Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers

Norman P. Jouppi

Digital Equipment Corporation Western Research Lab

100 Hamilton Ave., Palo Alto, CA 94301

Abstract

Projections of computer technology forecast processors with peak performance of 1,000 MIPS in the relatively near future. These processors could easily lose half or more of their performance in the memory hierarchy if the hierarchy design is based on conventional caching techniques. This paper presents hardware techniques to improve the performance of caches.

Miss caching places a small fully-associative cache between a cache and its refill path. Misses in the cache that hit in the miss cache have only a one cycle miss penalty, as opposed to a many cycle miss penalty without the miss cache. Small miss caches of 2 to 5 entries are shown to be very effective in removing mapping conflict misses in first-level direct-mapped caches.

Victim caching is an improvement to miss caching that loads the small fully-associative cache with the victim of a miss and not the requested line. Small victim caches of 1 to 5 entries are even more effective at removing conflict misses than miss caching.

Stream buffers prefetch cache lines starting at a cache miss address. The prefetched data is placed in the buffer and not in the cache. Stream buffers are useful in removing capacity and compulsory cache misses, as well as some instruction cache conflict misses. Stream buffers are more effective than previously investigated prefetch techniques at using the next slower level in the memory hierarchy when it is pipelined. An extension to the basic stream buffer, called *multi-way stream buffers*, is introduced. Multi-way stream buffers are useful for prefetching along multiple intertwined data reference streams.

Together, victim caches and stream buffers reduce the miss rate of the first level in the cache hierarchy by a factor of two to three on a set of six large benchmarks.

1. Introduction

Cache performance is becoming increasingly important since it has a dramatic effect on the performance of advanced processors. Table 1-1 lists some cache miss times and the effect of a miss on machine performance. Over the last decade, cycle time has been decreasing much faster than main memory access time. The average number of machine cycles per instruction has also been decreasing dramatically, especially when the transition from CISC machines to RISC machines is included. These two effects are multiplicative and result in tremen-dous increases in miss cost. For example, a cache miss on a VAX 11/780 only costs 60% of the average instruction execution. Thus even if every instruction had a cache miss, the machine performance would slow down by only 60%! However, if a RISC machine like the WRL Titan [10] has a miss, the cost is almost ten instruction times. Moreover, these trends seem to be continuing, especially the increasing ratio of memory access time to machine cycle time. In the future a cache miss all the way to main memory on a superscalar machine executing two instructions per cycle could cost well over 100 instruction times! Even with careful application of well-known cache design techniques, machines with main memory latencies of over 100 instruction times can easily lose over half of their potential performance to the memory hierarchy. This makes both hardware and software research on advanced memory hierarchies increasingly important.

Machine	cycles per instr	cycle time (ns)	mem time (ns)	miss cost (cycles)	miss cost (instr)
VAX11/780	10.0	200	1200	6	.6
WRL Titan	1.4	45	540	12	8.6
?	0.5	4	280	70	140.0

Table 1-1: The increasing cost of cache misses

This paper investigates new hardware techniques for increasing the performance of the memory hierarchy. Section 2 describes a baseline design using conventional caching techniques. The large performance loss due to the memory hierarchy is a detailed motivation for the techniques discussed in the remainder of the paper. Techniques for reducing misses due to mapping conflicts (i.e., lack of associativity) are presented in Section 3. An extension to prefetch techniques called stream buffering is evaluated in Section 4. Section 5 summarizes this work and evaluates promising directions for future work.

2. Baseline Design

Figure 2-1 shows the range of configurations of interest in this study. The CPU, floating-point unit, memory management unit (e.g., TLB), and first level instruction and data caches are on the same chip or on a single high-speed module built with an advanced packaging technology. (We will refer to the central processor as a single chip in the remainder of the paper, but chip or

module is implied.) The cycle time off this chip is 3 to 8 times longer than the instruction issue rate (i.e., 3 to 8 instructions can issue in one off-chip clock cycle). This is obtained either by having a very fast on-chip clock (e.g., superpipelining [8]), by issuing many instructions per cycle (e.g., superscalar or VLIW), and/or by using higher speed technologies for the processor chip than for the rest of the system (e.g., GaAs vs. BiCMOS).

The expected size of the on-chip caches varies with the implementation technology for the processor, but higher-speed technologies generally result in smaller on-chip caches. For example, quite large on-chip caches should be feasible in CMOS but only small caches are feasible in the near term for GaAs or bipolar processors. Thus, although GaAs and bipolar are faster, the higher miss rate from their smaller caches tends to decrease the actual system performance ratio between GaAs or bipolar machines and dense CMOS machines to less than the ratio between their gate speeds. In all cases the first-level caches are assumed to be direct-mapped, since this results in the fastest effective access time [7]. Line sizes in the on-chip caches are most likely in the range of 16B to 32B. The data cache may be either write-through or write-back, but this paper does not examine those tradeoffs.
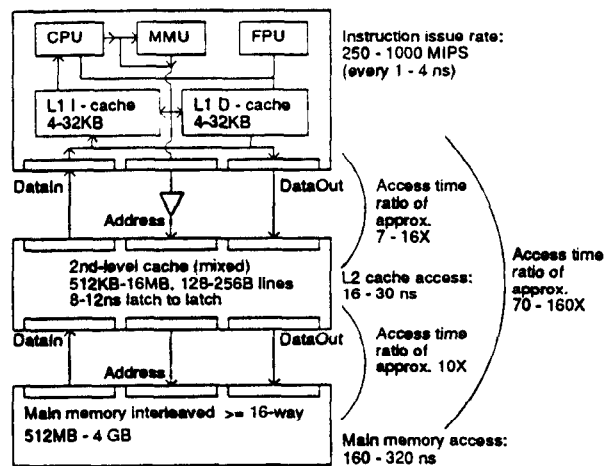

Figure 2-1: Baseline design

The second-level cache is assumed to range from 512KB to 16MB, and to be built from very high speed static RAMs. It is assumed to be direct-mapped for the same reasons as the first-level caches. For caches of this size access times of 16 to 30ns are likely. This yields an access time for the cache of 4 to 30 instruction times. The relative speed of the processor as compared to the access time of the cache implies that the second-level cache must be pipelined in order for it to provide sufficient bandwidth. For example, consider the case where the first-level cache is a write-through cache. Since stores typically occur at an average rate of 1 in every 6 or 7 instructions, an unpipelined external cache would not have even enough bandwidth to handle the store traffic for access times greater than seven instruction times. Caches have been pipelined in mainframes for a number of years [12], but this is a recent development for workstations. Recently cache chips with ECL I/O's and registers or latches on their inputs and outputs have appeared; these are ideal for pipelined caches. The number

of pipeline stages in a second-level cache access could be 2 or 3 depending on whether the pipestage going from the processor chip to the cache chips and the pipestage returning from the cache chips to the processor are full or half pipestages.

In order to provide sufficient memory for a processor of this speed (e.g., several megabytes per MIP), main memory should be in the range of 512MB to 4GB. This means that even if 16Mb DRAMs are used that it will contain roughly a thousand DRAMs. The main memory system probably will take about ten times longer for an access than the second-level cache. This access time is easily dominated by the time required to fan out address and data signals among a thousand DRAMs spread over many cards. Thus even with the advent of faster DRAMs, the access time for main memory may stay roughly the same. The relatively large access time for main memory in turn requires that second-level cache line sizes of 128 or 256B are needed. As a counter example, consider the case where only 16B are returned after 320ns. This is a bus bandwidth of 50MB/sec. Since a 10 MIP processor with this bus bandwidth would be bus-bandwidth limited in copying from one memory location to another [11], little extra performance would be obtained by the use of a 100 to 1,000 MIP processor. This is an important consideration in the system performance of a processor.

Several observations are in order on the baseline system. First, the memory hierarchy of the system is actually quite similar to that of a machine like the VAX 11/780 [3, 4], only each level in the hierarchy has moved one step closer to the CPU. For example, the 8KB board-level cache in the 780 has moved on-chip. The 512KB to 16MB main memory on early VAX models has become the board-level cache. Just as in the 780's main memory, the incoming transfer size is large (128-256B here vs. 512B pages in the VAX). The main memory in this system is of similar size to the disk subsystems of the early 780's and performs similar functions such as paging and file system caching.

The actual parameters assumed for our baseline system are 1,000 MIPS peak instruction issue rate, separate 4KB first-level instruction and data caches with 16B lines, and a 1MB second-level cache with 128B lines. The miss penalties are assumed to be 24 instruction times for the first level and 320 instruction times for the second level. The characteristics of the test programs used in this study are given in Table 2-1. These benchmarks are reasonably long in comparison with most traces in use today, however the effects of multiprocessing have not been modeled in this work. The first-level cache miss rates of these programs running on the baseline system configuration are given in Table 2-2.

program name	dynamic instr.	data refs.	total refs.	program type
ccom	31.5M	14.0M	45.5M	C compiler
grr	134.2M	59.2M	193.4M	PC board CAD
yacc	51.0M	16.7M	67.7M	Unix utility
met	99.4M	50.3M	149.7M	PC board CAD
linpack	144.8M	40.7M	185.5M	100x100 numeric
liver	23.6M	7.4M	31.0M	LFK (numeric)
total	484.5M	188.3M	672.8M	

Table 2-1: Test program characteristics

The effects of these miss rates are given graphically in Figure 2-2. The region below the solid line gives the net performance of the system, while the region above the solid line gives the performance lost in the memory hierarchy. For example, the difference between the top dotted line and the bottom dotted line gives the performance lost due to first-level data cache misses. As can be seen in Figure 2-2, most benchmarks lose over half of their potential performance in first level cache misses. Only relatively small amounts of performance are lost to second-level cache misses. This is primarily due to the large second-level cache size in comparison to the size of the programs executed. Longer traces [2] of larger programs exhibit significant numbers of second-level cache misses. Since the test suite used in this paper is too small for significant second-level cache activity, second-level cache misses will not be investigated in detail, but will be left to future work.

program name	baseline miss rate instr.		data
ccom	0.096		0.120
grr	0.061		0.062
yacc	0.028		0.040
met	0.017		0.039
linpack	0.000		0.144
liver	0.000		0.273

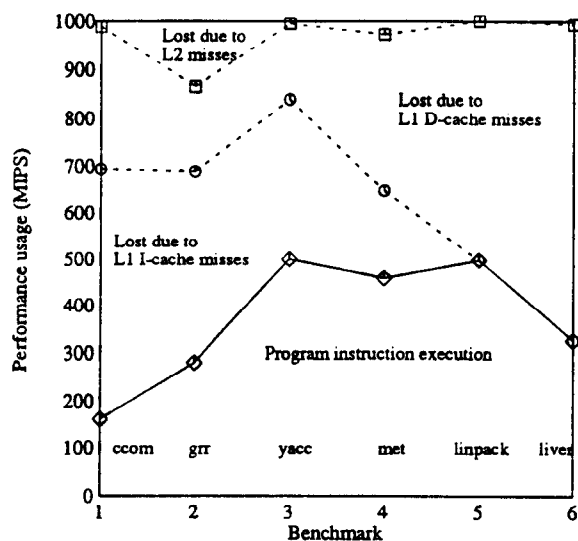**Table 2-2:** Baseline system first-level cache miss rates

Figure 2-2: Baseline design performance

Since the exact parameters assumed are at the extreme end of the ranges described (maximum performance processor with minimum size caches), other configurations would lose proportionally less performance in their memory hierarchy. Nevertheless, any configuration in the range of interest will lose a substantial proportion of its potential performance in the memory hierarchy. This means that the greatest leverage on system performance will be obtained by improving the memory hierarchy performance, and not by attempting to further increase the performance of the CPU (e.g., by more aggressive parallel issuing of instructions). Techniques for improving the performance of the baseline memory

hierarchy at low cost are the subject of the remainder of this paper. Finally, in order to avoid compromising the performance of the CPU core (comprising of the CPU, FPU, MMU, and first level caches), any additional hardware required by the techniques to be investigated should reside outside the CPU core (i.e., below the first level caches). By doing this the additional hardware will only be involved during cache misses, and therefore will not be in the critical path for normal instruction execution.

3. Reducing Conflict Misses: Miss Caching and Victim Caching

Misses in caches can be classified into four categories: conflict, compulsory, capacity [7], and coherence. Conflict misses are misses that would not occur if the cache was fully-associative and had LRU replacement. Compulsory misses are misses required in any cache organization because they are the first references to an instruction or piece of data. Capacity misses occur when the cache size is not sufficient to hold data between references. Coherence misses are misses that occur as a result of invalidation to preserve multiprocessor cache consistency.

Even though direct-mapped caches have more conflict misses due to their lack of associativity, their performance is still better than set-associative caches when the access time costs for hits are considered. In fact, the direct-mapped cache is the only cache configuration where the critical path is merely the time required to access a RAM [9]. Conflict misses typically account for between 20% and 40% of all direct-mapped cache misses [7]. Figure 3-1 details the percentage of misses due to conflicts for our test suite. On average 39% of the first-level data cache misses are due to conflicts, and 29% of the first-level instruction cache misses are due to conflicts. Since these are significant percentages, it would be nice to "have our cake and eat it too" by somehow providing additional associativity without adding to the critical access path for a direct-mapped cache.
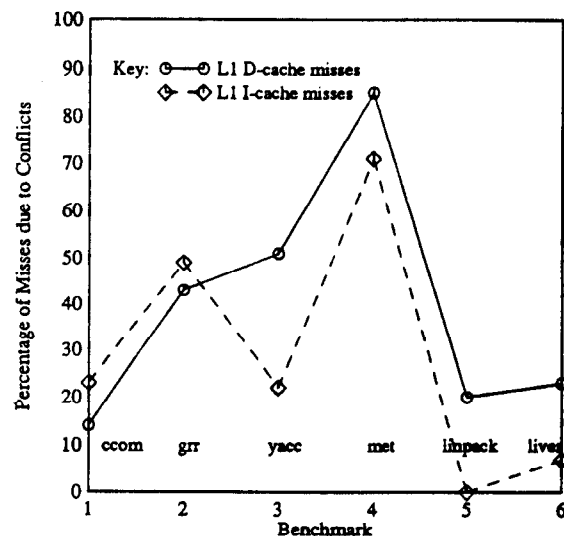

Figure 3-1: Conflict misses, 4KB I and D, 16B lines

390

3.1. Miss Caching

We can add associativity to a direct-mapped cache by placing a small *miss cache* on-chip between a first-level cache and the access port to the second-level cache (Figure 3-2). A miss cache is a small fully-associative cache containing on the order of two to five cache lines of data. When a miss occurs, data is returned not only to the direct-mapped cache, but also to the miss cache under it, where it replaces the least recently used item. Each time the upper cache is probed, the miss cache is probed as well. If a miss occurs in the upper cache but the address hits in the miss cache, then the direct-mapped cache can be reloaded in the next cycle from the miss cache. This replaces a long off-chip miss penalty with a short one-cycle on-chip miss. This arrangement satisfies the requirement that the critical path is not worsened, since the miss cache itself is not in the normal critical path of processor execution.
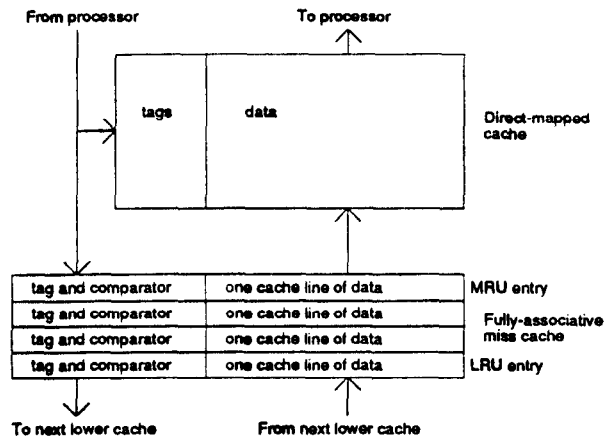

Figure 3-2: Miss cache organization

The success of different miss cache organizations at removing conflict misses is shown in Figure 3-3. The first observation to be made is that many more data conflict misses are removed by the miss cache than instruction conflict misses. This can be explained as follows. Instruction conflicts tend to be widely spaced because the instructions within one procedure will not conflict with each other as long as the procedure size is less than the cache size, which is almost always the case. Instruction conflict misses are most likely when another procedure is called. The target procedure may map anywhere with respect to the calling procedure, possibly resulting in a large overlap. Assuming at least 60 different instructions are executed in each procedure, the conflict misses would span more than the 15 lines in the maximum size miss cache tested. In other words, a small miss cache could not contain the entire overlap and so would be reloaded repeatedly before it could be used. This type of reference pattern exhibits the worst miss cache performance.

Data conflicts, on the other hand, can be quite closely spaced. Consider the case where two character strings are being compared. If the points of comparison of the two strings happen to map to the same line, alternating references to different strings will always miss in the cache. In this case a miss cache of only two entries

would remove all of the conflict misses. Obviously this is another extreme of performance and the results in Figure 3-3 show a range of performance based on the program involved. Nevertheless, for 4KB data caches a miss cache of only 2 entries can remove 25% percent of the data cache conflict misses on average,[1] or 13% of the data cache misses overall. If the miss cache is increased to 4 entries, 36% percent of the conflict misses can be removed, or 18% of the data cache misses overall. After four entries the improvement from additional miss cache entries is minor, only increasing to a 25% overall reduction in data cache misses if 15 entries are provided.
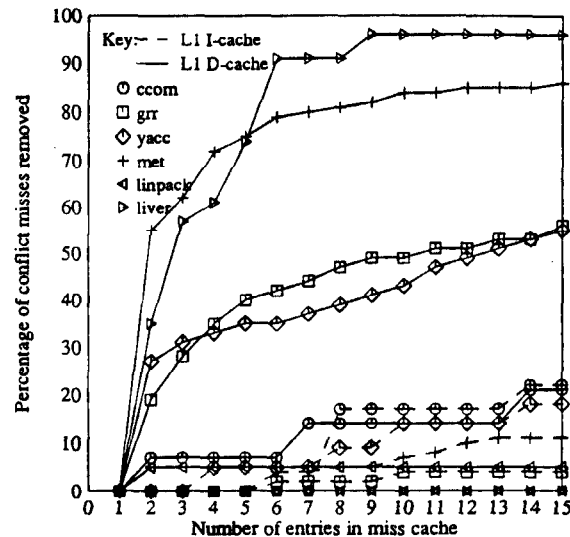

Figure 3-3: Conflict misses removed by miss caching

Since doubling the data cache size results in a 32% reduction in misses (over this set of benchmarks when increasing data cache size from 4K to 8K), each additional line in the first level cache reduces the number of misses by approximately 0.13%. Although the miss cache requires more area per bit of storage than lines in the data cache, each line in a two line miss cache effects a 50 times larger marginal improvement in the miss rate, so this should more than cover any differences in layout size.

Comparing Figure 3-3 and Figure 3-1, we see that the higher the percentage of misses due to conflicts, the more effective the miss cache is at eliminating them. For example, in Figure 3-1 *met* has by far the highest ratio of conflict misses to total data cache misses. Similarly, *grr* and *yacc* also have greater than average percentages of conflict misses, and the miss cache helps these programs significantly as well. *linpack* and *ccom* have the lowest

---

[1]Throughout this paper the average reduction in miss rates is used as a metric. This is computed by calculating the percent reduction in miss rate for each benchmark, and then taking the average of these percentages. This has the advantage that it is independent of the number of memory references made by each program. Furthermore, if two programs have widely different miss rates, the average percent reduction in miss rate gives equal weighting to each benchmark. This is in contrast with the percent reduction in average miss rate, which weights the program with the highest miss rate most heavily.

percentage of conflict misses, and the miss cache removes the lowest percentage of conflict misses from these programs. This results from the fact that if a program has a large percentage of data conflict misses then they must be clustered to some extent because of their overall density. This does not prevent programs with a small number of conflict misses such as *liver* from benefiting from a miss cache, but it seems that as the percentage of conflict misses increases, the percentage of these misses removable by a miss cache increases.

3.2. Victim Caching

Consider a system with a direct-mapped cache and a miss cache. When a miss occurs, data is loaded into both the miss cache and the direct-mapped cache. In a sense, this duplication of data wastes storage space in the miss cache. The number of duplicate items in the miss cache can range from one (in the case where all items in the miss cache map to the same line in the direct-mapped cache) to all of the entries (in the case where a series of misses occur which do not hit in the miss cache).

To make better use of the miss cache we can use a different replacement algorithm for the small fully-associative cache [5]. Instead of loading the requested data into the miss cache on a miss, we can load the fully-associative cache with the victim line from the direct-mapped cache instead. We call this *victim caching* (see Figure 3-4). With victim caching, no data line appears both in the direct-mapped cache and the victim cache. This follows from the fact that the victim cache is loaded only with items thrown out from the direct-mapped cache. In the case of a miss in the direct-mapped cache that hits in the victim cache, the contents of the direct-mapped cache line and the matching victim cache line are swapped.
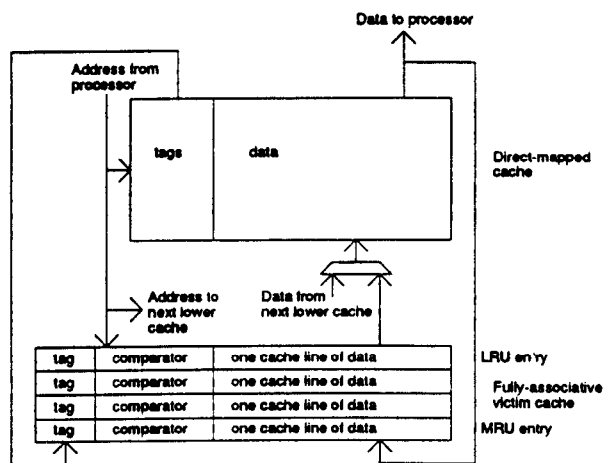

Figure 3-4: Victim cache organization

Depending on the reference stream, victim caching can either be a small or significant improvement over miss caching. The magnitude of this benefit depends on the amount of duplication in the miss cache. Victim caching is always an improvement over miss caching.

As an example, consider an instruction reference stream that calls a small procedure in its inner loop that conflicts with the loop body. If the total number of con-

flicting lines between the procedure and loop body were larger than the miss cache, the miss cache would be of no value since misses at the beginning of the loop would be flushed out by later misses before execution returned to the beginning of the loop. If a victim cache is used instead, however, the number of conflicts in the loop that can be captured is doubled compared to that stored by a miss cache. This is because one set of conflicting instructions lives in the direct-mapped cache, while the other lives in the victim cache. As execution proceeds around the loop and through the procedure call these items trade places.

The percentage of conflict misses removed by victim caching is given in Figure 3-5. Note that victim caches consisting of just one line are useful, in contrast to miss caches which must have two lines to be useful. All of the benchmarks have improved performance in comparison to miss caches, but instruction cache performance and the data cache performance of benchmarks that have conflicting long sequential reference streams (e.g., *ccom* and *linpack*) improve the most.
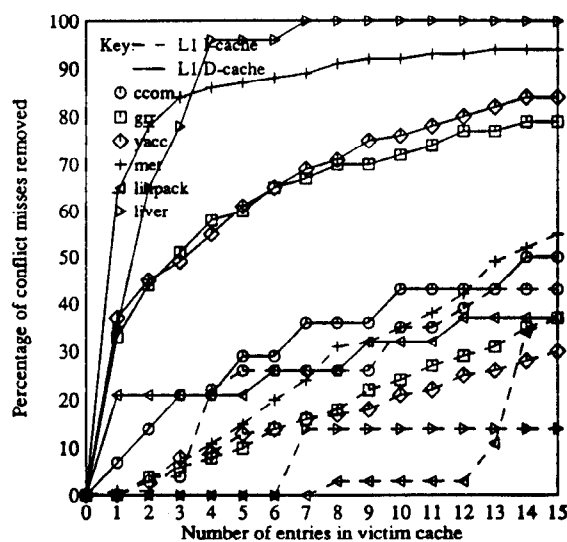

Figure 3-5: Conflict misses removed by victim caching

3.3. The Effect of Direct-Mapped Cache Size on Victim Cache Performance

Figure 3-6 shows the performance of 1, 2, 4, and 15 entry victim caches when backing up direct-mapped data caches of varying sizes. In general smaller direct-mapped caches benefit the most from the addition of a victim cache. Also shown for reference is the total percentage of conflict misses for each cache size. There are two factors to victim cache performance versus direct-mapped cache size. First, as the direct-mapped cache increases in size, the relative size of the victim cache becomes smaller. Since the direct-mapped cache gets larger but keeps the same line size (16B), the likelihood of a tight mapping conflict which would be easily removed by victim caching is reduced. Second, the percentage of conflict misses decreases slightly from 1KB to 32KB. As we have seen previously, as the percentage of conflict misses decreases, the percentage of these misses removed by the victim cache decreases. The first effect dominates, however, since as the percentage of

conflict misses increases with very large caches (as in [7]), the victim cache performance only improves slightly.
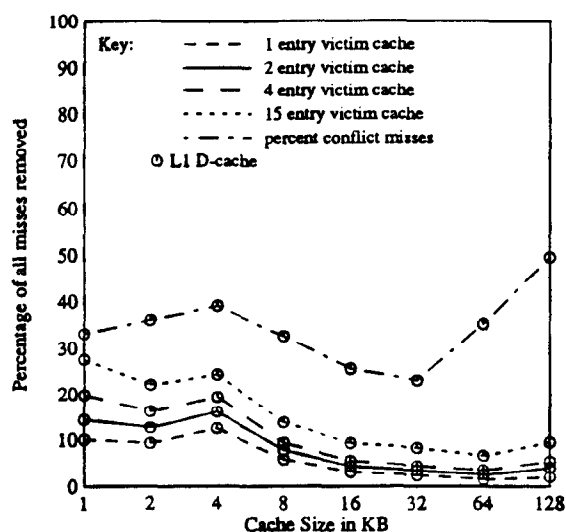

Figure 3-6: Victim cache: vary direct-map cache size

3.4. The Effect of Line Size on Victim Cache Performance

Figure 3-7 shows the performance of victim caches for 4KB direct-mapped data caches of varying line sizes. As one would expect, as the line size at this level increases, the number of conflict misses also increases.
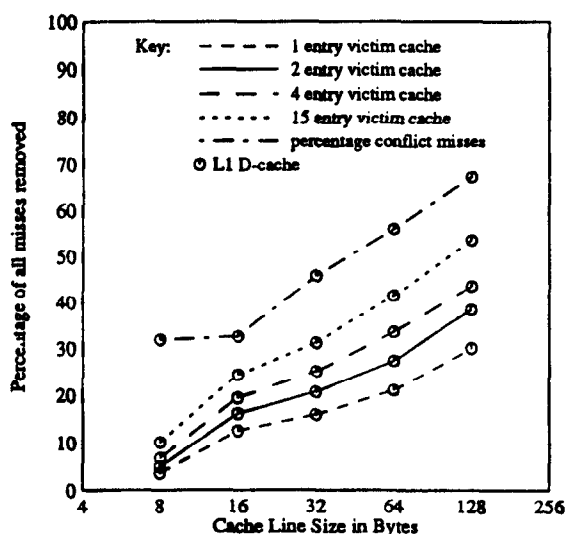

Figure 3-7: Victim cache: vary data cache line size

The increasing percentage of conflict misses results in an increasing percentage of these misses being removed by the victim cache. Systems with victim caches can benefit from longer line sizes more than systems without victim caches, since the victim caches help remove misses caused by conflicts that result from longer cache lines. Note that even if the area used for data storage in the victim cache is held constant (i.e., the

number of entries is cut in half when the line size doubles) the performance of the victim cache still improves or at least breaks even when line sizes increase.

3.5. Victim Caches and Second-Level Caches

As the size of a cache increases, a larger percentage of its misses are due to conflict and compulsory misses and fewer are due to capacity misses. (Unless of course the cache is larger than the entire program, in which case only compulsory misses remain.) Thus victim caches might be expected to be useful for second-level caches as well. Since the number of conflict misses increases with increasing line sizes, the large line sizes of second-level caches would also tend to increase the potential usefulness of victim caches.

One interesting aspect of victim caches is that they violate inclusion properties [1] in cache hierarchies. However, the line size of the second level cache in the baseline design is 8 to 16 times larger than the first-level cache line sizes, so this violates inclusion as well.

Note that a first-level victim cache can contain many lines that conflict not only at the first level but also at the second level. Thus, using a first-level victim cache can also reduce the number of conflict misses at the second level. In investigating victim caches for second-level caches, both configurations with and without first-level victim caches will need to be considered.

A thorough investigation of victim caches for megabyte second-level caches requires traces of billions of instructions. At this time we only have victim cache performance for our smaller test suite, and work on obtaining victim cache performance for multi-megabyte second-level caches is underway.

4. Reducing Capacity and Compulsory Misses

Compulsory misses are misses required in any cache organization because they are the first references to a piece of data. Capacity misses occur when the cache size is not sufficient to hold data between references. One way of reducing the number of capacity and compulsory misses is to use prefetch techniques such as longer cache line sizes or prefetching methods [13, 6]. However, line sizes can not be made arbitrarily large without increasing the miss rate and greatly increasing the amount of data to be transferred. In this section we investigate techniques to reduce capacity and compulsory misses while mitigating traditional problems with long lines and excessive prefetching.

A detailed analysis of three prefetch algorithms has appeared in [13]. *Prefetch always* prefetches after every reference. Needless to say this is impractical in our base system since many level-one cache accesses can take place in the time required to initiate a single level-two cache reference. This is especially true in machines that fetch multiple instructions per cycle from an instruction cache and can concurrently perform a load or store per cycle to a data cache. *Prefetch on miss* and *tagged prefetch* are more promising techniques. On a miss *prefetch on miss* always fetches the next line as well. It can cut the number of misses for a purely sequential reference stream in half. *Tagged prefetch* can do even better. In this technique each block has a tag bit associated with it. When a block is prefetched, its tag bit is set to zero. Each time a block is used its tag bit is set to

one. When a block undergoes a zero to one transition its successor block is prefetched. This can reduce the number of misses in a purely sequential reference stream to zero, if fetching is fast enough. Unfortunately the large latencies in the base system can make this impossible. Consider Figure 4-1, which gives the amount of time (in instruction issues) until a prefetched line is required during the execution of *ccom*. Not surprisingly, since the line size is four instructions, prefetched lines must be received within four instruction-times to keep up with the machine on uncached straight-line code. Because the base system second-level cache takes many cycles to access, and the machine may actually issue many instructions per cycle, tagged prefetch may only have a one-cycle-out-of-many head start on providing the required instructions.
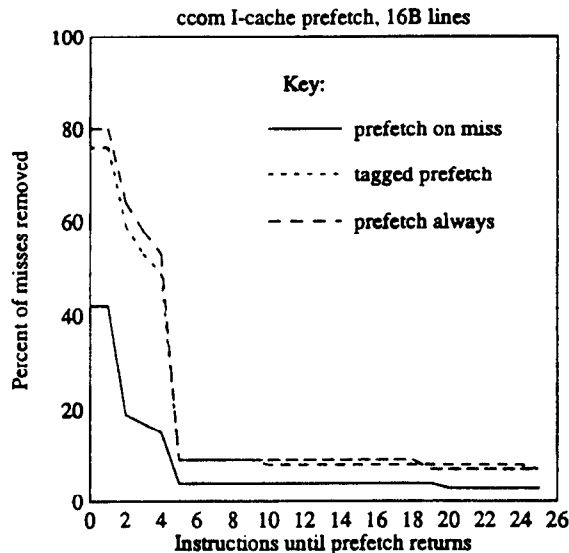

Figure 4-1: Limited time for prefetch

4.1. Stream Buffers

What we really need to do is to start the prefetch before a tag transition can take place. We can do this with a mechanism called a *stream buffer* (Figure 4-2). A stream buffer consists of a series of entries, each consisting of a tag, an available bit, and a data line.

When a miss occurs, the stream buffer begins prefetching successive lines starting at the miss target. As each prefetch request is sent out, the tag for the address is entered into the stream buffer, and the available bit is set to false. When the prefetch data returns it is placed in the entry with its tag and the available bit is set to true. Note that lines after the line requested on the miss are placed in the buffer and not in the cache. This avoids polluting the cache with data that may never be needed.

Subsequent accesses to the cache also compare their address against the first item stored in the buffer. If a reference misses in the cache but hits in the buffer the cache can be reloaded in a single cycle from the stream buffer. This is much faster than the off-chip miss penalty. The stream buffers considered in this section are simple FIFO queues, where only the head of the queue has a tag comparator and elements removed from the buffer must be removed strictly in sequence without

skipping any lines. In this simple model non-sequential line misses will cause a stream buffer to be flushed and restarted at the miss address even if the requested line is already present further down in the queue.

When a line is moved from a stream buffer to the cache, the entries in the stream buffer can shift up by one and a new successive address is fetched. The pipelined interface to the second level allows the buffer to be filled at the maximum bandwidth of the second level cache, and many cache lines can be in the process of being fetched simultaneously. For example, assume the latency to refill a 16B line on an instruction cache miss is 12 cycles. Consider a memory interface that is pipelined and can accept a new line request every 4 cycles. A four-entry stream buffer can provide 4B instructions at a rate of one per cycle by having three requests outstanding at all times. Thus during sequential instruction execution long latency cache misses will not occur. This is in contrast to the performance of tagged prefetch on purely sequential reference streams where only one line is being prefetched at a time. In that case sequential instructions will only be supplied at a bandwidth equal to one instruction every three cycles (i.e., 12 cycle latency / 4 instructions per line).
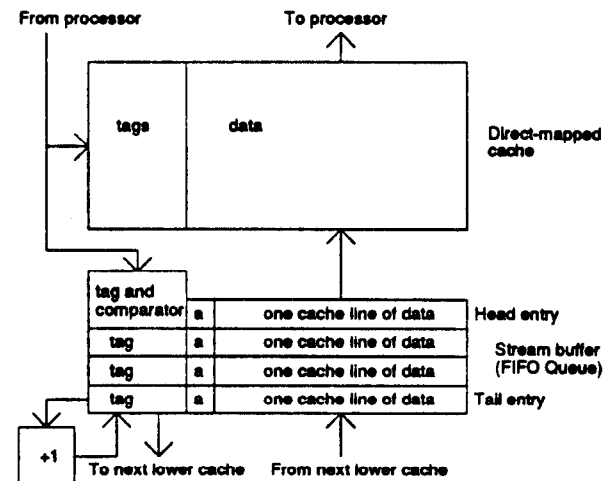

Figure 4-2: Sequential stream buffer design

Figure 4-3 shows the performance of a four-entry instruction stream buffer backing a 4KB instruction cache and a data stream buffer backing a 4KB data cache, each with 16B lines. The graph gives the cumulative number of misses removed based on the number of lines that the buffer is allowed to prefetch after the original miss. (In practice the stream buffer would probably be allowed to fetch until the end of a virtual memory page or a second-level cache line. The major reason for plotting stream buffer performance as a function of prefetch length is to get a better idea of how far streams continue on average.) Most instruction references break the purely sequential access pattern by the time the 6th successive line is fetched, while many data reference patterns end even sooner. The exceptions to this appear to be instruction references for *liver* and data references for *linpack. liver* is probably an anomaly since the 14 loops of the program are executed sequentially, and the first 14 loops do not generally call other procedures or do excessive branching, which would

cause the sequential miss pattern to break. The data reference pattern of *linpack* can be understood as follows. Remember that the stream buffer is only responsible for providing lines that the cache misses on. The inner loop of *linpack* (i.e., saxpy) performs an inner product between one row and the other rows of a matrix. The first use of the one row loads it into the cache. After that subsequent misses in the cache (except for mapping conflicts with the first row) consist of subsequent lines of the matrix. Since the matrix is too large to fit in the on-chip cache, the whole matrix is passed through the cache on each iteration. The stream buffer can do this at the maximum bandwidth provided by the second-level cache. Of course one prerequisite for this is that the reference stream is unit-stride or at most skips to every other or every third word. If an array is accessed in the non-unit-stride direction (and the other dimensions have non-trivial extents) then a stream buffer as presented here will be of little benefit.
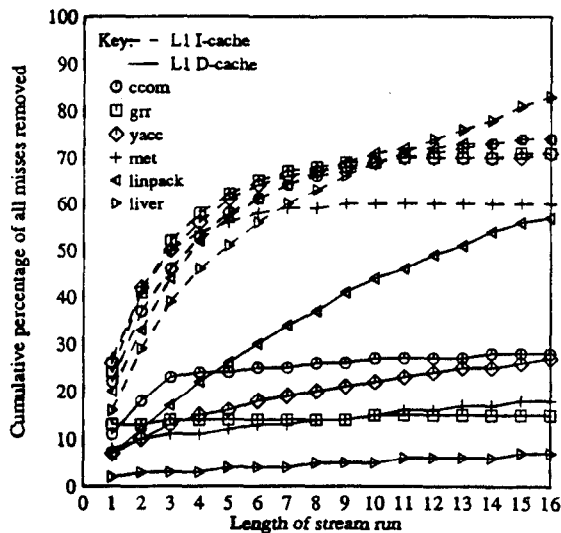

Figure 4-3: Sequential stream buffer performance

4.2. Multi-Way Stream Buffers

Overall, the stream buffer presented in the previous section could remove 72% of the instruction cache misses, but it could only remove 25% of the data cache misses. One reason for this is that data references tend to consist of interleaved streams of data from different sources. In order to improve the performance of stream buffers for data references, a multi-way stream buffer was simulated (Figure 4-4). It consists of four stream buffers in parallel. When a miss occurs in the data cache that does not hit in any stream buffer, the stream buffer hit least recently is cleared (i.e., LRU replacement) and it is started fetching at the miss address.

Figure 4-5 shows the performance of the multi-way stream buffer on our benchmark set. As expected, the performance on the instruction stream remains virtually unchanged. This means that the simpler single stream buffer will suffice for instruction streams. The multi-way stream buffer does significantly improve the performance on the data side, however. Overall, the multi-way stream buffer can remove 43% of the misses for the six programs, almost twice the performance of the single stream buffer. Although the matrix operations of *liver*

experience the greatest improvement (it changes from 7% to 60% reduction), all of the programs benefit to some extent.
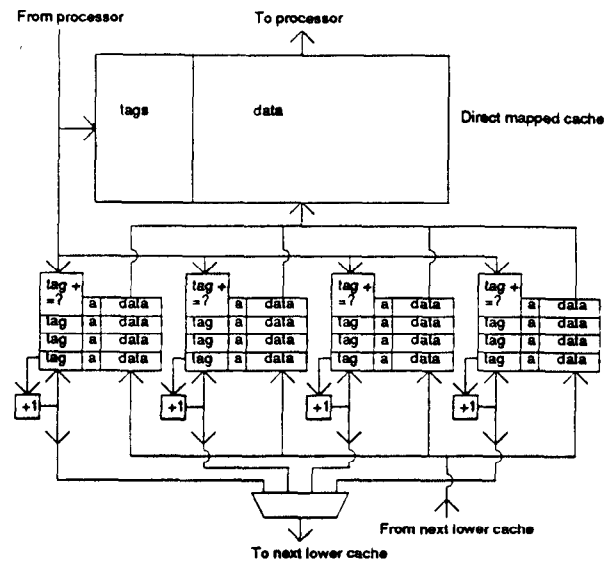

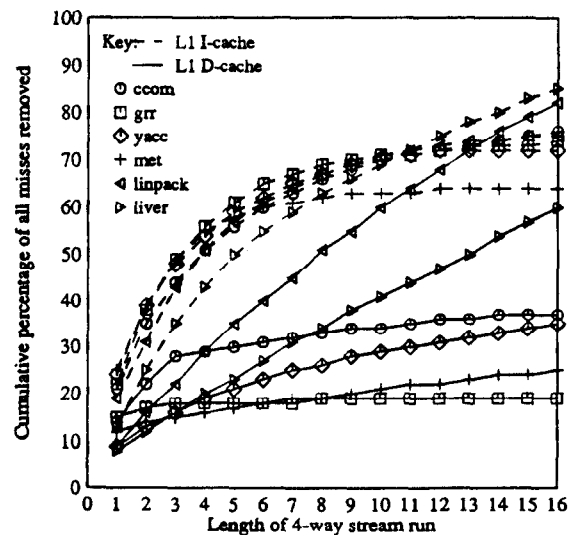
Figure 4-4: Four-way stream buffer design

Figure 4-5: Four-way stream buffer performance

4.3. Stream Buffer Performance vs. Cache Size

Figure 4-6 gives the performance of single and 4-way stream buffers with 16B lines as a function of cache size. The instruction stream buffers have remarkably constant performance over a wide range of cache sizes. The data stream buffer performance generally improves as the cache size increases. This is especially true for the single stream buffer, whose performance increases from a 15% reduction in misses for a data cache size of 1KB to a 35% reduction in misses for a data cache size of 128KB. This is probably because as the cache size increases, it can contain data for reference patterns that access several sets of data, or at least all but one of the

sets. What misses that remain are more likely to consist of very long single sequential streams. For example, as the cache size increases the percentage of compulsory misses increase, and these are more likely to be sequential in nature than data conflict or capacity misses.
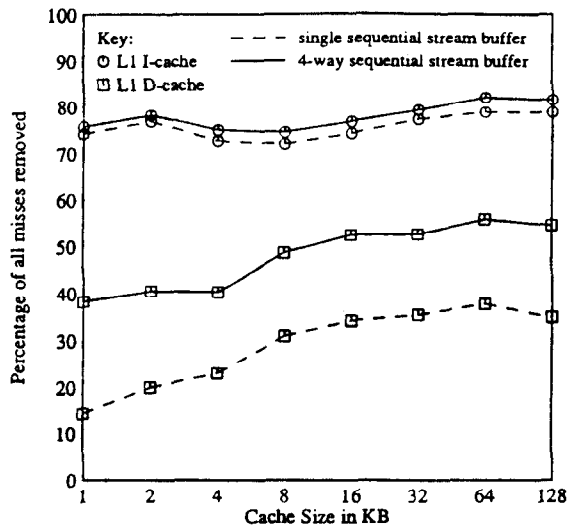

Figure 4-6: Stream buffer performance vs. cache size

4.4. Stream Buffer Performance vs. Line Size

Figure 4-7 gives the performance of single and 4-way stream buffers as a function of the line size in the stream buffer and 4KB cache. The reduction in misses provided by a single data stream buffer falls by a factor of 6.8 going from a line size of 8B to a line size of 128B, while a 4-way stream buffer's contribution falls by a factor of 4.5. This is not too surprising since data references are often fairly widely distributed. In other words if a piece of data is accessed, the odds that another piece of data 128B away will be needed soon are fairly low. The single data stream buffer performance is especially hard hit compared to the multi-way stream buffer because of the increase in conflict misses at large line sizes.
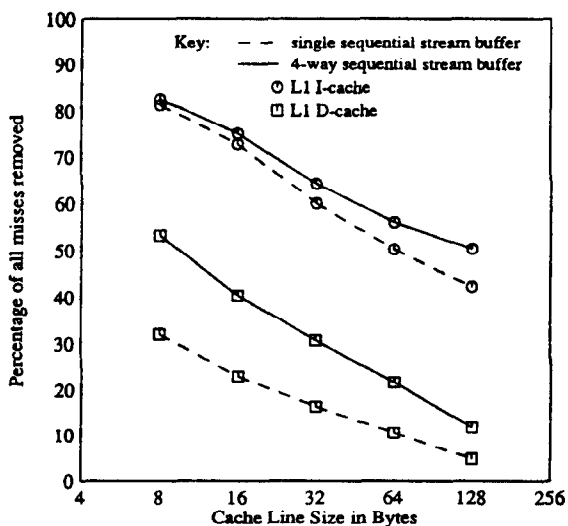

Figure 4-7: Stream buffer performance vs. line size

The instruction stream buffers perform well even out to 128B line sizes. Both the 4-way and the single stream buffer still remove at least 40% of the misses at 128B line sizes, coming down from an 80% reduction with 8B lines. This is probably due to the large granularity of conflicting instruction reference streams, and the fact that many procedures are more than 128B long.

5. Conclusions

Small miss caches (e.g., 2 to 5 entries) have been shown to be effective in reducing data cache conflict misses for direct-mapped caches in range of 1K to 8K bytes. They effectively remove tight conflicts where misses alternate between several addresses that map to the same line in the cache. Miss caches are increasingly beneficial as line sizes increase and the percentage of conflict misses increases. In general it appears that as the percentage of conflict misses increases, the percent of these misses removable by a miss cache also increases, resulting in an even steeper slope for the performance improvement possible by using miss caches.

Victim caches are an improvement to miss caching that saves the victim of the cache miss instead of the target in a small associative cache. Victim caches are even more effective at removing conflict misses than miss caches.

Stream buffers prefetch cache lines after a missed cache line. They store the line until it is requested by a cache miss (if ever) to avoid unnecessary pollution of the cache. They are particularly useful at reducing the number of capacity and compulsory misses. They can take full advantage of the memory bandwidth available in pipelined memory systems for sequential references, unlike previously discussed prefetch techniques such as tagged prefetch or prefetch on miss. Stream buffers can also tolerate longer memory system latencies since they prefetch data much in advance of other prefetch techniques (even prefetch always). Stream buffers can also compensate for instruction conflict misses, since these tend to be relatively sequential in nature as well.

Multi-way stream buffers are a set of stream buffers that can prefetch down several streams concurrently. Multi-way stream buffers are useful for data references that contain interleaved accesses to several different large data structures, such as in array operations. However, since the prefetching is of sequential lines, only unit stride or near unit stride (2 or 3) access patterns benefit.

The performance improvements due to victim caches and due to stream buffers are relatively orthogonal for data references. Victim caches work well where references alternate between two locations that map to the same line in the cache. They do not prefetch data but only do a better job of keeping data fetched available for use. Stream buffers, however, achieve performance improvements by prefetching data. They do not remove conflict misses unless the conflicts are widely spaced in time, and the cache miss reference stream consists of many sequential accesses. These are precisely the conflict misses not handled well by a victim cache due to its relatively small capacity. Over the set of six benchmarks, on average only 2.5% of 4KB direct-mapped data cache misses that hit in a four-entry victim cache also hit in a four-way stream buffer for *ccom*, *met*, *yacc*, *grr*, and *liver*. In contrast, *linpack*, due to its se-

quential data access patterns, has 50% of the hits in the victim cache also hit in a four-way stream buffer. However only 4% of *linpack*'s cache misses hit in the victim cache (it benefits least from victim caching among the six benchmarks), so this is still not a significant amount of overlap between stream buffers and victim caching.

Figure 5-1 shows the performance of the base system with the addition of a four entry data victim cache, a instruction stream buffer, and a four-way data stream buffer. (The base system has on-chip 4KB instruction and 4KB data caches with 24 cycle miss penalties and 16B lines to a three-stage pipelined second-level 1MB cache with 128B lines and 320 cycle miss penalty.) The lower solid line in Figure 5-1 gives the performance of the original base system without the victim caches or buffers while the upper solid line gives the performance with buffers and victim caches. The combination of these techniques reduces the first-level miss rate to less than half of that of the baseline system, resulting in an average of 143% improvement in system performance for the six benchmarks. These results show that the addition of a small amount of hardware can dramatically reduce cache miss rates and improve system performance.
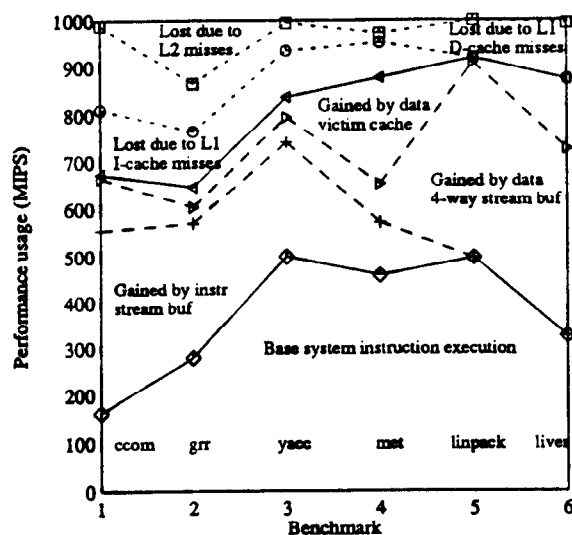

Figure 5-1: Improved system performance

This study has concentrated on applying victim caches and stream buffers to first-level caches. An interesting area for future work is the application of these techniques to second-level caches. Also, the numeric programs used in this study used unit stride access patterns. Numeric programs with non-unit stride and mixed stride access patterns also need to be simulated. Finally, the performance of victim caching and stream buffers needs to be investigated for operating system execution and for multiprogramming workloads.

Acknowledgements

References

1. Baer, Jean-Loup, and Wang, Wenn-Hann. On the Inclusion Properties for Multi-Level Cache Hierarchies. The 15th Annual Symposium on Computer Architecture, IEEE Computer Society Press, June, 1988, pp. 73-80.

2. Borg, Anita, Kessler, Rick E., Lazana, Georgia, and Wall, David W. Long Address Traces from RISC Machines: Generation and Analysis. Tech. Rept. 89/14, Digital Equipment Corporation Western Research Laboratory, September, 1989.

3. Digital Equipment Corporation, Inc. *VAX Hardware Handbook, volume 1 - 1984.* Maynard, Massachusetts, 1984.

4. Emer, Joel S., and Clark, Douglas W. A Characterization of Processor Performance in the VAX-11/780. The 11th Annual Symposium on Computer Architecture, IEEE Computer Society Press, June, 1984, pp. 301-310.

5. Eustace, Alan. Private communication.

6. Farrens, Matthew K., and Pleszkun, Andrew R. Improving Performance of Small On-Chip Instruction Caches. The 16th Annual Symposium on Computer Architecture, IEEE Computer Society Press, May, 1989, pp. 234-241.

7. Hill, Mark D. *Aspects of Cache Memory and Instruction Buffer Performance.* Ph.D. Th., University of California, Berkeley, 1987.

8. Jouppi, Norman P., and Wall, David W. Available Instruction-Level Parallelism For Superpipelined and Superscalar Machines. Third International Conference on Architectural Support for Programming Languages and Operating Systems, IEEE Computer Society Press, April, 1989, pp. 272-282.

9. Jouppi, Norman P. Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU. The 16th Annual Symposium on Computer Architecture, IEEE Computer Society Press, May, 1989, pp. 281-289.

10. Nielsen, Michael J. K. Titan System Manual. Tech. Rept. 86/1, Digital Equipment Corporation Western Research Laboratory, September, 1986.

11. Ousterhout, John. Why Aren't Operating Systems Getting Faster As Fast As Hardware? Tech. Rept. Technote 11, Digital Equipment Corporation Western Research Laboratory, October, 1989.

12. Smith, Alan J. "Sequential program prefetching in memory hierarchies." *IEEE Computer 11,* 12 (December 1978), 7-21.

13. Smith, Alan J. "Cache Memories." *Computing Surveys* (September 1982), 473-530.