FETCH CACHES

Brian K. Bray
Michael J. Flynn

Technical Report: CSL-TR-93-561

February 1993

Supported by NASA under NAG2-248, using facilities supplied under NAGW-419.

FETCH CACHES

by

Brian K. Bray

Michael J. Flynn

Technical Report No. CSL-TR-93-561

February 1993

Computer Systems Laboratory

Departments of Electrical Engineering and Computer Science

St anford University

Stanford, California 94305-4055

Abstract

For high performance, data caches must have a low miss rate and provide high bandwidth, while maintaining low latency. Larger and more complex set associative caches provide lower miss rates but at the cost of increased latency. Interleaved data caches can improve the available bandwidth, but the improvement is limited by bank conflicts and increased latency due to the switching networks required to distribute cache addresses and to route the data.

We propose using a small buffer to reduce the data read latency or improve the read bandwidth of an on-chip data cache. We call the small read-only buffer a fetch *cache.* The *fetch cache* attempts to capture the immediate spatial locality of the data read reference stream by utilizing the large number of bits that can be fetched in a single access of an on-chip cache.

There are two ways a processor can issue multiple instructions per cache access: the cache access can require multiple cycles (i.e. superpipelined), or multiple instructions are issued per cycle (i.e. superscalar). In the first section, we show the use of *fetch caches* with multi-cycle per access data caches. When there is a read hit in the *fetch cache,* the read request can be serviced in one cycle, otherwise the latency is that of the primary data cache. For a four line, 16 byte wide *fetch cache,* the hit rate ranged from 40 to GO percent depending on the application. In the second part, WC show the use of *fetch caches* when multi-accesses per cycle are requested. When there is a read hit in the *fetch cache,* a read can be satisfied by the *fetch cache,* while the primary cache performs another read or write request. For a four line, 16 byte wide *fetch cache,* the cache bandwidth increased by 20 to 30 percent depending on the application.

Key **Words and Phrases:** cache, fetch cache, latency, bandwidth, interleave

Contents

1 Introduct ion

Initially, on-chip caches were small so the main concern was to reduce the miss rate and the miss penalty. Increased levels of device integration allow larger on-chip caches resulting in decreased cache miss rates, while multi-level caches minimize the first-level cache miss penalty [PHH89].

The increase in device integration also allows architects to put multiple copies of functional units (or even processors) on a chip, thus increasing the need for bandwidth. A higher performance memory system must be developed to support increases in processor performance. A processor which exploits instruction-level parallelism or one which can issue instructions faster than a cache access needs more bandwidth from the instruction and data caches. Most increases in instruction bandwidth requirements can be satisfied by just increasing the fetch width and/or using self-aligning caches with the increased fetch width. The instruction stream has high sequential locality so moderate increases in the fetch width works well. However, the data stream has much less sequential locality, thus just increasing the fetch width does not significantly improve performance. The only advantage the data stream has is that not every instruction requires a data memory reference so the frequency of the data references is lower than instruction referen ces.

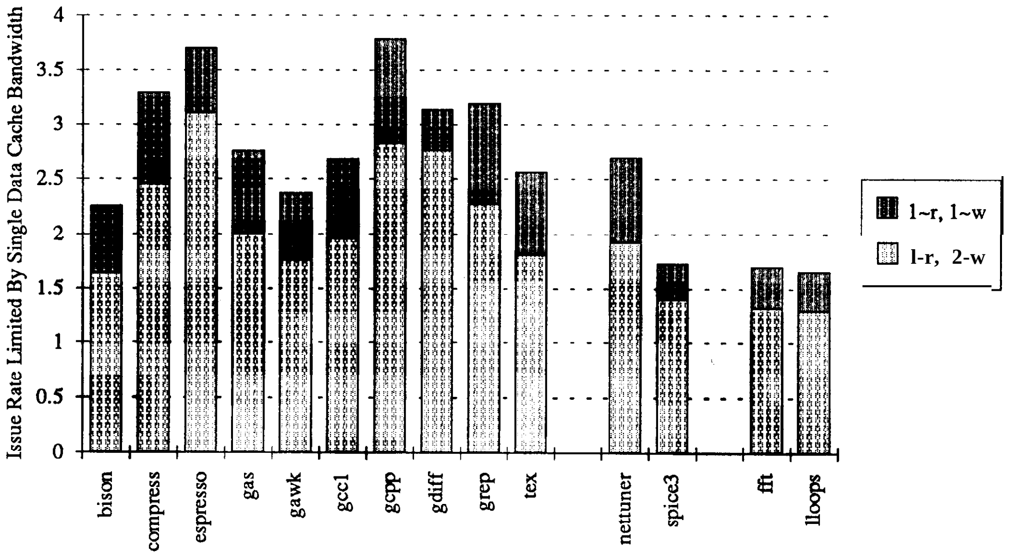

Figure 1: Performance Limited By Single Data. Port

Figure 1 shows what the issue rate would be if a processor was able to issue multiple instructions per cache access and the only limit was the bandwidth of the single port data cache. The benchmarks form three categories (non-numeric, non-structured numeric, and structured numeric) (see Appendix). For example, the issue rate of the benchmark bison is limited to approximately 2.2 instructions per cache access if a read or write occupies the cache for one access. If a write occupies the cache for two accesses, the issue rate is now limited to approximately 1.6 instructions per cache access. (This assumes that no speculative reads or writes are executed, and that the cache never misses.) Notice that the structured numeric benchmarks (fft, lloops) which, in general, have higher degrees of instruction-level parallelism are the most limited by bandwidth from the data cache.

We propose using a small buffer to reduce the data read latency or improve the read bandwidth of an on-chip data cache. We call the small read-only buffer a *fetch cache.* The *fetch cache* attempts to capture the immediate spatial locality of the data read reference stream by utilizing the large number of bits that can be fetched in a single access of an on-chip cache.

There are two ways a processor can issue multiple instructions per cache access: the cache access can require multiple cycles (i.e. superpipelined), or multiple instructions are issued per cycle (i.e. superscalar). In the first part we show the use of *fetch caches* with multi-cycle per access data caches, and in the second part we show the use of *fetch caches* when multi-accesses per cycle are requested.

2 Methodology

Trace driven simulation produced the presented results. Fourteen C benchmarks were optimized with the Ultrix 4.2 (Mips 2.1) C compiler (see Appendix). The architecture simulated was the MIPS R2000/R3000 which had been extended to use 64-bit floating point loads and stores. The fourteen C benchmarks were separated into three categories: non-numeric (essentially no floating point), non-structured numeric (floating point and the accesses are not very structured), and structured numeric (floating point and the accesses are very structured). Also used is a set of VAX traces (ATUM) which include operating system references [ASH86]. Within a benchmark set, the results are averaged and each benchmark is weighted equally.

3 Multi-Cycle Data Caches

With decreasing device feature size, it becomes increasingly important to reduce the on-chip miss rate because going off-chip becomes increasing slow and thus increases the miss penalty (even with second-level caches). One way to reduce the miss rate is to increase the cache size. As the number of devices per chip increases, larger caches are possible. Larger caches take longer to access since they are physically bigger and hence further away from the processor (even when on-chip). In addition, the decoder is bigger and hence slower. Another way to reduce the miss rate is to add associativity, however associativity increases access time.

To obtain a cache with a low miss rate, the cache size and associativity must be increased. However, this increases the latency of a cache access. Since the cache access is a critical path in many processors, the cycle time of those processors has to increase. Sometimes this tradeoff (increasing the cycle time for decreases in cache miss rate) can yield better performance [PHH88].

An alternative to increasing the cycle time is to make the cache access multiple cycles [OMB92]. Instead of being one cycle, the cache access may have the latency of two instruction issue cycles. However since the cache is on-chip it can be pipelined and thus be able to accept an access per cycle, but still have a latency of two cycles (e.g. first-level caches on the MIPS R4000 [KH92]). A single cycle latency to the data cache is desirable, since a larger latency for load instructions decreases the available parallelism by requiring the processing unit to wait longer for operands. This creates longer dependency chains in the execution pipeline. A longer pipeline is also less desireable because it increases the control complexity and data forwarding hardware, which might become the critical path.

Instead of slowing down the processor so that the cache has one cycle access, we propose that another level of cache be used. Multi-levels of caches have previously been used for decreasing the miss penalty of the first-level cache [PHH89], not for affording a lower latency of the primary cache (or for more bandwidth, as described in the next section). This cache between the primary data cache and the processor has to be small, but the the miss penalty must be very low. If the rniss penalty is too high, it might be better to use longer latency loads or a slower cycle time.

3.1 Fetch Cache For Multi-Cycle Access Data Caches

One way to solve the need for a faster data cache access is to include a small cache to buffer the most recently referenced cache lines (or subblock). This small cache can be filled with multiple words in a single access since many bits are fetched from an on-chip cache even though only a portion is sent to the processing unit. We call this very small read-only cache a *fetch cache* (Figure 2). On a cache read access, the read access is sent to the *fetch cache* and to the primary data cache (if the primary cache is not busy). This is done to minimize the miss penalty. If the *fetch cache* misses, the miss penalty is only the difference between the latency of the *fetch cache* and the primary cache instead of the latency of the primary data cache. On a miss, the *fetch cache* is reloaded with the line that missed while the primary cache sends the data to the processor. The *fetch cache* has the additional benefit that while a write operation occupies the primary on-chip data cache, the *fetch cache* can supply data read bandwidth. This is especially useful since writes can require multiple cache accesses.

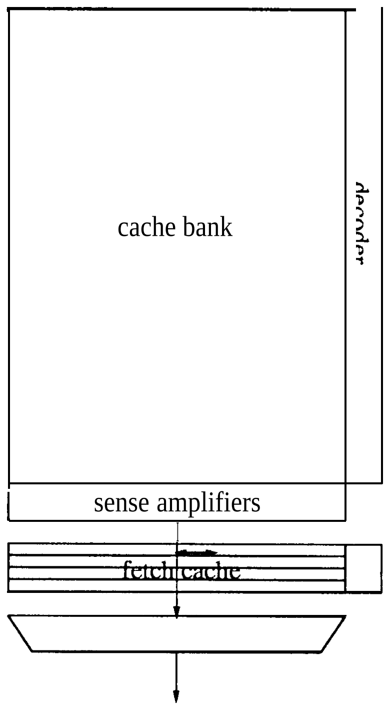

Figure 2: Fetch Cache For Multi-Cycle Cache

The scheme whereby the data address is sent to two levels of cache has been used before in the MIPS R6000 [RLT90]. In the MIPS R6000, the primary data cache was single cycle, 16KB, and off-chip. The secondary cache was two-cycle, non-pipelined, 512KB, and also off-chip. If the second-level cache was not busy with a previous request, then the second-level cache would be started at the same time as the first-level cache. If the first-level cache missed, the data would be available one cycle later, thus minimizing the miss penalty.

The scheme where a small specialized cache is used to reduce the access time of the primary cache has previously been developed for subword data references [BF92a]. It was shown that there was very good spatial locality for subword data references. The high hit rate showed that implementing a very small *subword cache* could eliminate most of the penalty due to removing the subword extraction/insertion hardware from the primary cache access path.

Caches work because of locality: temporal and spatial. Allocating only the item that missed to a very small cache only gets the benefit of temporal locality. When possible, compilers allocate variables that have temporal locality to registers, thus a very small cache usually does not benefit from temporal locality. So the benefit must be obtained from spatial locality, which can be obtained by fetching several neighboring objects on a cache miss. If the primary cache is on-chip with this small *fetch cache,* we can take advantage of the large number of wires afforded by VLSI and transfer many bits on a cache miss. *Fetch caches* may not be practical if the primary cache is off-chip due to pin limitations. When a line (or subblock) is accessed from the primary cache a large transfer unit can be sent to the *fetch cache.* Since the fill bandwidth can be large, the *fetch cache* can be filled with a single access. The large line fill allows the *fetch cache* to take advantage of spatial locality.

The idea of using the large fetch width afforded by VLSI has been used in the Intel i486, but only for the instruction stream. In the Intel i486 the instruction buffer is filled 16 bytes at a time from the unified cache [Sho90]. This works well in reducing the contention between instruction and data access to the unified cache. The large fetch width for the instruction buffer works well because the instruction stream has a large degree of spatial locality.

In the Figures 3, 4, 5, and 6, we show the read hit rate for some small *fetch caches.* Two policies for dealing with writes was simulated: invalidate on write hit (wi), and update on write hit (wu). As expected, updating on write hits had better performance than invalidation on write hits. However, the circuitry required to update the *fetch cache* on a write hit is probably too complex to implement for the benefit.

As expected, increasing the number of lines and/or fetch width increased the read hit rate. Also as expected, increasing the associativity increased the read hit rate. Interestingly, increasing the fetch width had more of an effect as the number of lines and the associativity increased. The contents of the line can stay around longer, thus increasing the chance that the increased spatial locality afforded by a larger fetch width is used.

The non-numeric benchmark set (Figure 3) had the best all-around performance. The read requests are dominated by word and byte sized objects which exhibit good spatial locality.

The non-structured numeric benchmark set (Figure 4) had the worse all-around performance. The read requests have a significant proportion of double word sized objects and the reference pattern has considerably less immediate spatial locality.

The structured numeric benchmark set (Figure 5) has its worse performance when the line size is eight bytes, but has its best performance when the line size is larger and there are four lines and associativity greater than direct-mapped. The structured numeric benchmark set (Figure 5) is the benchmark set that benefited most by having the line size above eight bytes, because most of the read requests are double word sized objects. The structured numeric benchmark set also benefits by having the associativity increase when there are four lines. This is because the matrix elements that are being accessed collide in the direct-mapped *fetch cache.*

The ATUM benchmark set (Figure 6) has similar, although slightly lower, performance than the non-numeric benchmark set even though the ATUM benchmark set includes some non-structured benchmarks and operating system references.

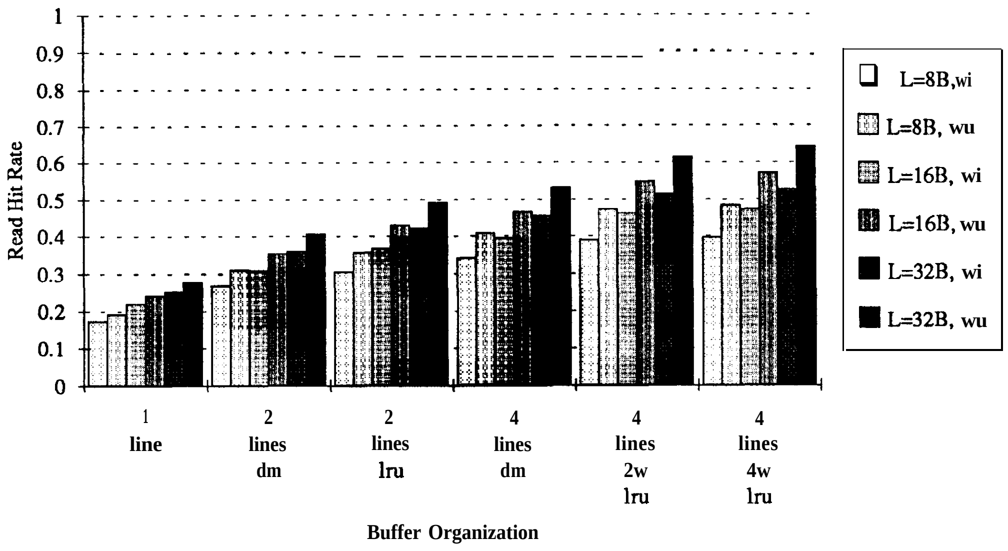

Figure 3: Read Hit Rate For Non-Numeric Benchmark Set

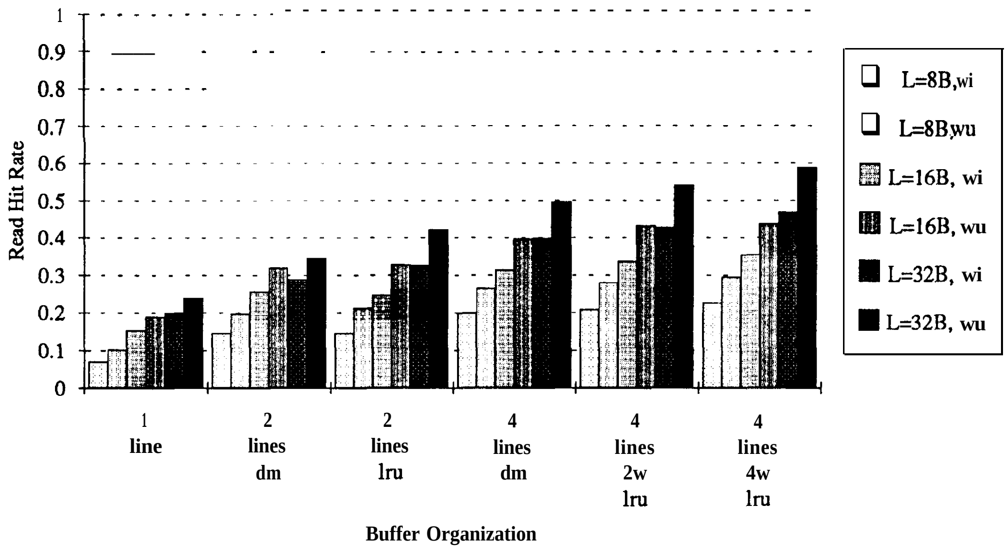

Figure 4: Read Hit Rate For Non-Structured Numeric Benchmark Set

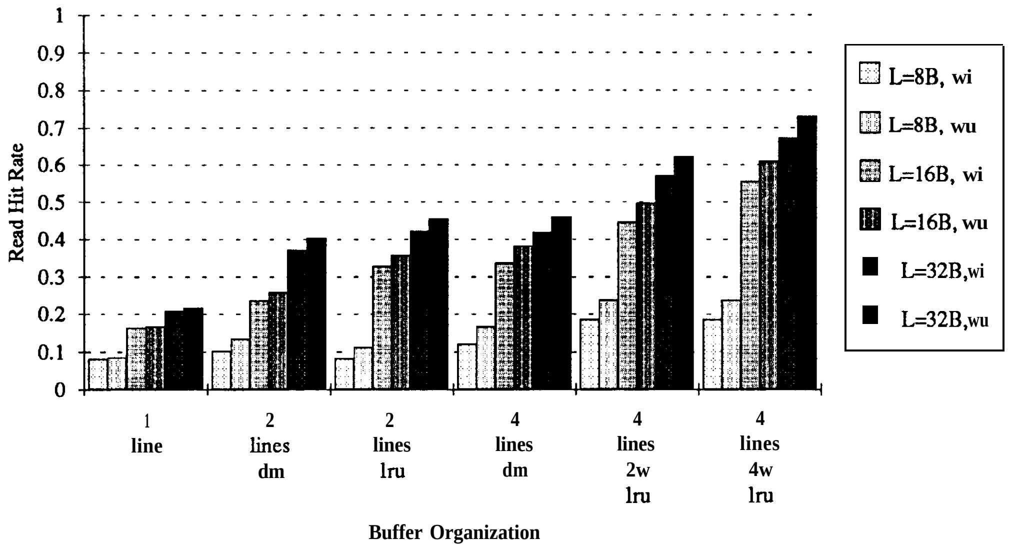

Figure 5: Read IIit Rate For Structured Numeric Benchmark Set

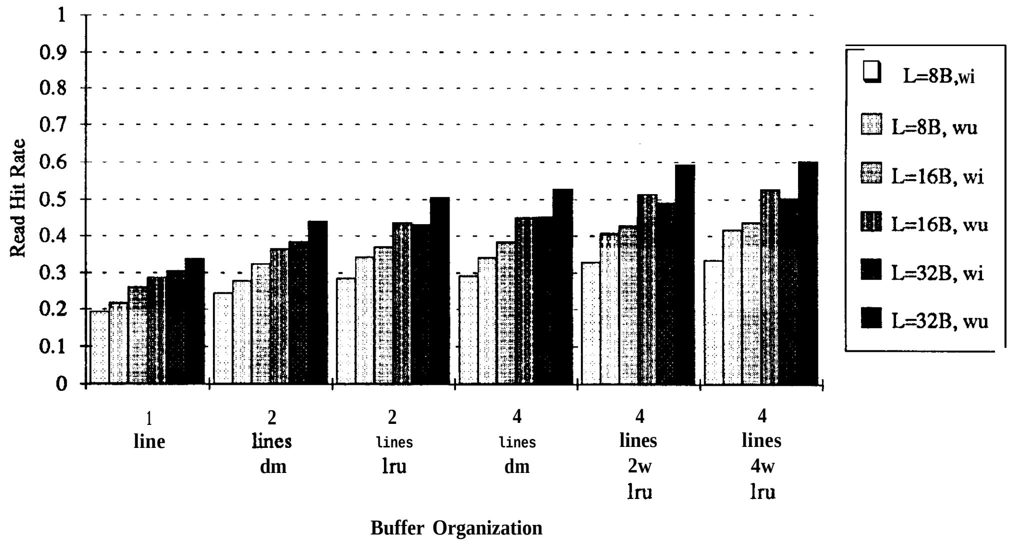

Figure 6: Read Hit Rate For ATUM Benchmark Set

4 Multi-Access Data Cache Per Cycle

Three simple ways to increase the data cache bandwidth are to replicate, multiport, or interleave the cache. Replicating the cache with *n* copies would increase the read bandwidth by *n* fold, but the cost in area would be prohibitive. Also, replicating the cache does not increase the write bandwidth since writes have to go to all copies of the cache. Multi-porting the cache significantly complicates the design and significantly increases the area requirements. The number of transistors per bit of storage, the wires for selecting and routing the data, the decoder, and the sensing circuitry all have to increase. Interleaving does not waste area as the other two methods would, but it also has some drawbacks, as we discuss in the next section.

4.1 Interleaved Caches

Interleaving has the possibility of increasing bandwidth on both reads and writes [SF91, BP91]. However with interleaving, bank contention reduces possible bandwidth. Also with interleaving, one switching network is needed to distribute cache addresses, and another switching network is needed to route the cache data. On a cache access the address must be switched to the appropriate bank after some sort of priority encoder has determined which cache bank is connected to a particular cache port. After the cache is read, the data must then go through a switching network to get to the appropriate input register. The extra levels of logic and wires can slow down the cache access, which is a critical path in many processors.

The degree of interference depends on the degree of interleaving (number of banks), the boundary size at which a bank is interleaved, and the application. A two-ported cache was simulated where the number of banks and the interleaving boundary was varied (Figures 7, 8, and 9). The caches were presented with the two most recent data addresses per cycle, so at most two requests could be serviced per cycle. (A read or write was assumed to require only one cache access.) When interleaving on a four byte boundary, the path from the cache banks is only 32 bits wide (also in this case the two data ports are also only 32 bits wide) thus a double precision floating point reference occupies both ports. When interleaving on eight or sixteen byte boundaries, the ports can be 64 bits wide so a double precision floating point reference occupies only one port.

As can be seen in the Figure 7, the structured numeric benchmark set whose memory references are heavily dominated by double precision floating point references (Appendix B,C), do not have much performance benefit from interleaving when the ports to the cache are 32 bits wide. For the non-structured numeric benchmark set, the load and store distribution is not as heavily dominated by double precision floating point references so there is some benefit, but not as much as seen by the non-numeric or the ATUM benchmark sets. As the boundary of interleaving increases above the most prevelent load and store object size, the service rate decreases. This is because of the spatial locality of the reference stream. The larger the interleaving boundary compared to the load and store object size, the higher the probability that the next reference was to the same cache block, resulting in contention and in a lower service rate.

For comparison purposes we also included the performance predicted by that of the binomial approximation.

Binomial Approximation:

$$B(m, n) = m + n - \frac{1}{2} - \sqrt{(m + n - \frac{1}{2})^2 - 2mn}$$

where m is the number of modules and n is the number of requests.

The binomial approximation does not include the interleaving boundary size, so the binomial approximation does not vary as the interleaving boundary size is changed. Thus, when the dominant load and store object size is different from the size of the interleaving boundary, the binomial approximation becomes overly optimistic. The binomial approximation is also overly optimistic with an increasing number of banks because it assumes a uniform distribution of requests which does not occur because of locality. With only two cache ports, the binomial approximation quickly approaches two memory requests serviced per cycle with more than four banks, while the service rate from actual traces starts to level out with more than four banks.

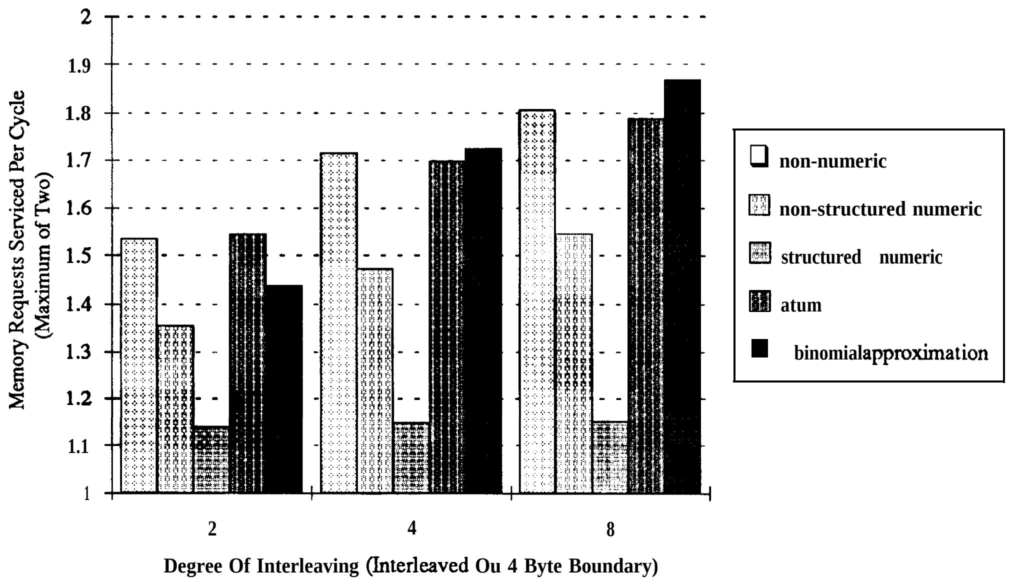

Figure 7: Effect Of Interference With Four Byte Interleaving

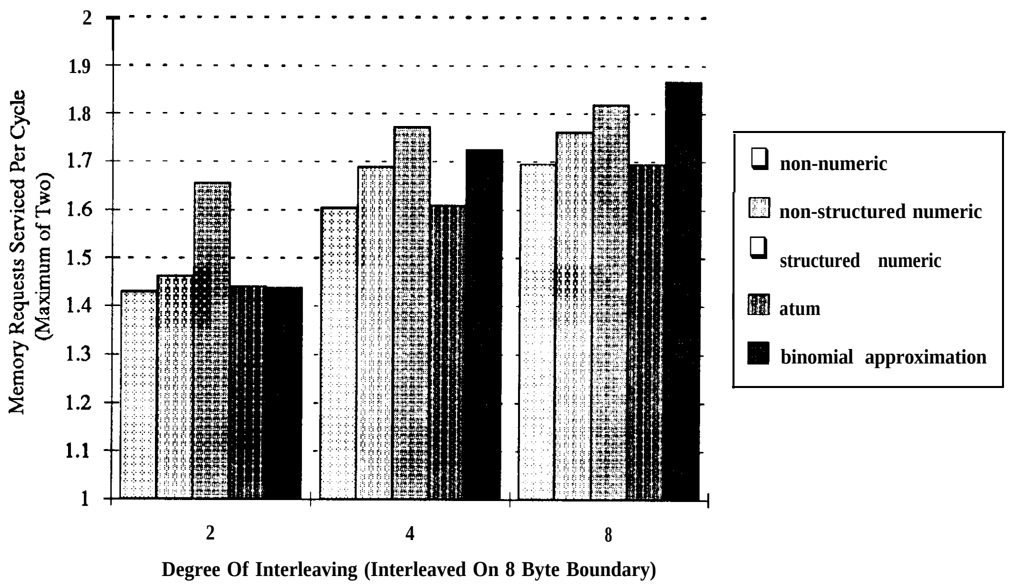

Figure 8: Effect Of Interference With Eight Byte Interleaving

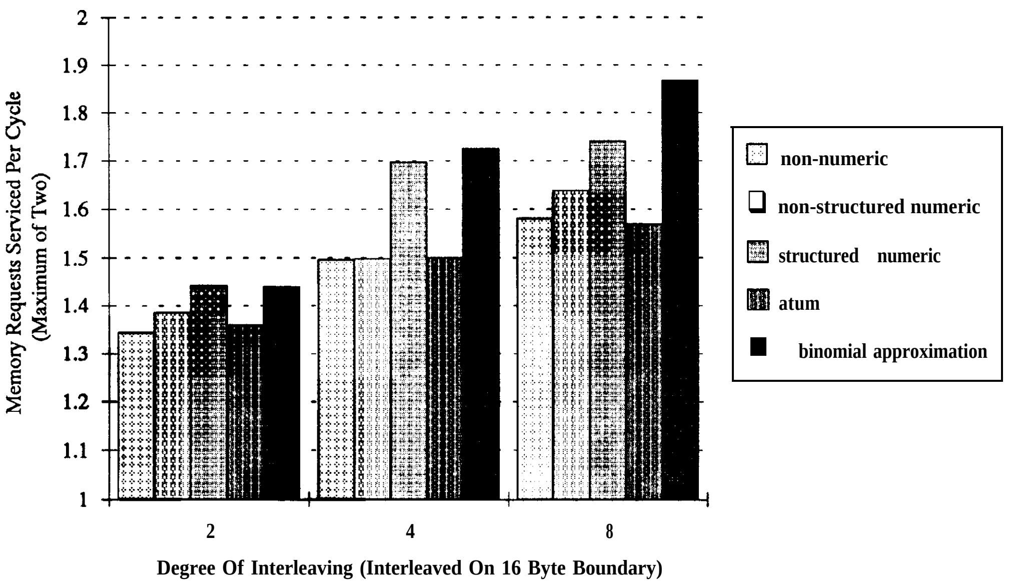

Figure 9: Effect Of Interference With Sixteen Byte Interleaving

4.2 Fetch Cache For Multi-Access Data Caches

In the previous section, we saw that interleaving the data cache could improve the available bandwidth, but the improvement is limited by bank conflicts, and interleaving comes with a cost in increased access time. One way to solve the need for multiple data cache accesses is to use a fetch *cache* to supply more read bandwidth (Figure 10).

For a two access system, the primary data cache can supply all the write bandwidth and some of the read bandwidth. The rest of the read bandwidth is supplied by the *fetch cache.* The primary cache is connected to one port and the *fetch cache* is connected to the other port, so there is no switch network as with the interleaved cache to increase the access time.

On a primary cache read, the *fetch cache* is loaded with the cache line (or subblock) that is accessed. If there is spatial locality in the next few read requests, there will be another read to that cache line which the *fetch cache* could satisfy. The *fetch cache* is small enough to be fast enough to perform a write from the previous cache read in the first half of a cache access and have a read access in the second half of a cache access.

The following happens to the two most ready to issue data requests when attempting to send the memory requests to the appropriate cache. The first request goes to the primary cache port. Only if the second request is a read request is the second request sent to the *fetch cache.* If the *fetch cache* misses and the primary cache request was to the same line (the one that is being automatically filled the next cycle), then in the next cycle the *fetch cache* request is retried. Otherwise, the *fetch cache* request that missed (and is not to be retried or was not issued because it was a write) becomes the top most request which the primary cache port will service in the next cycle.

Figures 11, 12, 13, and 14, show the bandwidth improvement for some small *fetch caches.* Two policies for dealing with writes were simulated: invalidate on write hit (wi), and update on write hit (wu). As expected, updating on write hits had better performance than invalidation on write hits. However, the circuitry required to update the *fetch cache* on a write hit is probably too complex

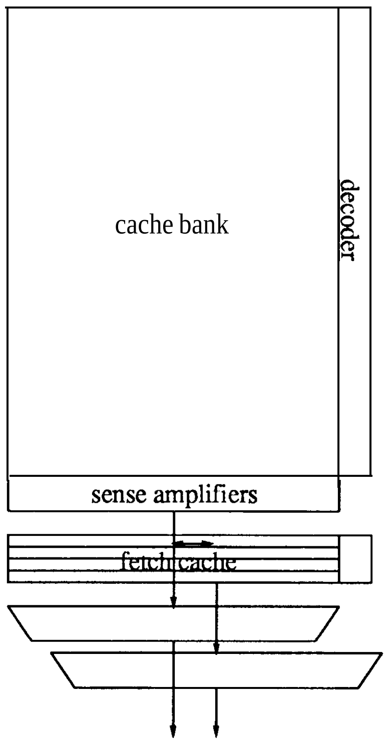

Figure 10: Fetch Cache For Multi-Access Caches

to reasonably implement.

The bandwidth improvement of a *fetch cache* is directly related to the hit rate. Overall, the non-numeric benchmark set had the best read hit rate (Figure 3), so it is expected that the bandwidth benefit would be the best (Figure 11). However, once the line size was larger than four bytes, the structured numeric benchmark set has better bandwidth improvements than the non-numeric benchmark set. This is because the ratio of reads to writes is also important. For the structured numeric benchmark set the read to write ratio is higher, so improving the read bandwidth has a more significant effect. A *fetch cache* only improves the read bandwidth, the write bandwidth does not benefit. To improve write bandwidth a *write cache* [BF91] or a *tag cache* [BF92b] can be used.

Earlier we saw that the ATUM benchmark set had essentially the same read hit rate as the non-numeric benchmark set. However, the bandwidth (Figure 14) is less than that of the non-numeric benchmark set (Figure 11). The ATUM benchmark set did not achieve as good a bandwidth benefit, because the ATUM benchmark set had a lower read to write ratio. The ATUM benchmark set traces were from a VAX and the register technology (number of registers and compiler) is poorer than that of the MIPS R3000 like machine from which the other traces were generated.

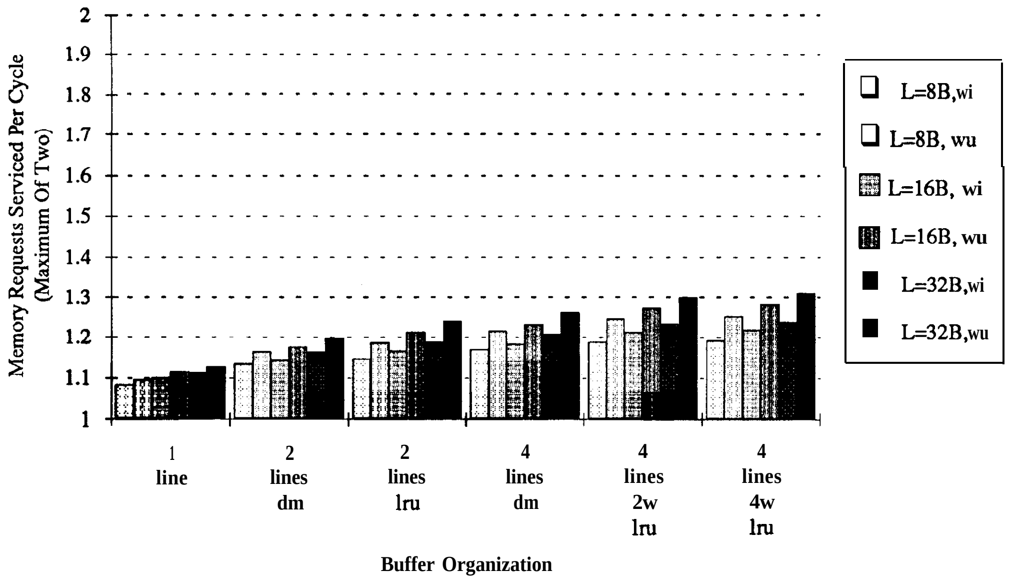

Figure 11: Service Rate For Non-Numeric Benchmark Set

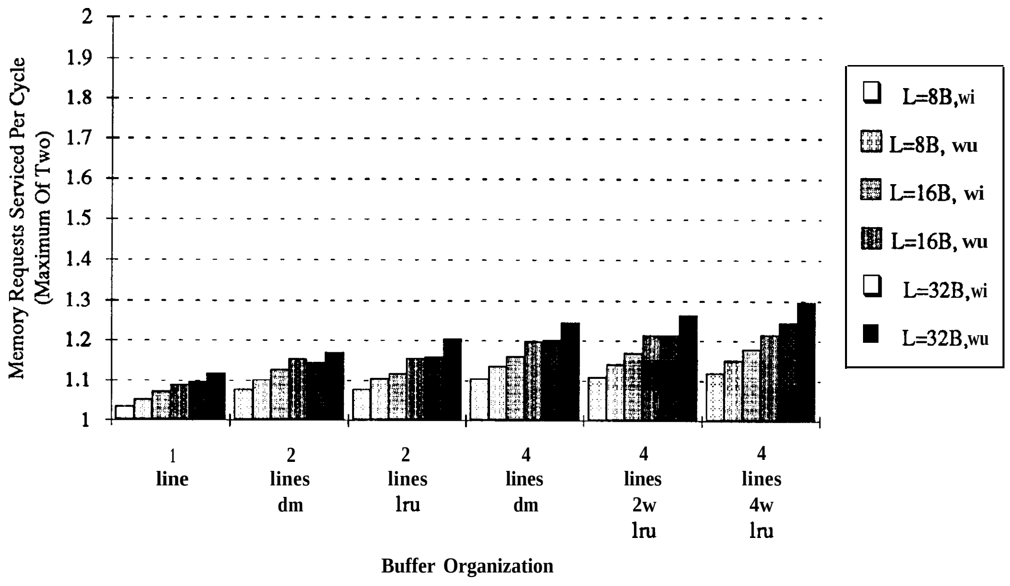

Figure 12: Service Rate For Non-Structured Numeric Benchmark Set

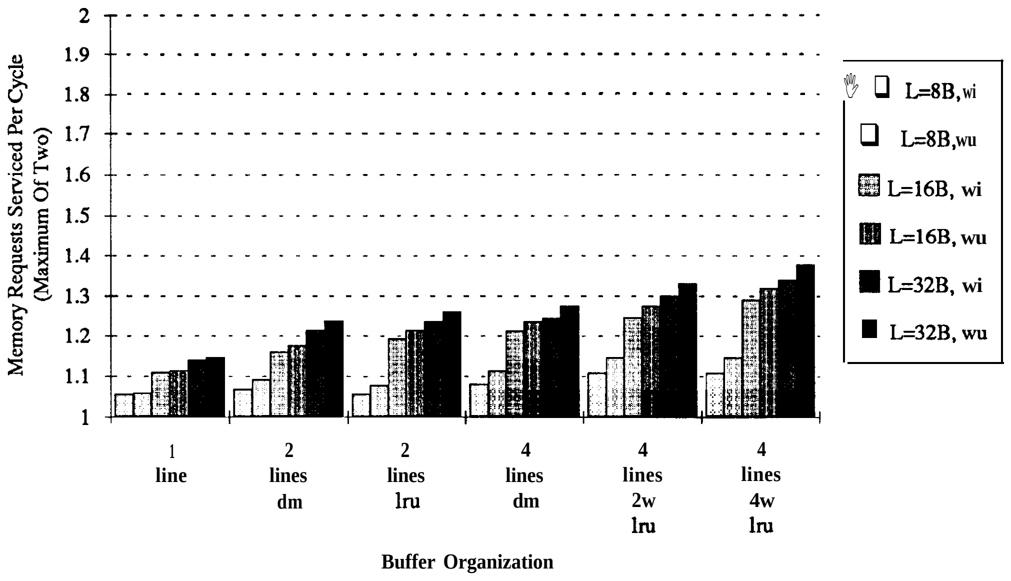

Figure 13: Service Rate For Structured Numeric Benchmark Set

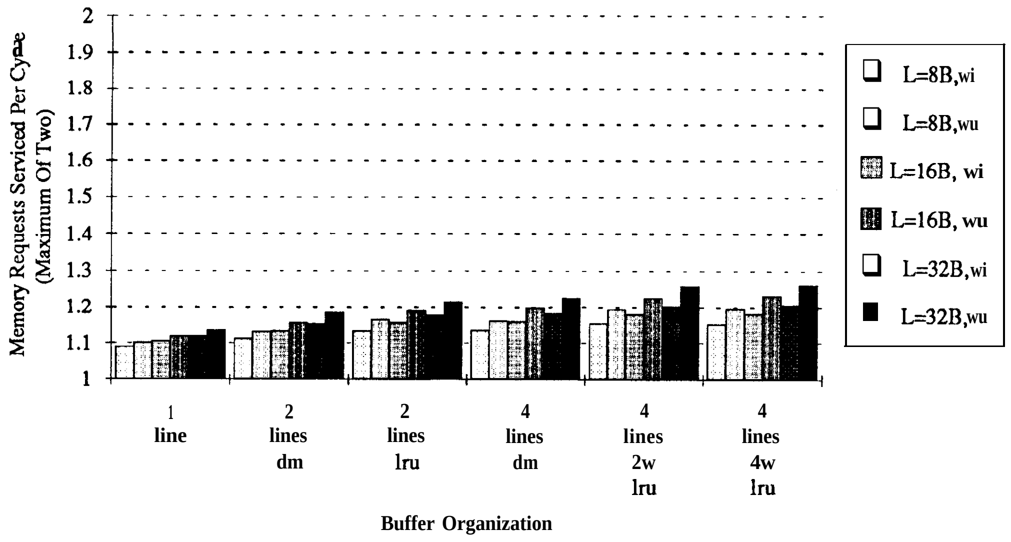

Figure 14: Service Rate For ATUM Benchmark Set

5 Conclusion

There is a enough immediate spatial locality in the data read reference stream to take advantage of the large fetch width provided by an on-chip data cache. When a data cache access takes multiple cycles, the fetch *cache* can be used to reduce the latency of the data reads by the amount of the *fetch cache* hit rate. For a four *line*, 16 byte wide *fetch cache*, the hit rate ranged from 40 to 60 percent depending on the application.

When a data cache is required to supply multiple requests per cycle, the cache can be interleaved. The interleaving needs to be the size of the dominant object, so interleaving on eight bytes for double precision floating point objects is not the proper choice if most applications are integer based which would benefit more from a four byte interleaving. With two cache ports, the binomial approximation is overly optimistic of interleaved cache performance if the dominant object size is not the same as the interleaving boundary or if the number of banks is greater than four. Interleaving does improve the bandwidth available but the interconnect and control needed for interleaving can slow down the cache access.

When a data cache is required to supply multiple requests per cycle, a *fetch cache* can be used to increase in the data read bandwidth. However, the *fetch cache* can only support a modest increase in the data bandwidth since it is limited by the *fetch cache* read hit rate and the ratio of reads to writes. For a four line, 16 byte wide *fetch cache*, the cache bandwidth increased by 20 to 30 percent depending on the application. A *fetch cache* does not yield as much extra bandwidth as interleaving but does not require the switching network that could cause the access time to increase. Because *fetch caches* only improve the read bandwidth, other methods of improving write bandwidth *(write caches* and *tag caches)* can be used to improve the data cache bandwidth.

References

[ASH86] A. Agarwal, R. Sites, and M. Horowitz. ATUM: A New Techinique for Capturing Address Traces Using Microcode. In *Conference Proceedings, The 13th International Symposium on Computer Architecture,* pages 119-127, June 1986.

[BF91] Brian K. Bray and M. J. Flynn. Write Caches As An Alternative To Write Buffers. Technical Report No. CSL-TR-91-470, Computer Systems Laboratory, Stanford University, April 1991.

[BF92a] Brian K. Bray and M. J. Flynn. Efficiently Supporting Subword Loads And Stores. Technical Report No. CSL-TR-92-514, Computer Systems Laboratory, Stanford University, April 1992.

[BF92b] Brian K. Bray and M. J. Flynn. Tag Caching For Improving Write-Back Cache Bandwidth. Technical Report No. CSL-Tlt-92-509, Computer Systems Laboratory, Stanford University, February 1992.

[BP91] M. Butler and Y. Patt. The Effect Of Real Data Cache Behavior on the Performance of a Microarchitecture that Supports Dynamic Scheduling. In *Conference Proceedings, Micro-24,* pages 34-4 1, November 1991.

[KH92] Gerry Kane and Joe Hienrich. *MIPS RISC Architecture.* Prentic Hall, 1992.

[OMB92] Kunle Oluktun, T. Mudge, and R. Brown. Performance Optimization of Pipelined Primary Caches. In *Conference Proceedings, The 19th International Symposium on Computer Architecture,* pages 181-190, May 1992.

[PHH88] S. Przybylski, M. Horowitz, and J. Hennessy. Performance Tradeoffs In Cache Design. In *Conference Proceedings, The 15th International Symposium on Computer Architecture,* pages 290–298, May 1988.

[PHH89] S. Przybylski, M. Horowitz, and J. Hennessy. Characteristics of Performance-Optimal Multi-Level Cache Hierarchies. In *Conference Proceedings, The 16th International Symposium on Computer Architecture,* pages 114-121, May 1989.

[RLT90] D. Roberts, T. Layman, and G. Taylor. An ECL RISC Microprocessor Designed for Two Level Cache. In *Conference Proceedings, Compcon,* pages 228-231, Spring 1990.

[SF91] G. Sohi and M. Franklin. High-Bandwidth Data Memory Systems for Superscalar Processors. In *Conference Proceedings, ASPLOS* **4,** pages 53-62, April 1991.

[Sho90 Ken Shoemaker. The i486 Microprocessor Integrated Cache and Bus Interface. In *Conference Proceedings, Compcon,* pages 248-253, Spring 1990.

A Benchmarks: Instruction Distribution

Name	Description	Instr. (10^6)	% loads	% stores	% branches
bison	gnu yacc (LALR parser generator) - parsing cexp.y of the gnu C compiler	4.79	28.1	16.4	22.8
compress	reduces the size of a file using adaptive Lempel-Ziv coding - compressing a 150KB tar file	12.1	20.0	10.4	21.3
espresso	boolean expression minimizer - reducing a 14-bit input, 8-bit output PLA matrix	151	21.7	5.2	17.6
gas	gnu assembler - assembling a 1800 line file	6.54	23.0	13.4	24.6
gawk	gnu awk - pattern matching and processing on a 304 line file	1.41	26.9	14.3	25.0
gcc1	gnu C compiler version 1.36 - compiling (and optimizing) to assembly code a 1500 line C program	45.2	23.7	13.7	19.6
gcpp	gnu C preprocessor - preprocessing a 1500 line C program	1.09	17.4	9.0	24.9
gdif f	gnu diff - comparing two 1112 line files	1.14	27.5	4.4	16.8
grep	grep - search file for regular expression	0.191	18.6	12.8	29.3
tex	document preparation system - formatting of a 14 page technical report	81.3	22.9	16.2	17.6

Table 1: Instruction Distribution of Non-Numeric Type Applications

Name	Description	Instr. (10^6)	% loads	% stores	% branches
nettuner	nettuner - reads a netlist of a 4x4 multiplier and pads the circuit delays	12.3	21.0	14.6	21.8
spice3	circuit simulator - simulation of a Schottky TTL edge-triggered register	150	38.2	9.0	20.5

Table 2: Instruction Distribution of Non-Structured Numeric Type Applications

Name	Description	Instr. (10^6)	% loads	% stores	% branches
fft	fast fourier transform - 1024x1024 2-D fft	7.93	31.7	16.1	4.5
lloops	composite of the 14 Livermore Loops	0.0649	31.1	12.2	3.4

Table 3: Instruction Distribution of Structured Numeric Type Applications

B Benchmarks: Load Distribution

Name	Percent Loads Executed			Percent Instructions Executed		
	% sub32b ld	% 32b int ld	% fp ld	% sub32b ld	% 32b int ld	% fp ld
bison	20.1	79.9	0.0	5.6	22.5	0.0
compress	18.5	81.5	0.0	3.7	16.3	0.0
espresso	0.1	99.9	0.0	0.0	21.7	0.0
gas	29.7	70.3	0.0	6.8	16.2	0.0
gawk	30.7	64.8	4.5	8.3	17.4	1.2
gcc1	9.3	90.7	0.0	2.2	21.5	0.0
gcpp	43.2	56.8	0.0	7.5	9.9	0.0
gdiff	49.8	50.2	0.0	13.7	13.8	0.0
grep	48.9	51.1	0.0	9.1	9.5	0.0
tex	11.5	88.5	0.0	2.6	20.3	0.0

Table 4: Load Type Distribution of Non-Numeric Type Applications

Name	Percent Loads Executed			Percent Instructions Executed		
	% sub32b ld	% 32b int ld	% fp ld	% sub32b ld	% 32b int ld	% fp ld
nettuner	16.1	71.2	12.7	3.4	15.0	2.7
spice3	0.6	54.0	45.4	0.2	20.6	17.4

Table 5: Load Type Distribution of Non-Structured Numeric Type Applications

Name	Percent Loads Executed			Percent Instructions Executed		
	% sub32b ld	% 32b int ld	% fp ld	% sub32b ld	% 32b int ld	% fp ld
fft	0.0	32.8	67.2	0.0	10.4	21.3
lloops	0.0	1.3	98.7	0.0	0.4	30.7

Table 6: Load Type Distribution of Structured Numeric Type Applications

C Benchmarks: Store Distribution

Name	Percent Stores Executed			Percent Instructions Executed		
	% sub32b st	% 32b int st	% fp st	% sub32b st	% 32b int st	% fp st
`bison`	23.4	76.6	0.0	3.8	12.6	0.0
`compress`	16.4	83.6	0.0	1.7	8.7	0.0
`espresso`	0.2	99.8	0.0	0.0	5.2	0.0
`gas`	14.9	85.1	0.0	2.0	11.4	0.0
`gawk`	13.4	82.2	4.4	1.9	11.8	0.6
`gcc1`	5.7	94.3	0.0	0.8	12.9	0.0
`gcpp`	44.6	55.4	0.0	4.0	5.0	0.0
`gdiff`	7.4	92.6	0.0	0.3	4.1	0.0
`grep`	31.2	68.8	0.0	4.0	8.8	0.0
`tex`	4.6	95.4	0.0	0.7	15.5	0.0

Table 7: Store Type Distribution of Non-Numeric Type Applications

Name	Percent Stores Executed			Percent Instructions Executed		
	% sub32b st	% 32b int st	% fp st	% sub32b st	% 32b int st	% fp st
`nettuner`	15.3	79.2	5.5	2.2	11.6	0.8
`spice3`	2.1	9.0	88.9	0.2	0.8	8.0

Table 8: Store Type Distribution of Non-Structured Numeric Type Applications

Name	Percent Stores Executed			Percent Instructions Executed		
	% sub32b st	% 32b int st	% fp st	% sub32b st	% 32b int st	% fp st
`fft`	0.0	64.3	35.7	0.0	10.4	5.8
`lloops`	0.0	0.1	99.9	0.0	0.0	12.2

Table 9: Store Type Distribution of Structured Numeric Type Applications