Instruction Prefetching and Object-Oriented Programs

An-Cheng Huang
<pach@cs.cmu.edu>

Leejay Wu
<lw2j@cs.cmu.edu>

November 22, 1999

Abstract

Object-oriented and procedural programming styles yield programs with differing characteristics; therefore, the same optimizations may have different effects upon them. We examine the performance impact of both next-N-line prefetching and prefetching predicted paths on a simulated architecture across numerous cache configurations and a variety of benchmarks. We also examine the impact of copying instructions fetched from the prefetch buffer into the level one instruction cache. Our results show that the magnitude of effect is very dependent upon the nature of the benchmark and the instruction level one cache configuration, but that in general target-line prefetching yields minimal gains beyond that of purely next-N-line prefetching and copying into the instruction cache reduces performance.

1 Introduction

Object-oriented programs have remarkably different characteristics compared to systems adhering to standard procedural design [2, 6]. Consequently, architectural optimizations may not have identical impact on software written in these two styles.

One such optimization is instruction prefetching. As object-oriented programs tend to have more instruction cache misses, we would expect instruction prefetching to show a more significant effect on these compared to programs following traditional organizations [2].

In this project, we assessed the differences under many conditions. In addition to using multiple benchmarks written in the two styles, we considered both associative and direct-mapped caches. We also varied our prefetching scheme; with some tests, an instruction retrieved from our prefetch buffer was copied into the level 1 instruction cache. Independent of that aspect, some tests allowed target-line prefetching, which takes advantage of the branch predictor and prefetches at the predicted target in contrast to the usual prefetch next-N-line scheme.

2 Survey

Prefetching has been recognized as a useful optimization for over two decades [18]. Much work has been done with data prefetching, with linked data structures [15, 17, 10]; and memory forwarding to promote prefetching [12].

http://www.cs.cmu.edu/~pach/740/proj740.html

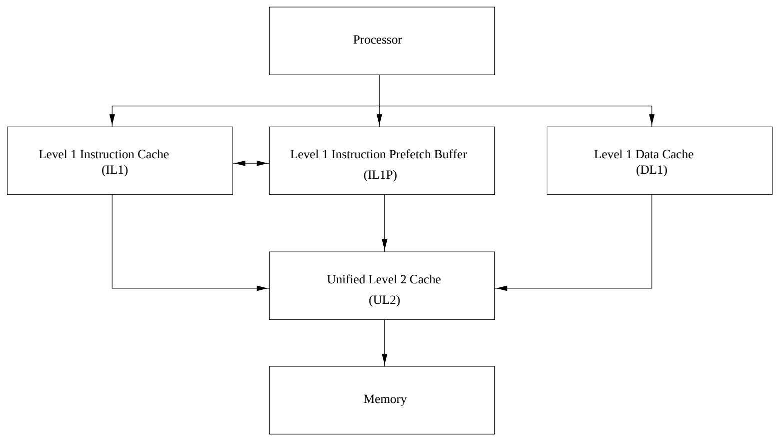

Figure 1: The memory hierarchy with Level 1 instruction prefetching.

While instruction cache miss rates tend to be lower than those for data caches [2], a single miss can have a serious impact on modern, highly pipelined, multiple-issue processors. Whereas a data miss might not stall other basic blocks from execution, failure to fetch instructions can starve all the function units.

Instruction prefetching has long been studied in order to minimize this price [18, 19]. Early study included next-N-line prefetching [18]; later work has included fetching both the fall-through and targets of branches [14];using better prediction methods [9]; and utilizing both software and hardware support [11].

Object-oriented program design has itself resulted in much consideration. Characteristics such as the far fewer average instructions per function call [2] have led to ways of supporting cross-procedural optimizations [13]. The higher cost of virtual function calls have led to research on VFC address prediction methods [16].

Prediction methods applicable to the general class of indirect jumps have also been researched [3, 7, 8].

3 System Design

We focus on instruction prefetching. Therefore, we modify a standard two-level cache hierarchy as per Figure 1.

Here, the prefetch buffer is shown to be a peer of the level one instruction cache. In this system, when the processor attempts to fetch an instruction, both the prefetch buffer and this instruction cache are checked simultaneously; if either contains the desired instruction the level two cache is not consulted.

Should it be in neither, then the instruction cache sends a request to the level two cache and the prefetch buffer initializes a queue of requests for succeeding lines. These requests are issued to the level two cache only when the level one cache is not simultaneously sending requests; they are cancelled immediately upon another such instruction fetch miss. It is assumed that the caches are pipelined.

The prefetch buffer itself is implemented as a fully associative cache with LRU replacement policy, and is not flushed on a instruction cache miss. It will not prefetch across a page boundary, in order to avoid generating spurious page faults.

Many other aspects of our design were stable. The instruction cache itself always had a maximum capacity of 512 32B lines; the data cache in all tests was direct-mapped and had a constant maximum of 512 32B lines; and the level two unified cache was always 8-way associative with a capacity of 4096 64B lines.

We varied other aspects. In early testing, the prefetch buffer contained 16 32B lines; later tests, 32 32B lines. The instruction cache was direct-mapped in half the tests; the rest of the time, it was 8-way associative.

In half the tests, an instruction retrieved from the prefetch buffer was copied back to the instruction cache, in order to preserve it as the prefetch buffer might be expected to be less stable.

Our later tests examine whether one can improve on next-N-line prefetching by taking advantage of branch target prediction in the case of unconditional jumps and calls found in lines that it prefetches. With target-line prefetching, we allowed cancelling subsequent requests in the prefetch queue and replacing them with a sequence starting at the predicted target, with the same restriction that it will not prefetch across a page boundary.

4 Methodology

We simulated the system using SimpleScalar 3.0 [1]. The majority of our modifications resided in a C file based directly on the included out-of-order simulator, which provides an out-of-order superscalar pipelined design. This required that all the benchmarks be built specifically for SimpleScalar.

This required building modified `gcc`, `binutils`, and – for the C++ programs – `libg++` packages. In addition, some benchmarks required extensive modification.

For benchmarks, we used parts of the SPEC95 [4] and OOCSB [5] suites; the former consists of procedural C and Fortran programs, while the latter includes object-oriented C++ benchmarks.

We omitted certain benchmarks due to compiling and correctness issues; all of these correct issues were manifested by the normal out-of-order simulator, and were unrelated to our prefetching additions. We acknowledge that it is a work in progress. For the SPEC95 suite, we used the "test" data instead of "reference" due to computational cost; the out-of-order simulator and our derivative simulate few instructions per second compared to that of a machine running in native mode.

Our testing was performed on two major types of platforms. We used Intel Pentium II-based workstations running Red Hat Linux, for running the precompiled little-endian SPEC95 benchmarks provided as part of the SimpleScalar suite; programs we compiled ourselves were tested on Sun Ultra-1 machines running Solaris. The reason for the split is that while the SimpleScalar compiler appeared to generate binaries even on the Linux/x86 platform, the simulator itself failed to load them. This problem did not exist with the pre-compiled SPEC95 benchmarks.

In our first set of tests, there were two possible non-prefetching configurations – "direct" with a direct-mapped level one instruction cache, and "assoc8" with an 8-way associative level one instruction cache. Prefetching configurations used 16-line prefetch buffers, with simple next-N-line prefetching, but varied whether or not a line fetched from the prefetch buffer was copied into the level one instruction cache. See Table 4 for the benchmarks used in this set.

Benchmark	Suite	Benchmark	Suite
`deltablue`	OOCSB	`hydro2d`	SPEC95
`eqn`	OOCSB	`ijpeg`	SPEC95
`ixx`	OOCSB	`li`	SPEC95
`lcom`	OOCSB	`m88ksim`	SPEC95
`richards`	OOCSB	`mgrid`	SPEC95
`applu`	SPEC95	`perl`	SPEC95
`apsi`	SPEC95	`su2cor`	SPEC95
`compress`	SPEC95	`swim`	SPEC95
`fpppp`	SPEC95	`tomcatv`	SPEC95
`gcc`	SPEC95	`turb3d`	SPEC95
`go`	SPEC95	`wave5`	SPEC95

Table 1: The benchmarks for the tests with 16-line prefetch buffers

Benchmark	Suite	Benchmark	Suite
`deltablue`	OOCSB	`fpppp`	SPEC95
`eqn`	OOCSB	`ijpeg`	SPEC95
`ixx`	OOCSB	`m88ksim`	SPEC95
`lcom`	OOCSB	`perl`	SPEC95
`richards`	OOCSB	`su2cor`	SPEC95
`compress`	SPEC95		

Table 2: The subset of benchmarks chosen for the tests with 32-line prefetch buffers

In our second set of tests, we expanded the prefetch buffer capacity to hold a maximum of 32 lines – with 64-bit SimpleScalar instructions, this allows prefetching up to 128 instructions ahead. In addition, we ran tests where the prefetching unit scans the lines it prefetches for control-flow statements. Finding a single control-flow statement stops the scan; if the instruction happens to be an unconditional branch or jump, we consult the branch predictor without changing its state. If the target is within the same page, we continue our prefetching at the target's line.

The benchmarks we used in this set of tests are listed in Table 4.

Due to computational time, not all benchmarks were executed with all configurations. For instance, we found that the `applu` benchmark in SPEC95 commonly took approximately 35 hours to run, on the test data; thus, we did not run this with the larger prefetch buffer and all its attendant configurations. The total real time elapsed during all SimpleScalar tests was approximately 52 processor-days.

5 Results

We measured five particular statistics. These were the number of cycles in a run, the level one instruction cache miss count, the mis-speculation rate, the branch address prediction rate, and the average number of instructions per branch.

For the first three, we note the impact of the various prefetching configurations. For all graphs, the baseline coincident with the X axis is that of an unmodified out-of-order SimpleScalar binary; each point plotted shows the degree of improvement of one style of prefetching versus that baseline, with higher always being better.

The latter two statistics are largely unaffected and we present them because of their impact on overall benchmark performance as a limiting factor on what prefetching can accomplish, and because the branch address prediction accuracy matters for the tests involving target-line prefetching.

5.1 Prefetching versus the baseline

First, we note that as per Figure 7, and in line with previous studies [2], the average number of instructions per branch is quite low among the OOCSB suite benchmarks we used compared to those of the SPEC95 suite. The natural hypothesis is that this relatively non-sequential control flow – compared to that of traditional procedural programs – will reduce the positive effects of next-N-line prefetching. On the other hand, the SPEC95 benchmarks `applu`, `fpppp`, and `mgrid` all have relatively high instructions per branch, and might be expected to benefit disproportionately.

Figure 8 shows branch prediction rates. This is partially motivated by the fact that next-N-line prediction by itself would not be expected to alleviate branch prediction penalties.

5.1.1 Number of cycles

Figures 3 and 6 show the improvement in the number of cycles for prefetching on associative and direct-mapped instruction caches, respectively. In both cases, the SPEC95 benchmarks `fpppp` and `tomcatv` get the most benefit; interestingly enough, `applu` and `mgrid` do not improve at all in either case. `applu` may have been hampered by a 28% miss rate in the level two unified cache, whereas `mgrid` has a 20% miss rate in the same cache; furthermore, more than one in every three instruction issues in `mgrid` was a load.

Most of the benchmarks, including those in the OOCSB suite, gained more from prefetching with a direct-mapped level one instruction cache than on an associative cache. An 8-way associative cache by itself significantly reduces the miss rate, reducing the possible benefits of prefetching.

We also note that in almost all cases, copying from the prefetch buffer to the instruction cache reduces performance gains, and that this effect is generally larger with an associative instruction cache.

5.1.2 Instruction cache misses

Figures 2 and 5 show the improvement in the number of instruction cache misses. With a direct-mapped instruction cache, prefetching reduced the number of misses dramatically, with many benchmarks seeing 40% to 90% reductions. The OOCSB benchmarks, except for `richards`, fare approximately as well as the SPEC95 benchmarks in this case.

When tested with an associative instruction cache, the reductions range widely – again, up to 90% – and the OOCSB benchmarks still gain approximately as much as their SPEC95 peers. We do note that the `m88ksim` benchmark, without prefetching, already attains an instruction cache miss rate of approximately 0.027%, and thus the impact of prefetching on miss rate is minimized.

5.1.3 Mis-speculation rate

Figures 4 and 9 show the effects of prefetching on the mis-speculation rate. We note that the impact is generally negative; these mis-speculation penalties are as high as 9% and 10% for associative and direct-mapped instruction caches, respectively. Interestingly enough, the benchmarks with the highest penalties are among those with the highest improvements in the number of cycles per run. With a few of the benchmarks, this penalty is greater when the prefetched instructions are not copied into the instruction cache.

The OOCSB benchmarks behave approximately the same as the non-exceptional SPEC95 benchmarks on this measure.

5.2 Prefetching with larger prefetch buffers and branch prediction

As previously mentioned, later tests used larger prefetch buffers and added an additional variable – whether the prefetch unit uses next-N-line or target-line prefetching.

5.2.1 Number of cycles

Figures 11 and 14 show the effects of prefetching with the larger buffer on the number of cycles required to run a benchmark, on 8-way associative and direct-mapped caches respectively. We note that with the associative instruction cache, that the main difference is that `fpppp` loses even more of the prefetching benefit when copying into the instruction cache; the rest is relatively unchanged.

With the direct-mapped cache, prefetching effects for certain benchmarks are significantly enhanced, relative to those with the smaller prefetch buffer. Using the branch prediction had relatively little effect compared to sequential prefetching.

5.2.2 Instruction cache misses

Figures 10 and 13 show the effects on the number of level one instruction cache misses. With the associative cache, the differences caused by the larger prefetch buffer, are mostly negligible; using the branch predictor also did not significantly change performance gains. The major exception is that the cache performance for `fpppp` dropped dramatically in terms of misses, when fetched instructions where copied from the prefetch buffer to the instruction cache.

With the direct-mapped cache, we note that overall, again there is a slight, positive difference with the results using the smaller prefetch buffer. However, we do note that for certain benchmarks – `deltablue`, `su2cor` there is a reduction in cache misses using branch prediction versus purely sequential prefetching.

5.2.3 Mis-speculation rate

Figures 12 and 15 show the effects on the mis-speculation rate. In the 8-way associative case, the results are almost completely identical to that with the smaller prefetch buffer, with the exception that `fpppp` actually improves its mis-speculation rate if lines are copied to the instruction cache.

With the direct-mapped instruction cache, there are no drastic changes; in general, the mis-speculation rates are slightly worse, and again target-line prefetching has minimal impact compared to next-N-line prefetching.

6 Analysis

First, we note that there appears to be a discontinuity between the improvements in instruction cache miss rate, and the reductions in the number of cycles elapsed during a run. Without the former, there is minimal gain in the latter – but it is by no means sufficient. Reasons for this include the fact that we took no measures to reduce latency due to other aspects of the benchmarks, such as data misses.

Our results also show that the configurations yielding the highest improvements in performance as measured by cycles elapsed also were among those that had the highest increases in mis-speculation rate. Prefetching more instructions allowed the simulated processor to execute more instructions speculatively.

Our results suggest that with direct-mapped caches, certain benchmarks – including the OOCSB benchmarks, with the exception of the small synthetic benchmark `richards` – benefit significantly due to prefetching. This is reasonable, as instruction cache miss rates tend to be quite high for object-oriented programs with direct-mapped caches; this miss rate tends to be lower, and thus afford less improvement, on systems with associative caches [2].

We found that increasing the prefetch buffer size had very slight beneficial effects in general. It seems reasonable to propose that diminishing returns occur at least partly because the prefetching buffer can only prefetch so quickly, so it does not prefetch that much further ahead. In addition, programs might not be so completely sequential that looking ahead more than 64 instructions proves especially useful.

Copying fetched instructions from the prefetch buffer to the instruction cache reduced prefetching gains almost universally in every statistic we measured. This contradicts our initial hopes that this would improve performance by preserving fetched instructions given that the prefetch buffer would theoretically have high replacement rates; however, this was not the case. One possible explanation is that by not duplicating entries, we effectively have an additional fully associative instruction cache, and that copying instructions may evict useful lines from the instruction cache. This hypothesis might be testable by using a victim cache, which would help offset any penalty from these evictions.

We also found that target-line prefetching usually performed only on par with next-N-line prefetching, despite our having limited it to unconditional branches and jumps within the same page. Contrary to our expectations, this held true in general even for the object-oriented programs. One possible reason for this is that we used the default bimodal branch predictor, whose accuracy at predicting addresses leaves something to be desired; see Figure 8.

7 Conclusion

Based on our results, we conclude the following. First, next-N-line prefetching can provide significant performance improvements, especially on a direct-mapped cache. Second, the benefits observed for target-line prefetching would not seem to be worth the probably significant hardware expense and complexity. Third, it is not beneficial in general to copy an instruction into the level one instruction cache when it is fetched from the prefetch buffer. Fourth, prefetch buffer of size 16 lines had approximately as much impact as that of one twice its size. Lastly, object-oriented C++ programs can indeed benefit more than the average procedural C program on platforms with direct-mapped instruction caches, and this benefit comes from the reduced number of instruction cache misses.

Our results suggest various possibilities for further work, including adding victim caches when copying lines to the level one instruction cache; using a better branch predictor in conjunction with target-line prefetching; and using both instruction and data prefetching simultaneously.

8 Acknowledgements

We would like to thank CMCL for providing copious amounts of disk space and processor cycles, and Chi-Keung Luk for feedback during the formulation of this project.

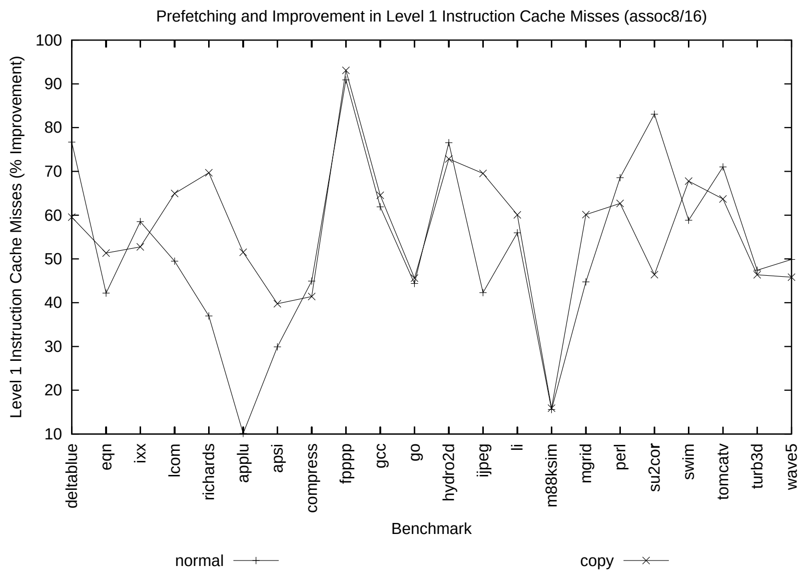

Figure 2: Improvement in level 1 instruction cache misses with a 8-way associative level one instruction cache and a 16-line prefetch buffer

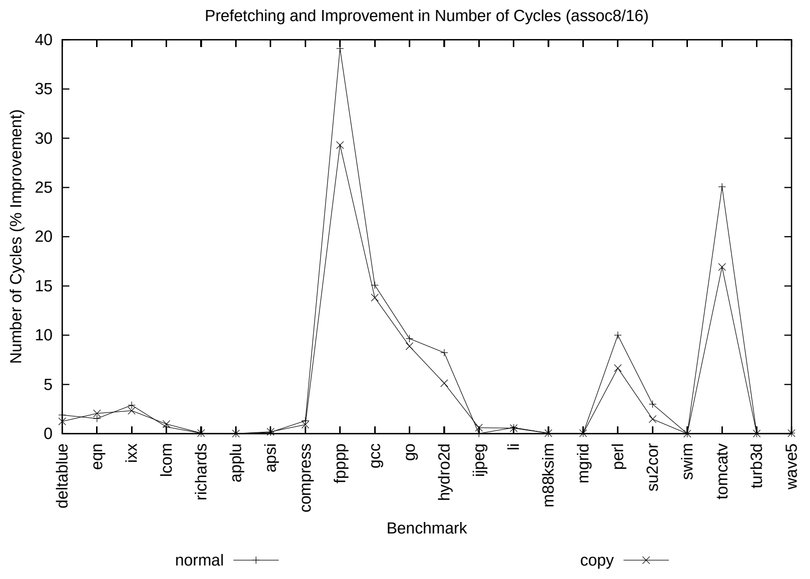

Figure 3: Improvement in number of cycles with a 8-way associative level one instruction cache and a 16-line prefetch buffer

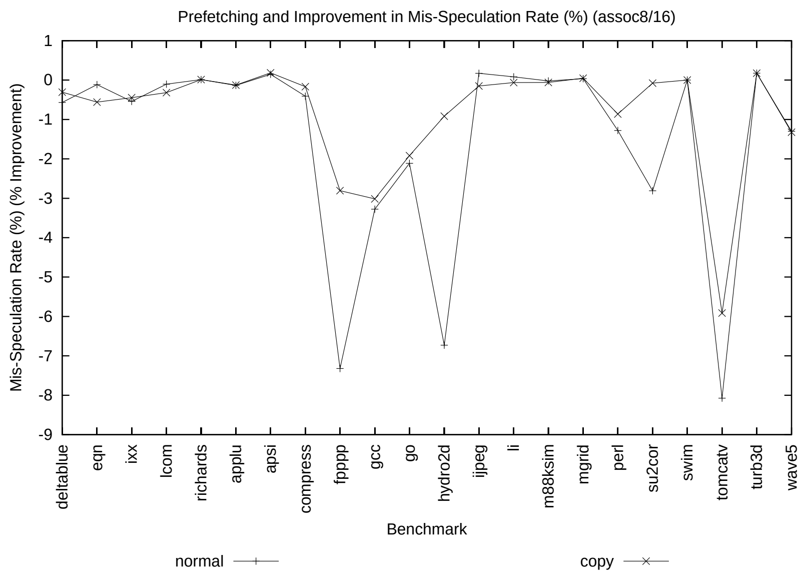

Figure 4: Improvement in mis-speculation rate (%) with a 8-way associative level one instruction cache and a 16-line prefetch buffer

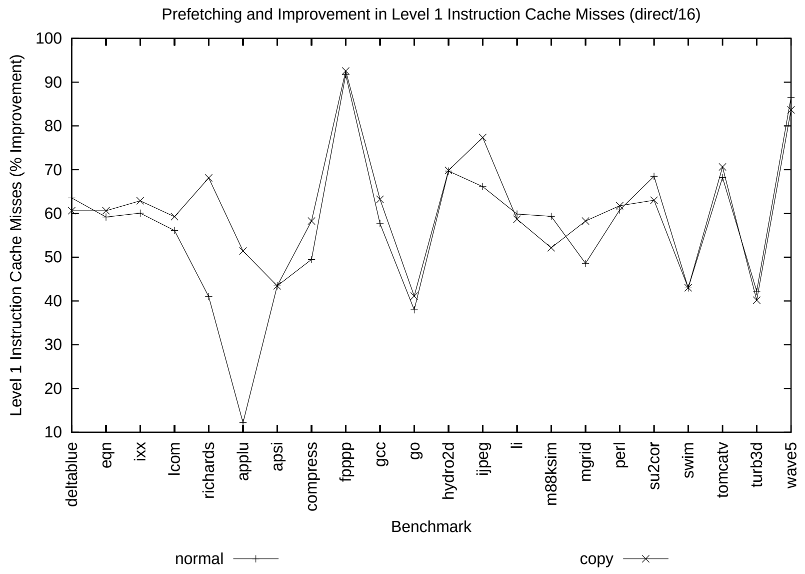

Figure 5: Improvement in level 1 instruction cache misses with a direct-mapped level one instruction cache and a 16-line prefetch buffer

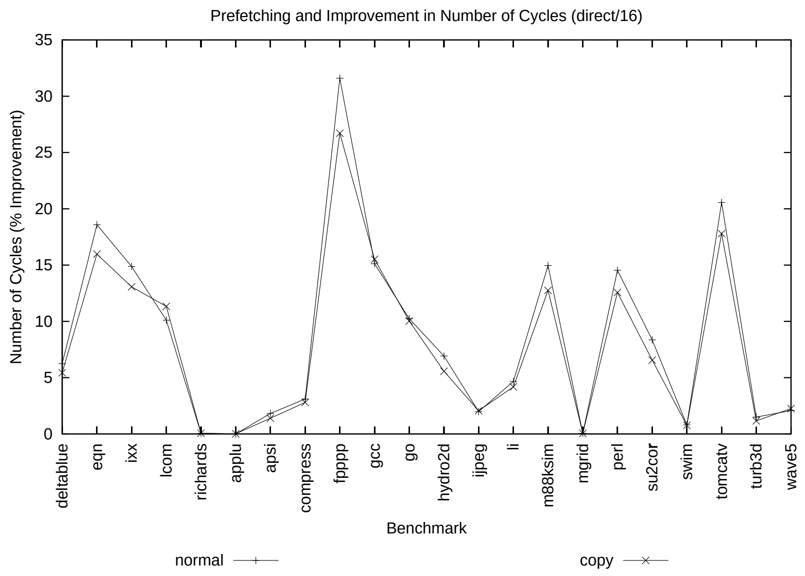

Figure 6: Improvement in number of cycles with a direct-mapped level one instruction cache and a 16-line prefetch buffer

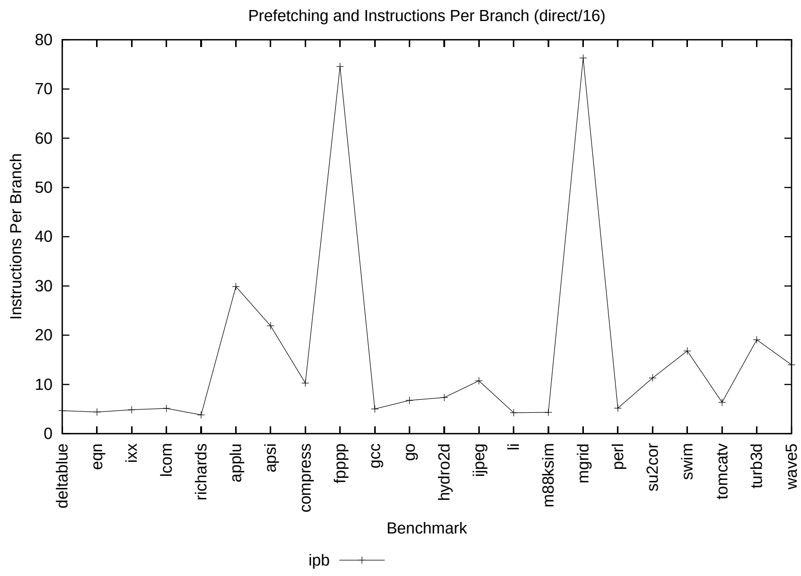

Figure 7: Instructions per branch with a direct-mapped level one instruction cache and a 16-line prefetch buffer; we omit the identical graphs for the associative configuration and those with 32-line prefetch buffers.

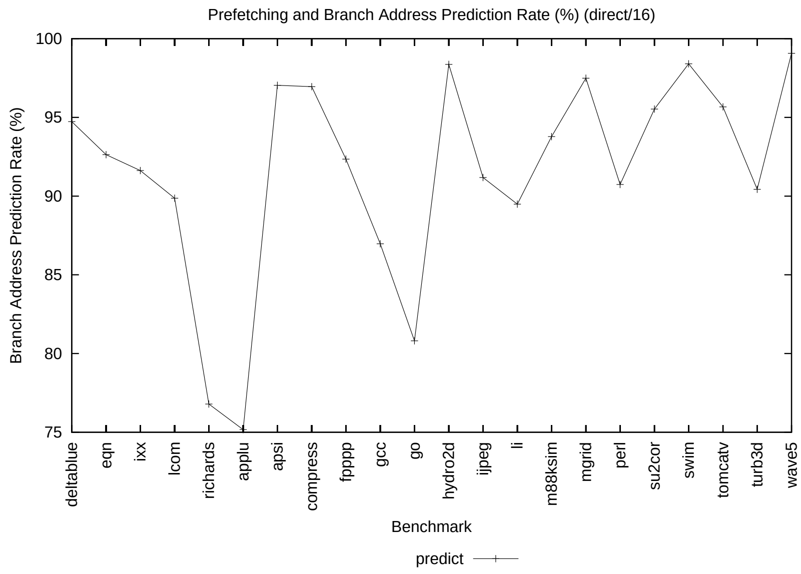

Figure 8: Branch address prediction rate (%) with a direct-mapped level one instruction cache and a 16-line prefetch buffer

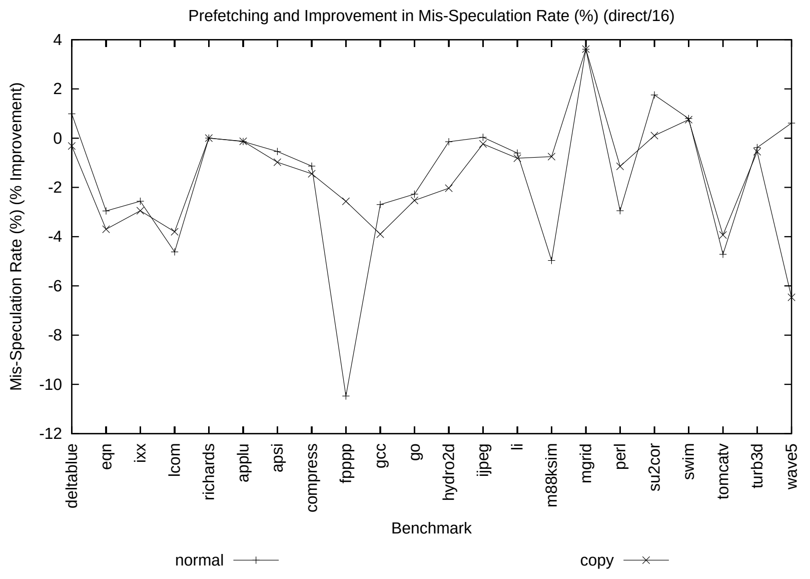

Figure 9: Improvement in mis-speculation rate (%) with a direct-mapped level one instruction cache and a 16-line prefetch buffer

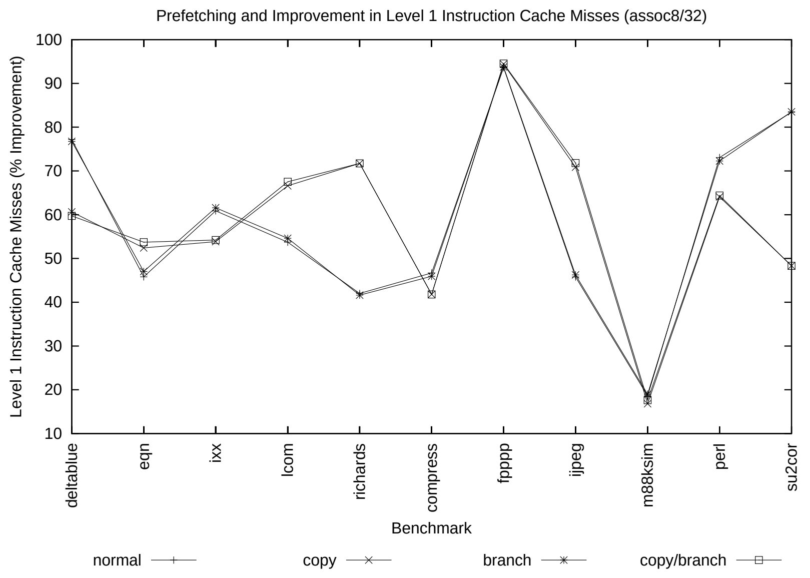

Figure 10: Improvement in level 1 instruction cache misses with a 8-way associative level one instruction cache and a 32-line prefetch buffer

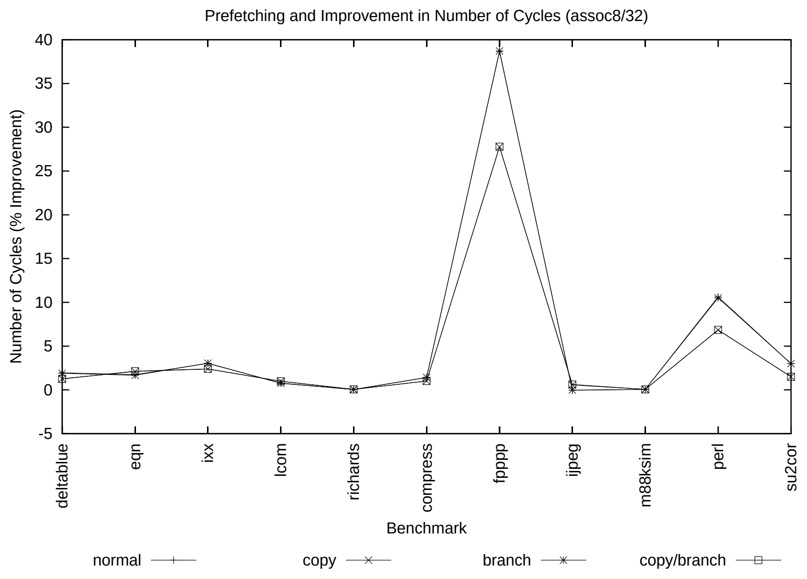

Figure 11: Improvement in number of cycles with a 8-way associative level one instruction cache and a 32-line prefetch buffer

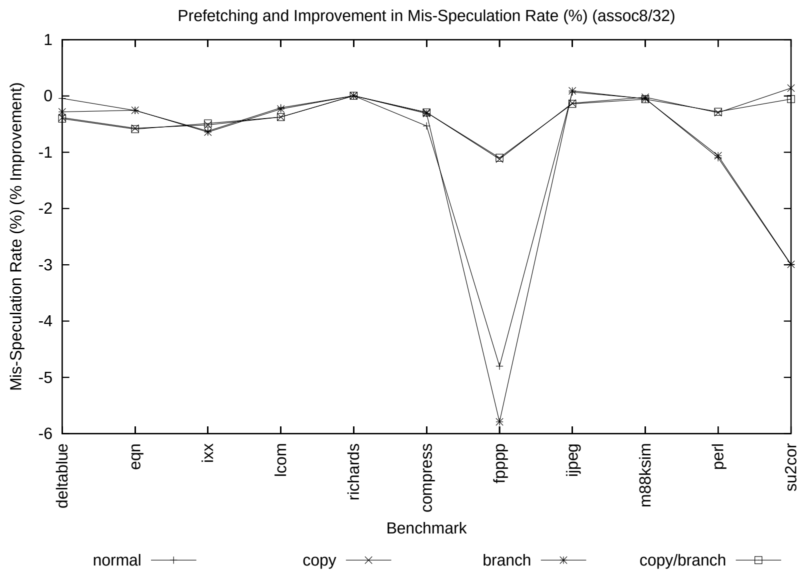

Figure 12: Improvement in mis-speculation rate (%) with a 8-way associative level one instruction cache and a 32-line prefetch buffer

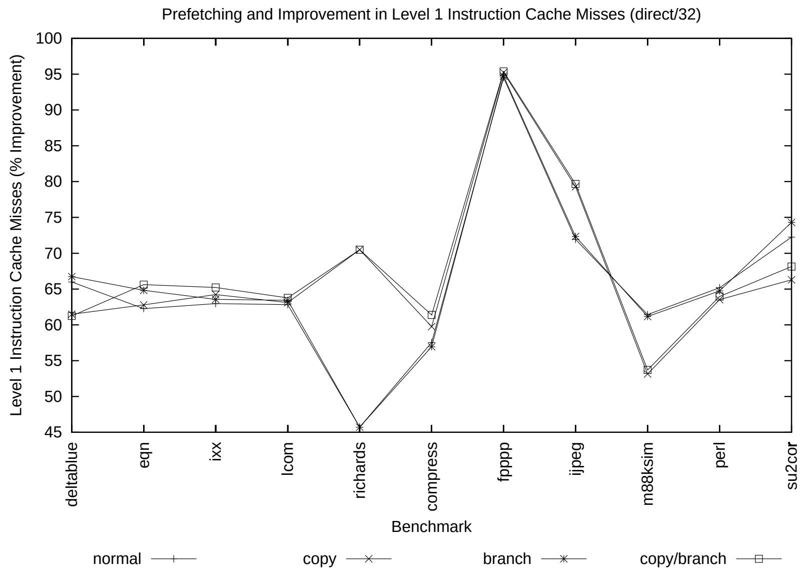

Figure 13: Improvement in level 1 instruction cache misses with a direct-mapped level one instruction cache and a 32-line prefetch buffer

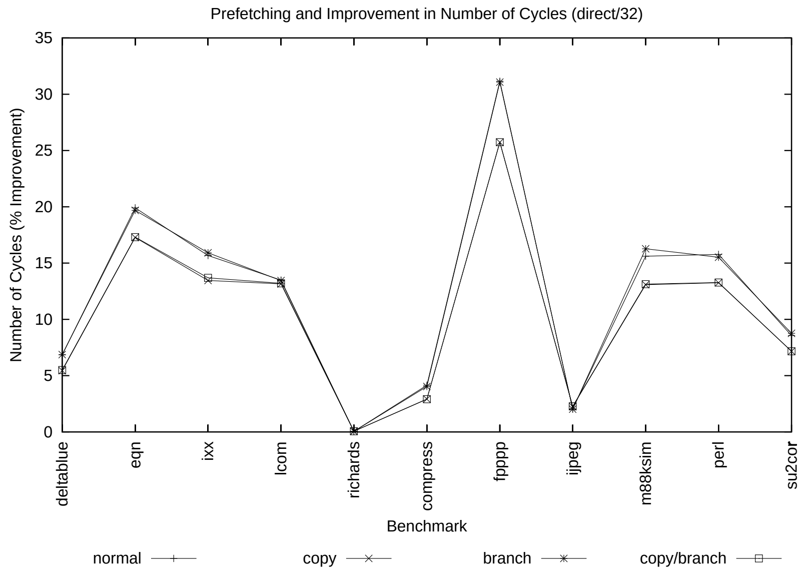

Figure 14: Improvement in number of cycles with a direct-mapped level one instruction cache and a 32-line prefetch buffer

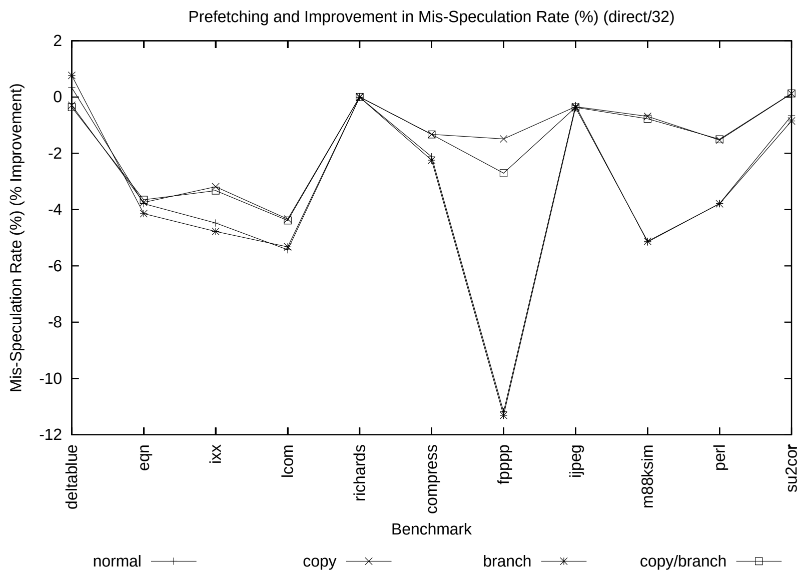

Figure 15: Improvement in mis-speculation rate (%) with a direct-mapped level one instruction cache and a 32-line prefetch buffer

References

[1] Doug Burger and Todd M. Austin. The SimpleScalar tool set, version 2.0. Technical report, University of Wisconsin-Madison, June 1997.

[2] Brad Calder, Dirk Grunwald, and Benjamin Zorn. Quantifying behavioral differences between C and C++ programs. *Journal of Programming Languages*, 2(4), 1994.

[3] Po-Yung Chang, Eric Hao, and Yale N. Patt. Target prediction for indirect jumps. *Proceedings of the 24th International Symposium on Computer Architecture*, 1997.

[4] Standard Performance Evaluation Corporation. SPEC CPU95 Benchmarks `http://www.spec.org/osg/cpu95`

[5] Karel Driesen. OOCSB Benchmarks. `http://www.cs.ucsb.edu/oocsb/benchmarks`

[6] Karel Driesen and Urs Hölzle. The direct cost of virtual function calls in C++. *OOPSLA '96 Proceedings*, October 1996.

[7] Karel Driesen and Urs Hölzle. Accurate indirect branch prediction. *Proceedings of the 25th International Symposium on Computer Architecture*, July 1998.

[8] Karel Driesen and Urs Hölzle. The cascaded predictor: Economical and adaptive branch target prediction. *Proceedings of Micro-31*, December 1998.

[9] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. *Proceedings of the 24th International Symposium on Computer architecture*, June 1997.

[10] Chi-Keung Luk and Todd C. Mowry. Compiler-based prefetching for recursive data structures. *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[11] Chi-Keung Luk and Todd C. Mowry. Cooperative prefetching: Compiler and hardware support for effective instruction prefetching in modern processors. *Proceedings of Micro-31*, December 1998.

[12] Chi-Keung Luk and Todd C. Mowry. Memory forwarding: Enabling aggressive layout optimizations by guaranteeing the safety of data relocation. *Proceedings of the 26th Annual International Symposium on Computer Architecture*, May 1999.

[13] Milo M. Martin, Amir Roth, and Charles N. Fischer. Exploiting dead value information. *Proceedings of Micro-30*, December 1997.

[14] Jim Pierce and Trevor Mudge. Wrong-path instruction prefetching. *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, 1996.

[15] Amir Roth, Andreas Moshovos, and Gurindar S. Sohi. Dependence based prefetching for linked data structures. *Proceedings of ASPLOS-8*, October 1998.

[16] Amir Roth, Andreas Moshovos, and Gurindar S. Sohi. Improving virtual function call target prediction via dependence-based pre-computation. *Proceedings of ICS-99*, June 1999.

[17] Amir Roth and Gurindar S. Sohi. Effective jump-pointer prefetching for linked data structures. *Proceedings of the 26th International Symposium on Computer Architecture*, May 1999.

[18] Alan J. Smith. Sequential program prefetching in memory hierarchies. *IEEE Computer*, 11(2), December 1978.

[19] J. E. Smith and W.-C. Hsu. Prefetching in supercomputer instruction caches. *Supercomputing '92*, July 1992.

Contents

List of Figures

List of Tables