Appeared in the proceedings of the International Conference on Computer Design (ICCD '99), October 1999.

Pursuing the Performance Potential of Dynamic Cache Line Sizes*

Peter Van Vleet Eric Anderson Lindsay Brown Jean-Loup Baer Anna Karlin
Department of Computer Science & Engineering
University of Washington
Seattle WA 98195-2350
{pvv, eric}@cs.washington.edu, lbrown@reed.edu, {baer, karlin}@cs.washington.edu

Abstract

In this paper we examine the application of offline algorithms for determining the optimal sequence of loads and superloads (a load of multiple consecutive cache lines) for direct-mapped caches. We evaluate potential gains in terms of miss rate and bandwidth and find that in many cases optimal superloading can noticeably reduce the miss rate without appreciably increasing bandwidth. Then we examine how this performance potential might be realized. We examine the effectiveness of a dynamic online algorithm and of static analysis (profiling) for superloading and compare these to next-line prefetching. Experimental results show improvements comparable to those of the optimal algorithm in terms of miss rates.

§1 Introduction

Since their introduction over thirty years ago, caches have become ubiquitous as components of the memory hierarchy. Caches have been successful because programs exhibit locality: spatial locality, the tendency for neighboring memory locations to be referenced close together in time; and temporal locality, the tendency for referencing in the future those locations that have been referenced in the recent past. However, as the speed of processors increases much faster than the decrease in memory latency, the efficiency of caches has received more scrutiny.

Combinations of hardware and software techniques have been proposed and often implemented to improve locality and to reduce or tolerate memory latency. The basic goal is to reduce cache miss rates without unduly increasing the amount of bytes transferred between levels of the memory hierarchy. When couched in terms of improving spatial locality for data caches, the main theme of this paper, the usual policy is to support larger cache lines. Potential detrimental effects of this policy are a possible increase in cache miss rate because of more frequent conflict misses and the lack of reuse of portions of the larger lines, and to lengthen the occupancy of the bus between levels of the memory hierarchy servicing the miss. In order to palliate these effects and to take advantage of large lines, when deemed profitable, we examine the potential benefit of implementing the cache controller so that on a miss, either the missing regular size line is loaded - hereafter called the base case - or the line is *superloaded*, i.e., the missing line and surrounding lines are brought into the cache. Note that the advantages of superloading depend on the cost model for the level of the memory hierarchy under investigation. Of particular importance are the relative costs of a load and a superload.

Although the impact of these techniques has been investigated using heuristics and software or hardware assists, how much can be gained if these techniques were used optimally is not known. In this paper, after briefly introducing an optimal offline algorithm for choosing between loads and superloads, we derive the margin of maximum possible improvement on the integer Spec95 benchmark suite. We then analyze the performances, when compared to the optimal and base cases, of (1) an online algorithm that dynamically monitors the reuse and conflicts of cached lines, (2) a static profiling tool that decides, via the value of the Program Counter (PC), which load instructions should be modified to become superloads, and (3) traditional next-line prefetching.

The rest of the paper is as follows. In Section 2, we present a brief summary of related work and in Section 3 the methodology that we used for simulating the performance of the various algorithms. Section 4 is devoted to the description and performance of the optimal superloading algorithm. Section 5 and Section 6 describe respectively the proposed online and profiling algorithms, present results of simulations, and compare these results with those obtained by the optimal algorithm. Section 7 examines how superloading compares to next-line prefetching. Section 8 concludes and suggests further study.

*This research was supported in part by National Science Foundation grants MIP-9700970 and EIA-9870740, US-Israel Binational Science Foundation grant 96-00247, an IBM Graduate Research Fellowship and a gift from the Intel Corporation.

§2 Related Work

Closest to our own study, Johnson et al. [7] [8] investigate hardware assists, a Spatial Locality Detection Table (SLT) and a Memory Address Table (MAT), for dynamic fetch size choices. As with our work, they deal with what we define as *atomic* superloads, i.e. the superload operation will always bring in all of its constituent lines into the cache. The goals of our study differ from theirs in two ways. First, we are interested in the optimal case and thus are at this time restricted to the study of direct-mapped caches. Second, our online algorithm focuses on the trade-off between cache misses and bytes transferred between memory and cache while, because the machine simulated in their work has ample bandwidth, their emphasis is principally on the reduction of cache misses.

Kumar and Wilkerson [9] follow up on Johnson's work and present results for an online prediction mechanism that (in our terminology) performs *patterned* superloading. That is to say, any arbitrary pattern of lines within a superline may be loaded. The cache content of the unloaded lines is preserved. This was designed as a mechanism to reduce cache pollution from unnecessary loads rather than improve cache performance from prefetching.

Superlines are one mechanism to improve spatial locality. As we shall see, our proposed online algorithm monitors the reuse of lines in superlines in order to decide whether loads or superloads should be performed. Taking advantage of reuse information is at the heart of other methods aimed at improving cache efficiency via bypassing [16], or at having two data caches one of which is devoted especially for those data that exhibit spatial locality [11].

Loading and superloading require cache technology that is able to accommodate multiple line sizes. Several studies have explored the feasibility of this idea, including Conti's ''Sector Cache'' [4] (the first cache implementation) and Seznec's ''Decoupled Sector Cache'' [13].

Belady's MIN algorithm is the most well-known offline algorithm for the study of memory hierarchies [2]. Developed in the context of paging systems, MIN gives the minimum number of page faults for a given program. Recently Temam [15] has extended Belady's algorithm to include optimal spatial reuse as well as temporal reuses. His work differs from ours in two aspects. First, his results are applicable to fully associative caches with optimal set replacement policy whereas we deal with direct mapped caches. Second, Temam's work deals with loading arbitrary *patterns* of lines within a superline while we consider *atomic* superloads.

Dynamic sizing has also been studied at other levels in the memory hierarchy. For example, page promotion policies from a page to a superpage have been proposed with the goal of either facilitating superpage management [10] or to have better TLB coverage [12].

§3 Methodology

To evaluate the effectiveness of the superload algorithms, we use the eight Spec95 integer benchmarks. Traces were collected with the SimpleScalar simulator [3]. The default machine settings for the simulator, an 64-bit 256 register RISC machine, were used. The binaries for the benchmarks were those provided with the SimpleScalar simulator.

In the context of comparing loads versus superloads, we have chosen to include only reads in our traces, though the analysis can be readily extended to include writes as well. Since we are only modeling relatively small direct-mapped first-level caches, we kept our traces between 7M and 23M reads in length for the optimal and online algorithms.

The input sets for 126.gcc and 132.ijpeg use the *test* input set. All others use the *train* input set (134.perl uses just the ''scrabbl'' input).

124.m88ksim, 129.compress and 134.perl were traced in their entirety. They contained 7.5M, 11.6M and 23.2M reads respectively. The five other applications were sampled with five separate sections of four million consecutive reads. The five sections were uniformly spaced throughout the trace and then concatenated together to form one 20 million read trace. This sampling method has been shown to give accurate results for the cache sizes we are considering [5].

For our experiments with profiling, we used the above traces as the training input. When evaluating profiling, we ran a full simulation of the reference input for each benchmark. In each case, the simulations of the profiling method were at least 10 times longer and used different data sets than the traces used as training input.

The running time of the optimal algorithms becomes an issue if profiling is to be of practical value. Currently, on a dual processor 200MHz Pentium Pro machine, a 20-million reference trace can be processed in about five minutes, a reasonable overhead that has to be incurred only once per application.

§4 Optimal Algorithm

§4.1 Machine Model

We assume an architecture with the ability to choose between two line sizes on a cache miss, a load and a superload. On a load, the architecture brings the ''normal'' cache line into the cache. On a superload, a superline, defined as L consecutive lines determined by address masking, is brought into the cache. For the remainder of this paper, we will assume four lines to a superline. The labeling of these operations with ''lines and superlines'' as opposed to ''sublines and lines'' is arbitrary. In the paper we shall always use the former.

§4.2 Cost Model

Ideally, an optimal algorithm would present a sequence of loads and superloads which would provide the most improvement in overall running time of an application on a given machine. However, encompassing all the details of a machine into an algorithm would quickly become too unwieldy. Instead we optimize for a simplified memory access model, where every load has a cost of *l* and a superload has a cost of *s*. This is expressed as the ratio *s*:*l*. (Where two actions have the same effect in terms of ''cost'', we break ties in favor of loading.)

In this machine model there is an intrinsic trade-off between improving cache hit ratio and minimizing additional bytes transferred into the cache. At one extreme, the cost ratio of 1:1 effectively optimizes for cache hit ratio, disregarding bytes transferred. Superloads are done if one useful prefetch from another line will occur, even if at the expense of transferring two potentially useless lines into the cache. At the other extreme, the cost ratio 3:1 may perform superloads only if there is a useful prefetch to every line in the superline. This guarantees that bandwidth will not be increased over the base case, at the expense of limited superloading and thus limited ability to improve cache hit ratio. The cost ratio of 2:1 is a hybrid, where superloads may occur only if at least 3 of the lines will be used.

In the cost models, only the relative cost ratios are significant. Furthermore, where the cost of a cache hit is disregarded, only the integral cost ratios are of interest. Cost models greater than 3:1 will always perform loads, whereas cost models less than 1:1 are unrealistic. Hence (as we assume four lines in a superline) we need only consider the cost models 1:1, 2:1 and 3:1. We refer to the optimal algorithms based on each of these cost models as Opt 1:1, Opt 2:1 and Opt 3:1 respectively.

§4.3 Algorithm Description

We have developed two offline algorithms which given a cost model as described above, can provide an optimal (lowest cost) sequence of loads and superloads. Here we briefly characterize the algorithms. For more detailed information see the technical report [1].

The algorithm we use in this paper is a linear time, constant space algorithm for computing the optimal cost in the case of superloads. This algorithm is based on a dynamic programming paradigm applied to the suffix of the reference stream. It calculates optimal cost by classifying all possible cache states, defining each optimal cost as a minimum over possible cache operations of the sum of (1) the cost of the operation and (2) the optimal cost of satisfying the suffix from the resulting state. The number of distinct states for one superline is a constant depending only on the size of the superline, and is small for direct-mapped caches.

The algorithm runs in time linear in the length of the reference sequence, but exponential in the number of lines in a superline. The algorithm can be extended to set associativity with LRU replacement, but at a significant cost in execution time.

The second, more complex algorithm also relies on the assumption of a direct-mapped cache, but can be applied to compute the optimal cost in the context of bypassing. However, optimal bypassing combined with optimal superloading is beyond the scope of this paper. This second algorithm, and some initial bypassing results, are described in [1].

§4.4 Results of the Optimal Algorithm

In this subsection, we show the results of applying the optimal results algorithms with various costs and compare these results with those obtained for the base cases of ''only loads'' and ''only superloads''. These experiments will give us the potential margins of improvement in cache hit rates and the accompanying penalties in the number of bytes read. For example (cf. Figure 3 and Figure 4), the Opt 2:1 algorithm with a line size of 16 bytes for a 16K direct-mapped cache yields a decrease of cache miss rate between 11% and 29% (average 19%) with a decrease in byte reads from 23% to 60% (average 43%) with respect to the best static line size of 32 bytes for a 16K cache (cf. Table 1).

	Miss rate (%)				Bytes read			
Line size	8	16	32	64	8	16	32	64
compress	5.10	4.79	4.79	5.51	100.00	188.06	375.81	864.33
gcc	6.18	5.22	4.67	4.80	100.00	169.00	302.46	621.94
go	4.14	4.56	5.73	7.27	100.00	220.40	553.78	1403.95
ijpeg	3.99	2.50	1.74	1.32	100.00	125.16	174.17	263.87
li	8.55	6.14	4.21	3.21	100.00	143.72	196.92	300.67
m88ksim	2.54	2.09	1.60	1.37	100.00	164.19	251.35	430.77
perl	3.96	3.73	3.67	3.89	100.00	188.46	370.81	787.03
vortex	8.15	6.62	6.57	6.55	100.00	162.33	322.14	642.75
average	5.33	4.46	4.12	4.24	100.00	170.17	318.43	664.41

Table 1: Miss rates and bytes read of the base case for the eight benchmark applications at a series of cache line sizes, 8, 16, 32, and 64 bytes, and a cache size of 16K. For each application, bytes read is normalized to the quantity for the case of an 8-byte line size (set to 100).

We analyze each application of the Spec95 benchmark suite, comparing the base cases of the algorithm that performs no superloads and the algorithm that performs only superloads to the three optimal algorithms, Opt 1:1, Opt 2:1, and Opt 3:1, at the same line size. Each of the algorithms is simulated using a 32-byte line size, which is the line size that minimizes average miss rate in the base case for the benchmark suite as a whole. The resulting miss rates are shown in the top graph of Figure 1. In all cases except Opt 3:1 for *li* (*li* favors long cache lines, cf. Table 1), the optimal algorithms achieve lower miss rates than

either of the base case algorithms, and in some cases, fairly significantly. As expected, we also see that (a) the miss rate decreases as the superload cost used by the optimal algorithm decreases, and (b) the percentage of misses on which a superload is performed increases as superload cost decreases.

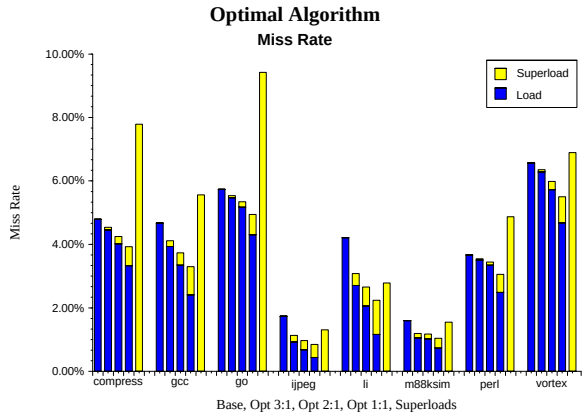

Figure 1: Miss rates for the eight benchmark applications. The base case (no superloads), the three optimal algorithms Opt 3:1, Opt 2:1, and Opt 1:1 (see text), and the algorithm that always performs a superload are each simulated at a cache size of 16K and a line size of 32 bytes.

Our results show that the proportion of superloads in the optimal algorithms is generally greatest for those applications, such as *ijpeg* and *li*, that already have good performance in the sense of a low miss rate for larger line sizes. We believe this is because these programs share the same underlying factor, spatial locality. That is, an application that exhibits a high degree of spatial locality will tend both to have a low miss rate and to make effective use of superloads.

Another interesting observation is that for some applications a small number of superloads can lead to a significant improvement in miss rate. The superloads that are chosen in the Opt 3:1 algorithm are precisely those that result in complete use of the superloaded lines, because three additional hits are necessary for the cost of the superload to be preferred by that algorithm. These high-efficiency superloads do not lead to an increase in bandwidth (see Figure 2, left two columns), but can lead to a substantial decrease in miss rate (*li* or *ijpeg*).

In Figure 2, we display for each application the bytes read under the base case (no superloads) and under the three optimal algorithms, Opt 3:1, Opt 2:1, and Opt 1:1. (The base case of the only-superload algorithm is not displayed; its values are generally much higher, and would distort the scale.) We see from the graph that (as noted above) the Opt 3:1 algorithm uses the same bandwidth, the Opt 2:1 algorithm uses slightly greater bandwidth, and the Opt 1:1 algorithm significantly greater bandwidth than the corresponding base case. The substantial additional bandwidth required by Opt 1:1 compared to Opt 2:1 can be contrasted to the modest additional benefit in miss rate, suggesting that there are diminishing returns to the additional superloads, and indicating that for Opt 1:1 in many cases only one additional extra line is used by a superload.

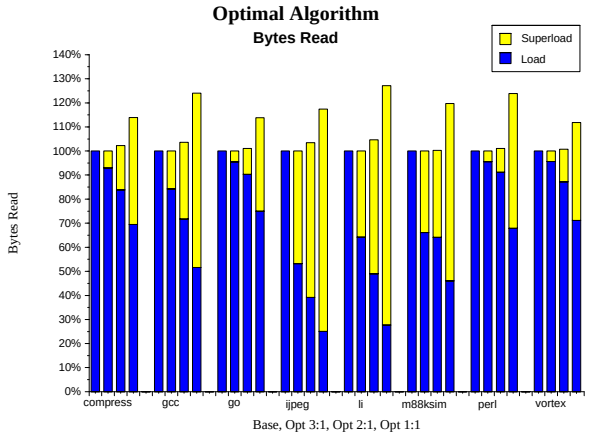

Figure 2: Bytes read for the eight benchmark applications. The only "superloads" algorithm is not displayed in the bytes read graph; its values are much higher than the others, and would distort the scale.

Since the ratio 2:1 seems to be the cost ratio that provides the best overall trade-off between miss rate and bytes transferred, we performed additional experiments and comparisons using this cost model.

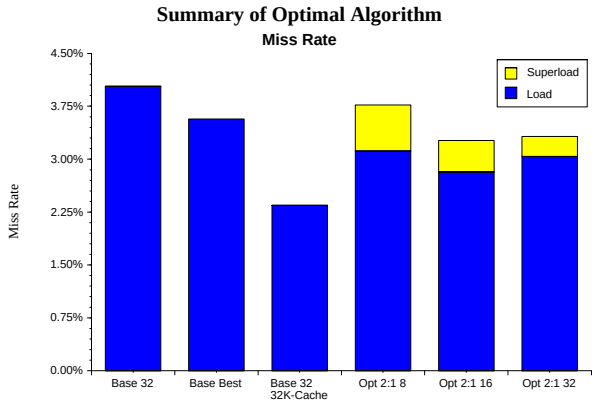

Figure 3: Miss rate for Opt 2:1 over the range of line sizes of interest. Cache size is 16K (except as noted). The base case (no superloads) is presented in three versions: at the best overall line size for miss rate (32 bytes), at the best individual line sizes for each application; and at a cache size twice as large (32K, 32 byte line size). Results for the individual applications are averaged.

In Figure 3, we display a summary of the miss rate results of Opt 2:1 over a range of line sizes. For comparison, we also display the base case results (loads only, at the best overall size, i.e., 32 bytes), the results when best per-application line size is selected (e.g., 8 bytes for *go* and 64 bytes for *li*, cf. Table 1), and a cache which is twice as large. In each case, the miss rates (and the respective contributions of loads and superloads) for the eight applications are averaged to obtain the figure displayed in the graph.

We see that the Opt 2:1 algorithm produces a miss rate at any line size that is at least as low as the average miss rate for the base case with best overall cache line. Even when the base case is simulated at each application's individual optimal line size, the Opt 2:1 algorithm yields competitive miss rates despite the fact that the cache line and superline sizes are fixed for all applications. In this sense the optimal algorithm captures the individuality of the separate applications, by producing a miss rate size for each that is comparable to its performance at its own optimal line size. Note however that the effect of increasing cache capacity cannot be matched. Doubling the cache size results, as expected, in better improvements than superloading.

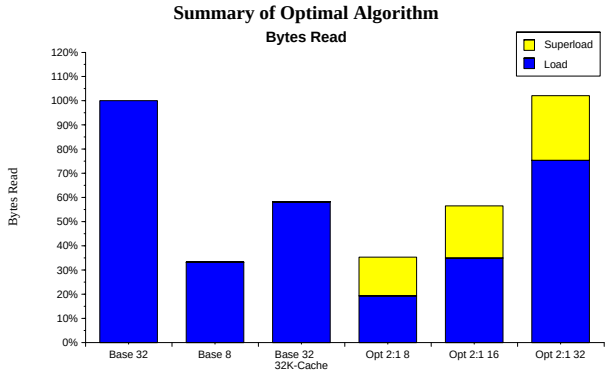

Figure 4: Bytes read for the algorithm Opt 2:1 over the range of line sizes of interest. Figure's parameters are the same as Figure 3, with the exception of the best base case, which in terms of minimizing bandwidth, is an 8-byte line for each benchmark. Measurements are normalized to the 32-byte base case.

Figure 4 presents the corresponding results for bytes read. We compare the optimal results, to the standard 32-byte line size as well as to the configuration with a cache twice as large. The best possible cache line configuration for reducing the number of bytes read is the smallest, 8 bytes. Overall the Opt 2:1 algorithm yields only a slight increase in the number of bytes transferred compared to the corresponding base case with the same line size.

In order to gauge the effect of cache size on superloads, we have carried out these same experiments across a range of cache size values. The results are qualitatively similar. A particular item of interest is the proportion of superloads in the Opt 2:1 algorithm as the cache size increases. We expect to see a higher proportion of superloads, since as the cache size increases, there are fewer conflicts in the cache, and the optimal algorithm can more often take advantage of the data placement of the additional lines. This is confirmed by our experiments. For example, for Opt 2:1 the absolute number of superloads decreases from 781K to 491K (accompanied by a drop in the overall number of misses), but the proportion of cache misses that are superloads increases from 11% to 23%, as the cache size increases from 8K to 32K.

§5 Online Prediction

§5.1 Predictor Anatomy

When implementing online algorithms, we must rely solely on previous behavior, associating past patterns with the desired outcome and assuming that this is the correct behavior the next time the pattern occurs. Fundamentally, the effectiveness of any such predictor will be limited by the intrinsic usefulness of past information in predicting the future. Practical considerations such as limited knowledge (feedback) and limited storage space (to store patterns) will further inhibit this effectiveness.

Limited knowledge is a problem inherent in memory hierarchy studies such as paging system replacement algorithms and, in our case, line size prediction. Specifically, after a prediction is made, one must provide ''correct'' feedback, which is incorporated in the history information of the predictor. While this is relatively simple for some predictors such as branch predictors or even data value predictors where the true results are known within a few cycles, and more importantly, are absolutely correct, this is quite complex for line size predictors as evidenced by the (relative) complexity of the optimal algorithm. Thus, the predictor must rely on some form of independent knowledge mechanism, which evaluates the situation and provides (imperfect) feedback.

Our prediction scheme, the Line Size Predictor (LSP), contains two distinct components, the Operation Counter Table (OCT), a lookup table which determines whether a load or superload is performed, and the Line Size Detector (LSD), a knowledge mechanism which attempts to determine in retrospect what the correct line size of a load operation should have been.

Overall, our strategy on a miss to a particular line is to consider the reference stream since the previous miss to that particular line. Given this segment of the reference stream, we determine, approximately, if it would have been profitable to perform a superload on the previous miss. If so, we'd like to superload on the current miss, on the theory that the recent past is a good predictor of the future. If, on the other hand, a superload would not have been profitable on the previous miss, we'd like to load on the current miss. In fact, LSP incorporates hysteresis so that the decision is affected to some extent also by reference behavior prior to the last miss.

§5.2 Operation Counter Table

While developing an effective lookup table is an important and interesting issue, well understood in the context of branch predictors, it is not the focus of our current efforts. To this end, we use unlimited space to uniquely record each load by both its program counter (PC) and effective address (EA). This should reduce most aliasing.

For hysteresis, the OCT contains a 2-bit saturating counter for each reference. This counter corresponds to

the following states and operations: 00 SL (Strong Load), 01 WL (Weak Load), 10 WSL (Weak Superload), 11 SSL (Strong Superload).

The OCT differs from Tyson [16], which just used parts of the PC, and Johnson's MAT [8], which just used the upper bits of the EA as the index in their lookup tables. Kumar's SHT [9] considered several possible combinations of PC and EA with both infinite and finite storage.

§5.3 Line Size Detector

The role of LSD, the knowledge mechanism of the online algorithm, is to determine whether a particular reference should have been a load or a superload. Our approximation of this question is to ask ''How many of the relatives (lines belonging to the same superline) of the line were referenced between the time the line was loaded and the time the line was evicted?'' Based on this result, the OCT (prediction table) is updated. Generally speaking, if this number is sufficiently large, we estimate that the reference should have been a superload.

We shall define some terms to facilitate the discussion. Let *l* be a cache line. The three other cache lines that would be loaded with *l* on a superload are called *relatives*. Any line that resides in the cache at the same superline of *l*, will be called a *neighbor*. All relatives are neighbors but a neighbor is not necessarily a relative.

Each line *l* tracks itself and its three neighbors. For each of these it associates one of four possible states R,C,H,X. This reference pattern reflects the condition of a line's neighbors since the time the line was brought into the cache. An H is a hit, the result of the neighbor being a relative at the time the line is loaded. An R indicates a reuse, the next miss to that neighbor loaded a relative. The C represents a conflict, the next miss to that neighbor was a load of a non-relative. An X means that there was no subsequent access to that neighbor before the line was evicted from the cache.

When a line *l* is loaded into the cache, its own entry is marked with an R and all its other entries are checked to see if they are related and marked with an H or X as appropriate. Furthermore, the three neighbor lines may update the entry of their patterns corresponding to *l* with either an R or C.

As subsequent references occur, the conditions of the X state are updated. State can only be changed from an X into an R or C. A cache hit as well as a cache miss can change the state. A superload will wipe clean all state. However, only the particular line which caused the superline will be marked with an R.

When a line is evicted from the cache, its reference pattern is used to update its operation counter. If the reference pattern produced by the LSD contains 1 or more H's, the operation counter is decremented. The intuition behind this rule is that while H's do denote spatial locality, it is in some sense too late, as superloading at this point would be partially redundant. If the pattern contains 2 or more C's the counter is decremented. This action deters superloading from occurring when the extra lines brought in would have been replaced before being used. The counter is incremented if it contains 3 or more R's, or 2 R's and 2 X's. These are situations where superloading was or probably would have been profitable.

Table 2 shows the effects of a simplified but illustrative set of references on the LSD and OCT. The references, the decision that was made based on their operation count, the resulting state of the cache, the evicted cache line and the update to its operation counter are all shown in the table.

Ref	OCT(Ref)	Cache State	Eviction	OCT(Eviction) - Action
		Cache Line 1 Cache Line 2 Cache Line 3 Cache Line 4		
--	--	?1 [????] ?2 [????] ?3 [????] ?4 [????]	?? [????]	OCT(??) - Unchanged
A1	WSL (Weak Superload)	A1 [RXXX] A2 [XXXX] A3 [XXXX] A4 [XXXX]	?? [????]	OCT(??) - Unchanged
B2	WL (Weak Load)	A1 [RCXX] B2 [XRXX] A3 [XCXX] A4 [XCXX]	A2 [XXXX]	OCT(A2) - Unchanged
B3	WL	A1 [RCCX] B2 [XRRX] B3 [XHRX] A4 [XCCX]	A3 [XCXX]	OCT(A3) - Unchanged
C1	WL	C1 [RXXX] B2 [CRRX] B3 [CHRX] A4 [CCCX]	A1 [RCCX]	OCT(A1) - Decremented
C2	WL	C1 [RRXX] C2 [HRXX] B3 [CHRX] A4 [CCCX]	B2 [CRRX]	OCT(B2) - Unchanged
C3	WL	C1 [RRRX] C2 [XRRX] C3 [HHRX] A4 [CCCX]	B3 [CHRX]	OCT(B3) - Unchanged
A1	WL	A1 [RXXH] C2 [CRXX] C3 [CHRX] A4 [CCCX]	C1 [RRRX]	OCT(C1) - Incremented

Table 2: Online Algorithm Example. The columns of the table are, the incoming reference, what the OCT (look-up table) says should be done, the state of the cache after the incoming reference has been loaded, the line and pattern that were evicted and finally, how this pattern changes the OCT state. For a given reference, the letter denotes a superline, the number corresponds to the line within a superline. In the example, C1 replaces A1, A1 and B2 are neighbors, and B2 and B3 are relatives.

When a prediction needs to be made, the counter associated with the missing line is checked and it indicates the operation to be performed. Currently, if a reference has any relatives in the cache, a superload will be suppressed. This is due to the observation that superloading is

usually not profitable if 1 or more relatives are already present in the cache in the Opt 2:1 model. The OCT counters are initialized to weak superloads.

The LSD is comparable to Johnson's SLDT [8] and Kumar's AST [9]. The LSD operates at a much finer grain than the SLDT. It restricts its prediction to a binary decision, unlike the AST which predicts a mask of 16 ''lines'' to be loaded in a ''superline''. The LSD also uses four states, R,X,C,H, to record cache behavior, while the SLDT and AST effectively only use R and X.

As with any online prediction scheme, there are many design parameters for the prediction lookup table (OCT) and knowledge mechanism (LSD). Just a few examples are, the initial state of the OCT, the size of the OCT counter, which combination of bits (PC, EA) should index the OCT, which states should the LSD track, which patterns should modify the OCT. Our initial choices for these parameters, as described above, are based largely on common practice in similar settings such as branch prediction. We hope in the future to tune these parameters, using the optimal offline results as a guide.

§5.4 Results of the Online Algorithm

Our online prediction scheme can be quite effective in reducing cache misses when compared to the base case. Typically the performance of the LSP varies between those of the Opt 2:1 and the Opt 3:1 algorithms.

For example, the online algorithm with a line size of 16 bytes in a direct mapped cache of 16K yields a decrease of a cache miss rate between 6% and 30% (average 17%) with an increase in byte reads from -9% to 18% (average 9%) when compared to the base case with a 32-byte line. Recall that on average, the application of the Opt 2:1 algorithm resulted in an average 19% decrease in miss rate with a 43% *decrease* in bytes read. Therefore, the online algorithm can catch 90% of the margin of improvement in decreasing the miss rate but suffers from extra bandwidth requirements.

The miss rate is shown in Figure 5 where it can be seen that the LSP (fourth from the left) is close to the Opt 2:1 (second from the left for each application) for the same line size. In the case of *ijpeg*, LSP actually outperforms Opt 2:1 in miss rate for line sizes 16 and 32 bytes. Note, this is not a contradiction, as Opt 2:1 is a hybrid between optimizing the cache hit ratio (Opt 1:1) and not increasing the number of bytes read (Opt 3:1).

As noted earlier, applications can show large variations in miss rate across different line sizes. The LSP exhibits a similar but less pronounced variation in performance across different line sizes.

The LSP rarely performs worse than always loading or always superloading. Exceptions occur when the LSP uses large line sizes on applications that prefer small line sizes. One example is the application *go*, where the base case beats the LSP for a line size of 32 bytes.

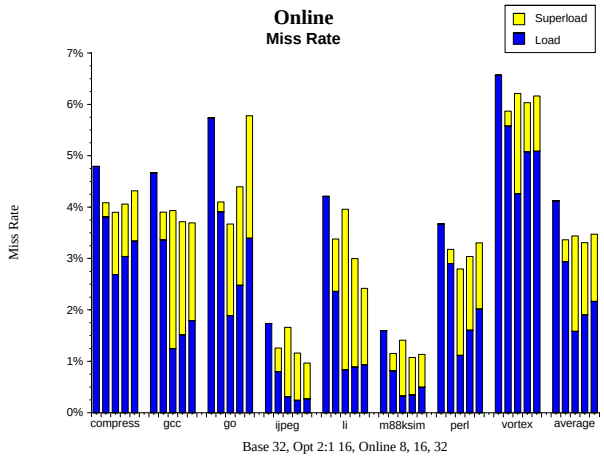

Figure 5: Miss rates for the eight benchmark applications. The LSP algorithm is simulated at line sizes of 8, 16, and 32 bytes. For comparison, the base case (no superloads) and the optimal algorithm Opt 2:1 are simulated at a line size of 16 bytes, the best line size for the optimal algorithms. The cache size is 16K.

Another observation is that the LSP does more superloads than the optimal algorithms. This translates into a significant increase in the number of bytes transferred as shown in Figure 6. However, the largest factor for the number of bytes read still remains the line size. The LSP at an 8-byte line size is roughly equivalent in bytes read to the Opt 2:1 algorithm at a 16-byte line size.

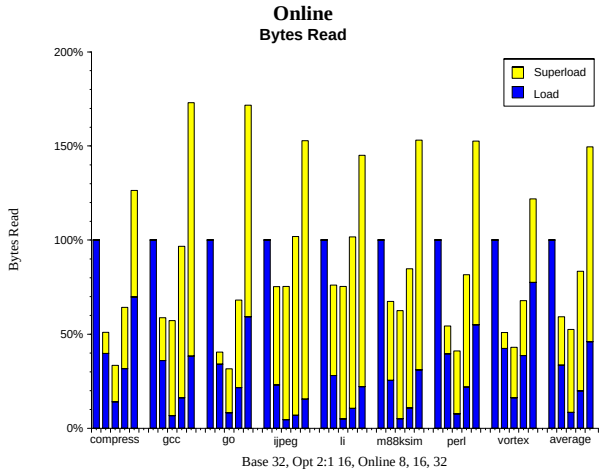

Figure 6: Bytes read for the eight benchmark applications. Figure's parameters are the same as Figure 5.

The number of bytes transferred under the LSP at a 16-byte line size is comparable to the base case, a 32-byte line size. This implies that the additional bandwidth requirements of LSP could easily be negated if its inclusion allows the cache to be designed with smaller line sizes. The increase in superloads also suggests that the LSP's current parameters may be too aggressive.

Even without significant tuning, the LSP shows promise in being able to obtain the benefits of superloading.

The algorithm for the LSP clearly demonstrates that it is possible to create a knowledge mechanism for determining line size that will provide good feedback to an online predictor.

§6 Profiling

Our statistical analysis of the optimal sequences shows that there is some positive correlation between the load type and its PC (Program Counter). While this correlation isn't as strong as between the load type and the EA (Effective Address), trying to identify load type by the PC does allow us to take advantage of static techniques such as profiling.

The inherent advantage of profiling is that little hardware support is required. All that is needed is the ability of the hardware to perform loads or superloads and an extra bit in the load instruction, or an extra opcode, to indicate the appropriate action. The disadvantage of all profiling techniques is performing offline analysis. In our case this means running the offline optimal algorithm on a trace of references.

We've simulated profiling by grouping unique PC's together in a hash table and keeping counters for the number of times each PC performed a load or a superload. We then run the program through a modified cache simulator, which looks up the PC of each miss in the hash table and decides whether to perform a load or a superload. Our experimental results show that a threshold of 60% for superloading is best. That is to say, a superload instruction will be generated when less than 60% of the misses for a given PC were loads in the optimal algorithm.

For our experiments we profiled the optimal 1:1 sequence of loads and superloads. Higher cost model ratios actually have more accurate and consistent profiling information, but they produce fewer superloads and hence have less impact overall. All results presented in this and the next section are for the full simulations of the benchmarks as described in Section 3.

§6.1 Results of Profiling

We choose the 16-byte line size for the online and profiled simulations, as this is the best overall line size for superloading. As a comparison we use two base cases, the base case from the previous sections, all loads at 32 bytes, which has the best overall cache hit ratio, and loads at a line size of 16 bytes, which provides a reference by which to examine the increase in the number of bytes read.

Figure 7 shows the miss ratios for our experiments. In most cases, the online algorithm performs slightly better than profiling. With the exception of *m88ksim* for the 16-byte line, the online and profiling do better than either base line size. On average, the online algorithm reduces the miss rate by 10%, whereas profiling reduces it by 7% (compared to the 32-byte line base).

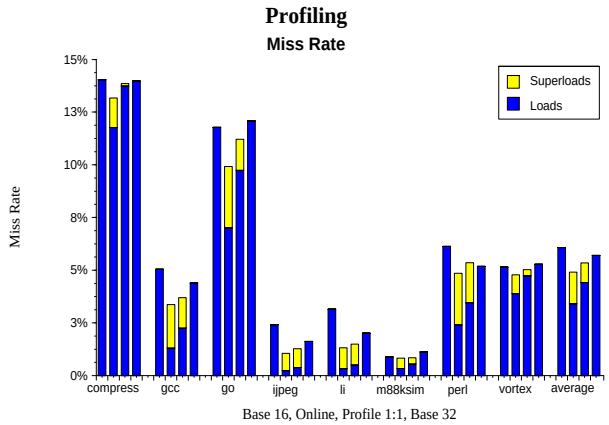

Figure 7: Miss rates for Base 16, Online, Profiling 1:1 and Base 32. Base 16 is all loads with a 16-byte line, Online is the online algorithm described in Section 5 with a 16-byte line, Profiling 1:1 is the profiled optimal 1:1 sequence with a 16-byte line and Base 32 is all loads with a 32-byte line.

In Figure 8, we examine the number of bytes read by each scheme. With few exceptions, the online algorithm is the most aggressive and reads in the most bytes. As to be expected, there is quite a large difference in the number of bytes read between the two base cache-line sizes. Profiling makes more efficient use of its superloads in many cases and reads 22% fewer bytes than the 32-byte line base case.

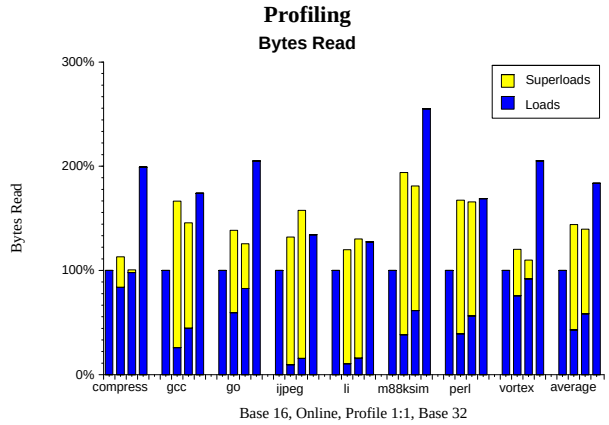

Figure 8: Bytes read for profiling. Each benchmark is normalized to the Base 16 case. Parameters are the same as Figure 7.

Overall, the results for profiling are encouraging. It provides most of the cache hit ratio of the online algorithm, while being reasonable in terms of bytes read. Compared to the 32-byte line base, there is a reduction in both cache hit ratio and bytes read.

§7 Prefetching

Dynamic cache lines are a method for localized prefetching. Thus comparison to other prefetching schemes is warranted. ''Next-line'' prefetching [14] is a

very simple yet effective scheme. With next-line prefetching, on a cache miss to line *l*, line *l+1*, if not already present in the cache, will be prefetched.

We examined next-line prefetching added to the base cache mechanism for both 16 and 32-byte lines. On average prefetching improves the miss rate 5% and 1% respectively. Using a 16-byte line, profiling and the online mechanism show a 19% and 12% respective improvement in the average miss rate.

Increases in the number of bytes read is more dramatic. On average, basic next-line prefetching presents increases of 64% and 70% for 16-byte lines and 32-byte lines, whereas online and profiling superloading result in an average increase of 44% and 39% and respectively.

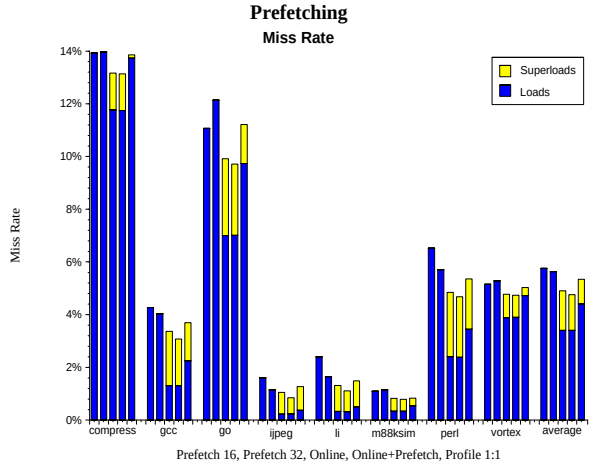

Figure 9: Miss rates for next-line prefetching schemes. Prefetch 16 and Prefetch 32 are the 16 and 32-byte base case with next-line prefetching. The superloading variations use a 16-byte line.

Next-line prefetching can also be combined with dynamic cache line sizes. We experimented with several approaches, but found the most effective was on a cache miss that was going to be superloaded, if the first line of the next superline was not already in the cache and that line would be superloaded, prefetch the next superline into the cache.

The combination of the online mechanism and next-line prefetching shows a 21% improvement in the cache miss rate, a 2% improvement over the online mechanism. In terms of the number of bytes read, the combination results in an average increase of 51% over the base case and 7% over the online mechanism.

The addition of next-line prefetching to both the base case and the online mechanism provides a comparable decrease miss rate. However, the online mechanism allows the prefetching to be more selective resulting in a relatively small increase in the number of bytes read.

Both in terms of miss rate and bytes read, dynamic cache line schemes have the potential to out perform the simple though unintelligent next-line prefetching. There is also merit for using these two techniques in conjunction.

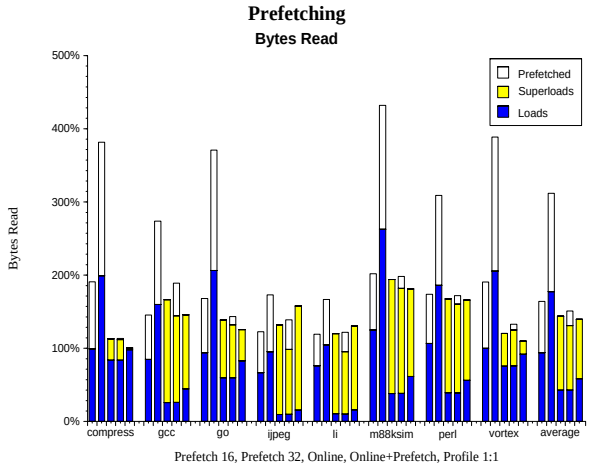

Figure 10: Bytes read for prefetching. This figure's parameters are the same as Figure 9.

§8 Conclusions and Future Work

We have introduced offline algorithms for determining the optimal sequence of loads and superloads for a given reference stream for direct-mapped caches under various cost models. We have presented the experimental results of these algorithms applied over a range of cache parameters for a set of traces based on the Spec95 integer benchmarks. Each optimal algorithm incorporates a specific trade-off between cache miss rate and the number of bytes read into the cache. In many cases, optimal superloading can noticeably reduce miss rate when compared to the base case without appreciably increasing bandwidth. In other cases, superloading can achieve a comparable miss rate with smaller line sizes, translating into a substantial reduction in bandwidth.

We have also presented an online algorithm for determining the sequence of loads and superloads. Experimental results for this algorithm, compared to the optimal algorithm, indicate that comparable improvements in cache miss rate can be achieved, although there is a noticeable increase in the number of bytes read in some cases. This suggests that further refinement, using the knowledge gained from analyzing optimal sequences, could improve the performance of the online algorithm.

We have examined the effectiveness of static analysis of the optimal traces (profiling). With our application suite, profiling seems to be fairly effective, nearly achieving the same miss rate reduction as our online prediction algorithm, while being more efficient in terms of the number of bytes transferred.

Finally, we compared superloading and next-line prefetching. Profiling and the online mechanism for superloading perform better than next-line prefetching in miss rate and significantly better in the number of bytes read. Furthermore, it appears that superloading and next-line prefetching can be successfully combined.

The algorithms that we have currently developed will allow us to characterize other features of optimal superloading. In future work, we expect to analyze other line/superline size ratios, loading arbitrary patterns of lines (cf. [15] and [9]) within a superline, and similar variations. The development of corresponding algorithms for set-associative caches is another interesting area for future research.

The results of these algorithms can be used to further develop software and hardware techniques that exploit spatial locality. Combining profiling and online prediction is one promising idea. And perhaps most significant, optimal results allow the construction of a framework for online methods which can lead to the determination of better parameters for lookup tables and knowledge mechanisms.

§9 Bibliography

[1] Eric Anderson, Peter Van Vleet, Lindsay Brown, Jean-Loup Baer and Anna Karlin, ‘‘On the Performance Potential of Dynamic Cache Lines’’, Technical Report UW-CSE-99-02-01, University of Washington, February, 1999.

[2] Laszlo Belady, "A Study of Replacement Algorithms for a Virtual-Storage Computer," in IBM Systems Journal, volume 5, number 2, pages 78-101, 1966.

[3] David Burger and Todd Austin, "Evaluating Future Microprocessors: the SimpleScalar Tool Set," Technical Report #1342, University of Wisconsin, June 1997.

[4] Charles Conti, ‘‘537r Buffer Storage’’, Computer Group News, 2:9-13, 1969.

[5] Patrick Crowley and Jean-Loup Baer ‘‘On the Use of Trace Sampling for Architectural Studies of Desktop Applications’’, *Proceedings of the first Workshop on Workload Characterization (WWC ’98)*, November 1998.

[6] Antonio González, Carlos Aliagas, and Mateo Valero, "A Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality," in *Proceedings of ICS ’95*, July 1995.

[7] Teresa Johnson and Wen-mei Hwu, "Run-time Adaptive Cache Hierarchy Management via Reference Analysis," in *Proceedings of the 24th International Symposium on Computer Architecture*, pages 315-326, June 1997.

[8] Teresa Johnson, Matthew Merten and Wen-mei Hwu, "Run-time Spatial Locality Detection and Optimization," in *Proceedings of the 30th International Symposium on Microarchitecture*, pages 57-64, 1997.

[9] Sanjeev Kumar and Chistopher Wilkerson, "Exploting Spatial Locality in Data Caches using Spatial Footprints," in *Proceedings of the 25th International Symposium on Computer Architecture*, pages 357-368, 1998.

[10] Madusudhan Talluri and Mark Hill, ‘‘Surpassing the TLB Performance of Superpages with less Operating System Support’’, in *Proceedings of ASPLOS-VI*, pages 171-182, 1994.

[11] Jude Rivers, Edward Tam, Gary Tyson and Edward Davidson, "Utilizing Reuse Information in Data Cache Management," in *Proceedings of the 12th ACM International Conference on Supercomputing*, pages 449-456, July 1998.

[12] Ted Romer, Wayne Ohlrich, Anna Karlin and Brian Bershad, ‘‘Reducing TLB and Memory Overhead using Online Superpage Promotion’’, in *Proceedings of the 22nd International Conference on Computer Architecture*, pages 176-187, 1995.

[13] André Seznec, "Decoupled Sectored Caches: Conciliating Low Tag Implementation Cost and Low Miss Ratio," in *Proceedings of the 21st International Symposium on Computer Architecture*, pages 384-393, 1994.

[14] Alan J. Smith, "Cache Memories," *Computing Surveys*, vol 14, pages 473-530, September 1982.

[15] Olivier Temam, "Investigating Optimal Local Memory Performance," in *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[16] Gary Tyson, Michael Farrens, J. Matthews, and A. R. Plezkun, "A Modified Approach to Data Cache Management," in *Proceedings of the 28th Annual International Symposium on Microarchitectures*, pages 93-103, December 1995.