# On the Performance Potential of Dynamic Cache Line Sizes

**Eric Anderson, Peter Van Vleet, Lindsay Brown,**
**Jean-Loup Baer and Anna R. Karlin**

{eric, pvv, lbrown, baer, karlin}@cs.washington.edu

**Technical Report UW-CSE-99-02-01**
**February, 1999**

**Department of Computer Science and Engineering**
**University of Washington**
**Box 352350**
**Seattle, WA 98195**

**Abstract**

In this paper we present offline algorithms for determining the optimal sequence of loads, superloads and bypasses 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. We also present an online algorithm for determining the sequence of loads and superloads. This algorithm operates by monitoring the reuse and conflicts of cache lines. Experimental results show comparable improvements to 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. All modern microprocessors have on-chip, or level one, instruction and data caches and a large majority of these level one caches are backed up by second level caches. Caches have been successful because programs exhibit locality: spatial locality, i.e., the tendency for neighboring memory locations to be referenced close together in time, and temporal locality which is 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.

Many techniques, either hardware or software oriented or both, 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, the cache controller can be implemented such 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. The cache controller can also be directed not to load the line in the cache at all, i.e., *bypass* the cache and bring the data directly into a register. Note that the advantages of *superloading* and *bypassing* depend on the cost model for the level of the memory hierarchy under investigation. Of particular importance are the relative costs of a load, a bypass, 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. The contributions of this paper are to: (1) present optimal offline algorithms for these two techniques in the case of direct-mapped caches for cost models that provide tradeoffs between the reduction of cache misses and memory bandwidth, (2) empirically evaluate how

much could be gained if one were able to use the optimal algorithms rather than the base case of cache operation, i.e., provide the margin of improvement, and (3) analyze the performance of an online algorithm that monitors the reuse and conflicts of cached lines when compared to the optimal and to the base cases.

The rest of the paper is as follows. In section 2, we present the motivation for this study, our basic terminology, the cost functions that the optimal algorithms will minimize, and a brief summary of related work. Section 3 is devoted to the description of the optimal algorithms for superloading and bypassing. Section 4 describes the proposed online algorithm. The methodology that we used for simulating the performance of the various algorithms is given in Section 5. Results of the simulations, and their analysis, are given in Section 6. Section 7 concludes and suggests further study.

# 2 Motivation

## 2.1 Terminology and cost model

There exists a large body of work devoted to the analysis of the numerous trade-offs involved in the design of efficient caches [**?**]. For a given real estate, i.e., cache capacity, and target cycle time, i.e., access time to the on-chip cache, one has to decide on line size, set-associativity, and various latency tolerance techniques. Under the simplifying assumption of a cache backed-up by main memory, the main metrics that relate to the contribution of the memory hierarchy to the execution time of a program are the cache miss rate and the time to transfer a line from memory to cache, i.e., factors closely related to latency and bandwidth.

It is well known [**?**] that for a given application and set-associativity, there exists an optimal cache line that will minimize the cache miss rate. This optimal cache line size depends not only on the cache size but also on the application. Applications that have a fair amount of spatial locality will favor large lines while others will behave better with smaller lines. In this paper we are presenting algorithms that, on a cache miss to line $a_i$ of size $l$, determine the choice between loading $a_i$ or loading the *superline* of size $L$ which consists of $a_i$ and its $(L/l) - 1$ *relatives*, i.e., those lines that would have been loaded on a cache miss to $a_i$ if the cache had a line size of $L$ (in the remainder of this paper, we use $l = 1, L = 4$). A line residing in the cache at the same superline as $a_i$ which is not a relative of $a_i$ will be called a *neighbor* of $a_i$. In offline algorithms we will also consider the possibility of *bypassing*, i.e., on a cache miss we bring directly the missing data to a register without modifying the contents of the cache.

We use a simplified model of memory access that offers a tradeoff between bandwidth and latency effects, while remaining tractable. We assign a cost to each cache miss with the cost of a superload being a multiple of the cost of a load and ignore the cost of cache hits since they do not contribute more to the execution time than any other instruction. In the case

where we decide that the cost of a superload is the same as the cost of a load (denoted 1:1, i.e., we normalize to the cost of a load) the bandwidth effect of superloads is effectively ignored; in the optimal case, minimizing the cost is equivalent to minimizing the miss rate. At the other extreme, a cost of $L/l : 1$ that assigns to a superload a cost equal to the number of lines in the superline corresponds to focusing solely on bandwidth effects. With our choice of $L$ and $l$, we will consider costs of 1:1, 2:1,and 3:1 with the increase in the ratio indicating that bandwidth effects are taken more in consideration.

We will use the following notation. The execution of a program is represented by a stream $\sigma = \sigma_1, \ldots, \sigma_n$ of memory references. A cache hit is satisfied at no cost; a cache miss can be satisfied either by a cache *load* of a single line, at cost $C_1$, or a cache *superload* of $k$ consecutive lines that comprise a single superline, at cost $C_2$. The optimal offline strategy described in the next section minimizes the aggregate cost of satisfying cache misses through a combination of loads and superloads, taking the entire reference stream $\sigma$ into account. For simplicity, we assume that the cost of a bypass is also $C_1$, the same as the cost of a miss, and that there is no possible reuse of bypassed references through a bypass buffer.

### 2.2 Related work

Belady's MIN algorithm is the most well-known offline algorithm for the study of memory hierarchies [**?**]. Developed in the context of paging systems, MIN gives the minimum number of page faults for a given program. Until recently, efficient implementations of MIN required two passes over the input string. A good example is the OPT stack algorithm [**?**]. By using limited look-ahead windows and correcting the contents of the stack when necessary, a "one-pass" optimal algorithm can be devised for fully associative and set-associative caches [**?**].

Heuristics and hardware mechanisms to explore the possibility of using dynamic page sizes (page and superpage with a size a power of two of the page size) for the interface between main and secondary memories, and of having lines and superlines for the cache - main memory interface have been investigated by a number of authors. For example, page promotion policies from a page to a superpage have been proposed with the goal of either facilitating superpage management [**?**] or to have better TLB coverage [**?**].

Closer to our study is the work of Johnson et al. [**?**, **?**] that investigates hardware assists, a Spatial Locality Detection Table (SLT) and a Memory Address Table (MAT), for dynamic fetch size choices. The goals of our study differ from theirs in two ways. First, we are interested in how much the superline concept can improve performance in the optimal case and thus we restrict ourselves, at this time, to the study of direct-mapped caches. Second, our online algorithm focuses on the tradeoff between cache misses and bytes transferred between memory and cache while, because the machine simulated in [**?**, **?**] has ample bandwidth, their emphasis is principally on the reduction of cache misses.

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 [**?**], or at having two data caches one of which is devoted especially for those data that exhibit spatial locality[**?**, **?**]. Of course the SLT and MAT schemes cited previously also monitor the same type of information.

Loading and superloading start from the premise that one should look at multiples of the default line size. Another approach is to start with a very large line size and be able to load only sub-lines when it appears that this is efficient. The first cache implementation, named sector cache [**?**], did in fact use this principle. Predictors for sub-lines have been studied in [**?**] and several schemes for sub-line invalidation in the context of cache coherent multiprocessors have been proposed [**?**, **?**].

# 3 Algorithms for computing the optimal costs

## 3.1 Description of the algorithm

We note first that the optimal cost of satisfying a sequence in the absence of superloads is, for a direct-mapped cache without bypassing, immediately determined by direct simulation. For a direct-mapped cache with bypassing, the optimal cost is readily determined by adapting the $MIN$ algorithm of Belady[**?**], here phrased as

> On a miss to block $b$, conflicting with $a$ currently in the cache, fetch $b$; replace $a$ if $b$ will next be referenced before $a$ will, and bypass otherwise.

For superloads, the optimal algorithm is more complex. Interestingly, straightforward offline strategies such as majority voting (superload if the majority of the relatives will be next referenced before neighbors mapping to the same location) fail to be optimal, and relatively simple counterexamples suffice to demonstrate this [**?**]. For our optimal algorithm, we first decompose the optimal cost determination into separate computations for each superline, adding each of them together at the end to reconstruct the cost for the entire cache. In the discussion below, we abuse notation and use $\sigma$ to denote just those references that map to some particular superline in the cache. For the rest of this section, we restrict our attention to computing the optimal cost on just these references.

We use the following notation:

- $Opt(j)$: the optimal cost of satisfying the reference stream $\sigma$ up through and including reference $\sigma_j$;
- $P(j)$: the optimal cost of satisfying the reference stream $\sigma$ through and including $\sigma_j$, *assuming a superload at time* $j$;

- $c(i, j)$: the optimal cost of satisfying references $\sigma_{i+1}, \ldots, \sigma_j$, assuming a superload on $\sigma_i$, and loads or bypasses on all remaining references (computed as discussed in the first paragraph of this section);
- $\tilde{c}(i, j)$: the optimal cost of satisfying references $\sigma_{i+1}, \ldots, \sigma_{j-1}$, assuming a superload on $\sigma_i$, loads or bypasses on all references up to the $(j-1)$st, and that a superload on $\sigma_j$ *is possible*. If the superload is not possible, $\tilde{c}(i, j) = \infty$. (What it means for the superload to be possible is discussed below.)

We note that once an algorithm performs a superload, the contents of that superline are entirely replaced. Thus any optimal sequence of loads and superloads that ends in such a superload can be substituted for any other. This observation is the foundation for a dynamic programming approach, which results in a simple and effective algorithm for superloads. In particular, we observe that in any optimal sequence for satisfying all of $\sigma$ there is one last superload time, that the superline at that time is completely determined by the reference that is superloaded, and that – by the dynamic programming observation above – the optimal cost can be obtained by minimizing over all such last superload times. Hence, we have

$$Opt(n) = \min_{j<n}(P(j) + c(j, n)). \quad (1)$$

Similarly, for the computation of $P(j)$, we observe that

$$P(j) = \min_{i<j}(P(i) + \tilde{c}(i, j) + C_2). \quad (2)$$

We now discuss the computation of $\tilde{c}(i, j)$. For this, we distinguish between three cases, based on the hardware design:

**Case 1:** Superloads can be performed regardless of the state of the superline:

In this case $\tilde{c}(i, j) = c(i, j)$.

While this case is easy to analyze, superloading in the case of a cache hit is in effect a form of prefetching, and outside the scope of this paper.

**Case 2:** Loads and superloads can be performed only on a miss (and no bypassing is allowed):

In this case, it is possible that the superline state resulting from a superload at time $i$, together with the simulation from $i + 1$ to $j - 1$, would lead to a hit at time $j$ and hence a superload at time $j$ is not possible. In this case, we compute $\tilde{c}(i, j) = c(i, j - 1)$ if the result would lead to a miss at time $j$ and set $\tilde{c}(i, j) = \infty$ otherwise.

**Case 3:** Loads, bypasses or superloads can be performed, but only on a miss:

In this case, a slightly more complex computation is required for $\tilde{c}(i,j)$. First, note that the costs $\tilde{c}$ can for this purpose be computed individually for each line $m$ within the given superline, $\tilde{c}(i,j) = \sum_m \tilde{c}(i,j;m)$, since the only effect one line's reference stream has on another arises through superloads, and they occur here only at times $i$ and (perhaps) $j$.

Next suppose, without loss of generality, that $\sigma_j = b$, and that $b$ maps to line $m$ in the superline. A superload at time $j$ is possible only if some line $c \neq b$ mapping to line $m$ is referenced at some time between $i+1$ and $j-1$ inclusive or, failing that, if the superline referenced at time $i$ does not contain $b$.

Let $k$ be the index of the last reference between $i$ and $j$ such that $\sigma_k$ maps to line $m$ and $\sigma_k \neq b$ ($k$ could be $i$ if $\sigma_i$ is not a relative of $b$). Then all references after index $k$ and up to index $j$ that map to line $m$ are to $b$. In order for $\sigma_j$ to be a miss, the contents of line $m$ after the reference to $\sigma_k$ must not be $b$, and all subsequent references after $\sigma_k$ that map to line $m$ (all references to $b$) must be misses – any other possibility results in a hit on $\sigma_j$. We claim that this implies that

$$\tilde{c}(i,j;m) = c(i,k;m) + rC_1,$$

where $r$ is the number of references between $k$ and $j$ that map to $m$. The validity of the equation is straightforward in the case that during the computation of $c(i,k;m)$, the reference $\sigma_k$ was a hit. In this case, in order for $\sigma_j$ to be a miss, all subsequent references to $b$ must be bypassed (at a cost of $C_1$ each), and the equation follows. On the other hand, if reference $\sigma_k$ was a miss, performing a load on this reference (instead of a possible bypass) does not increase the cost $c(i,k;m)$ of servicing this part of the sequence, and is the only way (with bypasses on subsequent references to $b$) to end up in the state where $\sigma_j$ is a miss and hence can be superloaded. If there is no such $k \geq i$, $\tilde{c}(i,j;m)$ is set to $\infty$.

All of the given recurrence relations (for $Opt$, $P$, $\tilde{c}(i,j)$ and $c(i,j)$) can be readily converted into dynamic programming computations.

### 3.2 Implementation issues

At each $j$, the space requirement for computing $P(j)$ in terms of $P(0), P(1), \ldots, P(j-1)$ is no worse than linear in $j$, hence $O(n)$ in all. To analyze time complexity, we note that the values $c(i,j)$ and $\tilde{c}(i,j)$ can be updated in constant time from $c(i,j-1)$ and $\tilde{c}(i,j-1)$. Thus the time bound is worst-case quadratic in $n$ (corresponding to a linear number of updates at each reference $\sigma_j$).

One observes that not all values of $P(i)$ need be maintained at each step. If for any $i_1, i_2$ we have $P(i_1) + c(i_1,j) < P(i_2) + c(i_2,j)$, we know that $Opt(n)$ need not take $i_2, P(i_2)$ into account; and similarly, if $P(i_1) + \tilde{c}(i_1,j) < P(i_2) + \tilde{c}(i_2,j)$, the computation of $P(j)$ need not take $i_2$ into account. We observe that if the simulated superlines are identical, and $c(i_1,j) \leq c(i_2,j)$, then not only $P(j)$ but $P(k)$, $k > j$ need not take $i_2$ into account. By maintaining a list of superline states, we can eliminate all but a few from consideration as

| Application | Number of references | Maximum pst |
|---|---|---|
| compress | 7.5 | 18 |
| gcc | 20.0 | 31 |
| go | 20.0 | 15 |
| ijpeg | 20.0 | 9 |
| li | 20.0 | 14 |
| m88ksim | 23.2 | 12 |
| perl | 11.6 | 18 |
| vortex | 20 | 41 |

Table 1: Maximum number of previous superload times (pst) considered in the calculation of $P(j)$ (see text). Number of references in millions. Cache size 16K, line size 32 bytes.

previous superload times. In practice, this optimization is hugely successful; the maximum number of potential previous superload times at any point in the algorithm is small, as compared to the number of references. By way of illustration, we measured the maximum number of previous superload times considered in a simulation of our sample workload at a single cache configuration; the results are displayed in Table **??**.

### 3.3 An alternative algorithm for superloads without bypassing.

We have also implemented a linear time, constant space algorithm for computing the optimal cost in the case of superloads without bypassing. 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 configurations, 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. In the case of superloads without bypassing, 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. This method readily generalizes to $n-$way set-associative caches that follow LRU replacement, though the value of the constant grows rapidly with $n$.

## 4 Online algorithm

Moving from offline to online algorithms presents the significant restriction that we can no longer use future information when making a decision. Instead, 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. This moves us into the area of hardware predictors.

## 4.1 Predictor Anatomy

Fundamentally, the effectiveness of any 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 will further inhibit this effectiveness.

Limited knowledge is a problem inherent in line size prediction. Specifically, after a prediction is made, one must provide "correct" feedback, which is incorporated in the history information of the predictor. In the case of branch prediction, this is a simple matter, as the result of the branch is known within a few cycles, and more importantly, is absolutely correct. Since optimal line size prediction requires future knowledge, providing absolutely correct feedback online is not possible. Instead, the predictor must rely on some form of independent knowledge mechanism, which evaluates the situation and provides (imperfect) feedback.

Limited space in which to store information introduces aliasing effects, which can distort the perception of the past. Aliasing can have a significant effect on prediction performance, usually negative.

Thus 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.

## 4.2 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 harmful 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 [?], which just used parts of the PC, and Johnson's MAT [?], which just used the upper bits of the EA as the index in their lookup tables. Kumar's SHT [?] considered several possible combinations of PC and EA with both infinite and finite storage.

### 4.3 Line Size Detector

For the line size predictor, our knowledge mechanism will be called the Line Size Detector (LSD). Its role 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 neighbors 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.

Each line 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 miss to that neighbor before the line was evicted from the cache.

When a line 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, all other lines may update the appropriate entry of their patterns with either an R or an 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.

The following table 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. 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.

| Ref | OCT (Ref) | Cache State 1 | 2 | 3 | 4 | Eviction | OCT(Eviction) |
|---|---|---|---|---|---|---|---|
| – | – | ?1 [????] | ?2 [????] | ?3 [????] | ?4 [????] | ?? [????] | OCT(??) unch |
| A1 | WSL | A1 [RXXX] | A2 [XXXX] | A3 [XXXX] | A4 [XXXX] | ?? [????] | OCT(??) unch |
| B2 | WL | A1 [RCXX] | B2 [XRXX] | A3 [XCXX] | A4 [XCXX] | A2 [XXXX] | OCT(A2) unch |
| B3 | WL | A1 [RCCX] | B2 [XRRX] | B3 [XHRX] | A4 [XCCX] | A3 [XXXX] | OCT(A3) unch |
| C1 | WL | C1 [RXXX] | B2 [CRRX] | B3 [CHRX] | A4 [CCCX] | A1 [RCCX] | OCT(A1) decr |
| C2 | WL | C1 [RRXX] | C2 [HRXX] | B3 [CHRX] | A4 [CCCX] | B2 [CRRX] | OCT(B2) unch |
| C3 | WL | C1 [RRRX] | C2 [XRRX] | C3 [HHRX] | A4 [CCCX] | B3 [CHRX] | OCT(B3) unch |
| A1 | WL | A1 [RXXH] | C2 [CRXX] | B3 [CHRX] | A4 [CCCX] | C1 [RRRX] | OCT(C1) incr |

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, 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 [**?**] and Kumar's AST [**?**]. 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. Having the optimal offline results presents the novel ability to highly tune these structures. However, as we are just beginning this process, we have used default settings, based on common practice in similar settings, such as branch prediction.

# 5 Methodology

To evaluate the effectiveness our optimal algorithms, we used all eight of the Spec95 integer benchmarks. Traces were collected with the SimpleScalar simulator [**?**]. The default setting for the simulator, an 64-bit 256 register RISC machine, was used. The binaries for the benchmarks were those provided with SimpleScalar simulator.

Our traces include only reads. To accurately capture the behavior of first level caches of the size used in our study, it is our observation that a trace of several million reads is sufficient. Therefore, we collected traces over 5 million reads in length but less than 25 million.

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 from the *train* input set).

124.m88ksim, 129.compress and 134.perl were not sampled. They contained 7.5M, 11.6M and 23.2M reads respectively. Others were sampled with 5 separate sections of 4 million consecutive reads. The 5 sections were uniformly spaced through out the trace and then concatenated together to form one 20 million read trace.

For this study, we consider direct-mapped caches of sizes 8K, 16K, and 32K. We also consider line sizes of 8, 16, 32, and 64 bytes. Throughout the remainder of the paper, each reference to a line size implicitly assumes a superline size of four times the line size.

The optimal algorithms have good performance, allowing us to run approximately 500 cache simulations for this study. On a 200 MHz Pentium Pro machine, the algorithm described in section **??** takes about 30 minutes to run on a 20 million read trace. The algorithm in section **??** takes about 2 minutes to run on a 20 million read trace. Both algorithms exhibit large amounts of coarse grain parallelism, as each superline in the cache can be optimized independently.

# 6 Results

## 6.1 Optimal algorithms

This section describes our observations from applying the optimal algorithm described in section **??** to the benchmark application suite described in section **??**.

### 6.1.1 Evaluation of optimal algorithms

As discussed earlier, there are two metrics we use to evaluate superloading algorithms: miss rate, as a measure of latency, and bytes transferred, as a measure of bus bandwidth utilized. By setting the cost parameters associated with the optimal algorithms, we can choose to place different degrees of importance on these two metrics. To minimize bytes transferred, the optimal strategy for all applications is to use the smallest possible line size and perform no superloading. To minimize miss rate, the appropriate line size and extent of superloading depend on the spatial and temporal locality of the application [**?**]. We examine a range of optimal algorithms each optimizing within a specific tradeoff between latency and bandwidth.

The Opt 1:1 algorithm minimizes the miss rate, defined as the sum of the number of loads and the number of superloads. The tie-breaking mechanism we use in all optimal algorithms favors loads over superloads. Hence, Opt 1:1 will never superload unless at least one line in the superline other than the missed line will be referenced before its next eviction. The Opt 2:1 algorithm minimizes the sum of $L + 2SL$, where $L$ is the number of loads performed and $SL$ is the number of superloads performed. Since the corresponding number of misses

| Application | Miss rate 8 | 16 | 32 | 64 | Bytes read 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 2: Miss rates and bytes read 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).

is $L + SL$ and the corresponding number of lines transferred is $L + 4SL$, this is equivalent to placing twice as much importance on minimizing miss rate as on minimizing bandwidth ($L + 2SL = (2/3)(L + SL) + 1/3(L + 4SL)$). As above, Opt 2:1 never superloads unless at least two lines in the superline other than the missed line will be referenced prior to their next eviction. Finally, Opt 3:1 places greater importance on minimizing bandwidth by minimizing the sum $L + 3SL$. Opt 3:1 never superloads unless all lines in the superline will be referenced prior to their eviction.

It is not difficult to verify that there is no need to consider fractional values for superload cost – all optimal algorithms with $C_2$ fractional behave identically to the optimal algorithm with $C_2$ rounded down to the closest integer.

### 6.1.2 Workload characteristics

We begin by describing the cache characteristics of the benchmark suite, examining the operation of the "base case" no-superload algorithm on each of the eight applications. We illustrate the results in Table **??**. As mentioned above, the best line size to minimize bytes read is always the smallest line size, and the best line size to minimize miss rate varies across applications. For this benchmark suite, the average application reaches a minimum miss rate at a line size of 32 bytes for a 16K cache. At the extremes, the application *go*, with poor spatial locality, has a lower miss rate at the smallest line size, 8 bytes; while the application *ijpeg*, with strong spatial locality, fares dramatically better as the line size is increased over this entire range.

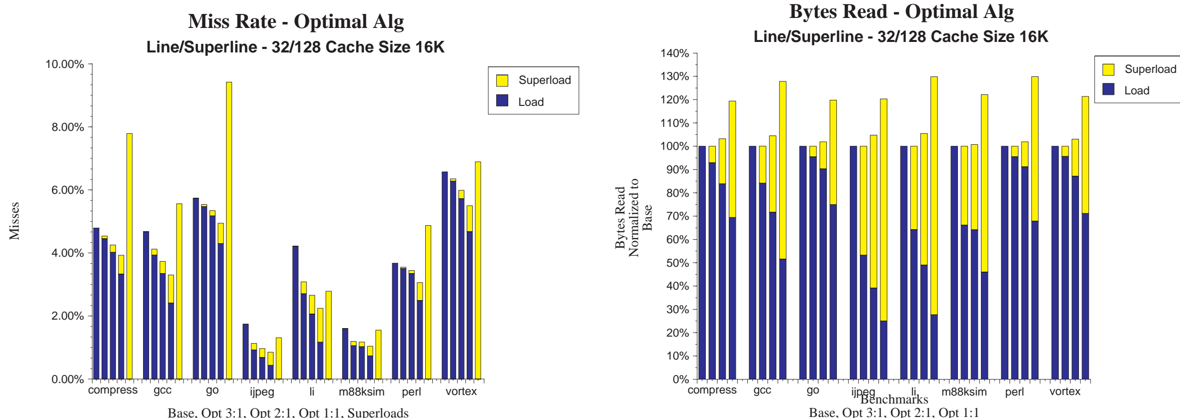


Figure 1: Miss rates and bytes read for the eight benchmark applications. The base case (no superloads), the three optimal algorithms Opt 3:1, Opt 2:1, and Opt 1:1 (see text), and the algorithm that performs a superload at every miss are each simulated at a cache size of 16K and a line size of 32 bytes. For each application, the figure for bytes read for each algorithm is normalized to the respective base case (set to 100). The "superloads" algorithm is not displayed in the bytes read graph; its values are much higher than the others, and would distort the scale.

### 6.1.3 Results of applying the optimal algorithms

We analyze each application, 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 this benchmark suite. The resulting miss rates are shown in the left-hand graph of Figure **??**. In all cases except Opt 3:1 for *li*, 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 $C_2$ the superload cost used by the optimal algorithm decreases, and (b) the percentage of misses on which a superload is performed increases as $C_2$ decreases.

We also see that the proportion of superloads in the optimal algorithms is generally greatest for those applications, such as *ijpeg*, 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.

We also see 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

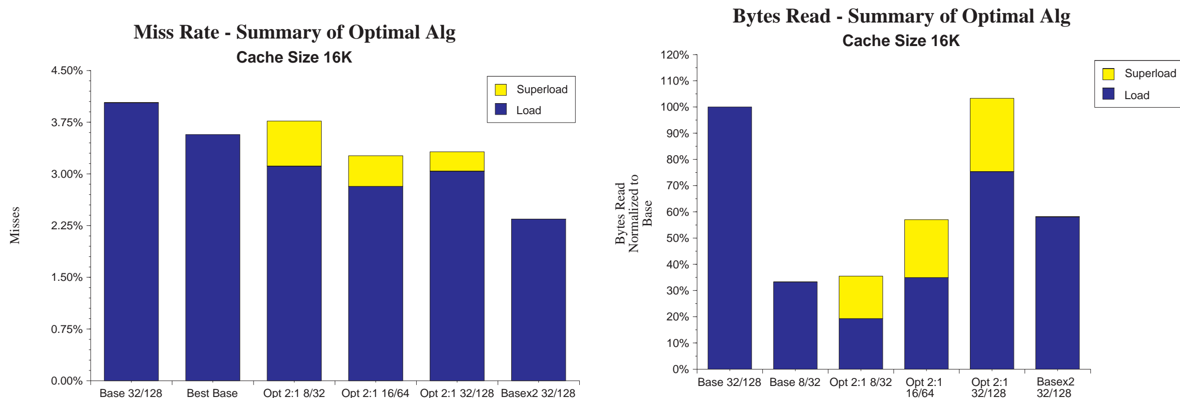


Figure 2: The miss rate and bytes read for the algorithm Opt 2:1 over the range of line sizes of interest. Cache size is 16K. The base case (no superloads) is presented in three versions: at the best overall line size for miss rate (32 bytes) and bytes read (8 bytes), respectively; at the best individual line sizes for each application; and at a cache size twice as large (32K, 32 byte line size). Measurements of bytes read are normalized to the 32-byte base case. Results for the individual applications are averaged to produce a combined figure.

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 **??**, right-hand graph, left two columns), but can lead to a substantial decrease in miss rate (*li* or *ijpeg*, for example, in Figure **??**, left-hand graph).

In the right-hand graph of Figure **??**, 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.) Each quantity in the table is normalized to the corresponding base case for that application (set to 100). We see from the graph that (as noted above) the Opt 3:1 algorithm uses the same bandwidth, the Opt 2:1 algorithm uses somewhat greater bandwidth, and the Opt 1:1 algorithm considerably greater bandwidth than the corresponding base case. The considerable 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.

In Figure **??** (left-hand graph), 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, at the best overall line size (32 bytes) and the best per-application line size, and at a cache size 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. The right-hand graph of Figure **??** presents the corresponding results for bytes read.

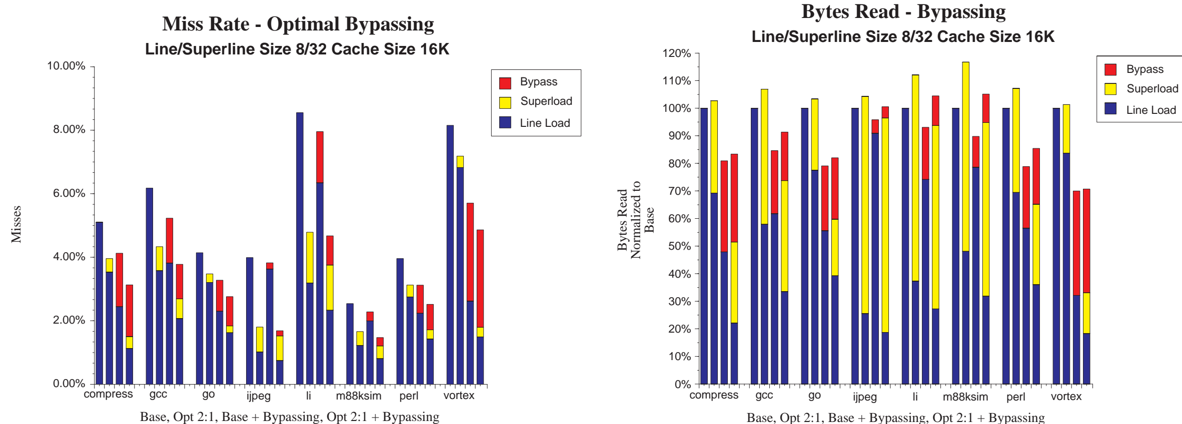


Figure 3: Performance comparison between superloading with and without bypassing. The miss rates and bytes read associated with the base case and Opt 2:1 are displayed, with and without bypassing, for all applications. Cache size 16K and line size 8 bytes. The bytes read are normalized to the base case with no bypassing (set to 100) for each application.

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, even where the base case is simulated at each application's individual optimal line size. In this sense the optimal algorithm captures the individuality of the separate applications, by producing a miss rate for each that is comparable to its performance at its own optimal line size. The brute-force approach of doubling the cache size still leads to a lower miss rate than superloading.

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. We examine here 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. 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 11.12% to 23.25%, as the cache size increases from 8K to 32K.

### 6.1.4 Bypassing

We move on to consider the effects of bypassing both with and without superloading. The important question we seek to answer from these experiments is how the potential performance improvement suggested by Figure **??** will be affected if the hardware is enhanced to provide bypassing capability.

Figure **??** shows the performance improvement offered by an optimal superload algorithm in the context of bypassing. We compare the performance improvement as between base and Opt 2:1 in the no-bypassing case to the same experiment simulated on a cache with bypassing. We see that, overall, the benefits of each modification (bypassing and superloads) are independent of each other. The relative improvement due to superloading is generally preserved when we move to the simulation of a cache with bypassing. Second, we might expect bypassing to increase the number of superloads, since – on initial analysis – the data caching property of a superload can be enhanced by bypassing. However, we see from the results in Figure **??** (left-hand graph) that the number of superloads declines slightly when bypassing is introduced; the improvement in miss rate due to bypassing is reflected entirely in a lower number of loads. Finally, we note that neither technique clearly outperforms the other: in some applications, such as *vortex*, bypassing offers tremendous improvement even compared to superloads; while in others, such as *ijpeg*, bypassing offers little improvement, while superloading offers substantial improvement.

In Figure **??** (right-hand graph), we consider the corresponding effect on bandwidth in the context of bypassing. We see that, as with miss rate, the overall effect of bypassing varies across applications, but the proportional impact of superloads – this time to increase bytes read – is approximately the same as in the no-bypassing case. We note that the precise bandwidth implications of bypassing are complex, especially at larger line sizes where bypassing may result in smaller data loads. Our simulations make two simplifications, that the cost of bypassing is equal (in both latency and bandwidth) to the cost of a line load; and that there is no bypass buffer common across superlines to offer temporary cache storage.

#### 6.1.5 Predictability

We next turn to a more detailed analysis of the optimal algorithm and the miss characteristics of the applications. The purpose of this analysis is to help in understanding online adaptive algorithms, by illustrating the degree to which the load/superload decision follows predictable patterns in the optimal sequence. We focus on predicting the correct decision for a given line based on the previous decision the optimal algorithm took for that line (or for any of its relatives), taken at the most recent miss reflected in the simulation on that line (or on a relative). If the previous decision is a good indicator of the correct current decision, it suggests that an online adaptive algorithm based on that technique could be successful; if however the previous decision is a poor indicator, it suggests that other online approaches should be considered instead.

**Cold misses** A threshold question in considering such an adaptive scheme is the most appropriate action in the case of cold misses, where there is no history on which to base the load/superload decision. We ask two specific questions in Table **??**: what proportion of misses were cold misses; and of those cold misses, how often did the optimal algorithm perform loads

| Application | Loads | Superloads | Pct of total misses |
| --- | --- | --- | --- |
| compress | 60.2722 | 39.7278 | 1.4322 |
| gcc | 64.9284 | 35.0716 | 1.7965 |
| go | 51.3278 | 48.6722 | 0.2150 |
| ijpeg | 34.6594 | 65.3406 | 5.5759 |
| li | 36.9082 | 63.0918 | 0.3301 |
| m88ksim | 18.9395 | 81.0605 | 12.7968 |
| perl | 34.0139 | 65.9861 | 0.6890 |
| vortex | 58.1453 | 41.8547 | 2.2000 |
| average | 44.8993 | 55.1007 | 3.1294 |

Table 3: Cold miss breakdown: the proportions of cold misses on which the optimal sequence performed loads and superloads; and the proportion of total misses that such cold misses represents. A reference is considered a "cold miss" if it is the first reference to that line and it is a miss. (Because of superloads, the first reference to a line can be and often is a "cold hit"; these are not included in this table.) Opt 2:1 is used to produce the optimal sequence. Cache size 16K, line size 32 bytes.

or superloads. This data helps to indicate how an adaptive algorithm should initialize its decision process, and how important that initialization decision is to the algorithm's performance. We see from Table **??** that the contribution of cold misses to the overall performance is substantial in some cases (*m88ksim*) and relatively unsubstantial in others (*go, li*). We see that different applications have significantly different cold miss behavior. In the LSP described in section **??**, we incorporate a single initialization (weak superloads) suggested by the overall trend in Table **??**.

**Superloads** We examine the optimal sequence to estimate the predictability of superloads based on previous load and superload decisions for the same line. Considering the sequence of references to a single line, we extract from that sequence those references that (i) resulted in a miss (load or superload) and (ii) occurred at a time when the reference had no relatives in the cache. The sequence so extracted is then examined for consecutive actions. The percentage of superloads (respectively, loads) immediately preceded by a load or superload, as the case may be, is displayed in Table **??**. The loads and superloads analyzed in Table **??** do not include the "cold misses" displayed in Table **??**; they also do not include the first miss after a "cold hit" (where the first reference to a line is a hit in the cache). We see from Table **??** that a load is extremely well predicted by a preceding load, while a superload is fairly accurately indicated by a preceding superload. The LSP is designed on this premise; its effectiveness is analyzed in the following subsection.

| This reference: | Superload | | Load | |
|---|---|---|---|---|
| Previous reference: | Load | Superload | Load | Superload |
| compress | 37.8772 | 62.1228 | 97.2410 | 2.7590 |
| gcc | 31.1431 | 68.8569 | 93.5538 | 6.4462 |
| go | 42.2985 | 57.7015 | 96.6771 | 3.3229 |
| ijpeg | 13.3187 | 86.6813 | 88.6827 | 11.3173 |
| li | 34.9917 | 65.0083 | 78.2467 | 21.7533 |
| m88ksim | 8.0170 | 91.9830 | 98.7273 | 1.2727 |
| perl | 52.6607 | 47.3393 | 97.1489 | 2.8511 |
| vortex | 23.2043 | 76.7957 | 99.5481 | 0.4519 |
| average | 30.4389 | 69.5611 | 93.7282 | 6.2718 |

Table 4: Predecessors of loads and superloads. Consecutive references to the same line for which each occasioned a miss and each occurred when no relatives occupied the cache, broken down by cache actions. Opt 2:1 is used to produce the optimal sequence. Cache size 16K, line size 32 bytes.

### 6.2 Line Size Predictor

Our online prediction scheme can be quite effective in reducing cache misses. Typically the performance of the LSP varies between those of the Opt 2:1 and the Opt 3:1 algorithms. The miss rate is shown in Figure **??** (left-hand graph) where it can be seen that the LSP is close to the Opt 2:1 for the same line size (2nd and 4th bars). In the case of *ijpeg*, LSP actually outperforms Opt 2:1 in miss rate. This is not a contradiction; as previously mentioned in section **??**, the Opt 2:1 algorithm minimizes not cache misses but the sum of $L + 2SL$.

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, not shown in Figure **??**, is the application *go*, where the base case beats the LSP for a line size of 64 bytes.

Another observation from Figure **??** is that the LSP does more superloads than the optimal algorithms. This translates into a significant increase in the number of bytes transferred. 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. 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. This 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

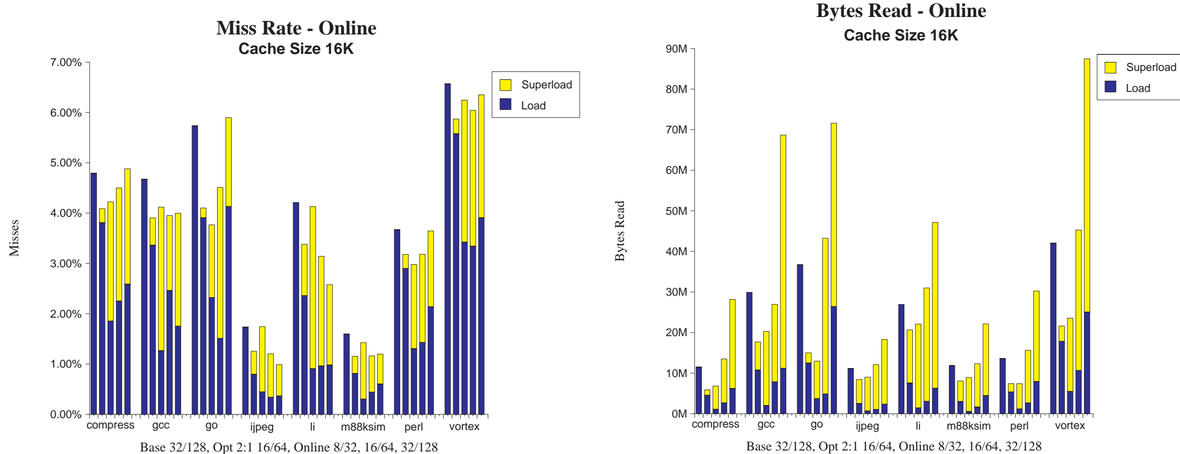


Figure 4: Miss rates and bytes read for the eight benchmark applications. The LSP online algorithm is simulated at line sizes of 8, 16, and 32 bytes. For comparison, the base case (no superloads) and the optimal algorithm Opt 2:1 are simulated at a line size of 32 bytes. Cache size of 16K. For each application, the figure for bytes read for each algorithm is normalized to the respective base case (set to 100).

of superloading. This implementation of 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.

# 7 Conclusion and future work

We have presented offline algorithms for determining the optimal sequence of loads, superloads and bypasses for a given reference stream for direct-mapped caches. 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 tradeoff 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 analyzed the interaction of bypassing and superloading, and conclude that the performance improvements for each are comparable in magnitude and largely independent of each other.

We have also presented an online algorithm for determining the sequence of loads and super-

loads. 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.

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 combinations of lines 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. Profiling using optimal algorithms and using the results to statically identify profitable loads and superloads is one promising idea. And perhaps most significant, optimal results allow the construction of a framework which can lead to the determination of better parameters for lookup tables and knowledge mechanisms.

# References


[1] Craig Anderson and Jean-Loup Baer. Two techniques for improving performance on bus-based multiprocessors. *Future Generation Computer Systems*, 11:537–551, 1995.

[2] Anonymous. Private communication.

[3] Lazlo Belady. A study of replacement algorithms for a virtual storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[4] Doug Burger and Todd M. Austin. The simplescalar tool set, version 2.0. Technical Report 1342, Computer Science Department, University of Wisconsin, Madison, WI., 1997.

[5] Charles Conti. Concepts for buffer storage. *Computer Group News*, 2:9–13, 1969.

[6] Czarek Dubnicki and Thomas LeBlanc. Adjustable block size coherent caches. In *Proc. of 19th Int. Symp. on Computer Architecture*, pages 170–180, 1992.

[7] Antonio Gonzalez, Carlos Aliaga, and Mateo Valero. A data cache with multiple caching strategies tuned to different types of locality. In *Proc. 1995 Int. Conf. on Supercomputing*, pages 338–347, 1995.

[8] John Hennessy and David Patterson. *Computer architecture : a quantitative approach.* Morgan Kaufmann, 2nd Edition, San Francisco, CA, 1996.

[9] Teresa Johnson and Wen mei Hwu. Run-time adaptive cache hierarchy management via reference analysis. In *Proc. 24th Int. Symp. on Computer Architecture*, pages 315–326, 1997.

[10] Teresa Johnson, Matthew Merten, and Wen mei Hwu. Run-time spatial locality detection and optimization. In *Proc. 30th Int. Symp. on Microarchitecture*, pages 57–64, 1997.

[11] Sanjeev Kumar and Chris Wilkerson. Exploiting spatial locality in data caches using spatial footprints. In *Proc. 25th Int. Symp. on Computer Architecture*, pages 357–368, 1998.

[12] Richard Mattson, Jan Gecsei, Donald Slutz, and Irving Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[13] Steven Przybylski, Mark Horowitz, and John Hennessy. Performance tradeoffs in cache design. In *Proc. 15th Int. Symp. on Computer Architecture*, pages 290–298, 1988.

[14] Jude Rivers, Edward Tam, Gary Tyson, Edward Davidson, and Matthew Farrens. Utilizing reuse information in data cache management. In *Proc. 1998 Int. Conf. on Supercomputing*, pages 449–456, 1998.

[15] Ted Romer, Wayne Ohlrich, Anna Karlin, and Brian Bershad. Reducing tlb and memory overhead using online superpage promotion. In *Proc. 22nd Int. Symp. on Computer Architecture*, pages 176–187, 1995.

[16] Rabin Sugumar and Santosh Abraham. Efficient simulation of caches under optimal replacement with applications to miss characterization. In *Proc. SIGMETRICS Conf.*, pages 24–35, 1993.

[17] Madusudhan Talluri and Mark Hill. Surpassing the tlb performance of superpages with less operating system support. In *Proc. ASPLOS-VI*, pages 171–182, 1994.

[18] Gary Tyson, Matthew Farrens, John Matthews, and Andrew Pleskun. A modified approach to data cache management. In *Proc. 28th Int. Symp. on Microarchitecture*, pages 93–103, 1995.