AUGUST 1990

WRL
Technical Note TN-53

Reducing Compulsory and Capacity Misses

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 are two other research laboratories located in Palo Alto, the Network Systems Lab (NSL) and the Systems Research Center (SRC). Another Digital research group is located 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 technical 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.

Reports and technical notes may also be accessed via the World Wide Web: `http://www.research.digital.com/wrl/home.html`.

Reducing Compulsory and Capacity Misses

Norman P. Jouppi

August , 1990

Abstract

This paper investigates several methods for reducing cache miss rates. Longer cache lines can be advantageously used to decrease cache miss rates when used in conjunction with miss caches. Prefetch techniques can also be used to reduce cache miss rates. However, stream buffers are better than either of these two approaches. They are shown to have lower miss rates than an optimal line size for each program, and have better or near equal performance to traditional prefetch techniques even when single instruction-issue latency is assumed for prefetches. Stream buffers in conjunction with victim caches can often provide a reduction in miss rate equivalent to a doubling or quadupling of cache size. In some cases the reduction in miss rate provided by stream buffers and victim caches is larger than that of any size cache. Finally, the potential for compiler optimizations to increase the performance of stream buffers is investigated.

This tech note is a copy of a paper that was submitted to but did not appear in ASPLOS-4. It has not been changed since its submission or submitted for publication in any other forum.

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

Table of Contents

List of Figures

List of Tables

Reducing Compulsory and Capacity Cache Misses

Norman P. Jouppi
Digital Equipment Corporation
Western Research Laboratory
100 Hamilton Avenue
Palo Alto, CA 94301
(415)-853-6617

August 6, 1990

Abstract

This paper investigates several methods for reducing cache miss rates. Longer cache lines can be advantageously used to decrease cache miss rates when used in conjunction with miss caches. Prefetch techniques can also be used to reduce cache miss rates. However, stream buffers are better than either of these two approaches. They are shown to have lower miss rates than an optimal line size for each program, and have better or near equal performance to traditional prefetch techniques even when single instruction-issue latency is assumed for prefetches. Stream buffers in conjunction with victim caches can often provide a reduction in miss rate equivalent to a doubling or quadupling of cache size. In some cases the reduction in miss rate provided by stream buffers and victim caches is larger than that of any size cache. Finally, the potential for compiler optimizations to increase the performance of stream buffers is investigated.

Keywords

Cache memories, prefetch, stream buffer, block size, line size.

1. Introduction

Cache misses can be classified into four categories: conflict, compulsory, capacity [3], 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.

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 [9, 1]. 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 paper we investigate techniques to reduce capacity and compulsory misses while mitigating traditional problems with long lines and excessive prefetching.

The characteristics of the test programs used in this study are given in Table 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 default cache parameters are 4KB direct-mapped instruction and 4KB direct-mapped data caches each with 16B lines unless specified otherwise. A large off-chip second-level cache is implicitly assumed, however second-level cache performance is beyond the scope of this paper.

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 1: Test program characteristics

2. Reducing Capacity and Compulsory Misses with Long Lines

One way to reduce the number of capacity and compulsory misses is to choose a line size that maximizes processor performance [10, 5]. If conflict misses did not exist, caches with larger line sizes would be appropriate, even after accounting for transfer costs. Figure 1 shows the reduction in compulsory and capacity misses with increasing line size, compared to a baseline design with 8B lines. (The other cache parameters are the default: 4KB size for both instruction and data and direct-mapping.) 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.

However, when the effects of conflict misses are included, the picture changes dramatically (see Figure 2). 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

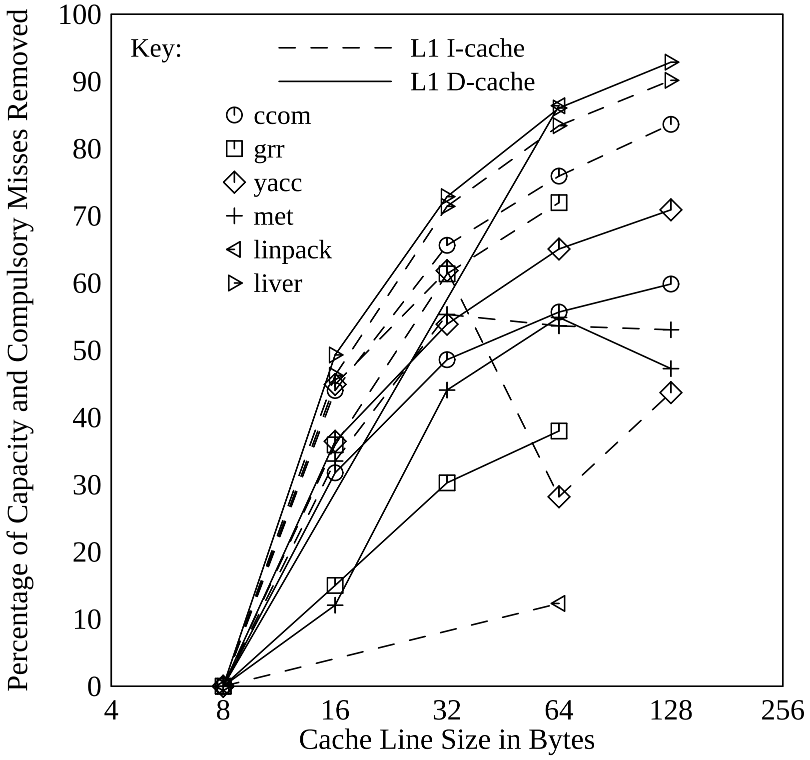

Figure 1: Effect of increasing line size on capacity and compulsory misses

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.

Although curves of average performance such as Figure 2 appear to roll off fairly smoothly, the performance for individual programs can be quite different. Figure 3 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 1 increases fairly monotonically with increasing line size, we know the steep drops in performance in Figure 3 are due to increasing numbers of conflict misses. Since miss caches [4] tend to remove a higher percentage of conflict misses when conflicts are frequent, miss caches should allow us to take better advantage of longer cache line sizes. Figure 4 shows the average effectiveness over all the benchmarks of increasing line size in con-

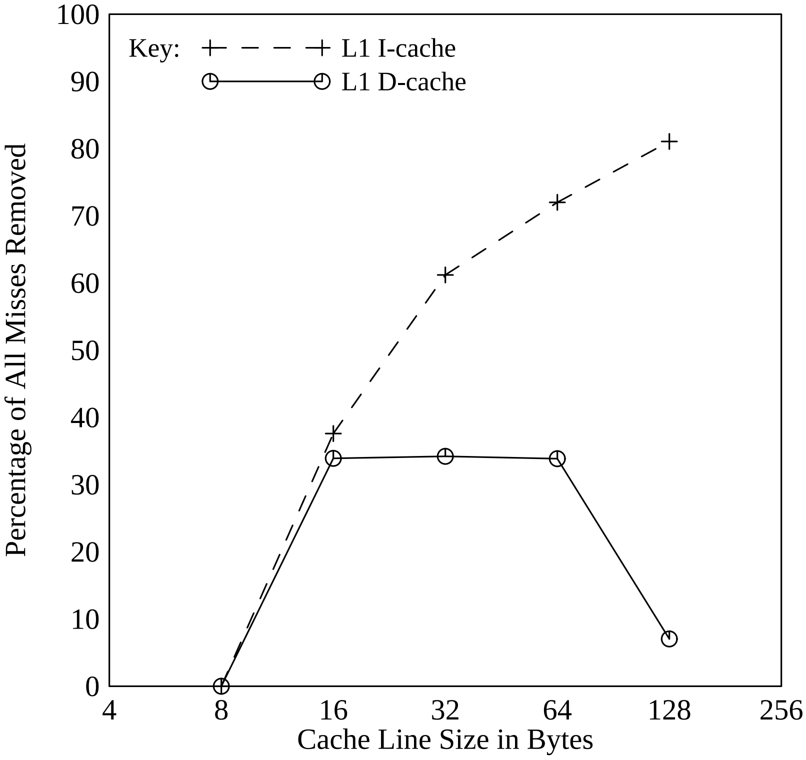

Figure 2: Effect of increasing line size on overall miss rate

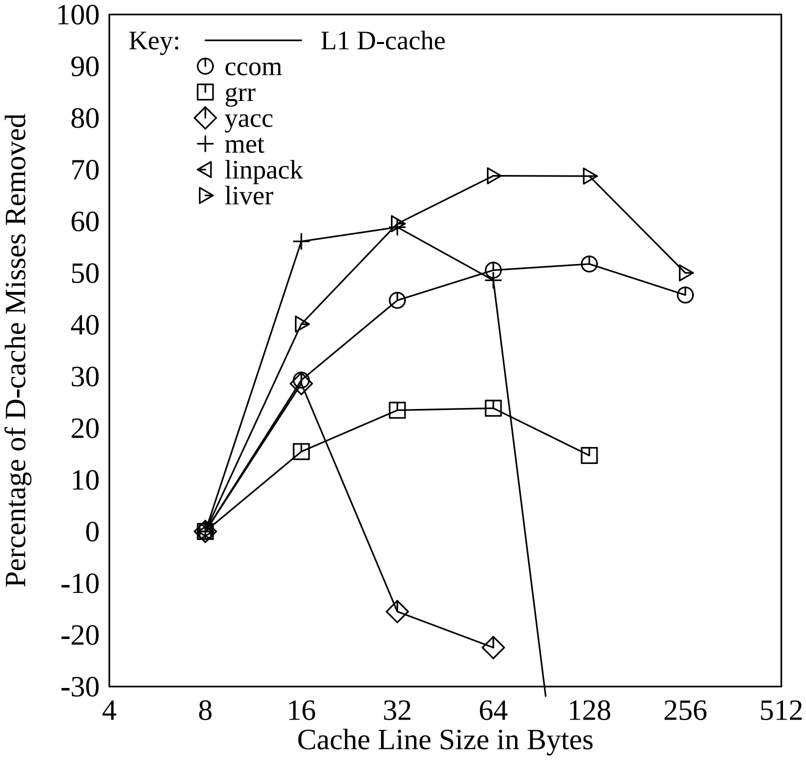

Figure 3: Effect of increasing data cache line size on each benchmark

figurations 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 2 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.

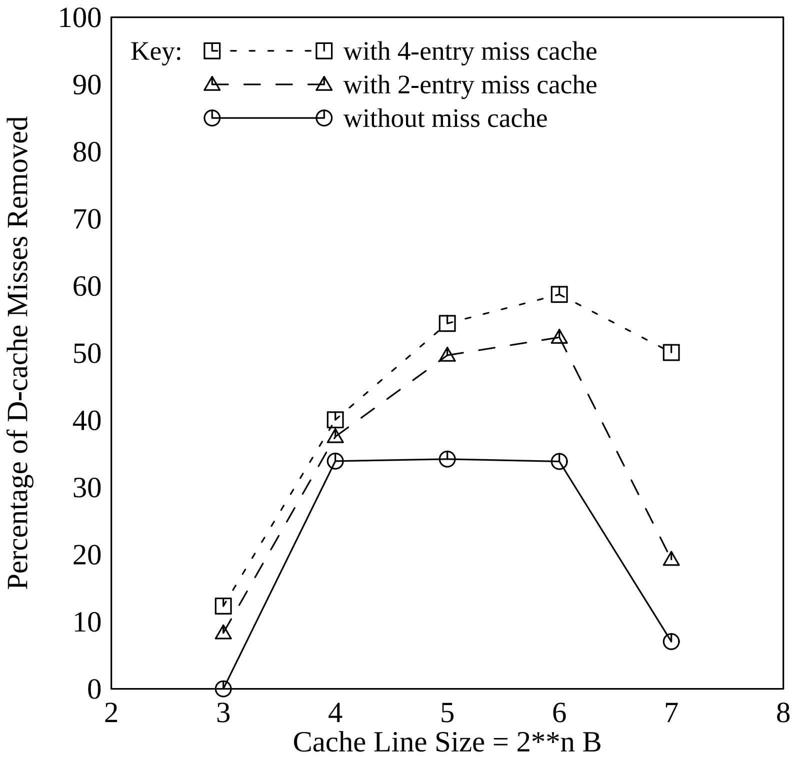

Figure 4: Effect of increasing data cache line size with miss caches

miss cache entries	line size with minimum miss rate: 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 2: Line sizes with minimum miss rates by program

Systems with miss caching continue to obtain benefits from longer line sizes where systems without miss caches have flat or decreasing performance. Figure 5 shows the detailed behavior of most of the programs. Figure 6 shows the effects of longer cache line sizes on *yacc* and *met* with varying miss cache sizes. The performance of *yacc* is affected most dramatically - the sharp drop at line sizes above 16B is completely eliminated even with miss caches with as few as two entries. Similarly, adding a miss cache with four entries can turn a 100% increase in miss rate for *met* at 128B lines into only a 14% 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 4 diverge at a line size of 128B.

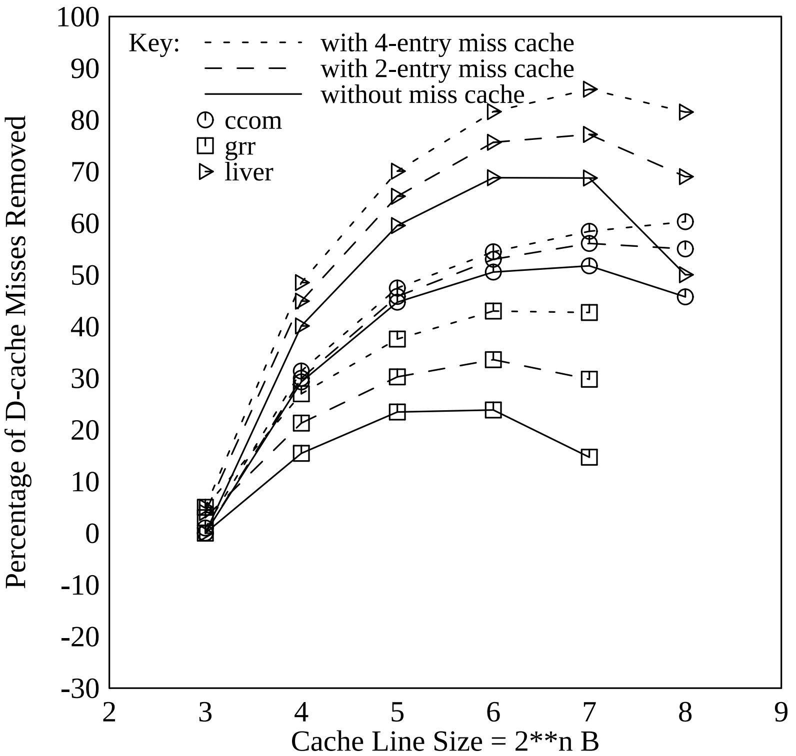

Figure 5: Benchmark-specific performance with increasing data cache line size

Miss caches for very large lines or with more than four entries at moderate line sizes were not simulated. Besides prohibitive transfer costs, as line sizes become larger the amount of storage required by the miss cache increases dramatically. For example, with a 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 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.

3. 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 [8, 2, 6, 7] 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 [9]. *Prefetch always* prefetches after every reference. Needless to say this is impractical in most systems since many level-one cache accesses can take place in the time required to initiate a single level-two cache or

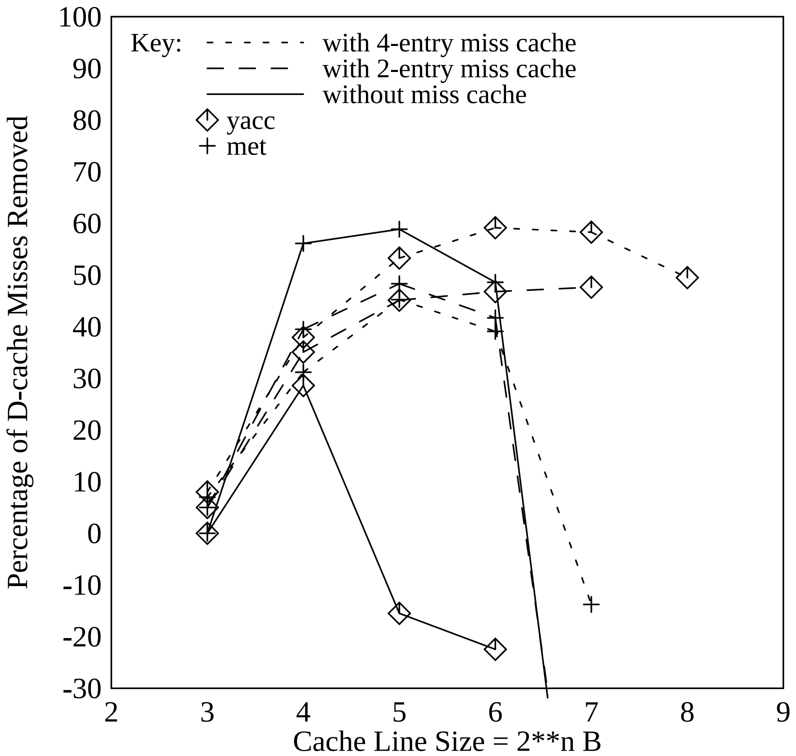

Figure 6: *yacc* and *met* performance with increasing data cache line size

main memory 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. Clearly *prefetch always* provides an upper bound on prefetch performance. *Prefetch on miss* and *tagged prefetch* are more practical 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 7, 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.

3.1. 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* [4] (Figure 8). A stream buffer consists of a series of entries, each consisting of a tag, an available bit, and a data line.

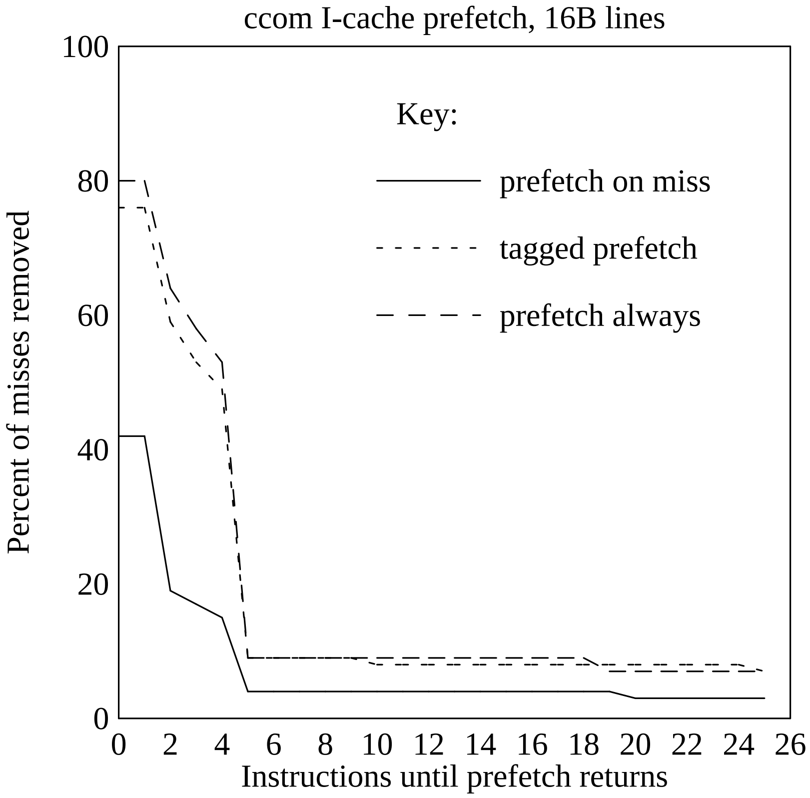

Figure 7: Limited time for prefetch

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 [4] 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.

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).

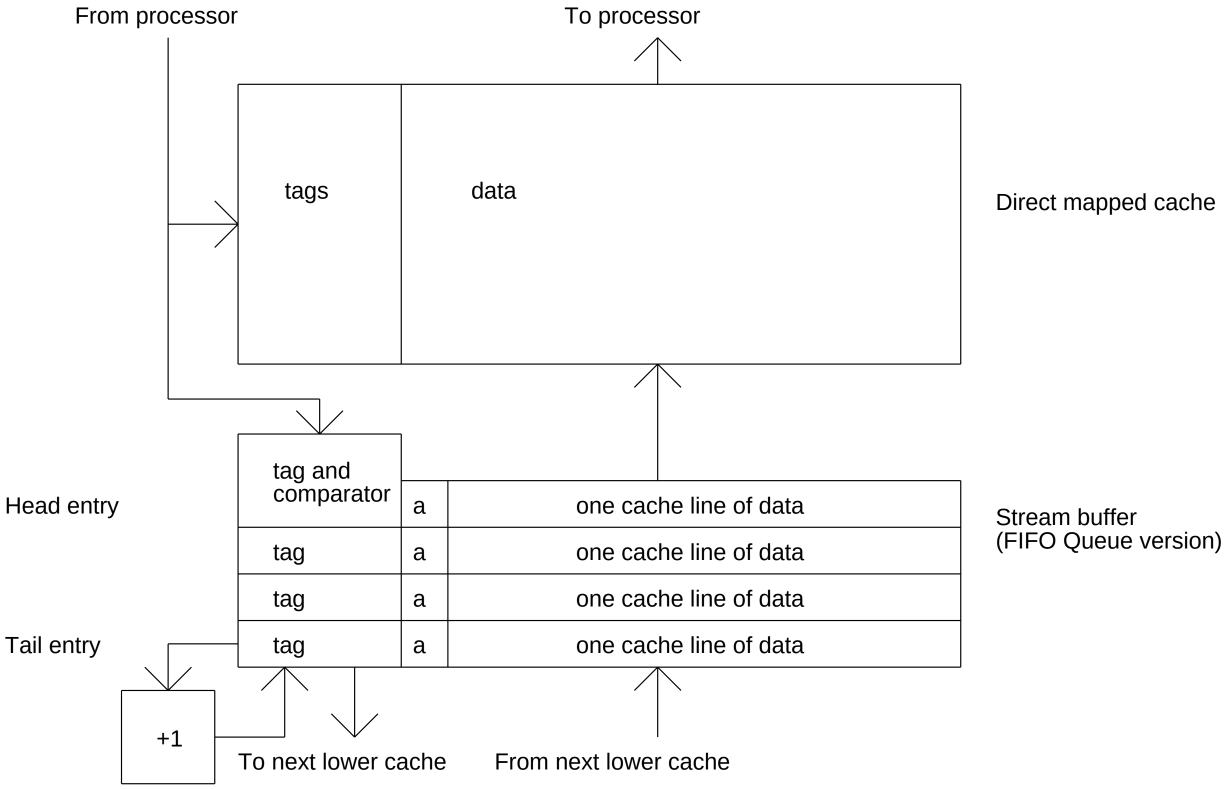

Figure 8: Sequential stream buffer design

Figure 9 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.)

3.1.1. Stream Buffer Bandwidth Requirements

Figure 10 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.

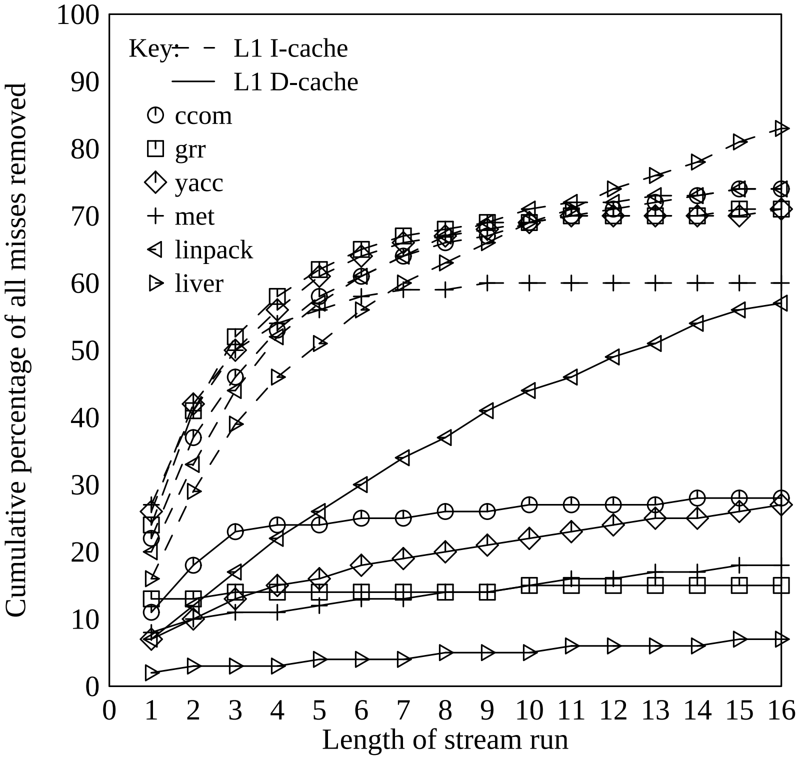

Figure 9: Sequential stream buffer performance

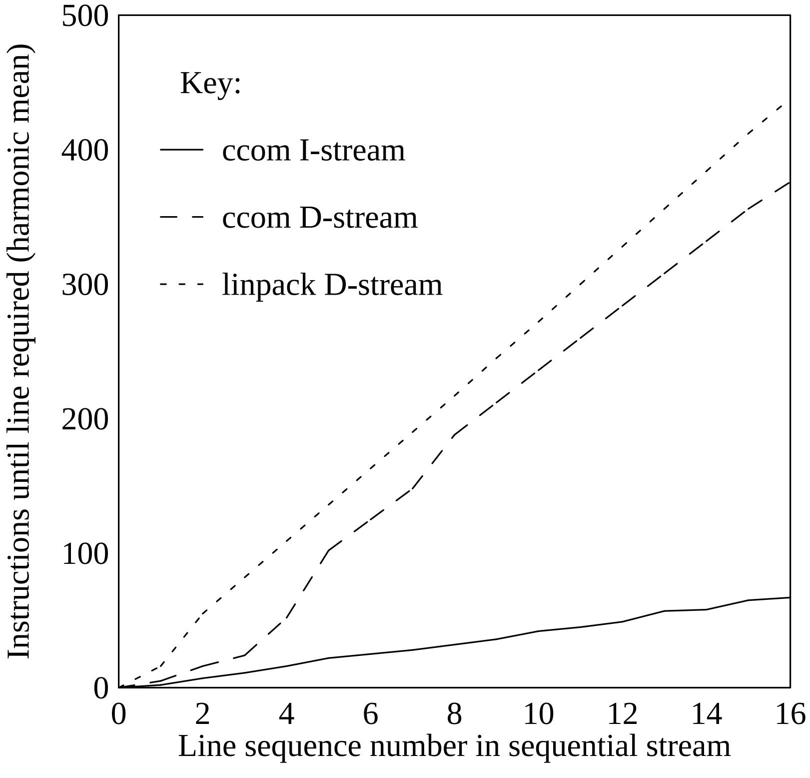

Figure 10: Stream buffer bandwidth requirements

Data stream buffer reference timings for *linpack* and *ccom* are also given in Figure 10. The reference rate for new 16B lines for *linpack* averages one every 27 instructions. Since this version 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. 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 7.

3.1.2. 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 12 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 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. (A multi-way stream buffer consists of several 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 13 shows the performance of a 4-way quasi-stream buffer with three comparators. Overall, a four-way quasi-stream buffer can remove 47% of all misses, which is 4% more than the purely sequential four-way stream buffer.

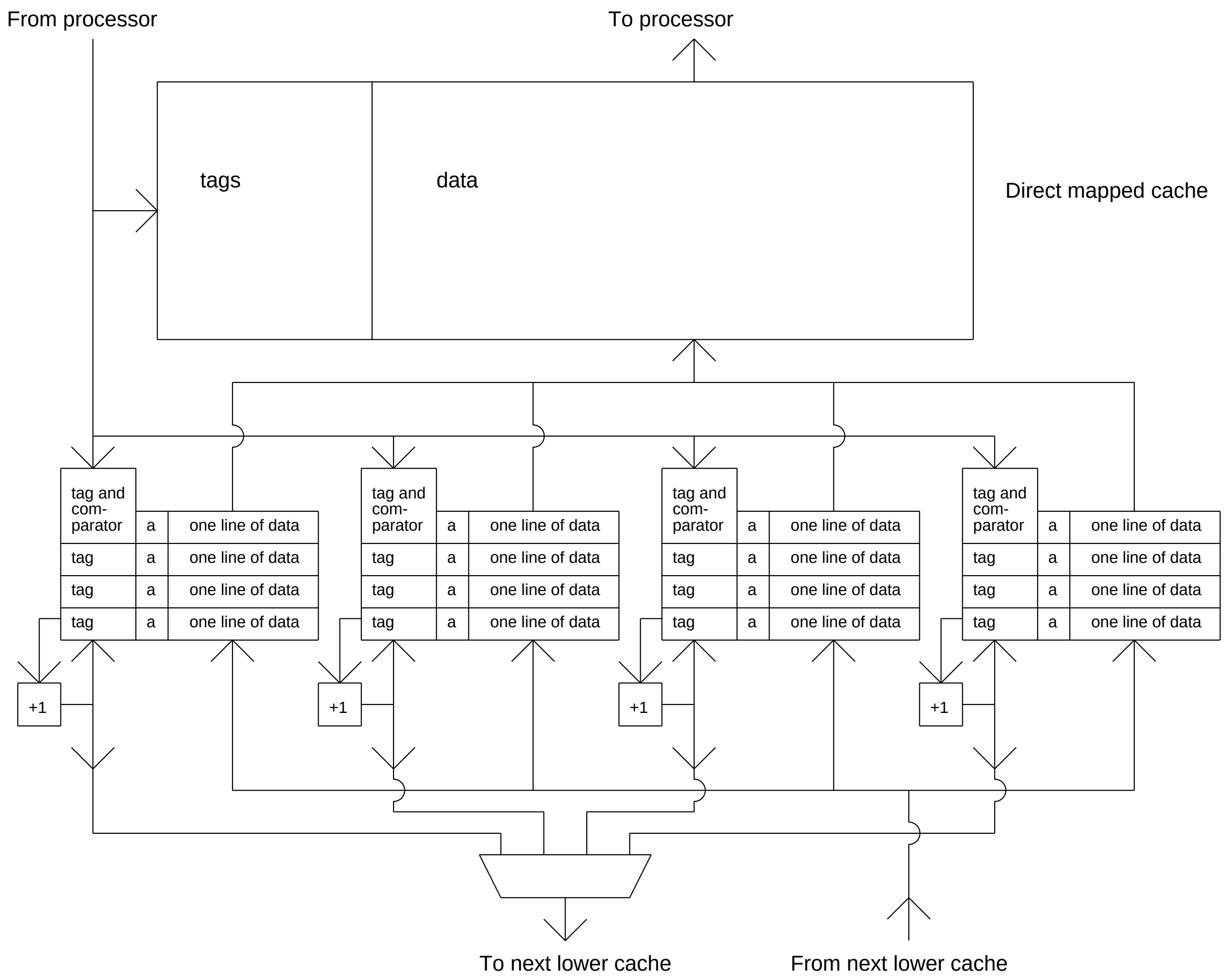

Figure 11: Four-way stream buffer design

In reality, the extra comparators required to convert a data stream buffer into a quasi-stream buffer are usually required anyway to maintain data consistency. Otherwise stores that hit in the data cache and also match an item in the stream buffer will not update or invalidate the stream buffer entry. If the cache line written by the store is replaced by another line and then read from the stream buffer, the old data will be read.

If quasi-stream buffer operation is allowed, the bandwidth required by the stream buffer will increase. Since lines can be skipped over in a quasi-stream buffer, the bandwidth requirements of a quasi-stream buffer are potentially as many times larger than a sequential stream buffer as the ratio in the number of tag comparators between them. However, to the extent that quasi-stream buffers show little marginal improvement over sequential stream buffers, the actual bandwidth increase required should be small.

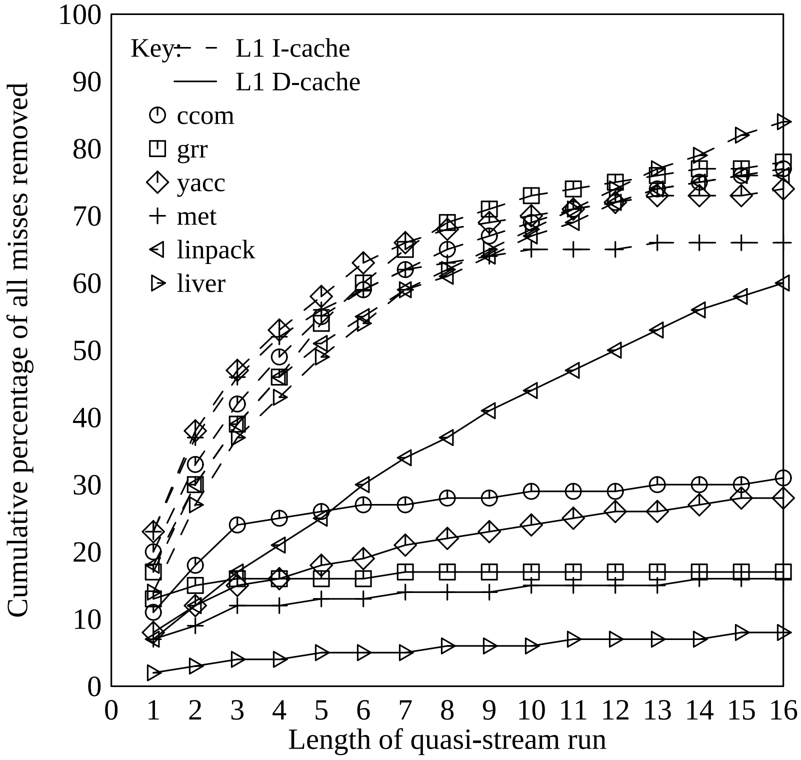

Figure 12: Quasi-sequential stream buffer performance

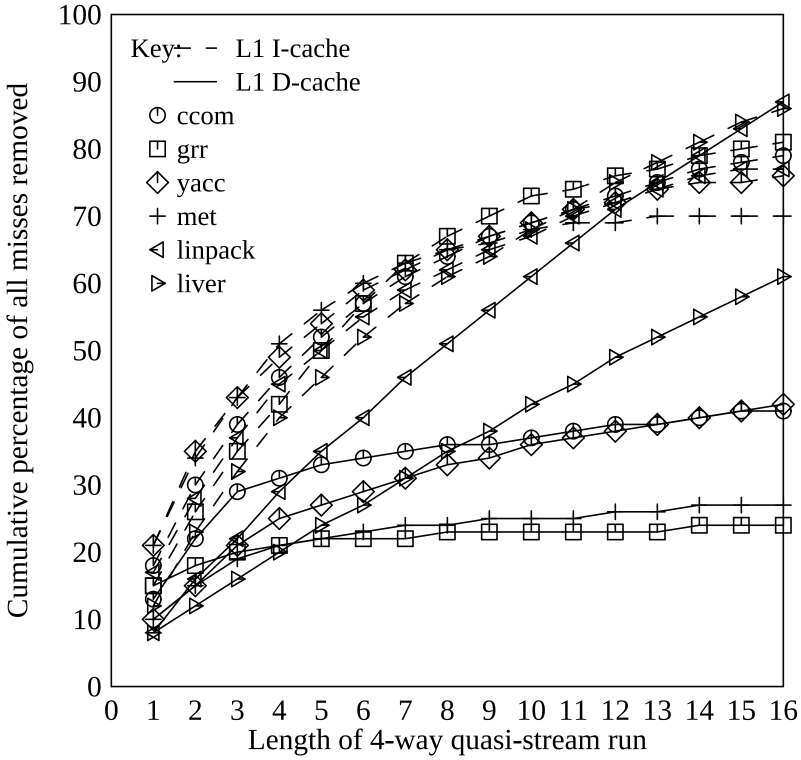

Figure 13: Quasi-sequential 4-way stream buffer performance

3.2. Stream Buffer vs. 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 3. 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. For example, in [9] 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.

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 3: Upper bound on prefetch performance: percent reduction in misses

Table 4 compares the prefetch performance from Table 3 with the stream buffer performance presented earlier. On the instruction side, a simple single stream buffer outperforms prefetch on 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 prefetch always 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.

Table 5 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

technique	misses eliminated
for 4KB direct-mapped instruction cache w/ 16B lines:	
prefetch on miss (with 1-instr latency)	46.3%
single stream buffer	72.0%
quasi-stream buffer (3 comparator)	76.0%
tagged prefetch (with 1-instr latency)	76.8%
always prefetch (with 1-instr latency)	80.1%

Table 4: Upper bound of prefetch performance vs. instruction stream buffer performance

that of instruction references, and prefetched data is more likely to be pollution than are prefetched instructions.

technique	misses eliminated
for 4KB direct-mapped data cache w/ 16B lines:	
single stream buffer	25.0%
prefetch on miss (with 1-instr latency)	33.8%
tagged prefetch (with 1-instr latency)	40.0%
always prefetch (with 1-instr latency)	42.3%
4-way stream buffer	43.0%
4-way quasi-stream buffer	47.0%

Table 5: Upper bound of prefetch performance vs. data stream buffer performance

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. 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 6 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.

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

Table 7 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 buffer 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.

Table 8 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.

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

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

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 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. Effective Increase in Cache Size

One way of looking at the performance of systems with stream buffers is to consider the effective increase in cache size provided by using them. Table 9 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 9 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 name	1K	2K	4K	8K	16K	32K	64K
	multiple increase in effective cache size						
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 9: 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-way stream buffer and a 4-entry victim cache with 16B lines are given in Table 10. *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.

program name	1K	2K	4K	8K	16K	32K	64K
	multiple increase in effective cache size						
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 10: Effective increase in data cache size provided with stream buffers and victim caches

6. Compiler Optimizations for Stream Buffers

An interesting area for research is the ability of compilers to reorganize code and data layouts to maximize the utility of stream buffers. If techniques to optimize sequentiality of references are successful, the performance improvement from all types of stream buffers will increase while the need for more complex stream buffers may be lessened.

In order to estimate the potential value of compiler optimizations for stream buffers, several experiments were performed. A number of the benchmarks, particularly *met* and *grr* receive little benefit from data stream buffers (e.g., less than a 25% reduction in misses). These programs perform extensive manipulation of small linked record structures. In one experiment, record accesses and record layouts were rearranged to increase the degree of sequential access, either by changing the record layout, the code sequences accessing them, or both depending on the conflicting requirements of different pieces of code. This resulted in a only very small further reduction in miss rate (about 1%). This poor improvement is probably due to a number of reasons. First, the records involved were quite small to begin with, only one or two 16B cache lines long. Therefore, the number of misses that would be removed by making their access sequential ranged from small to zero (depending on their starting address when allocated on the heap). Second, often programs access record elements sequentially as a mater of programming style, since it is much easier to make sure that operations on one element are not neglected if operations on a record are performed in element order.

Another potential optimization is to rearrange loops to process arrays in increasing storage order. Here again this seems to already be the case in most programs except when the data structure must be accessed in the opposite order as part of the algorithm.

Thus, except to prevent degenerative cases, compiler optimizations to increase the sequentiality of data access may have little additional benefit for stream buffer operation.

7. Conclusions

One way to reduce the number of compulsory and capacity cache misses is to increase the line size used in a cache. Increasing the line size will also increase the number of conflict misses that occur. This results in the traditional rather shallow minima on curves of miss rate versus line size. By using miss caches (or victim caches) to ameliorate the increased numbers of conflict cache misses due to longer cache line sizes, lower overall miss rates can be attained at larger line sizes than would normally be useful. Of course, the transfer cost is also a crucial factor when selecting a cache line size. However, even after accounting for transfer costs, miss caches (and presumably victim caches) can make longer cache lines more attractive.

Another way to reduce compulsory and capacity cache misses is with a stream buffer. Stream buffers perform strictly better than any fixed line size for a cache by effectively providing a variable-sized cache line on a per-reference basis. Stream buffers avoid polluting the cache with prefetched data since they only load lines into the cache as they are requested. This yields significantly better performance than even the optimal line size per program.

Stream buffers have also been shown to be more effective than ideal (i.e., have 1-instruction issue latency) implementations of *prefetch always*, *prefetch on miss*, and *tagged prefetch* for data caches. For instruction caches stream buffers are almost as good as ideal implementations of *tagged prefetch* or *prefetch always*, and better than *prefetch on miss*. However, since stream buffers can take advantage of pipelined memory systems by having multiple outstanding requests, the performance of systems with stream buffers in practice can be much higher than these prefetch techniques.

Stream buffers in conjunction with victim caches can often provide a reduction in miss rate equivalent to a doubling or quadrupling of cache size. In some cases the effective increase in cache size provided by stream buffers and victim caches is larger than that of any size cache.

Finally, the potential for compiler optimizations to increase the performance of stream buffers was investigated. Preliminary results suggest that most of the potential cache miss sequentiality in a program is already present independent of additional optimizations.

This study has concentrated on applying 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 stream buffers needs to be investigated for operating system execution and for multiprogramming workloads. To the extent that stream buffers can quickly supply blocks of instructions and data, they may be able to reduce the cost of cache misses due to context switches.

Acknowledgements

To be supplied in final draft.

References

[1] 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.

[2] Gindle, B. S. Buffer Block Prefetching Method. *IBM Technical Disclosure Bulletin* :696-697, July, 1977.

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

[4] Jouppi, Norman P. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *The 17th Annual Symposium on Computer Architecture*, pages 364-373. IEEE Computer Society Press, May, 1990.

[5] Przybylski, Steven A. The Performance Impact of Block Sizes and Fetch Strategies. In *The 17th Annual Symposium on Computer Architecture*, pages 160-169. IEEE Computer Society Press, May, 1990.

[6] Rau, B. R. *Sequential Prefetch Strategies for Instructions and Data*. Technical Report SU-SEL 77-008, Digital Systems Laboratory, Stanford University, January, 1977.

[7] Rau, B. R. *Program Behaviour and the Performance of Memory Systems*. PhD thesis, Stanford University, 1977. Chapter 5.

[8] Smith, Alan J. Sequential Program Prefetching in Memory Hierachies. *IEEE Computer* :7-21, December, 1978.

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

[10] Smith, Alan J. Line (Block) Size Choice for CPU Cache Memories. *IEEE Transactions on Computers* :1063-1075, September, 1987.

WRL Research Reports

‘‘Titan System Manual.’’ **Michael J. K. Nielsen.** WRL Research Report 86/1, September 1986.

‘‘Global Register Allocation at Link Time.’’ **David W. Wall.** WRL Research Report 86/3, October 1986.

‘‘Optimal Finned Heat Sinks.’’ **William R. Hamburgen.** WRL Research Report 86/4, October 1986.

‘‘The Mahler Experience: Using an Intermediate Language as the Machine Description.’’ **David W. Wall and Michael L. Powell.** WRL Research Report 87/1, August 1987.

‘‘The Packet Filter: An Efficient Mechanism for User-level Network Code.’’ **Jeffrey C. Mogul, Richard F. Rashid, Michael J. Accetta.** WRL Research Report 87/2, November 1987.

‘‘Fragmentation Considered Harmful.’’ **Christopher A. Kent, Jeffrey C. Mogul.** WRL Research Report 87/3, December 1987.

‘‘Cache Coherence in Distributed Systems.’’ **Christopher A. Kent.** WRL Research Report 87/4, December 1987.

‘‘Register Windows vs. Register Allocation.’’ **David W. Wall.** WRL Research Report 87/5, December 1987.

‘‘Editing Graphical Objects Using Procedural Representations.’’ **Paul J. Asente.** WRL Research Report 87/6, November 1987.

‘‘The USENET Cookbook: an Experiment in Electronic Publication.’’ **Brian K. Reid.** WRL Research Report 87/7, December 1987.

‘‘MultiTitan: Four Architecture Papers.’’ **Norman P. Jouppi, Jeremy Dion, David Boggs, Michael J. K. Nielsen.** WRL Research Report 87/8, April 1988.

‘‘Fast Printed Circuit Board Routing.’’ **Jeremy Dion.** WRL Research Report 88/1, March 1988.

‘‘Compacting Garbage Collection with Ambiguous Roots.’’ **Joel F. Bartlett.** WRL Research Report 88/2, February 1988.

‘‘The Experimental Literature of The Internet: An Annotated Bibliography.’’ **Jeffrey C. Mogul.** WRL Research Report 88/3, August 1988.

‘‘Measured Capacity of an Ethernet: Myths and Reality.’’ **David R. Boggs, Jeffrey C. Mogul, Christopher A. Kent.** WRL Research Report 88/4, September 1988.

‘‘Visa Protocols for Controlling Inter-Organizational Datagram Flow: Extended Description.’’ **Deborah Estrin, Jeffrey C. Mogul, Gene Tsudik, Kamaljit Anand.** WRL Research Report 88/5, December 1988.

‘‘SCHEME->C A Portable Scheme-to-C Compiler.’’ **Joel F. Bartlett.** WRL Research Report 89/1, January 1989.

‘‘Optimal Group Distribution in Carry-Skip Adders.’’ **Silvio Turrini.** WRL Research Report 89/2, February 1989.

‘‘Precise Robotic Paste Dot Dispensing.’’ **William R. Hamburgen.** WRL Research Report 89/3, February 1989.

‘‘Simple and Flexible Datagram Access Controls for Unix-based Gateways.’’ **Jeffrey C. Mogul.** WRL Research Report 89/4, March 1989.

‘‘Spritely NFS: Implementation and Performance of Cache-Consistency Protocols.’’ **V. Srinivasan and Jeffrey C. Mogul.** WRL Research Report 89/5, May 1989.

‘‘Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines.’’ **Norman P. Jouppi and David W. Wall.** WRL Research Report 89/7, July 1989.

‘‘A Unified Vector/Scalar Floating-Point Architecture.’’ **Norman P. Jouppi, Jonathan Bertoni, and David W. Wall.** WRL Research Report 89/8, July 1989.

‘‘Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU.’’ **Norman P. Jouppi.** WRL Research Report 89/9, July 1989.

‘‘Integration and Packaging Plateaus of Processor Performance.’’ **Norman P. Jouppi.** WRL Research Report 89/10, July 1989.

‘‘A 20-MIPS Sustained 32-bit CMOS Microprocessor with High Ratio of Sustained to Peak Performance.’’ **Norman P. Jouppi and Jeffrey Y. F. Tang.** WRL Research Report 89/11, July 1989.

‘‘The Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance.’’ **Norman P. Jouppi.** WRL Research Report 89/13, July 1989.

‘‘Long Address Traces from RISC Machines: Generation and Analysis.’’ **Anita Borg, R.E.Kessler, Georgia Lazana, and David W. Wall.** WRL Research Report 89/14, September 1989.

‘‘Link-Time Code Modification.’’ **David W. Wall.** WRL Research Report 89/17, September 1989.

‘‘Noise Issues in the ECL Circuit Family.’’ **Jeffrey Y.F. Tang and J. Leon Yang.** WRL Research Report 90/1, January 1990.

‘‘Efficient Generation of Test Patterns Using Boolean Satisfiablilty.’’ **Tracy Larrabee.** WRL Research Report 90/2, February 1990.

‘‘Two Papers on Test Pattern Generation.’’ **Tracy Larrabee.** WRL Research Report 90/3, March 1990.

‘‘Virtual Memory vs. The File System.’’ **Michael N. Nelson.** WRL Research Report 90/4, March 1990.

‘‘Efficient Use of Workstations for Passive Monitoring of Local Area Networks.’’ **Jeffrey C. Mogul.** WRL Research Report 90/5, July 1990.

‘‘A One-Dimensional Thermal Model for the VAX 9000 Multi Chip Units.’’ **John S. Fitch.** WRL Research Report 90/6, July 1990.

‘‘1990 DECWRL/Livermore Magic Release.’’ **Robert N. Mayo, Michael H. Arnold, Walter S. Scott, Don Stark, Gordon T. Hamachi.** WRL Research Report 90/7, September 1990.

‘‘Pool Boiling Enhancement Techniques for Water at Low Pressure.’’ **Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.** WRL Research Report 90/9, December 1990.

‘‘Writing Fast X Servers for Dumb Color Frame Buffers.’’ **Joel McCormack.** WRL Research Report 91/1, February 1991.

‘‘A Simulation Based Study of TLB Performance.’’ **J. Bradley Chen, Anita Borg, Norman P. Jouppi.** WRL Research Report 91/2, November 1991.

‘‘Analysis of Power Supply Networks in VLSI Circuits.’’ **Don Stark.** WRL Research Report 91/3, April 1991.

‘‘TurboChannel T1 Adapter.’’ **David Boggs.** WRL Research Report 91/4, April 1991.

‘‘Procedure Merging with Instruction Caches.’’ **Scott McFarling.** WRL Research Report 91/5, March 1991.

‘‘Don’t Fidget with Widgets, Draw!.’’ **Joel Bartlett.** WRL Research Report 91/6, May 1991.

‘‘Pool Boiling on Small Heat Dissipating Elements in Water at Subatmospheric Pressure.’’ **Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.** WRL Research Report 91/7, June 1991.

‘‘Incremental, Generational Mostly-Copying Garbage Collection in Uncooperative Environments.’’ **G. May Yip.** WRL Research Report 91/8, June 1991.

‘‘Interleaved Fin Thermal Connectors for Multichip Modules.’’ **William R. Hamburgen.** WRL Research Report 91/9, August 1991.

‘‘Experience with a Software-defined Machine Architecture.’’ **David W. Wall.** WRL Research Report 91/10, August 1991.

‘‘Network Locality at the Scale of Processes.’’ **Jeffrey C. Mogul.** WRL Research Report 91/11, November 1991.

‘‘Cache Write Policies and Performance.’’ **Norman P. Jouppi.** WRL Research Report 91/12, December 1991.

‘‘Packaging a 150 W Bipolar ECL Microprocessor.’’ **William R. Hamburgen, John S. Fitch.** WRL Research Report 92/1, March 1992.

‘‘Observing TCP Dynamics in Real Networks.’’ **Jeffrey C. Mogul.** WRL Research Report 92/2, April 1992.

‘‘Systems for Late Code Modification.’’ **David W. Wall.** WRL Research Report 92/3, May 1992.

‘‘Piecewise Linear Models for Switch-Level Simulation.’’ **Russell Kao.** WRL Research Report 92/5, September 1992.

‘‘A Practical System for Intermodule Code Optimization at Link-Time.’’ **Amitabh Srivastava and David W. Wall.** WRL Research Report 92/6, December 1992.

‘‘A Smart Frame Buffer.’’ **Joel McCormack & Bob McNamara.** WRL Research Report 93/1, January 1993.

‘‘Recovery in Spritely NFS.’’ **Jeffrey C. Mogul.** WRL Research Report 93/2, June 1993.

‘‘Tradeoffs in Two-Level On-Chip Caching.’’ **Norman P. Jouppi & Steven J.E. Wilton.** WRL Research Report 93/3, October 1993.

‘‘Unreachable Procedures in Object-oriented Programing.’’ **Amitabh Srivastava.** WRL Research Report 93/4, August 1993.

‘‘An Enhanced Access and Cycle Time Model for On-Chip Caches.’’ **Steven J.E. Wilton and Norman P. Jouppi.** WRL Research Report 93/5, July 1994.

‘‘Limits of Instruction-Level Parallelism.’’ **David W. Wall.** WRL Research Report 93/6, November 1993.

‘‘Fluoroelastomer Pressure Pad Design for Microelectronic Applications.’’ **Alberto Makino, William R. Hamburgen, John S. Fitch.** WRL Research Report 93/7, November 1993.

‘‘A 300MHz 115W 32b Bipolar ECL Microprocessor.’’ **Norman P. Jouppi, Patrick Boyle, Jeremy Dion, Mary Jo Doherty, Alan Eustace, Ramsey Haddad, Robert Mayo, Suresh Menon, Louis Monier, Don Stark, Silvio Turrini, Leon Yang, John Fitch, William Hamburgen, Russell Kao, and Richard Swan.** WRL Research Report 93/8, December 1993.

‘‘Link-Time Optimization of Address Calculation on a 64-bit Architecture.’’ **Amitabh Srivastava, David W. Wall.** WRL Research Report 94/1, February 1994.

‘‘ATOM: A System for Building Customized Program Analysis Tools.’’ **Amitabh Srivastava, Alan Eustace.** WRL Research Report 94/2, March 1994.

‘‘Complexity/Performance Tradeoffs with Non-Blocking Loads.’’ **Keith I. Farkas, Norman P. Jouppi.** WRL Research Report 94/3, March 1994.

‘‘A Better Update Policy.’’ **Jeffrey C. Mogul.** WRL Research Report 94/4, April 1994.

‘‘Boolean Matching for Full-Custom ECL Gates.’’ **Robert N. Mayo, Herve Touati.** WRL Research Report 94/5, April 1994.

‘‘Software Methods for System Address Tracing: Implementation and Validation.’’ **J. Bradley Chen, David W. Wall, and Anita Borg.** WRL Research Report 94/6, September 1994.

‘‘Performance Implications of Multiple Pointer Sizes.’’ **Jeffrey C. Mogul, Joel F. Bartlett, Robert N. Mayo, and Amitabh Srivastava.** WRL Research Report 94/7, December 1994.

‘‘How Useful Are Non-blocking Loads, Stream Buffers, and Speculative Execution in Multiple Issue Processors?.’’ **Keith I. Farkas, Norman P. Jouppi, and Paul Chow.** WRL Research Report 94/8, December 1994.

‘‘Drip: A Schematic Drawing Interpreter.’’ **Ramsey W. Haddad.** WRL Research Report 95/1, March 1995.

‘‘Recursive Layout Generation.’’ **Louis M. Monier, Jeremy Dion.** WRL Research Report 95/2, March 1995.

‘‘Contour: A Tile-based Gridless Router.’’ **Jeremy Dion, Louis M. Monier.** WRL Research Report 95/3, March 1995.

‘‘The Case for Persistent-Connection HTTP.’’ **Jeffrey C. Mogul.** WRL Research Report 95/4, May 1995.

‘‘Network Behavior of a Busy Web Server and its Clients.’’ **Jeffrey C. Mogul.** WRL Research Report 95/5, October 1995.

‘‘The Predictability of Branches in Libraries.’’ **Brad Calder, Dirk Grunwald, and Amitabh** Srivastava. WRL Research Report 95/6, October 1995.

‘‘Shared Memory Consistency Models: A Tutorial.’’ **Sarita V. Adve, Kourosh Gharachorloo.** WRL Research Report 95/7, September 1995.

‘‘Eliminating Receive Livelock in an Interrupt-driven Kernel.’’ **Jeffrey C. Mogul and K. K. Ramakrishnan.** WRL Research Report 95/8, December 1995.

‘‘Memory Consistency Models for Shared-Memory Multiprocessors.’’ **Kourosh Gharachorloo.** WRL Research Report 95/9, December 1995.

‘‘Register File Design Considerations in Dynamically Scheduled Processors.’’ **Keith I. Farkas, Norman P. Jouppi, Paul Chow.** WRL Research Report 95/10, November 1995.

‘‘Optimization in Permutation Spaces.’’ **Silvio** Turrini. WRL Research Report 96/1, November 1996.

‘‘Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory.’’ **Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath.** WRL Research Report 96/2, November 1996.

‘‘Efficient Procedure Mapping using Cache Line Coloring.’’ **Amir H. Hashemi, David R. Kaeli, and Brad Calder.** WRL Research Report 96/3, October 1996.

‘‘Optimizations and Placement with the Genetic Workbench.’’ **Silvio Turrini.** WRL Research Report 96/4, November 1996.

‘‘Memory-system Design Considerations for Dynamically-scheduled Processors.’’ **Keith I. Farkas, Paul Chow, Norman P. Jouppi, and Zvonko Vranesic.** WRL Research Report 97/1, February 1997.

‘‘Performance of the Shasta Distributed Shared Memory Protocol.’’ **Daniel J. Scales and Kourosh Gharachorloo.** WRL Research Report 97/2, February 1997.

‘‘Fine-Grain Software Distributed Shared Memory on SMP Clusters.’’ **Daniel J. Scales, Kourosh Gharachorloo, and Anshu Aggarwal.** WRL Research Report 97/3, February 1997.

‘‘Potential benefits of delta encoding and data compression for HTTP.’’ **Jeffrey C. Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy.** WRL Research Report 97/4, July 1997.

WRL Technical Notes

‘‘TCP/IP PrintServer: Print Server Protocol.’’ **Brian K. Reid and Christopher A. Kent.** WRL Technical Note TN-4, September 1988.

‘‘TCP/IP PrintServer: Server Architecture and Implementation.’’ **Christopher A. Kent.** WRL Technical Note TN-7, November 1988.

‘‘Smart Code, Stupid Memory: A Fast X Server for a Dumb Color Frame Buffer.’’ **Joel McCormack.** WRL Technical Note TN-9, September 1989.

‘‘Why Aren’t Operating Systems Getting Faster As Fast As Hardware?.’’ **John Ousterhout.** WRL Technical Note TN-11, October 1989.

‘‘Mostly-Copying Garbage Collection Picks Up Generations and C++.’’ **Joel F. Bartlett.** WRL Technical Note TN-12, October 1989.

‘‘Characterization of Organic Illumination Systems.’’ **Bill Hamburgen, Jeff Mogul, Brian Reid, Alan Eustace, Richard Swan, Mary Jo Doherty, and Joel Bartlett.** WRL Technical Note TN-13, April 1989.

‘‘Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers.’’ **Norman P. Jouppi.** WRL Technical Note TN-14, March 1990.

‘‘Limits of Instruction-Level Parallelism.’’ **David W. Wall.** WRL Technical Note TN-15, December 1990.

‘‘The Effect of Context Switches on Cache Performance.’’ **Jeffrey C. Mogul and Anita Borg.** WRL Technical Note TN-16, December 1990.

‘‘MTOOL: A Method For Detecting Memory Bottlenecks.’’ **Aaron Goldberg and John** Hennessy. WRL Technical Note TN-17, December 1990.

‘‘Predicting Program Behavior Using Real or Estimated Profiles.’’ **David W. Wall.** WRL Technical Note TN-18, December 1990.

‘‘Cache Replacement with Dynamic Exclusion.’’ **Scott McFarling.** WRL Technical Note TN-22, November 1991.

‘‘Boiling Binary Mixtures at Subatmospheric Pressures.’’ **Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.** WRL Technical Note TN-23, January 1992.

‘‘A Comparison of Acoustic and Infrared Inspection Techniques for Die Attach.’’ **John S. Fitch.** WRL Technical Note TN-24, January 1992.

‘‘TurboChannel Versatec Adapter.’’ **David Boggs.** WRL Technical Note TN-26, January 1992.

‘‘A Recovery Protocol For Spritely NFS.’’ **Jeffrey C. Mogul.** WRL Technical Note TN-27, April 1992.

‘‘Electrical Evaluation Of The BIPS-0 Package.’’ **Patrick D. Boyle.** WRL Technical Note TN-29, July 1992.

‘‘Transparent Controls for Interactive Graphics.’’ **Joel F. Bartlett.** WRL Technical Note TN-30, July 1992.

‘‘Design Tools for BIPS-0.’’ **Jeremy Dion & Louis Monier.** WRL Technical Note TN-32, December 1992.

‘‘Link-Time Optimization of Address Calculation on a 64-Bit Architecture.’’ **Amitabh Srivastava and David W. Wall.** WRL Technical Note TN-35, June 1993.

‘‘Combining Branch Predictors.’’ **Scott McFarling.** WRL Technical Note TN-36, June 1993.

‘‘Boolean Matching for Full-Custom ECL Gates.’’ **Robert N. Mayo and Herve Touati.** WRL Technical Note TN-37, June 1993.

‘‘Piecewise Linear Models for Rsim.’’ **Russell Kao, Mark Horowitz.** WRL Technical Note TN-40, December 1993.

‘‘Speculative Execution and Instruction-Level Parallelism.’’ **David W. Wall.** WRL Technical Note TN-42, March 1994.

‘‘Ramonamap - An Example of Graphical Groupware.’’ **Joel F. Bartlett.** WRL Technical Note TN-43, December 1994.

‘‘ATOM: A Flexible Interface for Building High Performance Program Analysis Tools.’’ **Alan Eustace and Amitabh Srivastava.** WRL Technical Note TN-44, July 1994.

‘‘Circuit and Process Directions for Low-Voltage Swing Submicron BiCMOS.’’ **Norman P. Jouppi, Suresh Menon, and Stefanos Sidiropoulos.** WRL Technical Note TN-45, March 1994.

‘‘Experience with a Wireless World Wide Web Client.’’ **Joel F. Bartlett.** WRL Technical Note TN-46, March 1995.

‘‘I/O Component Characterization for I/O Cache Designs.’’ **Kathy J. Richardson.** WRL Technical Note TN-47, April 1995.

‘‘Attribute caches.’’ **Kathy J. Richardson, Michael J. Flynn.** WRL Technical Note TN-48, April 1995.

‘‘Operating Systems Support for Busy Internet Servers.’’ **Jeffrey C. Mogul.** WRL Technical Note TN-49, May 1995.

‘‘The Predictability of Libraries.’’ **Brad Calder, Dirk Grunwald, Amitabh Srivastava.** WRL Technical Note TN-50, July 1995.

‘‘Simultaneous Multithreading: A Platform for Next-generation Processors.’’ **Susan J. Eggers, Joel Emer, Henry M. Levy, Jack L. Lo, Rebecca Stamm and Dean M. Tullsen.** WRL Technical Note TN-52, March 1997.

WRL Research Reports and Technical Notes are available on the World Wide Web, from `http://www.research.digital.com/wrl/techreports/index.html`.