6-1-1995

An Empirical Study on How Program Layout affects Cache Miss Rates

Jeffrey P. Bradford
Purdue University School of Electrical and Computer Engineering

Russell W. Quong
Purdue University School of Electrical and Computer Engineering

An Empirical Study on How Program Layout Affects Cache Miss Rates

Jeffrey P. Bradford
Russell W. Quong

TR-ECE 95-20
June 1995

School of Electrical
and Computer Engineering
Purdue University
West Lafayette, Indiana 47907-1285

An Empirical Study on How Program Layout affects Cache Miss Rates

Jeffrey P. Bradford **Russell W. Quong**

School of Electrical Engineering
Purdue University
West Lafayette, IN 47907
{jbradfor,quong}@ecn.purdue.edu

Abstract

Cache miss rates are quoted for a specific program, cache configuration, and input set; the effect of program layout on the miss rate has largely been ignored. We examine the variation of the miss rate resulting from randomly chosen layouts, the ***miss variation***, for several cache configurations (cache size, lines size, and set-associativity), input sets, and optimization levels for five programs in the SPEC benchmark suite. We observed miss rates that varied from $0.6m$ to $1.8m$, where m is the mean miss rate. We did not observe any consistently good layouts across different parameters; in contrast, several layouts were consistently bad. Overall, cache line size has little effect on the miss variation, while increasing the cache size (decreasing the miss rate), decreasing the set-associativity, or increasing the optimization level increased the miss variation. We question the validity of using a single layout to represent the miss rate of a given program for a direct-mapped cache.

1. Introduction

In designing a memory system, computer architects run trace-driven simulations for many programs to determine the miss rate for various cache configurations. For a given cache configuration, the resultant miss rate depends on three factors: (i) the program being executed; (ii) the input data; (iii) the layout, namely the specific compilellink time mapping of the object code and data objects to memory addresses. To address the first two factors, standard benchmark suites (SPEC, Perfect, etc.) have been developed and are commonly used.

We are concerned with the last factor, the layout. Different layouts have different miss rates, as changing the layout changes which instructions and data map to the same cache line(s). Almost invariably only one layout is used when calculating the miss rates. The layout is affected by many factors including compiler optimization, the order in which the program is linked together, and the specific libraries on a system.

The original impetus for this work is the gap model [Quo94], which analytically predicts the "true" miss rate averaged over all possible layouts. As expected, there was some deviation between the miss rates as predicted by the gap model and the miss rates as measured from a cache simulation on the default layout. In this paper, we explore the related issue of how much miss rates vary "randomly" due to layouts.

Furthermore, it makes sense to purposely consider different layouts for two reasons. First, the notion of a single "standard" layout for a program is erroneous. Compilers and system libraries evolve *over* time, both of which affect the layout in unforeseen ways. More importantly, different architectures have different object code densities which greatly affect layouts and miss rates. Second, we show that layouts do affect the miss rate which in turn can affect system performance. For time-critical applications, it would be wise to view the finding of a layout with a low miss rate as another standard compile/link optimization.

Our primary goal was to determine if measuring the miss rate for a single layout gives an accurate estimate of the true miss rate. In addition, we hoped to answer the following questions.

- What is the typical variation in the miss rates (the miss variation) for programs?
- How do cache size, line size, associativity, optimization, and input set affect miss variation?
- Are there consistently good or bad layouts as we change the cache: size, line size, optimization, and input set?
- How does the miss variation affect system performance?

The paper is organized as follows. Section 2 discusses miss variation in detail and gives some definitions. Section 3 describes the experiments we performed to generate the data. Section 4 summarizes the results and discusses some interesting observations. Section 5 answers the four questions posed above, and Section 6 presents conclusions and offers ideas for further research.

2. Background and Previous Work

For a program to have a high miss variation requires two memory fragments that obey three properties (a memory fragment being a code section or data object). First, the fragments must be executed frequently. Second, the fragments must have high temporal locality, namely after one fragment is referenced, the other is likely to be referenced soon. Third, the relative addresses of the fragments must be layout-dependent, namely in some but not all layouts the fragments map to the same cache lines.

Note that the only layout-dependent data objects are static (global) scalars and arrays; in particular, the addresses of stack-allocated variables (locals) and heap-allocated variables (via malloc) are not affected by the layout. Similarly, the relative addresses of elements of the same array are not affected by the layout. All code is layout dependent.

A program consists of modules; a module consists of procedures; and a procedure consists of basic blocks. Usually when generating the object file, the compiler places procedures and its constituent basic blocks consecutively in memory based on their order in the source module. Similarly, the linker places object files consecutively in memory based on their order in the link command. There is rarely any thought given to the ordering, both because of and despite of the fact that procedures and modules can be arbitrarily ordered in a program.

We can rearrange the layout of a program at three granularities, module, procedure, and basic block. While we can rearrange modules and procedure arbitrarily, rearranging basic blocks requires adding unconditional jumps to preserve the original control flow. These unconditional jumps change the program in two ways. They make the code larger, and they change the instruction stream. For these reasons, we did not rearrange at the basic block level.

For a program with M modules and P procedures, there are M! module-level and P! procedure-level rearrangements; even for small M and P there many possible layouts. Note that we have ignored layouts that have holes or unused addresses. In practice, a code or data segment is contiguous (no holes), to minimize memory and disk space usage.

We would expect the miss variation to increase as we rearrange at finer granularities, due to intra-module and intra-procedure locality. Namely, after referencing a procedure we expect to reference another procedure in the same module soon. By default, the compiler places intra-module procedures consecutively in memory which guarantees these do not conflict (unless the procedures fill up the cache and wrap around). Procedure-level rearranging removes this guarantee. This argument applies to even a greater extent when rearranging at the basic-block level, as there is stronger temporal locality among intra-procedural basic blocks than among intra-module procedures.

We expect a direct-mapped cache to show greatest variation, with variation decreasing as the set-associativity increases. In the extreme case, an LRU cache is insensitive to the layout. In addition, we expect decreasing the compiler optimization level to decrease the miss variation, as

a common optimization is to remove redundant loads/stores [FL91]. A redundant load/store to address a duplicates a previous reference to a, so it is very likely a is in the cache. Thus, redundant loads/stores are "sure" cache hits, which effectively reduces both the miss rate and the miss variation (but increases cache accesses).

A *parameter* is any of the factors we change, namely the cache configuration (line size, size, or set-associativity), the compiler optimization, and input set. We call a layout good (bad) if it yields a miss rate, for a given set of parameters, that is significantly lower (higher) than the average miss rate. We call a layout *extreme* if the layout is either good or bad. Note that a layout that is bad for one set of parameters might be good for another set of parameters. Finally, we use m to denote the experimental mean miss rate over all measured layouts for a given set of parameter!;.

We found no reference in previous literature to how randomly chosen layouts affect the miss rate, and very few references to variation in execution time in general. Sarkar [Sar89] discusses execution time variation caused by loops processing different data in the context of picking the optimal grain size for parallelization. Various cache models have been proposed [AHH89] [Quo94], but none of these models consider miss variation.

The idea of rearranging code to improve cache performance is not new. [PH90] used two compiler optimizations to improve cache performance (and TLB and virtual memory performance). Their first optimization arranged procedures using a "closest is best" heuristic by repeatedly coalescing procedures that were adjacent in a weighted call graph until a single node (the entire program) existed. Their second optimization split each procedure into two procedures, one of commonly used basic blocks and one of "fluff", and within each of the two new procedures chained together basic blocks based on usage counts of the internal control-flow graph for that procedure. Unfortunately, they did not report the improvement in the miss rate, only the improvement in execution time (10%-26%). [McF89] reordered basic block using compile-time usage estimates and showed that a direct-mapped cache could perform better than a set-associative cache, if instructions could selectively be excluded from the cache. Rearranging code is also a variation of the well-studied problem of rearranging the address map to minimize paging [Ker71] [BC72][Har88][OMHL93]. Our research is unique in that we are the first to consider the range of miss rates resulting from randomly chosen layouts.

3. Method

We ran six types of experiments to empirically answer to the questions posed in the introduction. We chose five programs from the SPEC92 benchmark suite as shown in Table 1. Two are C programs, `espresso` and `gcc`, and three are FORTRAN programs, `spice`, `doduc`, and `fpppp`. We ran all six types of experiments under `SunOS` 5.3 and compiled them with `cc` 2.0.1 using static linking. We used `cachesim5` 5.2 with `shade` 4.1.6 [CK93] to calculate the miss rates. To run the experiments, we changed the SPEC makefile and wrote shell scripts that generated (differentlayouts, changed the compile/link flags, and changed the input set used.

Benchmark	Language	Lines of Source Code	Number of Procedures	Number of Modules	Static Code Size (kB)	Input Set Used
espresso	C	14,841	365	44	439	input.ref/ti.in
spice	Fortran	18,912	144	13	851	input.short/greycode.in
doduc	Fortran	5,334	41	41	508	input.ref/doduc.in
	C	103,458	1600	44	1,112	input.short/*
fpppp	Fortran	2,718	13	13	403	input.short/natoms

Table 1: Properties of the Benchmarks Used
The *static code size* is the number of bytes of the executable program on disk.

We ran all types of experiments on the same 10 cache configurations, consisting of five cache sizes, 256, 1-kB, 4-kB, 16-kB, 64-kB, and two line sizes, 16 bytes and 64 bytes. Unless otherwise noted in Table 2, we measured the miss rates for 21 layouts (20 random layouts and the original layout) on these 10 cache configurations, with full optimization, a direct-mapped cache, the input sets shown in Table 1, and module-level rearrangement. We generated the random layouts by changing the order the modules are specified to the linker.

For the first type of experiment, we used the parameters listed in the preceding paragraph to measure a baseline miss variation.

Experiment Number	Experiment Type	Resultant figures(s)
1	Baseline	1, 6, 10, 12, 13
2	Rearrange at Procedure Level	7
3	100 Layouts	11, 15
4	Different Input Sets	2, 3, 14
5	Different Optimization Level	4, 5, 8, 9
6	4-way set-associative Cache	16

Table 2: Description of Experiments Performed

For the second type of experiment, we generated 21 layouts by rearranging spice at the procedure level. instead of at the module level. We split the existing modules into new modules so that each new module contained exactly one procedure. We then linked the new modules in 21 random orders. We chose spice because we wanted a FORTRAN program, as FORTRAN programs are easier to break into procedures than C programs, and of the three FORTRAN programs only spice has multiple procedures per module.

For the third type of experiment, we measured the miss rates for **doduc** and **fpppp** for 100 layouts. We were concerned that "only" 21 layouts might not yield miss variation representative of the true miss variation; we wanted to confirm that the variation was due to the program itself and not to the small number of layouts chosen. We chose **doduc** because it gave a low, evenly spread out variation with no outlying points in Experiment 1, and we chose `fpppp` because it gave a wide, unevenly distributed variation.

For the fourth type of experiment, we ran **espresso** and **fpppp** with different input sets. We used **bca.in** and `tial.in` for **espresso** and **input.ref/natoms** for **fpppp.**

For the fifth experiment, we compiled **espresso** and **spice** with medium optimization (-02) and with no optimization, instead of full optimization (-04) which we used in the other five types of experiments. Full optimization (on the SUN compilers we use) can cause the following problems: (i) an increase in code size due to loop unrolling and inlining; (ii) a significant increase in compile time; (iii) incorrect code being generated by a risky optimization. For these reasons, many programmers use medium optimization instead. Compiling with no optimization is common during debugging. Table 3 shows how different optimization levels affect the size of the executable, the number of instructions executed, and the number of data references (the number of loads/stores) for the three optimization levels. Note that shifting from no optimization to medium optimization reduced the number of instructions executed by at least a third in all cases.

Bench-mark	No Optimization			Optimization Level 2			Optimization Level 4		
	Static Code Size (kB)	Instruc-tions executed ($x10^6$)	Data Refer-ences ($x10^6$)	Static Code Size (kB)	Instru c-tions execut ed ($x10^6$)	Data Refer-ences ($x10^6$)	Static Code Size (kB)	Instruc-tions executed ($x10^6$)	Data Refer-ences ($x10^6$)
espresso	481	962	241	387	611	163	439	593	160
spice	1,037	6,682	2,396	848	3,913	1,461	851	2,553	924
doduc	612	3,132	1,439	512	1,847	710	508	1,665	599
gcc	1,416	1,936	530	995	1,222	344	1,112	1,188	325
fpppp	433	2,245	897	405	1,542	767	403	1,444	707

Table 3: Effect of Optimization on static code size, the number of instructions executed, and the number of data references

For the sixth experiment, we ran **fpppp** with a four-way set-associative cache. We chose **fpppp** because its miss variation was greatest for the direct-mapped cache.

4. Results

Please refer to Table 4, which summarizes all the figures. To aid comprehension we use the same scale for all figures. The X-axis shows the size of the cache on a log scale. The Y-axis shows the miss rate on a log scale over the range from 0.3% to 30%, which is likely to be the range of interest to a computer architect. When the miss rate is above 30%, performance is likely to be poor irrespective of the variation of the miss rate. Similarly, unless the miss penalty is high (100's of cycles), a cache miss rate less than 0.3% will have little impact on program execution time. The following descriptions point out a few of the interesting features; see the figures for full detail!;.

Figure 1 shows the miss rate for `espresso` for twenty-one layouts and the input set `ti.in`. The I-cache has three consistently bad layouts for both 16-byte and 64-byte lines, For example, for the 4-kB I-cache and 16-byte lines, the bad layouts have a miss rate of 3.3%, compared to 2% for the remaining 18 layouts. For the 16-kB I-cache, there was only one bad layout (with miss rate of 2.3% for 16-byte lines), which was one the three bad layouts. The data cache shows little miss variation except for the 1-kB D-cache. For 64-byte lines, the miss rates split into two groups, with three layouts yielding a 18.5% miss rate and the other 18 layouts yielding a 15.3% miss rate. (These three layouts are not the three bad layouts for the I-cache.)

Figure 2 shows the miss rate for `espresso` for the same 21 layouts as in Figure.1 and the second input set, `bca.in`. For both 16-byte and 64-byte lines, this second input set yields a much greater variation than the first input set, which is most noticeable for the 1-kB I-cache. The 4-kB I-cache has four bad layouts, three of which are bad for the first input set too.

Figure 3 shows the miss rate for `espresso` with the third input set, `tial.in`. Only one layout is bad for the 4-kB I-cache. This layout is also bad for the 16-kB I-cache for all three input sets; obviously, this is a layout to avoid. There is also a very slight splitting in the miss rates for the 4-kB D-cache. Changing to 64-byte lines changed the split found in the 4-kB D-cache; 16-byte lines has 1 bad layout, 64-byte lines has 7 bad layouts.

Figure 4 shows the miss rate for `espresso` with medium optimization. For 16-byte lines, the miss rate is similar to full optimization (Figure 1), with three bad layouts for the 4-kB I-cache, and one of these three layouts are bad for the 16-kB I-cache. But the three bad layouts for medium optimization are *different* layouts than the three bad layouts for full optimization. For 64-byte lines the miss rates for the 4-kB and 16-kB I-cache do not split and the overall variation is decreased. The three layouts that are bad in Figure 1 for the 1-kB D-cache are bad here as well.

Figure 5 shows the miss rate for `espresso` with no optimization. Compared to the results for full optimization (Figure 1), the biggest difference is the 64-kB I-cache. While all the miss rates are low (below 0.6%), they are higher than the miss rates for the other two optimization levels. Again we see one bad layout for the 16-kB I-cache, but this is a different layout than the 6 bad

Figure	Bench-mark	Experiment Type	Input Set	Opt Level	Summary of Results (MV = Miss Variation)
1	espresso	1	input.ref/ ti.in	Full	16 kB I-cache: High MV 4, 16 kB I-cache: few bad layouts
2	espresso	4	input.ref/ bca.in	Full	1, 4 kB I-cache: High MV 4, 16 kB I-cache: few bad layouts
3	espresso	4	input.ref/ tial.in	Full	16 kB I-cache: High MV 4, 16 kB I-cache: few bad layouts
4	espresso	5	input.ref/ ti.in	Med	Similar to Figure 1, except MV decreased
5	espresso	5	input.ref/ ti.in	None	Smaller MV than Figure 4 excpet 64 kB I-cache
6	spice	1	input.short/ greycode.in	Full	16 kB I-cache: High MV 64 kB I-cache: Bimodal
7	spice	2	input.short/ greycode.in	Full	(Rearranged at Procedure Level) Similar to Figure 6
8	spice	5	input.short/ greycode.in	Med	Similar to Figure 6
9	spice	5	input.short/ greycode.in	None	Lower MV than Figure 6, esp 16 kB I-cache
10	doduc	1	input.ref/ doduc.in	Full	16, 64 kB I-cache: High MV
11	doduc	3	input.ref/ doduc.in	Full	(100 Layouts) Similar to Figure 10
12	gcc	1	input.short/ *	Full	Low MV
13	fpppp	1	input.short/ natoms	Full	16 kB D-cache: Large MV 64 kB D-cache: Bimodal
14	fpppp	4	input.ref/ natoms	Full	Similar to Figure 13
15	fpppp	3	input.short/ natoms	Full	(100 Layouts) Similar to Figure 13
16	fpppp	6	input.short/ natoms	Full	(4-way set-associative cache) No MV

Table 4: Summary of Figures. Unless otherwise noted, for each figure we generated 21 layouts by rearraning at the module level and used a direct mapped cache.

layouts for the other two optimization levels. Besides a slight splitting in the 1-kB I-cache, the results for 64-byte lines are almost identical to the results for 16-byte lines.

Figure 6 shows the miss rate for **spice.** The miss rates for the 16-kB I-cache are evenly spread out between 0.3% and 1.2%. The miss rates for the 64-kB I-cache yield a bi-modal distribution, with 14 layouts yielding low miss rates (most below 0.1%), and 7 bad layouts yielding miss rates grouped around 0.8%. These 7 layouts are also bad for 64-byte lines.

Figure 7 shows the miss rate for **spice** when rearranged at the procedure level instead of at the module level. Note that rearranging **spice** at the procedure level slightly *decreases* the variation for 16-byte lines, but slightly increases the variation for 64-byte lines. While the 64-kB I-cache gives similar results when rearranged at both the module and procedure level, the bad layouts differ. Of the 7 bad layouts when rearranged at the module level and 8 bad layouts when rearranged at the procedure level, only 2 layouts are bad in both cases.

Figure 8 shows the miss rate for **spice** with medium optimization; the results are very similar to the results for full optimization (Figure 6). Of the 7 bad layouts for the 64-kB I-cache, 6 of these are among the 7 bad layouts in Figure 6. For 64-byte lines, the 4-kB and 16-kB D-cache has 1 bad layout and the 1-kB D-cache has 2 bad layouts.

Figure 9 shows the miss rate for **spice** with no optimization. Overall, the miss variation has decreased, especially for the 16-kB I-cache. One layout yields a miss rate above 0.3% for the 64-kB I-cache; this layout is not bad for the other cases. As in Figure 8, changing to 64-byte lines causes an interesting split in the D-cache which we can not currently explain. Three layouts are bad for the 1-kB D-cache, two layouts are bad for the 4-kB D-cache, and one layout is bad for the 16-kB D-cache. Of the three layouts that are bad for the D-cache, two are also bad in Figure 8.

Figure 10 shows the miss rate for **doduc.** For 16-byte lines, the miss rates for the 16-kB I-cache are evenly spread out between 3.8% and 7.8%. The miss rates for the 64-163 I-cache are also even spread out, with fifteen layouts yielding a miss rate between 1.0% and 2.0%, and all twenty-one layouts yielding a miss rate between 0.7% and 3.4%.

Figure 11 shows the miss rate for **doduc** with 100 layouts instead of the previous 21. The miss rates for the 16-kB I-cache are still evenly spread out, with miss rates between 3.3% and 7.8%, except for one layout which yields a miss rate of 2.65%. For the 64-kB I-cache, 94 of the 100 layouts yield miss rates between 0.75% and 2.8%. The remaining six layouts yield slightly higher miss rates', the highest being 4.1%. For 64-byte lines there is a slight decrease in variance compared to 16-byte lines.

Figure 12 shows the miss rates for **gcc.** All cases have very little variation. As the results are the sum of 19 different input files, it is possible that much of the variation has been averaged out. The miss rates for each individual input case are not shown, due to the low number of instructions executed for each input. The results for the 64-byte line case are very similar, except for a slightly larger, but still small, variation for the D-cache.

Figure 13 shows the miss rates for fpppp with the first input set, `input.short/natoms`. The miss rates for the 16-kB D-cache yield a quad-modal distribution. A majority of the layouts (eleven) yield miss rates spread out between 0.9% and 1.5%, four are between 2.2% and 2.7%, five are grouped together around 4.7%, and one layout yields a miss rate of 5.9%. The 64-kB cache yields a bimodal distribution, with 16 layouts yielding miss rates belovv 0.3% and 5 bad layouts yielding a miss rate over ten times higher (near 4%). While otherwise the results are similar for 64-byte lines, the quad-modal distribution found in the 16-kB D-cache is almost non-existent for 64-byte lines, and the number of layouts in each group is different.

Figure 14 shows the miss rates for fpppp with the second input set, `input:. ref/natoms`. The results are very similar to the results with the first input set (Figure 13) for both line sizes, the only difference being the lower two groupings for the 16-kB D-cache in Figure 13b have merged into one group in Figure 14b.

Figure 15 shows the miss rate for fpppp for 100 layouts. For 16-byte lines, we see a clear bi-modal distribution for the 16-kB D-cache, with 65 layouts in the lower group and 35 layouts in the upper group. The 64-kB D-cache shows a bi-modal split for both lines sizes, with 73 layouts in the lower group and 27 layouts in the upper group.

Figure 16 shows the miss rate for fpppp with a 4-way set-associative cache. As expected, the variation decreased to almost nothing for both 16-byte and 64-byte lines.

5. Analysis

We now return to the four questions posed in the introduction.

> What is the typical miss variation for programs?

Our figures show that programs exhibit a wide variety of variation, from no .variation, to wide variation, to multi-modal variation. The I-cache shows more variation than the D-cache for three (espresso, spice, and doduc) of the five benchmarks, and the D-cache shows more variation for one benchmark, fpppp. Finally, `gcc` shows almost no variation for both the I-cache and D-cache, but the `gcc` miss rates were the average of 19 input cases. There were no consistently good layaats, namely layouts that were good for a wide variety of parameters. There were, however, many consistently bad layouts.

We define the relative deviation to be the standard deviation divided by m, the mean. For example, if m = 3.0% and the standard deviation is 1.5%, then the relative deviation is 50%. We used the formula for standard deviation for measured data, $\sqrt{\sum (x_i^2 - \bar{x}^2)/(n-1)}$ [GKP89], to calculate the relative deviation for all figures as shown in Table 5. There are three important observations. First, when m is moderate (1% to 5%), the relative deviation is often between 40% and 80%. Second, all the very high relative deviation values (>90%) occur when m is low (<0.3%). Third, while most of the miss rates lie within ± 2 standard deviations of the mean, the extreme layouts lie even further out.

Figure Number	Instruction Cache				Data Cache			
	1-kB	4-kB	16-kB	64-kB	1-kB	4-kB	16-kB	64-kB
1	6.8/5.0	18.4/17.2	55.8/50.4	94.9/96.1	0.3/7.1	0.5/0.3	0.2/0.7	0.5/0.8
2	28.5/25.5	53.1/52.2	81.1/78.8	71.6/90.4	0.2/0.2	0.2/0.4	0.0/0.1	0.1/0.1
3	4.6/6.3	12.1/13.4	49.4/43.0	103.2/105.6	2.9/2.1	3.0/7.0	0.2/3.8	0.1/0.2
4	8.9/6.6	17.5/14.6	42.3/35.8	124.8/124.8	0.9/6.9	0.6/0.4	0.3/0.9	1.2/1.2
5	3.5/3.3	10.8/11.1	23.9/23.9	89.4/83.6	0.3/3.6	0.5/0.6	0.2/0.7	0.7/1.2
6	6.0/5.4	7.0/9.4	35.0/32.6	125.5/118.3	0.7/0.6	2.5/1.8	2.0/1.5	0.5/0.7
7	4.3/6.8	5.3/9.4	31.0/35.0	108.4/97.9	0.9/0.8	2.1/1.4	3.0/2.5	2.0/1.4
8	5.1/5.7	5.8/7.4	32.5/30.3	105.1/102.5	1.5/11.6	2.3/15.5	2.5/29.0	2.7/3.2
9	4.4/3.4	6.0/6.5	5.7/8.6	132.6/139.2	3.9/24.8	3.9/40.9	2.7/37.5	1.7/2.7
10	1.0/1.6	4.1/5.3	22.3/21.3	41.8/42.4	2.2/2.2	4.1/3.3	7.8/8.4	13.4/13.5
11	0.9/1.5	3.5/4.3	19.9/18.8	45.7/42.7	2.0/2.3	5.3/4.1	7.9/10.5	13.8/12.4
12	0.6/1.1	1.8/1.9	5.7/4.8	10.4/12.2	2.6/4.5	1.6/4.9	1.4/3.8	2.8/6.6
13	0.4/0.9	0.5/0.8	6.6/6.3	77.2/77.6	1.6/2.8	6.6/8.7	65.6/49.4	143.8/124.6
14	2.2/3.5	2.3/3.8	5.5/5.1	47.1/47.5	2.8/4.0	7.4/7.9	64.6/42.7	141.4/123.8
15	0.4/0.7	0.4/0.7	5.2/5.0	103.7/90.2	1.7/2.8	5.2/9.2	63.5/51.3	132.1/123.5
16	0.5/0.8	0.4/0.8	0.8/1.4	109.6/153.9	1.3/2.7	1.9/2.0	7.7/9.0	7.9/9.6

Table 4: Relative deviation (the standard deviation divided by the mean) for all figures. Within each box, the first number is for 16-byte lines and the second number is for 64-byte lines. The relative deviation values are low for 256-byte caches, and thus are not shown.

How do *(i)* cache size, *(ii)* line size, *(iii)* set-associativity, *(iv)* optimization, and *(v)* input set affect miss variation?

(i) The variation is low at small cache sizes (high miss rate); as the cache size increases, and the miss rate decreases, the variation either increases or remains low.

(ii) The two lines sizes (16 and 64 bytes) give nearly identical variations, with one exception. In the following four cases, the miss rates split into two groups for 64-byte lines but not for 16-byte lines. For the 1-kB D-cache for `espresso`, Figure 1b shows no split while Figure 1d shows a 1813 split. For the 4-kB D-cache for `espresso`, Figure 3b shows a 20/1 split while Figure 3d shows a 14/7 split. For the 1-kB D-cache for `spice`, Figures 8b and 9b show no split, while Figures 8d and 9d show a 19/2 and a 1813 split, respectively.

(iii) Direct-mapped caches show much more variation than four-way set-associative caches, which show no variation.

(iv) Decreasing the optimization usually decreases the variation; we give two examples and one counter-example. First, for the 4-kB and 16-kB I-caches for `espresso`, full optimization (Figure 1) yields several bad layouts; for medium optimization (Figure 4) the bad layouts are "mediocre", while no optimization (Figure 5) yields no bad layouts. Second, for the 16-kB I-cache for `spice`, full optimization (Figure 6) and medium optimization (Figure 8) have significant variation while no optimization (Figure 9) has little variation. One counter-example is the 64-kB I-cache for `espresso`, for which no optimization yields a higher variation than either full optimization or medium optimization.

(v) Changing the input set does not change the variation, with the following single exception. For the 1-kB I-cache for `espresso`, two input sets (Figures 1a and 3a) show low variation, while one input set (Figure 2a) shows high variation. In a separate case, it appears that the 16-kB D-cache for `fpppp` shows a difference between the two input sets, with Figure 13b showing a quad-modal distribution and Figure 14b showing a bi-modal distribution. However, running 100 layouts (Figure 15) seems to show that the true distribution is bimodal, and the quad-modal distribution is an artifact of using only 21 layouts.

> Are there consistently good or bad layouts as we change the (i) cache size, (ii) line size, *(iii)* optimization, and (iv) input set?

(i) As we change the cache size, bad layouts remain bad, but there are no consistently good layouts. We give two examples of consistently bad layouts. First, for the D-cache for `fpppp` (Figures 13b and 14b), of the 6 bad layouts for the 16-kB cache, 5 are bad for the 64-kB cache. Second, for the I-cache for `spice`, (Figure 6a) of the 7 bad layouts for the 64-kB cache, 6 are bad or yield above average miss rates for the 16-kB cache.

(ii) Line size has no effect on layouts, with the following single exception. For the D-cache for `espresso` and `spice`, 64-byte lines occasionally show a splitting while 16-byte lines do not.

(iii) Changing the optimization level affects which layouts are bad, as seen in the following three examples. First, for the 4-kB I-cache for `espresso`, both full optimization (Figure 1a) and medium optimization (Figure 4a) have 3 bad layouts, but they are a different 3 layouts. Second, for the 64-kB I-cache for `espresso`, full optimization yields 2 bad layouts, medium optimization yields no bad layouts, and no optimization (Figure 5a) yields 6 bad layouts. Furthermore, the 2 bad layouts for full optimization are not among the 6 bad layouts for no optimization. Third, for the 64-kB I-cache for `spice`, 7 layouts are bad for both full optimization (Figure: 6a) and medium optimization (Figure 8a); of these 7 layouts, 6 are bad in both cases. No optimization (Figure 9a) yields 1 bad layout, which is not among the 8 bad layouts for full optimization and medium optimization.

(iv) Changing the input set usually does out change which layouts are bad, as the following two examples show. First, for the 4-kB I-cache for `espresso`, of the four bad layouts in Figure 2a, three are bad in Figure 1a and one is bad in Figure 3a. In addition, the bad layout in Figure 3a is bad for all three inputs set for the 16-kB I-cache. Second, for the D-cache for `fpppp` (Figures 13b and 14b), both input sets have the same 6 bad layouts for the 16-kB cache and the same 5 bad layouts for the 64-kB cache.

How does the miss variation affect system performance?

When the :mean measured miss rate m is between 0.3% to 7.0%, we often observed miss rates that varied from $0.6m$ to $1.8m$, which can effect system performance significantly. As a representative example, consider the 1-kB I-cache for espresso (Figure 2a). Here, $m = 2.3\%$, the lowest miss rate is 1.4% ($0.6m$) and the highest miss rate is 3.7% ($1.6m$). Assuming a cache miss penalty of 4 cycles, the best layout has a cache miss penalty of 0.06 CPI while the worst layout has a cache miss penalty of 0.15 CPI, for a difference of 0.09 CPI. For a processor with a base CPI of 0.7, this difference amounts to a 13% penalty on system performance.

To perform a "real-world" test of how layout affects execution time, we measured the execution time of `spice` on 21 layouts. We used a Sun SPARC 5, which has a 4-kB 32-byte line I-cache with a 6 cycle miss penalty and a 2-kB 16-byte line D-cache with a 4 cycle miss penalty. We measured the execution time for each layout ten times, discarded the longest and shortest times, and used the average of the remaining eight times. The (averaged) execution times ranged from 695.2 seconds to 727.4 seconds, with a mean of 708.0 seconds and a standard deviation of 8.5 seconds. Thus, execution times varied $\pm 2.3\%$ of the mean which would not normally be noticeable.,

We also compared the preceding execution time variation to the variation we would expect based on our previous measurements of miss variation. For spice on this cache configuration, the miss rate standard deviation was 9% for the I-cache and 2% for the D-cache,. We measured a 2.6% miss rate for the I-cache, a 38.4% miss rate for the D-cache, and found 35% of the instructions were loads/stores. Assuming a CPI of 1.3 without cache effects, we expect the CPI to be

$$1.3 + (0.026 \pm 9\%)(6) + (0.35)(0.38 \pm 2\%)(4) = 1.99 \pm 0.025$$

Thus we expected a variation of $\pm 2.5\%$ of the mean (+2 standard deviations), which is very close to the measured variation (2.3%).

One concern about the general validity of our results is that we examined a very small subset of all possible:layouts, and there is no guarantee that another subset of layouts would produce similar results. However, we have confidence in our results, as for both `doduc` and `fpppp` the results for 21 layouts versus 100 layouts are essentially identical.

6. Conclusion

In summary, our data show that miss rates vary considerably on direct-mapped caches, with a typical variation from $0.6m$ to $1.8m$ across just 21 layouts, where m is the measured mean miss rate. We observed many layouts that had consistently poor miss rates on different caches, but we found no consistently good layouts. Thus, when picking a layout, the problem is not so much picking a good layout, but rather not picking a bad layout. In practice, when linking a time-crucial program for a specific system, we recommend picking the best of five random layouts. Experimentally, we found a bad layout occurs less than ⅓ of the time, so picking the best of five layouts reduces the odds of a bad layout to under 0.5%.

We conclude with some suggestions for future work phrased as unanswered questions.

- Is there a method to analytically predict the miss variation? We have started to extend the gap model in this regard.

- Is there a practical method to find a good layout? An exhaustive search is impractical given the huge number of possible layouts. Furthermore, are there consistently a good layout? Based on our results, there are no consistently good layouts, only consistently average and bad ones. (We have ignored the experimental compiler/linker systems [PH90][McF89] which seem to work, as these systems are not generally available.)

- Do our results extend to other caches, programs, and machines? Our measurements were limited, as we have measured data for only five SPEC benchmark programs on one platform (SunOS 5.3, SPARC V8, SunOS cc). While we see no reason that other systems should have significantly different results, our results should be confirmed.

References

[AHH89] Anant Argawal, Mark Horowitz, and John Hennessy, An Analytical Cache Model, ACM *Transactions* on Computer *Systems*, 7(2):184-215, May 1989

[BC72] Jean-Loup Baer and R. Caughey, Segmentation and Optimization of Programs from Cyclic Structure Analysis, AFZPS, pages 23-36, 1972.

[CK93] Robert F. Cmelik and D. Keppel, Shade: A Fast Instruction Set Simulator for Execution Profiling, Technical Report TR-93-12, Sun Microsystenis Inc., July 1993

[FL91] Charles N. Fischer and Richard J. LeBlanc, *Crafting* a Compiler with C, Benjamin/Cummings, Reading, MA, 1991

[GKP89] R. Graham, D. E. Knuth, and O. Patashnik, Concrete Mathematics, Addison-Wesley, Reading, MA, 1989.

[Har88] Stephen J. Hartley, Compile-Time Program Restructuring in Multiprogrammed Virtual Memory Systems, ZEEE Transactions on *Software* Engineering, 14(11):1640-1644, November 1988

[Ker71] Brian W. Kernighan. Optimal Sequential Partitions of Graphs, *Journal* of the ACM, 18:34-40, January 1971.

[McF89] Scott McFarling, Program Optimization for Instruction Caches, ASPLOS-ZZZ, Boston, MA, April 3-6, 1989.

[OMHL93] Douglas B. Orr, Robert W. Mecklenburg, Peter J. Hoogenboom, and Jay Lepreau, Dynamic Program Monitoring and Transformation Using the OMOS Object Server, Hawaii *International* Conference of System Sciences, pages 232-241, January 1993. Also available as technical report UUCS-92-034.

[PH90] Karl Pettis and Robert C. Hansen, Profile Guided Code Positioning, Programming Language Design and Implementation, pages 16-27, White Plains, NY, June 20-22, 1990

[Quo94] Russell W. Quong, Expected I-cache Miss Rates via the Gap Model, International Symposium on Computer Architecture, pages 372-383, April 18-21, 1994

[Sar89] Vivek Sarkar, Determining Average Program Execution Times and their Variance, Programming Language Design and Implementation, pages 298-312, 1989

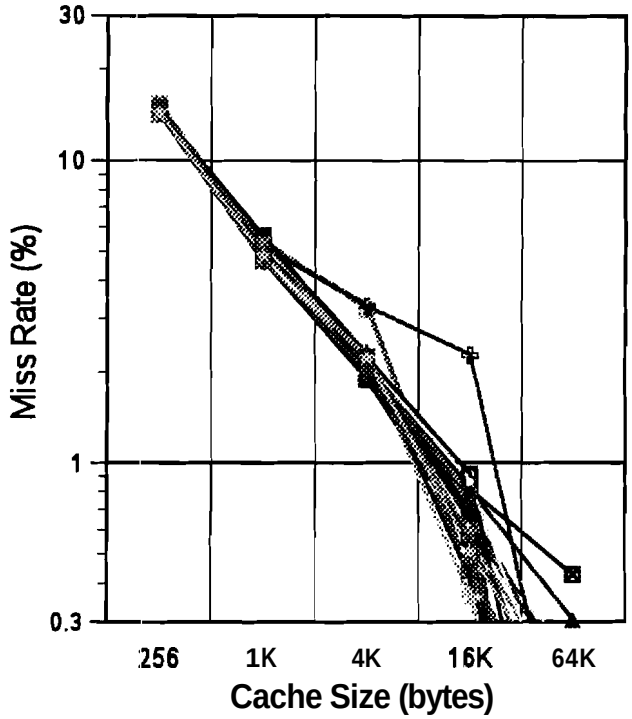

Figure 1a: Instruction Cache miss rate for espresso, first input set, 16 byte lines

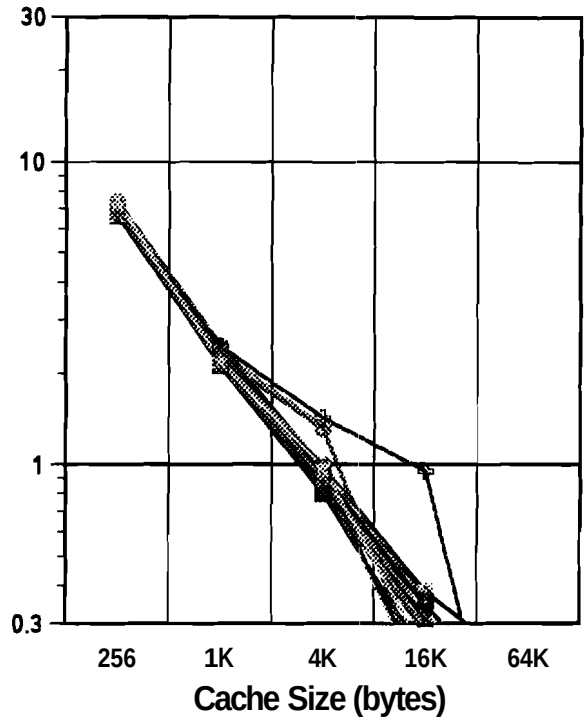

Figure 1c: Instruction Cache miss rate for espresso, first input set, 64 byte lines

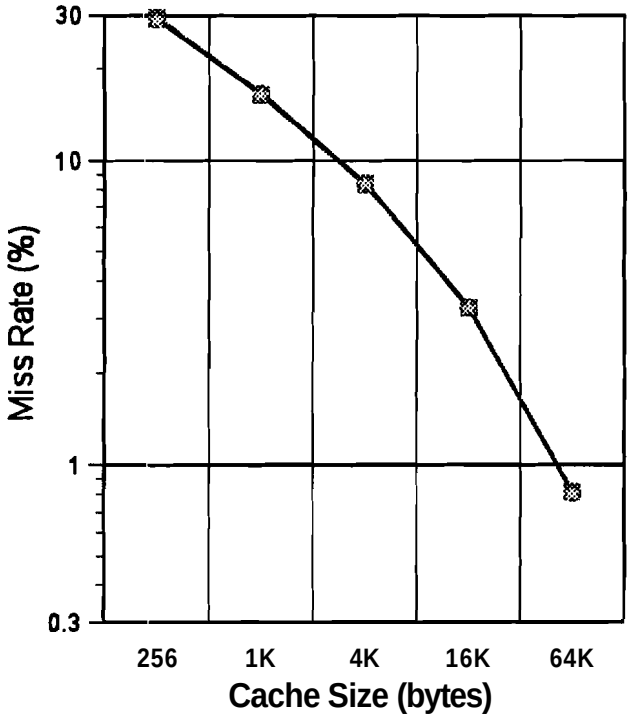

Figure 1b: Data Cache miss rate for espresso, first input set, 16 byte lines

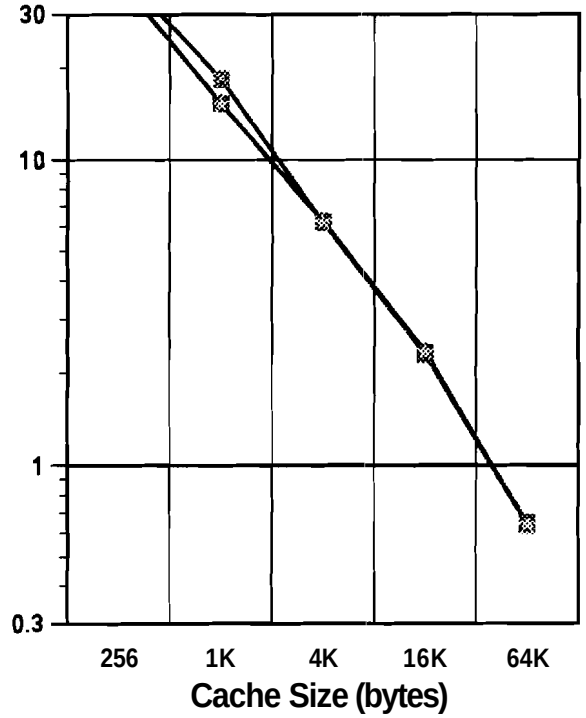

Figure 1d: Data Cache miss rate for espresso, first input set, 64 byte lines

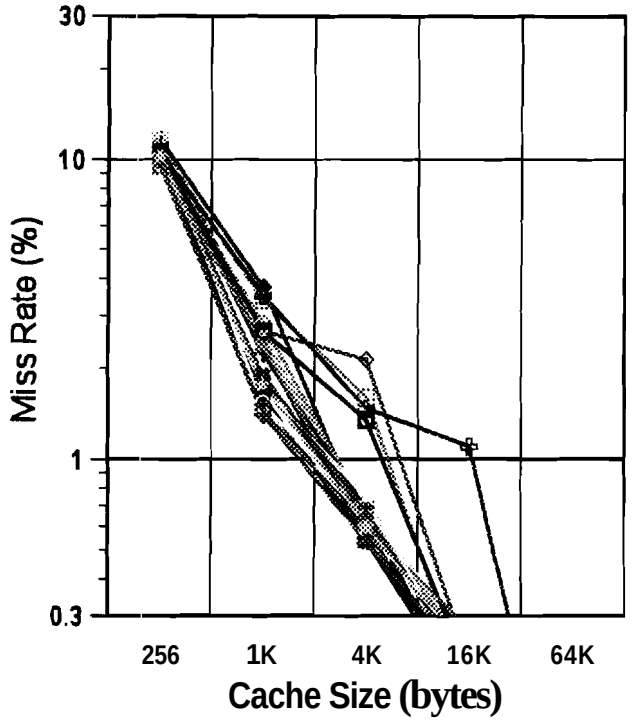

Figure 2a: Instruction Cache miss rate for `espresso`, second input set, 16 byte lines

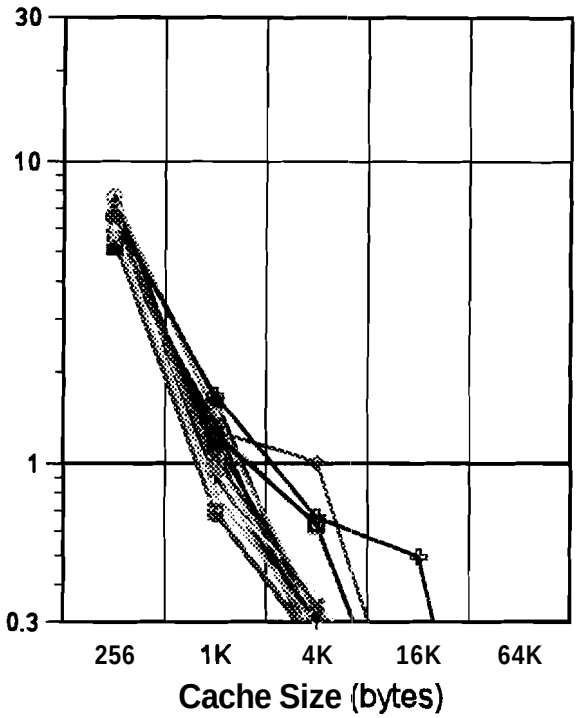

Figure 2c: Instruction Cache miss rate for `espresso`, second input set, 64 byte lines

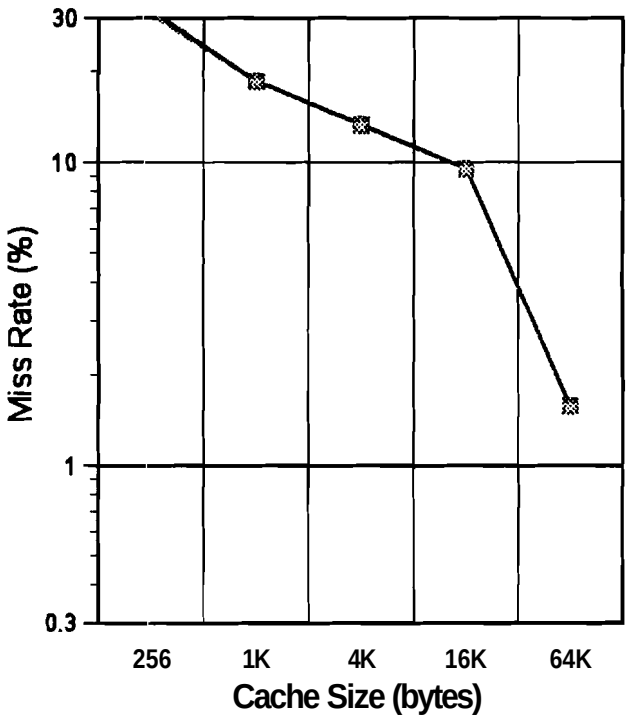

Figure 2b: Data Cache miss rate for `espresso`, second input set, 16 byte lines

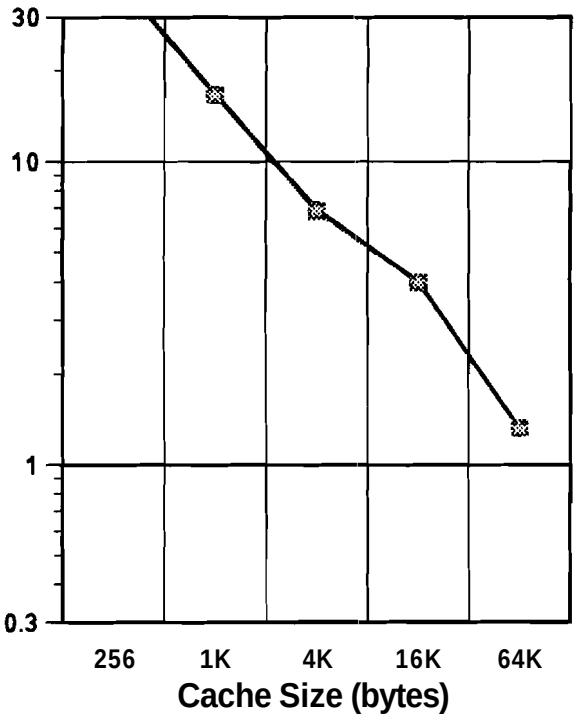

Figure 2d: Data Cache miss rate for `espresso`, second input set, 64 byte lines

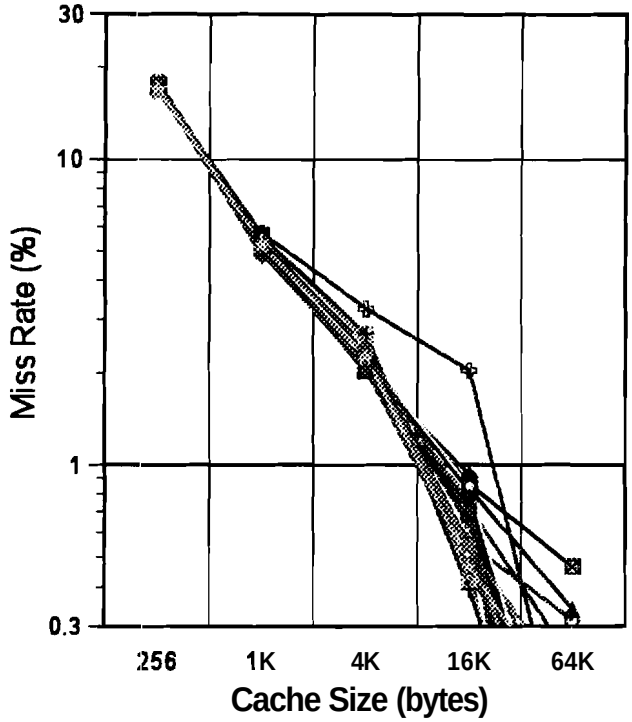

Figure 3a: Instruction Cache miss rate for espresso, third input set, 16 byte lines

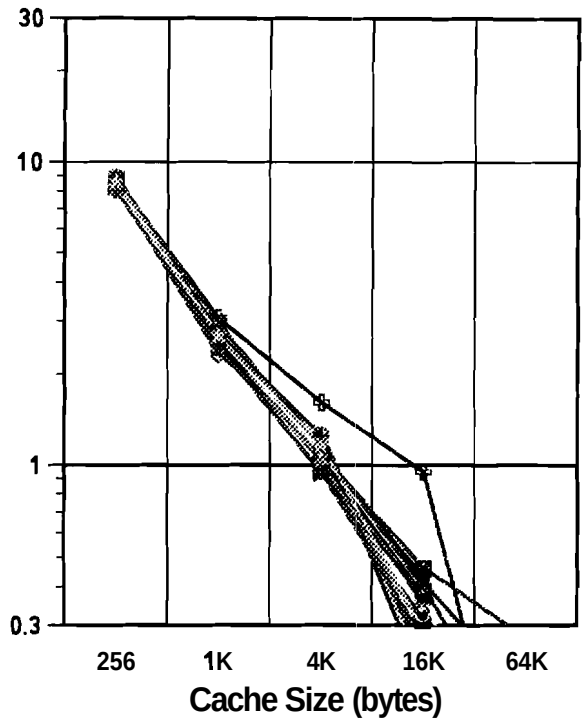

Figure 3c: Instruction Cache miss rate for espresso, third input set, 64 byte lines

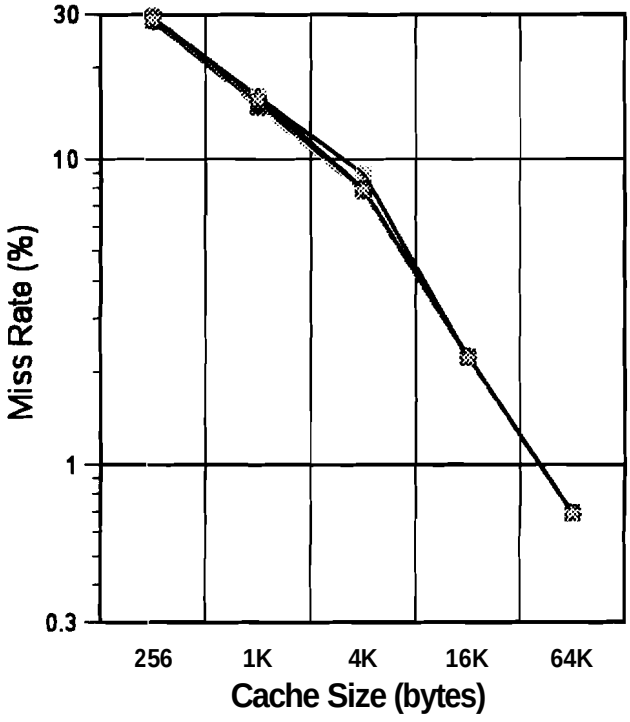

Figure 3b: Data Cache miss rate for espresso, third input set, 16 byte lines

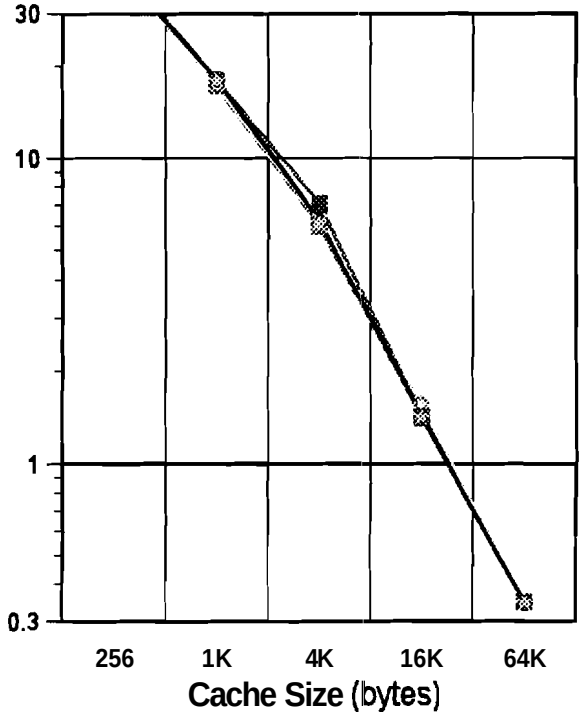

Figure 3d: Data Cache miss rate for espresso, third input set, 64 byte lines

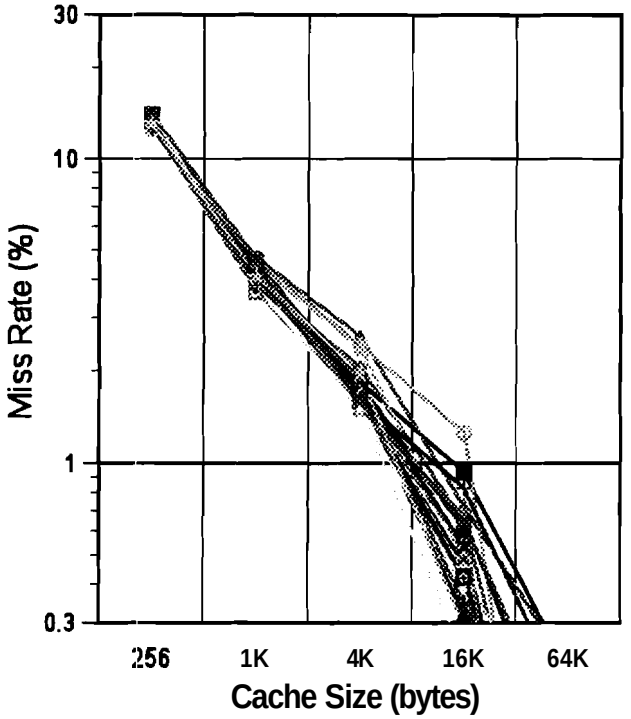

Figure 4a: Instruction Cache miss rate for espresso, medium optimization, **16** byte lines

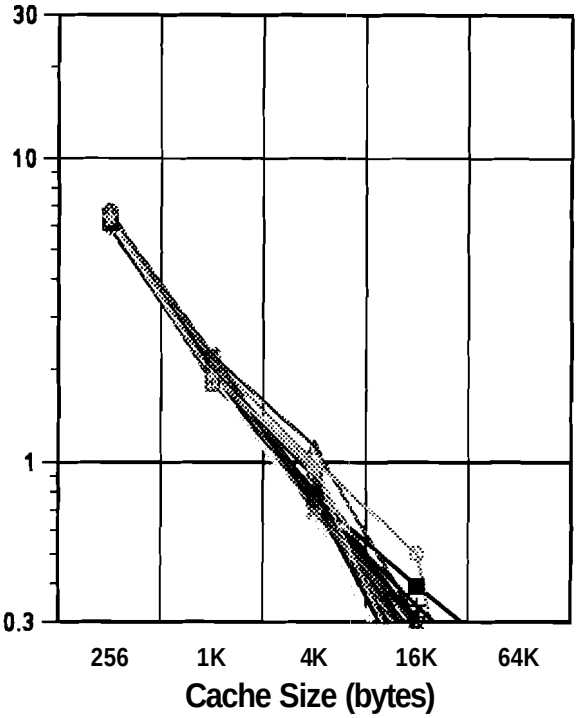

Figure 4c: Instruction Cache miss rate for espresso, medium optimization, **64** byte lines

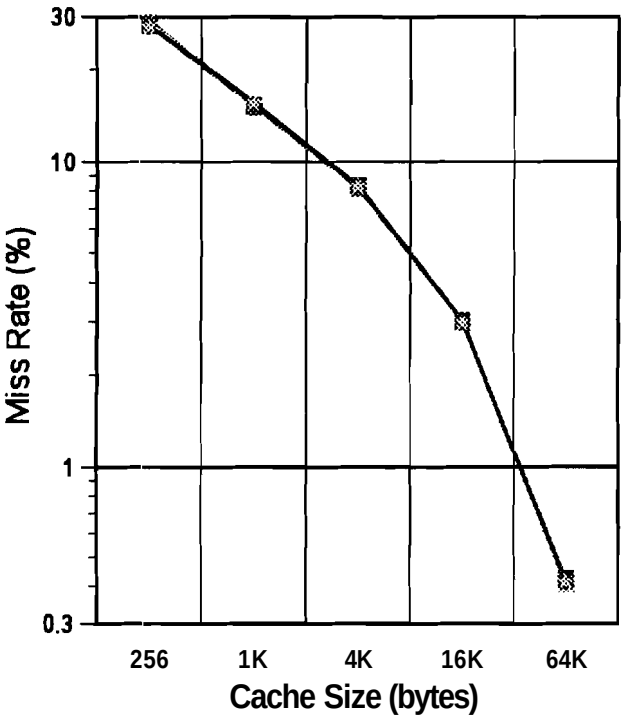

Figure 4b: Data Cache miss rate for espresso, medium optimization, **16** byte lines

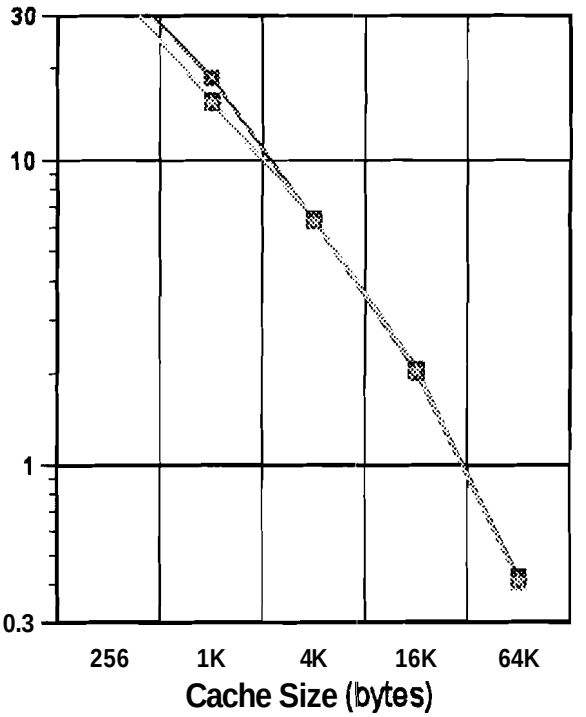

Figure 4d: Data Cache miss rate for espresso, medium optimization, **64** byte lines

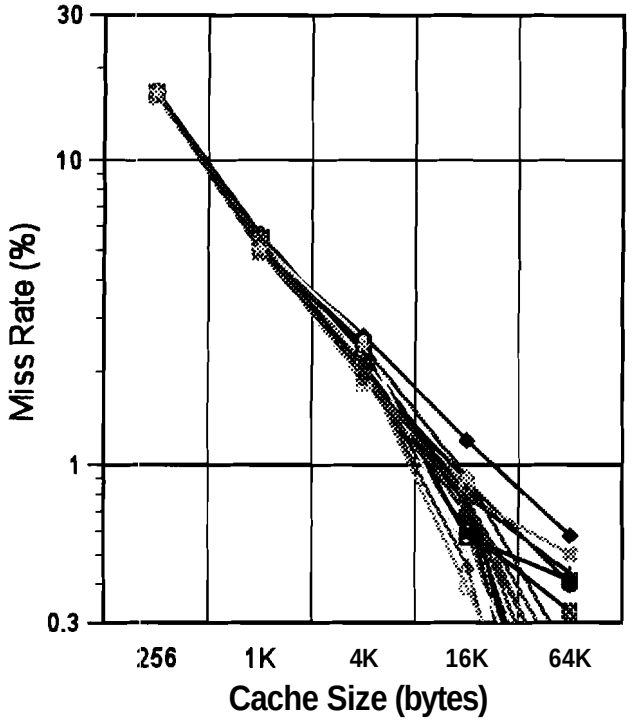

Figure 5a: Instruction Cache miss rate for espresso, no optimization, 16 byte lines

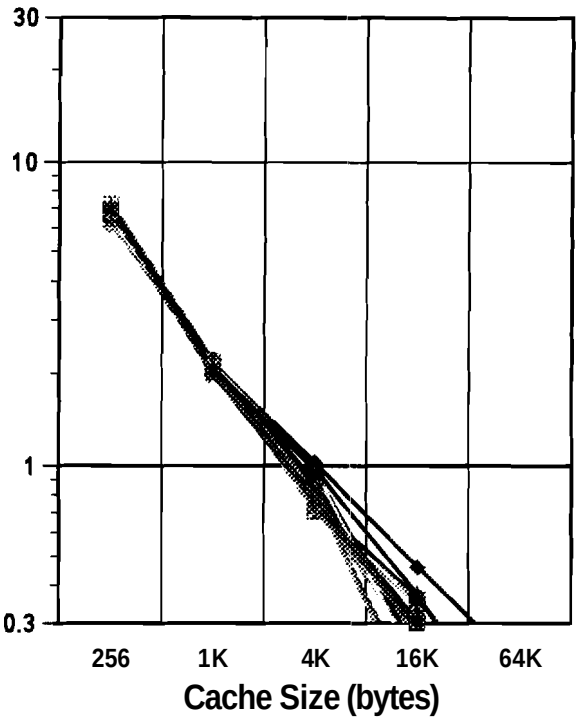

Figure 5c: Instruction Cache miss rate for espresso, no optimization, 64 byte lines

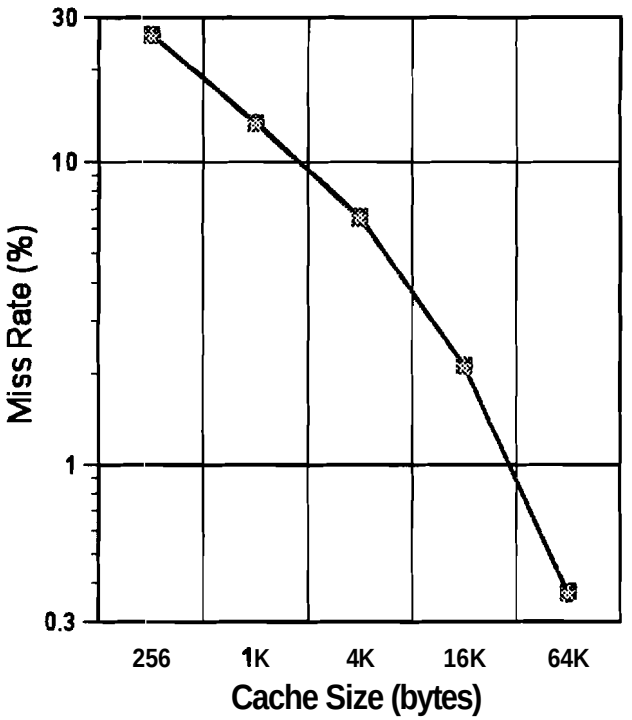

Figure 5b: Data Cache miss rate for espresso, no optimization, 16 byte lines

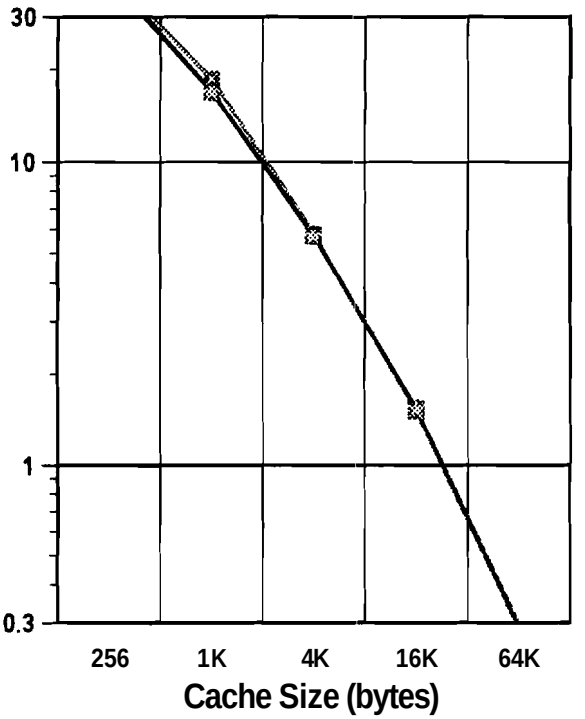

Figure 5d: Data Cache miss rate for espresso, no optimization, 64 byte lines

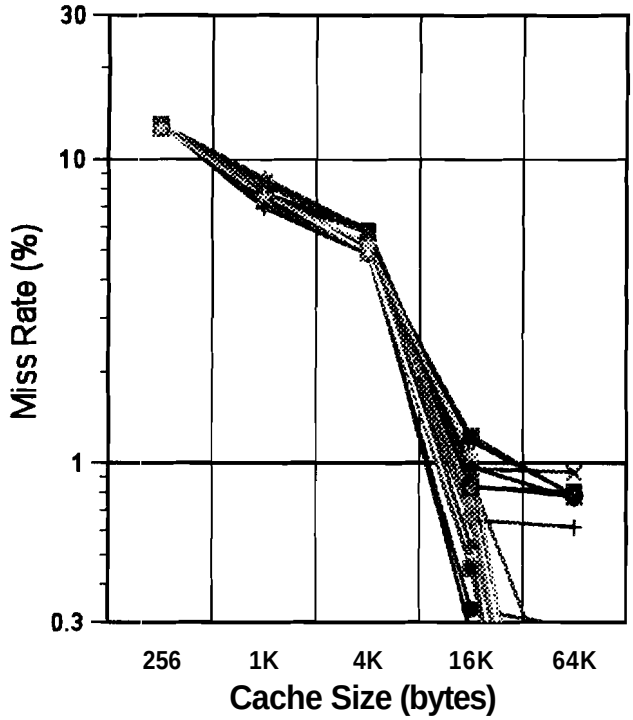

Figure 6a: Instruction Cache miss rate for `spice`, 16 byte lines

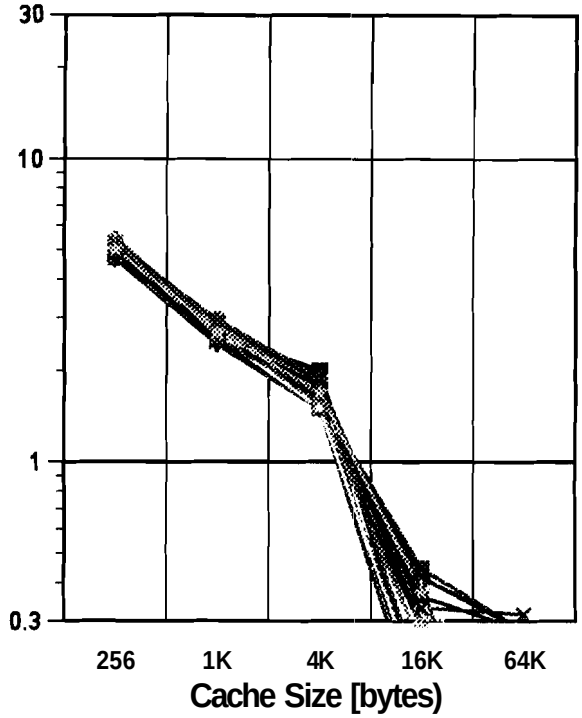

Figure 6c: Instruction Cache miss rate for `spice`, 64 byte lines

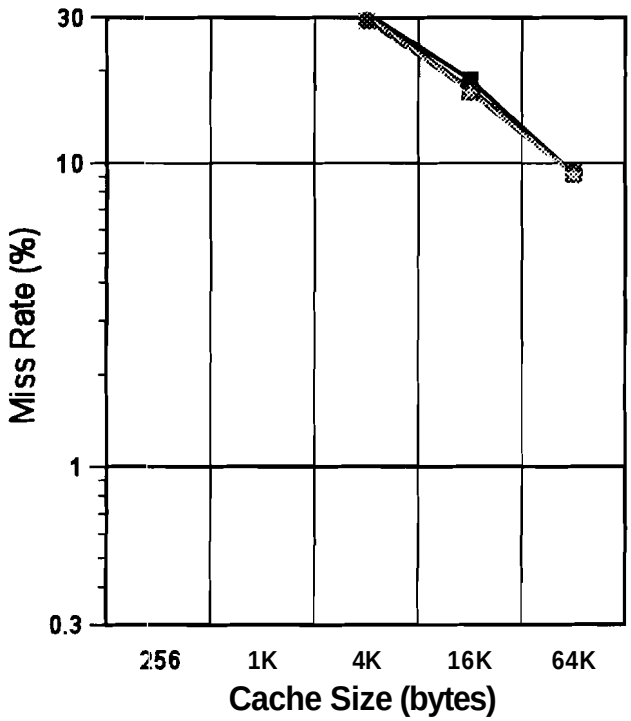

Figure 6b: Data Cache miss rate for `spice`, 16 byte lines

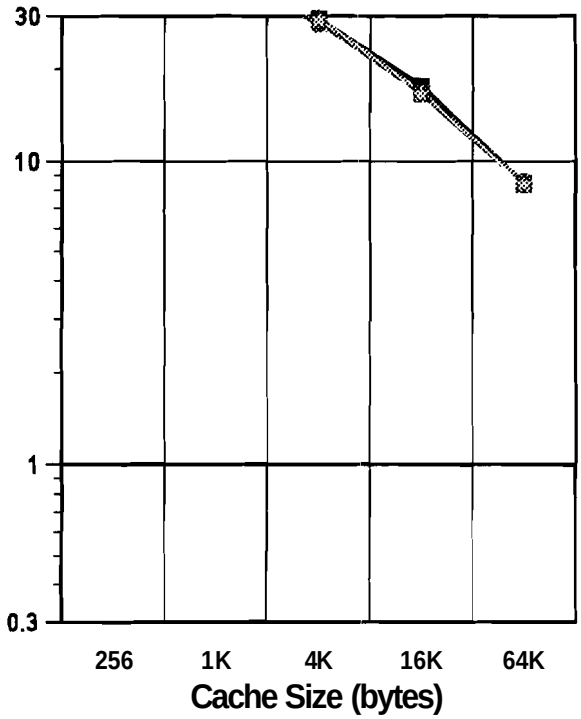

Figure 6d: Data Cache miss;rate for `spice`, 64 byte lines

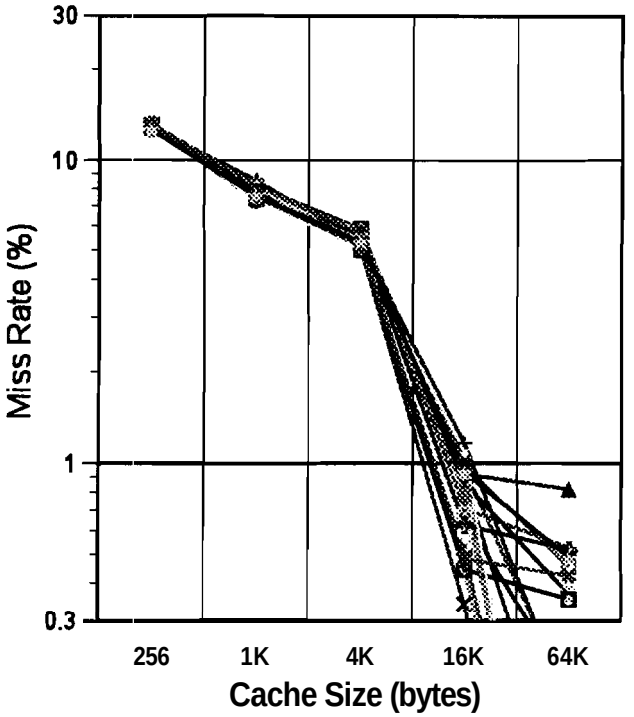

Figure 7a: Instruction Cache miss rate for spice, 16 byte lines, procedure-level rearrangenient

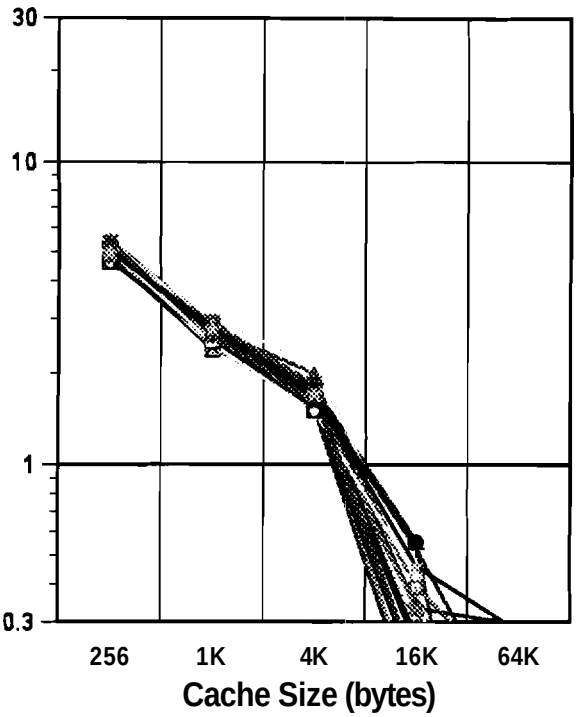

Figure 7c: Instruction Cache miss rate for spice, 64 byte lines, procedure-level rearrangement

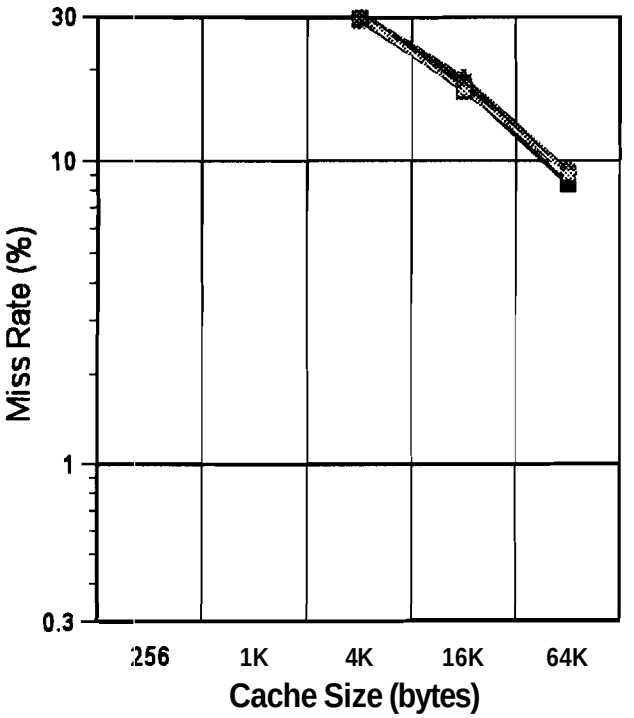

Figure 7b: Data Cache miss rate for spice, 16 byte lines, procedure-level rearrangement

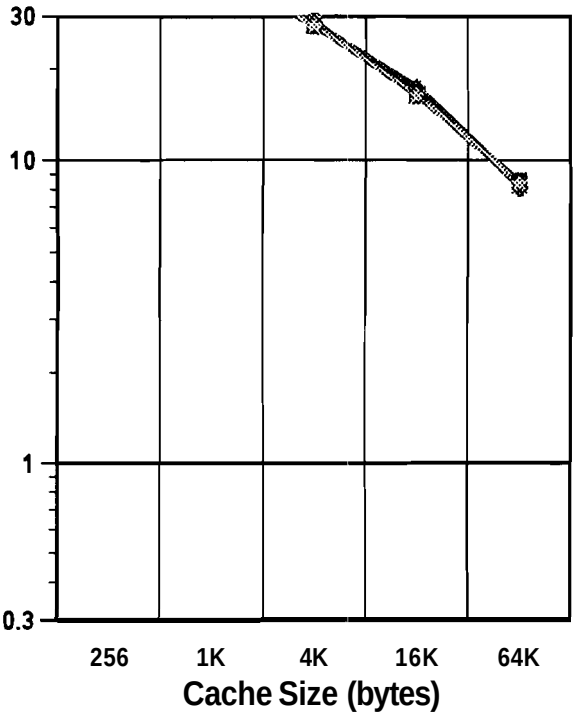

Figure 7d: Data Cache miss rate for spice, 64 byte lines, procedure-level rearrangement

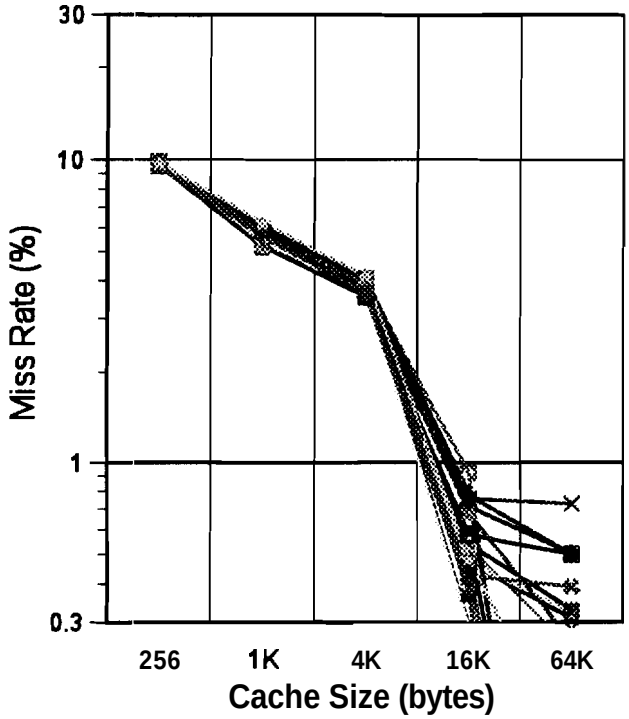

Figure 8a: Instruction Cache miss rate for spice, 16 byte lines, medium optimization

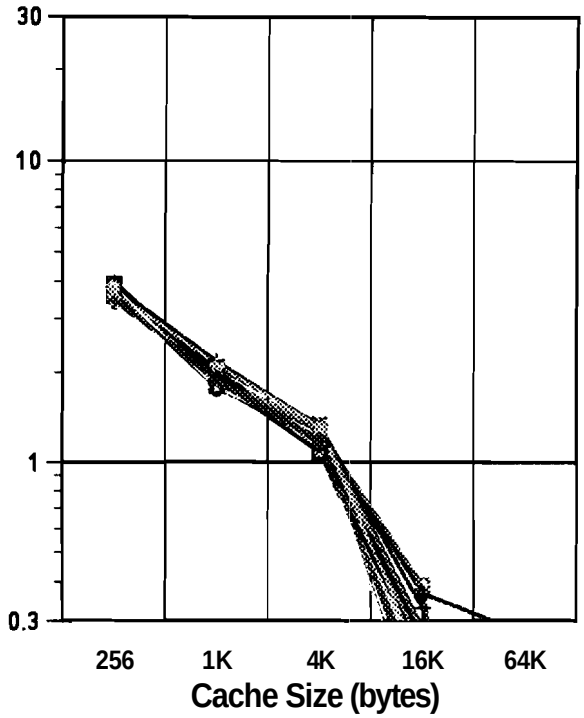

Figure 8c: Instruction Cache miss rate for spice, 64 byte lines, medium optimization

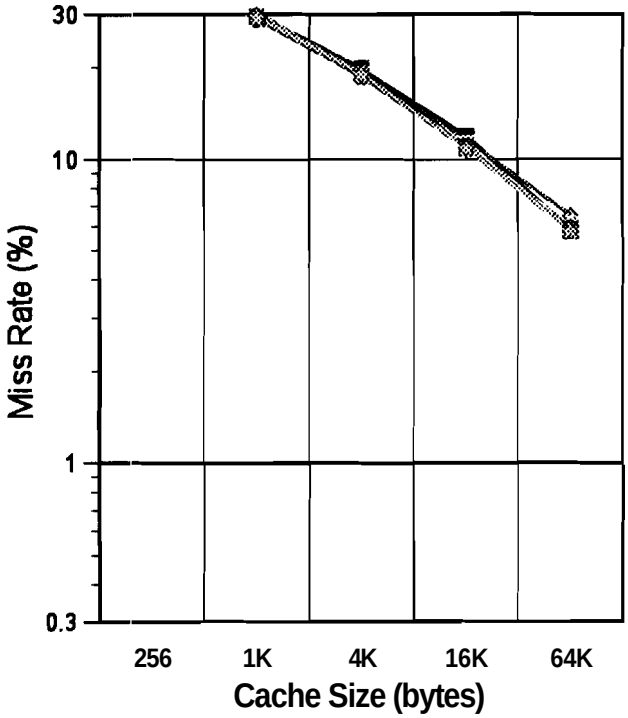

Figure 8b: Data Cache miss rate for spice, 16 byte lines, medium optimization

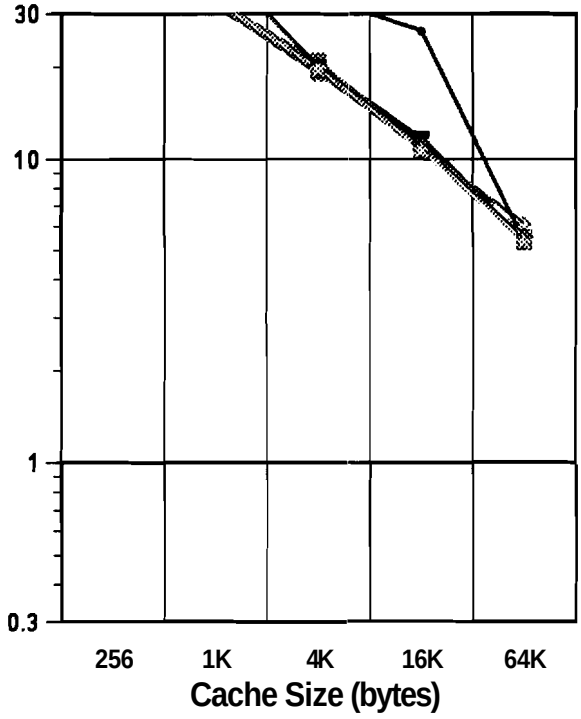

Figure 8d: Data Cache miss rate for spice, 64 byte lines, medium optimization

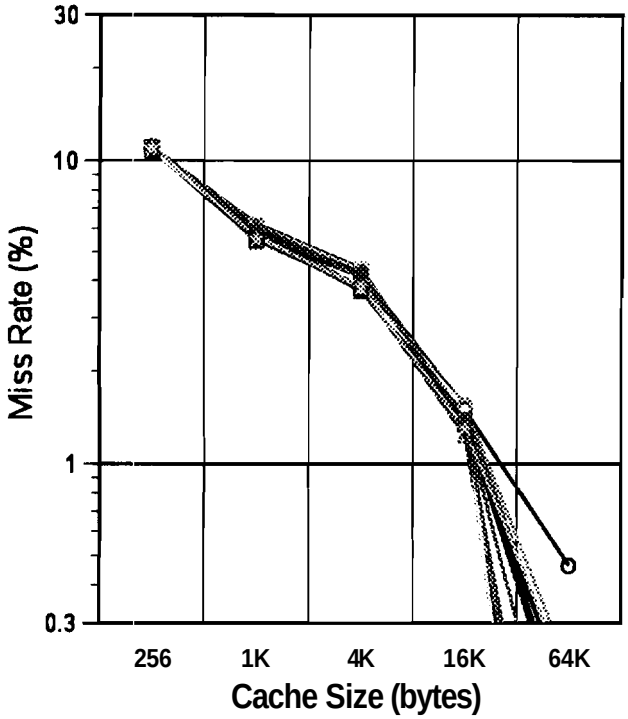

Figure 9a: Instruction Cache miss rate for s p i c e , no optimization, 16 byte lines

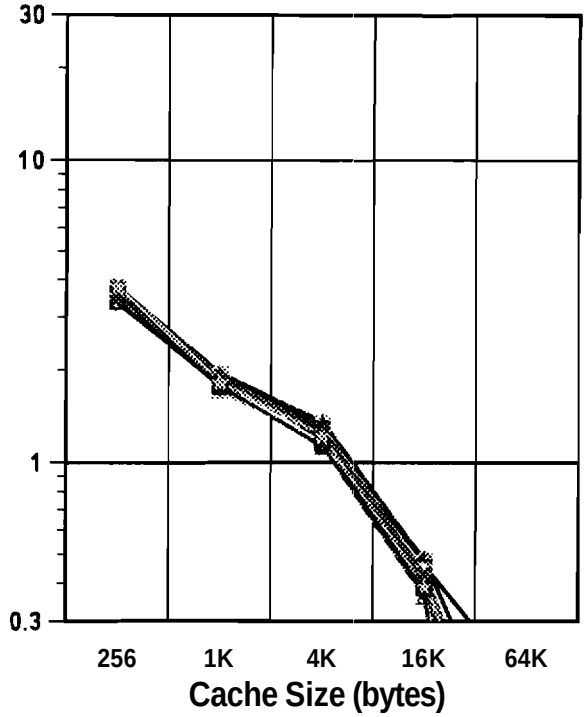

Figure 9c: Instruction Cache miss rate for s p i c e , no optimization, 64 byte lines

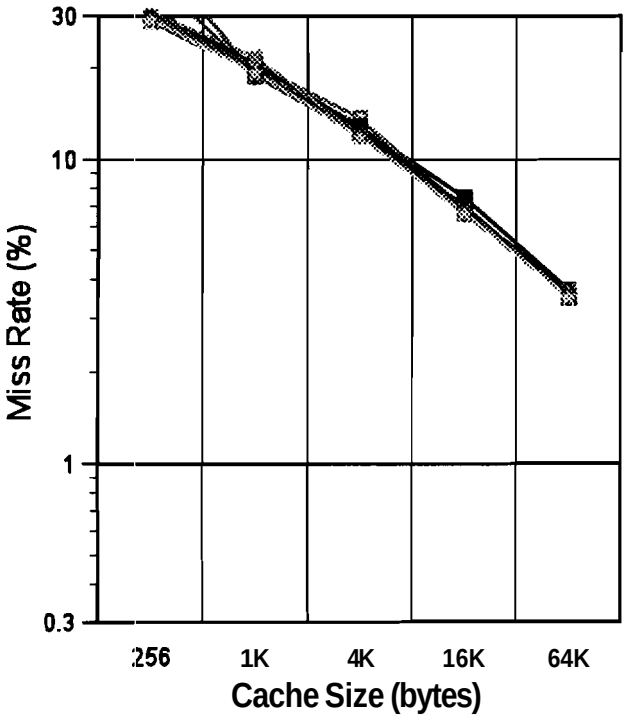

Figure 9b: Data Cache miss rate for s p i c e , no optimization, 16 byte lines

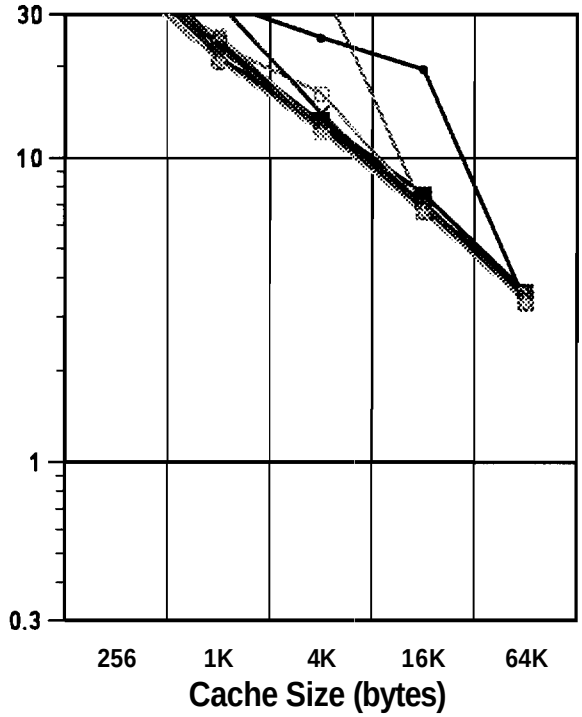

Figure 9d: Data Cache miss rate for s p i c e , no optimization, 64 byte lines

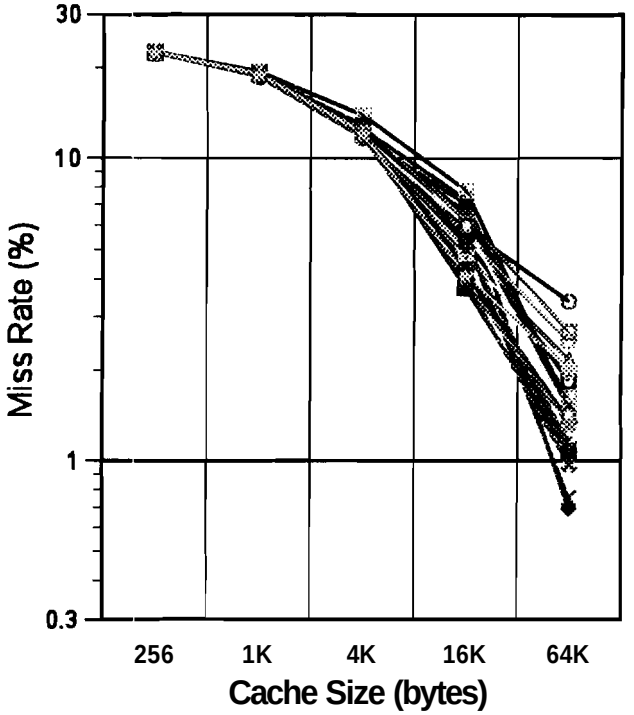

Figure 10a: Instruction Cache miss rate for **doduc**, 16 byte lines

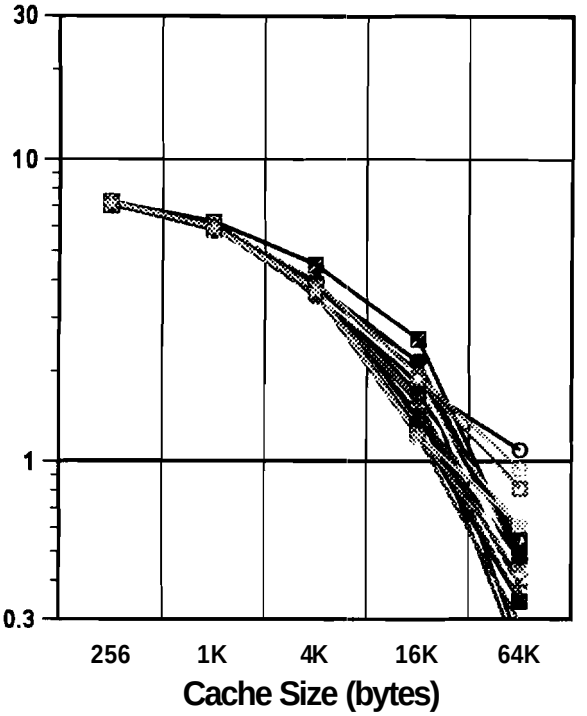

Figure 10c: Instruction Cache miss rate for **doduc, 64** byte lines

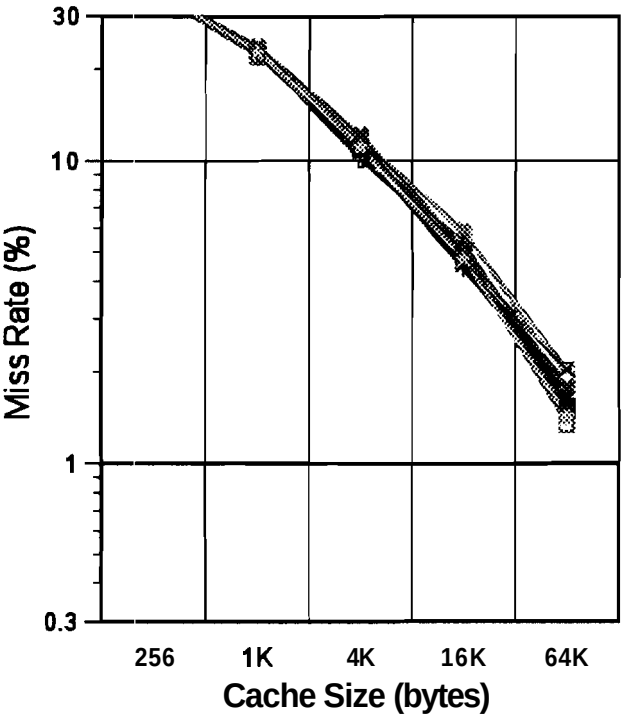

Figure 10b: Data Cache miss rate for **doduc, 16** byte lines

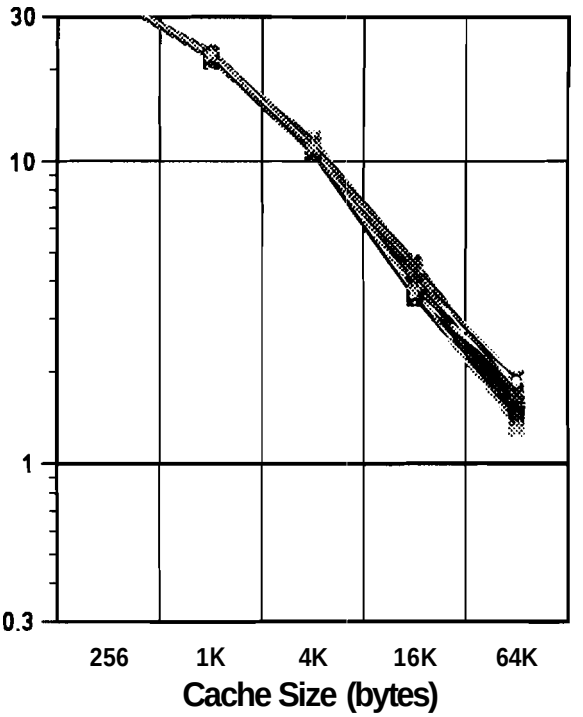

Figure 10d: Data Cache miss rate for **doduc, 64** byte lines

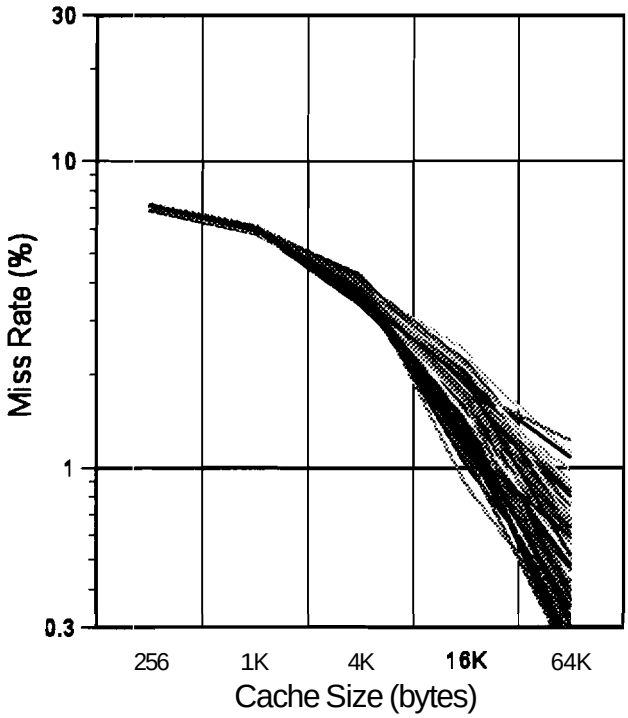

Figure 11a: Instruction Cache miss rate for doduc, 100 layouts, 16 byte lines

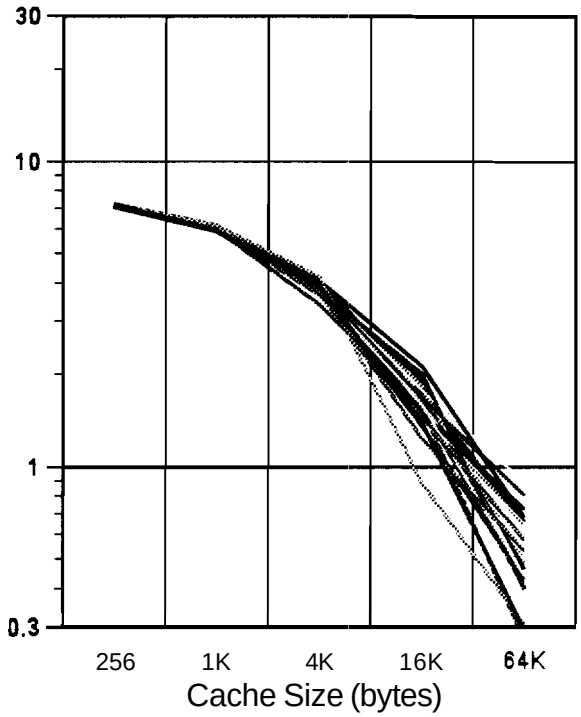

Figure 11c: Instruction Cache Miss rate for doduc, 100 layouts, 64 byte lines

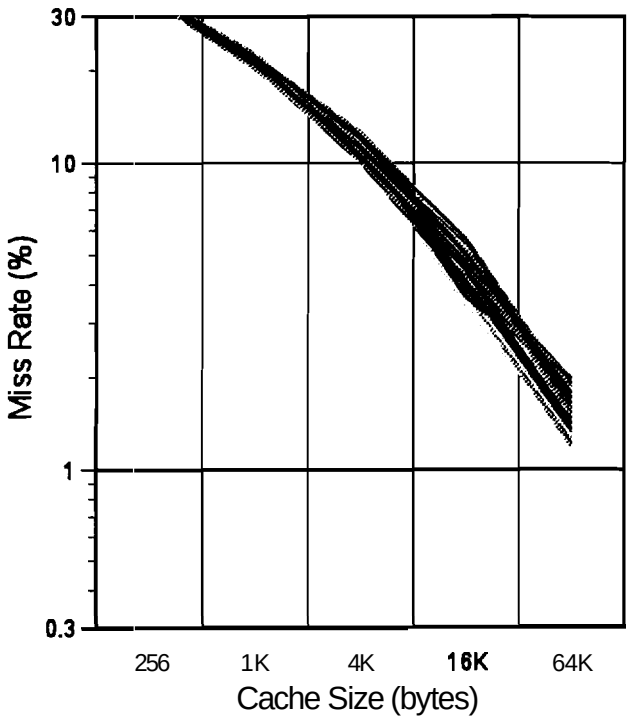

Figure 11b: Data Cache miss rate for doduc, 100 layouts, 64 byte lines

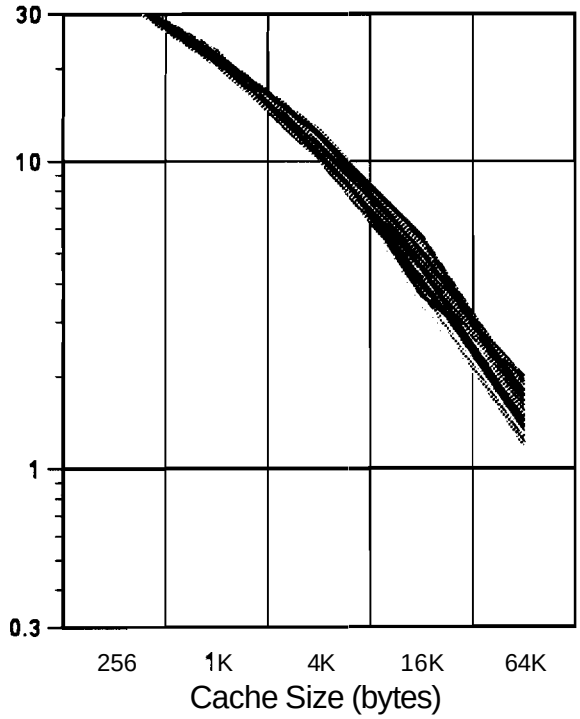

Figure 11d: Data Cache miss rate for doduc, 100 layouts, 64 byte lines

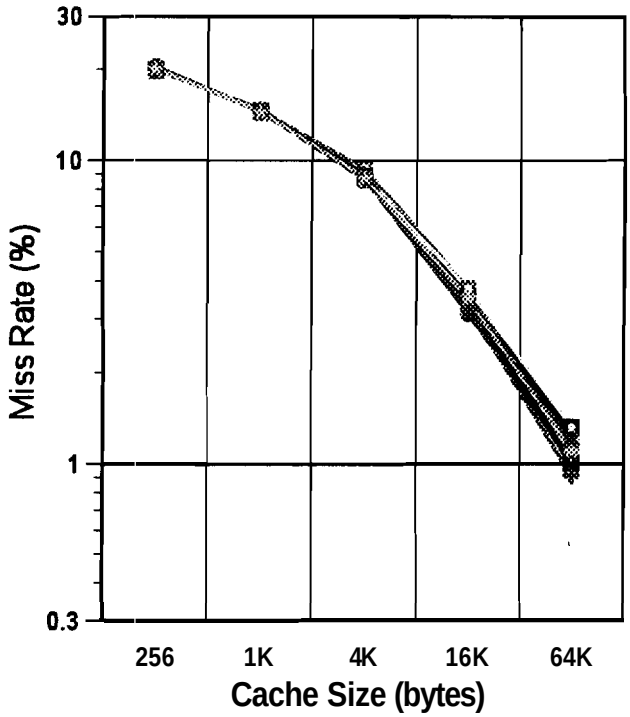

Figure 12a: Instruction Cache miss rate for gcc, 16 byte lines

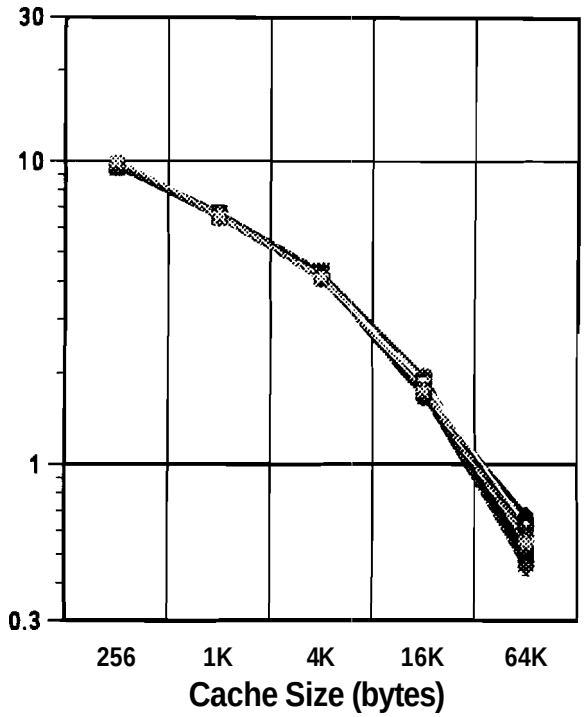

Figure 12c: Instruction Cache miss rate for gcc, 64 byte lines

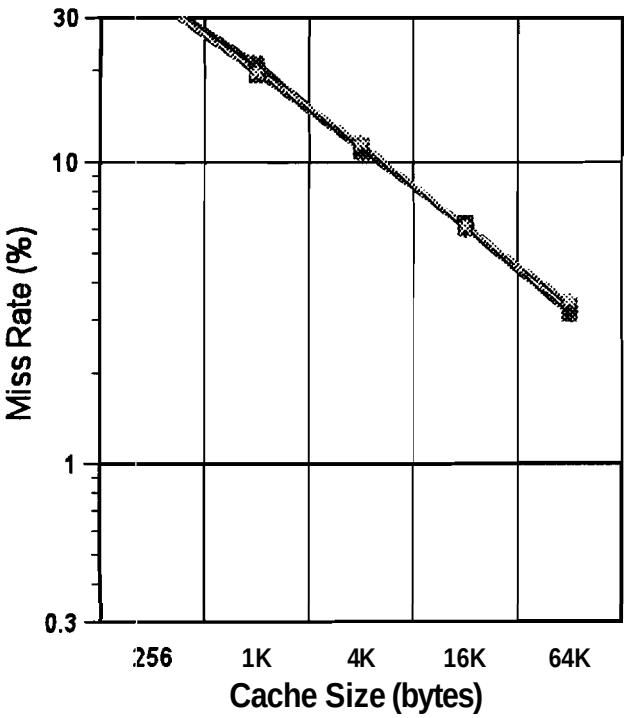

Figure 12b: Data Cache miss rate for gcc, 16 byte lines

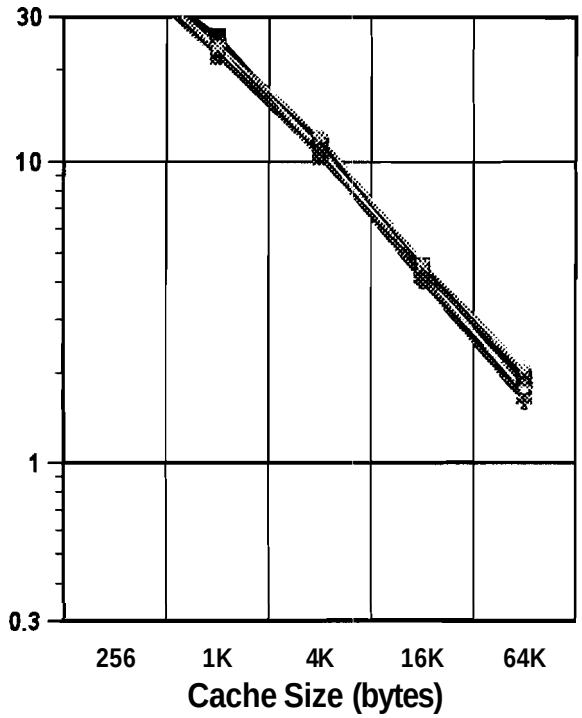

Figure 12d: Data Cache miss rate for gcc, 64 byte lines

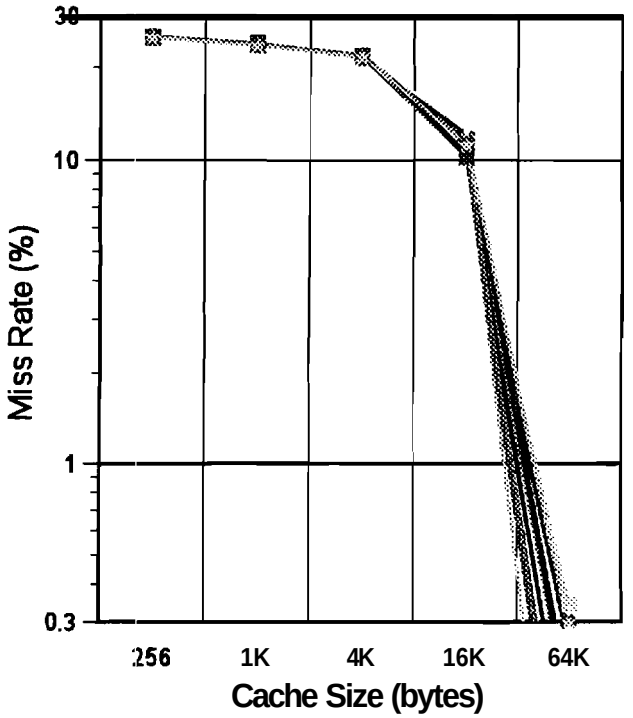

Figure 13a: Instruction Cache miss rate for **fpppp,** 16 byte lines

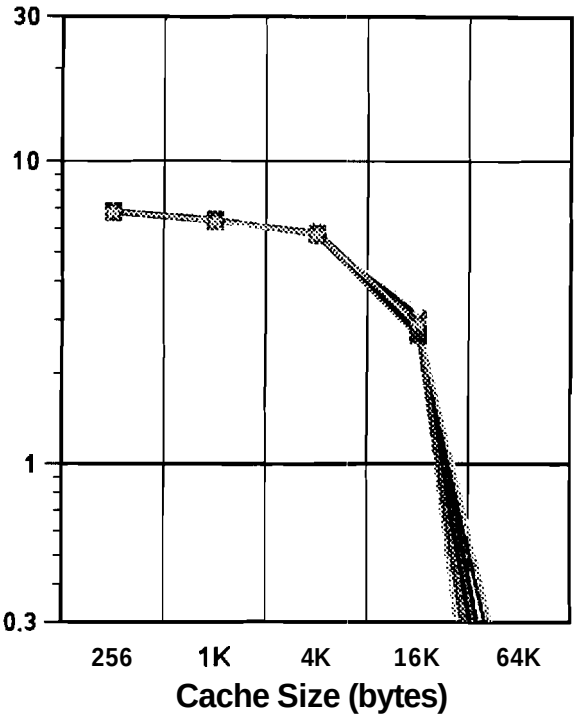

Figure 13c: Instruction Cache miss rate for **fpppp, 64** byte lines

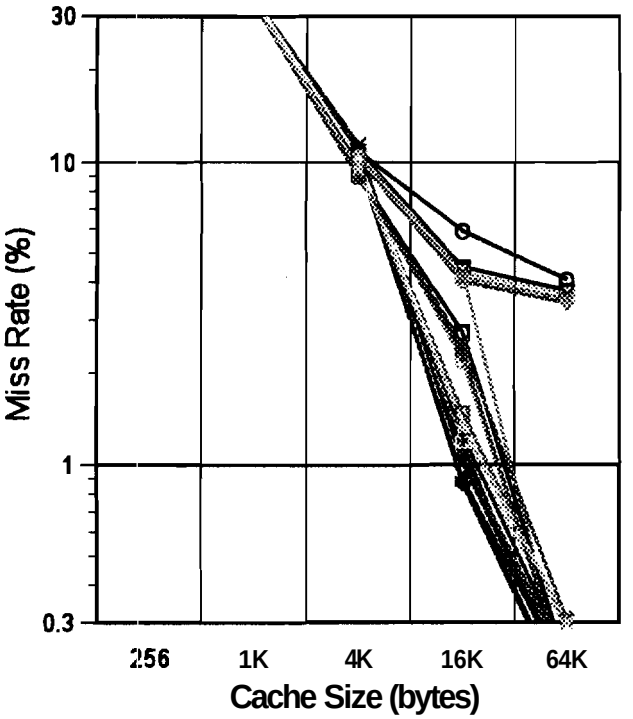

Figure 13b: Data Cache miss rate for `fpppp`, **16** byte lines

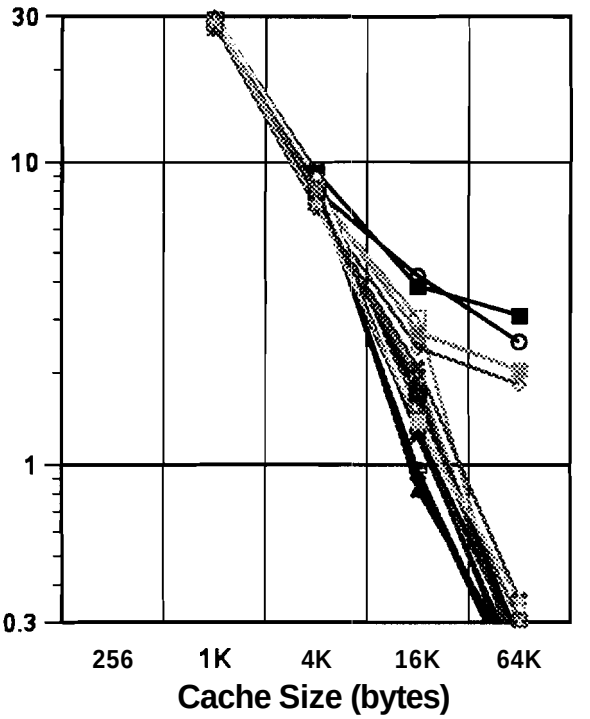

Figure 13d: Data Cache miss rate for **fpppp, 64** byte lines

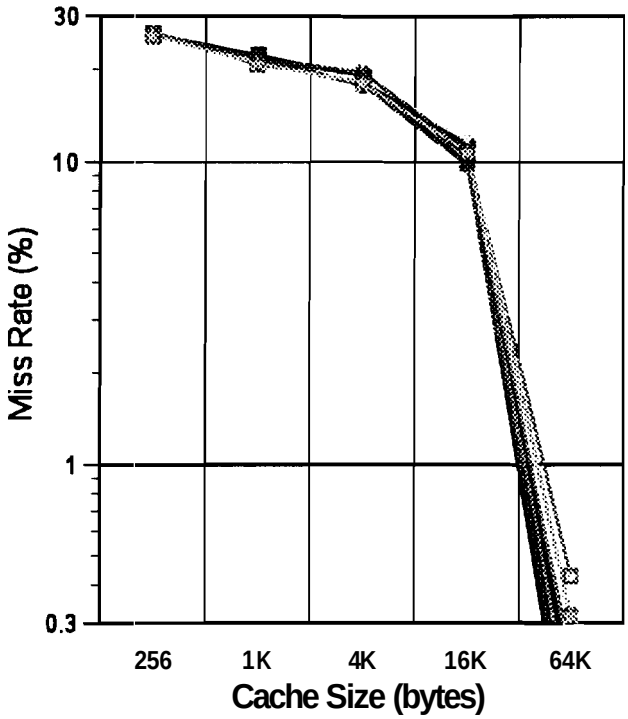

Figure 14a: Instruction Cache miss rate for fpppp, second input set, 16 byte lines

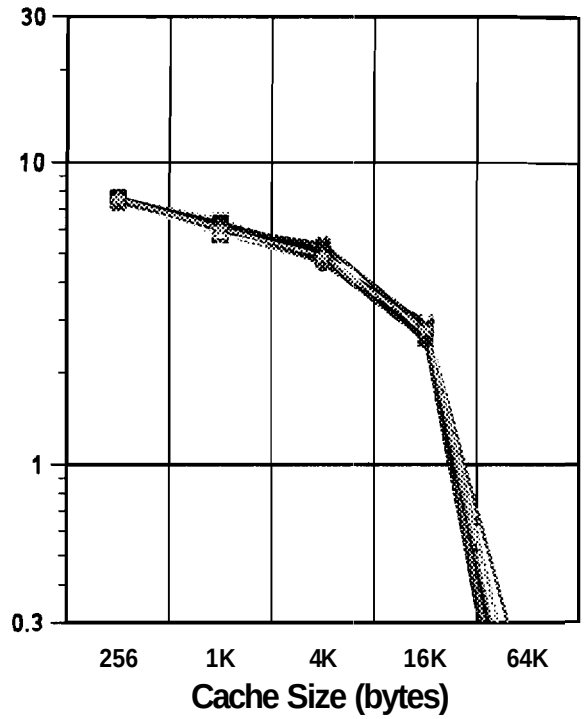

Figure 14c: Instruction Cache miss rate for fpppp, second input set, 64 byte lines

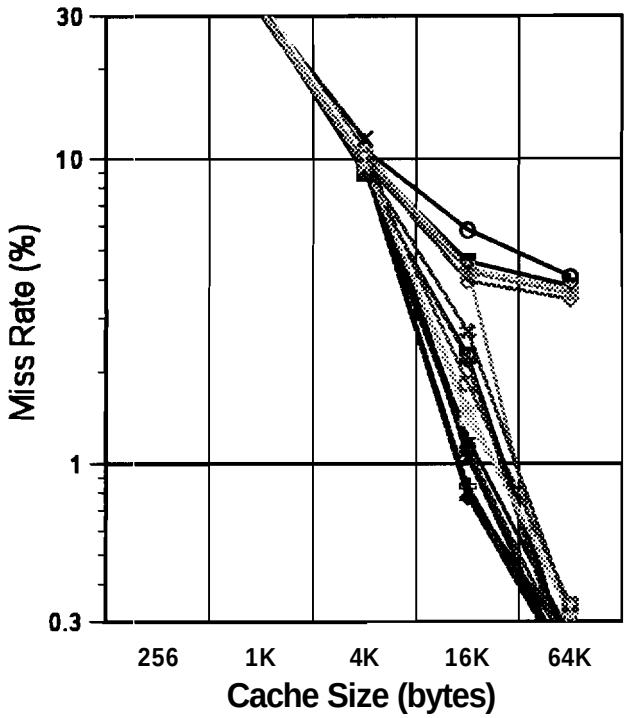

Figure 14b: Data Cache miss rate for fpppp, second input set, 16 byte lines

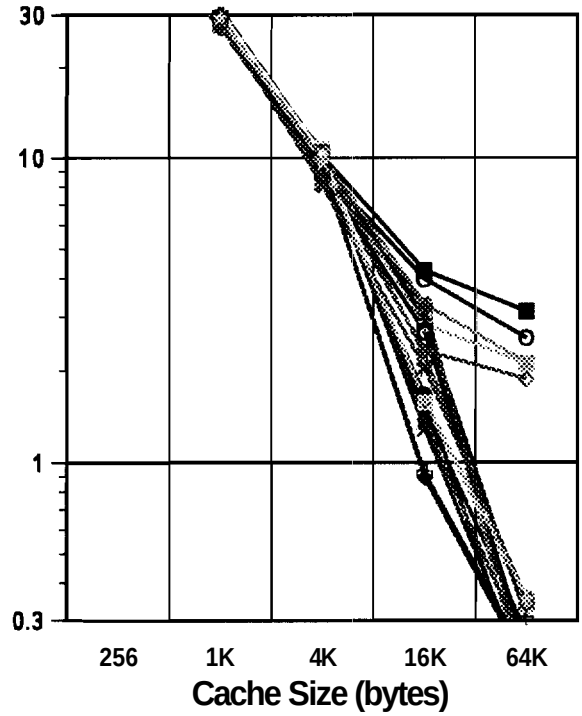

Figure 14d: Data Cache miss rate for fpppp, second input set, 64 byte lines

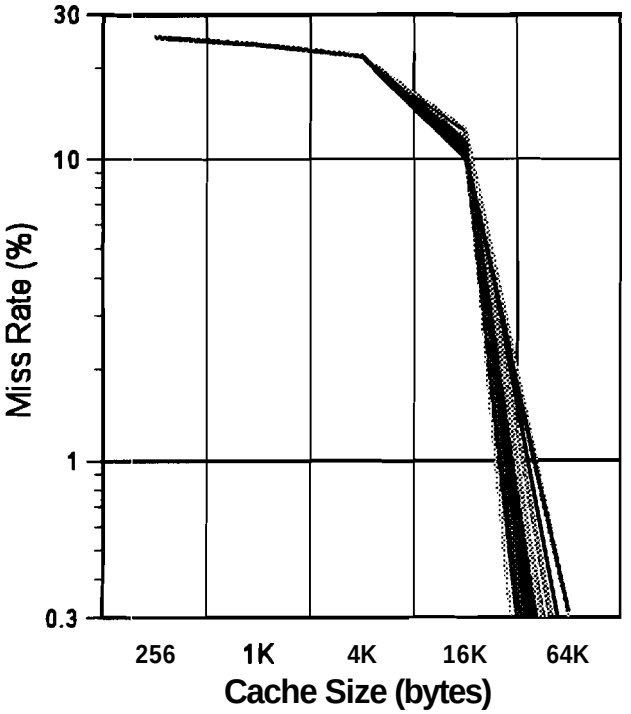

Figure 15a: Instruction Cache miss rate for fpppp, 100 layouts, 16 byte lines

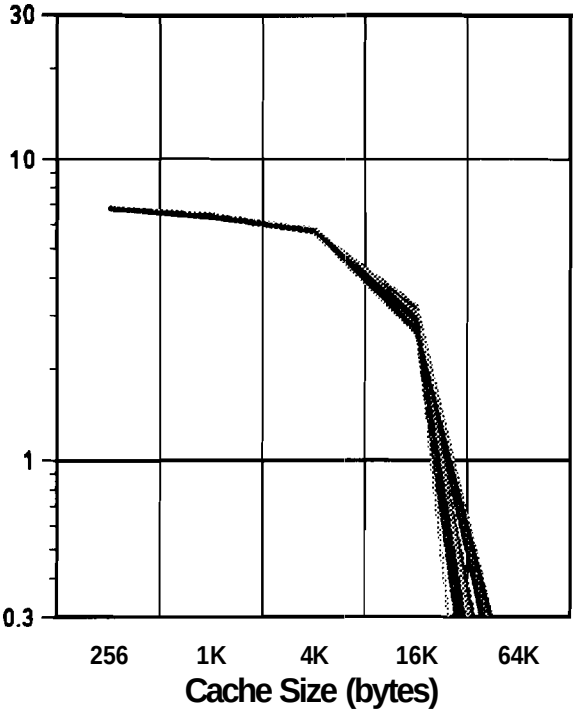

Figure 15c: Instruction Cache miss rate for fpppp, 100 layouts, 64 byte lines

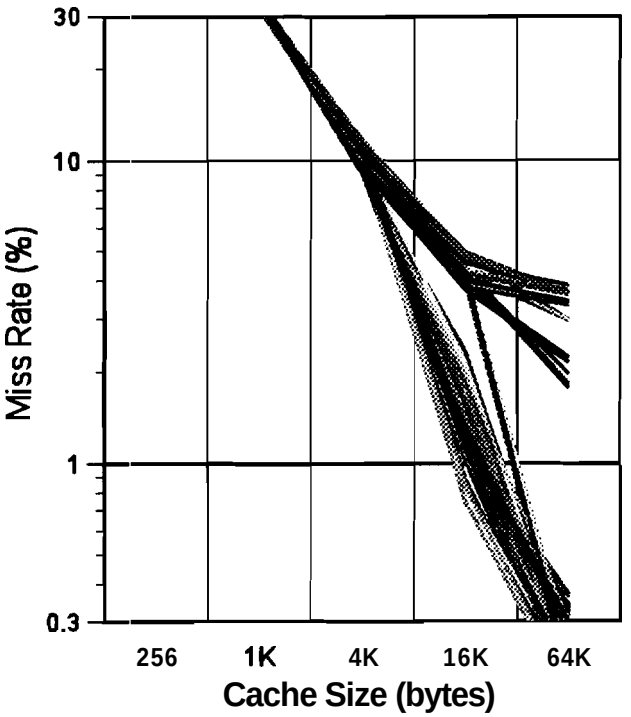

Figure 15b: Data Cache miss rate for fpppp, 100 layouts, 16 byte lines

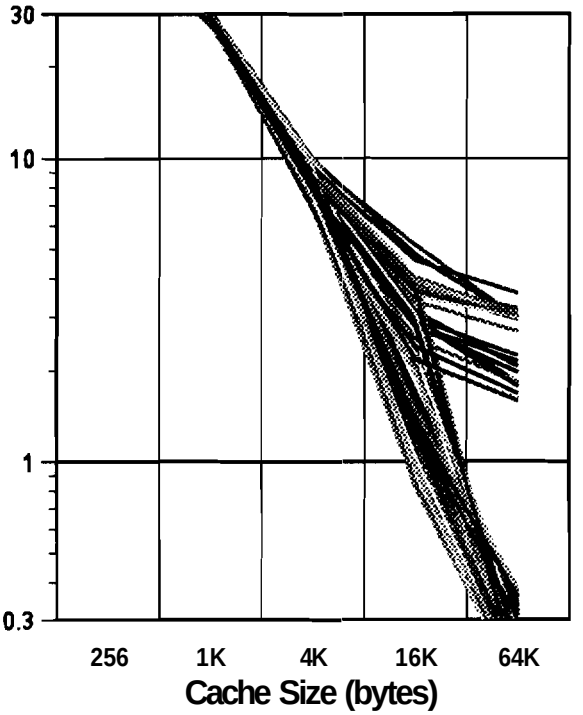

Figure 15d: Data Cache miss rate for fpppp, 100 layouts, 64 byte lines

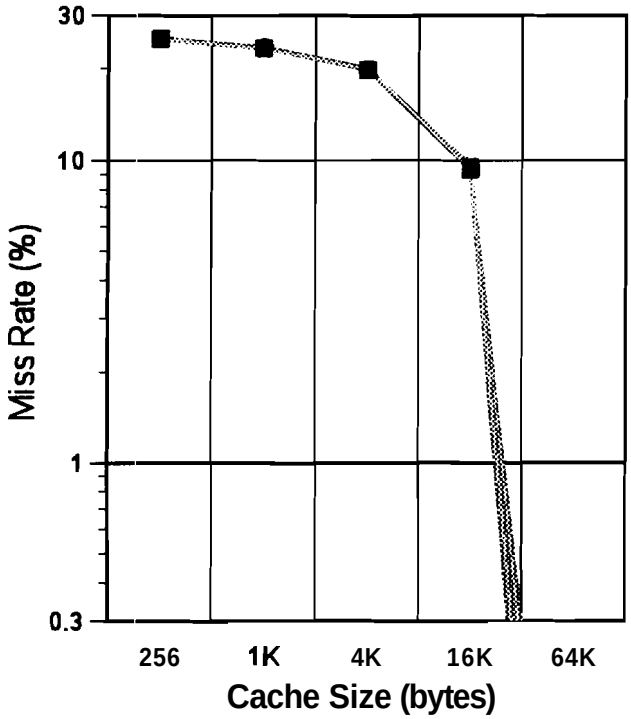

Figure 16a: Instruction Cache miss rate for fpppp, 4-way associative, 16 byte lines

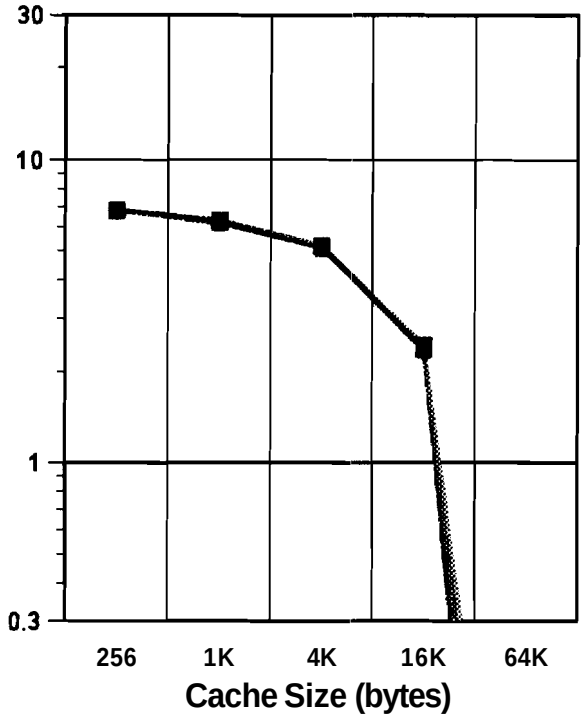

Figure 16c: Instruction Cache miss rate for fpppp, 4-way associative, 64 byte lines

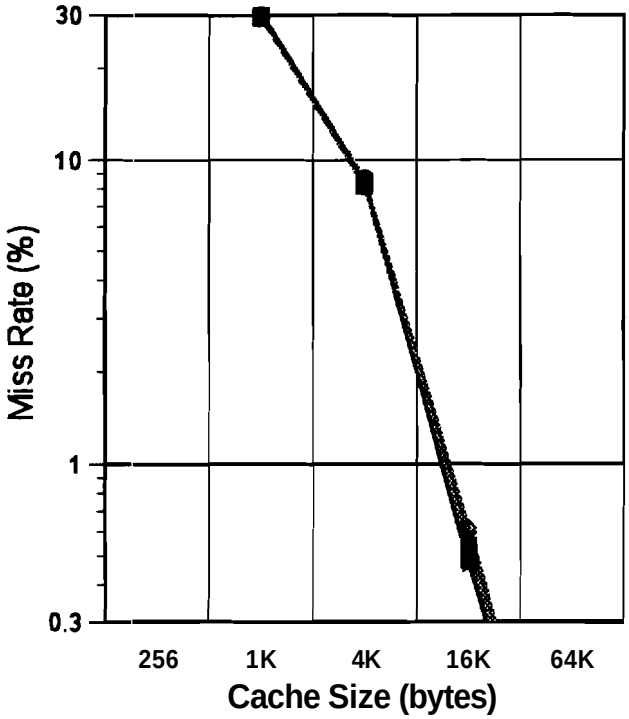

Figure 16b: Data Cache miss rate for fpppp, 4-way associative, 16 byte lines

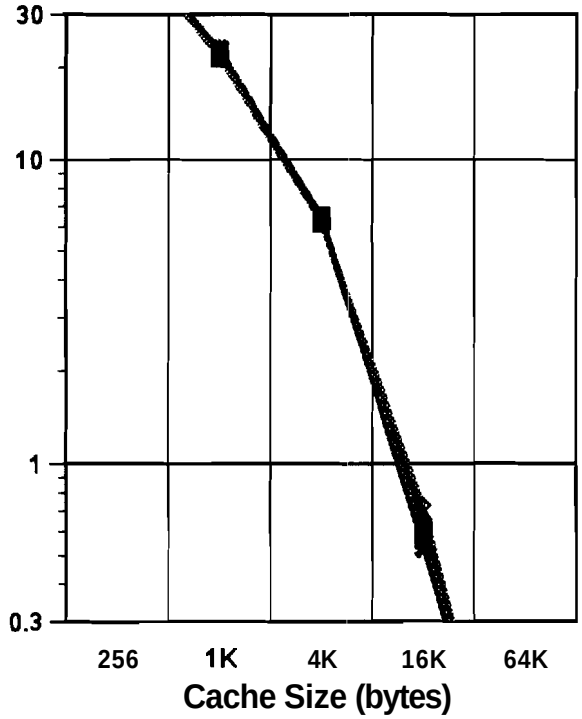

Figure 16d: Data Cache miss rate for fpppp, 4-way associative, 64 byte lines