched cache lines will replace other lines in the cache, which may cause a subsequent cache miss if they are needed again by the CPU.

Four branch prediction methods were implemented and compared. In the first method, predictions are based on the calculated target address of the conditional branch instruction. If it is a backward going branch, it will likely be taken, whereas forward going branches are assumed to not be taken. The basis for this is that in conventional languages such as C or Pascal, loops are frequently used, which implies a backward going conditional branch at the end of the loop bringing the flow of execution back to the beginning. This is also the method employed in the PA-RISC design by Hewlett Packard. They claim that by making this policy, the compiler can better optimize the branches to the CPU's performance [Asprey et al. 93].

The second method is just the inverse of this which means that backward conditional branches are assumed to be not taken, and forward branches are taken. This method was tried simply to provide a comparison with the previous method.

The third method was derived from current research papers on the topic [Yeh & Patt 93], where a two level branch history is used. Two bits are added to the width of the storage for each instruction in the cache. When a new line is loaded, these bits are set to the binary value 10. When a conditional branch is executed, the bit pair for that instruction are either incremented if the branch was taken, or decremented if the branch was not taken. Then in the process of prefetching, if a conditional branch instruction is encountered, these bits are consulted and used to predict if the branch will be taken. If their binary value is greater than or equal to 10, the branch is predicted to be taken.

The fourth method is simply to predict that all branches will be taken, regardless of the other factors. The basis for this is that there is an equal probability of a branch instruction being located in any word of a cache line, so in only a small fraction of the time will one be the last instruction in that line. Therefore if the branch is not taken, there will be instructions available within that same cache line for the CPU to execute before the next miss occurs. Since the branch correlation stage knows that the prefetch engine has predicted a branch incorrectly as soon as the CPU executes it, the PAC will quickly be corrected and prefetching will immediately begin down the correct path. Therefore it is best to predict that the branch is taken.

When simulating the seven benchmark programs across a wide variety of cache and line sizes (4-byte through 512-byte lines, with 8 lines through 1024 lines in the cache), we measured the accuracy of the prediction for each method. The results are shown in Table 4.1. 100% would mean that we correctly predicted the direction of the branch in all cases.

Table 4.1: Branch prediction accuracy

Benchmark:	Target based	Inverse target based	History based	Always take
straight:	38.1%	56.0%	93.6%	94.8%
dhry21:	38.2%	31.9%	57.3%	60.6%
heapsort:	17.2%	36.2%	51.0%	54.3%
hanoi:	45.3%	64.9%	69.7%	74.0%
gzip:	30.7%	37.7%	61.2%	63.4%
cc1:	26.5%	46.5%	60.7%	64.3%
xlisp:	32.6%	59.6%	67.3%	69.6%
overall:	32.7%	47.5%	65.8%	68.7%

The numbers for the history based method are lower than what is seen in the research papers [Yeh & Patt 93]. The reason for this is that the prefetch engine is sometimes far ahead of the CPU, predicting many branches. If the CPU executes a branch that was predicted incorrectly, that branch plus all other predicted branches not yet processed are marked as predicted incorrectly.

As can be seen, the most accurate method was to always predict branches to be taken. It is only slightly more accurate than the history based method, however it is also very attractive because it is much simpler and consumes less space in VLSI. As discussed above, it will also likely perform best at run-time, because in case the branch is not taken there are likely to be instructions after it which are in the same cache line. For these reasons, the always take method was used as the basis for the rest of the research.

4.2 Sub-line prefetching

The off-chip bus bandwidth is a very limited commodity, therefore any prefetches should be done as accurately as possible, loading only what is necessary so that it does not consume too much memory bandwidth. It is also desirable to keep the prefetching activity from impeding normal processor activity, so it is best to make prefetches complete as quickly as possible. Our simulations show that when prefetching a full cache line, there is usually a processor request waiting for the bus by the time the prefetch finishes, causes stalling. However, if a portion of the contents of the line being prefetched will be used, it will save some stall cycles later.

It is therefore important to consider whether the prefetched lines are being used only as temporal locality sources, or also as spacial locality sources. If they are just used for their temporal locality, the program is only benefiting by the prefetching engine loading the line slightly before its need. However if it is also being used as a spacial locality source, the CPU will benefit from the use of many words within each prefetched line.

If prefetched lines were used mainly as a temporal source, it would be desirable to make the prefetched cache lines smaller so the prefetch would complete quickly. However, making the cache lines smaller makes the hit ratio of demand fetched lines lower because of the reduced use of spacial locality. It is therefore desirable to make the demand fetched cache lines larger. It would seem that an ideal solution would be to have each element adjustable independently.

This was implemented by making the cache lines small and having prefetches load a single line at a time, while demand fetches load an even multiple of this prefetched cache line size. For example, when the CPU does a fetch that causes a cache miss, load 64 bytes, but when doing a prefetch, only load 16 bytes. This would be implemented by having a cache line size of 16 bytes, and have the CPU demand fetch cause 4 cache lines to be loaded.

The PowerPC 601 from Motorola implements a similar idea in which it has 16 byte lines consisting of two 8 byte sectors which are used as the basic coherency unit [Young 94]. When the CPU requests an item not in the cache, all 16 bytes are loaded from memory into the cache, but not necessarily in order. The sector with the requested item is loaded first, thereby decreasing the maximum wait time to two latency times rather than four (for a 32 bit bus). Once the first sector is loaded, the CPU can use the data while the adjacent sector is being loaded.

Experiments were done to see how changing the line size multiplier affected prefetching performance. This analysis reveals how much the prefetching activity acts as a spacial locality source along with a temporal locality source. If prefetching smaller lines performs better, the prefetching activity is being used mainly as a temporal reference source, whereas if larger lines are better the prefetching activity is being used both as a spacial and temporal reference source.

The SPIM simulator was modified to collect the number of instruction fetch stall cycles. When a cache miss occurred, a counter was incremented every simulated CPU clock cycle until the instruction was retrieved from memory. During the total run of a program, this statistic gives an accurate representation of the improvements made by any particular latency reduction technique.

The first caches considered for this experiment were 4K caches with 16 byte, 32 byte, and 64 byte cache lines, which are similar to the on-chip caches in the Intel 960CF, Motorola 68040, ARM 610, and TI MicroSPARC. All seven benchmark programs were simulated three times using these caches, with the sub-line multiplier changed from one to two to four, where one means that the prefetch lines are the same size as the demand fetched lines and four means that the prefetch line is one fourth as large as the demand fetched line. The cache line size indicated in the graphs is the size of the demand fetched lines.

The resulting graphs are presented in Figures 4.1 through 4.3. The vertical axis marks the normalized number of instruction stall cycles as compared to the same cache with no prefetching. A result of 1.0 indicates the same number of stall cycles as the same cache with prefetching turned off, while numbers less than 1.0 indicate less stall cycles, and therefore better performance.

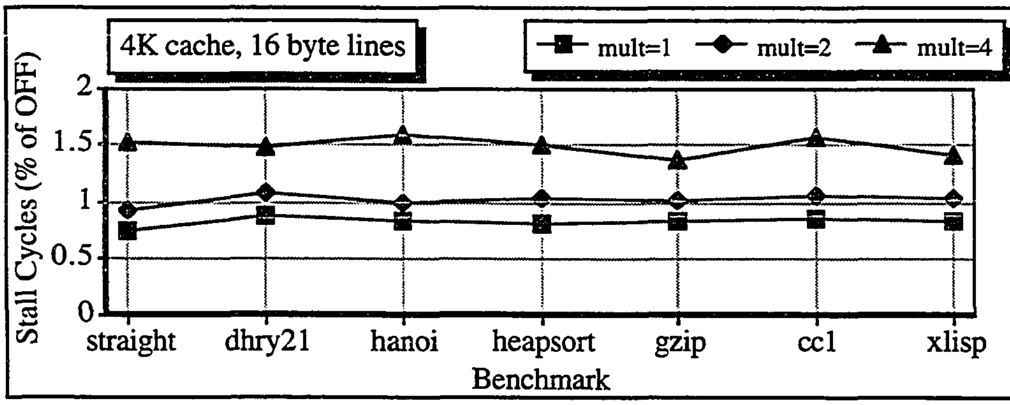

Figure 4.1: Comparing multipliers – 4K cache, 16 byte cache lines, prefetching on

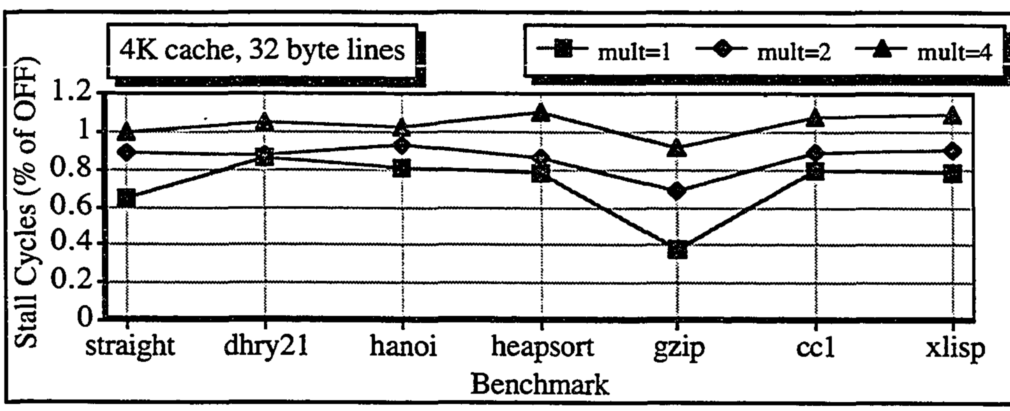

Figure 4.2: Comparing multipliers – 4K cache, 32 byte cache lines, prefetching on

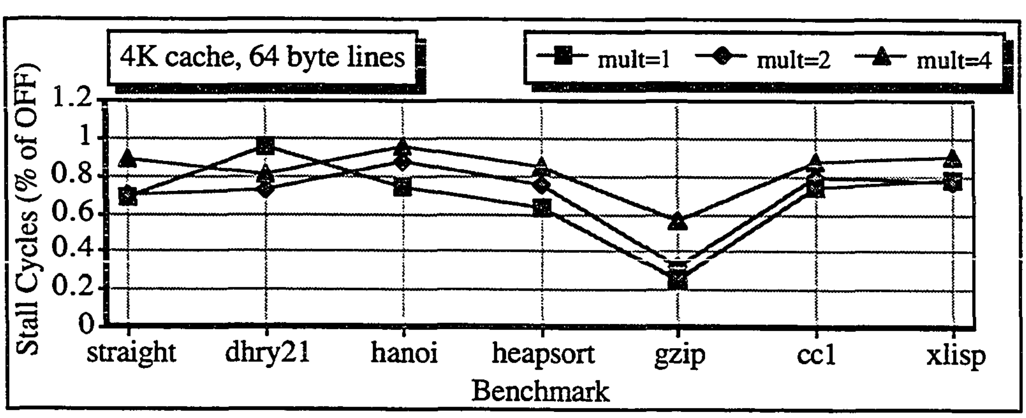

Figure 4.3: Comparing multipliers – 4K cache, 64 byte cache lines, prefetching on

A multiplier of one resulted in the best performance in every case except the dhrystone benchmark with a 64 byte cache line. In all other cases the smaller multiplier results in better performance. The dhrystone benchmark causes the inverted result due to its increased use of the bus for data. Therefore this benchmark performs better with less instruction prefetching. As will be seen later, this pattern persists in the dhrystone benchmark through the other experiments as well.

The next caches considered were 8K caches, with cache line sizes of 16, 32, 64, and 128 bytes. These caches are similar to the ones on-chip in the PowerPC 603, Intel Pentium, Intel 860-XR, Motorola 88110, DEC Alpha 21064 and MIPS R4000 processors. To date, no CPU uses 128 byte cache lines for their primary instruction cache, however this case is presented because it shows an interesting trend in the data, as will be described. The graphs for these caches are shown in Figures 4.4 through 4.7.

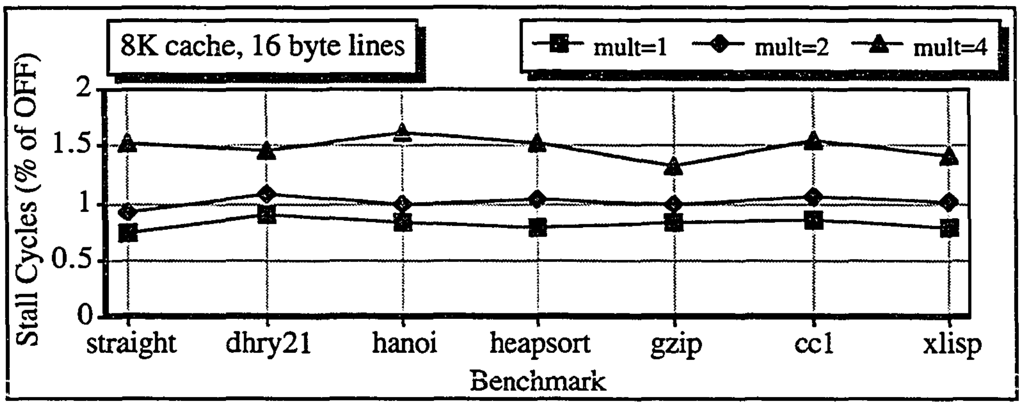

Figure 4.4: Comparing multipliers – 8K cache, 16 byte cache lines, prefetching on

Other than the dhrystone benchmark, which has inverted characteristics here as well, the multiplier of one works best for the 16 byte, 32 byte and 64 byte cache lines. With the 128 byte cache line, the difference between the three multiplier values is very small because the multiplier of four improves with larger cache lines. This is due to the fact that prefetching a larger cache line takes longer than prefetching a short line so the overhead of an incorrect guess becomes more significant.

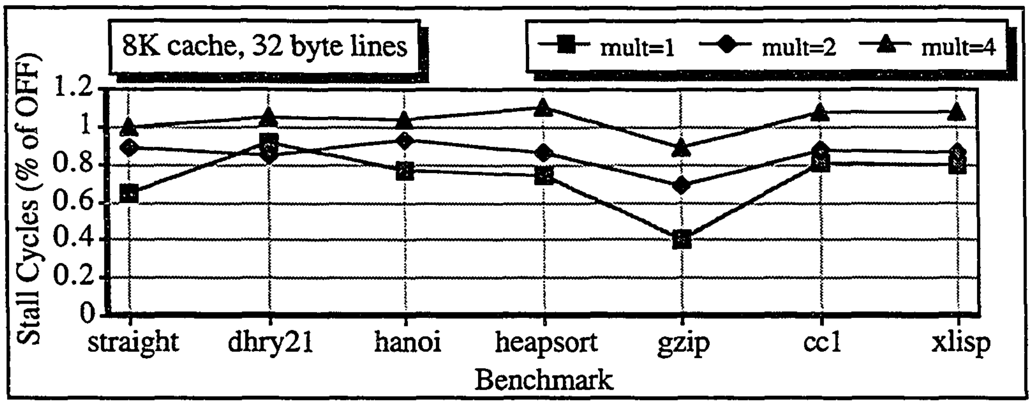

Figure 4.5: Comparing multipliers – 8K cache, 32 byte cache lines, prefetching on

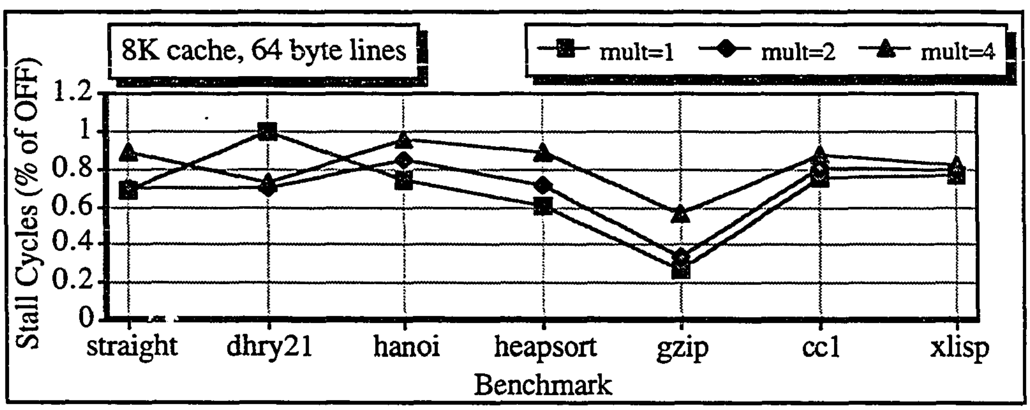

Figure 4.6: Comparing multipliers – 8K cache, 64 byte cache lines, prefetching on

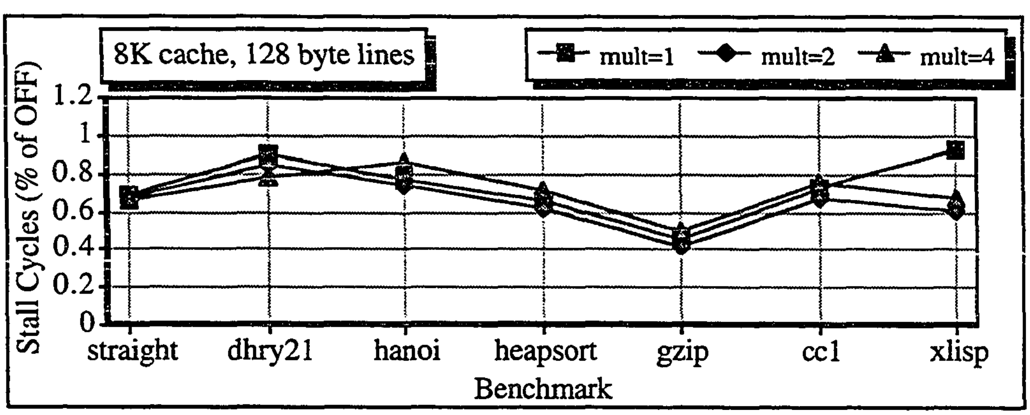

Figure 4.7: Comparing multipliers – 8K cache, 128 byte cache lines, prefetching on

The next caches considered were 16K caches, with cache line sizes of 32, 64, and 128 bytes. These caches are similar to the ones on-chip in the PowerPC 604, Intel 860-XP, DEC Alpha 21064A, MIPS R4200, MIPS R4400, and Fujitsu MicroSPARC-II processors. Again, no CPU uses 128 byte cache lines for their primary instruction cache, however this case is presented because it shows an interesting trend in the data, as will be described. The graphs for these caches are shown in Figures 4.8 through 4.10.

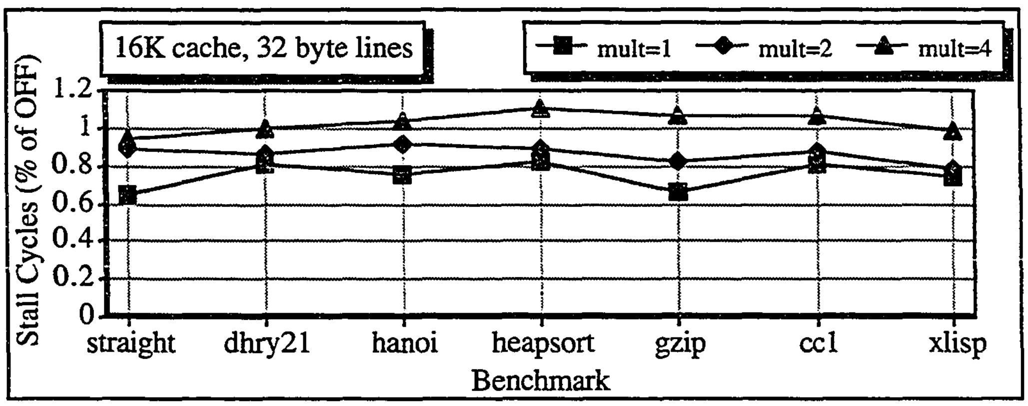

Figure 4.8: Comparing multipliers – 16K cache, 32 byte cache lines, prefetching on

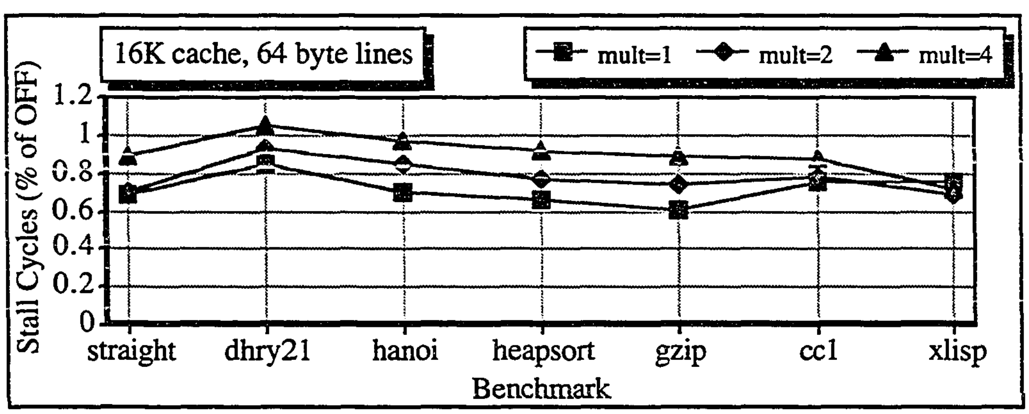

Figure 4.9: Comparing multipliers – 16K cache, 64 byte cache lines, prefetching on

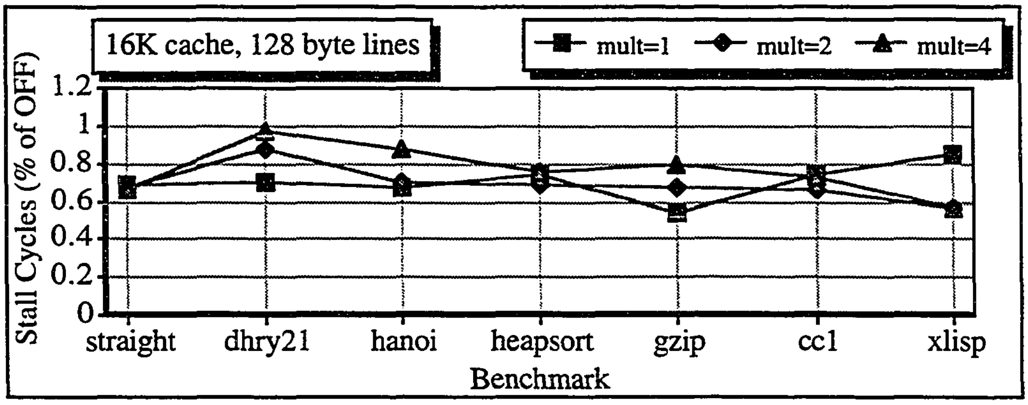

Figure 4.10: Comparing multipliers – 16K cache, 128 byte cache lines, prefetching on

There is more separation between the three multiplier values in the 16K caches. The values are also more consistent with each other and do not overlap as much. Both of these effects are due to the higher inherent hit ratio in the larger cache. The prefetching engine has less prefetching to do, and any speculative prefetches it makes will have a lower risk of replacing a useful item in the cache.

When looking at the graphs, it is clear that for small cache line sizes a multiplier of one (which indicates that the prefetched lines and demand fetched lines are the same size) results in the best performance for all cache sizes. As the cache line size is increased, the results for the multiplier of one get slightly worse, and the results for the larger multipliers get better. This again indicates that the overhead of prefetching cache lines becomes more significant as their size becomes larger, so the sub-line prefetching then becomes more of an advantage. However for the sizes typically used for on-chip primary caches in RISC CPU's today, the multiplier of one nearly always performs the best. For this reason, a multiplier of one will be used for the remainder of the analysis of the pipelined prefetching engine.

This observation also indicates that the prefetched lines are acting as a spacial reference source as well as a temporal source since the larger lines perform better. Larger lines provide more space for spacial locality use.

The same analysis was done for the one block look-ahead (OBL) method. The results are graphed in Figures 4.11 through 4.20.

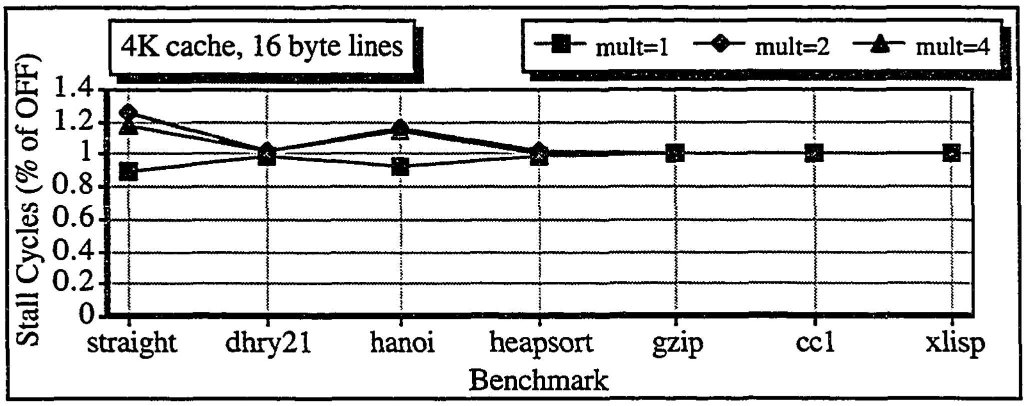

Figure 4.11: Comparing multipliers – 4K cache, 16 byte cache lines, OBL

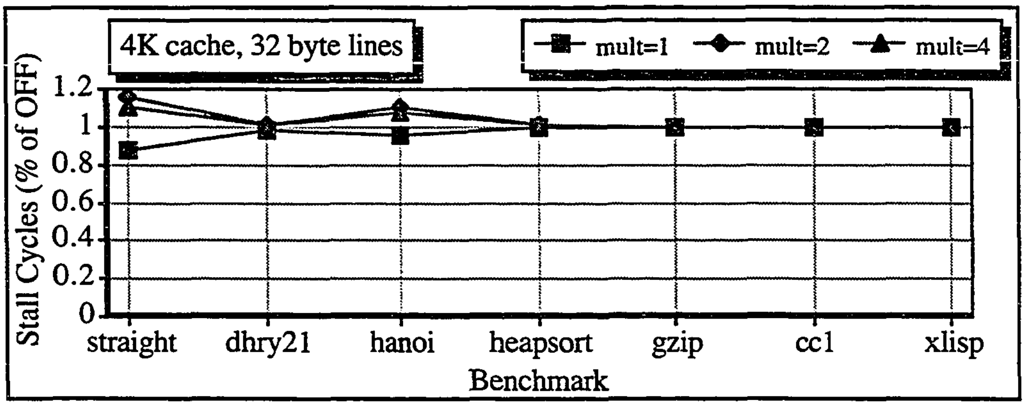

Figure 4.12: Comparing multipliers – 4K cache, 32 byte cache lines, OBL

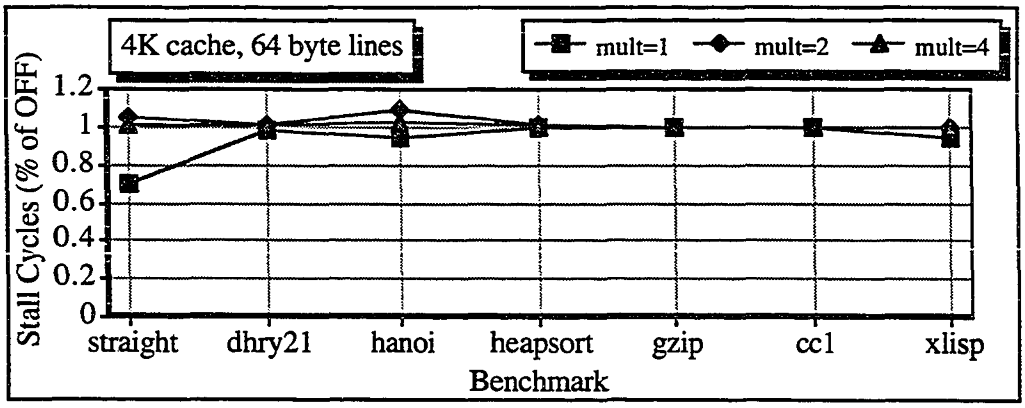

Figure 4.13: Comparing multipliers – 4K cache, 64 byte cache lines, OBL

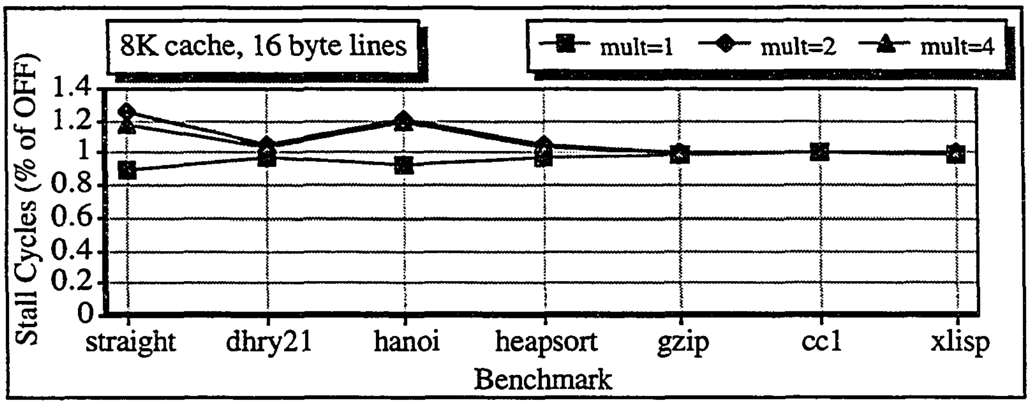

Figure 4.14: Comparing multipliers – 8K cache, 16 byte cache lines, OBL

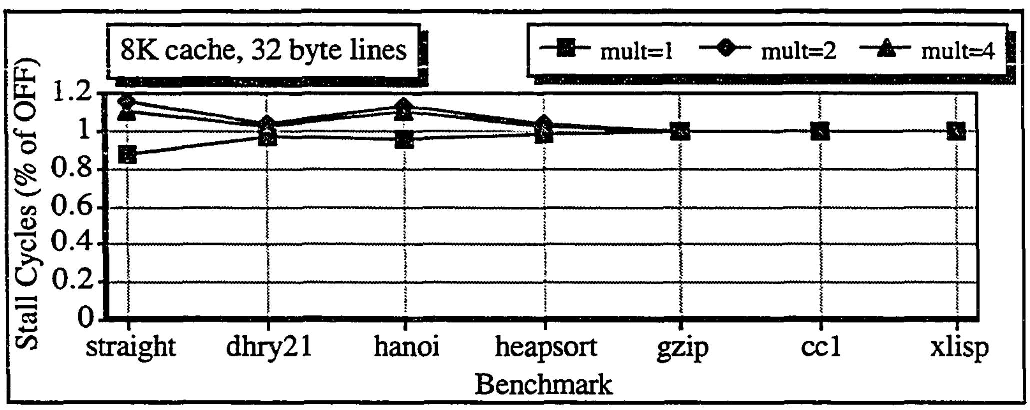

Figure 4.15: Comparing multipliers – 8K cache, 32 byte cache lines, OBL

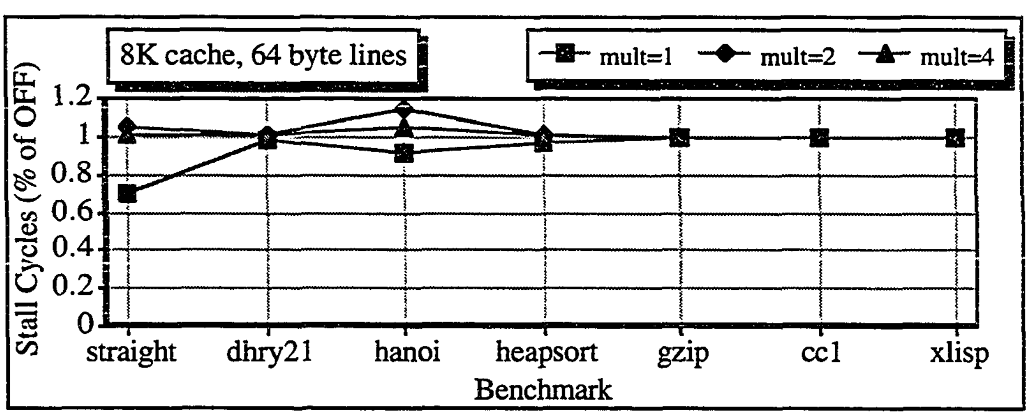

Figure 4.16: Comparing multipliers – 8K cache, 64 byte cache lines, OBL

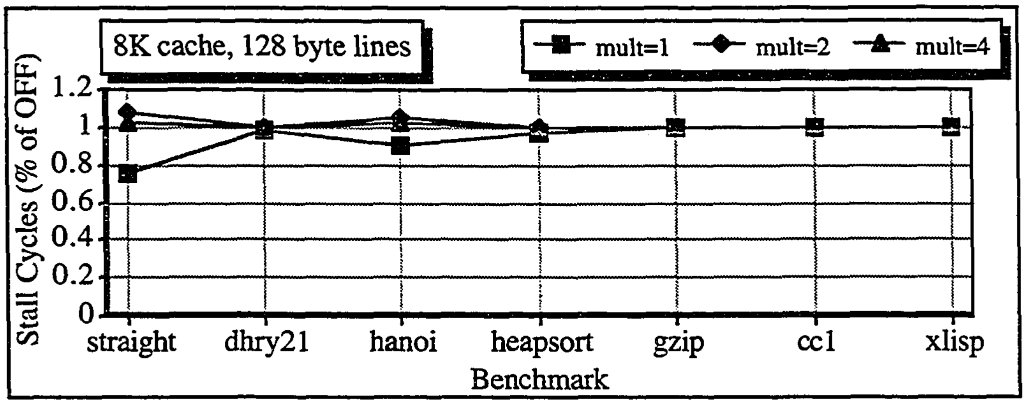

Figure 4.17: Comparing multipliers – 8K cache, 128 byte cache lines, OBL

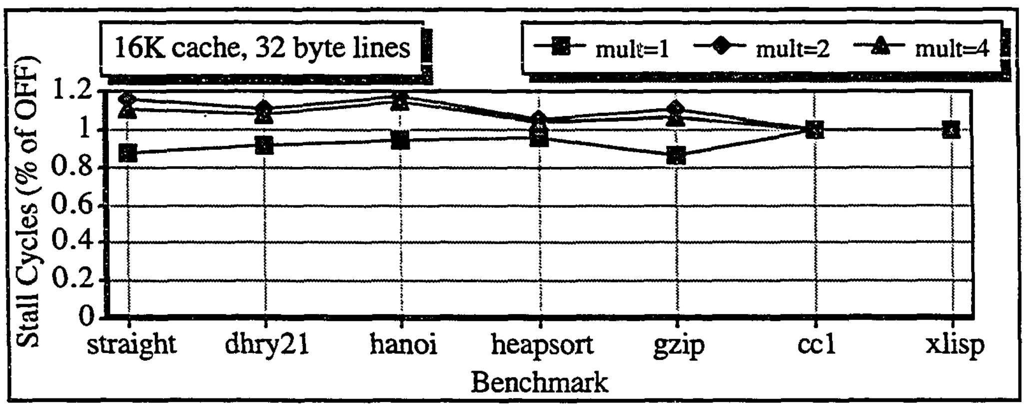

Figure 4.18: Comparing multipliers – 16K cache, 32 byte cache lines, OBL

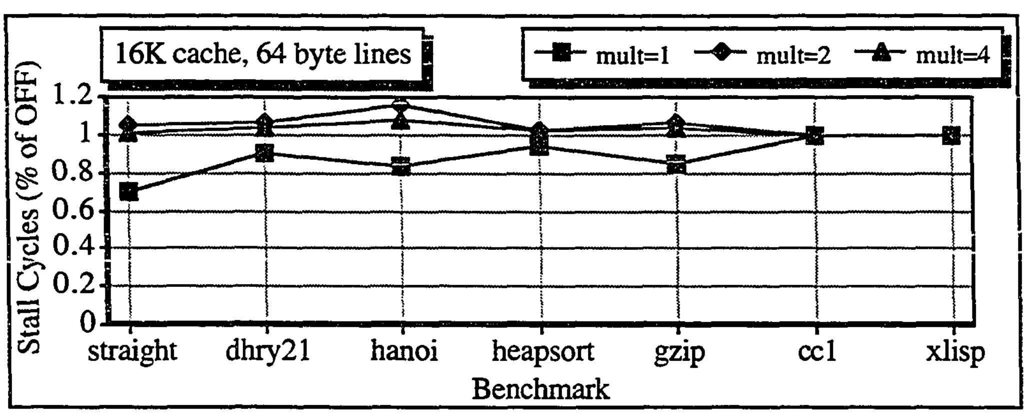

Figure 4.19: Comparing multipliers – 16K cache, 64 byte cache lines, OBL

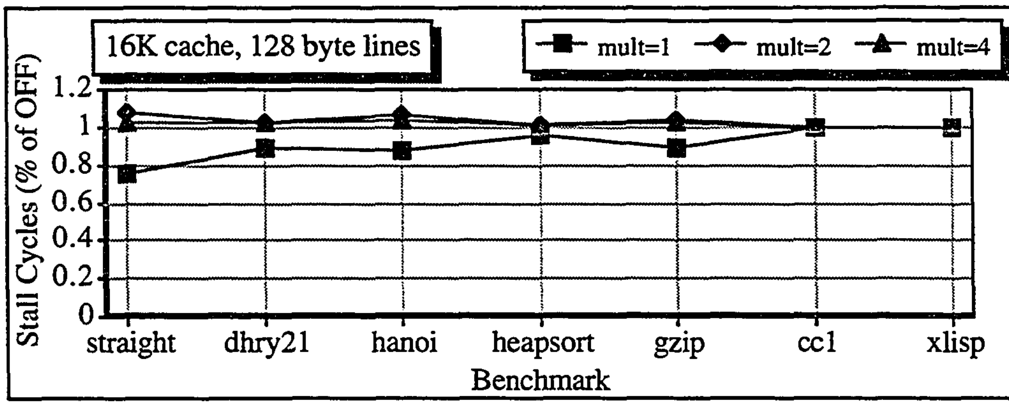

Figure 4.20: Comparing multipliers – 16K cache, 128 byte cache lines, OBL

The multiplier of one gives the best performance in almost every case. When looking at the data being graphed, there are two places where a multiplier of one performs worse than with larger multipliers. These occur in the cc1 and xlisp programs each with an 8K cache having 128 byte lines, and is only noted when calculated out to the fourth significant digit. From these graphs it is clear that OBL is acting as a spacial reference source more than a temporal source.

Note that the multipliers of two and four give worse performance than with no prefetching in every case. This is most likely caused by lines that were prefetched but not used by the CPU, which wastes bus bandwidth. The prefetching also replaces a line in the cache which may have been needed and will have to be reloaded again.

4.3 Instruction fetch stall cycles

The main goal of cache prefetching is to decrease memory latency by reducing the number of CPU stall cycles caused by cache misses. In the instruction stream, this means reducing the number of stall cycles caused by the instruction fetch stage because the instruction being fetched was not in the cache.

The SPIM simulator was modified to collect the number of instruction fetch stall cycles. When a cache miss occurred, a counter was incremented every simulated CPU clock cycle until the instruction was retrieved from memory. During the total run of a program, this statistic gives an accurate representation of the improvements made by any particular latency reduction technique.

As explained in Section 2.2.2, there is a throttling mechanism present in the prefetching engine so that it does not get too far ahead. This throttling mechanism is based on the number of branch instructions which have been predicted, but not yet executed. The simulations were run on each benchmark program, with the maximum lead ranging from one branch instruction, through five branch instructions.

The first caches considered were 1K and 2K caches each with 16 byte lines. Caches this small are not typically found in current RISC processors, but they represent how the prefetching engine works with small cache sizes and bring out one of its limitations. The resulting graphs are presented in Figures 4.21 and 4.22. The vertical axis indicates the normalized performance as compared to the same cache with no prefetching. A result of 1.0 indicates the same number of stall cycles as the same cache with prefetching turned off and numbers less than 1.0 indicate less stall cycles, and therefore better performance.

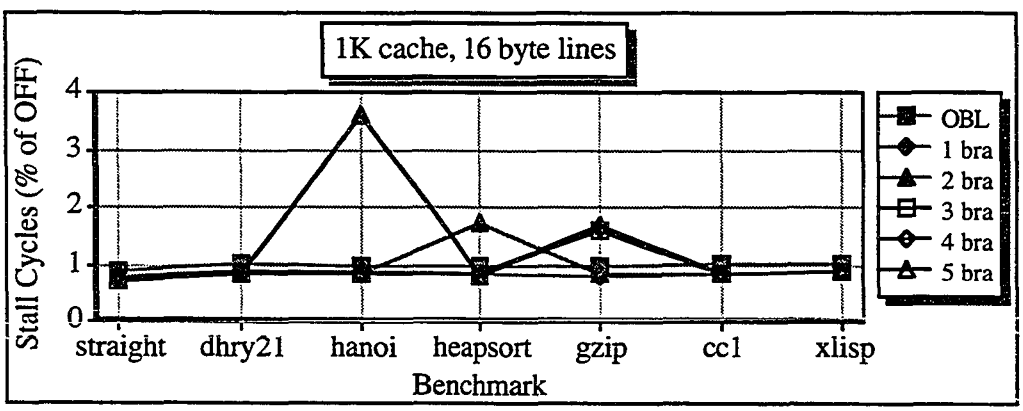

Figure 4.21: Instruction stall cycles – 1K cache, 16 byte cache lines

The prefetching engine can perform poorly when allowed a large lead and working with a small cache. This is partially due to the small number of cache lines present, because incorrectly prefetched lines will consume a significant portion of the cache, replacing potentially useful information. Table 4.2 shows that the problem becomes worse the farther ahead the prefetching engine is allowed to go because it has a greater potential of prefetching unneeded lines. Studying the assembly code for these programs shows that branch instructions occur every four to 16 instructions on average, which gives an average of around 30 bytes between branches. If the prefetch engine is allowed to be five branches ahead, this is 150 bytes, which is 15% of the 1K cache. That is a significant portion of the cache to consume on a guess.

Note in Figure 4.25 that a 4K cache with 64 byte cache lines performs better even though it has the same number of cache lines. This indicates that poor performance is not so much a function of the number of lines in the cache, but rather the total size of the cache.

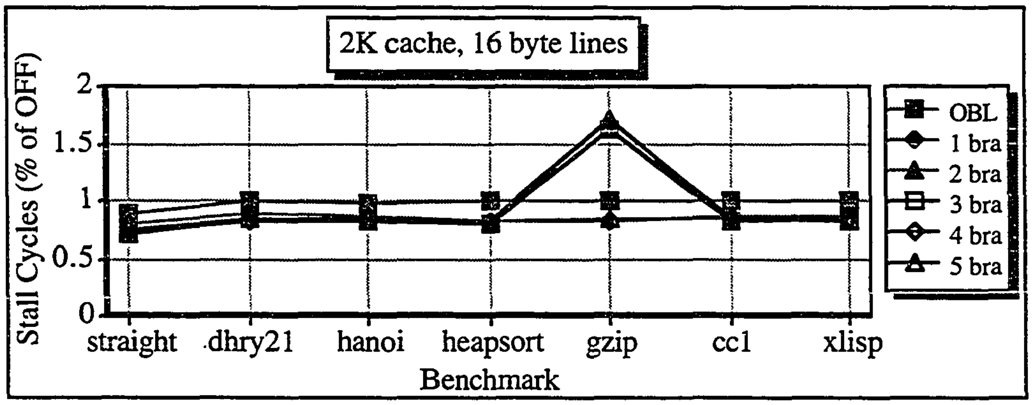

Figure 4.22: Instruction stall cycles – 2K cache, 16 byte cache lines

Table 4.2: Percent decrease in stall cycles – 1K and 2K caches

Cache configuration	OBL	1 bra	2 bra	3 bra	4 bra	5 bra
1K cache, 16 byte lines	1.51%	14.68%	2.48%	6.25%	4.57%	-34.52%
2K cache, 16 byte lines	1.82%	15.50%	17.04%	7.52%	7.36%	6.02%

The next caches considered for this experiment were 4K caches with 16 byte, 32 byte, and 64 byte cache lines, which are similar to the on-chip caches in the Intel 960CF, Motorola 68040, ARM 610, and TI MicroSPARC. The graphs for these caches are shown in Figures 4.23 through 4.25.

In the 4K caches, the prefetching engine performs quite well as compared to both no prefetching and the OBL prefetching. It is also important to note that the farther ahead the prefetching engine is allowed to go, the less instruction stall cycles occur. It also appears that with larger cache lines, the prefetching engine performs comparatively better. The average gain of around 30% is a significant improvement.

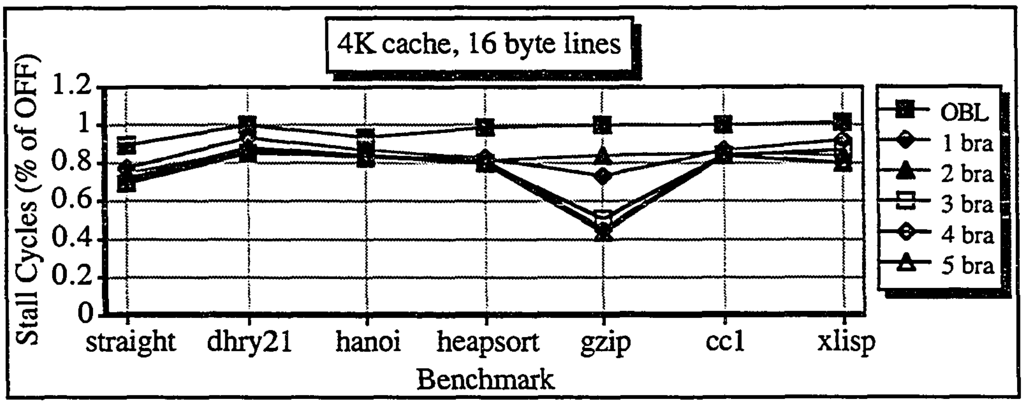

Figure 4.23: Instruction stall cycles – 4K cache, 16 byte cache lines

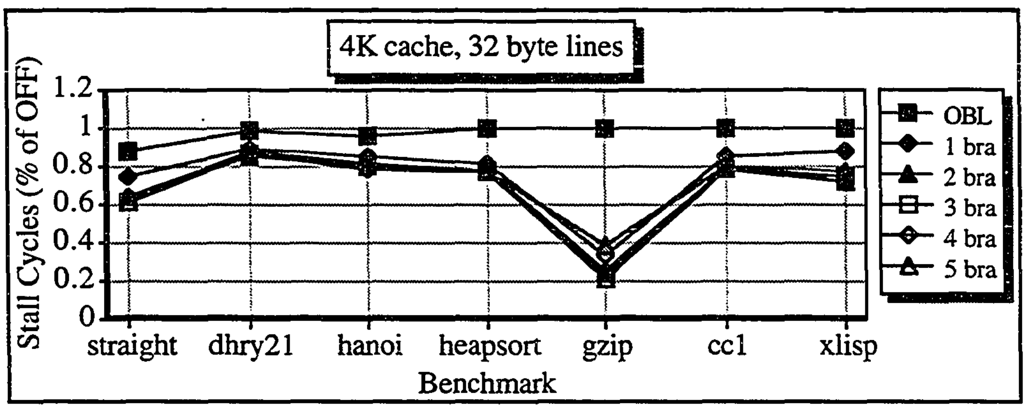

Figure 4.24: Instruction stall cycles – 4K cache, 32 byte cache lines

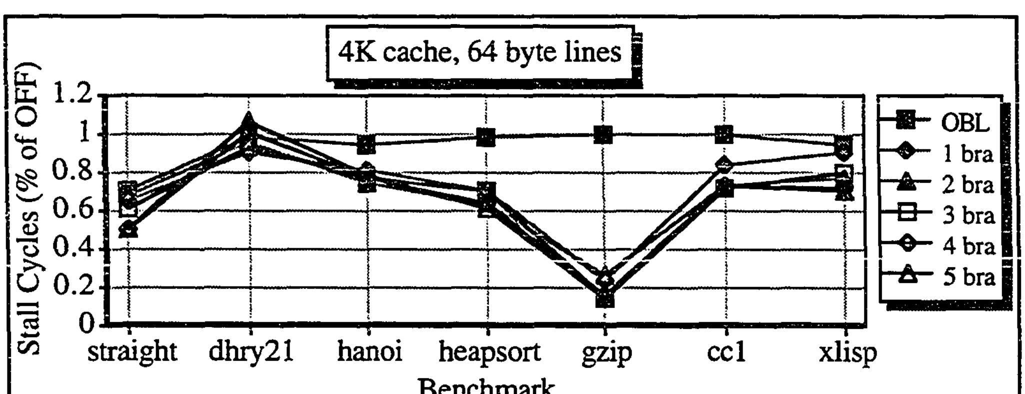

Figure 4.25: Instruction stall cycles – 4K cache, 64 byte cache lines

Table 4.3: Percent decrease in stall cycles – 4K caches

Cache configuration	OBL	1 bra	2 bra	3 bra	4 bra	5 bra
4K cache, 16 byte lines	2.56%	15.62%	17.25%	22.83%	23.83%	24.07%
4K cache, 32 byte lines	2.66%	23.12%	27.86%	31.04%	30.84%	31.26%
4K cache, 64 byte lines	6.09%	27.71%	31.50%	32.66%	34.87%	34.94%

Next, 8K caches with cache line sizes of 16, 32, and 64 bytes are presented. These caches are similar to the ones on-chip in the PowerPC 603, Intel Pentium, Intel 860-XR, Motorola 88110, DEC Alpha 21064 and MIPS R4000 processors. The graphs for these caches are shown in Figures 4.26 through 4.28.

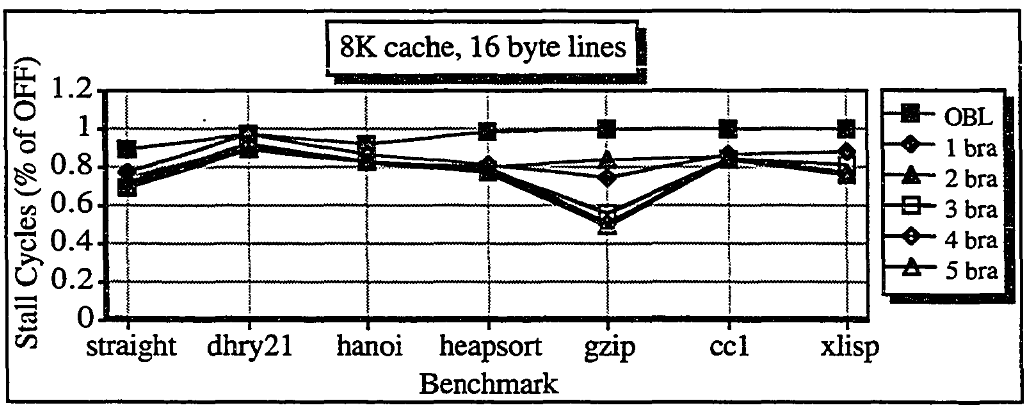

Figure 4.26: Instruction stall cycles – 8K cache, 16 byte cache lines

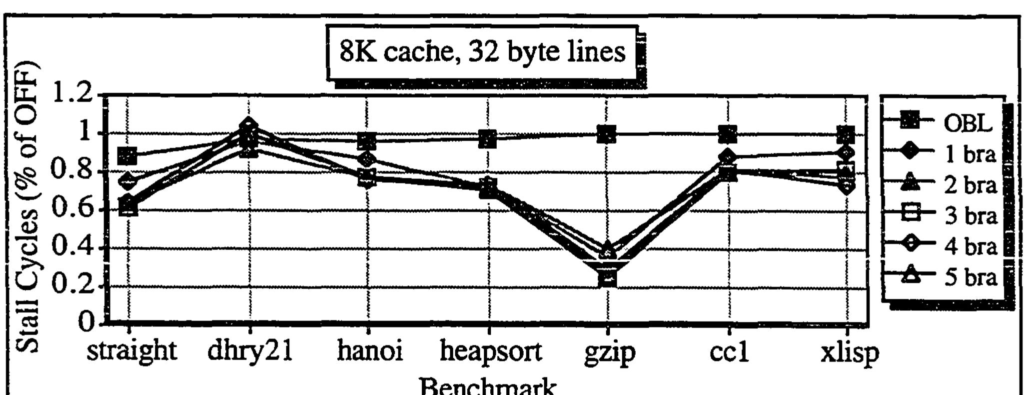

Figure 4.27: Instruction stall cycles – 8K cache, 32 byte cache lines

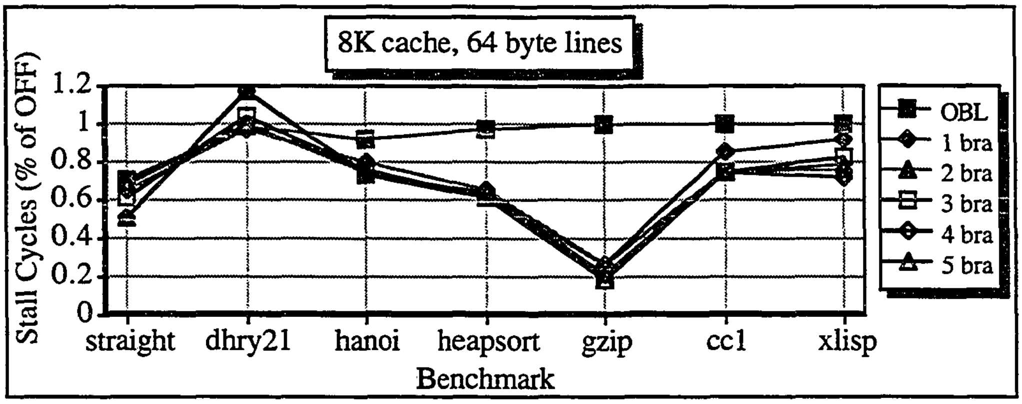

Figure 4.28: Instruction stall cycles – 8K cache, 64 byte cache lines

Table 4.4: Percent decrease in stall cycles – 8K caches

Cache configuration	OBL	1 bra	2 bra	3 bra	4 bra	5 bra
8K cache, 16 byte lines	3.44%	15.52%	17.93%	22.60%	23.37%	24.47%
8K cache, 32 byte lines	3.04%	22.12%	27.36%	28.62%	28.48%	29.31%
8K cache, 64 byte lines	6.20%	27.02%	31.04%	31.52%	32.20%	32.08%

In the 8K caches, the prefetching engine also performs quite well as compared to both no prefetching and the OBL prefetching. Here too, the farther ahead the prefetching engine is allowed to go, the less instruction stall cycles occur. The average gain of around 30% is significant improvement.

The next caches considered were 16K caches, with cache line sizes of 32 and 64 bytes. These caches are similar to the ones on-chip in the PowerPC 604, Intel 860-XP, DEC Alpha 21064A, MIPS R4200, MIPS R4400, and Fujitsu MicroSPARC-II processors. The graphs for these caches are shown in Figures 4.29 and 4.30.

As the caches become larger, there is more separation gained from larger throttling leads. This is most likely attributed to the naturally higher hit ratio present in larger caches. The prefetching engine can therefore prefetch farther forward in the instruction stream before reaching a missing line. It is also less of a risk to prefetch many lines in a large cache because they consume a smaller percentage of the total storage.

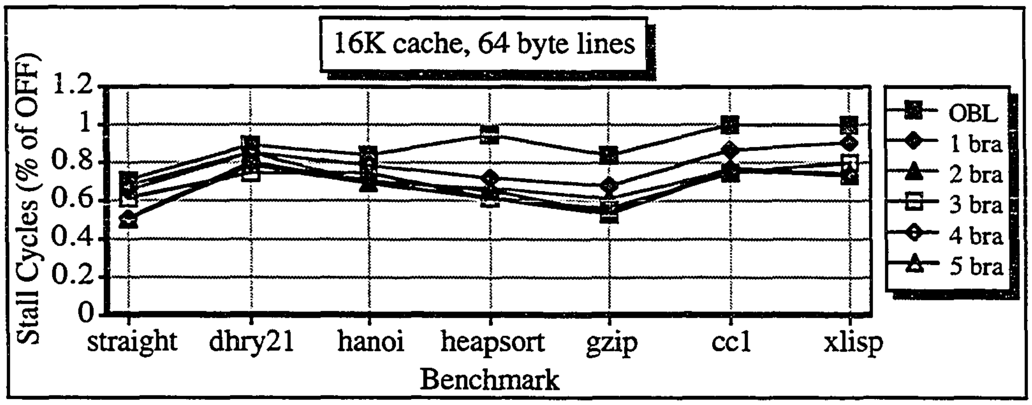

Figure 4.29: Instruction stall cycles – 16K cache, 32 byte cache lines

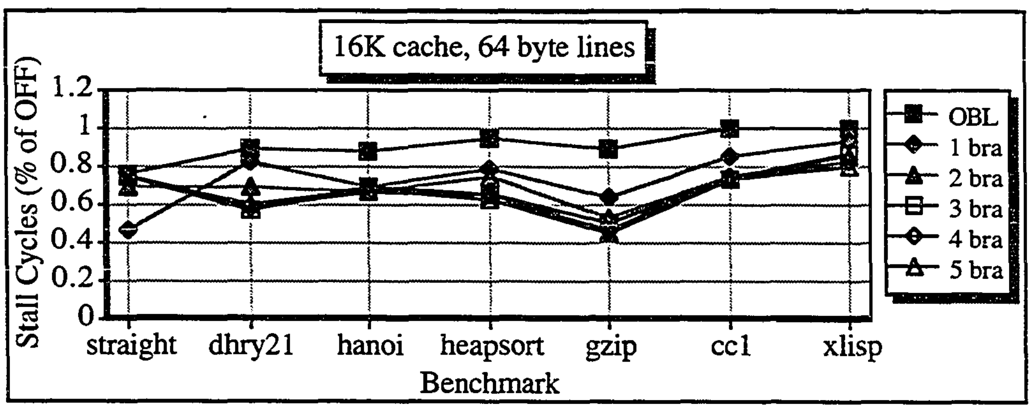

Figure 4.30: Instruction stall cycles – 16K cache, 64 byte cache lines

Table 4.5: Percent decrease in stall cycles – 16K caches

Cache configuration	OBL	1 bra	2 bra	3 bra	4 bra	5 bra
16K cache, 32 byte lines	6.44%	17.81%	25.15%	27.15%	28.49%	29.14%
16K cache, 64 byte lines	10.97%	21.92%	28.32%	30.88%	32.55%	33.37%

It is clear that the pipelined prefetching engine generally performs very well. It performs only slightly worse for the dhrystone benchmark than it does for the other programs so it appears to be a fairly well rounded solution, working for both general purpose and scientific programs. The reason for the diminished performance in the dhrystone benchmark is that there is less bus bandwidth available because of the higher number of data fetches that occur in this program. The prefetching engine always performs best with the gzip program because it is a very computationally intensive program and performs comparatively fewer data fetches per instruction executed. The results for the straight program indicate that it works well for filling the cache from a cold start as well.

At least one of the throttling values outperforms OBL in every case and in every program tested. It is also true that in general, the farther the PAC is allowed to get ahead, the better the prefetching performs, although as can be seen, there is not a large difference between the values.

The programs were each simulated for 64 different cache configurations with cache lines ranging from one word lines to 128 word lines, and from 8 lines to 1024 lines in the cache. The number of instruction stalls was collected and are plotted in Figures 4.31 through 4.38. One axis represents the number of lines in the cache, and the other is a combination of the number of words in each line plus, OBL and each throttling value.

The prefetching engine gives good results for most cache sizes, and only in some very unlikely configurations does it perform poorly. For example with the gzip program in a cache with 8 lines of 64 words each, performance with prefetching on is much worse than with prefetching off. This again, is due to the large portion of the cache which can be consumed by prefetched lines, which may replace lines needed by the CPU.

4.4 Average PAC lead

Instruction stall times give a good macroscopic indication of the improvements made by a particular technique, but it is also important to see how well it is performing at a microscopic level. The pipelined prefetching engine works by scanning forward in the instruction stream, interpreting instructions as it goes. Its success depends on how far ahead of the PC it stays. Recall that the memory latency time for these simulations is five cycles, so it would be best if the PAC could on average maintain at least a five instruction lead. The graphs in Figures 4.39 through 4.49 show the average number of instructions the PAC stays ahead of the PC during the execution of the programs.

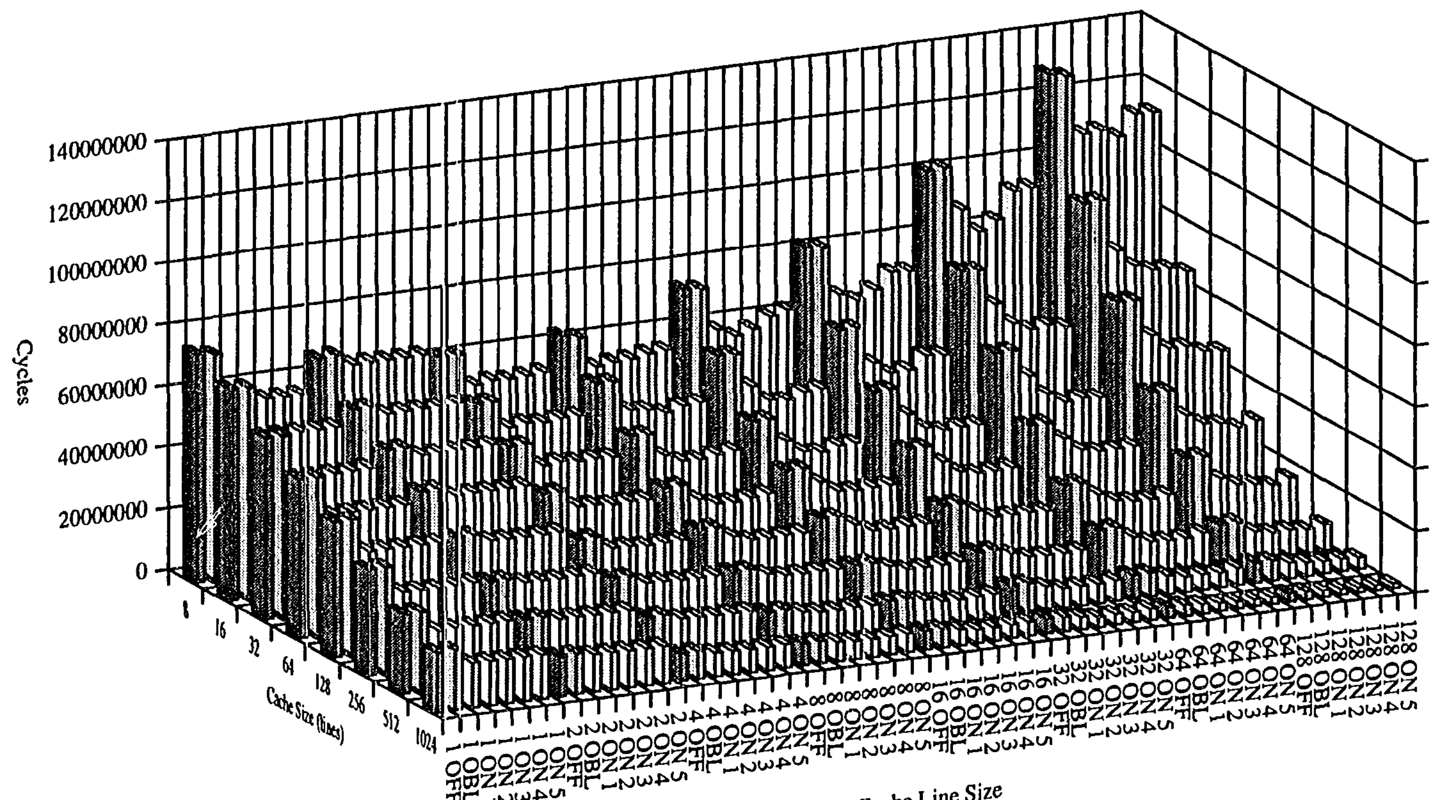

Figure 4.31: Instruction stall cycles (3D) cc1

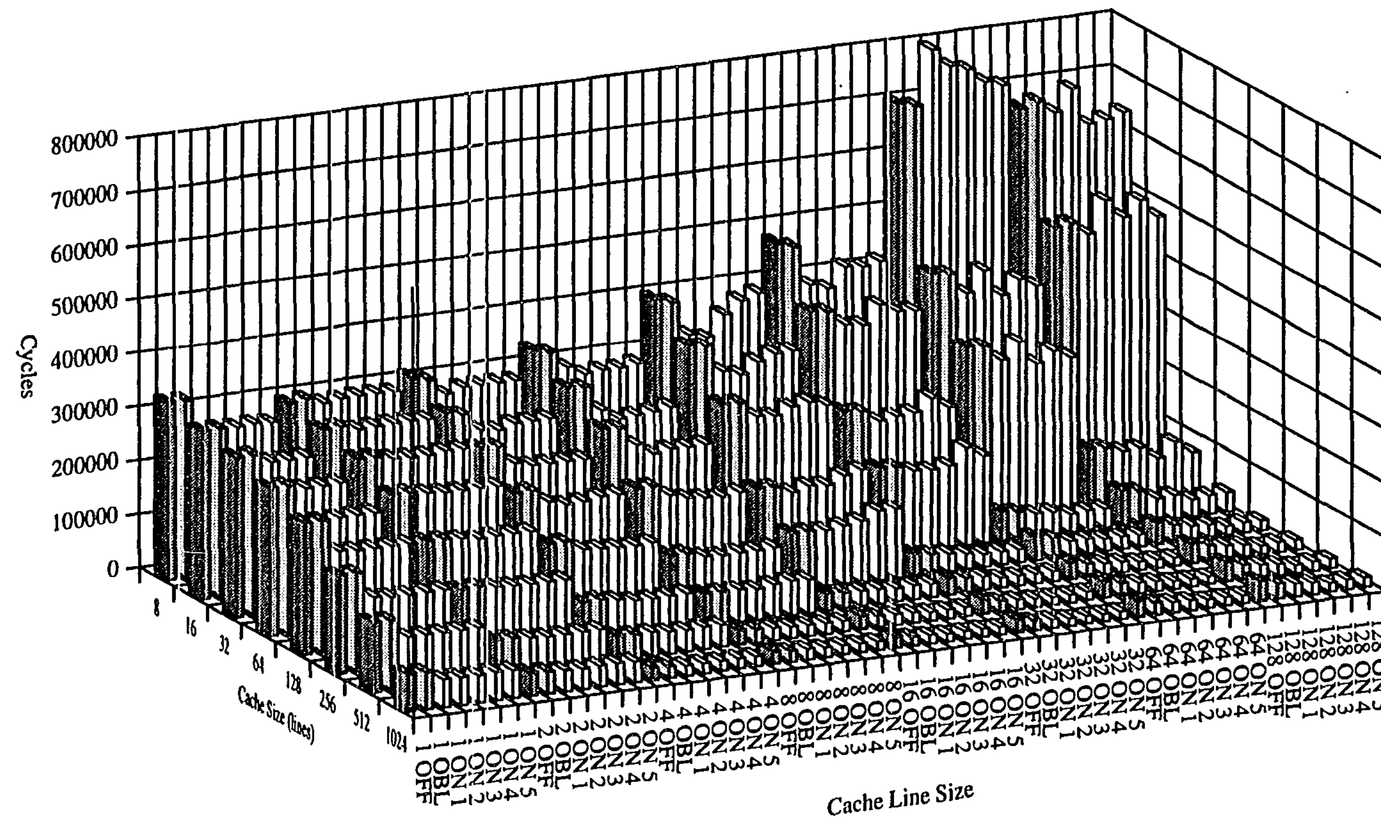

Figure 4.32: Instruction stall cycles (3D) dhry21

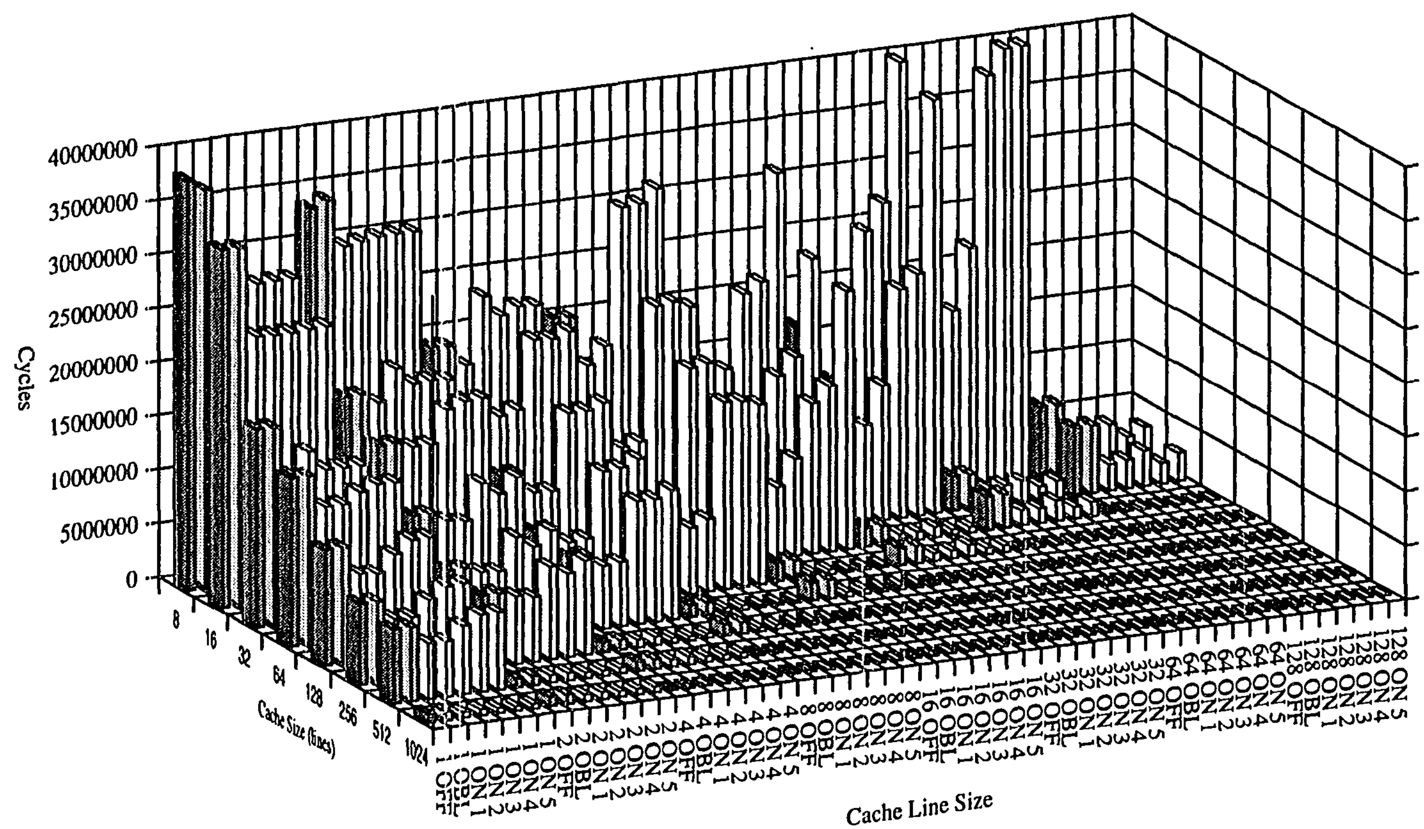

Figure 4.33: Instruction stall cycles (3D) gzip

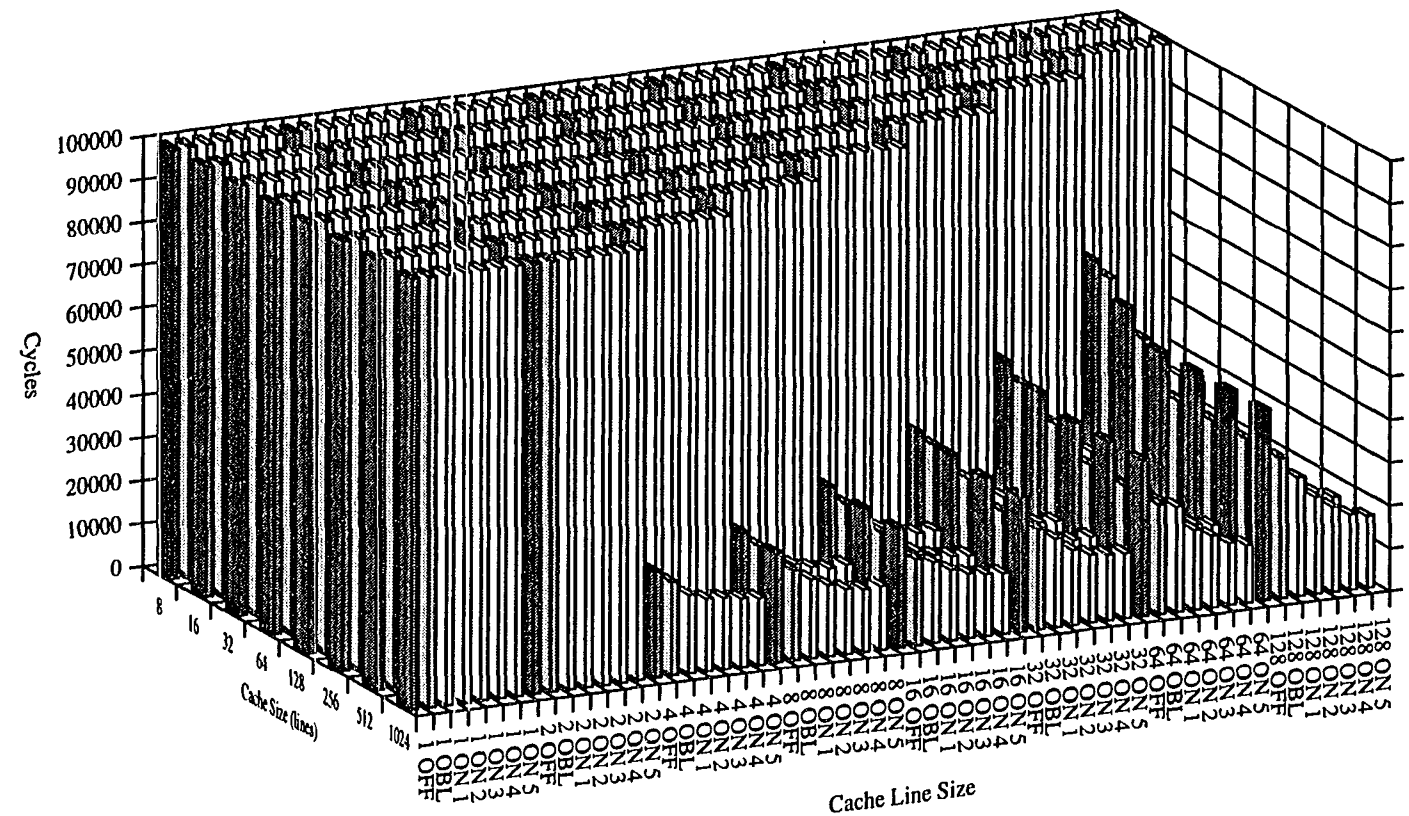

Figure 4.34: Instruction stall cycles (3D) gzip

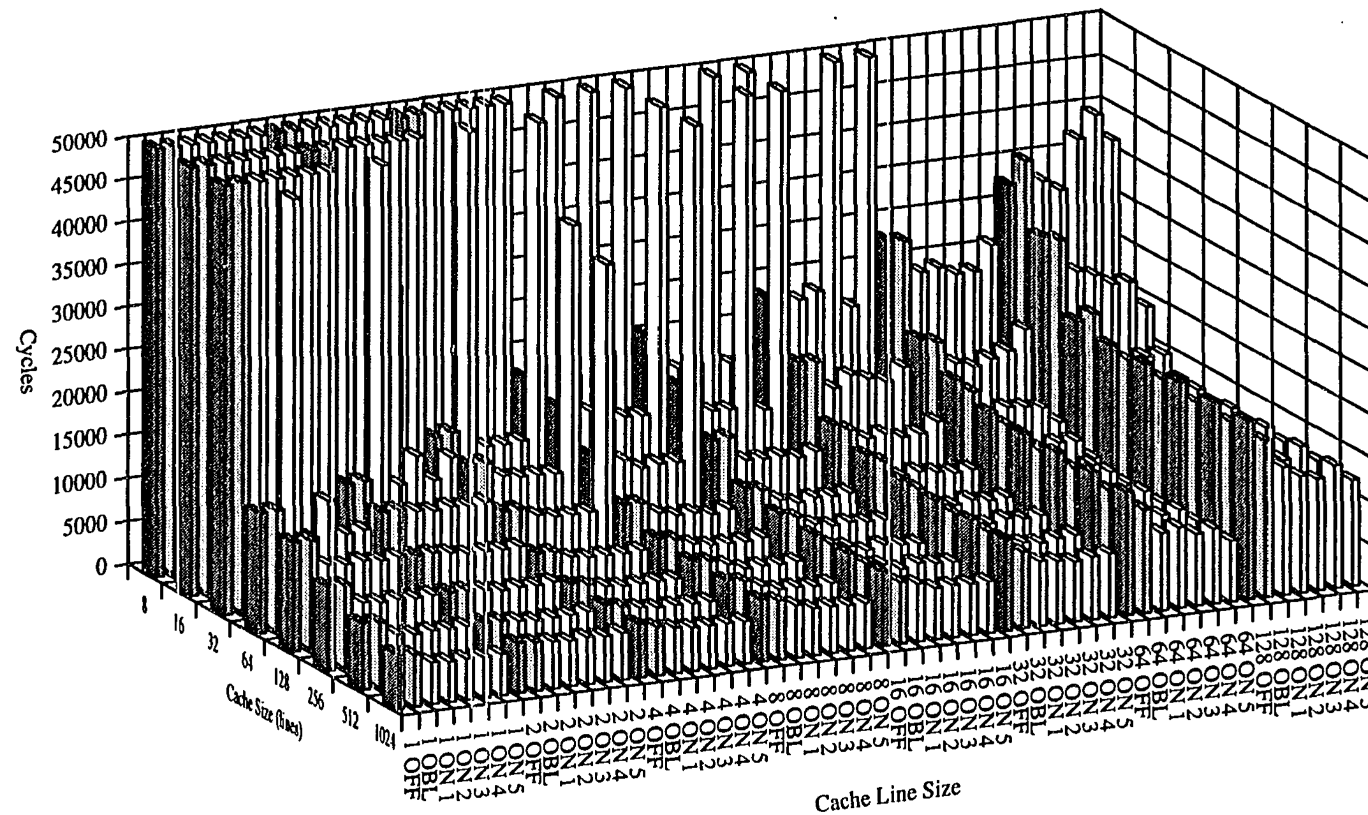

Figure 4.35: Instruction stall cycles (3D) hanoi

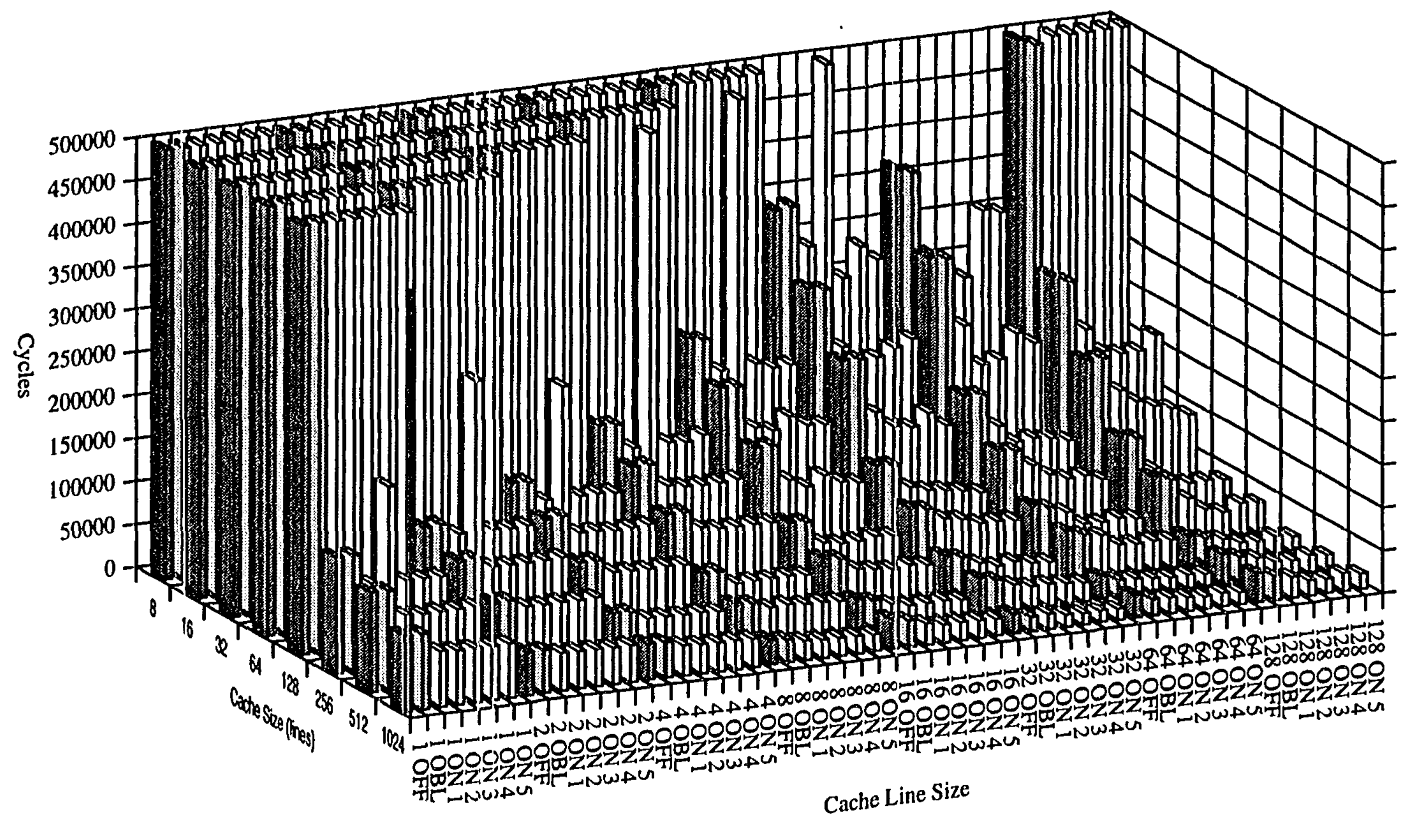

Figure 4.36: Instruction stall cycles (3D) heapsort

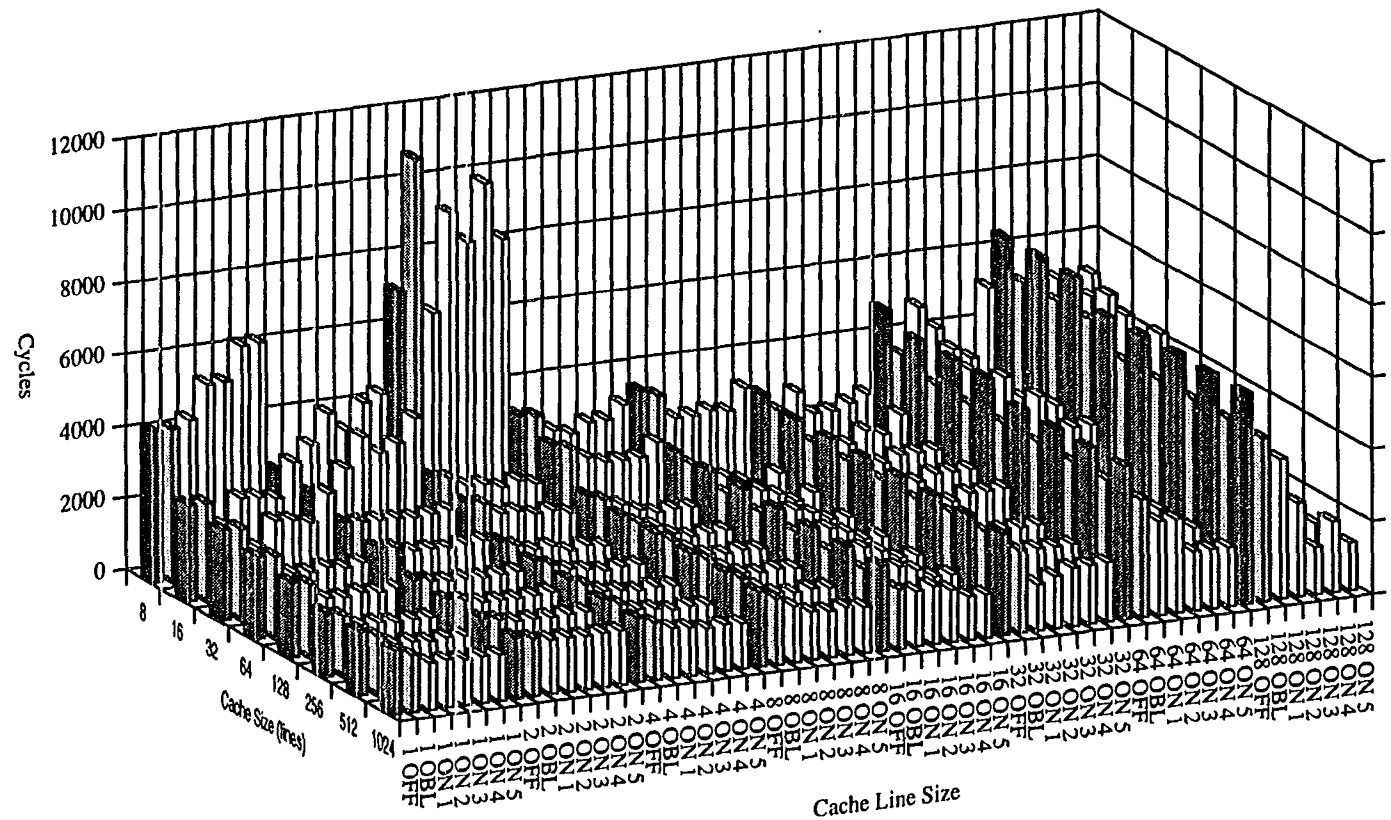

Figure 4.37: Instruction stall cycles (3D) straight

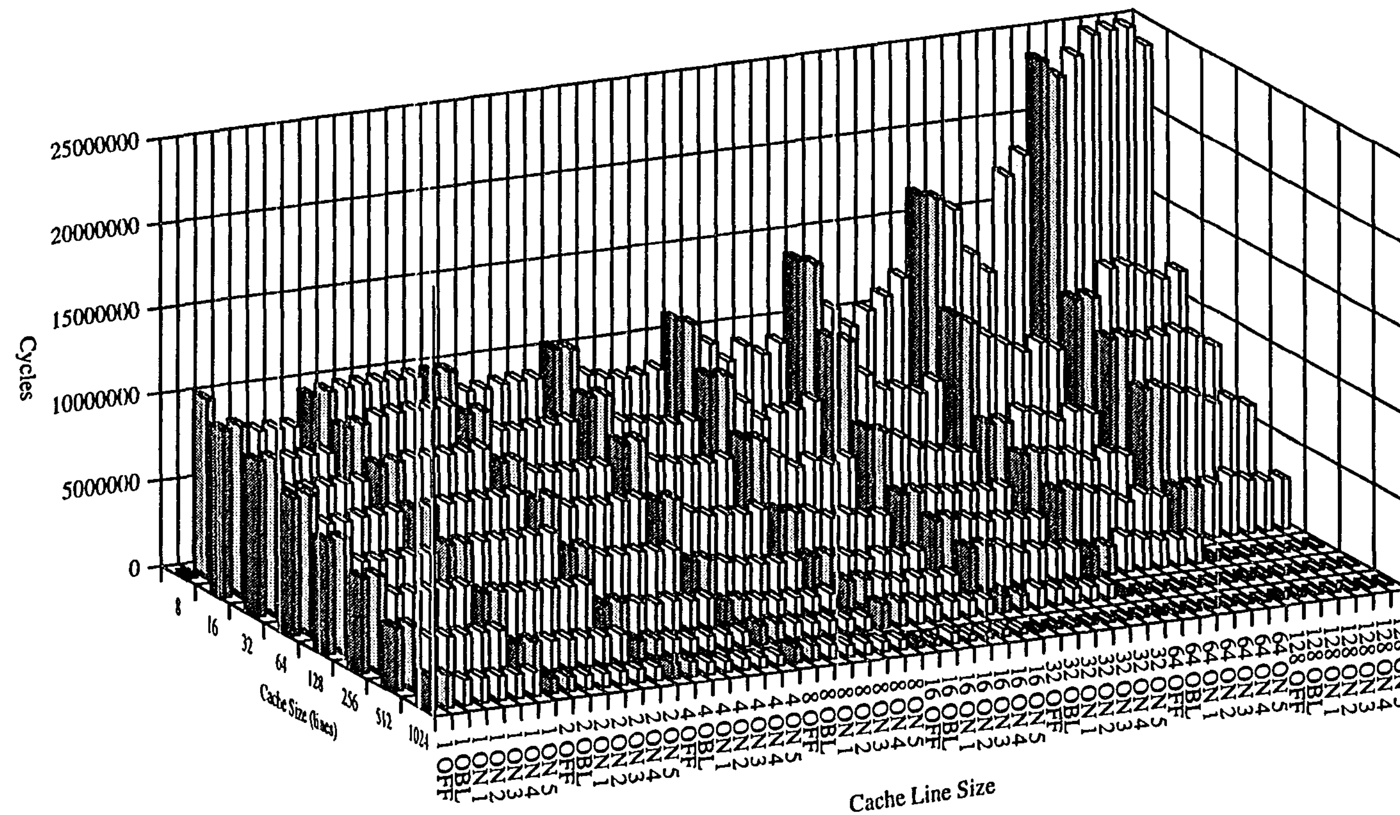

Figure 4.38: Instruction stall cycles (3D) xlisp

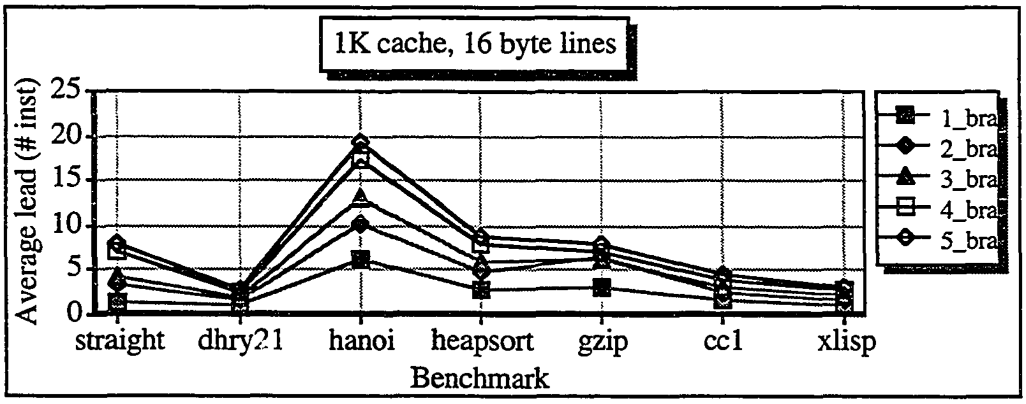

Figure 4.39: Average PAC lead – 1K cache, 16 byte cache lines

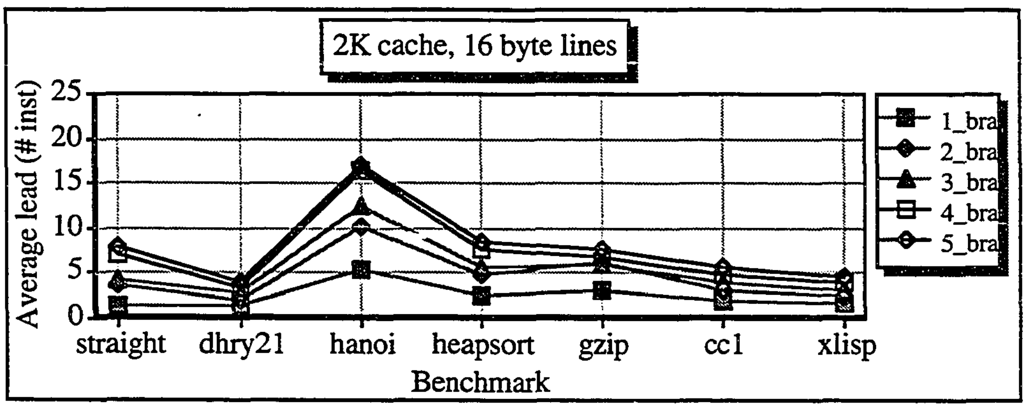

Figure 4.40: Average PAC lead – 2K cache, 16 byte cache lines

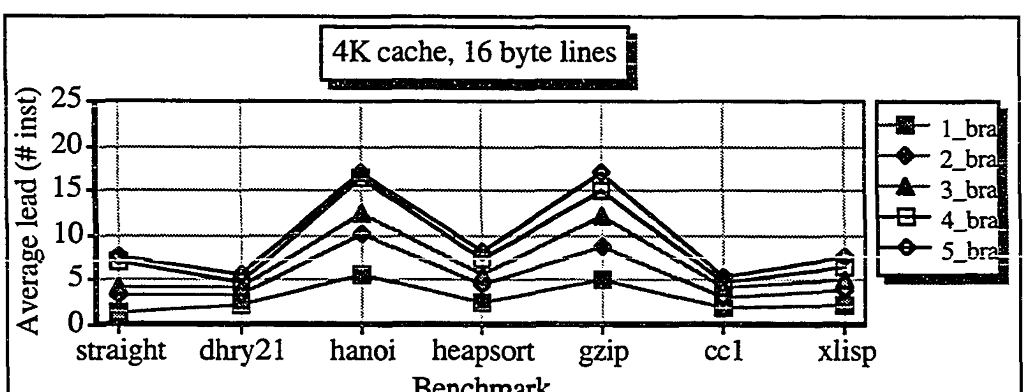

Figure 4.41: Average PAC lead – 4K cache, 16 byte cache lines

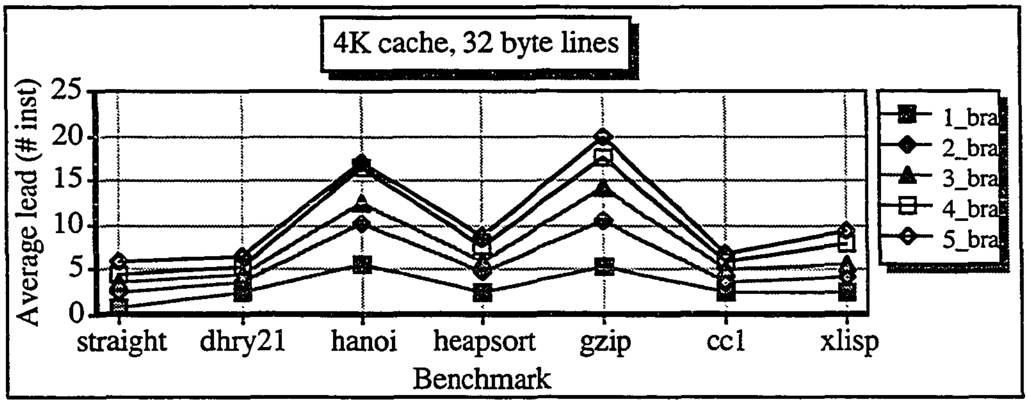

Figure 4.42: Average PAC lead – 4K cache, 32 byte cache lines

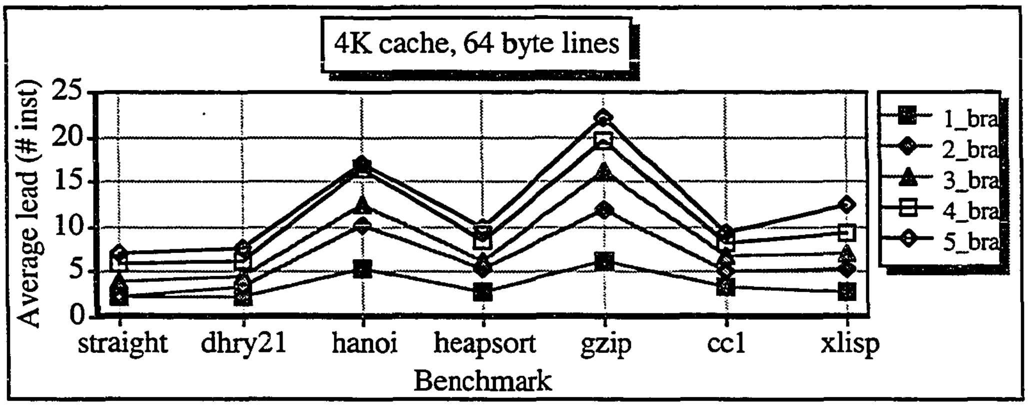

Figure 4.43: Average PAC lead – 4K cache, 64 byte cache lines

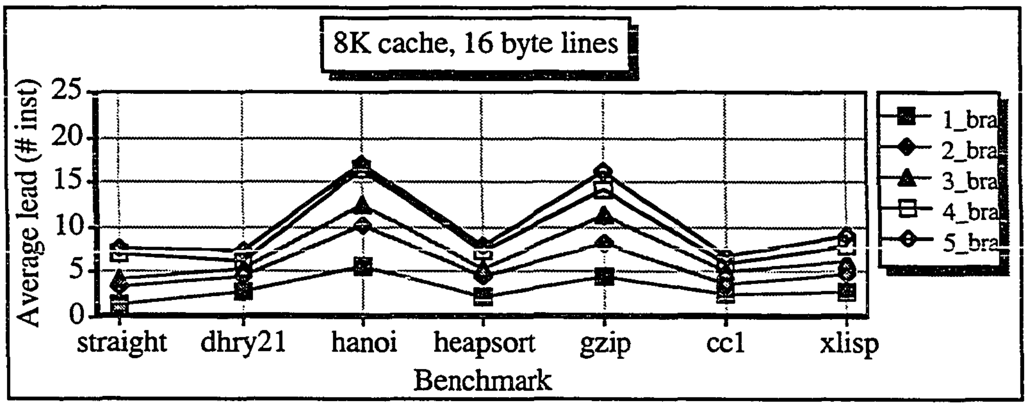

Figure 4.44: Average PAC lead – 8K cache, 16 byte cache lines

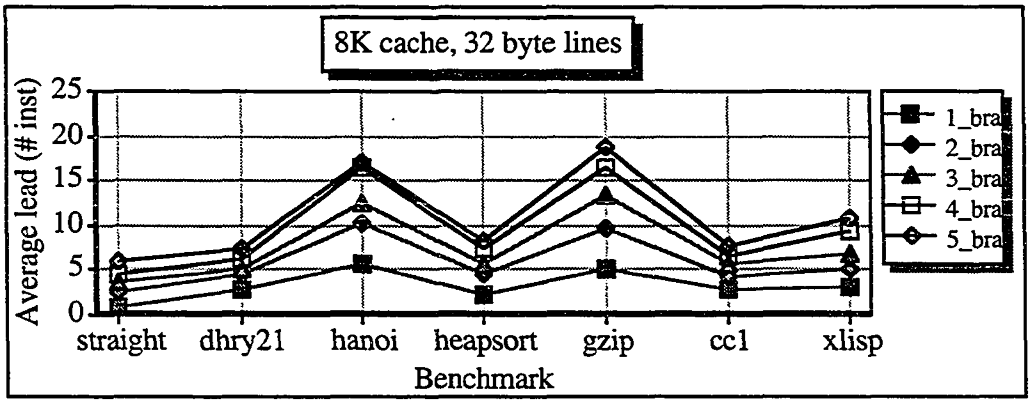

Figure 4.45: Average PAC lead – 8K cache, 32 byte cache lines

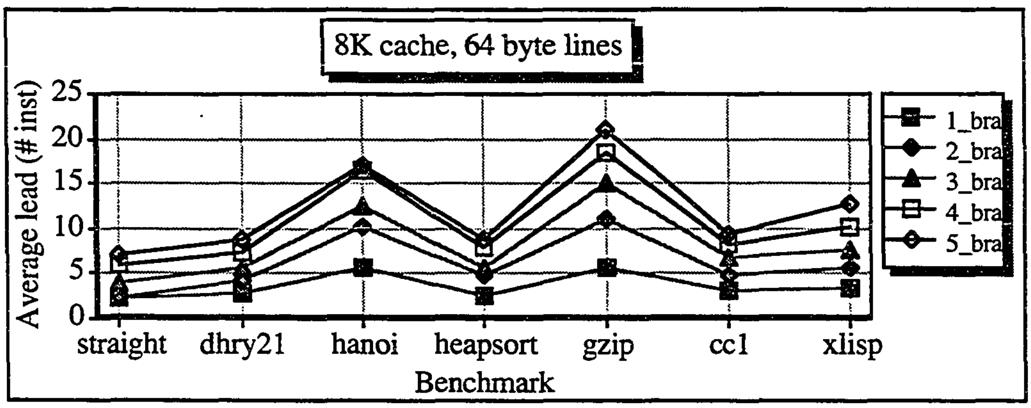

Figure 4.46: Average PAC lead – 8K cache, 64 byte cache lines

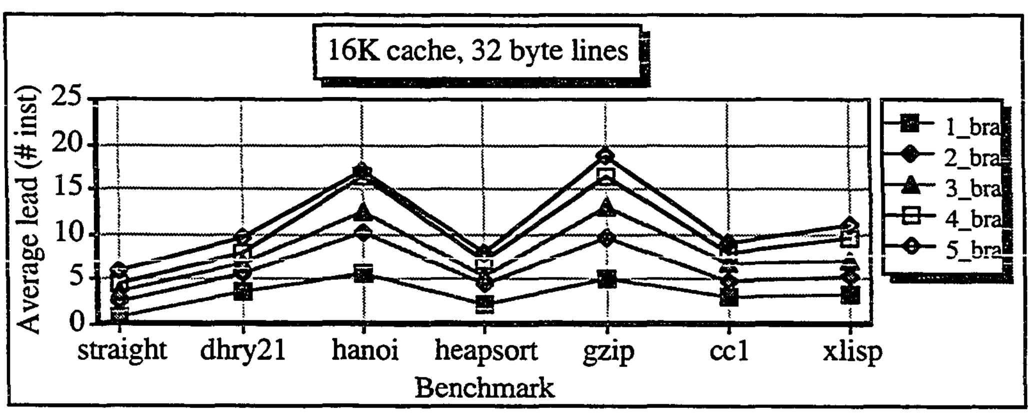

Figure 4.47: Average PAC lead – 16K cache, 32 byte cache lines

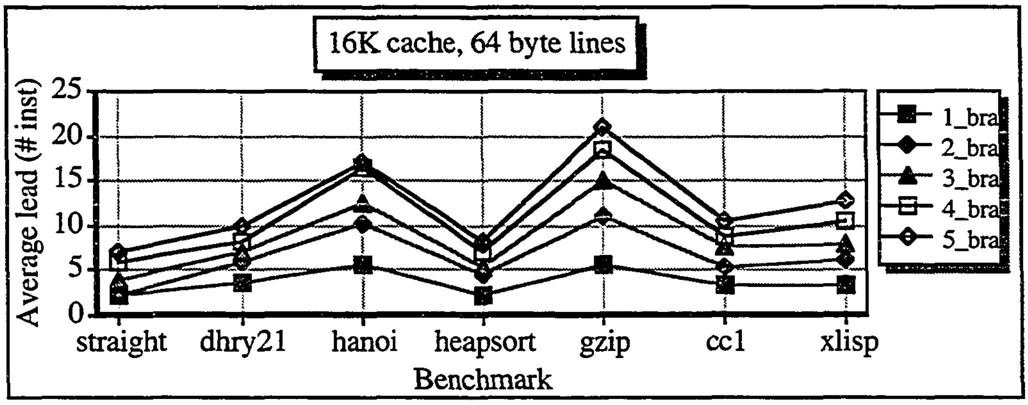

Figure 4.48: Average PAC lead – 16K cache, 64 byte cache lines

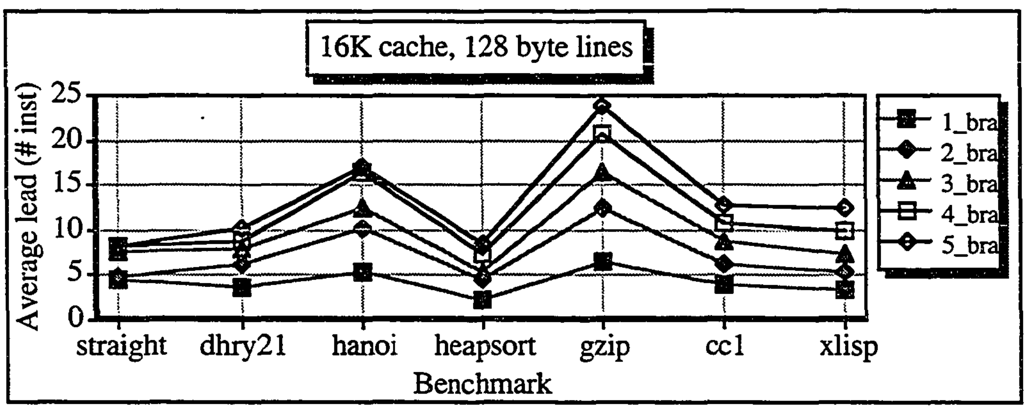

Figure 4.49: Average PAC lead – 16K cache, 128 byte cache lines

There are a few important things to observe from the graphs. First, the average lead (i.e. how far ahead the PAC is in the instruction stream) is dependent on the throttling value, as one would expect. The higher throttling values consistently get farther ahead than the smaller throttling values. It is also important to note that the average lead time is longer with longer cache line sizes. This also matches our intuition because with larger cache lines, the prefetching engine can get farther ahead before it reaches the end of a cache line.

With throttling values of two or more branches, the prefetching engine maintains the minimum five instruction lead. This helps keep the prefetching ahead of the CPU by at least one memory latency time, thereby eliminating many stall cycles. This explains why it performs as well as it does at the macroscopic level.

4.5 Bus bandwidth consumption

As with any prediction technique, the increases in performance do not come for free. In this case there is an increased consumption of the memory bus bandwidth due to incorrectly prefetched lines. To study the effect of this prefetching solution on the bus bandwidth, the number of idle bus cycles was measured during the simulations.

Because each program has an inherently different number of idle bus cycles during execution, the data values were normalized by dividing the number of idle cycles by the total execution time. This gives a percentage of idle bus cycles during the run of the program.

The first thing to consider is the amount of bus bandwidth available for the given programs. Table 4.6 shows the percent of idle bus cycles present for the cache configurations in this experiment.

The graphs in Figures 4.50 through 4.59 show the increased bus usage caused by the prefetching engine. The vertical axis shows the difference in idle bus percentage between having prefetching on and off. In other words, a value of 15% means that the there is 15% more bus bandwidth consumed with prefetching turned on as compared to it turned off.

Table 4.6: Percent idle bus cycles

Cache configuration	straight	dhry21	hanoi	heapsort	gzip	cc1	xlisp
1K cache, 16 byte lines	20.4%	18.2%	34.3%	72.1%	23.2%	16.6%	12.4%
2K cache, 16 byte lines	20.5%	22.1%	35.4%	75.5%	24.9%	19.9%	15.4%
4K cache, 16 byte lines	20.6%	30.4%	35.4%	78.5%	36.1%	23.3%	20.7%
4K cache, 32 byte lines	18.2%	26.3%	35.4%	75.2%	25.8%	19.2%	17.0%
4K cache, 64 byte lines	16.4%	20.1%	35.3%	69.2%	16.3%	14.5%	11.8%
8K cache, 16 byte lines	20.6%	35.8%	35.4%	81.2%	39.5%	27.9%	26.3%
8K cache, 32 byte lines	18.2%	32.1%	35.4%	79.5%	29.1%	23.6%	23.0%
8K cache, 64 byte lines	16.4%	24.5%	35.3%	76.5%	19.0%	18.4%	18.0%
16K cache, 32 byte lines	18.2%	40.9%	35.4%	82.6%	35.7%	29.7%	25.0%
16K cache, 64 byte lines	16.4%	38.6%	35.4%	81.8%	24.1%	23.9%	20.2%

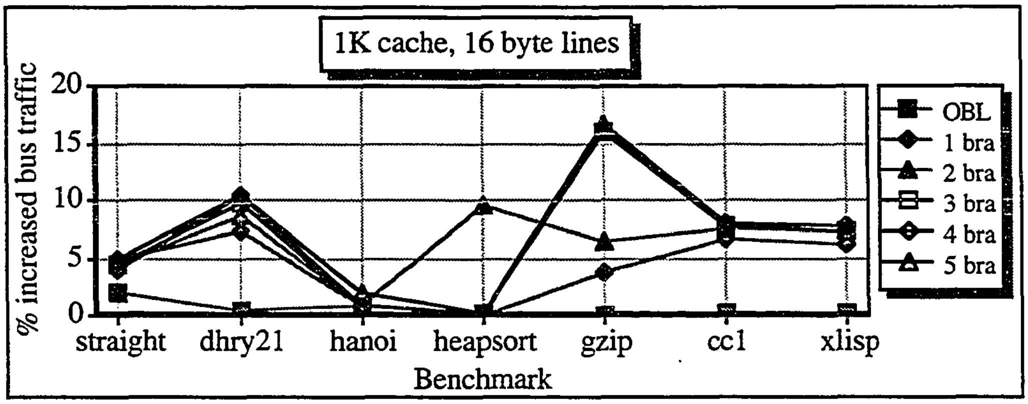

Figure 4.50: Increased bus traffic – 1K cache, 16 byte cache lines

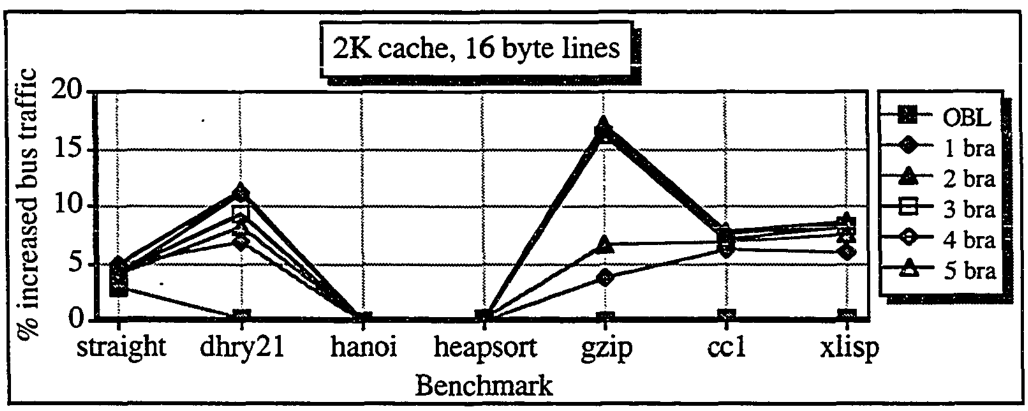

Figure 4.51: Increased bus traffic – 2K cache, 16 byte cache lines

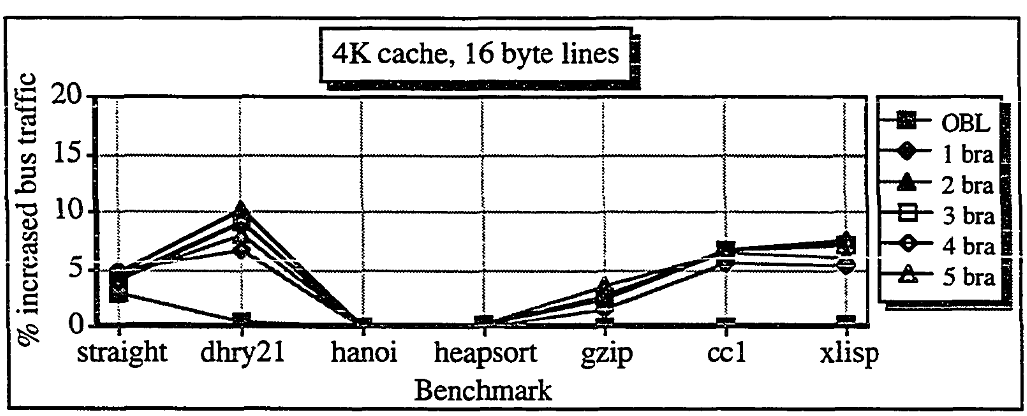

Figure 4.52: Increased bus traffic – 4K cache, 16 byte cache lines

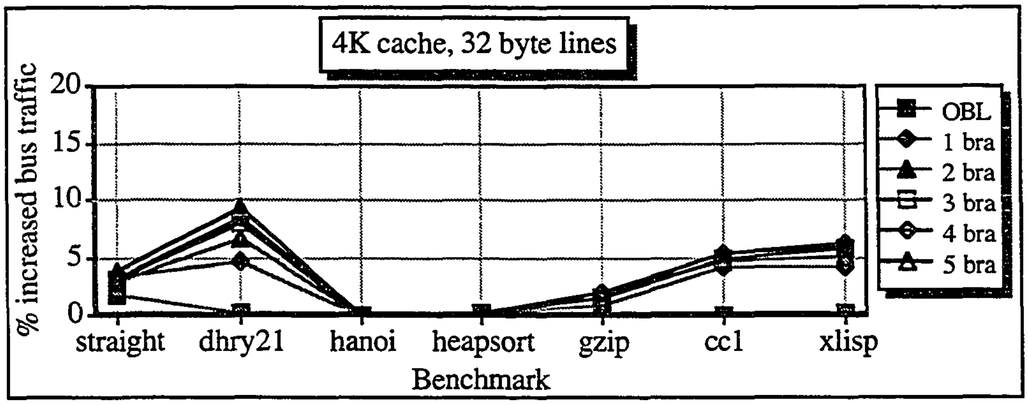

Figure 4.53: Increased bus traffic – 4K cache, 32 byte cache lines

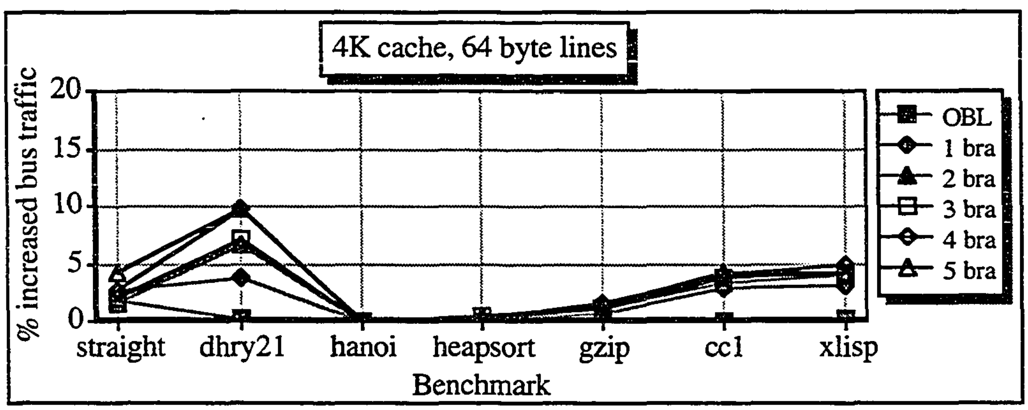

Figure 4.54: Increased bus traffic – 4K cache, 64 byte cache lines

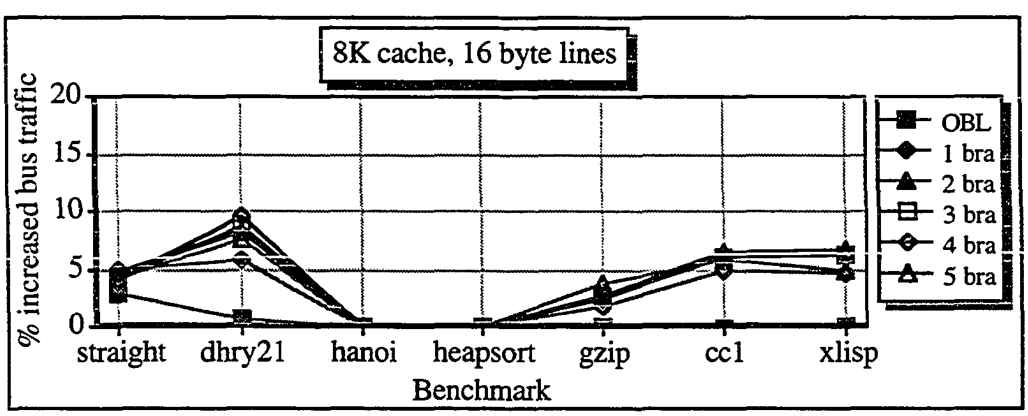

Figure 4.55: Increased bus traffic – 8K cache, 16 byte cache lines

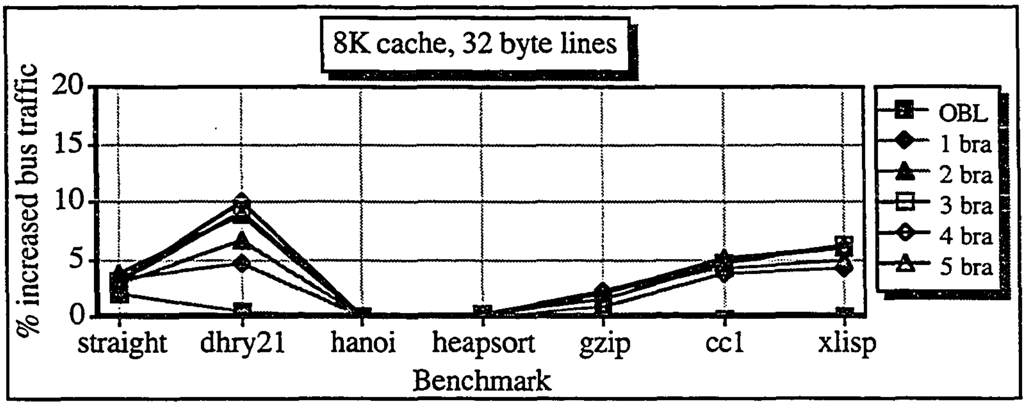

Figure 4.56: Increased bus traffic – 8K cache, 32 byte cache lines

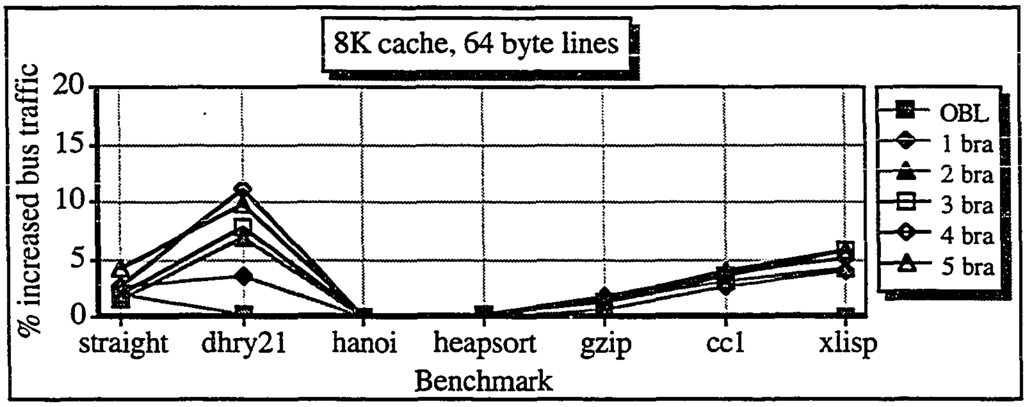

Figure 4.57: Increased bus traffic – 8K cache, 64 byte cache lines

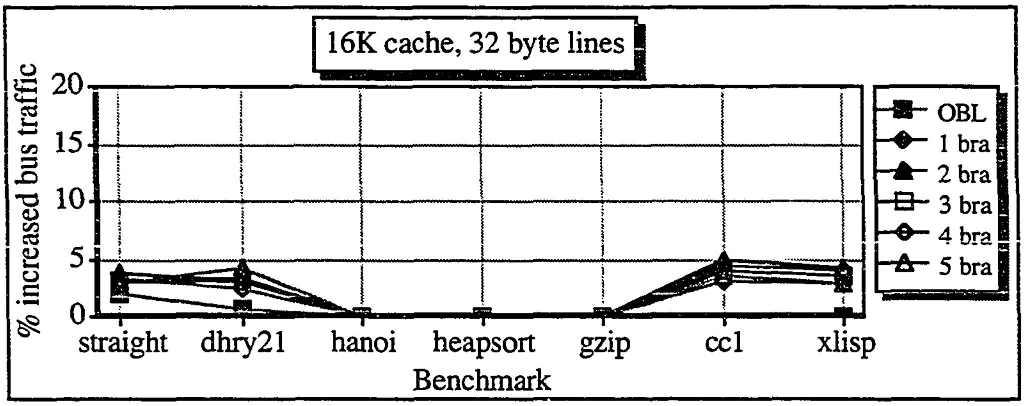

Figure 4.58: Increased bus traffic – 16K cache, 32 byte cache lines

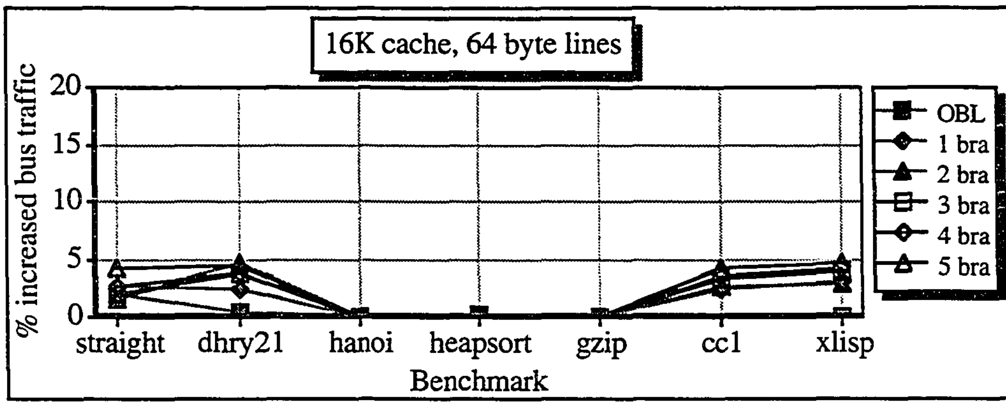

Figure 4.59: Increased bus traffic – 16K cache, 64 byte cache lines

As can be seen from the graphs, the prefetching activity does consume more bus bandwidth. In nearly every case, OBL consumes the least additional bandwidth, as would be expected because OBL is not as aggressive in its prefetching.

As the caches become larger, the percentage of extra bandwidth consumed decreases. This is due to the increased hit ratio present in larger caches, so less prefetching is necessary.

Table 4.7 averages the increased bus bandwidth consumption according to the maximum lead. From this table you can see that there is an increase in bus traffic, but not a significant amount. On any one particular program and set of cache characteristics, the bandwidth consumption can be increased by as much as 15% or more, but in general it is not as significant. It is important to note however that there is typically temporal clustering of the bus usage, so if these increases in bandwidth consumption come at times of peak usage, it could significantly hinder the CPU's performance.

4.6 Overall performance

Given the reduction of instruction stalls reported in Section 4.3, one would expect large overall performance improvements as well. This is not necessarily a direct result of reduced instructions stalls because the prefetching activity will slow the demand fetches as well as the data fetching activity. If this is the case, it is possible that performance would be hindered. The graphs of the overall performance for the same cache configurations described earlier are presented in Figures 4.60 through 4.70. Again, the vertical axis numbers were normalized to the times seen when no prefetching was used.

Table 4.7: Average increase in bus traffic

Cache configuration	OBL	1 bra	2 bra	3 bra	4 bra	5 bra
1K cache, 16 byte lines	0.51%	4.24%	6.41%	6.63%	6.76%	7.14%
2K cache, 16 byte lines	0.48%	4.00%	4.80%	6.50%	6.83%	7.12%
4K cache, 16 byte lines	0.51%	3.47%	4.06%	4.27%	4.25%	4.58%
4K cache, 32 byte lines	0.32%	2.47%	3.10%	3.40%	3.62%	3.84%
4K cache, 64 byte lines	0.34%	1.88%	2.42%	2.69%	3.31%	3.47%
8K cache, 16 byte lines	0.50%	3.17%	3.75%	4.01%	4.09%	4.13%
8K cache, 32 byte lines	0.31%	2.43%	3.03%	3.56%	3.79%	3.75%
8K cache, 64 byte lines	0.28%	1.95%	2.47%	2.97%	3.53%	3.68%
16K cache, 32 byte lines	0.30%	1.73%	1.96%	1.96%	2.12%	2.33%
16K cache, 64 byte lines	0.25%	1.52%	1.70%	1.89%	2.07%	2.52%
Overall Average	0.35%	2.53%	3.32%	3.76%	3.97%	4.24%

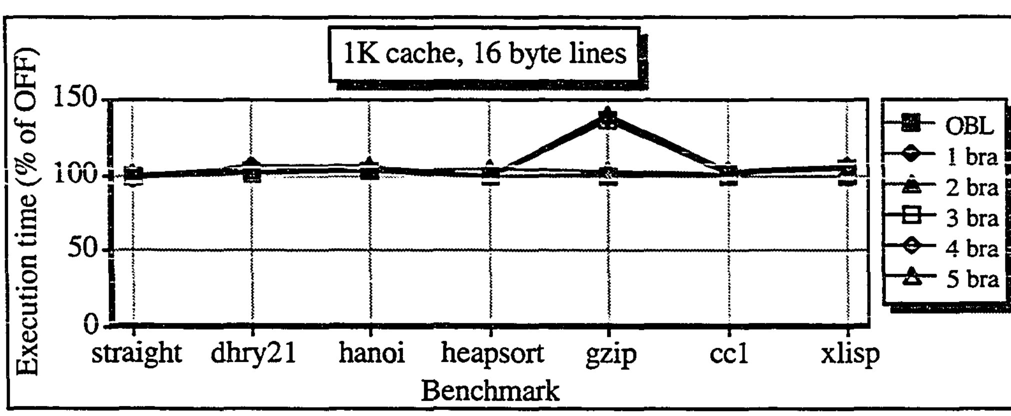

Figure 4.60: Execution time – 1K cache, 16 byte cache lines

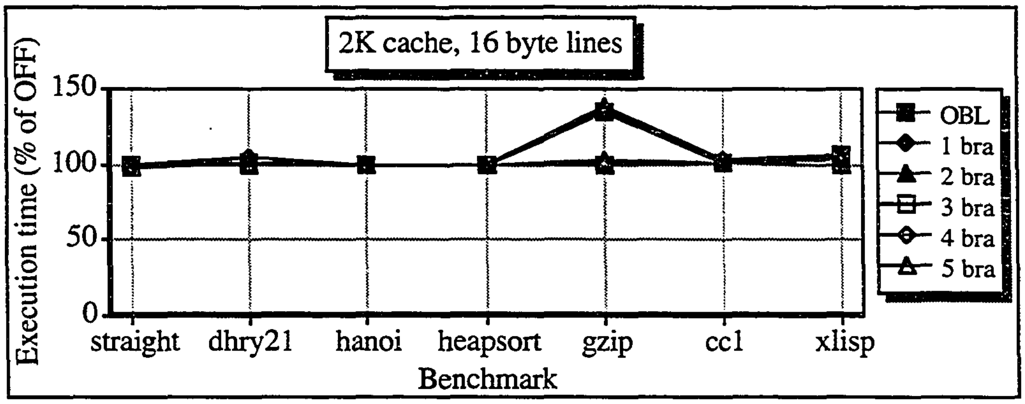

Figure 4.61: Execution time – 2K cache, 16 byte cache lines

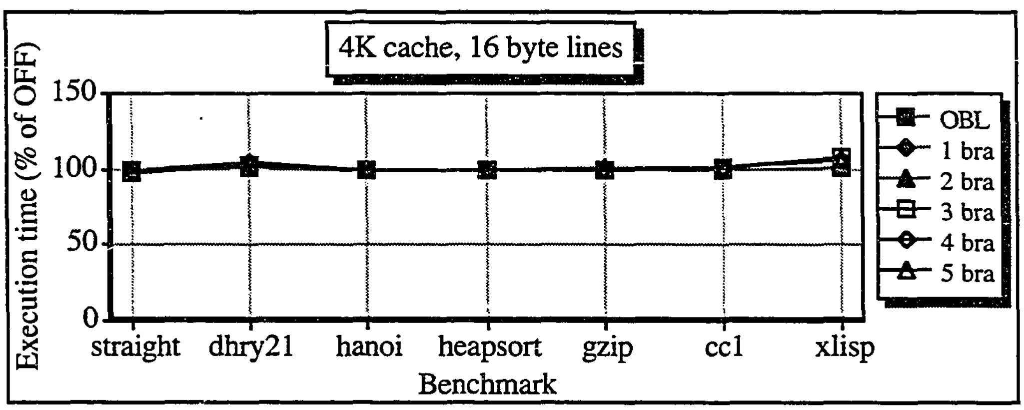

Figure 4.62: Execution time – 4K cache, 16 byte cache lines

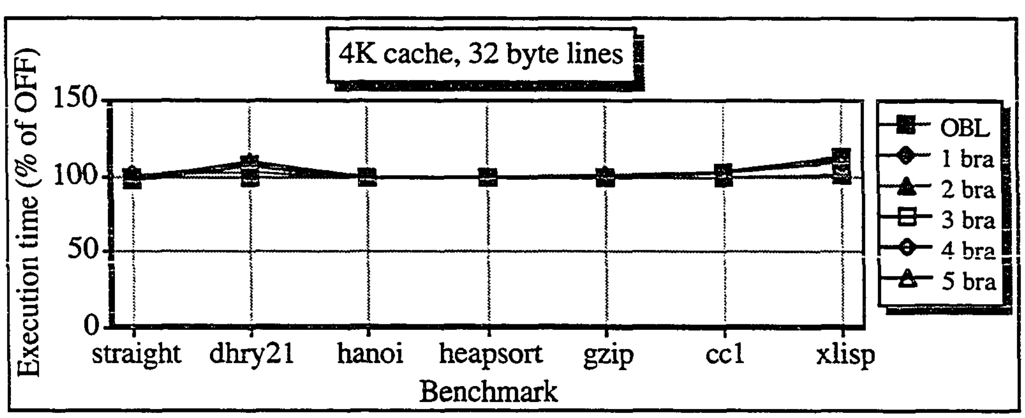

Figure 4.63: Execution time – 4K cache, 32 byte cache lines

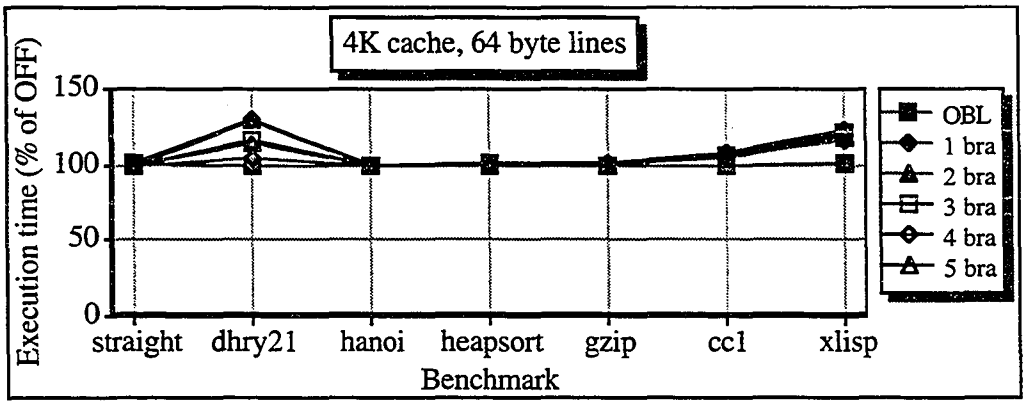

Figure 4.64: Execution time – 4K cache, 64 byte cache lines

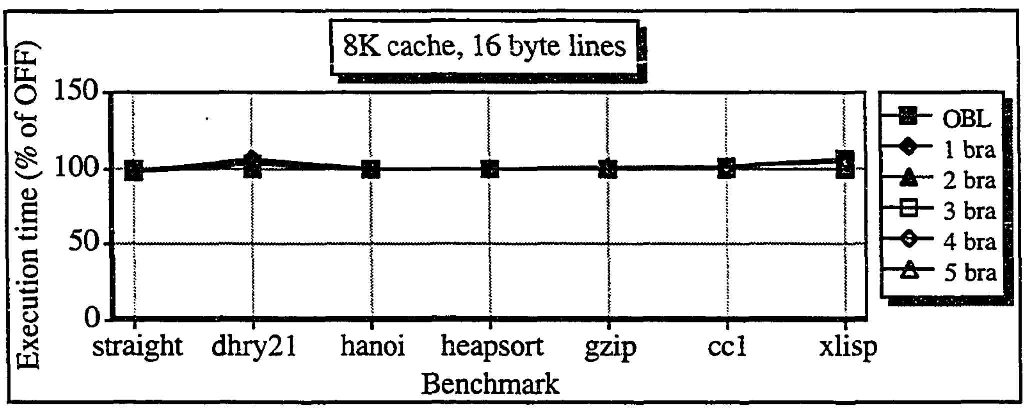

Figure 4.65: Execution time – 8K cache, 16 byte cache lines

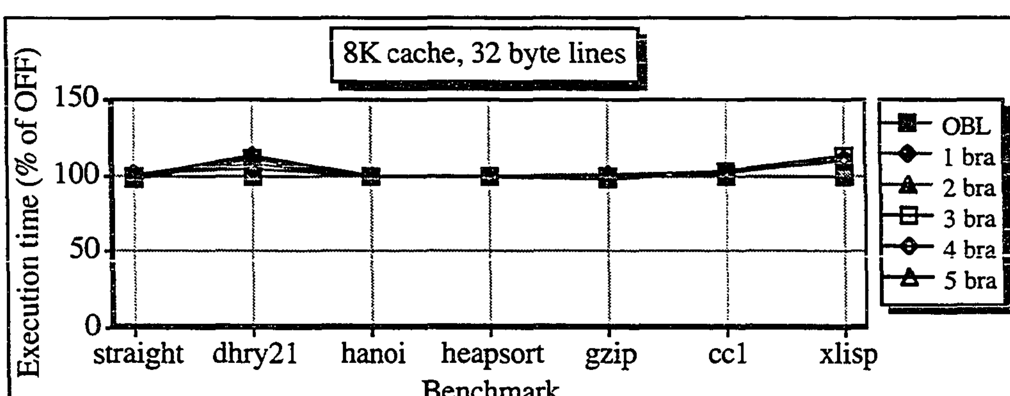

Figure 4.66: Execution time – 8K cache, 32 byte cache lines

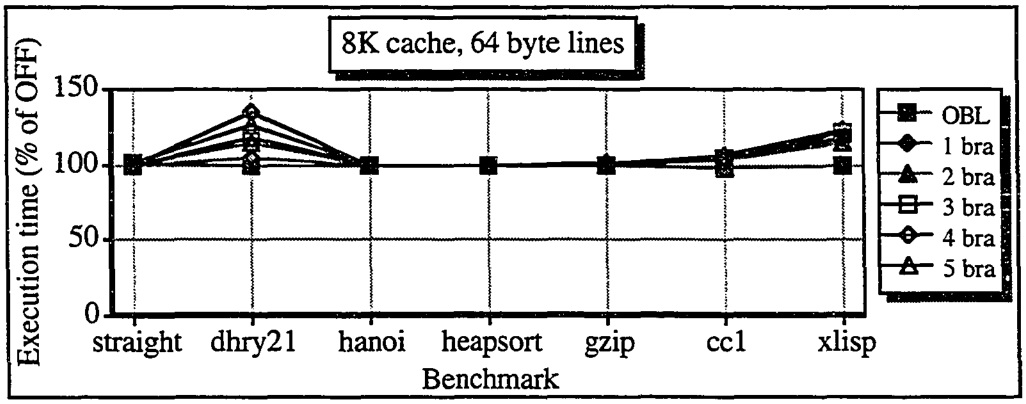

Figure 4.67: Execution time – 8K cache, 64 byte cache lines

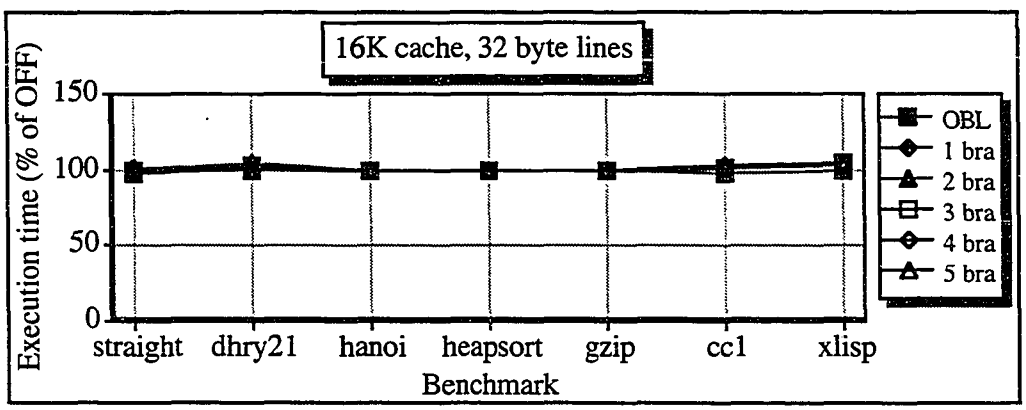

Figure 4.68: Execution time – 16K cache, 32 byte cache lines

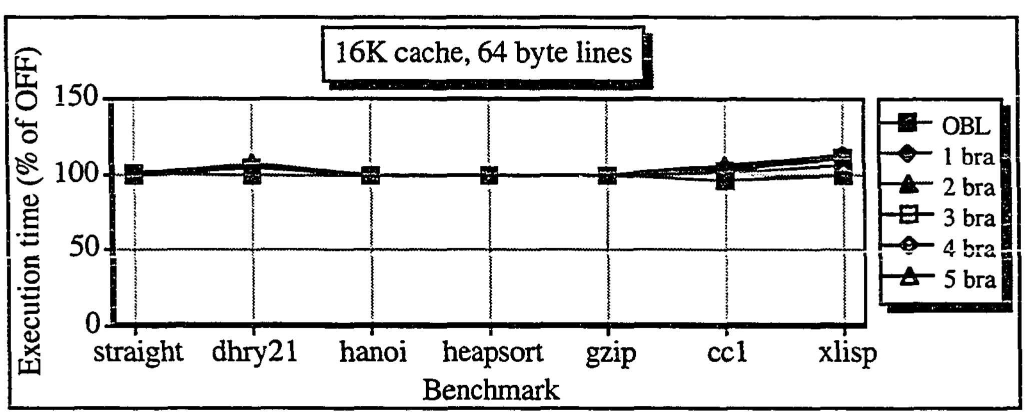

Figure 4.69: Execution time – 16K cache, 64 byte cache lines

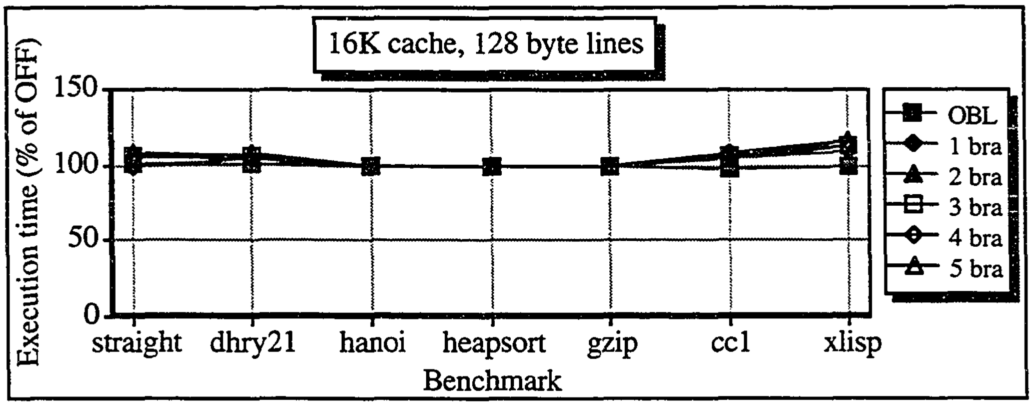

Figure 4.70: Execution time – 16K cache, 128 byte cache lines

As can be seen in the graphs, there is little performance improvement overall. In fact, in most cases the programs ran slightly slower with prefetching turned on. This is attributed to the limited bus bandwidth available and the additional competition for its use by the prefetching engine. Our simulations show that by the time a prefetch finishes there is usually a demand fetch queued up and waiting for the bus. The only case where this would slow down the processor is in the case of data reads and writes, because slow downs observed in the instruction stream are already factored into the instruction stall figures of Section 4.3. Therefore the bus bandwidth consumed by prefetching is causing extra stalls in data reads and writes. Further evidence of this is that the farther the prefetching engine is allowed to get ahead, the worse the overall performance becomes. This is caused by the extra prefetching activity that takes place with these larger leads.

To test this theory, the simulator was modified to collect the number of CPU stall cycles caused while waiting for data reads and writes. From the resulting data, it was clear that this was indeed the problem because there was a corresponding increase in stalls caused by data reads and writes for every decrease in stall cycles caused by instruction fetches.

To further test the theory, a program was developed consists of mathematical calculations on register based variables, so no bus bandwidth was consumed by data transfers. When this program was simulated, the overall performance improvements more closely matched the reductions in the number of instruction stall cycles.

As [Przybylski 90] observed, complicated prefetching strategies do not produce the performance improvements indicated by the accompanying reductions in miss ratios because of limited memory bandwidth and a strong temporal clustering of cache misses. This temporal clustering of memory bandwidth is evidenced here as well, because Table 4.6 shows that there is memory bandwidth available for extra prefetching, and Section 4.5 shows that the extra bandwidth consumed by this prefetching method is not significant. However if any prefetches occur during a time of peak demand, they will slow down the data throughput.

It is expected that with a faster memory bus, the reduction seen in the instruction stall cycles would be reflected in the overall performance. Since there is a throttling mechanism built into the prefetching engine, the amount of prefetching done with an infinitely faster bus would only increase slightly, so with a faster bus there would be a larger percentage of the bandwidth available for data throughput. Therefore the prefetching activity would not impede the data throughput as much. The bus system simulated was a 32-bit bus without interleaving and without split cycles. With today's trend toward 64-bit buses and interleaved memory, there is improvement already seen in the near future.

Close investigation of the graphs also shows that there is more separation in the performance of the various throttling limits when the cache lines become larger. This would imply that prefetching is definitely getting in the way and that perhaps the sub-line prefetching should be used instead. The overall performance of the other multipliers are slightly better but not significantly different from the ones shown here, so their graphs are not presented.

5. CONCLUSIONS

It has been shown that in today's high-speed computing systems, the performance bottleneck is the memory latency, and not the clock speed of the processor. It has also been shown that cache memory helps speed up the average memory access time, but when a cache miss occurs, the processor still stalls.

Many methods, ranging from software to hardware solutions, have been studied with varying amounts of success. Most techniques have concentrated on data prefetching. However, as I have shown, the CPU spends up to 50% (see Table 1.1) of its time stalled, waiting for instructions. The instruction memory latency reduction technique typically used in CPU designs today is the one block look-ahead (OBL) method.

I have presented a new hardware prefetching scheme based on dynamic interpretation of the instruction stream. This is done by adding a small pipeline to the cache which scans forward in the instruction stream interpreting each instruction and predicting the future execution path. It then prefetches what it predicts the CPU will be executing in the near future.

The accuracy of the prediction of the execution is directly dependent upon the accuracy of the branch predictions. If the prefetching engine predicts a conditional branch incorrectly, it will prefetch down the wrong path. This has two possible consequences: 1) the correct side of the branch may not be resident in the cache so the CPU may have a cache miss, and 2) the incorrectly prefetched cache lines will replace other lines in the cache. If these replaced lines will be needed again soon by the CPU, there will be a subsequent miss.

Again, the choice of branch prediction method is based on the CPU that the prefetch unit will be working with. In the simulations run for this research, the always branch method was the most accurate. This method is also simpler and requires less storage overhead and therefore less area in VLSI.

The idea of sub-line prefetching was presented and studied. It was thought that prefetching full cache lines might present too much overhead in terms of bus bandwidth, so prefetches should only fill partial cache lines instead. However it was determined that prefetching partial cache lines does not show any benefit when dealing with cache lines smaller than 128 bytes. When cache lines are 128 bytes or larger, it is a technique worth considering.

The pipelined prefetching engine has been shown to be a very effective technique for decreasing the instruction stall cycles in typical on-chip cache memories used today. It performs well, yielding reductions up to 30% or more for both scientific and general purpose programs, and has been shown to reduce the number of instruction stall cycles as compared to the OBL technique as well. However the overall performance increase shown by using this technique with the given architecture is limited. This is attributed to the limited bus bandwidth available and the additional competition for its use by the prefetching engine. The extra bus bandwidth consumed by prefetching causes extra stalls in data reads and writes.

As [Przybylski 90] observed, complicated prefetching strategies do not produce the performance improvements indicated by the accompanying reductions in miss ratios because of limited memory bandwidth and a strong temporal clustering of cache misses. This temporal clustering of memory bandwidth is evidenced here as well, as Table 4.6 shows that there is memory bandwidth available for extra prefetching, and Section 4.5 shows that the extra bandwidth consumed by this prefetching method is not significant. However if any prefetches occur during a time of peak demand, they will slow down the data throughput.

It is expected that with a faster memory bus, the reduction seen in the instruction stall cycles would be reflected in the overall performance. Because there is a throttling mechanism built into the prefetching engine, the amount of prefetching done with an infinitely faster bus would only increase slightly, so with a faster bus there would be a larger percentage of the bandwidth available for data throughput. Therefore the prefetching activity would not reduce the data throughput as much.

During the development of this method, close attention was paid to the feasibility of the design, and its actual implementation in hardware. Nothing was added to the simulator that could not be implemented in hardware. Each pipeline stage was carefully planned to make certain that it was doing no more work than is done in other typical CPU pipeline stages today. Therefore the speed of the pipelined prefetching engine should scale well with any VLSI technology used to implement a CPU.

The pipelined prefetching engine implementation adds on the order of 200 gates to the CPU chip, which would be on the order of 5000 transistors. Given that today's CPU's typically have millions of transistors on-chip, this would increase the size of the CPU far less than 1%. This is less area than is currently used by on-chip branch prediction logic in contemporary RISC processors. Also, note that the method described in this dissertation does not require a compiler to generate separate "branch likely" and "branch unlikely" instructions.

5.1 Future directions

The next area of study for this project would be the implementation of data prefetching. It seems an obvious extension to the pipelined prefetching engine to have it load the operands for instructions in addition to the instructions themselves, but it may not be possible to determine the address of all operands at run-time.

It should also be considered how this instruction prefetching method combines with data prefetching. Since the bus bandwidth is such a limited commodity, there may not be any gains possible from the addition of data prefetching. However, since prefetching has the effect of smoothing out the temporal clusters, an implementation of both data prefetching and instruction prefetching may help smooth out these rifts to a point where they are not as much of a problem.

Consideration should also be given to whether it is possible to detect or predict the temporal clusters of bus activity. If this were possible, prefetching should be halted during those times, which may result in increased performance.

BIBLIOGRAPHY

[Agarwal et al. 90] Agarwal, A., Beng-Hong, L., Kranz, D., and Kubiatowicz, J., *APRIL: A Processor Architecture for Multiprocessing*, Proceedings of the 17th International Symposium on Computer Architecture, p. 104–114.

[Asprey et al. 93] Asprey, T., Averill, G., DeLano, E., Mason, R., Weiner, B., and Yetter, J., *Performance Features of the PA7100 Microprocessor, IEEE Micro*, p. 22–35, June 1993.

[Baer & Chen 91] Baer, J.-L., and Chen, T.-F., *An Effective On-Chip Preloading Scheme To Reduce Data Access Penalty*, Proceedings of SuperComputing 1991.

[Bray & Flynn 91] Bray, B. K., and Flynn, M. J., *Write caches as an alternative to write buffers*, Technical Report CSL–TR–91–470, Stanford University.

[Chen & Baer 92] Chen, T.-F. and Baer, J.-L., *Reducing Memory Latency via Non-blocking and Prefetching Caches*, International Conference on Architectural Support for Programming Languages and Operating Systems, 27(7–9):51–61.

[DEC 92] Digital Equipment Corporation, *Alpha Architecture Handbook*, Digital Press, Maybard, MA, 1992.

[Fu & Patel 92] Fu, J. W. C., Patel, H. H., and Janssens, B. L., *Stride Directed Prefetching in Scalar Processors*, The 25th Annual International Symposium on Microarchitecture, p. 102–110, 1992.

[Gindele 77] Gindele, J. D., *Buffer block prefetching method*, IBM Technical Disclosure Bulletin, 20(2):696–697, July 1977.

[Gupta & Hennessy 91] Gupta, A. and Hennessy J., *Comparative Evaluation of Latency Reducing and Tolerating Techniques*, Proceedings of the 18th International Symposium on Computer Architecture, p. 254–263.

[Hennessy & Patterson 90] Hennessy, J. L. and Patterson, D. A., *Computer Architecture A Quantitative Approach*, Morgan Kaufmann Publishers, Inc. San Mateo, CA, 94403, 1990.

[Hennessy 91] Hennessy, J. L. and Jouppi, N. P., *Computer Technology and Architecture: An Evolving Interaction, IEEE Computer*, 24(9):18–29, September 1991.

[Jouppi 90] Jouppi, N. P., *Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers*, Proceedings of the 17th International Symposium on Computer Architecture, p. 364–373.

[Kane & Heinrich 92] Kane, G., and Heinrich, J., *MIPS RISC Architecture*, Prentice Hall, Englewood Cliffs, NJ 07632, 1992.

[Klaiber & Levy 91] Klaiber, A. C., and Levy, H. M., *An Architecture for Software-Controlled Data Prefetching*, Proceedings of the 18th International Symposium on Computer Architecture, p. 43–53.

[Kroft 81] Kroft, D., *Lockup-Free Instruction Fetch/Prefetch Cache Organization*, Proceedings of the 8th Symposium on Computer Architecture, p. 81–87.

[Larus 90] Larus, J. R., *SPIM S20: A MIPS R2000 simulator*, Technical Report 966, University of Wisconsin–Madison, September 1990.

[Motorola 93] Motorola, *PowerPC 601 – RISC Microprocessor User's Manual*, Schaumburg, IL, 1993.

[Mowry & Gupta 91] Mowry, T. C., Gupta, A., *Tolerating Latency Through Software-Controlled Prefetching in Shared-Memory Multiprocessors*, Journal of Parallel and Distributed Computing, v. 12, p. 87–106 (1991).

[Mowry et al. 92] Mowry, T. C., Lam, M. S., and Gupta, A., *Design and Evaluation of a Compiler Algorithm for Prefetching*, International Conference on Architectural Support for Programming Languages and Operating Systems, 27(7–9): 62–73.

[Öner & Dubois 1992] Öner, K., Dubois, M., *Improving the Performance of Data Caches in Systems with Large Miss Latencies*, Technical Report No. CENG 92–14, University of Southern California, July 1992.

[Przybylski 90] Przybylski, S., *The Performance Impact of Block Sizes and Fetch Strategies*, Proceedings of the 17th International Symposium on Computer Architecture, p. 160–169.

[Rogers & Li 92] Rogers, A. and Li, K., *Software Support for Speculative Loads*, International Conference on Architectural Support for Programming Languages and Operating Systems, 27(7–9): 38–50.

[Smith 78a] Smith, B. J., *A Pipelined, Shared Resource MIMD Computer*, Proceedings of the 1978 International Conference on Parallel Processing, 1978.

[Smith 78b] Smith, J. A., *Sequential Program Prefetching in Memory Hierarchies*, *IEEE Computer*, 11(12):7–21, December 1978.

[Smith 82] Smith, J. A., *Cache Memories*, ACM Computing Surveys, 14(3):473–530, September 1982.

[Weber & Gupta 89] Weber, W.-D., and Gupta, A., *Exploring the Benefits of Multiple Hardware Contexts in a Multiprosessor Architecture: Preliminary Results*, Proceedings of the 16th Symposium on Computer Architecture, p. 273–280.

[Yeh & Patt 93] Yeh, T.-Y., and Patt, Y. N., *A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History,* Proceedings of the 20th Symposium on Computer Architecture, p. 257–266.

[Young 94] Young, J. L., *Insider's Guide to PowerPC Computing*, Que Publishing, Carmel, IN, 1994.