RC 21852 (98345) 6 October 2000 **Computer Science**

IBM Research Report

Memory Behavior of the SPEC2000 Benchmark Suite

Suleyman Sair, Mark Charney
IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g. , payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home .

Memory Behavior of the SPEC2000 Benchmark Suite

Suleyman Sair　　Mark Charney

IBM T.J. Watson Research Center

Yorktown Heights, NY 10598

ssair@cs.ucsd.edu　　mark_charney@us.ibm.com

Abstract

The SPEC CPU benchmarks are frequently used in computer architecture research. The newly released SPEC'2000 benchmarks consist of fourteen floating point and twelve integer applications.

In this paper we present measurements of number of cache misses for all the applications for a variety of cache configurations. Prior studies have shown that SPEC benchmarks do not put much stress on the memory system. Our simulation results demonstrate that SPEC'2000 places only modest pressure on the first level caches confirming the results of similar experiments.

1 Introduction

SPEC CPU benchmarks have long been used to gauge the performance of uniprocessor systems as well as micro-architectural enhancements. The newly released SPEC'2000 benchmark suite replaced the previous release, SPEC'95.

Many studies [1, 3, 4]showed that only a few applications place more than modest stress on the memory system. The purpose of this study is to examine the memory behavior of the SPEC'2000 benchmark suite and determine how it compares to earlier releases of SPEC benchmarks.

Table 1 and Table 2 [2] briefly summarize the SPEC'2000 CFP2000 floating point and CINT2000 integer benchmarks respectively. Memory footprint size for each application is also provided [5].

The rest of the paper is organized as follows. In section 2, a description of prior similar studies can be found. Section 3 details the SPEC'2000 profile information we gathered. Simulation methodology and benchmark descriptions can be found in Section 4. Section 5 presents the results for this study, and our conclusions are summarized in section 6.

Program	Language	Resident Size in MB	Description
wupwise	F77	176	Physics / Quantum chromodynamics
swim	F77	191	Shallow water modeling
mgrid	F77	56	Multi-grid solver: 3-D potential field
applu	F77	181	Parabolic / Elliptic partial differential equations
galgel	F90	63	Computational fluid dynamics
art	C	3.7	Image recognition / Neural networks
equake	C	49	Seismic wave propagation simulation
facerec	F90	16	Image processing: Face recognition
ammp	C	26	Computational chemistry
lucas	F90	142	Number theory / Primality testing
fma3d	F90	103	Finite-element crash simulation
sixtrack	F77	26	High energy nuclear physics accelerator design
apsi	F77	191	Meteorology: Pollutant distibution

Table 1: Benchmark descriptions and resident memory size for CFP2000 programs.

Program	Language	Resident Size in MB	Description
gzip	C	181	Compression
vpr	C	50	FPGA circuit placement and routing
gcc	C	155	C progaramming language compiler
mcf	C	190	Combinatorial optimization
crafty	C	2.1	Game playing: Chess
parser	C	37	Word processing
eon	C++	0.7	Computer visualization
perlbmk	C	146	PERL programming language
gap	C	193	Group theory, interpreter
vortex	C	72	Object-oriented database
bzip2	C	185	Compression
twolf	C	1.9	Place and route simulator

Table 2: Benchmark descriptions and resident memory size for CINT2000 programs.

2 Related Work

Similar studies have been performed on earlier versions of the SPEC benchmark suite. It is customary to test memory hierarchy designs and optimizations targeting the memory subsystem on the SPEC benchmark suite. Therefore, many similar studies have been performed on earlier versions of these benchmarks.

Pnevmatikatos and Hill [7] looked at a subset of the SPEC'89 integer benchmarks in the context of a RISC processor. They inspected the available cache locality in these applications. Gee et al. [3] studied the SPEC'89 benchmark suite as well, reporting cache miss ratios. Later on, they extended this study for the SPEC'92 benchmarks [4]. They concluded that SPEC benchmarks may not represent actual performance of a time-shared, multi-programming system with operating system interference. This is due to each SPEC benchmark running as the single active user process until completion. Lebeck and Wood [6] used their CPROF cache profiling tool to analyze the cache bottlenecks on the SPEC'92 benchmark suite. CPROF provides cache hot-spot information at the source line and data structure level. This information is then used by the programmer to modify the code to improve the program's locality.

Charney and Puzak [1] repeated this study for the SPEC'95 benchmarks. They reported results in *misses per instruction* (MPI) for several reasons. MPI is a direct indication of the amount of memory bandwidth that must be supported for each instruction. Moreover, given the average memory cycles per cache miss, it is straightforward computing the memory component of the cycles per instruction. MPI is the metric we chose to report in this study. Besides cache analysis, Charney and Puzak studied the prefetching behavior of SPEC'95 as well.

Sherwood and Calder [9] looked at the cache behavior of SPEC'95 programs over their course of execution, analyzing the interaction between cache performance, IPC, branch prediction, value prediction, address prediction and reorder buffer occupancy. They found out that the large scale behavior of programs is cyclic in nature and pointed out where to simulate to achieve representative results.

Recently, Thornock and Flanagan [10] analyzed the SPEC'2000 integer benchmarks using the BACH trace collection mechanism. BACH is a hardware hardware monitor that enables the acquisition of trace data. They gathered traces for the first 100 million integer references and ran them through their cache simulators. Along with looking at only the integer benchmarks, they analyzed only a single input for multi-input programs.

3 Profile Information

During the simulations, we skipped the initialization part of each benchmark. In order to determine the fast forwarding amount, we profiled the applications, gathering statistics such as execution frequency, number of instruction and data cache misses as well as TLB misses caused by each basic block. Moreover, we recorded the number of

instructions committed until that basic block is seen for the first time.

The instruction and data caches used for this profile were 4K direct-mapped with 32 byte lines. We used a 2-way, 256 entry TLB. The page size was set to 4K.

Table 3 and Table 4 show the simulation results for the CFP2000 and CINT2000 benchmark suites respectively. The tables present the cumulative percentage of instructions executed, instruction and data cache misses, along with data TLB misses for the most frequently executed 10 basic blocks.

Data cache and TLB misses exhibit a uniform behavior for most applications. The number of misses is directly proportional to the relative size of most frequently executed basic blocks. There are only a few instruction cache misses however, suggesting good temporal locality–at least during the execution of these basic blocks.

It is interesting to note that an application running on different inputs may exhibit significantly different behavior. Some applications, such as gzip, have randomly generated inputs which exhibit significantly different behavior, e.g. extremely high miss rates, when compared to the reference inputs. Another example, vpr, is a placement and routing tool which has two inputs, one for routing and one for placement. Simulation results of these two inputs show that they exercise different parts of the application.

We then analyzed the basic block information to determine a window of 500 million contiguous instructions that would be similar to the full run in execution behavior. In order to make sure a representative window was selected, we verified that the number of cache misses generated by the shorter run closely matched those of the full run. We also tried to preserve the relative amount of time spent in the most frequently executed basic blocks. Once we determined that window, the fast forwarding amount is chosen as the number of instructions preceding the first basic block of the window. These results are shown in tables 5 and 6 .

4 Methodology

The simulator used in this study is an in-house IBM tool, Aria. Aria is an execution driven simulator, similar in nature to ATOM, written for the IBM PowerPC architecture.

The SPEC'2000 applications were compiled on a processor using the IBM C and C++ compilers under AIX 4.3 operating system using full compiler optimization (`-O2 -qarch=rs64c`). Tables 5 and 6show the number of instructions simulated and the number of instructions fast forwarded before gathering statistics.

Each application was run on all the reference inputs provided. Results for the different inputs are presented with the input an concatenated to the application name.

In order to increase the simulation speed, we utilized *trace stripping* [8]. With trace stripping, we filtered the reference stream with the help of four 4K direct mapped caches. These caches had different line sizes: 32, 64, 128

Program	% Inst	% I-Miss	% D-Miss	% DTLB-Miss
wupwise	38.45	0	86.46	93.40
swim	98.88	0.26	99.66	99.48
mgrid	82.13	1.35	91.90	58.18
applu	26.33	0	0.27	0
galgel	49.75	0.12	64.49	56.67
art	49.17	0	45.87	68.67
equake	54.18	0	67.58	58.91
facerec	36.03	0	31.31	7.36
ammp	22.05	0.01	49.62	37.06
lucas	69.98	0.01	35.03	14.57
fma3d	19.41	0.03	11.83	4.44
sixtrack	88.02	0.20	60.67	2.31
apsi	24.38	6.69	13.70	0

Table 3: Floating point benchmark profile information for the most frequently executed 10 basic blocks.

Program	% Inst	% I-Miss	% D-Miss	% DTLB-Miss
gzip-graphic	20.80	2.54	33.61	0.38
gzip-log	42.96	3.15	68.98	45.78
gzip-program	70.60	2.10	74.79	47.97
gzip-random	25.14	2.87	19.83	0.16
gzip-source	57.18	2.73	71.03	47.12
vpr-place	14.98	0	22.70	12.07
vpr-route	41.08	0	41.20	42.57
gcc-166	72.56	0	70.36	1.96
gcc-200	36.50	0	30.90	0.97
gcc-expr	49.95	0.01	49.81	0.54
gcc-integrate	62.79	0.01	61.50	0.89
gcc-scilab	40.42	0	39.42	1.45
mcf	53.08	0	62.93	26.80
crafty	12.25	4.46	3.71	0.28
parser	30.62	38.28	21.29	2.14
eon-cook	18.63	4.36	14.48	0.73
eon-kajiya	19.33	4.58	19.04	9.55
eon-rushmeier	20.20	5.88	18.26	15.61
perlbmk-2	20.03	4.08	13.01	24.04
perlbmk-850	38.89	3.74	0.18	10.99
perlbmk-b	23.41	4.24	31.32	3.81
gap	20.89	7.32	27.32	8.68
vortex1	38.64	1.73	33.60	24.55
vortex2	27.20	3.50	33.55	21.65
vortex3	38.33	1.73	33.59	24.31
bzip2-graphic	35.88	0.01	4.22	0.11
bzip2-program	30.85	0.02	3.29	0.07
bzip2-source	16.49	0.01	23.48	5.32
twolf	16.79	2.70	1.05	0

Table 4: Integer benchmark profile information for the most frequently executed 10 basic blocks.

Program	# inst in millions	# skipped in millions
wupwise	500	2500
swim	500	250
mgrid	500	5
applu	500	500
galgel	500	1250
art	500	2900
equake	500	3400
facerec	500	600
ammp	500	2000
lucas	500	2000
fma3d	500	3000
sixtrack	500	1500
apsi	500	30

Table 5: Number of instructions executed and amount of instructions skipped for each CFP2000 program during simulations.

Program	# inst in millions	# skipped in millions
gzip	500	40
vpr-place	500	190
vpr-route	500	1150
gcc	500	1000
mcf	500	4000
crafty	500	10
parser	500	250
eon	500	3
perlbmk	500	500
gap	500	65
vortex	500	0.5
bzip2	500	200
twolf	500	400

Table 6: Number of instructions executed and amount of instructions skipped for each CINT2000 program during simulations.

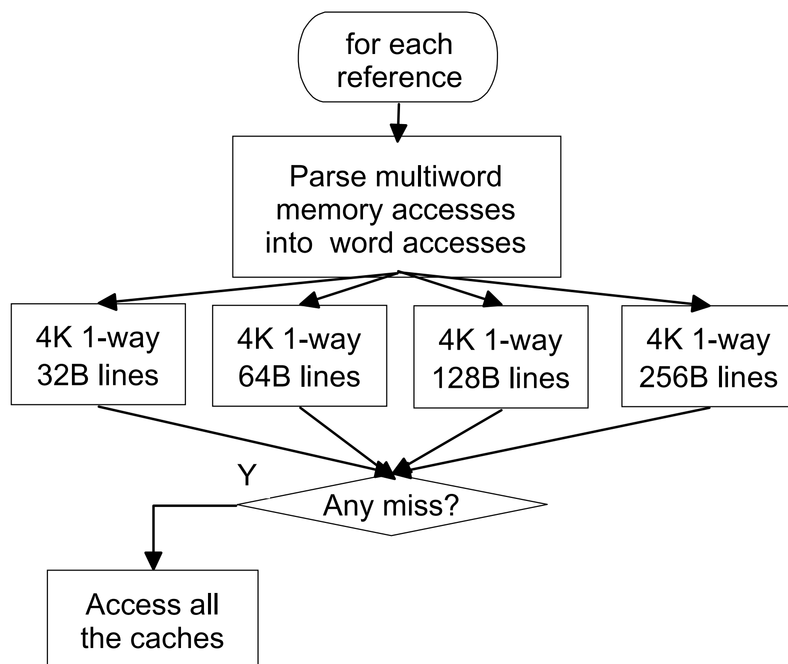

Figure 1: Simulation methodology.

and 256 bytes. Only the misses from these four caches were relayed to the larger simulated caches. The cache hits to these small caches are guaranteed to be cache hits in the larger caches. The smaller direct mapped cache acts as the MRU set of the larger n-way associative cache. By eliminating the hits which form a considerable chunk of the reference stream, we were able to speed up the simulation process significantly. Since not all references access the simulated caches, accurate miss rate numbers can not be obtained. Instead, misses-per-instruction data is provided.

Since certain PowerPC instructions access multiple cache blocks, they can cause multiple misses. Therefore, it is necessary to parse a multi-block memory reference into multiple single-block accesses. This is shown in Figure 1.

5 Cache Behavior of SPEC'2000

This section presents the main results of this paper. Instruction, data and unified cache behaviors are examined under varying cache parameters.

The different cache parameters that we looked at are:

- Capacity: 4K, 8K, 32K, 64K, 128K, 256K
- Associativity: 1-way, 2-way, 4-way, 8-way
- Line size: 32 bytes, 64 bytes, 128 bytes, 256 bytes
- Type: Instruction, Data, Unified

5.1 Instruction Cache

In this section, we examine the instruction cache behavior of the SPEC'2000 benchmarks while varying cache capacity, associativity, and line size.

5.1.1 Varying Capacity

For the floating point benchmarks, in order to achieve fewer than one miss per 1000 instructions, a 32K direct-mapped cache with 32B lines seems to be sufficient. For the CINT2000 instruction references, a 64K 4-way cache with 32 byte lines brings the average number of misses per 1000 instructions to under one. For a 32K direct-mapped cache, the integer benchmarks have on average 9.81 misses per 1000 instructions.

Figures 2 and 3 show the results for several direct-mapped caches with 32 byte lines. The y-axis represents the number of misses per 1000 instructions. Each curve corresponds to a cache capacity ranging from 4K to 256K. The

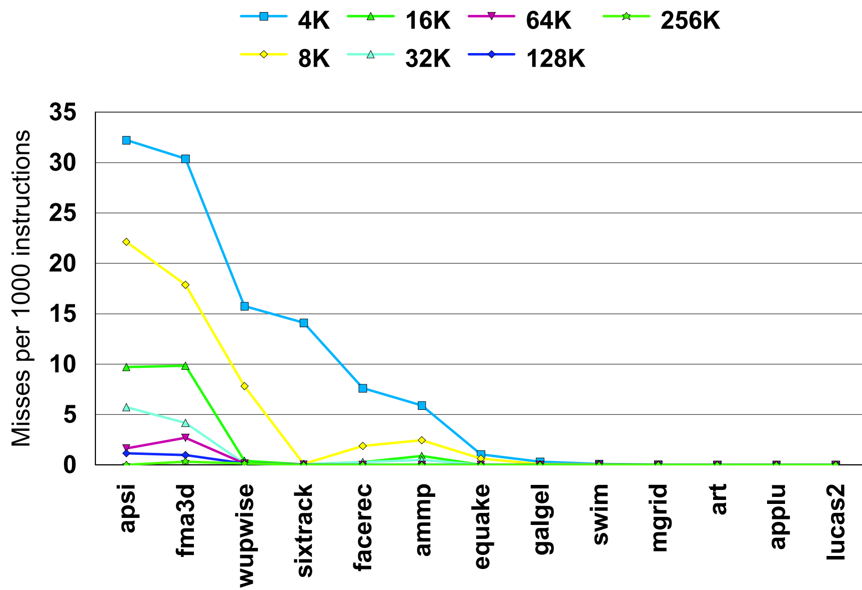

Figure 2: Varying instruction cache capacity for floating point programs . All caches are direct-mapped and have 32 byte lines. The results are sorted with respect to number of misses generated by a 4K cache.

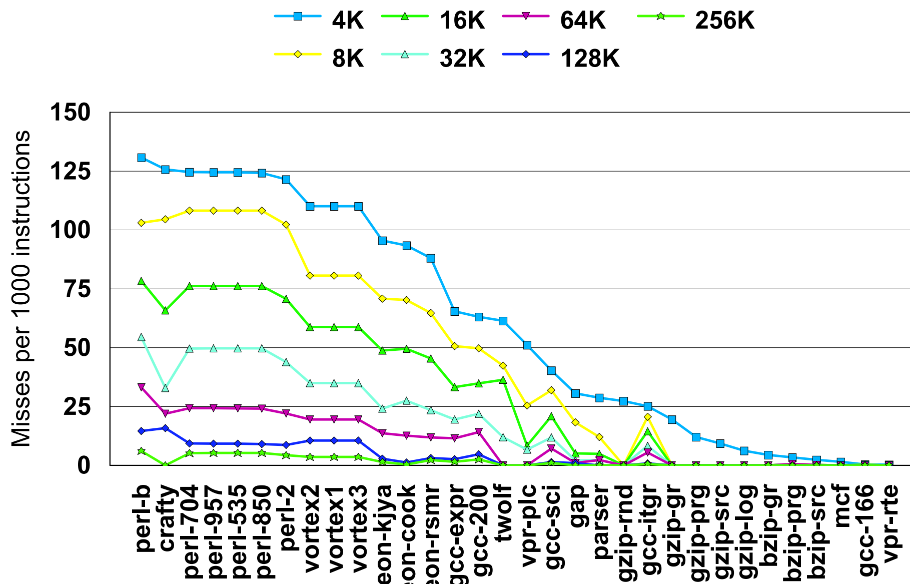

Figure 3: Varying instruction cache capacity for integer programs . All caches are direct-mapped and have 32 byte lines. The results are sorted with respect to number of misses generated by a 4K cache.

first set of applications on the x-axis is the floating point benchmarks while the second set consists of the integer benchmarks. Similar charts are used throughout the paper to present simulation results. Each set is sorted with respect to the number of misses generated by the 4K cache.

As one would expect, floating point applications have fewer instruction cache misses than integer benchmarks. With the inherent looping behavior of these applications most have fewer than 1 miss per 1000 instructions with a 16K direct-mapped cache. Only apsi and fma3d have higher miss numbers with this cache(about 10 misses per 1000 instructions). When we look at a 32K 4-way cache with 32B lines, these two applications go below the one miss per 1000 instructions mark as well.

When we analyze the integer benchmarks, perl, crafty, vortex, eon, and gcc are worth a closer look. A 128K 4-way or a 256K 2-way cache with 32 byte lines is required to reduce the number of misses per 1000 instructions to below one for these applications. For a 4K direct-mapped cache, perl, crafty, vortex and eon, approximately one out of every ten instruction references is a miss.

5.1.2 Varying Associativity

The simulation results for several 32K caches with 32 byte lines are presented in Figures 4 and 5. The different curves represent a particular associativity ranging between one(direct-mapped) and eight.

Floating point applications have less than one (0.84) miss per 1000 instructions with a 32K direct-mapped cache. For the integer benchmarks on the other hand, on average, every 18.6 out of 1000 instructions cause a cache miss.

Floating point benchmarks exhibit three distinct behaviors when associativity is changed: 1) Applications like art, lucas2, applu, equake remain fairly insensitive to associativity. 2) Applications such as apsi, fma3d, sixtrack and mgrid show significant reductions in miss numbers as each time associativity is increased. 3) ammp, facerec, wupwise, galgel have fewer misses when the associativity is increased from one to two. From then on, the miss numbers do not show any noticeable change.

The integer benchmarks mostly benefit from increased associativity. On average the number of misses per 1000 instructions are 18.6, 11.6, 7.95 and 4.3 respectively for a 1-way, 2-way, 4-way and 8-way set associative 32K cache. Vortex shows little benefit when an 8-way set associative cache is used instead of a 4-way cache. For perl every added degree of associativity helps as indicated by the fall in miss numbers. The different behaviors exhibited by the same application on different inputs can be seen for vpr. With a 32K direct-mapped cache with 32 byte lines, when doing placement, there are 6.7 misses per 1000 instructions, whereas while doing routing there are practically no misses (0.0006).

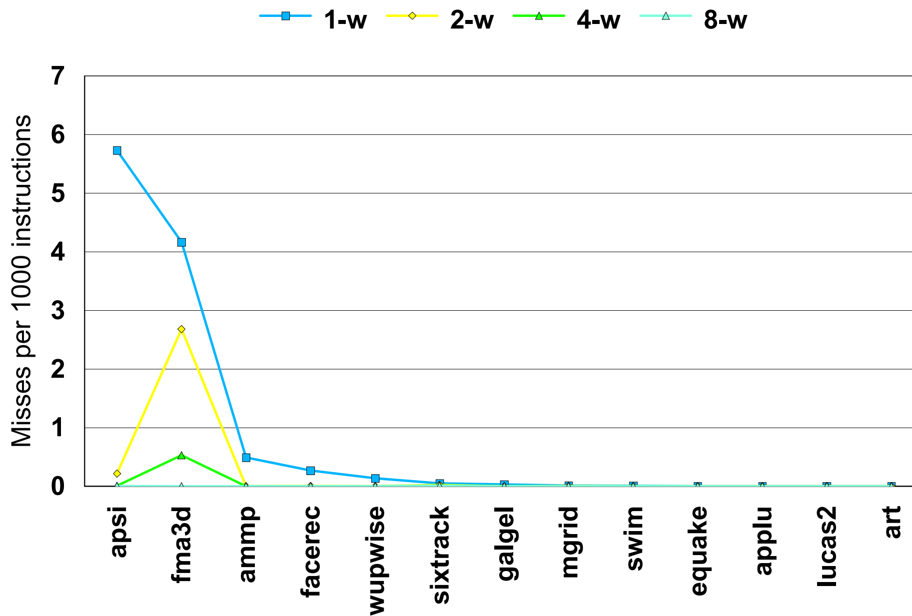

Figure 4: Varying instruction cache associativity for floating point programs . All caches are 32K and have 32 byte lines. The results are sorted with respect to number of misses generated by a direct-mapped cache.

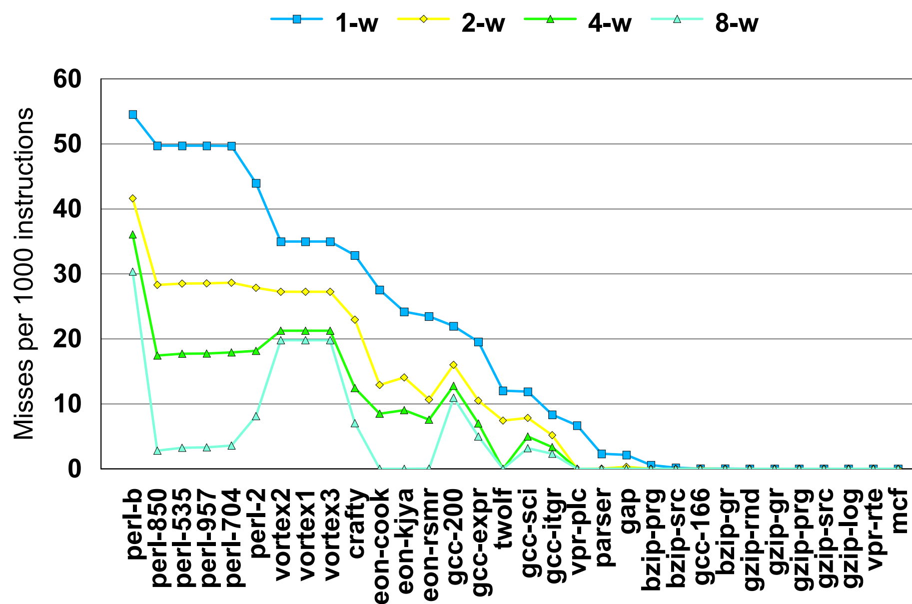

Figure 5: Varying instruction cache associativity for integer programs . All caches are 32K and have 32 byte lines. The results are sorted with respect to number of misses generated by a direct-mapped cache.

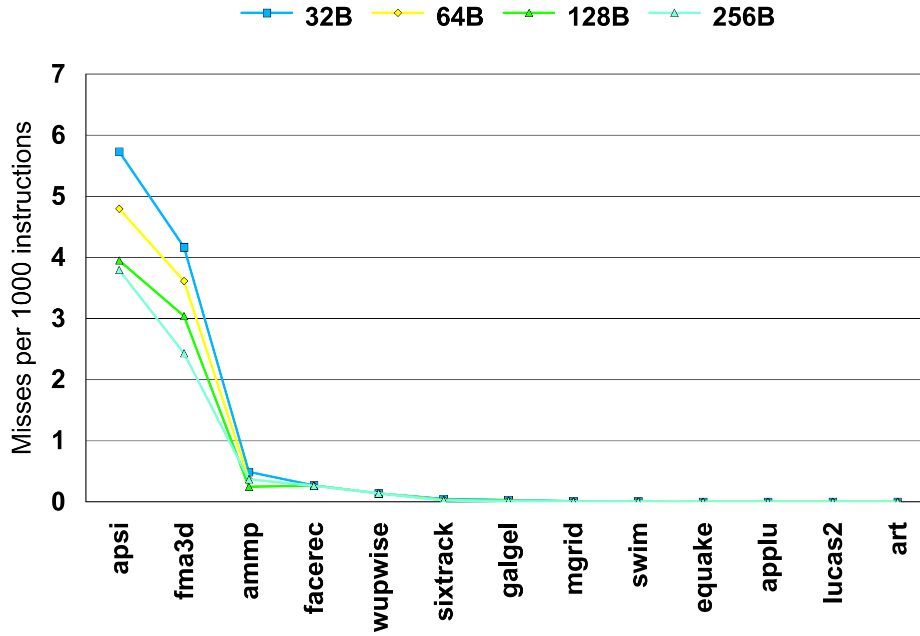

Figure 6: Varying direct-mapped instruction cache line size for floating point programs . All caches are 32K and direct-mapped. The results are sorted with respect to number of misses generated by the cache with 32 byte lines.

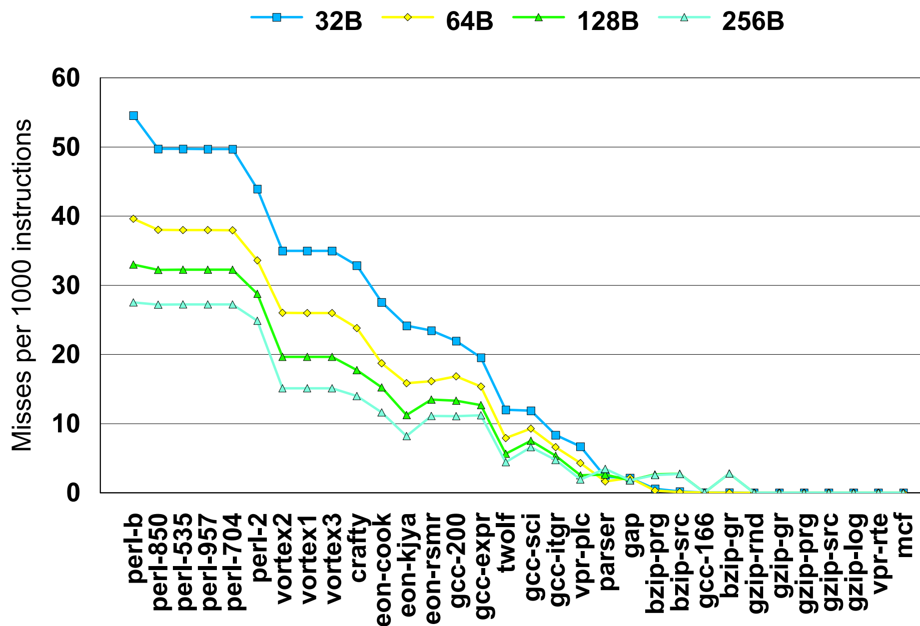

Figure 7: Varying direct-mapped instruction cache line size for integer programs . All caches are 32K and direct-mapped. The results are sorted with respect to number of misses generated by the cache with 32 byte lines.

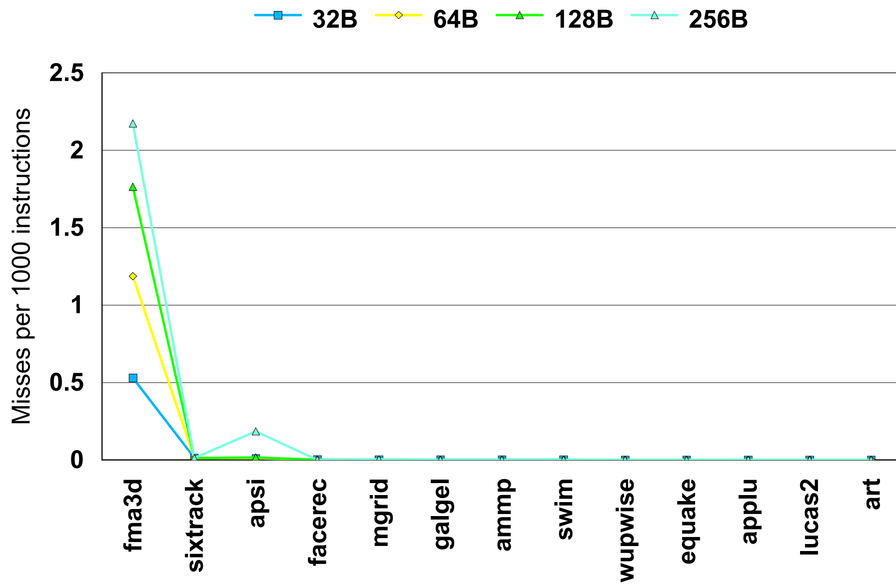

Figure 8: Varying 4-way instruction cache line size for floating point programs . All caches are 32K and 4-way set associative. The results are sorted with respect to number of misses generated by the cache with 32 byte lines.

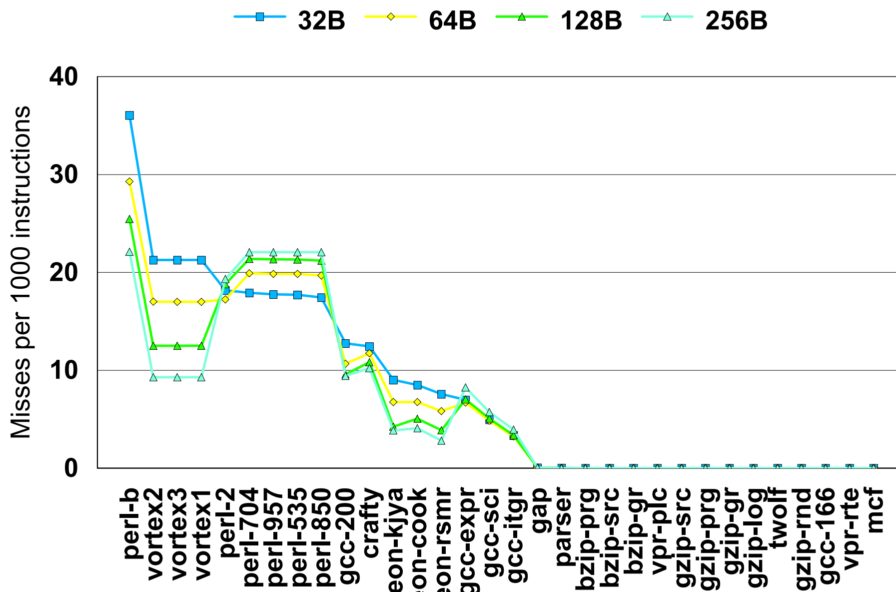

Figure 9: Varying 4-way instruction cache line size for integer programs . All caches are 32K and 4-way set associative. The results are sorted with respect to number of misses generated by the cache 32 byte lines.

5.1.3 Varying Line Size

Instructions exhibit good spatial locality and this is reflected in Figure 6 and Figure 7. Each curve represents the line size of a 32K direct-mapped cache. As the line size is increased from 32 bytes to 256 bytes, most applications show reduction in miss numbers.

Unlike associativity, we see some examples where miss numbers increase with the increased line sizes for a fixed capacity. This is an expected behavior caused by the reduction in the number of congruence classes. This is more visible in Figures 8 and 9 which show the effects of varying the line size for a 32K 4-way set associative cache.

For floating point applications, on average, when the line size is increased from 32 bytes to 64 bytes and from 64 bytes to 128 bytes, the number of misses are reduced by 15% each time. When 256 byte blocks are used instead of 128 byte blocks, the miss numbers go down by 8%.

Integer benchmarks benefit from increased block size more significantly when compared to floating point benchmarks for instruction references. Integer programs tend to exhibit less looping behavior than floating point programs, therefore there is less temporal locality, making spatial locality more dominant in instruction cache behavior. When the block size is increased from 32 bytes to 256 bytes , the number of misses per 1000 instructions go down by almost 50% on average. When we analyze individual block size increments, we see that most of this comes from the jump to 64 bytes from 32 bytes. This change results in 26% reduction in the miss numbers. The changes from 64 bytes to 128 bytes and from 128 bytes to 256 bytes each result in a 16% decrease in the average miss numbers.

5.2 Data Cache

This section is an analysis of data cache miss numbers, similar to that of the previous section on instruction caches. Again, we report results for the SPEC'2000 benchmarks while varying cache capacity, associativity, and line size.

5.2.1 Varying Capacity

Figures 10 and 11 respectively show the results for several direct-mapped and 4-way set associative caches with 32 byte lines. Each curve corresponds to a cache capacity ranging from 4K to 256K. Each set of benchmarks is listed in descending order of misses generated by a 4K cache.

For the floating point benchmarks, the average number of misses per 1000 instructions for a 32K direct-mapped cache with 32B lines is 52.2 whereas a 4-way cache with same capacity and block size generates 45.75 misses per 1000 instructions. With a 32K direct-mapped cache, the integer benchmarks have on average 27.9 misses per 1000 instructions. When the same cache is 4-way set associative, it averages about a miss every 50 instructions. Clearly, the floating point benchmarks stress the data cache more than integer benchmarks.

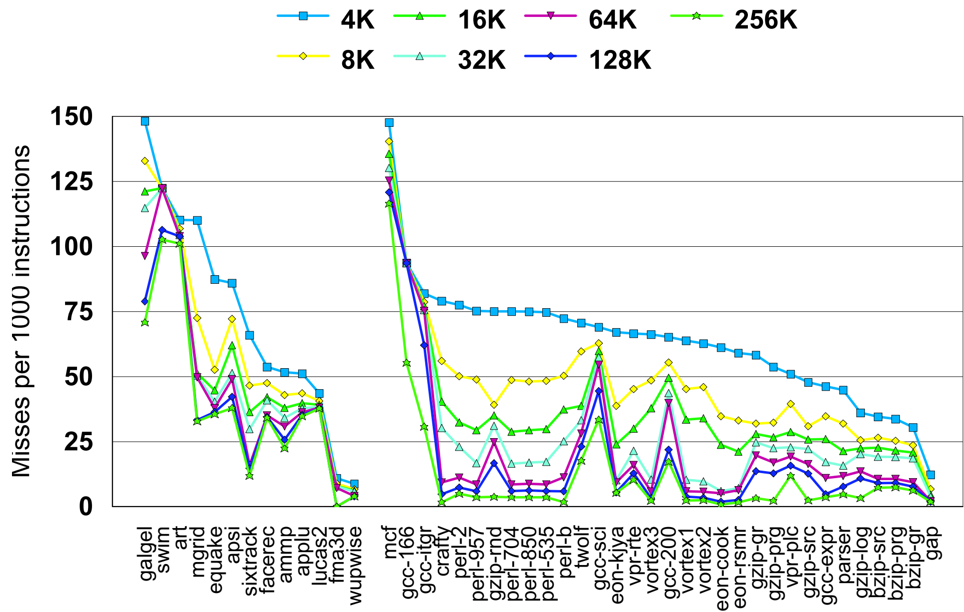

Figure 10: Varying direct-mapped data cache capacity. All caches have 32 byte lines. The results are sorted with respect to misses generated by the 4K cache.

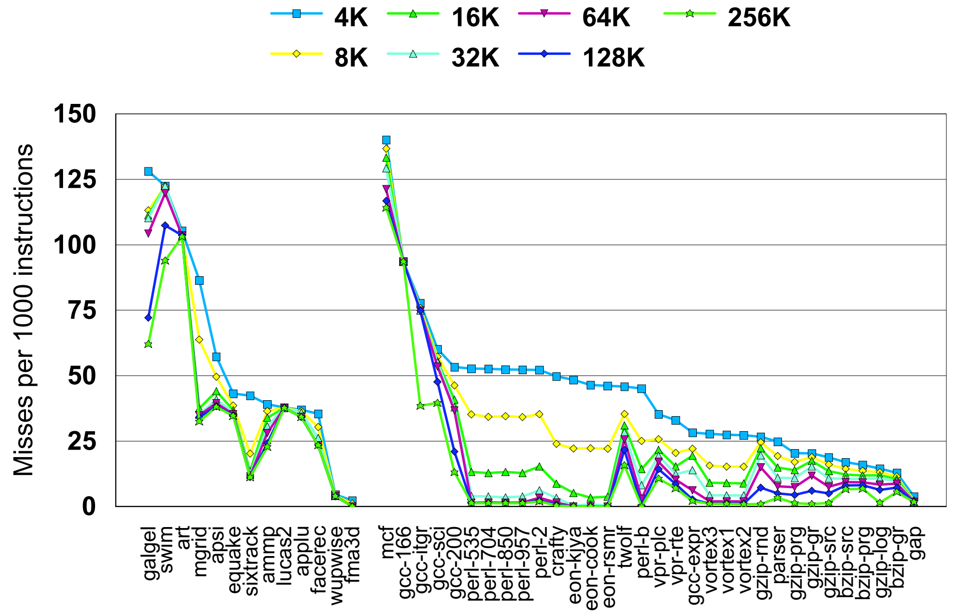

Figure 11: Varying 4-way data cache capacity. All caches have 32 byte lines. The results are sorted with respect to misses generated by the 4K cache.

Among the floating point benchmarks, galgel, swim and art have the most misses - more than a miss every ten instructions, for both direct-mapped and 4-way set associative caches with 32 byte lines and up to 64K in capacity. Even with a 256K 4-way cache, swim misses 94 times in every 1000 instructions. This is caused by the very large working set of this application which is close to 200 MBytes. Increasing the capacity from 4K to 256K results in a 33% reduction in misses for a 4-way cache. Most of this improvement is gained when the cache is increased from 4K to 8K with 11% reduction in average number if misses. The next significant jump occurs when the cache size is increased to 128K from 64K with a 9% reduction.

mcf, gcc and perl are the forerunners among integer benchmarks in causing data cache misses. As the cache size is increased from 4K to 256K, the number of misses generated by perl drop below four for every 1000 instruction. mcf has more than a miss per ten instructions even at 256K. gcc exhibits highly input-dependent behavior. For example, when running on input “166.i”, it has close to 94 misses per 1000 instructions for a 256K 4-way cache. For the same cache, the number of misses per 1000 instructions are 2.15 when running on “expr.i” Another interesting behavior exhibited by gcc is *congruence class starvation*[1]. As seen in the figures, for “166.i”, gcc has fewer misses with a direct mapped 256K cache than a 4-way 256K cache. Increased associativity for a fixed capacity causes the number of congruence classes to hold data to go down and hence hurts temporal locality. Increasing the cache capacity seems to be more effective for integer benchmarks as the number of misses decreases by more than 71% when the cache size increases from 4K to 256K for a 4-way cache.

5.2.2 Varying Associativity

Figure 12 shows miss results for a 32K data cache with 32 byte lines, when associativity is varied from one to eight.

The results suggest that for 32K cache, increasing the associativity from 1-way to 2-ways results in a reduction in the number of misses(11% for floating point and 25% for integer benchmarks). The graph also shows the diminishing returns as increasing the number of ways further on does not provide any significant gains for a fixed cache size.

For some integer applications such as crafty, utilizing a 2-way cache instead of a direct-mapped one results in significant advantages (78.5% reduction in number of misses). For applications with large miss rates, e.g. mcf and gcc, the increased associativity is of little benefit.

Similarly, mgrid, facerec and fma3d are among the floating point benchmarks that benefit from added associativity. Applications including swim, art, lucas2 and wupwise remain insensitive to associativity for a 32K cache.

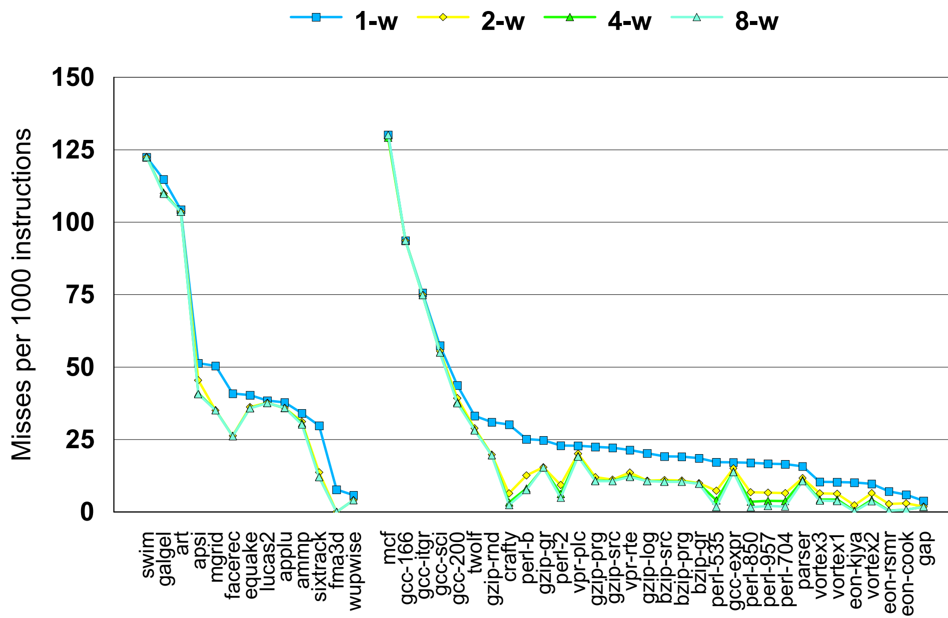

Figure 12: Varying data cache associativity. All caches are 32K with 32 byte lines. The results are sorted with respect to misses generated by the direct-mapped cache.

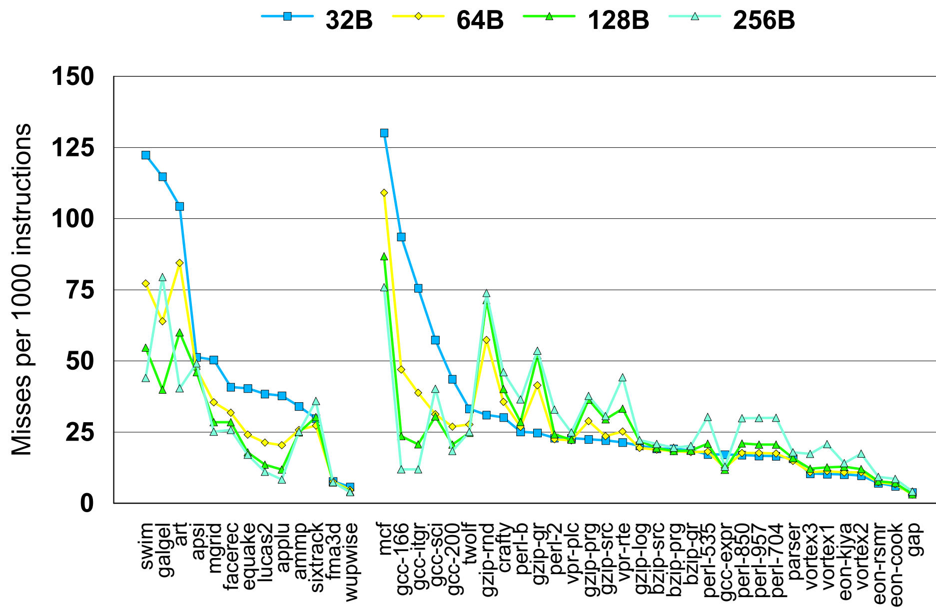

Figure 13: Varying data cache line size. All caches are 32K and direct-mapped. The results are sorted with respect to misses generated by the cache with 32 byte lines.

5.2.3 Varying Line Size

Figure 13 presents the data cache miss results for a 32K direct-mapped cache with different line sizes. As with instruction caches, larger cache blocks help the applications with good spatial locality.

The trade-offs in the choice of a line size is that a large line implies a smaller number lines for a fixed capacity, shrinking the temporal window and sacrificing temporal locality. Two other factors that must be considered in selecting a line size is that a large line size results in a longer miss penalty but the directory managing the cache is smaller.

The integer benchmarks average 27.8 misses per 1000 instructions for 32 and 256 byte lines. The best line size for the integer benchmarks is 64 bytes with one miss out of every 40 instructions on average. The floating point benchmarks have 52.2 misses per 1000 instructions with 32 byte blocks. When the line size is increased to 256 bytes, this number goes down to 28.7. The minimum number of misses for the floating point benchmarks is achieved with 128 byte lines with 28.3 misses.

This graph shows a diverse range of behavior:

- Most of the integer benchmarks have higher number of misses with larger cache blocks. galgel, apsi and

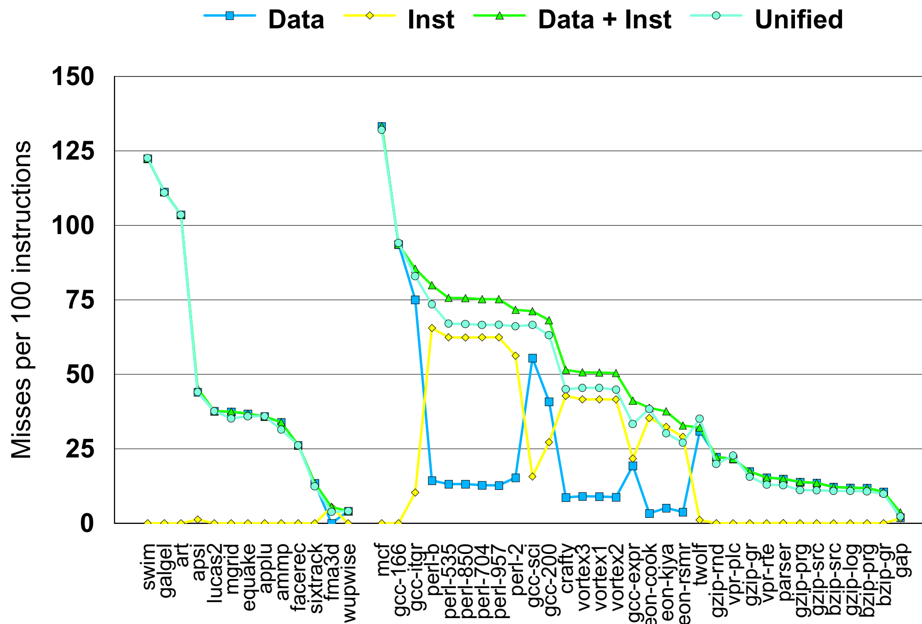

Figure 14: 16K 4-way split instruction and data caches vs. a 32K 4-way unified cache. All caches have 32 byte blocks. The results are sorted with respect to misses generated by split caches.

sixtrack are the floating point benchmarks that are hurt by the larger line sizes.

- fma3d, wupwise, bzip remain relatively unchanged.
- Most floating point applications benefit from larger cache lines. Also, mcf, twolf and gcc from the integer programs have fewer misses with increased block size.

5.3 Split vs. Unified Caches

In this section, we examine using split and unified caches and compare their respective miss rates.

The results are presented in Figure 14. It shows the number of misses per 1000 instructions for split 16K 4-way instruction and data caches and a 32K 4-way unified cache. All caches have 32 byte blocks. Along with the individual miss numbers for the split caches, we report the sum of these two numbers as a basis for our comparisons with the unified cache.

For some of the applications the unified cache results in fewer aggregate misses. Perl, vortex, crafty, gcc and eon are among these. For crafty, perl and vortex the number of misses are reduced by approximately 11%. As they incur

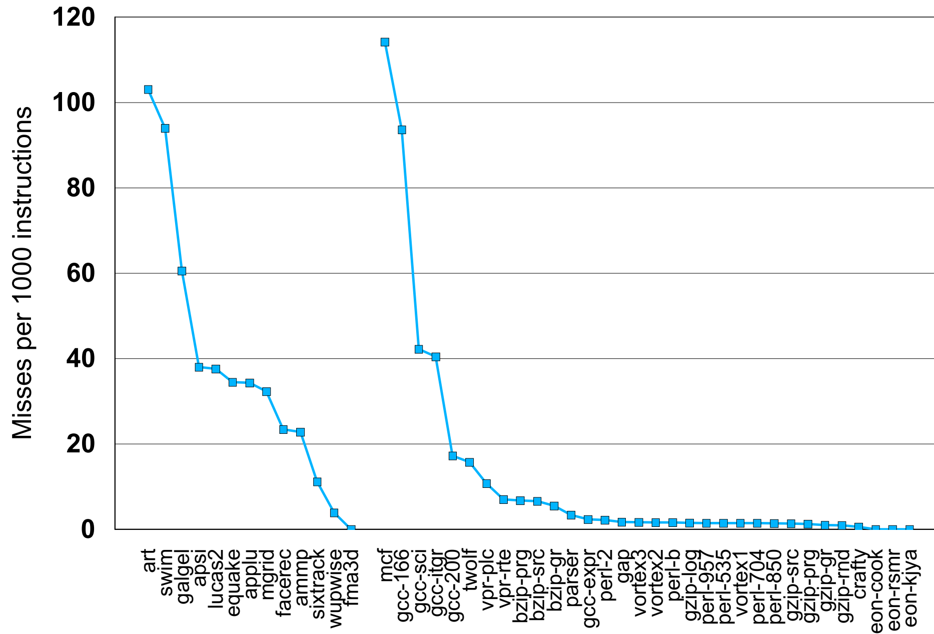

Figure 15: 256K 4-way unified second-level cache.

relatively few number of data misses, these gains are mainly from having more room to store the large instruction working set for these applications.

Several applications have a higher miss rate with a unified cache. Twolf is one of these. Some programs like gcc and vpr behave differently based on the input. When gcc is running on "166.i" and when vpr is doing placement, they incur more misses with a unified cache. For twolf, the performance degradation is quite significant, 10%. This is likely to be caused by thrashing between the relatively large instruction and data working sets.

5.4 Second Level Cache

This section focuses on the L2 miss behavior.

Figure 15 shows the number of L2 misses for a 256K 4-way unified instruction and data L2 cache with 32 byte lines. The first level split instruction and data caches are 32K, 2-way set associative and the cache blocks are 32 bytes. Floating point applications miss in the L2 cache 38.15 times every 1000 instructions. For integer programs, this number stands at 12.15. An interesting point is the average miss rate, floating point applications L2 references miss in the cache 80% of the time. The L2 miss rate is 27.2% for integer programs.

With around 100 misses per 1000 instructions, art and swim put the most pressure on the L2 cache among the

floating point programs. If we consider the fact that these two applications generate more than one data miss every 10 instructions even with a 256K data cache, they come out as the most aggressive of the floating point benchmarks in terms of stressing the memory system.

On the integer side, mcf and gcc (especially when running on "166.i") are worth mentioning. As a matter of fact, together these two programs generate more L2 misses per 1000 instructions then all the other integer benchmarks combined.

6 Summary

The ever-widening processor memory gap motivates a plethora of research projects on cache memories. Most of these techniques are tested on the SPEC suite of benchmarks to evaluate their merit.

Researchers [1, 3, 4] observed that miss rates are generally very low–especially for instruction references. Moreover, they have stressed the lack of operating system activity in SPEC benchmark simulations, concluding it does not represent the behavior of a multi-programmed, time-sharing system.

This study confirms that only a few applications place more than modest demands on the memory system. When we look at varying different cache parameters, increasing cache capacity provides the most significant gains. In general, instruction caches benefit from increased line size and associativity. Data cache behavior however is more diversified. Increasing the degree of associativity and line size result in improvements for some cases while it degrades performance for others.

References

[1] M.J. Charney and T.R. Puzak. Prefetching and memory system behavior of the spec95 benchmark suite. *IBM Journal of Research and Development*, 41(3), May 1997.

[2] SPEC: Standard Performance Evaluation Corporation. http://www.spec.org, September 2000.

[3] J.G. Gee, M.D. Hill, D.N. Pnevmatikatos, and A.J. Smith. Cache performance of the spec benchmark suite. Technical Report 1049, University of Wisconsin, September 1991.

[4] J.G. Gee, M.D. Hill, D.N. Pnevmatikatos, and A.J. Smith. Cache performance of the spec92 benchmark suite. *IEEE Micro*, 13(4):17–27, 1993.

[5] J.L. Henning. Spec cpu2000: Measuring cpu performance in the new millenium. *IEEE Computer*, July 2000.

[6] A. Lebeck and D. Wood. Cache profiling and the spec benchmarks: A case study. *IEEE Computer*, October 1994.

[7] D.N. Pnevmatikatos and M.D. Hill. Cache performance of the integer spec benchmarks on a risc. In *Computer Architecture News*, pages 53–68, 1990.

[8] T.R. Puzak. Analysis of cache replacement-algorithms. Ph.D. Dissertation, University of Massachusetts, Amherst MA, 1985.

[9] Timothy Sherwood and Brad Calder. Time varying behavior of programs. Technical Report CS99-630, University of California, San Diego, August 1999.

[10] N.C. Thornock and J.K. Flanagan. Using the bach trace collection mechanism to characterize the spec 2000 integer benchmarks. In *3rd IEEE Annual Workshop on Workload Characterization*, September 2000.