Cache Performance in Vector Supercomputers

L. I. Kontothanassis[†] R. A. Sugumar[‡]
G. J. Faanes[‡] J. E. Smith[§] M. L. Scott[†]

[†] Computer Science Dept.
University of Rochester
Rochester, NY 14627-0226
{kthanasi,scott}@cs.rochester.edu

[‡] Cray Research Inc.
900 Lowater Rd.
Chippewa Falls, WI 54729
{rabin,gjf}@romulus.cray.com

[§] Dept. of Electrical & Computer Engg.
University of Wisconsin-Madison
Madison, WI 53706
jes@ece.wisc.edu

In Proceedings of Supercomputing 1994

Abstract

Traditional supercomputers use a flat multi-bank SRAM memory organization to supply high bandwidth at low latency. Most other computers use a hierarchical organization with a small SRAM cache and slower, cheaper DRAM for main memory. Such systems rely heavily on data locality for achieving optimum performance. This paper evaluates cache-based memory systems for vector supercomputers. We develop a simulation model for a cache-based version of the Cray Research C90 and use the NAS parallel benchmarks to provide a large scale workload. We show that while caches reduce memory traffic and improve the performance of plain DRAM memory, they still lag behind cacheless SRAM. We identify the performance bottlenecks in DRAM-based memory systems and quantify their contribution to program performance degradation. We find the data fetch strategy to be a significant parameter affecting performance, evaluate the performance of several fetch policies, and show that small fetch sizes improve performance by maximizing the use of available memory bandwidth.

1 Introduction

Time-to-solution has traditionally been the metric that vector supercomputer designs have tried to optimize. As a result they often use high performance memory systems constructed of SRAM and multistage interconnection networks that can deal with the high data rate demands of the applications they are targeted for. Such memory systems provide high performance regardless of data size and memory referencing patterns, but are also a significant portion (often more than half) of the cost of the machine.

Most other types of computer systems are based on hierarchical memory organizations with a high-speed cache, backed by slower DRAM-based main memory. Using DRAM decreases main memory costs, and data caches reduce both memory access latency and main memory bandwidth requirements [6, 10]. The suitability of data caches for vector supercomputers has been the subject of controversy for many years. This controversy revolves around a variety of largely untested claims. For example:

1. Vector workloads do not exhibit enough spatial and temporal locality to make good use of the cache.
2. Caches do not have sufficient bandwidth to match the load bandwidth of the processor.
3. The low latency property of the cache is not as desirable as in scalar processors, because vector code usually has prefetching properties, and therefore can tolerate higher latencies.

There may be some advantages to using a hierarchical memory system in vector supercomputers. Such systems have potential for improving cost/performance and/or for increasing main memory size due to the higher density of DRAMs. Indeed, DRAM-only supercomputers like the Cray M-90 already have a place in the market for applications where performance gains from larger memory sizes offset performance losses due to lower bandwidth and higher latency. Furthermore, vector architectures may be suitable in lower cost, non-supercomputer systems and, in that environment, effective cache/DRAM systems are likely to be essential for achieving lower cost.

[1] This work was supported in part by NSF Institutional Infrastructure grant no. CDA-8822724 and ONR research grant no. N00014-92-J-1801 (in conjunction with the DARPA Research in Information Science and Technology—High Performance Computing, Software Science and Technology program, ARPA Order no. 8930).

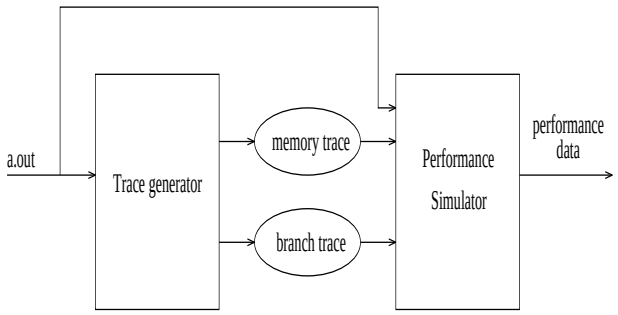

Figure 1: Simulation model

In this paper we examine the effectiveness of caches in (highly-detailed simulations of) vector machines, using medium and large scale vector applications as the experimental workload. We use runtime as the main metric of our studies, because the latency tolerance of vector codes may render cache miss ratios misleading. We focus our study on uniprocessor system performance. We show that a simple cache organization only improves performance marginally (or is worse) over a no-cache DRAM-based system and is much worse than an SRAM-based system. We then show that reducing the fetch size to eliminate memory traffic amplification makes caches a worthwhile addition to a DRAM memory system and in many cases competitive to SRAM memory systems as well.

The rest of the paper is organized as follows. Section 2 describes our experimental methodology and application suite. Section 3 presents our results while section 4 compares our results to those of related work. Section 5 summarizes our conclusions.

2 Experimental methodology

We use trace driven simulation to study the performance of a variety of memory systems including SRAM main memory without a cache, DRAM main memory without a cache, and DRAM main memory augmented with three different cache organizations.

The simulation environment is based on the Crystal[1] tool for generating and interpreting memory reference traces of vector programs and its structure is shown in Figure 1.

The trace generator takes an executable produced by the Cray Fortran Compiler. Because we trace the execution of hundreds of millions of memory references and want complete execution timing information, reducing trace file length is a very important consideration. To do this, the trace generator produces two files: one contains only conditional branch outcomes; the second contains memory references. The vector model allows compaction of the memory reference file because most vector references can be characterized with a base address, a vector length, and a stride.

The performance simulator takes as input the executable and the two trace files. Instruction timing is modeled in detail but the actual instruction computations are not done. This is possible because the C-90's instruction execution times are data-independent, except for memory references and conditional branches, for which we have traces. The performance simulator uses the executable file as a trace for straight-line pieces of code and when it encounters a conditional branch instruction it consults the "branch direction" file to ensure control flow correctness. On memory reference instructions, the addresses accessed are taken from the "memory reference" file.

The CPU component is an accurate (cycle by cycle) simulator of the Cray C-90 processor. The cache component simulates the behavior of a direct-mapped cache with 16 banks and 8 cycles of access latency (A few of our studies consider 2- or 4-way set-associative caches). We have used 128-byte cache-lines (16 64-bit words) and have simulated three different cache sizes: 512Kbytes, 2Mbytes, and 8Mbytes. Depending on the organization under study, it may or may not be possible for individual words within a cache line to be marked invalid. For each memory reference, the cache tag store is first checked. On a hit, the reference goes to the appropriate cache bank. Missing on an absent line forces a line to be allocated for the reference. If a dirty line has to be evicted from the cache to accommodate the new line, all the valid words in the line are written back to memory. The tag store is updated, and all words in the portion of the block being fetched are marked valid and outstanding (in transit from memory to cache). References to words that are not present in the cache (due to misses or unset valid bits) enter a special buffer called the *waiting store* where they wait for the data to return. When a data word returns from memory, the waiting store is checked for references that are waiting, and references matching the fetched word are returned to the CPU. Returning a word to the CPU clears the reservation of the register that was the target of the load instruction which caused the word retrieval. Subsequent instructions that use that register will then be able to proceed without stalling. The CPU continues processing on a miss and does not wait for the miss to be serviced. It stalls only when a subsequent operation tries to use the register that is the target of an uncompleted load. There is no limit on the number of outstanding misses

[1] Crystal stands for CRaY processors SimulaTor and memory AnaLyzer.

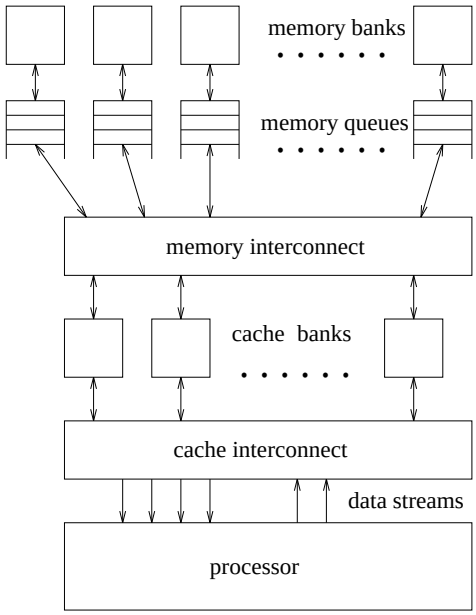

Figure 2: Memory system overview

for a cache bank, however we have a limit on the total number of outstanding misses for the processor so that we can place a limit on the size of the *waiting store*. Figure 2 illustrates the generic memory system that we use for simulations.

For vector processors, memory bandwidth is a crucial performance parameter. Consequently, we model a supercomputer-style highly interleaved main memory. Main memory is interleaved on word boundaries, and the number of banks is varied from 16 to 256. Memory bank conflicts are modeled, but conflicts in the processor-to-memory interconnection network are not. Each bank has an unbounded queue where memory requests to it are sent.

For DRAM memory systems the bank access time is 35 processor cycles and the cycle time is 70. For SRAM memories the corresponding numbers are 5 and 8 processor cycles respectively. Those numbers are in agreement with the speeds of commercial RAM chips and the clock speed of the C-90. Note that the C-90 processor can make up to 6 memory references per clock period (4 reads and 2 writes). This implies that 420 DRAM banks or 30 SRAM banks (512 and 32 respectively when rounded to a power of two) are required to satisfy the peak request rate, assuming no bank conflicts.

For the performance simulations, we selected seven of the eight NAS parallel benchmarks [2]: CG, SP, LU, MG, FFT, IS, and BT. The EP (embarrassingly parallel) benchmark has a very small data set and was considered uninteresting for a memory system study. Of the benchmarks, BT, LU, and SP are full-sized computational fluid dynamics applications; they use iterative techniques to solve partial differential equations. The rest of the applications are best described as kernels. MG is a simple multigrid kernel; CG uses a conjugate gradient method to compute the smallest eigenvalue of a large, sparse, symmetric positive definite matrix; FFT is a 3-D partial differential equation solver using FFT's; and IS is an integer sort program using the "counting sort" algorithm[8]. The benchmark characteristics are summarized in Table 1. The benchmarks range in size from 72 to 448M bytes and contain hundreds of millions to over a billion memory references. We feel that it is necessary to study benchmarks of this size to achieve a realistic characterization of a supercomputer workload. This is one of the major features that differentiates our work from other research on vector data caches. We traced one interation or one call to the main solution routine from the computationally intensive, predominately vector part of each application. Cache hits are therefore solely due to intra-iteration use. Considering the size of the data sets for these programs and the size of the caches we simulate we do not expect to see much inter-iteration reuse.

3 Results

3.1 Performance on a standard cache architecture

We performed an initial set of simulations for all the benchmarks with the following memory systems.

1. *Perfect*: all memory references take one cycle.
2. *Infinite*: this is a cacheless system with infinite bandwidth. The latency for all references is 59 cycles (network round-trip plus DRAM access) and there are no memory bank conflicts. Both *perfect* and *infinite* provide standards for comparison.
3. *SRAM*: a cacheless memory system with SRAM main memory.
4. *DRAM*: a cacheless memory system with DRAM main memory.
5. *64K, 256K, 1M*: cache-based systems with cache sizes of 64K words, 256K words, and 1M words (512K bytes, 2M bytes and 8M bytes) and a DRAM main memory.

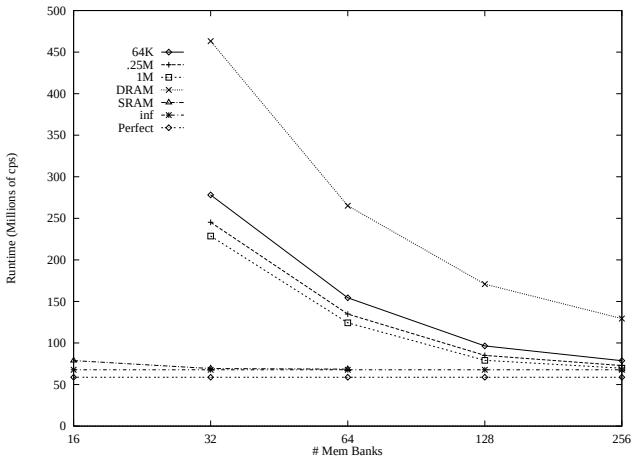

Figure 3: CG execution times for different memory systems

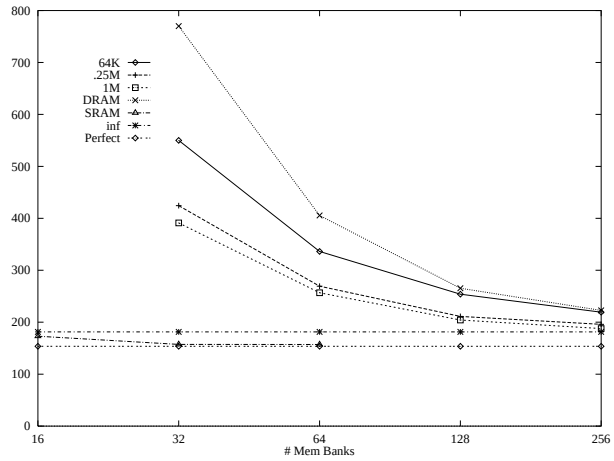

Figure 4: SP execution times for different memory systems

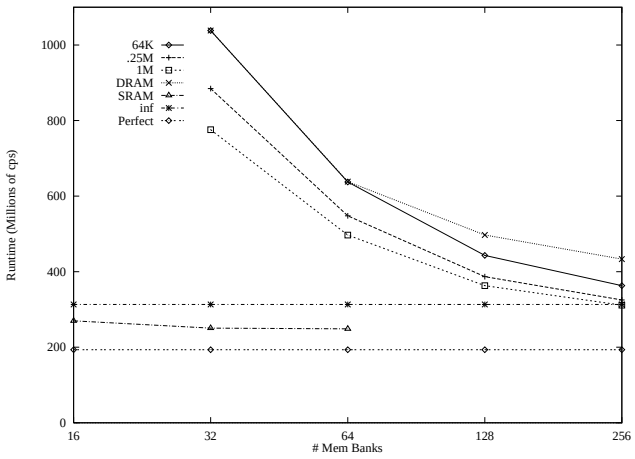

Figure 5: LU execution times for different memory systems

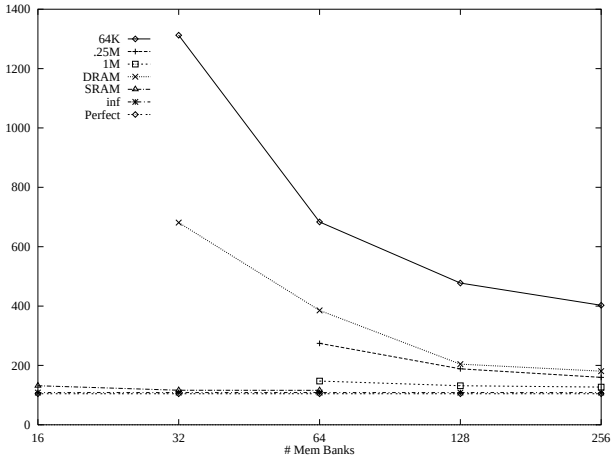

Figure 6: MG execution times for different memory systems

Benchmark	Size Mbytes	Refs $\times 10^6$	Reads $\times 10^6$	Writes $\times 10^6$	Vector $\times 10^6$	Scalar $\times 10^6$	Unit $\times 10^6$	N-Unit $\times 10^6$	Rnd $\times 10^6$
CG	80	182.49	164.58	17.91	169.13	0.44	134.23	0	47.82
SP	72	335.68	242.40	93.28	335.56	0.12	248.31	87.25	0
LU	256	393.75	282.34	111.41	392.83	0.92	70.71	322.12	0
MG	448	304.89	236.74	68.15	304.49	0.40	304.23	0.26	0
FFT	344	532.17	314.55	217.62	531.36	0.81	483.54	47.82	0
IS	248	316.65	199.74	116.91	293.62	1.58	231.19	0	83.88
BT	344	1056.25	778.30	277.95	1055.13	1.12	823.0	232.13	0

Table 1: Application Characteristics when compiled with the Cray Research Parallelizing-Vectorizing Compiler.

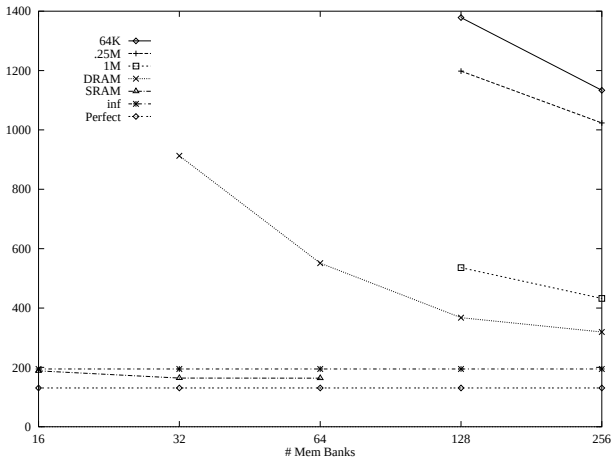

Figure 7: IS execution times for different memory systems

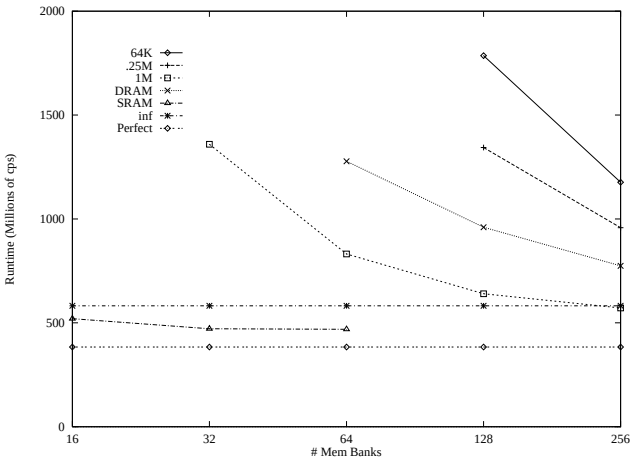

Figure 8: BT execution times for different memory systems

Application	VL	Miss rate		
		64Kw	.25Mw	1Mw
CG	109.8	4.01%	3.57%	3.37%
SP	85.8	2.71%	1.89%	1.66%
LU	48.0	4.53%	3.73%	3.21%
MG	106.4	10.52%	3.05%	1.21%
IS	124.2	23.96%	21.28%	9.12%
FFT	126.7	10.29%	2.00%	0.79%
BT	69.3	8.80%	6.42%	1.88%

Table 2: Average Vector Lengths and Miss rates for the NAS parallel benchmarks

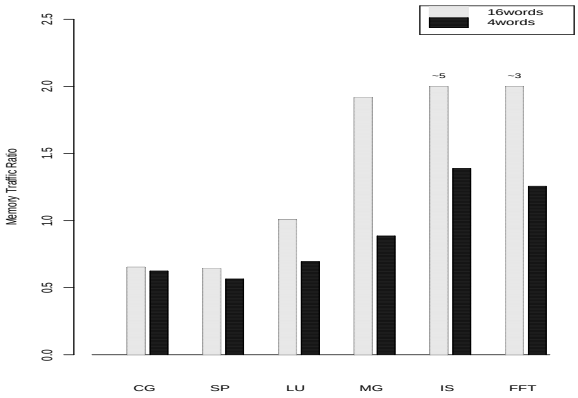

Figure 9: Traffic ratio for 16 and 4 word lines on a 64Kw cache

Figures 3 to 8[2] show the execution times of the benchmarks for the different memory systems as a function of the number of main memory banks. The graphs lead to the following observations.

1. The two flat lines, indicating performance for the *perfect* memory system and the *infinite* bandwidth system are close together in several cases. Their closeness indicates the relative importance of latency for a particular benchmark. Both lines represent unlimited bandwidth systems, but *perfect* has single cycle latency and *infinite* has 59 cycle latency. The performance extremes occur in FFT and MG, where *infinite* and *perfect* performance are almost identical and LU where *infinite* is about 50% slower than *perfect*. The different behavior among applications can be explained by looking at the average vector length for each benchmark (Table 2). Applications with short vectors are more sensitive to latency because they cannot amortize the vector start-up cost over a large number of references. IS deviates from the above generalization. It is relatively sensitive to latency, despite its long vector lengths. Its behavior can be attributed to the relatively high number of scalar references compared to the other benchmarks and the large number of random vector accesses.

2. SRAM performance tends to be bracketed by performance for the *infinite* and *perfect* systems. With 32 banks, SRAM has adequate bandwidth to support the observed memory reference rate. Hence, its performance is similar to the two sys-

[2]We have omitted the graph for FFT due to lack of space and since it is very similar to that for MG

tems with infinite bandwidth, but its performance falls in between the two because its latency falls in between. For 16 bank systems, bandwidth is an issue for some of the benchmarks and the SRAM performance is a little worse than the *infinite* case.

3. DRAM performance is very sensitive to the number of banks. This is due to restricted bandwidth. It takes 420 banks of DRAM to support the maximum demand of the processor. With only 32 banks, performance is typically about 5 to 10 times worse than the unlimited bandwidth memory system. Asymptotically, DRAM performance with a very large number of banks approaches the *infinite* system.

4. When data caches are added to the DRAM systems, the results become very dependent on specific benchmarks. In two of the benchmarks, CG and SP, caches improve performance significantly. Furthermore, for these two benchmarks there are improvements for all cache sizes. With CG, performance for all three caches approach the *infinite* performance with 256 DRAM banks. In SP, this is true of the medium and large caches; the small cache performance is closer to DRAM performance at 256 banks. In other benchmarks, performance with data caches can be worse than with no cache at all, depending on the cache size. The worst case is IS where none of the caches improves performance. In other benchmarks, the large caches give better performance, but the small cache does worse.

Caches can lead to degraded performance because they sometimes amplify memory bandwidth requirements. This occurs because a cache miss results in 16 words (a full line) being fetched from main memory. If some of these words are unused before the line is replaced (due to low spatial locality) then bandwidth was wasted in fetching the unused portion of the line. Non-unit stride or gather/scatter references are an obvious cause of fetches of unused data. Furthermore, this effect will be greater with smaller caches where there is less chance that words in a line will be used before the line has to be replaced.

Figures 9 and 10 show the memory traffic ratios for six of our applications[3] with a 16 word and a 4 word cache line on two different cache sizes. The *memory traffic ratio* is the number of words transferred between main memory and a data cache divided by the number of words transferred in a system without

[3]We were unable to gather the data for BT due to lack of time

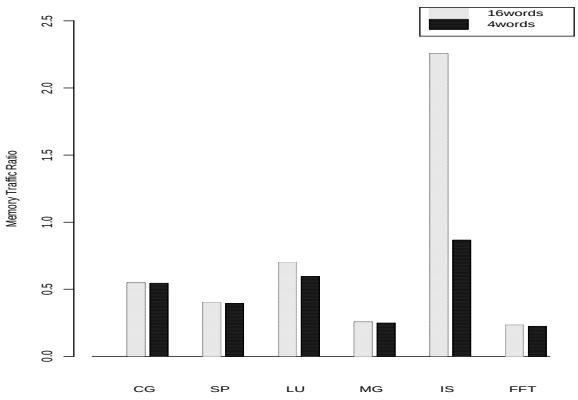

Figure 10: Traffic ratio for 16 and 4 word lines on a 1Mw cache

a cache. Hence, the unit line represents the relative memory traffic for a cacheless system. In all cases memory traffic is less with the smaller line size (with significant reductions for some of the applications). The ramifications of this finding are further discussed in section 3.2.

Application	Miss rate		
	64Kw	.25Mw	1Mw
CG	60.30%	56.13%	53.68%
SP	36.23%	29.39%	25.92%
LU	36.03%	32.10%	30.73%
MG	65.15%	46.06%	18.87%
IS	66.11%	64.15%	44.94%
FFT	79.25%	30.45%	6.51%
BT	45.57%	38.95%	27.75%

Table 3: Average Miss rates for the NAS parallel benchmarks using the single word fetch strategy

3.2 Line and fetch size effects on performance

In the previous section we saw that using long cache lines can sometime cause memory traffic amplification due to the fetching of unused data. While the prefetch effect of long cache lines may be desirable on a microprocessor based system with limited abilities to tolerate memory latency, it is of limited use in the types of machines we are examining. Vector machines with sophisticated compiler support can tolerate latencies quite well; the main benefit of a cache is to reduce the main memory bandwidth requirements.

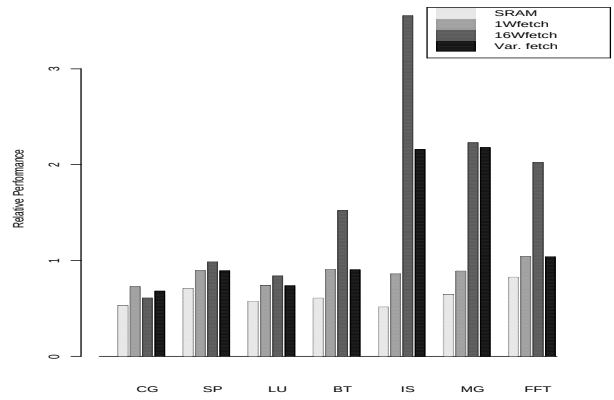

Figure 11: Normalized execution time under different fetching policies for a 64Kw cache

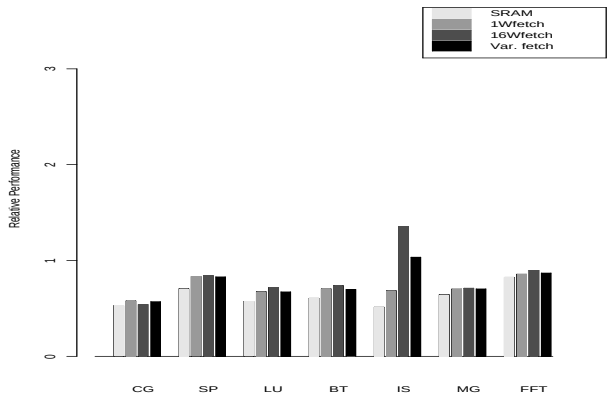

Figure 12: Normalized execution time under different fetching policies for a 1Mw cache

This would tend to indicate that single word lines should be used. Unfortunately such an approach leads to a high overhead because each word of data in the cache must be accompanied by a tag—an overhead of 30 to 50%. An alternative solution is to use a sector cache [9], in which a line is divided into a number of sectors, each with a valid bit. Tags are maintained for lines, but only the data belonging to a sector is fetched on a miss. Any sector fetched in this way is marked valid; other sectors are invalid. The type of sector cache we are interested in uses single word sectors—i.e. there is a valid bit per word. Smaller fetch sizes will decrease the amount of memory traffic but will also decrease spatial locality hits, thus increasing average memory access latency. The latency tolerances of vector computers tips the balance in favor of the reduced memory traffic. A more elaborate design variant exploits variable fetch sizes by taking advantage of vector stride information available in the vector instruction causing a miss. We can use this information to invoke a full line fetch for unit stride accesses and single word fetches for all other references.

Figures 11 and 12 show normalized execution times for the seven benchmarks under the three different fetching policies (*single word fetch, whole line (16 words) fetch, and variable fetch*) for 64Kw and 1Mw caches with a 256 bank DRAM main memory. Execution time with an SRAM memory system is also provided for comparison purposes. The unit line represents the execution time of a cacheless DRAM memory system. Miss rates for the *single word fetch* policy are shown in table 3.

The *single word fetch* policy is the most consistent at improving application performance. It is interesting to note that better performance is associated with the higher miss ratios as can be seen from tables 2 and 3. The explanation is simple; the cost of a miss is radically different across the *single word fetch* and *whole line fetch* organizations. Whole line fetches cost 16 times as much as single word fetches and low miss rates do not necessarily translate to less memory traffic. CG is the only application that performs better with a whole line fetch policy. This is surprising because CG has a fairly high proportion of gather/scatter references. Closer inspection of the benchmark reveals that CG still has good spatial locality; that is, its cache "working set" is small.

The variable fetch policy tries to take advantage of unit stride accesses but the benefit is small due to the latency tolerance property of the code. Furthermore in cases of pathological conflicts (as is the case in some of the 64K cache experiments) fetching a whole line can be detrimental to performance even for unit stride accesses, since the line may be replaced due to conflicts before it is fully used.

The improved performance with small fetch sizes is in conflict with the result obtained by Fu and Patel [4] in a similar study. They found that large fetch sizes were desirable and significantly improved performance. The reason for this difference is the type of processor simulated. Fu and Patel assumed that processors stall on a cache miss waiting for the data to return. Under such assumptions spatial locality becomes crucial to performance and the tradeoff of memory traffic to miss ratio is resolved in the opposite way.

In this study we have not taken into account page or nibble mode DRAMs. For such systems that can pro-

vide multiple successive words at low cost, larger sector sizes may be preferable. The preferred sector size will depend on cache parameters like size and associativity and application/compiler properties like spatial locality and data conflicts. If lines are replaced before the words in a sector can be used, large sector sizes will provide no performance benefits.

3.3 Sources of memory overhead

We have identified three categories that contribute to degraded memory performance.

- *Overall limited bandwidth.* Performance is degraded because more memory references are issued from the processor than the memory system can handle, even if the references are evenly distributed among the banks.
- *Memory bank conflicts due to uneven reference patterns.* In this case there are some "hot" memory banks. So while the memory system as a whole may be underutilized, references that access the same bank may be delayed because of bank conflicts.
- *Standard memory latency.* Performance is degraded because of the amount of memory latency even in the absence of bank contention.

The addition of a cache doubles the number of categories to six. The extra three categories are identical to the ones presented above but apply to the cache subsystem.

Figure 13 shows the execution breakdown of four of our benchmarks into processor time and the different types of memory overhead we have identified above. The bars from left to right represent the runtime for a no-cache DRAM memory system with 128 banks, a 1Mw cache system with 128 banks of main memory, a no-cache DRAM system with 256 banks, and a 1Mw cache system with 256 banks of main memory. The different overhead categories are: `BC` for bank conflicts, `BW` for bandwidth, and `lat` for latency. The prefix `M-` or `C-` signifies whether the overhead is due to the cache or the memory subsystem.

As can be seen caches reduce the amount of performance loss in the memory system for all applications, albeit for different reasons. In CG, all three overheads in the cacheless system are reduced when going to a cached system. The most dramatic improvements are in the memory bank conflict areas, although latency is reduced as well. We believe that the unit stride references of CG benefit from the increase in bandwidth while the random access and scalar references (of which CG has a fair number) benefit mostly from the reduction in latency. When going from a cacheless 128 bank system to a cacheless 256 banks system, overall bandwidth limitations are relieved, but losses due to hot banks remain virtually unchanged.

In SP, the biggest performance improvements are due to latency reductions. However, in the 128 bank system there are some improvements due to better bandwidth. The LU characteristics are similar to SP; i.e. the biggest improvements are due to latency reductions. The results for LU and SP are consistent with the observation that these two applications have a large number of short vector operations (mostly LU) that are more sensitive to latency.

In MG, both better bandwidth and latency contribute to performance improvement. Because bandwidth overhead is higher in the cacheless system, the overall improvement due to better bandwidth is higher than the improvement due to lower latency.

4 Related work

Vector caches have been studied previously by a number of researchers using miss ratios as the main performance metric [1, 3, 11]. Our work reports miss ratios similar to those observed by those studies, (i.e. predominantly unit stride applications have lower miss ratios than applications with a lot of non-unit stride references) but also provides the correlation of miss ratios with run time which is the ultimate system performance metric.

Gee and Smith [5] also report run time results for vector caches but with a significantly different workload and system architecture. They compare caches with a limited bandwidth flat memory (8 or 16 way interleaving only) and use applications that have small data sets and predominantly unit strides. Their environment is therefore more "cache-friendly" than ours, and they do not observe the memory traffic amplification phenomenon that appears in our experiments. Furthermore since their flat memory has considerably less bandwidth than the ones we consider, the reported benefits due to the addition of the cache are much higher than the ones we observe. The issue of cache fetch sizes has also been looked at by Fu and Patel [4] (see the discussion in section 3.2).

Hsu and Smith [7] study cached DRAMs for vector multiprocessors. They show that a shared cache at the memory can increase effective memory bandwidth by factors of about two to four. We recommend further studies to evaluate the tradeoff between shared and per-processor caches.

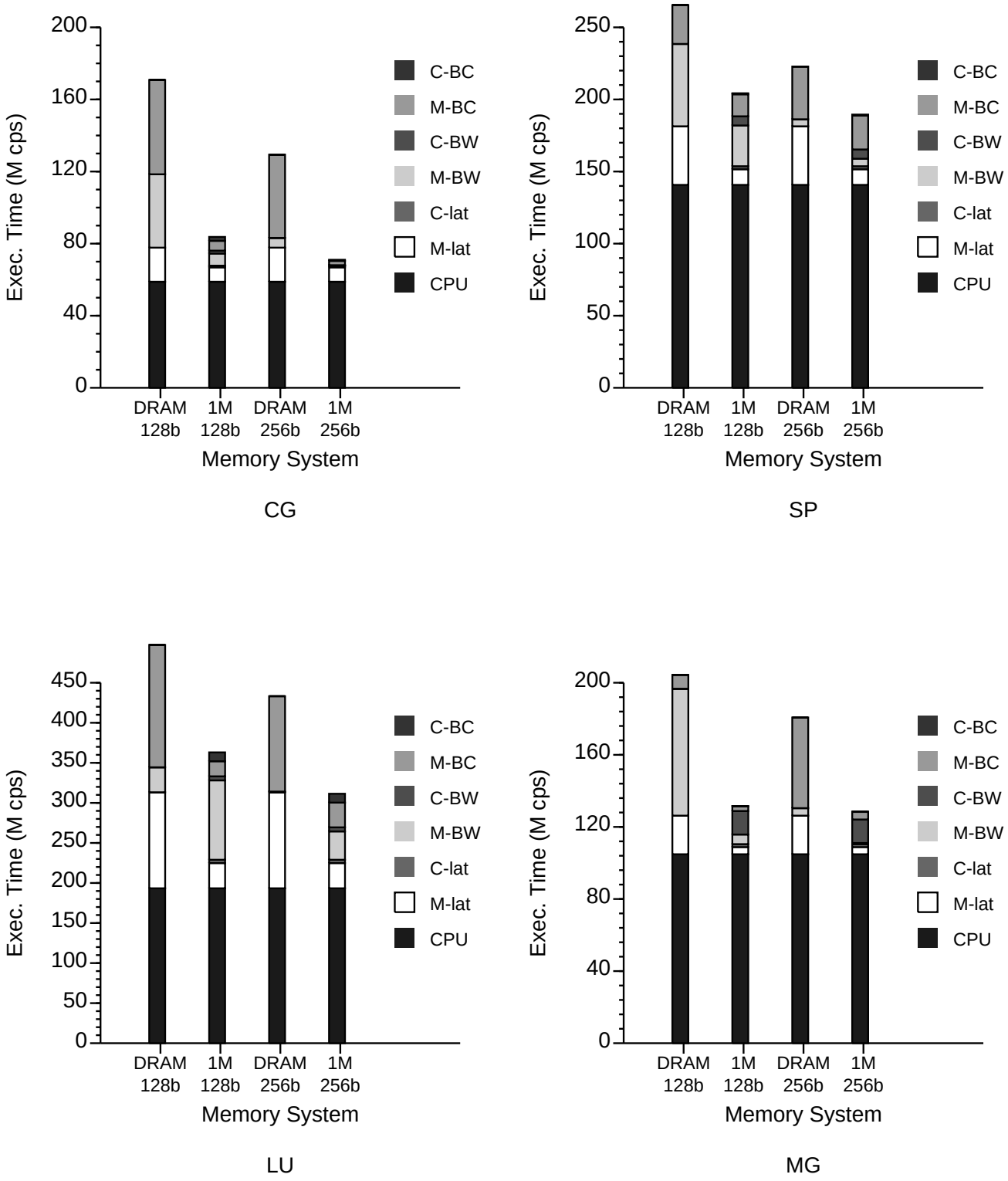

Figure 13: Execution time breakdown for system with and without caches

5 Conclusions

In this paper we have looked at the performance impact of caches and cache fetching strategies in vector processors using detailed simulations of state of the art vector machines. We have shown that SRAM memory systems provide the best overall performance with a relatively small number of memory banks. Their use in supercomputer-class applications where time-to-solution is at a premium appears to be justified. For some of the benchmarks, performance with a data cache plus DRAM comes close to SRAM performance, but about eight times as many DRAM memory banks are required to do so.

For systems where cost/performance is the main goal, rather than time-to-solution, lower cost systems based on DRAM main memory and local caches can help mitigate the two big performance problems with DRAM systems: relatively high latency and low bandwidth. Despite the unfriendliness of the benchmark suite toward cache memories we find that we can obtain competitive performance using a small fetch size (1 word) on every miss. The latency tolerating property of vector codes allows us to trade a higher miss ratio for a reduction in main memory traffic. We believe that caches can be used to provide cost/effective memory systems for vector processors, especially with the maturing of compilers that may allow applications to exploit higher cache reuse.

Acknowledgements

We would like to thank Ram Gupta for his helpful insights and comments on this paper.

References

[1] W. Abu-Sufah and A. D. Malony. Vector Processing on the Alliant FX/8 Multiprocessor. In *International Conference on Parallel Processing*, pages 559–566, August 1986.

[2] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS Parallel Benchmarks. Report RNR-91-002, NASA Ames Research Center, January 1991.

[3] R. S. Clark and T. L. Wilson. Vector System Performance on the IBM 3090. *IBM System Journal*, 25(1):63–82, 1986.

[4] J. W. C. Fu and J. H. Patel. Data Prefetching in Multiprocessor Vector Cache Memories. In *Proceedings of the Eighteenth International Symposium on Computer Architecture*, pages 54–63, Toronto, Canada, May 1991.

[5] J. D. Gee and A. J. Smith. The Performance Impact of Vector Processor Caches. *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, 1:437–449, January 1992.

[6] J. R. Goodman. Using Cache Memory to Reduce Processor/Memory Traffic. In *Proceedings of the Tenth International Symposium on Computer Architecture*, pages 124–131, June 1983.

[7] W.-C. Hsu and J. Smith. Performance of Cached DRAM Organizations in Vector Supercomputers. In *Proceedings of the Twentieth International Symposium on Computer Architecture*, San Diego, CA, May 1993.

[8] D. E. Knuth. *Sorting and Searching*, volume 3 of The Art of Computer Programming. Addison-Wesley, Reading, MA, 1973.

[9] J. S. Liptay. Structural Aspects of the System/360 Model 85, Part II: The Cache. *IBM Systems Journal*, 7(1):15–21, 1968.

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

[11] K. So and V. Zecca. Cache Performance on Vector Processors. In *Proceedings of the Fifteenth International Symposium on Computer Architecture*, pages 261–268, Honolulu, HI, June 1988.