Improving Cache Performance by Eliminating Transfers of Dead Data

Edward W. Felten
Eric J. Koldinger
Raj Vaswani
John Zahorjan

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195 U.S.A.

March 13, 1992

Improving Cache Performance by Eliminating Transfers of Dead Data

Edward W. Felten Eric J. Koldinger Raj Vaswani
John Zahorjan
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195 U.S.A.

Abstract

Traditional main memory caches provide semantics that deal with memory locations rather than the data they contain. By considering the cache as a repository of data, rather than locations, it is possible to eliminate transfers associated with *dead data*, data that is guaranteed not to be referenced again. The transfers eliminated correspond to fetches of cache lines that will be completely overwritten without being read, and to writebacks of data that will never again be read.

We describe the semantics and implementation of a set of *cache control instructions* that, when added to a conventional instruction set, allow the elimination of dead data transfers. We demonstrate how these instructions could be employed by a compiler, and give a quantitative assessment of their benefits. This assessment is based on trace-driven cache simulations of benchmarks from the Livermore Loops, SPEC, and Perfect Club suites. As part of this study, we determine the maximum benefit attainable through the use of these instructions; this is accomplished by simulating an "optimal" policy which makes infallible decisions regarding memory accesses. Furthermore, in addition to evaluating application programs, we also monitor the operating system kernel to estimate the impact of cache control instructions on the performance of kernel operations.

Our simulations show that on today's architectures the average performance improvement obtained using these techniques is quite modest, although some individual programs exhibit more dramatic gains. Using a combination of a simple analytic model and measurements of our benchmark programs, we show that the performance gains are expected to grow in the future, but cannot grow beyond a certain limiting factor, regardless of assumptions about CPU and memory speeds. Because the cache control instructions are inexpensive to implement, never result in a degradation in performance for any application, and provide noticeable improvements for some, we conclude that they provide a useful approach for helping to deal with the growing disparity in processor and main memory speeds.

1 Background and Motivation

As processors continue to get faster relative to memory, the design and performance of the memory hierarchy is becoming an increasingly crucial factor in system architecture. Caches are successful

This material is based upon work supported by the National Science Foundation (Grants DCR-8352098, CCR-8619663, and CCR-8703049), the Washington Technology Center, Digital Equipment Corporation (the External Research Program and the Systems Research Center), and Apple Computer Company. Felten was supported in part by an AT&T Ph.D. Scholarship and a Mercury Seven Fellowship.

in reducing the effects of memory latency, but as latency continues to grow, naive caching is no longer sufficient to meet the performance demands of applications. A great deal of research effort has gone into the study of alternative caching techniques [Jouppi 90, McFarling 92, Gupta et al. 91].

Traditional main memory caches provide fetch and writeback semantics that are based on memory locations, rather than on the data that those locations contain. However, by distinguishing memory locations from their data, some transfers between main memory and the cache can be eliminated. These transfers correspond to fetches or writebacks of *dead data*, data that will not be read again.

In this paper, we evaluate a set of cache control instructions that can be used to eliminate accesses involving dead data. We say a cache line is dead if all data in it are dead. One way to improve performance is to eliminate cache misses and writebacks that access dead lines.

There are two situations in which dead data is transferred between a write-back cache and main memory. First, a write to a dead line may cause a cache miss; the line is then needlessly loaded into the cache. (Note that a read miss is never to a dead line — the fact that the access is a read implies that the line is live.) Second, when a dead line is replaced in the cache, it may unnecessarily be written back to memory. Cache control instructions can be used to eliminate memory accesses in both of these situations. As discussed in section 3, opportunities to use these instructions arise in the following situations: loops in scientific programs, procedure prologs and epilogs, operating system and library code, and dynamic storage allocation.

We measure the impact of cache control instructions on the performance of a number of benchmarks (Livermore Loops and some of the SPEC and Perfect Club benchmarks), using a variety of techniques to insert our new instructions into the programs. In addition, we determine an upper bound on the possible performance benefit of such instructions by comparing against a design with "optimal" performance. This last scheme eliminates all accesses to dead data, using knowledge about the programs' future reference behavior; this is only a reference point since this knowledge is not available in practice. We also evaluate the effect of cache control instructions on the performance of kernel primitives. Finally, we use our data to extrapolate these effects to future architectures.

Wittenbrink *et al.* have proposed a similar idea [Wittenbrink et al. 92], but their implementation, based on special page-table bits, is considerably more expensive than the one described below, and their experiments used a smaller and more specialized benchmark suite than ours.

2 Instructions to Eliminate Dead Data Accesses

We envision two instructions to control the cache treatment of dead data. To eliminate write-misses to dead data, we imagine a `store&zero` instruction. This instruction behaves like an ordinary `store` instruction, except that it first zeroes the location's cache line. This allows the implementation to generate the cache line directly rather than fetching it from memory. In principle, the cache line could be filled with any value — we assume the line is filled with zeroes because this allows the use of this instruction to zero-fill memory quickly.

To eliminate writebacks of dead lines, we envision a `clean` instruction. `clean` is a hint to the implementation that a cache line will overwritten before it is read again. For concreteness, we assume the following implementation: if the location to be cleaned is in the cache, the cache line

is marked as unmodified; if the location is not in the cache, the instruction has no effect. This implementation ensures that a cleaned cache line will not be written back to memory.

For simplicity, if the system uses a multi-level cache, we envision tying these instructions only to the first-level cache. In principle, any level in the cache hierarchy could take advantage of these instructions.

We now discuss the consequences of the instructions just described on both uniprocessor and multiprocessor architectures. Cache control instructions exist in some current architectures, such as the Hewlett-Packard PA-RISC [Hew 90]. Accordingly, we first contrast the PA-RISC's cache control instructions with the abstract instructions defined above. We then consider multiprocessor issues.

2.1 Comparison with PA-RISC

The PA-RISC's "cache hint" is similar to our `store&zero` instruction. The main difference is that `store&zero` is assumed to be binding, while the PA-RISC directive is only a hint to the implementation. The PA-RISC cache hint also does an alignment check: a cache line is not zeroed unless the instruction accesses the first word in the line; this allows the line size to be changed (within architecturally defined limits) without affecting correctness of programs. The PA-RISC compiler need know only the maximum line size allowed by the architecture, whereas a compiler for a machine with `store&zero` must know the actual line size.

The PA-RISC `purge` instruction, which invalidates a cache line, is similar to our `clean`. On uniprocessors, the two instructions have the same effect — although `clean` leaves a valid copy of the line in the cache, this is useless because the program has promised (by executing `clean`) not to access this data before it is overwritten. However, `clean` has advantages over `purge` in a multiprocessor environment, as discussed below.

As stated previously, the `clean` instruction only affects the number of writebacks; since write-back latency can be hidden by a write buffer, we expect `clean` to have little effect on uniprocessor performance. Detailed simulations of some of our benchmarks confirmed our assumption that write-buffer stalls are extremely rare. (Raw data concerning the number of writebacks saved by `clean` instructions are given in Appendix B.)

Current implementations of the PA-RISC (in the HP9000/s700 and HP9000/s8x7 systems) do not use the cache hint [Wilkes 92]; we do not know whether the current PA-RISC compilers and assemblers generate the hint, or whether they use the `purge` instruction. In any case, we know of no study that evaluates the effectiveness of cache control instructions in the PA-RISC.

2.2 Cache-Coherent Multiprocessors

On a cache-coherent multiprocessor, the `store&zero` instruction must perform a global invalidation (or update), since `store&zero` must ensure that all processors see the line as containing zeroes.

However, since `clean` is only a hint, it can be implemented without a global invalidation — the implementation can simply mark the local copy of the line (if any) as unmodified [1]. The existence of stale copies in other caches or main memory is not a problem, since the application program is promising (as above) that it will not access the stale data. (In contrast, the semantics of the PA-RISC's `purge` require a global invalidation.)

[1] This assumes that this is allowed by the coherency protocol

Further, `clean` might offer a larger advantage on a multiprocessor than it does on a uniprocessor. Although, as previously discussed, write buffers can mask writeback latency, they cannot prevent bus contention caused by writebacks. Since `clean` marks a (dirty) cache line as unmodified, it prevents that line from being written back, thereby reducing bus contention. As appendices A and B show, writebacks and cache misses are about equally frequent. Thus, on a multiprocessor, the impact of `clean` might approach that of `store&zero`.

3 How Can the Instructions Be Used?

Having described the implementation and semantics of `store&zero` and `clean`, we now consider how they might be used. Cache control instructions can be useful in several situations. Among them are:

- *Loops in scientific programs:* Some scientific programs contain loops that write to the elements of an array in sequential order. A compiler that unrolls the loop by a factor equal to the number of array entries per cache line can generate a new loop that writes an entire cache line per iteration. It can then change the first `store` instruction touching the cache line into a `store&zero`. Similarly, consider an application that accesses some region of memory in two distinct, identifiable sections of the program. If the compiler can deduce that the memory region is dead between these accesses, it can insert `clean` instructions after the first section, avoiding writebacks.

- *Procedure prologs and epilogs:* At the beginning of the execution of a procedure, space is allocated on the stack for procedure arguments and local variables. The allocated area is dead, since it is either uninitialized or contains stale stack data; therefore `store&zero` can be used to initialize it. Similarly, `clean` can be used on procedure exit to ensure that the freed stack area is not written back to memory. These instructions can be inserted by the compiler; in order to make this tractable, the compiler should align procedure activation records on cache-line boundaries.

- *Operating system and runtime library code:* Certain procedures in the operating system and runtime library can be sped up by incorporating cache control instructions. These instructions could be inserted by systems programmers. Procedures that could be improved include `bcopy` (which copies memory), `bzero` (which zeroes memory), and the procedures that copy system call arguments between user and kernel space. One noteworthy area in which this technique could be applied is the virtual memory system: requests for zero-filled pages can be cheaply handled by using the augmented `bzero` to initialize the allocated page.

- *Dynamic storage allocation:* When a program dynamically allocates storage, it can assume that the data in the newly-allocated memory is dead. `store&zero` instructions can therefore be used to "pre-allocate" the region in the cache. Pre-allocating space in the cache may hinder performance in some cases, because currently useful cache lines may be replaced by generated lines that are not immediately useful. (Using cache control instructions in concert with storage allocation may be compared to prefetching: both schemes are inherently predictive, and performance can suffer if the prediction proves erroneous.) Thus, the storage allocation system must use `store&zero` with care. When space is deallocated, `clean` can safely be used

to ensure that the (now invalid) data in the region is never written back to memory. These instructions would be inserted into the storage allocation system by hand.

4 Experimental Procedure

We used trace-driven simulation to measure the effect of using cache control instructions on the performance of a set of benchmark programs. Our benchmarks included some of the Livermore Loops [McMahon 72], and members of the SPEC [SPE 89] and Perfect Club [Berry et al. 89] suites. The SPEC programs are intended to model a general-purpose workload, while Perfect Club programs are typical of scientific computations. Livermore Loops are small programs that model the inner loops of scientific programs.

Although we expect that cache control instructions would be inserted by a compiler, for the purpose of our experiments we simulated the compiler's activity. An assembly-language preprocessor inserted the instructions into procedure prologs and epilogs. The programs were linked to a library containing special hand-written `bzero` and `bcopy` procedures. Cache control instructions were inserted into loops by hand. Since unrolled loops are a prerequisite to inserting cache control instructions, we also unrolled some loops by hand [2].

We used the `pixie` tool [MIPS Computer Systems 86] to generate data-reference traces for the benchmark programs on a MIPS R3000 [Kane & Heinrich 92] based workstation. The traces were fed to a cache simulator that simulates a direct-mapped, virtually-addressed data cache with variable cache size and line size.

Given a trace, the simulator determines the performance under three scenarios:

1. *original*: without inserting cache control instructions,
2. *enhanced*: with the cache control instructions we inserted, and
3. *optimal*: with an (unrealizable) implementation which uses future information to eliminate *all* misses and writebacks on dead data.

The enhanced version of each program contains `store&zero` instructions inserted as described above. Unfortunately, we were unable to find many opportunities to insert the `clean` instruction. It may be equally difficult for a compiler to use `clean` effectively. In addition, as discussed in subsection 2.1, write buffers reduce the advantage of the `clean` instruction on uniprocessors. For these reasons, we chose to concentrate on the performance advantages of `store&zero`; however, `clean` is deserving of further study, especially on multiprocessor systems (see subsection 2.2).

Our final set of experiments evaluate the impact of `store&zero` on operating system performance. Since `pixie` does not trace the operating system, we instrumented the operating system kernel on our main departmental machine to estimate the benefit of cache control instructions in the operating system. We measured the number of `bzero` and `bcopy` calls and the average size of their arguments. For more details on the operating system measurements, see section 5.2.

4.1 Performance Measures

Our simulator measures the cache miss frequency of each benchmark (the number of misses per instruction executed). We can then calculate the performance loss due to cache misses by mul-

[2] To make our comparisons fair, all simulations were run on programs with loops unrolled.

tiplying the miss frequency by the miss penalty. The miss penalty P is the average number of instructions that could have been issued during the time the processor was stalled due to a cache miss. We count the number of instructions lost rather than the number of cycles lost, because this allows us to account for the effects of superscalar implementations, pipeline interlocks, instruction cache misses, and so on. On an "ideal" CPU that executes exactly one instruction per cycle, the miss penalty is just equal to the number of cycles required for a cache miss. On an ideal dual-issue machine (that executes exactly two instructions per cycle in the absence of cache misses), the miss penalty is twice the number of cycles lost due to a miss.

In general, if a cache miss costs M_{cyc} cycles, and the system executes an average of I instructions per cycle in the absence of data cache misses, then the baseline CPI C is $1/I$, and the miss penalty $P = M_{cyc}/C$. Note that the miss penalty may vary from benchmark to benchmark because I (and therefore the baseline CPI C) may vary.

Suppose we are comparing the performance advantage of an improved program (with miss frequency $f_{improved}$) with that of the original, unimproved program (which has miss frequency $f_{original}$). If M_{cyc}, I, and C are defined as above the relative performance improvement is

$$\Delta = \frac{Time_{original} - Time_{improved}}{Time_{improved}} = \frac{(C + M_{cyc} f_{original}) - (C + M_{cyc} f_{improved})}{C + M_{cyc} f_{improved}} \tag{1}$$

which reduces algebraically to

$$\Delta = \frac{P(f_{original} - f_{improved})}{1 + P f_{improved}}. \tag{2}$$

Throughout this paper, performance results are given in terms of Δ, the relative performance improvement due to cache control instructions.

An interesting feature of equation 2 is that the performance improvement Δ does not grow arbitrarily as the miss penalty increases; as P goes to infinity, Δ asymptotically approaches the constant value

$$\Delta_\infty = \frac{f_{original} - f_{improved}}{f_{improved}}. \tag{3}$$

Regardless of assumptions about the speeds of CPUs and memory, the performance improvement due to a reduction in cache misses cannot exceed Δ_∞.

4.2 Simulating the Optimal Implementation

To get an upper bound on the benefit due to cache control instructions, our simulator predicts the performance of a hypothetical optimal implementation. The optimal implementation handles dead cache lines perfectly — it never fetches a dead line from memory, and never writes a dead line back to memory [3]. Of course we cannot hope to build such a system, since identifying dead lines requires looking into the future.

Our simulator, in addition to gathering cache statistics on the executing program, determines how many cache misses and writebacks would have been avoided by the optimal system. This determination works in exactly the same way for misses as for writebacks. When the simulated program has (say) a cache miss, the simulator inserts a record describing the miss into a hash table.

[3] Note that because we assume a direct-mapped cache, the optimal implementation cannot choose a replacement policy. However, our methodology extends to set-associative caches, provided that the replacement policy is treated as a given, and is not under the control of the optimal implementation.

This record of the miss stays in the table until we are able to deduce whether the cache line was dead or alive at the time of the miss.

To determine whether a line was dead, we keep a bitmap in the miss record, with one bit per word in the line. A bit in the bitmap is 0 if there has been no reference to that word since the cache miss described by the record. A bit in the bitmap is 1 if the next reference to that word (after the miss) was a write; this implies the word was dead at the time of the cache miss. When a record is created, its bitmap is set to all 0's, indicating that the status of all words in the line is unknown. When the simulated program writes a word in the line, the corresponding bit is set; if all bits become set, we deduce the line was dead, and remove the entry from the table. When the simulated program reads a word in the line, and the corresponding bit is 0, we deduce the line was live, and remove the entry from the table. When the simulated program terminates, all entries remaining in the table are deduced to be dead.

In practice, the unresolved miss table becomes very large, and the simulation time is unacceptable. To avoid this problem, we do not monitor all misses and writebacks, but only a statistical sample. The sample is chosen randomly and uniformly from the set of all misses and writebacks generated by the program; when a miss or writeback occurs, we choose to include it in the sample with some fixed probability. The result is not an exact count of the number of misses and writebacks that could be avoided, but a statistical estimate with confidence intervals. We chose the sample probability large enough to ensure small confidence intervals, but not so large that simulation time was unacceptable. All numbers in this paper are accurate to the last decimal place given, with 95% confidence. (Numbers for the original and enhanced scenarios are exact.)

5 Results

We now present the results of our simulations and measurements. Subsection 5.1 presents performance results for benchmark programs, and subsection 5.2 discusses our measurements of operating system behavior. Subsection 5.3 summarizes our results.

5.1 Performance of Benchmark Programs

We used trace-driven cache simulation to predict the performance of benchmark programs from the Livermore Loops, SPEC, and Perfect Club suites. We were unable to simulate some benchmarks because the programs either did not compile on our system or did not run correctly. We also chose not to simulate seven of the twenty-four Livermore Loops, because we saw no opportunity for improvement — these loops had either non-unit stride or unpredictable stride, and thus could not benefit from cache control instructions. The complete list of programs we did simulate (with the raw data) appears in appendix A.

The first set of simulations we ran were on the Livermore Loops. Each loop was generated as a separate program and run on a large dataset, typically 50,000 elements. Each loop was unrolled four times (four double precision floating point numbers fit in a 32 byte cache line, the size simulated) and the first `store` in the loop was replaced with a `store&zero` instruction. The results of this simulation, run on a cache of 16 kbytes (with a 32-byte line size), are shown in figure 1.

The results of this test were encouraging. Twelve of the benchmarks showed some improvement. Improvements due to cache control instructions approached 35% for kernel 12. As shown in figure 5, the mean improvement over all kernels was 8.62 % for the enhanced scenario, and 9.20 % for the

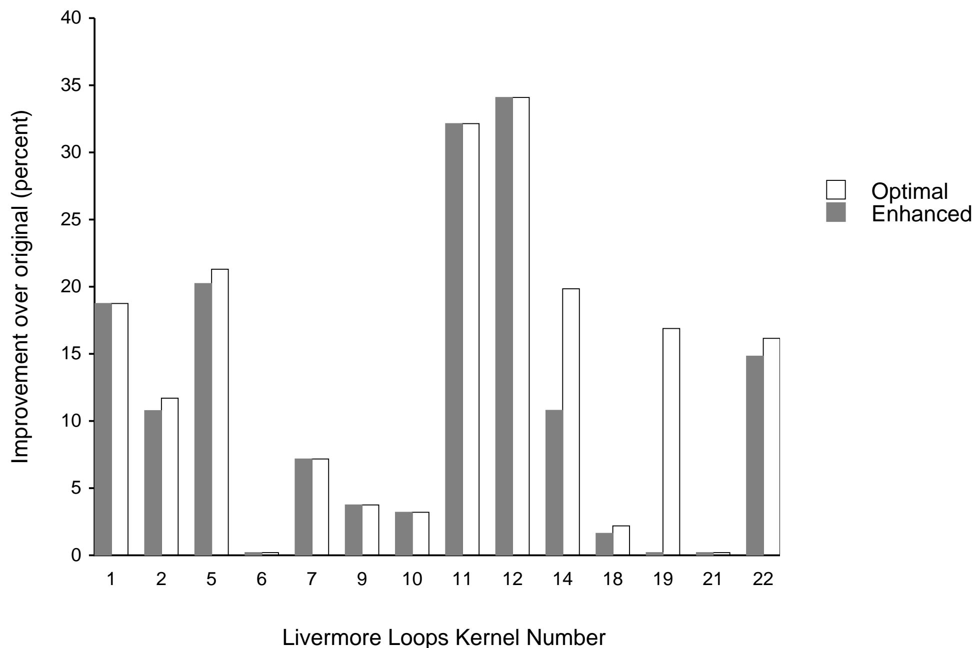

Figure 1: Improvement due to cache control instructions for the Livermore Loop benchmarks. For each program, we show the improvement of the enhanced and optimal programs, compared to the original program, when the cache miss penalty is 15 instruction-times.

optimal scenario. (These means include an assumption of no improvement for those loops which we did not measure. This is appropriate since, as stated above, we chose to ignore those loops precisely because we expected no improvement.)

Although the overall improvements are modest, several individual programs show excellent improvement. This improvement can be understood by examining the structure of these programs. For example, in the first Livermore kernel, a "Hydro Fragment", shown in figure 2, a single iteration of the loop can cause three cache misses: one to the array element `Y(k)`, one to array element `ZX(k+11)`, and one to array element `X(k)`. The reference to array element `ZX(k+10)` typically will not cause a cache miss because it was the element `ZX(k+11)` on the previous iteration of the loop, and so will already be in the cache. The scalar variables `Q`, `R` and `T` can all be allocated to registers by the compiler. The references to `Y(k)` and `ZX(k+11)` are read misses, so they cannot be eliminated. However, `X(k)` is dead data, so its cache miss can be eliminated; thus one-third of all cache misses for kernel 1 can be eliminated. Some of the other kernels have similar structures, so many of their cache misses can likewise be eliminated.

```
      DO 1 k = 1,n
1        X(k)= Q + Y(k)*(R*ZX(k+10) + T*ZX(k+11))
```

Figure 2: Livermore Loops kernel 1 – "Hydro Fragment"

The second experiment we conducted measured the impact of cache control instructions on the performance of complete benchmark programs. We used two standard benchmark suites, SPEC and Perfect Club. We modified our simulator to identify sections of code where avoidable misses were actually occurring. We examined these sections, and looked for opportunities to use `store&zero` (unrolling loops if necessary). As described above, we also used a preprocessor to insert cache control instructions into procedure prologs and epilogs.

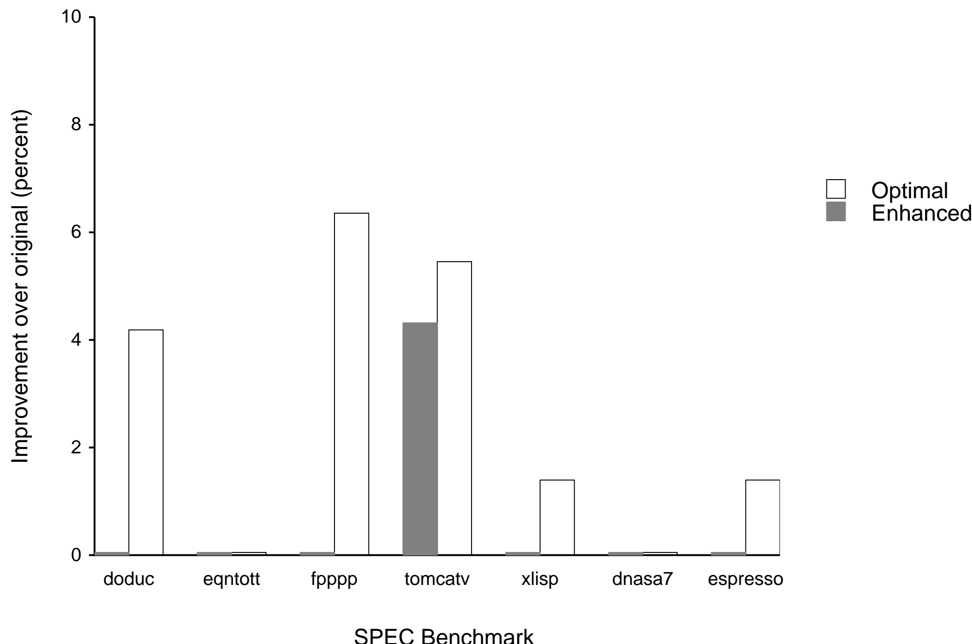

Figure 3: Improvement due to cache control instructions for the SPEC benchmarks. For each program, we show the improvement of the enhanced and optimal programs, compared to the original program, when the cache miss penalty is 15 instruction-times.

The results of these experiments are shown in figures 3 and 4. (Mean improvements over each suite are again shown in figure 5.) The performance gains due to cache control instructions were less dramatic for the SPEC and Perfect Club programs than for the Livermore Loops. This is not surprising, since the Livermore Loop programs are well-suited for cache optimizations — they contain a single loop, which writes an output array once, usually with unit stride. Realistic benchmarks have more complex behavior.

For example, most of the avoidable misses in the SPEC program `xlisp` occurred in a loop in one procedure, `xlsave`. Although this loop writes an array with unit stride, the loop terminates unpredictably. The compiler is therefore unable to insert `store&zero` instructions, because it cannot decide whether the remainder of a cache line will be overwritten by subsequent iterations.

Similarly, in the Perfect Club benchmark `cs`, most of the avoidable misses occur in a loop that uses indirection through a second array to determine the output location (ie, `A[J[k]] = expression`). While this loop often writes an entire cache line over the course of several iterations, the compiler is unable to determine this in advance, and is therefore unable to insert `store&zero` instructions.

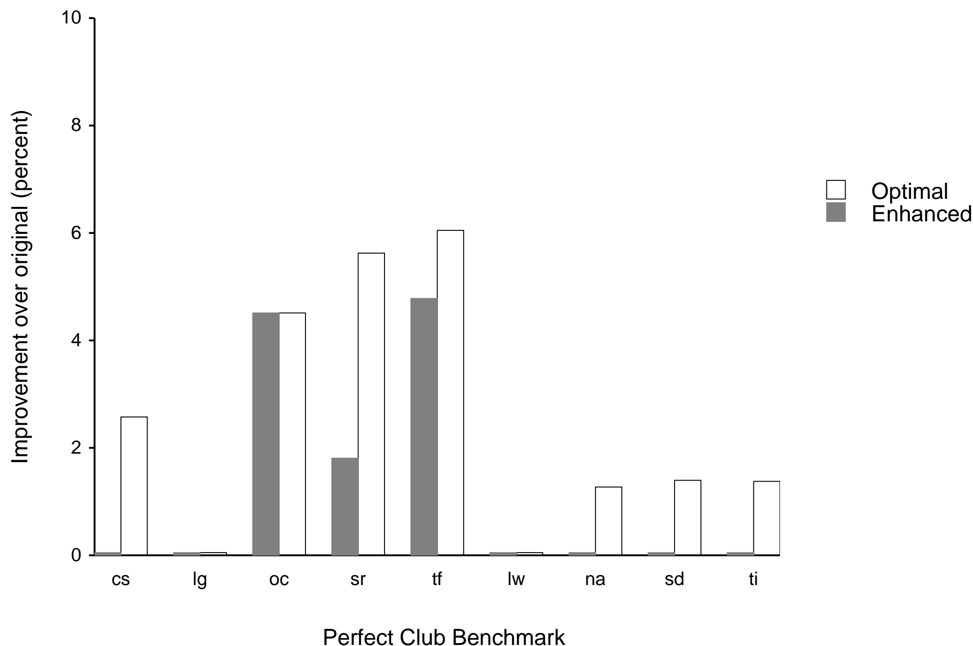

Figure 4: Improvement due to cache control instructions for the Perfect Club benchmarks. For each program, we show the improvement of the enhanced and optimal programs, compared to the original program, when the cache miss penalty is 15 instruction-times.

	enhanced improvement	optimal improvement
Livermore Loops	8.62 %	9.20 %
SPEC	0.85 %	3.26 %
Perfect Club	1.44 %	3.42 %

Figure 5: Mean improvement for each benchmark suite, for a 16 kbyte cache with 32 byte lines.

As these examples demonstrate, many misses that would be avoidable given perfect knowledge of the future are not avoidable in practice.

5.2 Operating System Measurements

To estimate the effect of cache control instructions on operating system performance, we instrumented the operating system kernel on our department's main computer. We chose to measure this machine because it represents a typical time-sharing workload. The machine is a DECsystem 5500 running Ultrix 4.1, and is the hub of our department's electronic mail system; it also acts as a fileserver and is used for document preparation and other miscellaneous tasks. On a typical afternoon, about 100 users are logged in (most of them inactive), several hundred processes exist, and the machine rarely pages to disk. Most CPU-intensive jobs in our department are run on other machines.

	9am – 5pm	5pm – 1am	1am – 9am	overall
`bzero` calls/sec	1422	1239	870	1183
`bzero` bytes/sec	179389	100119	51849	112552
`bzero` bytes/call	126	80	59	95
`bcopy` calls/sec	4783	3748	2676	3764
`bcopy` bytes/sec	394972	201945	111607	241230
`bcopy` bytes/call	82	53	41	64
`copyin` calls/sec	1707	1500	1084	1436
`copyin` bytes/sec	26878	20165	11538	19715
`copyin` bytes/call	15	13	10	13
`copyout` calls/sec	2119	1754	1364	1756
`copyout` bytes/sec	80078	44941	24753	50852
`copyout` bytes/call	37	25	18	28
total calls/sec	10031	8241	5994	8139
total bytes/sec	681317	367170	199747	424349
total bytes/call	67	44	33	52

Figure 6: Results of operating system measurements. We show the number of calls per second by the operating system kernel to each of four procedures, and the number of bytes per second handled by each procedure as a result of these calls. Results are averaged over a three-day measurement period. The kernel uses `bzero` to zero-fill memory and `bcopy` to copy memory. `copyin` and `copyout` are used to copy system call arguments and return values between user and kernel address spaces.

We added code to monitor all calls to `bzero` and `bcopy` within the kernel, as well as calls to the procedures that copy the arguments and return values of system calls between user and kernel address spaces. We measured the number of calls to each procedure, and the total number of bytes copied or zeroed by the procedure. Results averaged over several days are shown in figure 6.

Assuming a cache line size of L bytes, rewriting the four traced procedures to use `store&zero` would save at most one cache miss for every L bytes handled. (It might save less, because the region zeroed or copied might not be aligned on a cache line boundary, or might already be in the cache.) During the period with heaviest load, the four procedures handled 681317 bytes per second, so at most $681317/L$ cache misses per second would be saved. If we take a reasonable value for L, say 32, about 21000 misses per second might be saved; if the miss penalty is 500 nanoseconds, this translates to a maximum of 10.5 milliseconds saved per second of real time. Thus we expect that using cache control instructions in the operating system would lead at best to a 1% improvement in system performance. This figure is independent of the miss rate of the operating system as a whole, which is difficult to estimate.

5.3 Summary

On average, cache control instructions have only a small impact on performance — we measured a 0.85 % mean improvement on the SPEC benchmarks, and a 1.44 % mean improvement on the Perfect Club programs. We note, though, that none of the benchmarks slowed down as a result of adding these instructions, and some benchmarks sped up significantly. This last fact suggests that cache control instructions may be worthwhile, especially since they are so cheap to implement.

The effect on operating system performance was also small. We estimate our department's workhorse computer would speed up by less than 1% as a result of using cache control instructions in the operating system kernel.

6 Predicting Future Performance

We have seen that the average benefits of cache control instructions are modest on today's architectures. One might expect these benefits to be larger in the future, since memory latency is ever-increasing, and cache behavior will therefore become a more significant factor. We can use the data from our experiments to predict the usefulness of cache control instructions on future architectures. We assume that technology trends in the next few years will have three main effects:

- CPUs will get faster relative to memory. We can account for this effect by increasing the cache miss penalty in our simulations.
- First-level caches will get larger. As the transistor budget of chip designers increases, and cache misses become more costly, designers will respond by increasing the amount of on-chip cache. We can account for this effect by simulating larger caches.
- Applications will use larger data sets, as programmers and users respond to improvements in system performance. This effect is harder to model, but we can approximate it by running today's benchmarks with smaller caches.

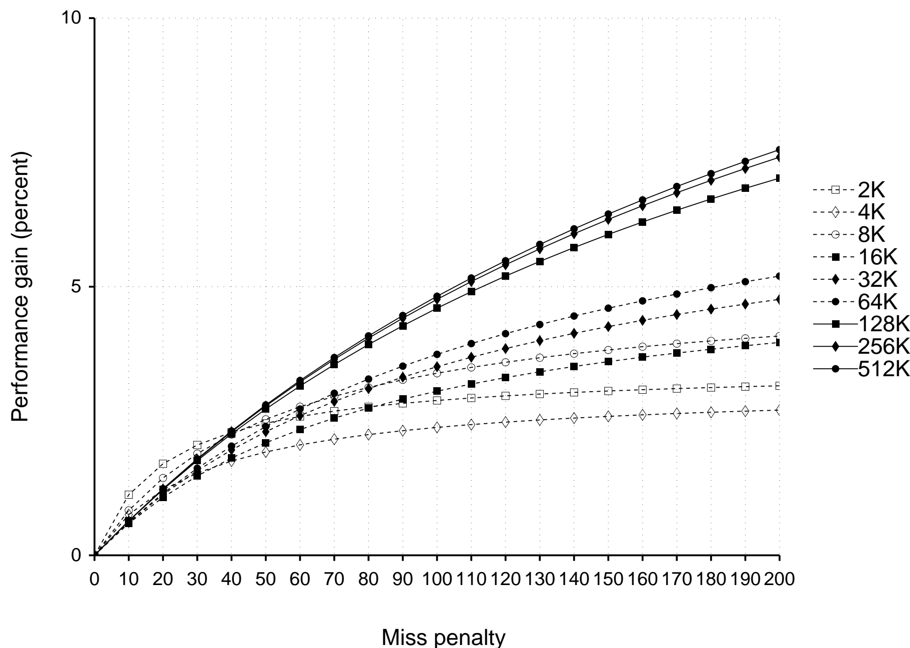

Figure 7: Predicted performance gain due to cache control instructions, under the *enhanced* scenario, as a function of miss penalty, averaged over the SPEC benchmark suite.

As this suggests, we used the results of our simulations for cache sizes ranging from 2 kbytes to 512 kbytes. From these numbers, we can predict the percentage improvement in running time as a function of the cache miss penalty (discussed in section 4.1).

Figures 7 and 8 show the predicted performance gain for the SPEC benchmarks, under the enhanced and optimal assumptions, respectively. Even for large values of the miss penalty P, the performance gains of the enhanced implementation are quite modest. The optimal implementation shows larger gains, as large as 17% for a 16 kbyte cache with $P = 200$.

Figures 9 and 10 show the results of the same experiment for the Perfect Club benchmarks. The enhanced implementation shows the same modest improvement we saw for the SPEC benchmarks, but the optimal implementation shows a larger improvement for the Perfect Club benchmarks than it did for SPEC.

7 Discussion

Our data suggest five main conclusions.

First, significant gains are possible on some scientific applications. For example, we calculate a 34% improvement on Livermore Loop 12, a 6.5% improvement on `fpppp`, and a 6% improvement on `tf`. These improvements are mostly due to loops that write an output array, one entry per loop iteration, with unit stride. If the compiler can recognize these, many cache misses can be avoided.

Second, average gains were quite modest for most of the benchmarks. With a 16 kbyte cache and 4 words per line, the average gain on the SPEC benchmarks was 0.85 %; for the Perfect Club benchmarks the average gain was 1.44 %.

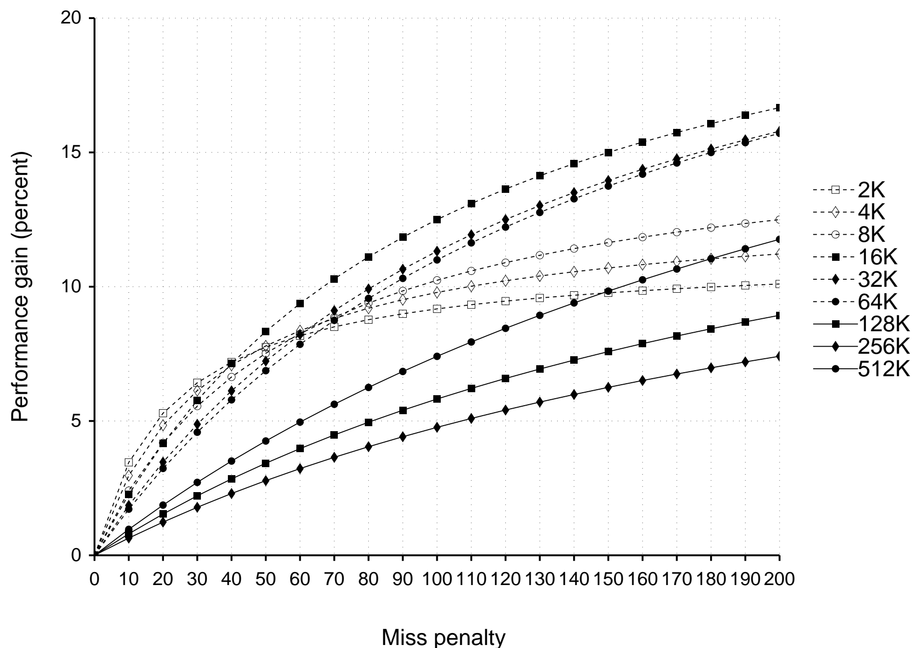

Figure 8: Predicted performance gain due to cache control instructions, under the *optimal* scenario, as a function of miss penalty, averaged over the SPEC benchmark suite.

Third, under current architectural assumptions, the average gain for SPEC and Perfect Club benchmarks is limited to about 3.3%. It is possible, though, that alternative programming methodologies may allow greater potential benefits.

Fourth, the effect of cache control instructions on operating system performance is small. Our measurements indicate that our department's workhorse computer would speed up by at most 1% due to the improved performance of operating system primitives.

Finally, the predicted increase in CPU speed relative to memory has a significant but bounded effect on the benefit due to cache control instructions. The bound stems from the fact that only a fixed fraction of cache misses are eliminated; the bulk of the cache misses still remain, and must be paid for. For current cache sizes, the performance gain is limited to about 30% for the SPEC benchmarks, regardless of assumptions about CPU and memory speeds.

In the final analysis, we believe cache control instructions are a useful feature. Although their average performance benefit is smaller than one might hope, it easily outweighs the small cost of adding these instructions. Furthermore, in no case did the addition of these instructions cause a performance loss. In combination with other techniques such as prefetching and compiler optimizations, cache control instructions can help to reduce the cost of memory access.

8 Acknowledgments

The authors would like to thank Hank Levy, Jean-Loup Baer and Susan Eggers for their comments on the ideas presented in this paper, Alex Klaiber for the tools that were used early in the research, Mike Smith for his help with pixie, and Jeff Chase, Jeff Gee, Bruce Cole and Chandu Thekkath for

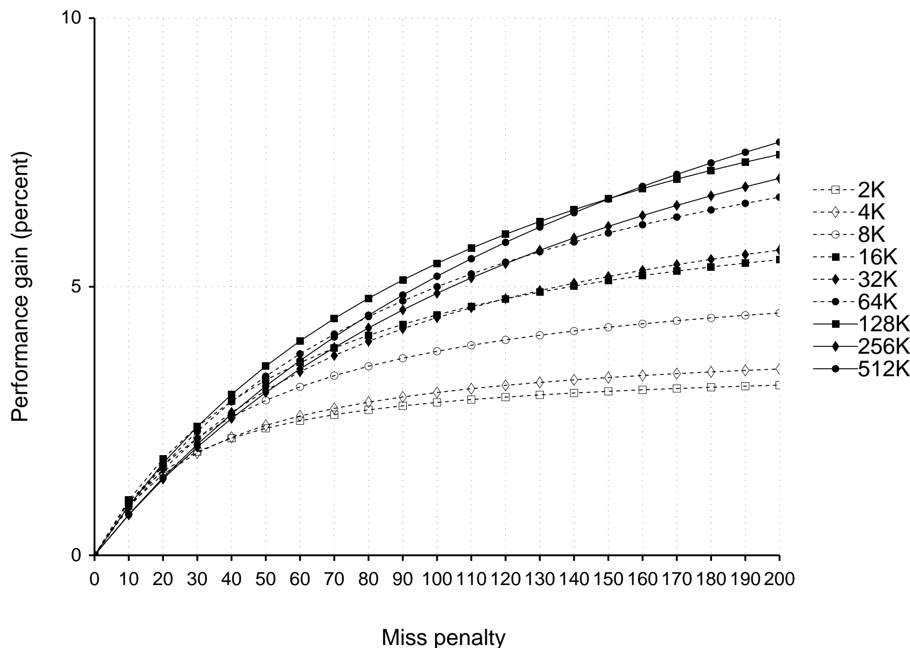

Figure 9: Predicted performance gain due to cache control instructions, under the *enhanced* scenario, as a function of miss penalty, averaged over the Perfect Club benchmark suite.

their help making Ultrix do the right thing. We'd especially like to thank Nancy Johnson-Burr for all her help keeping our modified operating system up and available.

References

[Berry et al. 89] M. Berry, D. Chen, D. Kuck, S. Lo, Y. Pang, L. Pointer, R. Roloff, A. Samah, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, R. Goodrum, and J. Martin. The Perfect Club benchmarks: Effective performance evaluation of supercomputers. *Intl. Journal of Supercomputer Applications*, 3(3):5–40, 1989.

[Gupta et al. 91] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W.-D. Weber. Comparative evaluation of latency reducing and tolerating techniques. In *18th Intl. Symposium on Computer Architecture*, pages 254–263, 1991.

[Hew 90] Hewlett-Packard Co. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, first edition, Nov. 1990.

[Jouppi 90] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *17th Intl. Symposium on Computer Architecture*, pages 364–373, 1990.

[Kane & Heinrich 92] G. Kane and J. Heinrich. *MIPS RISC Architecture*. Prentice Hall, Englewood Cliffs, New Jersey, 1992.

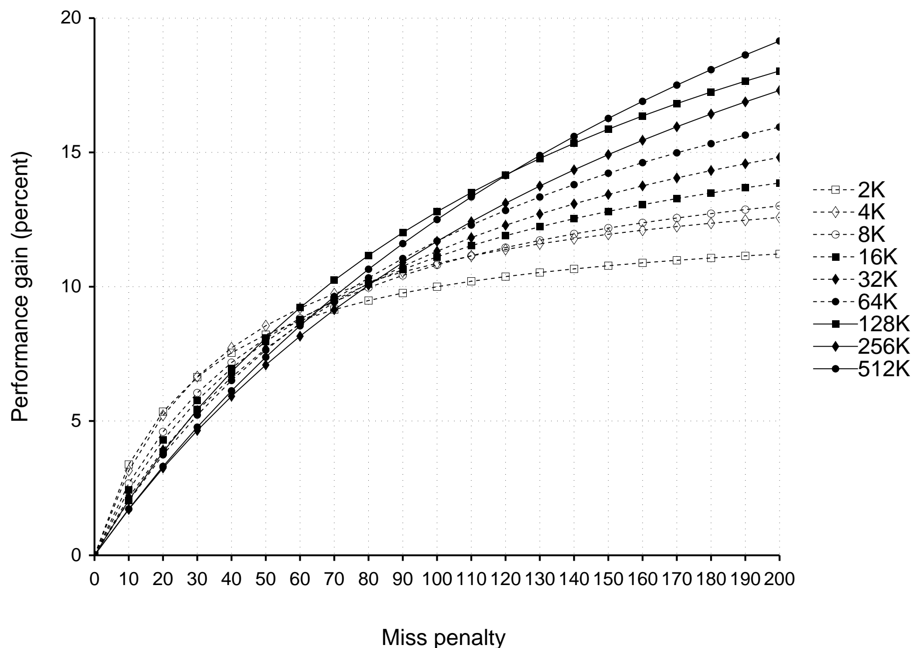

Figure 10: Predicted performance gain due to cache control instructions, under the *optimal* scenario, as a function of miss penalty, averaged over the Perfect Club benchmark suite.

[McFarling 92] S. McFarling. Cache replacement with dynamic exclusion. Technical Report TN-22, DEC WRL, 1992. To appear in Proceedings of 19th Intl. Symposium on Computer Architecture.

[McMahon 72] F. McMahon. FORTRAN CPU performance analysis. Technical report, Lawrence Livermore National Laboratories, 1972.

[MIPS Computer Systems 86] I. MIPS Computer Systems. *Languages and Programmer's Manual*, 1986.

[SPE 89] System Performance Evaluation Cooperative. *SPEC Benchmark Suite Release 1.0*, 1989.

[Wilkes 92] J. Wilkes. Personal communication, 1992.

[Wittenbrink et al. 92] C. Wittenbrink, A. Somani, and C. Chen. Cache write generate for high performance parallel processing. Submitted for publication, 1992.

A Detailed Data: Miss Frequency

This appendix shows the cache miss frequencies of all simulator runs. (Miss frequency is the number of cache misses per instruction executed.) Although the main paper gives performance figures in terms of improvement over the *original* scenario, this appendix quotes miss frequency, since that is a direct output of our simulator. For each benchmark, and each cache size, the data tables give three miss frequencies. The top number is the miss frequency under the *original* scenario, the middle number is the miss frequency for the *enhanced* scenario, and the bottom number is for the *optimal* scenario. We used a 32-byte line size in all our simulations. (Note to reviewers: N/A entries will be filled in for the final paper.)

Recall that, as explained in section 5.1, we were unable to run all of the SPEC and Perfect Club benchmarks, and we omitted some of the Livermore Loops because they presented no opportunity for improvement.

	2K	4K	8K	16K	32K	64K	128K	256K	512K
Kernel 1	0.139	0.139	0.139	0.060	0.060	0.060	0.060	0.060	0.060
	0.119	0.119	0.119	0.040	0.040	0.040	0.040	0.040	0.040
	0.119	0.119	0.119	0.040	0.040	0.040	0.040	0.040	0.040
Kernel 2	0.070	0.068	0.067	0.067	0.067	0.065	0.065	0.065	0.057
	0.057	0.055	0.054	0.054	0.054	0.052	0.052	0.052	0.044
	0.056	0.055	0.054	0.053	0.053	0.052	0.052	0.052	0.043
Kernel 5	0.070	0.070	0.070	0.070	0.070	0.070	0.070	0.070	0.070
	0.047	0.047	0.047	0.047	0.047	0.047	0.047	0.047	0.047
	0.046	0.046	0.046	0.046	0.046	0.046	0.046	0.046	0.046
Kernel 6	0.038	0.034	0.029	0.023	0.020	0.018	0.017	0.017	0.017
	0.038	0.034	0.029	0.023	0.020	0.018	0.017	0.017	0.017
	0.038	0.034	0.029	0.023	0.020	0.018	0.017	0.017	0.017
Kernel 7	0.023	0.023	0.023	0.023	0.023	0.023	0.023	0.023	0.023
	0.017	0.017	0.017	0.017	0.017	0.017	0.017	0.017	0.017
	0.017	0.017	0.017	0.017	0.017	0.017	0.017	0.017	0.017
Kernel 8	0.066	0.058	0.055	0.053	0.052	0.052	0.051	0.051	0.050
	0.059	0.051	0.048	0.046	0.045	0.045	0.044	0.044	0.043
	0.057	0.051	0.047	0.046	0.045	0.044	0.044	0.044	0.043
Kernel 9	0.044	0.044	0.044	0.044	0.044	0.044	0.044	0.044	0.044
	0.040	0.040	0.040	0.040	0.040	0.040	0.040	0.040	0.040
	0.040	0.040	0.040	0.040	0.040	0.040	0.040	0.040	0.040
Kernel 10	0.080	0.030	0.030	0.030	0.030	0.030	0.030	0.030	0.030
	0.077	0.027	0.027	0.027	0.027	0.027	0.027	0.027	0.027
	0.069	0.028	0.028	0.027	0.028	0.028	0.028	0.028	0.028
Kernel 11	0.069	0.069	0.069	0.069	0.069	0.069	0.069	0.069	0.069
	0.035	0.035	0.035	0.035	0.035	0.035	0.035	0.035	0.035
	0.034	0.034	0.034	0.034	0.034	0.034	0.034	0.034	0.034
Kernel 12	0.072	0.072	0.072	0.072	0.072	0.072	0.072	0.072	0.072
	0.036	0.036	0.036	0.036	0.036	0.036	0.036	0.036	0.036
	0.036	0.036	0.036	0.036	0.036	0.036	0.036	0.036	0.036
Kernel 13	0.041	0.039	0.038	0.007	0.006	0.006	0.006	0.006	0.006
	0.022	0.021	0.020	0.001	0.000	0.000	0.000	0.000	0.000
	0.022	0.020	0.019	0.001	0.000	0.000	0.000	0.000	0.000
Kernel 14	0.059	0.039	0.037	0.036	0.036	0.035	0.031	0.031	0.030
	0.049	0.029	0.027	0.026	0.026	0.025	0.023	0.023	0.022
	0.040	0.023	0.021	0.019	0.020	0.020	0.018	0.017	0.017
Kernel 18	0.120	0.120	0.120	0.120	0.120	0.116	0.110	0.110	0.110
	0.117	0.117	0.117	0.117	0.117	0.113	0.107	0.107	0.107
	0.116	0.116	0.116	0.116	0.116	0.113	0.107	0.107	0.107
Kernel 19	0.067	0.067	0.066	0.066	0.066	0.065	0.063	0.059	0.052
	0.056	0.056	0.055	0.055	0.055	0.054	0.052	0.048	0.041
	0.055	0.055	0.055	0.055	0.055	0.054	0.052	0.048	0.041
Kernel 20	0.046	0.040	0.037	0.036	0.035	0.035	0.028	0.027	0.027
	0.042	0.036	0.033	0.032	0.031	0.031	0.024	0.023	0.023
	0.041	0.036	0.033	0.032	0.031	0.031	0.024	0.024	0.024
Kernel 21	0.041	0.030	0.025	0.023	0.022	0.020	0.020	0.020	0.020
	0.041	0.030	0.025	0.023	0.022	0.020	0.020	0.020	0.020
	0.041	0.030	0.025	0.023	0.022	0.020	0.020	0.020	0.020
Kernel 22	0.040	0.036	0.035	0.034	0.033	0.033	0.033	0.033	0.033
	0.027	0.023	0.022	0.021	0.020	0.020	0.020	0.020	0.020
	0.026	0.023	0.021	0.020	0.020	0.020	0.020	0.020	0.020

Figure A.1: Cache miss frequencies for the Livermore Loop benchmarks.

	2K	4K	8K	16K	32K	64K	128K	256K	512K
Kernel 12	0.072	0.072	0.072	0.072	0.072	0.072	0.072	0.072	0.072
	0.036	0.036	0.036	0.036	0.036	0.036	0.036	0.036	0.036
	0.036	0.036	0.036	0.036	0.036	0.036	0.036	0.036	0.036
Kernel 13	0.041	0.039	0.038	0.007	0.006	0.006	0.006	0.006	0.006
	0.022	0.021	0.020	0.001	0.000	0.000	0.000	0.000	0.000
	0.022	0.020	0.019	0.001	0.000	0.000	0.000	0.000	0.000
Kernel 14	0.059	0.039	0.037	0.036	0.036	0.035	0.031	0.031	0.030
	0.049	0.029	0.027	0.026	0.026	0.025	0.023	0.023	0.022
	0.040	0.023	0.021	0.019	0.020	0.020	0.018	0.017	0.017
Kernel 18	0.120	0.120	0.120	0.120	0.120	0.116	0.110	0.110	0.110
	0.117	0.117	0.117	0.117	0.117	0.113	0.107	0.107	0.107
	0.116	0.116	0.116	0.116	0.116	0.113	0.107	0.107	0.107
Kernel 19	0.067	0.067	0.066	0.066	0.066	0.065	0.063	0.059	0.052
	0.056	0.056	0.055	0.055	0.055	0.054	0.052	0.048	0.041
	0.055	0.055	0.055	0.055	0.055	0.054	0.052	0.048	0.041
Kernel 20	0.046	0.040	0.037	0.036	0.035	0.035	0.028	0.027	0.027
	0.042	0.036	0.033	0.032	0.031	0.031	0.024	0.023	0.023
	0.041	0.036	0.033	0.032	0.031	0.031	0.024	0.024	0.024
Kernel 21	0.041	0.030	0.025	0.023	0.022	0.020	0.020	0.020	0.020
	0.041	0.030	0.025	0.023	0.022	0.020	0.020	0.020	0.020
	0.041	0.030	0.025	0.023	0.022	0.020	0.020	0.020	0.020
Kernel 22	0.040	0.036	0.035	0.034	0.033	0.033	0.033	0.033	0.033
	0.027	0.023	0.022	0.021	0.020	0.020	0.020	0.020	0.020
	0.026	0.023	0.021	0.020	0.020	0.020	0.020	0.020	0.020

Figure A.2: Cache miss frequencies for Livermore Loops 13–24.

	2K	4K	8K	16K	32K	64K	128K	256K	512K
cs	0.051	0.032	0.025	0.013	0.009	0.008	0.005	0.000	0.000
	0.051	0.032	0.025	0.013	0.009	0.008	0.005	0.000	0.000
	0.043	0.027	0.021	0.011	0.008	0.007	0.004	0.000	0.000
lg	0.011	0.006	0.006	0.004	0.003	0.002	0.002	0.002	0.002
	0.010	0.006	0.006	0.004	0.003	0.002	0.002	0.002	0.002
	0.009	0.005	0.005	0.004	0.003	0.002	0.002	0.002	0.001
lw	0.021	0.000	0.008	0.003	N/A	N/A	N/A	N/A	N/A
	0.021	0.000	0.008	0.003	N/A	N/A	N/A	N/A	N/A
	0.018	0.000	0.007	0.003	N/A	N/A	N/A	N/A	N/A
na	N/A	N/A	0.013	0.013	N/A	N/A	N/A	N/A	N/A
	N/A	N/A	0.013	0.013	N/A	N/A	N/A	N/A	N/A
	N/A	N/A	0.012	0.012	N/A	N/A	N/A	N/A	N/A
oc	0.055	0.035	0.028	0.026	0.024	0.008	0.000	N/A	N/A
	0.051	0.031	0.024	0.022	0.020	0.004	0.000	N/A	N/A
	0.049	0.030	0.024	0.022	0.020	0.005	0.000	N/A	N/A
sd	0.025	0.021	0.012	0.006	0.001	0.000	0.000	0.000	0.000
	0.025	0.021	0.012	0.006	0.001	0.000	0.000	0.000	0.000
	0.024	0.020	0.012	0.005	0.001	0.000	0.000	0.000	0.000
sr	0.075	0.065	0.047	0.046	0.037	0.030	0.028	0.027	0.024
	0.073	0.063	0.045	0.044	0.035	0.028	0.026	0.025	0.022
	0.069	0.058	0.041	0.040	0.031	0.024	0.022	0.021	0.018
tf	0.041	0.030	0.024	0.021	0.018	0.015	0.012	0.007	0.005
	0.037	0.026	0.020	0.017	0.015	0.012	0.009	0.005	0.003
	0.035	0.024	0.019	0.016	0.013	0.011	0.008	0.004	0.003
ti	0.000	N/A	0.020	0.007	N/A	N/A	N/A	N/A	N/A
	0.000	N/A	0.020	0.007	N/A	N/A	N/A	N/A	N/A
	0.000	N/A	0.019	0.006	N/A	N/A	N/A	N/A	N/A

Figure A.3: Cache miss frequencies for the Perfect Club benchmarks we simulated.

	2K	4K	8K	16K	32K	64K	128K	256K	512K
dnasa7	0.000	N/A	0.101	0.033	N/A	N/A	N/A	N/A	N/A
	0.000	N/A	0.101	0.033	N/A	N/A	N/A	N/A	N/A
	0.000	N/A	0.098	0.033	N/A	N/A	N/A	N/A	N/A
doduc	0.043	0.027	0.017	0.008	0.003	0.001	0.000	0.000	0.000
	0.041	0.026	0.016	0.008	0.003	0.001	0.000	0.000	0.000
	0.035	0.022	0.013	0.005	0.002	0.000	0.000	0.000	0.000
eqntott	0.017	0.011	0.009	0.008	0.007	0.006	0.004	0.003	0.002
	0.017	0.011	0.009	0.008	0.007	0.006	0.004	0.003	0.002
	0.016	0.011	0.009	0.008	0.007	0.006	0.004	0.003	0.001
espresso	0.028	0.017	0.013	0.006	0.003	0.002	0.001	0.000	0.000
	0.027	0.017	0.013	0.006	0.003	0.002	0.001	0.000	0.000
	0.026	0.015	0.011	0.005	0.002	0.001	0.001	0.000	0.000
fpppp	0.061	0.027	0.022	0.017	0.016	0.015	0.000	0.000	0.000
	0.060	0.027	0.022	0.017	0.016	0.015	0.000	0.000	0.000
	0.051	0.020	0.017	0.012	0.011	0.011	0.000	0.000	0.000
tomcatv	0.118	0.098	0.052	0.030	0.026	0.026	0.026	0.025	0.025
	0.114	0.094	0.048	0.026	0.022	0.022	0.022	0.021	0.021
	0.112	0.091	0.047	0.025	0.022	0.021	0.021	0.021	0.020
xlisp	0.030	0.018	0.010	0.006	0.003	0.001	0.000	0.000	0.000
	0.028	0.017	0.009	0.006	0.003	0.001	0.000	0.000	0.000
	0.027	0.016	0.009	0.005	0.002	0.001	0.000	0.000	0.000

Figure A.4: Cache miss frequencies for the SPEC benchmarks we simulated.

	2K	4K	8K	16K	32K	64K	128K	256K	512K
cs	0.018	0.011	0.009	0.004	0.003	0.002	0.002	0.000	0.000
	0.013	0.008	0.006	0.003	0.002	0.002	0.001	0.000	0.000
lg	0.005	0.002	0.001	0.001	0.000	0.000	0.000	0.000	0.000
	0.004	0.001	0.001	0.000	0.000	0.000	0.000	0.000	0.000
lw	0.007	0.000	0.003	0.001	N/A	N/A	N/A	N/A	N/A
	0.005	0.000	0.002	0.001	N/A	N/A	N/A	N/A	N/A
na	N/A	N/A	0.004	0.004	N/A	N/A	N/A	N/A	N/A
	N/A	N/A	0.002	0.003	N/A	N/A	N/A	N/A	N/A
oc	0.036	0.026	0.022	0.021	0.019	0.005	0.000	N/A	N/A
	0.035	0.026	0.022	0.021	0.019	0.005	0.000	N/A	N/A
sd	0.005	0.004	0.003	0.001	0.000	0.000	0.000	0.000	0.000
	0.005	0.003	0.003	0.000	0.000	0.000	0.000	0.000	0.000
sr	0.028	0.025	0.018	0.018	0.015	0.013	0.012	0.011	0.011
	0.028	0.024	0.018	0.018	0.014	0.012	0.011	0.011	0.010
tf	0.017	0.014	0.012	0.010	0.009	0.008	0.006	0.004	0.003
	0.016	0.013	0.011	0.010	0.008	0.007	0.005	0.003	0.002
ti	0.000	N/A	0.007	0.003	N/A	N/A	N/A	N/A	N/A
	0.000	N/A	0.007	0.003	N/A	N/A	N/A	N/A	N/A

Figure B.1: Writeback frequencies for the Perfect Club benchmarks we simulated.

B Detailed Data: Writeback Frequency

This appendix shows the writeback frequencies of all simulator runs. (Writeback frequency is the number of writebacks per instruction executed.) For each benchmark, and each cache size, the data tables give two writeback frequencies. The top number is the writeback frequency under the *original* scenario, and the bottom number is the writeback frequency for the *optimal* scenario. We used a 32-byte line size in all our simulations. (Note to reviewers: N/A entries will be filled in for the final paper.)

As above, we give results for only the programs that we simulated. We omit results for the Livermore Loops; since these programs do not re-use their data, essentially all their writebacks could be eliminated.

	2K	4K	8K	16K	32K	64K	128K	256K	512K
dnasa7	0.000	N/A	0.045	0.019	N/A	N/A	N/A	N/A	N/A
	0.000	N/A	0.044	0.018	N/A	N/A	N/A	N/A	N/A
doduc	0.014	0.010	0.006	0.004	0.002	0.000	0.000	0.000	0.000
	0.010	0.007	0.004	0.002	0.001	0.000	0.000	0.000	0.000
eqntott	0.001	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
	0.001	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
espresso	0.008	0.007	0.006	0.003	0.001	0.001	0.000	0.000	0.000
	0.007	0.006	0.005	0.003	0.001	0.001	0.000	0.000	0.000
fpppp	0.020	0.011	0.009	0.008	0.007	0.007	0.000	0.000	0.000
	0.016	0.006	0.006	0.004	0.004	0.004	0.000	0.000	0.000
tomcatv	0.030	0.027	0.017	0.008	0.008	0.007	0.007	0.007	0.007
	0.028	0.025	0.017	0.008	0.007	0.007	0.007	0.007	0.007
xlisp	0.015	0.010	0.006	0.005	0.002	0.001	0.000	0.000	0.000
	0.013	0.008	0.005	0.004	0.002	0.001	0.000	0.000	0.000

Figure B.2: Writeback frequencies for the SPEC benchmarks we simulated.