M A R C H 1 9 9 0

WRL Technical Note TN-14

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

Norman P. Jouppi

Western Research Laboratory 250 University Avenue Palo Alto, California 94301 USA

The Western Research Laboratory (WRL) is a computer systems research group that was founded by Digital Equipment Corporation in 1982. Our focus is computer science research relevant to the design and application of high performance scientific computers. We test our ideas by designing, building, and using real systems. The systems we build are research prototypes; they are not intended to become products.

There is a second research laboratory located in Palo Alto, the Systems Research Center (SRC). Other Digital research groups are located in Paris (PRL) and in Cambridge, Massachusetts (CRL).

Our research is directed towards mainstream high-performance computer systems. Our prototypes are intended to foreshadow the future computing environments used by many Digital customers. The long-term goal of WRL is to aid and accelerate the development of high-performance uni- and multi-processors. The research projects within WRL will address various aspects of high-performance computing.

We believe that significant advances in computer systems do not come from any single technological advance. Technologies, both hardware and software, do not all advance at the same pace. System design is the art of composing systems which use each level of technology in an appropriate balance. A major advance in overall system performance will require reexamination of all aspects of the system.

We do work in the design, fabrication and packaging of hardware; language processing and scaling issues in system software design; and the exploration of new applications areas that are opening up with the advent of higher performance systems. Researchers at WRL cooperate closely and move freely among the various levels of system design. This allows us to explore a wide range of tradeoffs to meet system goals.

We publish the results of our work in a variety of journals, conferences, research reports, and technical notes. This document is a technical note. We use this form for rapid distribution of technical material. Usually this represents research in progress. Research reports are normally accounts of completed research and may include material from earlier technical notes.

Research reports and technical notes may be ordered from us. You may mail your order to:

Technical Report Distribution DEC Western Research Laboratory, WRL-2
250 University Avenue Palo Alto, California 94301 USA

Reports and notes may also be ordered by electronic mail. Use one of the following addresses:

Digital E-net:	`JOVE::WRL-TECHREPORTS`
Internet:	`WRL-Techreports@decwrl.pa.dec.com`
UUCP:	`decpa!wrl-techreports`

To obtain more details on ordering by electronic mail, send a message to one of these addresses with the word ‘‘`help`’’ in the Subject line; you will receive detailed instructions.

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

Norman P. Jouppi

March, 1990

d|i|g|i|t|a|l **Western Research Laboratory** 250 University Avenue Palo Alto, California 94301 USA

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 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 tremendous 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 [11] 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: 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 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 [9]), 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.

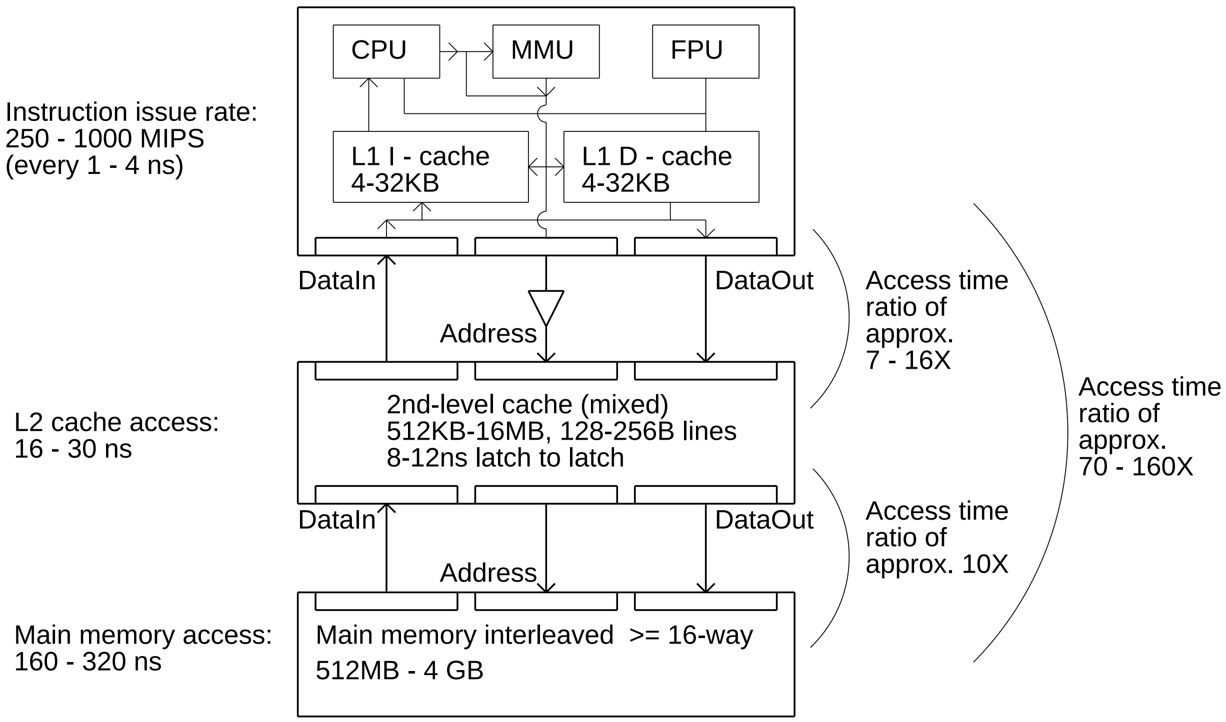

Figure 1: Baseline design

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 [8, 13], 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 [12], 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. 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 3.

The effects of these miss rates are given graphically in Figure 2. The region below the solid line gives the net performance of the system, while the region above the solid line gives the

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 tool
yacc	51.0M	16.7M	67.7M	Unix utility
met	99.4M	50.3M	149.7M	PC board CAD tool
linpack	144.8M	40.7M	185.5M	numeric, 100x100
liver	23.6M	7.4M	31.0M	LFK (numeric loops)
total	484.5M	188.3M	672.8M	

Table 2: Test program characteristics

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 3: Baseline system first-level cache miss rates

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, 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.

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.

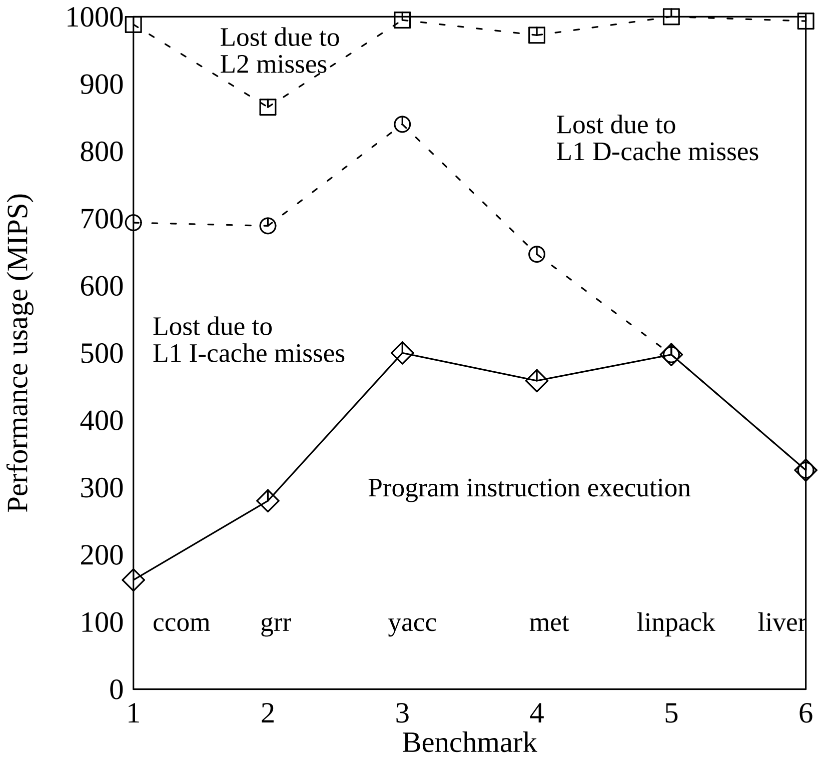

Figure 2: Baseline design performance

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 [10]. Conflict misses typically account for between 20% and 40% of all direct-mapped cache misses [7]. Figure 3 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.

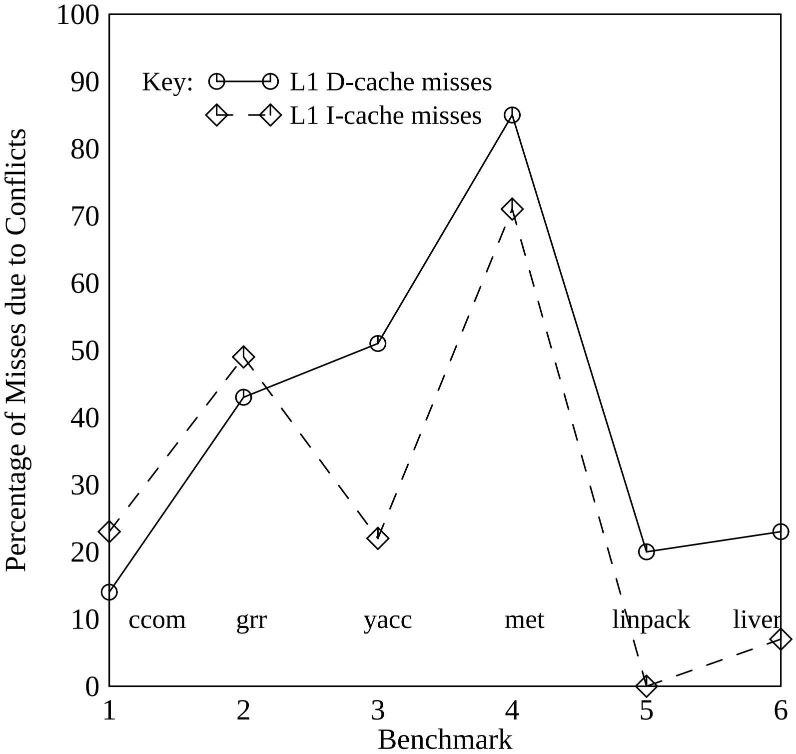

Figure 3: Percentage of conflict misses, 4K I and D caches, 16B lines

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 4). 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.

The success of different miss cache organizations at removing conflict misses is shown in Figure 5. 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.

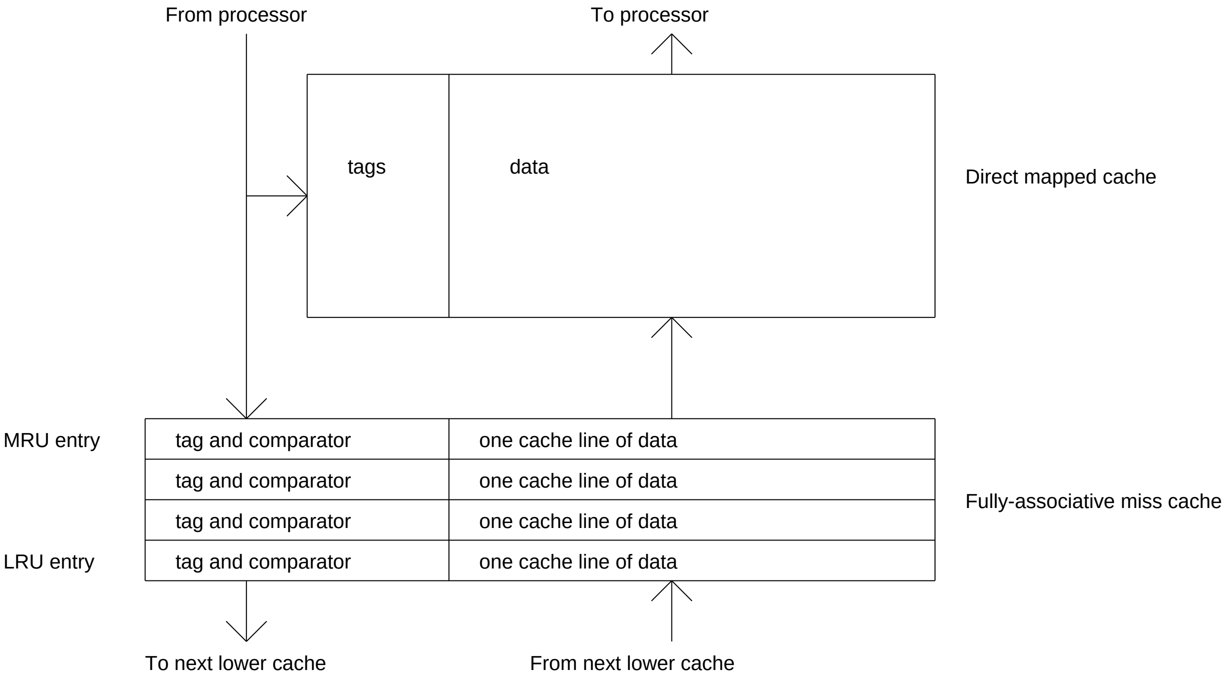

Figure 4: Miss cache organization

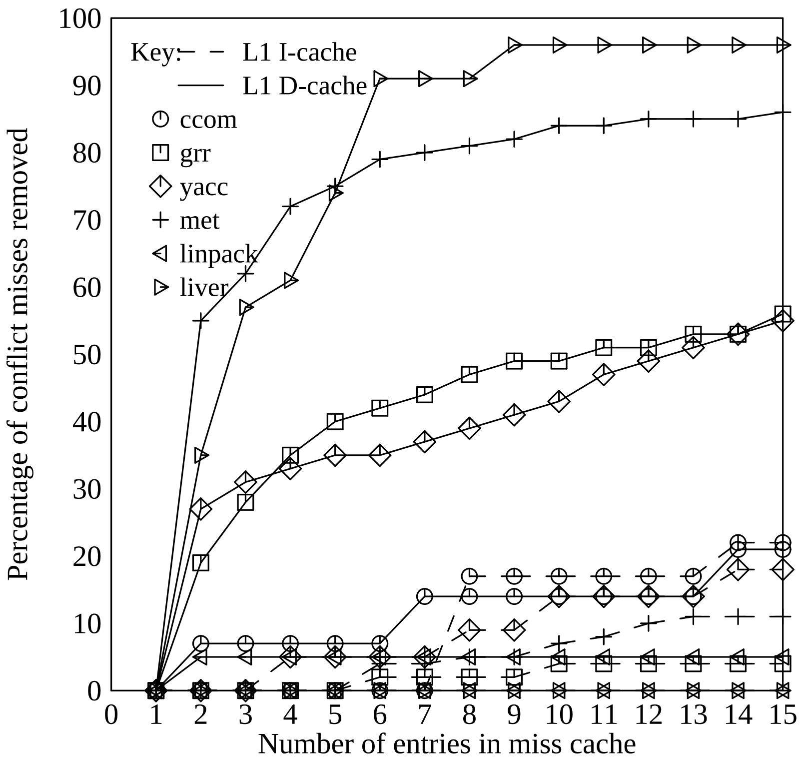

Figure 5: Conflict misses removed by miss caching

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 5 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,* or 13% of the data cache misses overall (see Figure 6). 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.

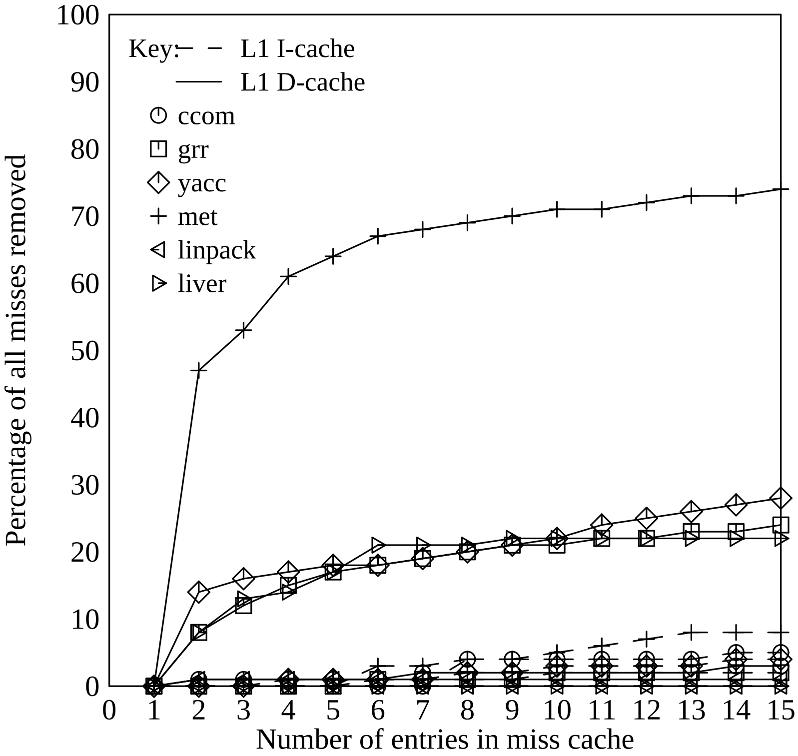

Figure 6: Overall cache misses removed by miss caching

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.

---

*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.

Comparing Figure 5 and Figure 3, 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 *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 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 7). 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.

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 conflicting 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 8. 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.

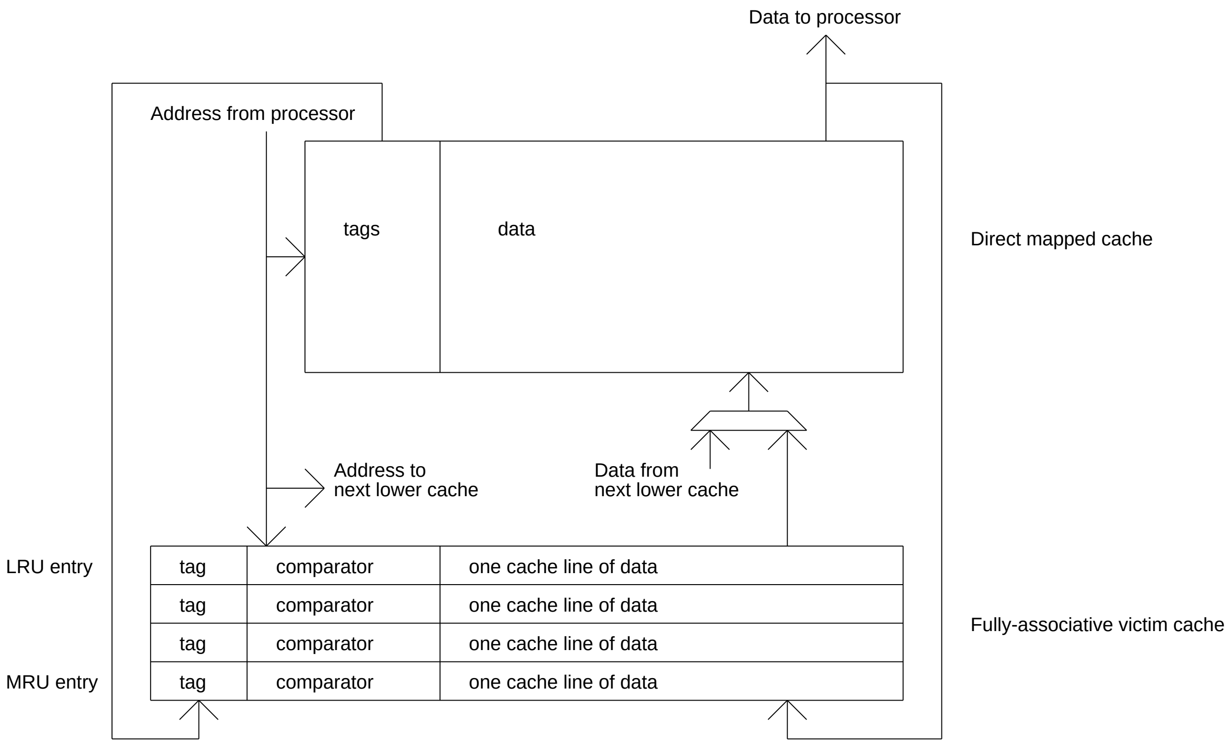

Figure 7: Victim cache organization

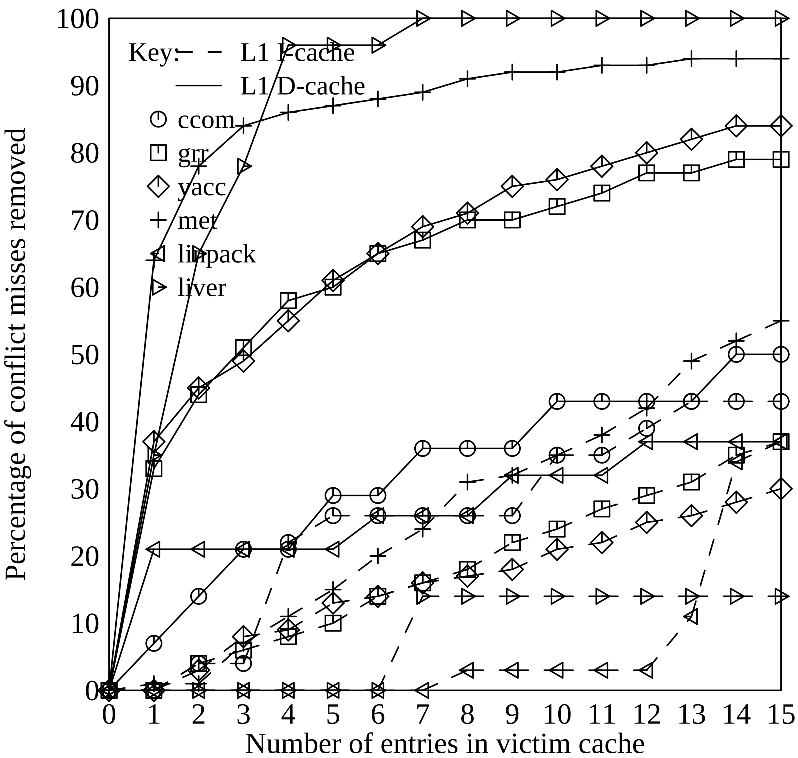

Figure 8: Conflict misses removed by victim caching

The reduction in conflict misses obtained by victim caching is shown in Figure 9 relative to the performance of a 2-way set associative cache (the key is the same as for Figure 8). Note that a one-entry victim cache provides about 50% of the benefit of 2-way set-associativity for *liver*, *met*, and *grr*. In fact, a direct-mapped cache with a 2-entry victim cache performs better than a 2-way set associative cache on *liver*. The dashed line for *linpack* instruction references at a victim cache size of seven results from the fact that a 2-way set-associative instruction cache performs slightly worse for *linpack* than a direct-mapped cache, and a direct-mapped cache with an 8-entry victim cache performs slightly better than a direct-mapped cache alone.

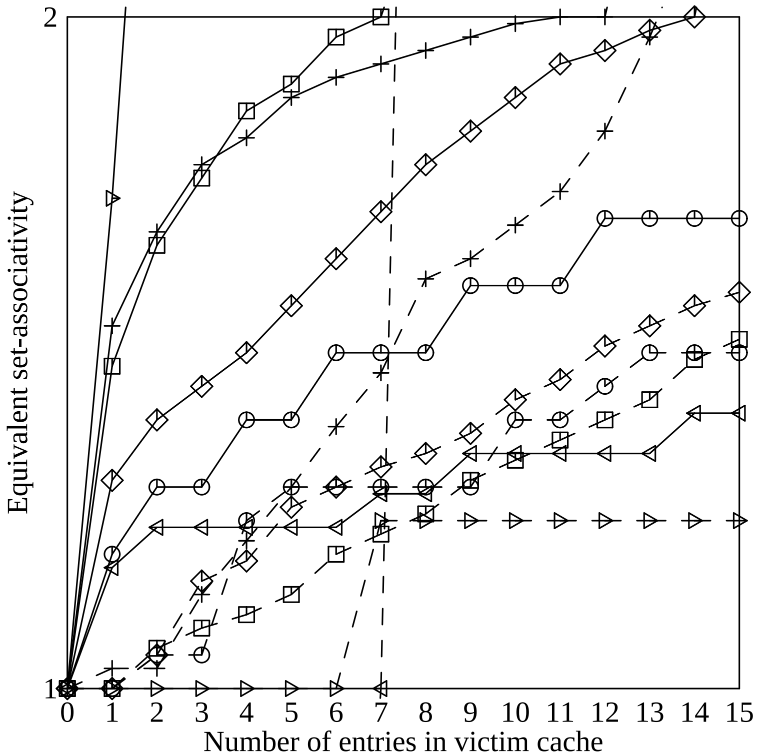

Figure 9: Equivalent set-associativity provided by victim caching

Figure 10 shows the overall reduction in miss rate possible with victim caching. As can be seen by comparing Figure 6 and Figure 10, the performance of the victim cache is in some cases better than a miss cache with twice the number of entries. For example, consider *yacc*'s data cache performance with a one-entry victim cache and a two-entry miss cache. Because the victim cache doesn't throw away the victim, in some situations victim caching can result in fewer misses than a miss cache with twice the number of entries. For example, imagine many cache misses occur accessing new data (i.e., compulsory misses), effectively flushing out both a miss cache and a victim cache. Next imagine another new line is referenced, causing a miss for both a system with a miss cache and a system with a victim cache. If the old contents of the line are referenced next, the miss cache will not contain the item, but a victim cache would. Thus the system with a miss cache would have two misses to the next level in the memory hierarchy, while the system with a victim cache would only have one.

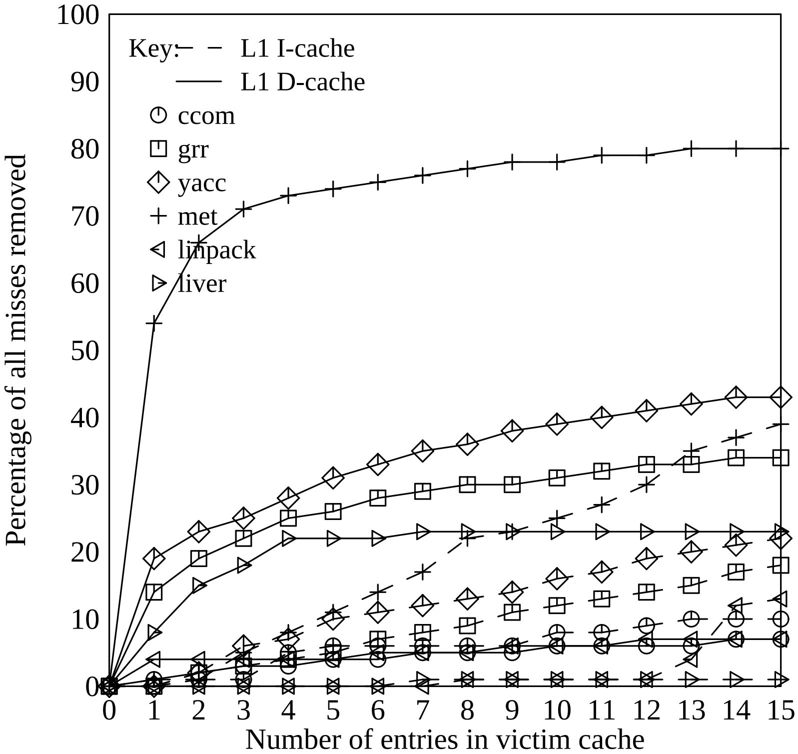

Figure 10: Overall cache misses removed by victim caching

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

Figure 11 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.

3.4. The Effect of Line Size on Victim Cache Performance

Figure 12 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. 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

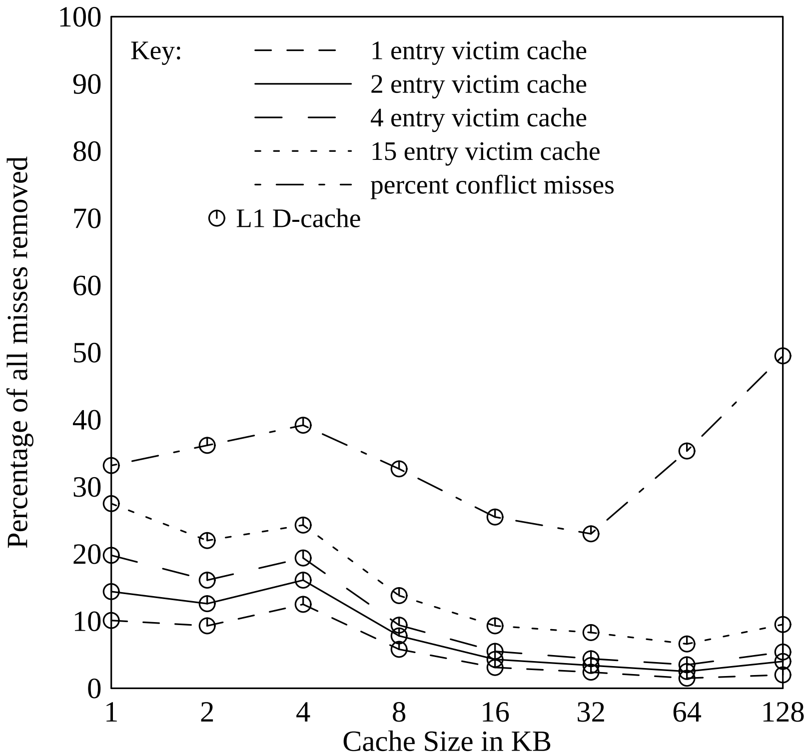

Figure 11: Victim cache performance with varying direct-mapped data cache size

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.

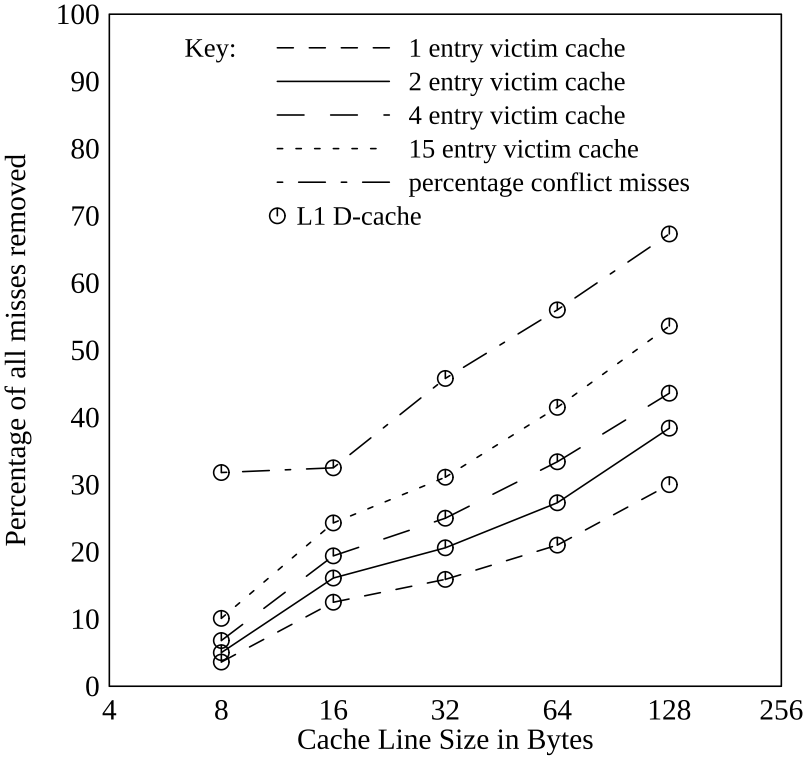

Figure 12: Victim cache performance with varying data cache line size

3.6. Miss Caches, Victim Caches, and Error Correction

Another important use for miss caches, especially on-chip at the first-level, is in yield enhancement and fault tolerance. If parity is kept on all instruction and data cache bytes, and the data cache is write-though, then cache parity errors can be handled as misses. If the refill path bypasses the cache, then this scheme can also allow chips with hard errors to be used. (In fact with byte parity, up to 1/9 of all bits in the cache could be faulty as long as there were at most one bad bit per byte.) Unfortunately, without miss caches if the inner loop of *linpack* (i.e., saxpy) happens to land on a line with a defect or if a frequently used structure variable is on a defective line, the performance of the system can be severely degraded (e.g., by greater than a factor of four on some code segments). Moreover the performance degradation would vary from chip to chip seemingly at random depending on defect location. This would limit the potential yield enhancement to the engineering development phase of a project. However, with the addition of miss caches, the penalty on a defect-induced parity miss is only one cycle, which would have a much smaller impact on machine performance than an off-chip miss. Thus, as long as the number of defects was small enough to be handled by the miss cache, chips with hard defects could be used in production systems. If miss caches are used to improve system performance in the presence of fabrication defects, then instruction miss caches and even miss caches with only one entry would be useful.

Victim caches as described earlier would not be useful for correction of misses due to parity errors. This is because the victim is corrupted by the parity error, and is not worth saving. However victim caches can also be used for error-correction with the following change. When a cache miss is caused by a parity error, the victim cache is loaded with the incoming (miss) data and not the victim. Thus it acts like a victim cache for normal misses and a miss cache for parity misses. With this minor modification the benefits of miss caches for error-recovery and the better performance of victim caching can be combined.

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 [14, 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.

4.1. Reducing Capacity and Compulsory Misses with Long Lines

If conflict misses did not exist, caches with much larger line sizes would be appropriate. Figure 13 shows the reduction in compulsory and capacity misses with increasing line size, compared to a baseline design with 8B lines. In general, all benchmarks have reduced miss rates as the line size is increased, although *yacc* has anomalous instruction cache behavior at 64B line sizes.

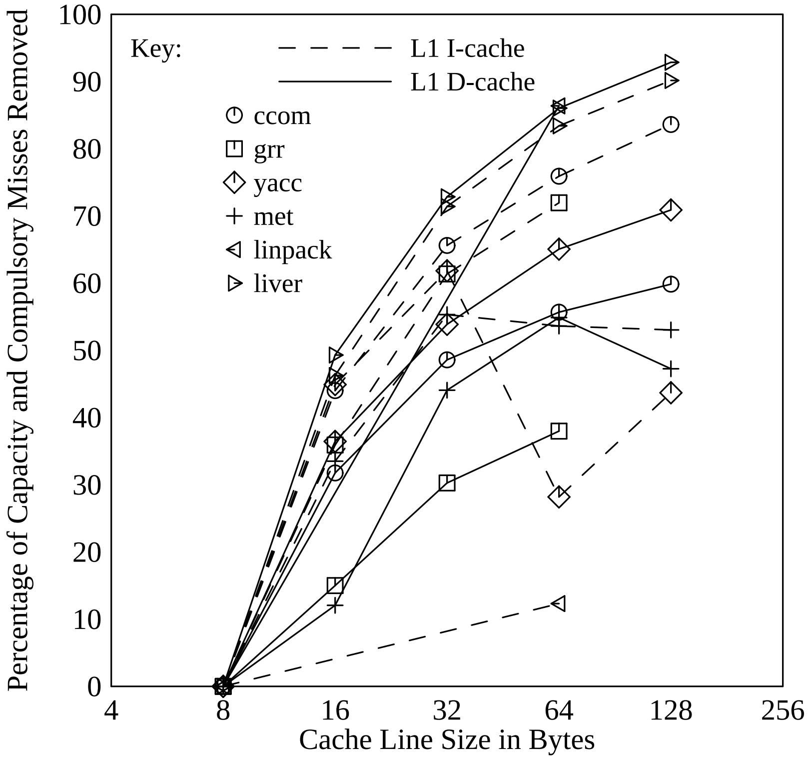

Figure 13: Effect of increasing line size on capacity and compulsory misses

However, when the effects of conflict misses are included, the picture changes dramatically (see Figure 14). As can be seen, the instruction cache performance still increases with increasing line size but the data cache performance peaks at a modest line size and decreases for further increases in line size beyond that. This is a well known effect and is due to differences in spatial locality between instruction and data references. For example, when a procedure is called, many instructions within a given extent will be executed. However, data references tend to be much more scattered, especially in programs that are not based on unit-stride array access. Thus the long line sizes are much more beneficial to quasi-sequential instruction access patterns than the more highly distributed data references.

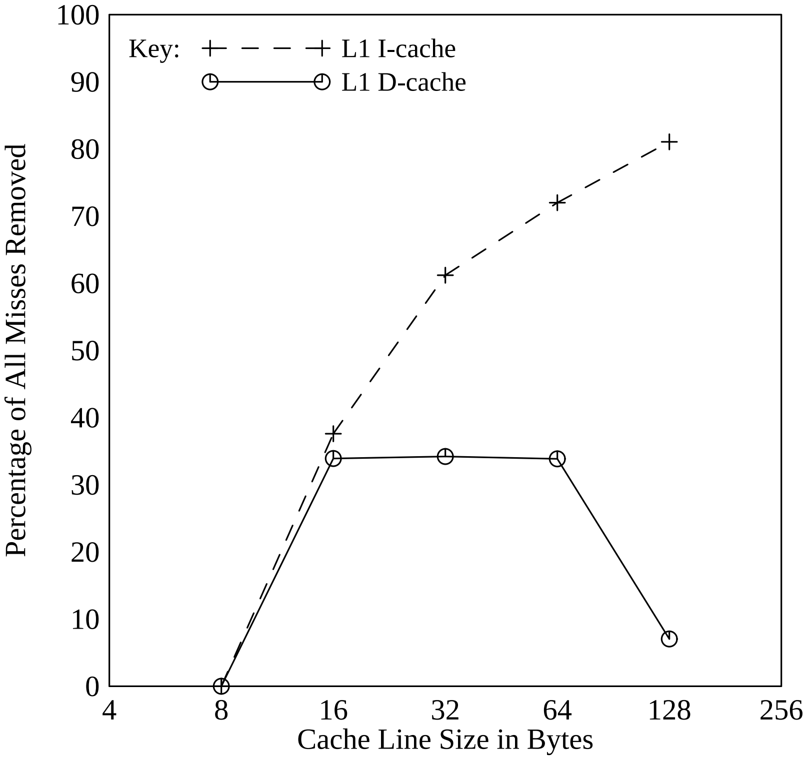

Figure 14: Effect of increasing line size on overall miss rate

Although curves of average performance such as Figure 14 appear to roll off fairly smoothly, the performance for individual programs can be quite different. Figure 15 shows that the data cache line size providing the best performance actually varies from 16B to 128B, depending on the program. Moreover, within this range programs can have dramatically different performance. For example, *liver* has about half the number of data cache misses at a line size of 128B as compared to 16B, but *met* has about three times the number of misses at 128B as compared to 16B. Similarly the performance of *yacc* degrades precipitously at line sizes above 16B. This shows one problem with large line sizes: different programs have dramatically different performance. For programs with long sequential reference patterns, relatively long lines would be useful, but for programs with more diffuse references shorter lines would be best. Taking it a step further, even within a given program the optimal line size is different for the different references that a program makes.

Since the performance in Figure 13 increases fairly monotonically with increasing line size, we know the steep drops in performance in Figure 15 are due to increasing numbers of conflict misses. Since miss caches tend to remove a higher percentage of conflict misses when conflicts

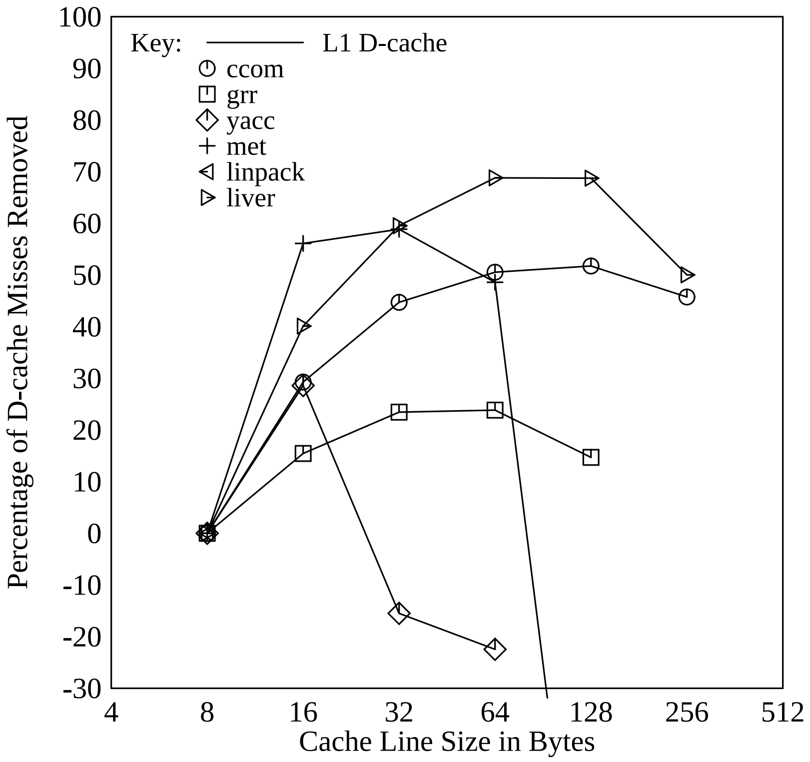

Figure 15: Effect of increasing data cache line size on each benchmark

are frequent, miss caches should allow us to take better advantage of longer cache line sizes. Figure 16 shows the average effectiveness of increasing line size in configurations with and without miss caches. By adding a miss cache more benefits can be derived from a given increase in line size, as well increasing the line size at which the minimum miss rate occurs. This effect can be quite significant: increasing the line size from 16B to 32B with a 4-entry miss cache decreases the miss rate by 36.3%, but only decreases it by 0.5% on average when increasing the line size without a miss cache. Table 4 shows the minimum miss rate for each benchmark with and without miss caches. Benchmarks with minimum miss rate line sizes that are not powers of two have equal miss rates at the next larger and smaller powers of two. The geometric mean over the six benchmarks of the line size giving the lowest miss rate increases from 46B to 92B with the addition of a 4-entry miss cache. The minimum line size giving the best performance on any of the six benchmarks also increases from 16B to 32B with the addition of a 4-entry miss cache.

Figure 17 shows the detailed behavior of most of the programs. The performance of systems with 8B lines are all normalized to zero, independent of the size of their miss cache (if any). This removes the reduction in misses simply due to miss caching from the comparison of the effects of longer cache lines. Thus the actual performance of systems with miss caches at 16B lines are all better than systems without miss caches.

Systems with miss caching continue to obtain benefits from longer line sizes where systems without miss caches have flat or decreasing performance. Figure 18 shows the effects of longer cache line sizes on *yacc* and *met* with varying miss cache sizes, similarly normalized to performance with 8B lines. The performance of *yacc* is affected most dramatically - the sharp dropoff

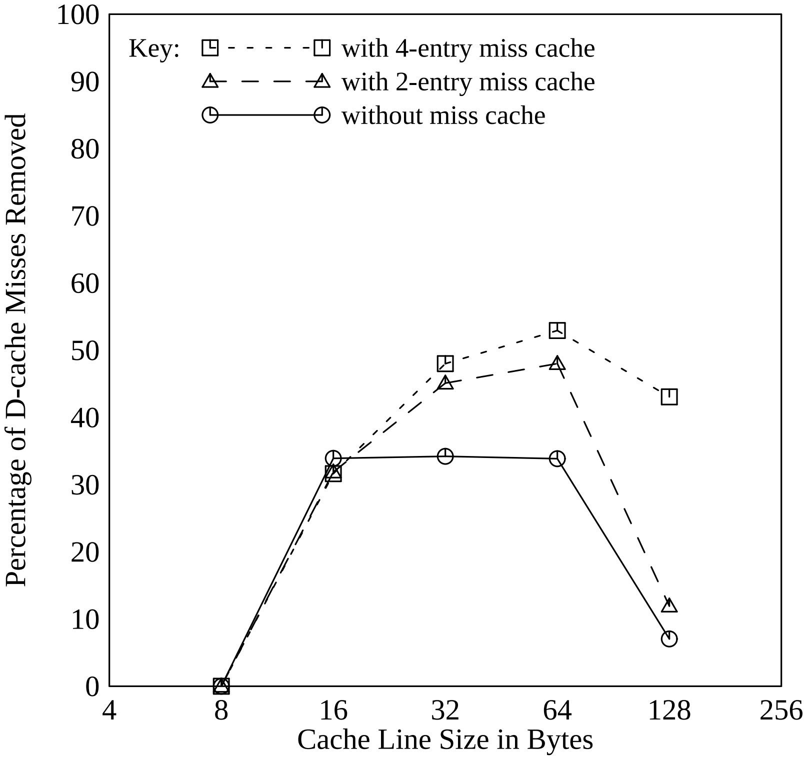

Figure 16: Effect of increasing data cache line size with miss caches

miss cache entries	ccom	grr	yacc	met	liver	geom mean	min
4	256	96	64	32	128	92	32
2	128	64	128	32	128	84	32
0	128	48	16	32	64	46	16

Table 4: Line sizes with minimum miss rates by program

at line sizes above 16B is completely eliminated even with miss caches with as few as two entries. The performance of *met* is a little more subtle. A system with a miss cache, although always performing better than a system without a miss cache, does not benefit as much on *met* from an increase in line size. Thus the number of additional misses removed with longer lines when using miss caches for *met* is lower than when not using a miss cache for line sizes in the range of 16B to 64B. However the absolute miss rate (not shown) is still lower when using the miss caches. At line sizes of 128B, adding a miss cache with four entries can turn a 100% increase in miss rate for *met* into only a 22% increase in miss rate, although a two entry miss cache has little effect. This benchmark is the primary reason why the average performance of two-entry and four-entry miss caches in Figure 16 diverge at a line size of 128B.

Miss caches for very large lines or with more than four entries at moderate line sizes were not simulated. As line sizes become larger, the amount of storage required by the miss cache increases dramatically: with our 4KB cache an 8-entry miss cache with 128B lines requires an amount of storage equal to 1/4 the total cache size! An interesting area of future research for

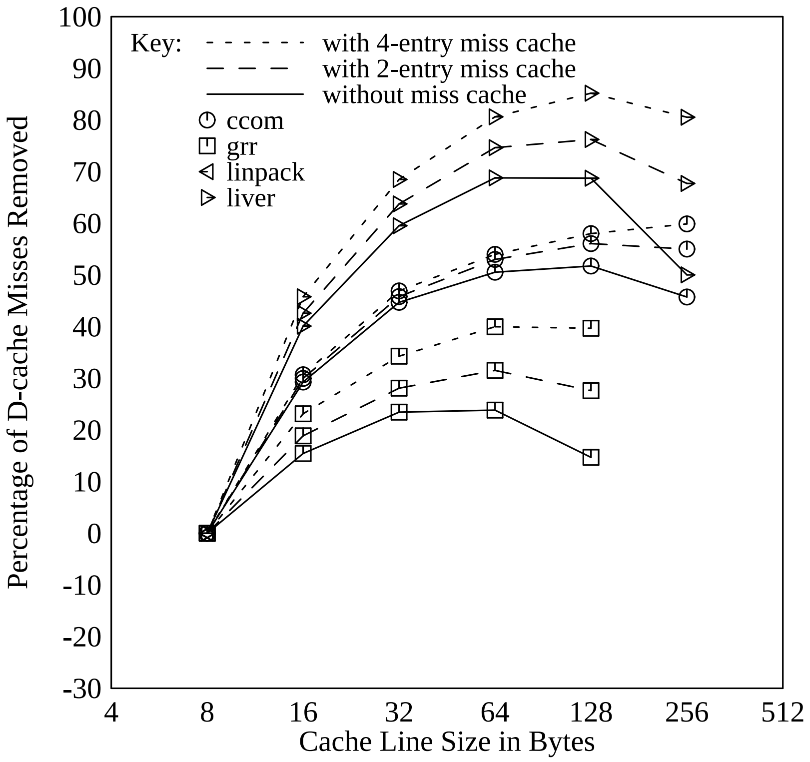

Figure 17: Benchmark-specific performance with increasing data cache line size

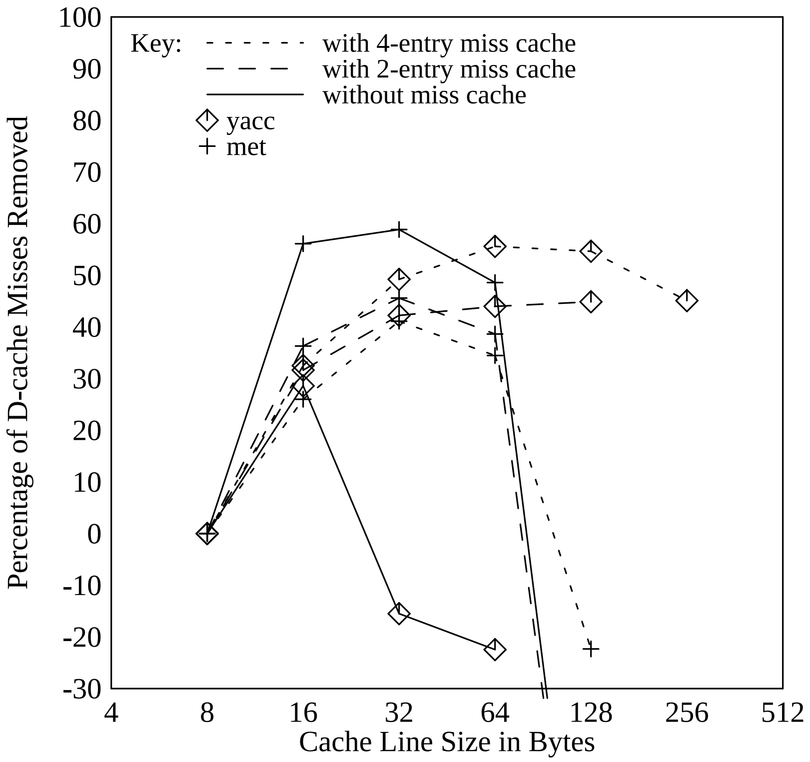

Figure 18: *yacc* and *met* performance with increasing data cache line size

systems with very long lines is the possibility of miss caching on subblocks. Much of the benefit of full-line miss caches might then be obtained with a fraction of the storage requirements.

4.2. Reducing Capacity and Compulsory Misses with Prefetch Techniques

Longer line sizes suffer from the disadvantage of providing a fixed transfer size for different programs and access patterns. Prefetch techniques are interesting because they can be more adaptive to the actual access patterns of the program. This is especially important for improving the performance on long quasi-sequential access patterns such as instruction streams or unit-stride array accesses.

A detailed analysis of three prefetch algorithms has appeared in [14]. *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 19, 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.

4.2.1. 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 20). 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. More complicated stream buffers that can provide already-fetched lines out of sequence are discussed in following sections.

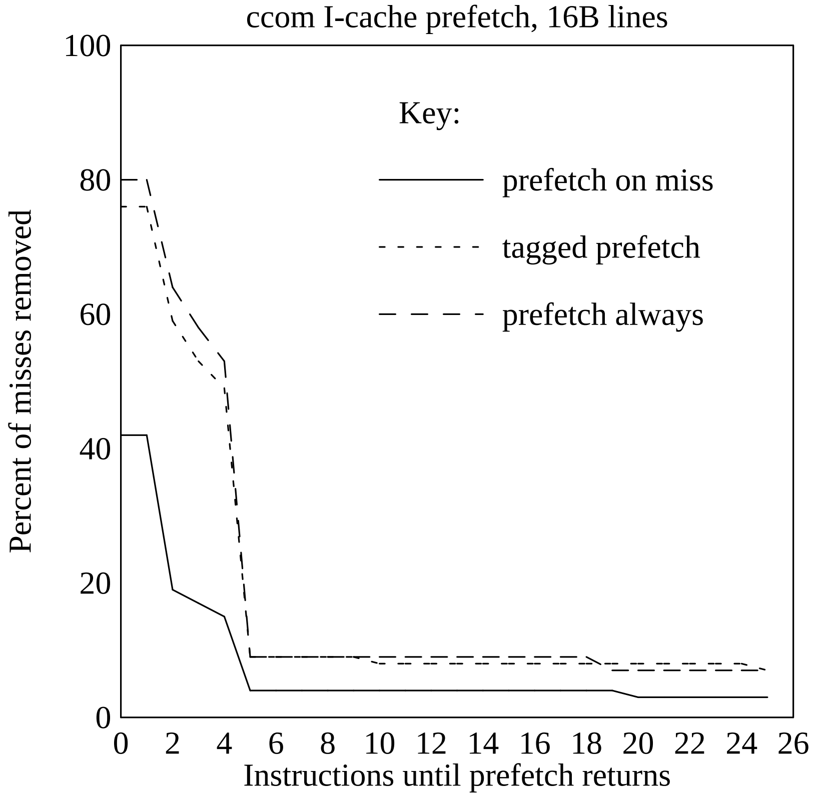

Figure 19: Limited time for prefetch

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 a 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 21 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

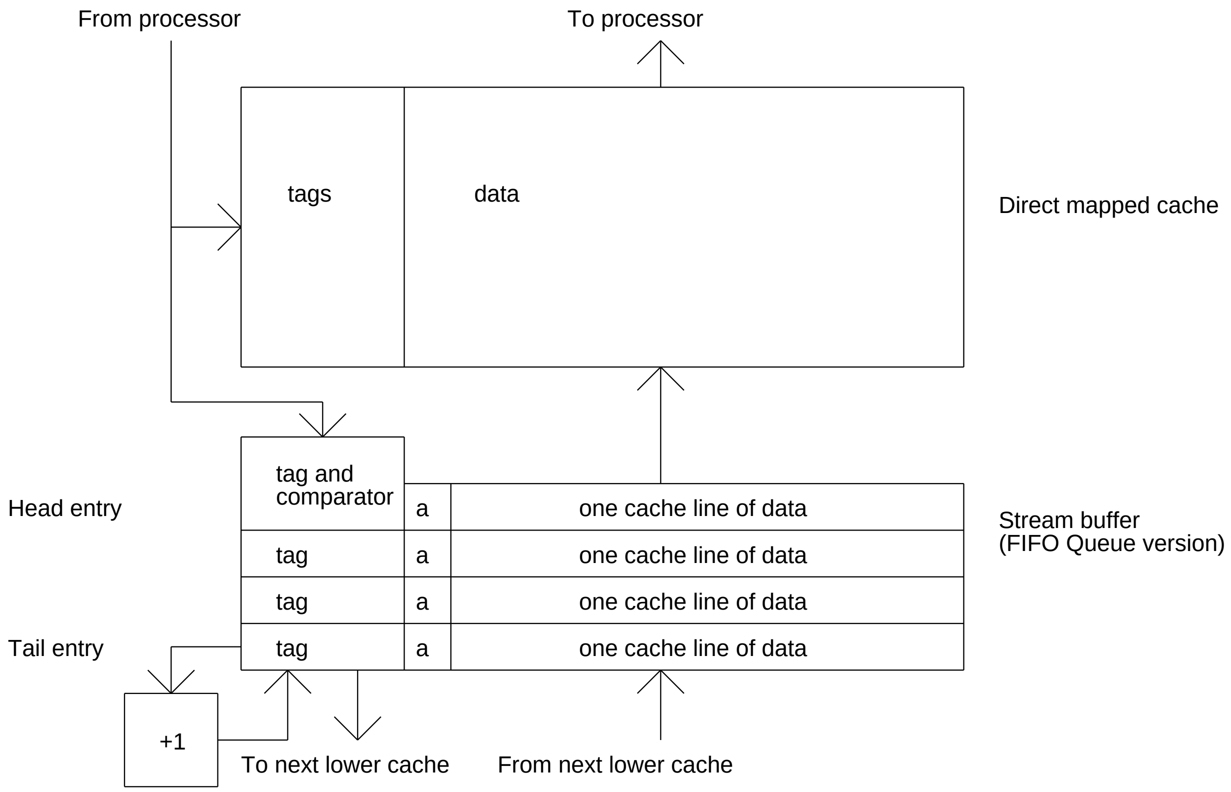

Figure 20: Sequential stream buffer design

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 22 gives the bandwidth requirements in three typical stream buffer applications. I-stream references for *ccom* are quite regular (when measured in instructions). On average a new 16B line must be fetched every 4.2 instructions. The spacing between references to the stream buffer increases when the program enters short loops and decreases when the program takes small forward jumps, such as when skipping an else clause. Nevertheless the fetch frequency is quite regular. This data is for a machine with short functional unit latencies, such as the MIPS R2000 or the MultiTitan CPU, so the CPI is quite close to 1 without cache misses.

Data stream buffer reference timings for *linpack* and *ccom* are also given in Figure 22. The reference rate for new 16B lines for *linpack* averages one every 27 instructions. Since this ver-

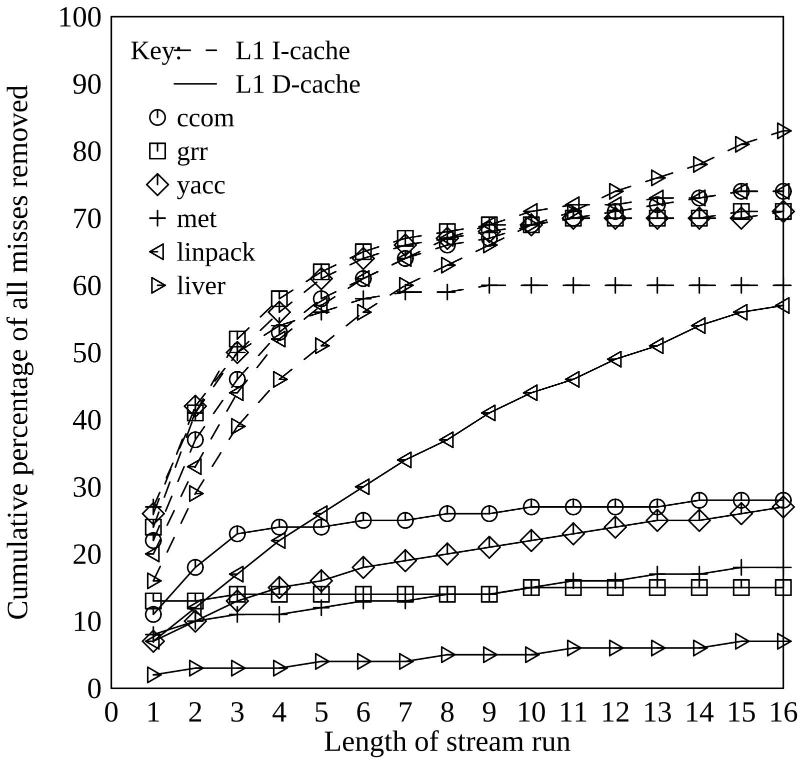

Figure 21: Sequential stream buffer performance

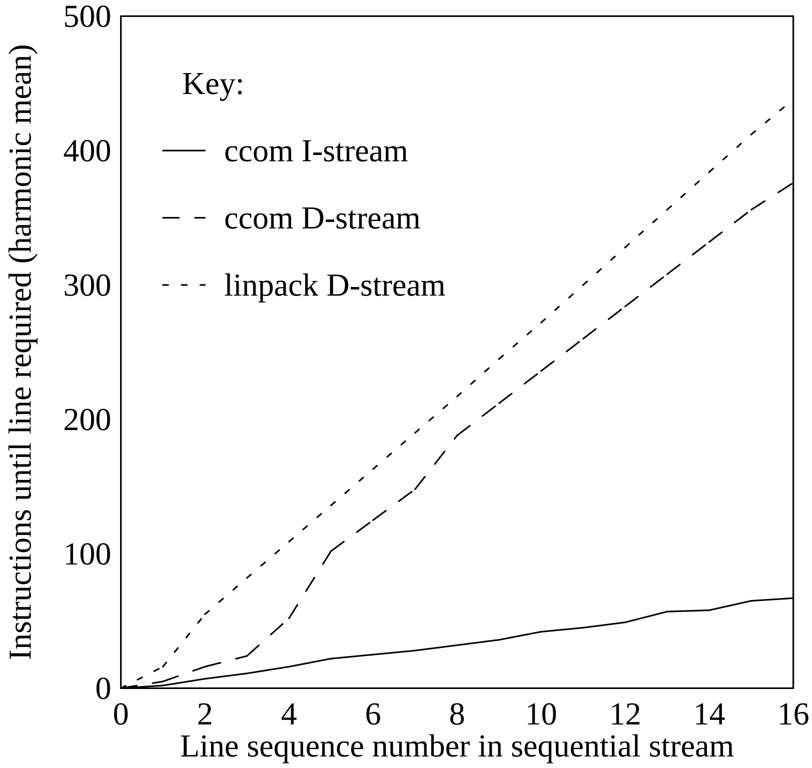

Figure 22: Stream buffer bandwidth requirements

sion of *linpack* is double-precision, this works out to a new iteration of the inner loop every 13.5

instructions. This is larger than one would hope. This version of *linpack* is rather loose in that it does an integer multiply for addressing calculations for each array element, and the loop is not unrolled. If the loop were unrolled and extensive optimizations were performed the rate of references would increase, but the rate should still be less than that of the instruction stream. *ccom* has interesting trimodal performance. If the next successive line is used next after a miss it is required on average only 5 cycles after the miss. For the next two lines after a miss, successive data lines (16B) are required every 10 instructions on average. The first three lines provide most (82%) of the benefit of the stream buffer. After that successive lines are required at a rate closer to that of *linpack*, about every 24 instructions on average.

In general, if the backing store can produce data at an average bandwidth of a new word (4B) every cycle, the stream buffer will be able to keep up with successive references. This should suffice for instruction streams, as well as for block copies that are heavily unrolled and use double-precision loads and stores. If this bandwidth is not available, the benefit of instruction stream buffers will be reduced and block copies and other similar operations will be negatively impacted as well. However, bandwidths equaling a new word every 1.5 to 2 cycles will still suffice for many of the data references. Note that these values are for bandwidths, which are much easier to achieve than total latencies such as required by the prefetch schemes in Figure 19.

4.2.2. 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 23). 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 24 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. Note also that *liver* makes unit stride accesses to its data structures.

4.2.3. Quasi-Sequential Stream Buffers

In the previous section only one address comparator was provided for the stream buffer. This means that even if the requested line was in the stream buffer, but not in the first location with the comparator, the stream buffer will miss on the reference and its contents will be flushed. One obvious improvement to this scheme is to place a comparator at each location in the stream buffer. Then if a cache line is skipped in a quasi-sequential reference pattern, the stream buffer will still be able to supply the cache line if it has already been fetched.

Figure 25 shows the performance of a stream buffer with three comparators. The quasi-stream buffer is able to remove 76% of the instruction-cache misses, an improvement of 4% over a

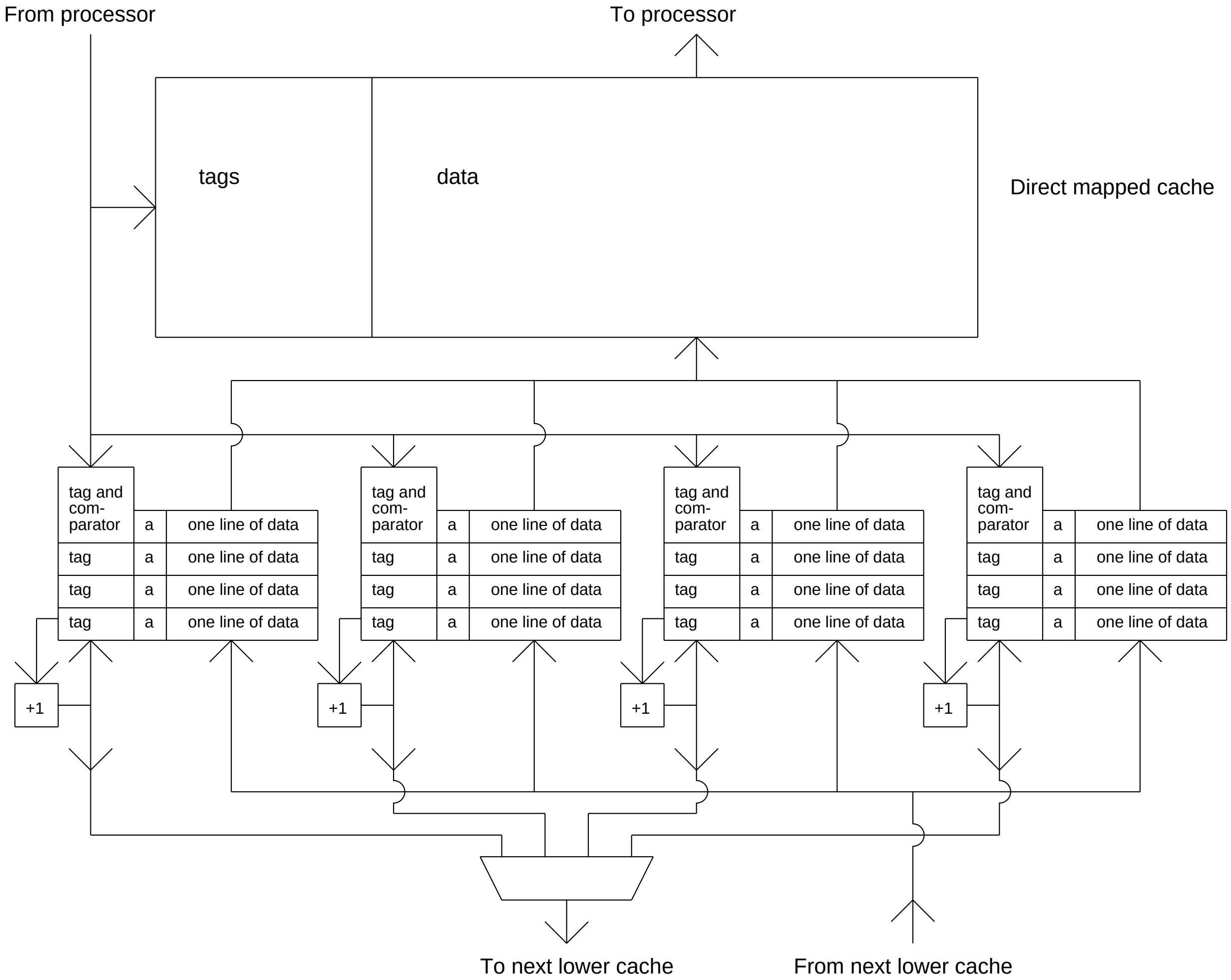

Figure 23: Four-way stream buffer design

purely sequential stream buffer, giving a 14% reduction in the number of misses remaining. This is probably due to the quasi-stream buffer's ability to continue useful fetching when code is skipped, such as when *then* or *else* clauses are skipped in *if* statements. The version simulated had three comparators, so it could skip at most 2 cache lines plus up to 3/4 of a cache line on either side depending on alignment, for a total of 16 to 22 instructions maximum. This compares with only 0 to 6 instructions that may be skipped in a sequential stream buffer (depending on branch alignment) without causing the stream buffer to be flushed.

The extra comparators of a quasi-stream buffer also improve the performance of a four-way data stream buffer. Overall, the four-way quasi-stream buffer can remove 47% of all misses, which is 4% more than the purely sequential four-way stream buffer.

Since the amount of hardware required for a few extra comparators on a single stream buffer is small, quasi-stream buffers seem like a useful generalization of sequential stream buffers for instruction streams. This is because only three additional comparators would be required to convert a four-entry sequential stream buffer into a quasi-stream buffer. However it may not be worthwhile for multi-way data quasi-stream buffers, since the number of extra comparators re-

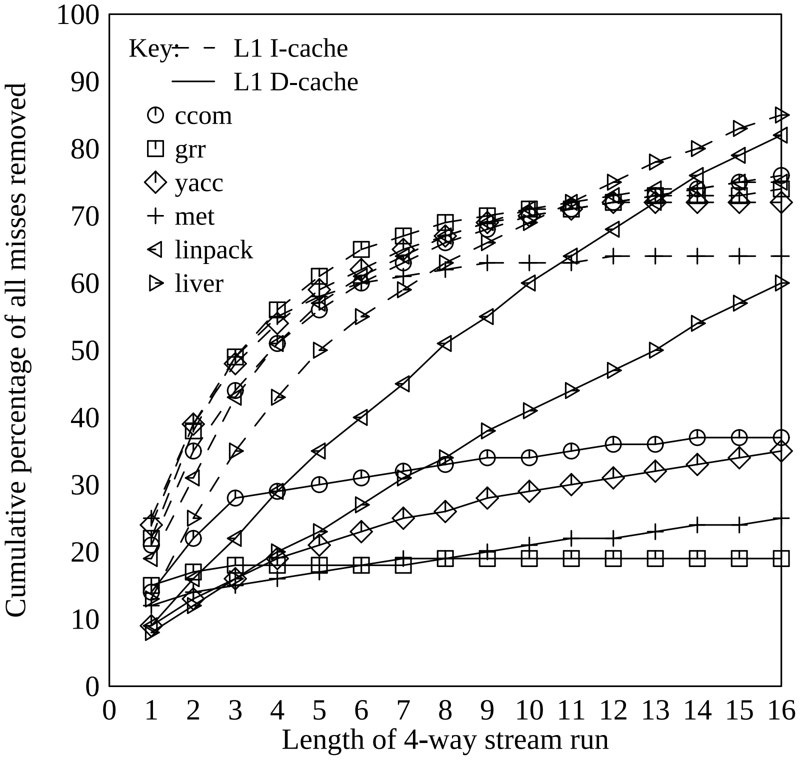

Figure 24: Four-way stream buffer performance

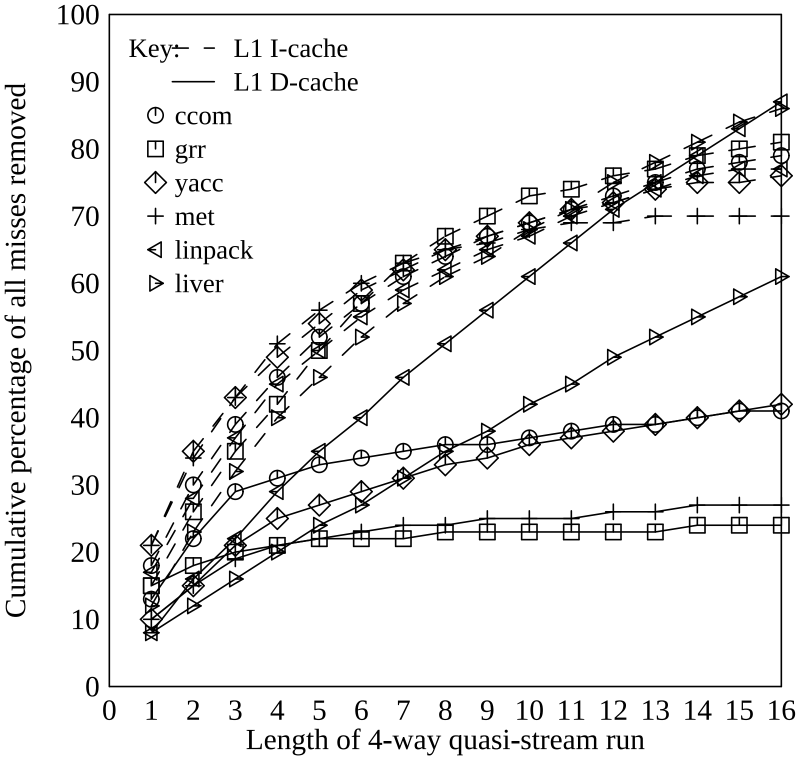

Figure 25: Quasi-sequential 4-way stream buffer performance

quired would be many times as large. An interesting area for software research is the ability of compilers to reorganize code and data layouts to maximize the use of stream buffers. If techniques to optimize sequentiality of references are successful, the need for extra comparators on stream buffers will be lessened.

4.2.4. Stream Buffer Performance vs. Cache Size

Figure 26 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.

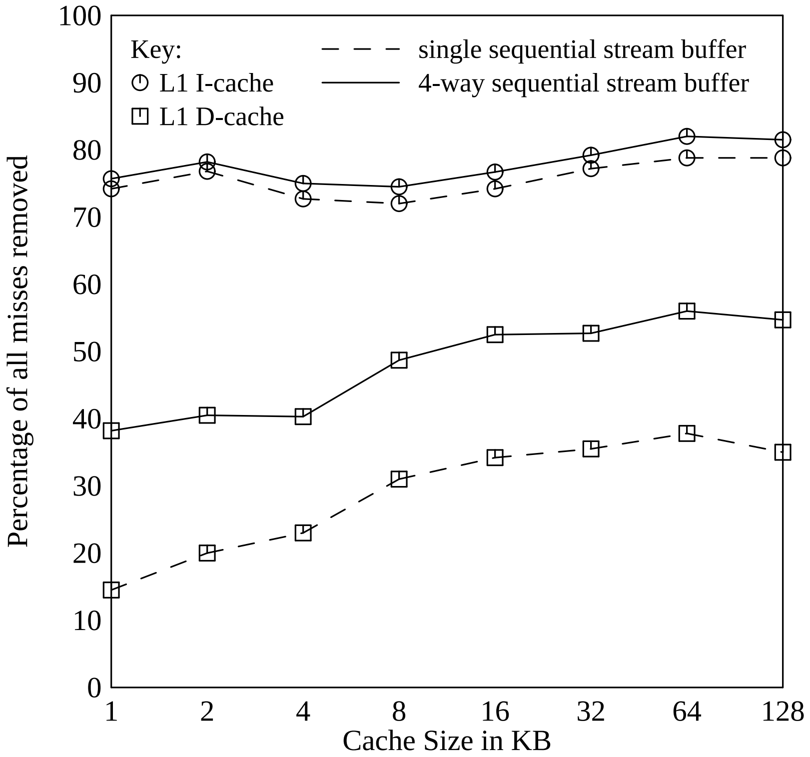

Figure 26: Stream buffer performance vs. cache size

4.2.5. Stream Buffer Performance vs. Line Size

Figure 27 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.

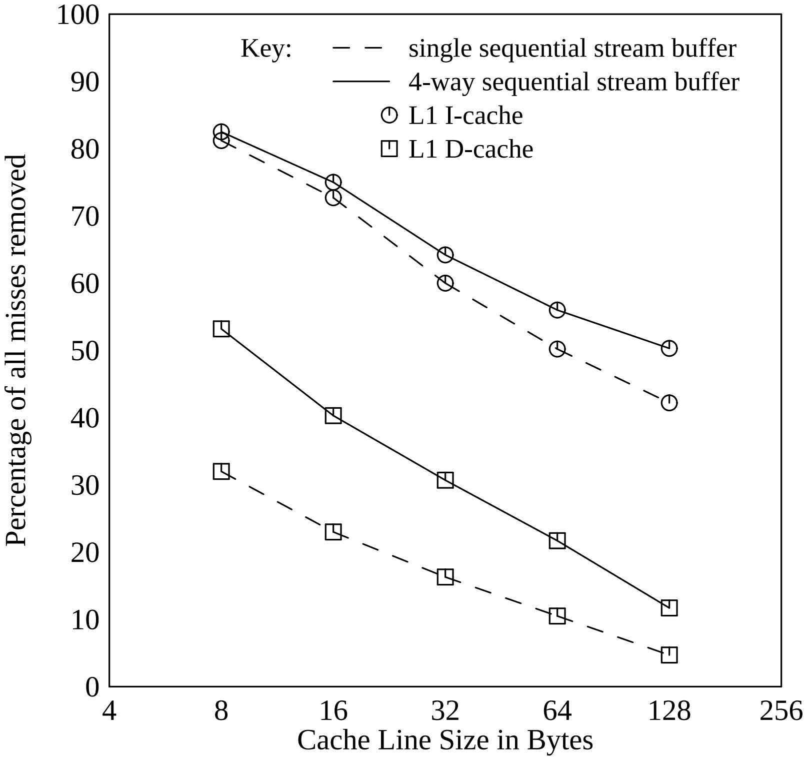

Figure 27: Stream buffer performance vs. line size

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.

4.2.6. Comparison to Classical Prefetch Performance

In order to put the performance of stream buffers in perspective, in this section we compare the performance of stream buffers to some prefetch techniques previously studied in the literature. The performance of prefetch on miss, tagged prefetch, and always prefetch on our six benchmarks is presented in Table 5. This data shows the reduction in misses assuming the use of these prefetch techniques with a second-level cache latency of one instruction-issue. Note that this is quite unrealistic since one-instruction issue latency may be less than a machine cycle, and second-level caches typically have a latency of many CPU cycles. Nevertheless, these figures give an upper bound of the performance of these prefetch techniques. The performance of the prefetch algorithms in this study is consistent with data earlier presented in the literature. In [14] reductions in miss rate for a PDP-11 trace on a 8KB mixed cache (only mixed caches were studied) with 16B lines and 8-way set associativity was found to be 27.8% for prefetch on miss, 50.2% for tagged prefetch, and 51.8% for prefetch always.

Table 6 compares the prefetch performance from Table 5 with the stream buffer performance presented earlier. On the instruction side, a simple single stream buffer outperforms prefetch on

fetch	ccom	yacc	met	grr	liver	linpack	avg
4KB instr. cache, direct-mapped, 16B lines, 1-instr prefetch latency:							
on miss	44.1	42.4	45.2	55.8	47.3	42.8	46.3
tagged	78.6	74.3	65.7	76.1	89.0	77.2	76.8
always	82.0	80.3	62.5	81.8	89.5	84.4	80.1
4KB data cache, direct-mapped, 16B lines, 1-instr prefetch latency:							
on miss	38.2	10.7	14.1	14.5	49.8	75.7	33.8
tagged	39.7	18.0	21.0	14.8	63.1	83.1	40.0
always	39.3	37.2	18.6	11.7	63.1	83.8	42.3

Table 5: Upper bound on prefetch performance: percent reduction in misses

miss by a wide margin. This is not surprising since for a purely sequential reference stream prefetch on miss will only reduce the number of misses by a factor of two. Both the simple single stream buffer and the quasi-stream buffer perform almost as well as tagged prefetch. As far as traffic is concerned, the stream buffer will fetch more after a miss than tagged prefetch, but it will not start fetching on a tag transition, so a comparison of traffic ratios would be interesting future research. The performance of the stream buffers on the instruction stream is slightly less than prefetch always. This is not surprising, since the performance of always prefetch approximates the percentage of instructions that are not taken branches, and is an upper bound on the reduction of instruction cache misses by sequential prefetching. However, the traffic ratio of the stream buffer approaches should be much closer to that of prefetch on miss or tagged prefetch than to prefetch always.

technique	misses eliminated
for 4KB direct-mapped instruction cache w/ 16B lines:	
prefetch on miss (1-instr latency)	46.3
single stream buffer	72.0
quasi-stream buffer (3 comparator)	76.0
tagged prefetch (1-instr latency)	76.8
always prefetch (1-instr latency)	80.1
for 4KB direct-mapped data cache w/ 16B lines:	
single stream buffer	25.0
prefetch on miss (1-instr latency)	33.8
tagged prefetch (1-instr latency)	40.0
always prefetch (1-instr latency)	42.3
4-way stream buffer	43.0
4-way quasi-stream buffer	47.0

Table 6: Upper bound of prefetch performance vs. stream buffer performance

Table 6 also compares the performance of stream buffers to other prefetch techniques for data references. Here both types of 4-way stream buffers outperform the other prefetch strategies. This is primarily because the prefetch strategies always put the prefetched item in the cache, even if it is not needed. The stream buffer approaches only move an item into to the cache if it is requested, resulting in less pollution than always placing the prefetched data in the cache. This is especially important for data references since the spatial locality of data references is less than that of instruction references, and prefetched data is more likely to be pollution than are prefetched instructions.

Independent of the relative performance of stream buffers and ideal prefetch techniques, the stream buffer approaches are much more feasible to implement. This is because they can take advantage of pipelined memory systems (unlike prefetch on miss or tagged prefetch for sequential reference patterns). They also have lower latency requirements on prefetched data than the other prefetching techniques, since they can start fetching a block before the previous block is used. Finally, at least for instruction stream buffers, the extra hardware required by a stream buffer is often comparable to the additional tag storage required by tagged prefetch.

4.3. Combining Long Lines and Stream Buffers

Long cache lines and stream buffers can be used advantageously together, since the strengths and weaknesses of long lines and stream buffers are complimentary. For example, long lines fetch data that, even if not used immediately, will be around for later use. However, the other side of this advantage is that excessively long lines can pollute a cache. On the other hand, stream buffers do not unnecessarily pollute a cache since they only enter data when it is requested on a miss. However, at least one reference to successive data must be made relatively soon, otherwise it will pass out of the stream buffer without being used.

Table 7 gives the performance of various long-line and stream-buffer alternatives for a 4KB instruction cache. The first thing to notice is that all the stream buffer approaches, independent of their line size, outperform all of the longer line size approaches. In fact, the stream buffer approaches outperform a hypothetical machine with a line size that can be set to the best value for each benchmark. The fact that the stream buffers are doing better than this shows that they are actually providing an effective line size that varies on a per reference basis within each program. Also note that the line size used in the stream buffer approaches is not that significant, although it is very significant if a stream buffer is not used. Finally, the quasi-stream buffer capability approximates the performance of purely sequential stream buffers with longer line sizes. Consider for example a quasi-stream buffer than can skip two 16B lines. It will have a "prefetch reach" of between 16 and 22 four-byte instructions depending on alignment. This is a little longer span than a sequential 32B line stream buffer (8 to 15 instructions depending on alignment) and a little shorter than a sequential 64B line stream buffer (16 to 31 instructions). Thus it is not surprising that the performance of the 16B three-comparator quasi-stream buffer is between that of a 32B and a 64B line sequential stream buffer. Given that it is usually easier to make the cache line size equal to the transfer size, and that transfer sizes larger than 16B seem unlikely in the near future (at least for microprocessor-based machines), it seems that the use of quasi-sequential stream buffers with smaller line sizes such as 16B would be the most promising approach for the instruction cache. In particular if a quasi-sequential stream buffer is used, line sizes of greater than 32B have little benefit for 4KB instruction caches.

Table 8 gives the results for data stream buffers in comparison with longer line sizes, assuming there is no miss cache. Here the superiority of stream buffers over longer data cache line sizes is much more pronounced than with long instruction cache lines. For example, a four-way quasi-sequential data stream buffer can eliminate twice as many misses as the optimal line size per program, in comparison to only about 14% better performance for an instruction stream buff-

instr cache configuration (default does not include a miss cache)	misses eliminated
32B lines	38.0%
64B lines	55.4%
128B lines	69.7%
optimal line size per program	70.0%
16B lines w/ single stream buffer	72.0%
32B lines w/ single stream buffer	75.2%
16B lines w/ quasi-stream buffer	76.0%
64B lines w/ single stream buffer	77.6%
32B lines w/ quasi-stream buffer	80.0%
64B lines w/ quasi-stream buffer	80.2%

Table 7: Improvements relative to a 16B instruction line size without miss caching

er over an optimal per-program instruction cache line size. This is due to the wider range of localities present in data references. For example, some data reference patterns consist of references that are widely separated from previous data references (e.g., manipulation of complex linked data structures), while other reference patterns are sequential for long distances (e.g., unit stride array manipulation). Different instruction reference streams are quite similar by comparison. Thus it is not surprising that the ability of stream buffers to provide an effective line size that varies on a reference-by-reference basis is more important for data caches than for instruction caches.

data cache configuration (default does not include a miss cache)	misses eliminated
64B lines	0.5%
32B lines	1.0%
optimal line size per program	19.2%
16B lines w/ single stream buffer	25.0%
16B lines w/ 4-way stream buffer	43.0%
16B lines w/ 4-way quasi-stream buffer	47.0%

Table 8: Improvements relative to a 16B data line size without miss caching

Table 9 presents results assuming that longer data cache line sizes are used in conjunction with a four-entry miss cache. The addition of a miss cache improves the performance of the longer data cache line sizes, but they still underperform the stream buffers. This is still true even for a system with a different line size per program.

One obvious way to combine longer lines and stream buffers is to increase the line size up to the smallest line size that gives a minimum miss rate for some program. In our previous examples with a four-line miss cache this is a 32B line since this provides a minimum miss rate for *met*. Then stream buffers can be used to effectively provide what amounts to a variable line size extension. With 32B lines and a stream buffer a 68.6% further decrease in misses can be obtained. This does in fact yield the configuration with the best performance. Further increasing

data cache configuration (default includes 4-entry miss cache)	misses eliminated
32B lines	24.0%
16B lines w/ single stream buffer	25.0%
64B lines	31.0%
optimal line size per program	38.0%
16B lines w/ 4-way stream buffer	43.0%
16B lines w/ 4-way quasi-stream buffer	47.0%
64B lines w/ 4-way quasi-stream buffer	48.7%
32B lines w/ 4-way quasi-stream buffer	52.1%

Table 9: Improvements relative to a 16B data line size and 4-entry miss cache

the line size to 64B with a stream buffer is ineffective even though it reduces the average number of misses in configurations without a stream buffer. This is because the stream buffer will provide the same effect as longer cache lines for those references that need it, but will not have the extra conflict misses associated with longer cache line sizes.

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. In this study the starting prefetch address is replaced over all stream buffers in LRU order. 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 sequential 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.

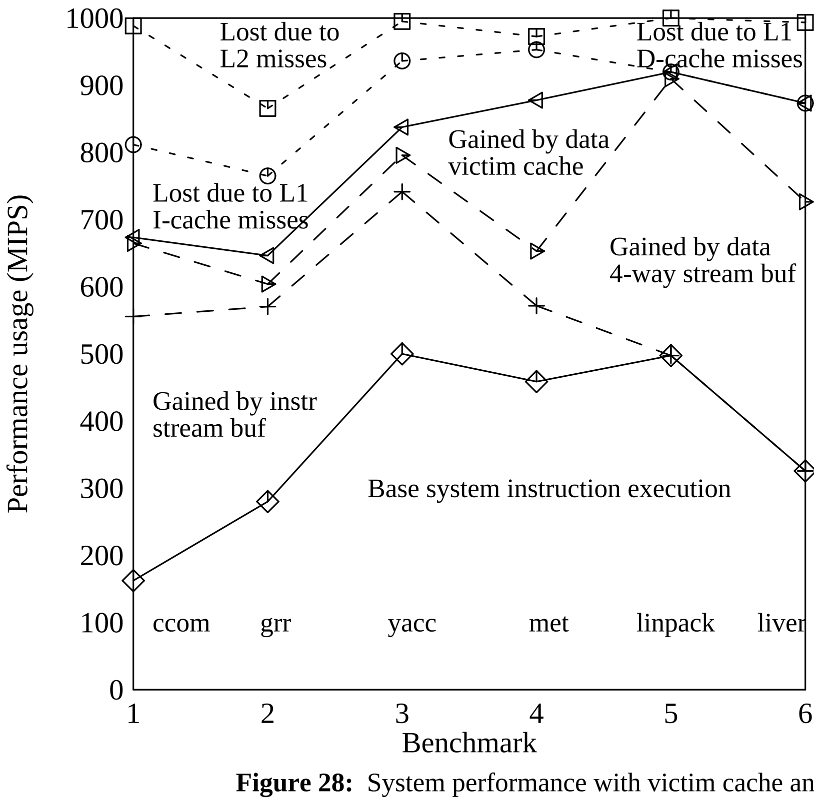

Figure 28: System performance with victim cache and stream buffers

Figure 28 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 28 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.

One way of looking at the performance of victim caching and stream buffers is to consider the effective increase in cache size provided by these techniques. Table 10 gives the increase in cache size required to give the same instruction miss rate as a smaller cache plus a stream buffer. It is possible that by adding a stream buffer the compulsory misses are reduced to an extent that reduces the overall miss rate to a rate lower than that achieved by any cache with a 16B line size. Asterisks in Table 10 denotes situations where this occurs, or at least the miss rate is reduced beyond that of a 128KB cache, the largest size simulated. *ccom* has a particularly bad instruction cache miss rate, and it has a very large working set, so it benefits the most from instruction stream buffering.

program	multiple increase in effective cache size						
name	1K	2K	4K	8K	16K	32K	64K
ccom	26.3X	16.1X	7.0X	6.1X	4.1X	3.5X	*
grr	6.0X	3.5X	4.3X	3.4X	1.8X	2.7X	1.7X
yacc	7.5X	4.1X	3.0X	2.8X	1.9X	1.7X	*
met	3.2X	1.8X	2.1X	2.9X	1.9X	3.0X	1.9X
linpack	1.7X	1.9X	3.6X	*	*	*	*
liver	4.0X	2.0X	*	*	*	*	*

*** denotes no cache size below 256KB attains as low a miss rate as cache with streambuffer**

Table 10: Effective increase in instruction cache size provided by streambuffer with 16B lines

Corresponding equivalent increases in effective data cache size provided by the addition of a 4-entry victim cache and a 4-way stream buffer are given in Table 11. *linpack* and *liver* sequentially access very large arrays from one end to the other before returning. Thus they have very large effective cache size increases since with stream buffering they have equivalent cache sizes equal to their array sizes. (This assumes the stream buffer can keep up with their data consumption, which is true for our baseline system parameters.)

program	multiple increase in effective cache size						
name	1K	2K	4K	8K	16K	32K	64K
ccom	6.3X	5.0X	3.9X	3.1X	2.3X	1.8X	1.8X
grr	1.6X	1.5X	1.4X	1.2X	3.8X	*	*
yacc	1.6X	2.5X	1.7X	1.6X	1.7X	2.1X	*
met	1.4X	3.3X	1.2X	1.6X	3.3X	1.8X	*
linpack	98.3X	53.6X	30.4X	15.8X	*	*	*
liver	26.0X	16.0X	9.5X	8.4X	6.3X	3.4X	1.9X

*** denotes no cache size below 256KB attains as low a miss rate as cache with 4-way streambuffer and 4-entry victim cache**

Table 11: Effective increase in data cache size provided with stream buffers and victim caches using 16B lines

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

Mary Jo Doherty, John Ousterhout, Jeremy Dion, Anita Borg, and Richard Swan provided many helpful comments on an early draft of this paper. Alan Eustace suggested victim caching as an improvement to miss caching.

References

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

[2] Borg, Anita, Kessler, Rick E., Lazana, Georgia, and Wall, David W.
Long Address Traces from RISC Machines: Generation and Analysis.
Technical Report 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.
In *The 11th Annual Symposium on Computer Architecture*, pages 301-310. IEEE Computer Society Press, June, 1984.

[5] Eustace, Alan.
Private communication.
February, 1989.

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

[7] Hill, Mark D.
Aspects of Cache Memory and Instruction Buffer Performance.
PhD thesis, University of California, Berkeley, 1987.

[8] International Business Machines, Inc.
IBM3033 Processor Complex, Theory of Operation/Diagrams Manual - Processor Storage Control Function (PSCF)
Poughkeepsie, N.Y., 1982.

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

[10] Jouppi, Norman P.
Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU.
In *The 16th Annual Symposium on Computer Architecture*, pages 281-289. IEEE Computer Society Press, May, 1989.

[11] Nielsen, Michael J. K.
Titan System Manual.
Technical Report 86/1, Digital Equipment Corporation Western Research Laboratory, September, 1986.

[12] Ousterhout, John.
Why Aren't Operating Systems Getting Faster As Fast As Hardware?.
Technical Report Technote 11, Digital Equipment Corporation Western Research Laboratory, October, 1989.

[13] Smith, Alan J.
Sequential program prefetching in memory hierarchies.
IEEE Computer 11(12):7-21, December, 1978.

[14] Smith, Alan J.
Cache Memories.
Computing Surveys :473-530, September, 1982.

Table of Contents

List of Figures

List of Tables