Victim management in a cache hierarchy

P. A. Franaszek
L. A. Lastras-Montaño
S. R. Kunkel
A. C. Sawdey

We investigate directions for exploiting what might be termed pattern locality in a cache hierarchy, based on recording cache discards or victims. An advantage of storing discard decisions is the reduced duplication of pertinent information, as well as the maintenance of information on the current location of discarded lines. Typical caches are designed to exploit combinations of temporal and spatial locality. Temporal locality, the likelihood that recently referenced data will be referenced again, is exploited by LRU-like algorithms. Spatial locality is the property that causes larger cache lines to yield improved miss ratios. Here we consider the exploitation of pattern locality—the property that lines accessed in temporal proximity tend to be re-referenced together. We describe some new cache structures including pattern-recording features, along with their miss ratio and transfer traffic performance as determined via simulations on traces drawn from several benchmark applications. We show that pattern locality information, based on discard statistics, can be useful in enhancing the quality of prefetch decisions.

1. Introduction

In the field of microprocessor cache hierarchy management, new viable strategies for mitigating the effects of memory latency continue to increase in importance. Well-established solutions such as increased cache sizes, prefetching of strides, larger cache lines, and multithreading are becoming increasingly insufficient to prevent processors from underperforming because of memory starvation. This is because of the trend toward increasing distances, in processor cycles, between on- and off-chip accesses. The problem of the resulting limitation on system performance is sometimes termed the memory wall. A question of some importance is whether there are unexplored potentially viable approaches to mitigating this problem. Viability here broadly means having real possibilities of successful application in future computing systems.

Caches and cache hierarchies are effective because of a combination of the temporal and spatial locality of accesses. Temporal locality ensures the success of least recently used (LRU)-like replacement algorithms, while spatial locality can be exploited by a proper choice of line size, such as a tradeoff between miss ratios and transfer traffic.

Spatial locality can be viewed as a property that holds across the contents of the entire memory—that is, that a randomly selected line is likely to be referenced in temporal proximity to another if they are neighbors in some general sense. For example, one may say that two lines are neighbors if their addresses differ numerically by no more than a given number. A property specific to actual items of data might be termed *pattern locality*. This is the property that items accessed in temporal proximity are likely to be accessed again in such proximity.

Not all neighboring cache lines necessarily exhibit pattern locality. Identifying lines having this property thus offers an opportunity for improving prefetch or replacement performance over methods such as simply increasing the size of cache lines. Detecting pattern locality can be done, for example, by recording data accesses.

Throughout this paper, we use the standard terminology of cache design. A cache victim is a cache line or entry that was chosen for replacement at the time a

0018-8646/06/$5.00 © 2006 IBM

new entry was installed. A physical realization of a cache consists of a memory that stores entries and tags or identifiers that, together with the memory storage location, identify the entries completely. The number of possible locations any given entry can take within the memory is called the associativity of the cache (when the associativity is 1, we say that the cache is direct-mapped). Line or entry identifiers and the information that is used to select cache victims are held in a separate memory called the cache directory.

In this paper, we discuss some new structures for the efficient exploitation of pattern locality. In particular, we consider the notion of *victim prefetching*, in which prefetching is based on locality information recorded at the time a line is discarded from a cache rather than at time of reference. We describe what we believe is a new construct, a Directory eXtension (DX), for tracking victim patterns within pages. This is in contrast to hardware for tracking lines which are referenced, information which exhibits some redundancy with L2 directory contents. We also consider the modifications and use of on-chip directories for off-chip caches. Such directories are repositories of substantial information on previous accesses.

We investigate cache performance enhancements resulting from these notions in L2 cache configurations in which L2 is the slowest on-chip cache. Our systems include L2 side buffers such as victim caches, into which lines may be prefetched, victim lines inserted, or both. The systems considered include ones with and without off-chip L3 caches. Our performance analysis is via simulations based on traces obtained from execution of various benchmarks on IBM processors equipped with special hardware. The benchmarks tested include 1) TPC-C**, an on-line transaction processing benchmark; 2) Trade2, an IBM internal benchmark which simulates stock trading, is written in Java**, and uses the WebSphere* application; 3) CPW, an IBM internal benchmark similar to TPC-C but with more complex transactions; and 4) NotesBench*, a benchmark that exercises the Lotus Notes* e-mail application. Because of space constraints, we give detailed results for only the first two.

The use of a relatively small (of the order of 12% of the L2 size) side buffer or cache to hold victim or prefetched lines has two main advantages. One is that the mapping of cache lines to sets can be different from the one used for the main cache, thus mitigating effects due to imbalance in references between equivalence classes. Another is that prefetched lines do not interfere with regular cache contents and vice versa. In this investigation, there is an additional effect or rationale. Our traces, as discussed below, constrain our investigation of L2 performance to caches whose size is an integer number of megabytes, with the number of congruence classes greater than or equal to that corresponding to a one-megabyte direct-mapped cache. (This is related to the stack property of the LRU algorithm used and the observation that if the number of congruence classes is identical in a larger cache, we simply have more items in each congruence class.) Further, since cache sizes appropriate for these traces consist of a few megabytes (benchmarks used for systems with larger caches would exhibit larger working sets), our investigation is limited to caches with small associativity. The low level of associativity exacerbates problems associated with mistaken prefetches and hot spots in the cache. These issues are mitigated with the use of side buffers.

For a prefetch strategy to be effective, it must issue a significant number of prefetches; a large fraction of those prefetches must be subsequently referenced before eviction, but such references should ideally happen after the prefetch has been finalized in order to maximize the associated benefit. These notions may be termed *coverage, accuracy*, and *timeliness* [1]; a full assessment of a prefetching mechanism should include these three. The focus in our paper is on the first two, and we include only a brief discussion of the third. One reason for this is that timeliness is very much implementation- or system-dependent, whereas the other two are largely a function of the reference string. Coverage and accuracy combined produce miss-ratio reductions.

Our results indicate that victim prefetching can yield significant miss-rate reductions compared with configurations in which the side buffer functions only as a victim cache. Similarly, for the caches we consider here, with a low degree of associativity, victim caching combined with victim prefetching yields improved miss-ratio performance over either policy used alone. Prefetching increases the amount of fetch traffic over simple demand fetching, but this increase appears in the simulations to be perhaps acceptable in practice; we quantify this effect directly by computing the probability that a prefetched line is not requested before eviction from the side buffer (this measures the accuracy of the prefetches). As we later see, for certain simple processors we can actually make significant positive statements about the timeliness of our prefetch strategies.

The paper is organized as follows: Section 2 describes prior related work, and Section 3 introduces some basic assumptions with respect to the system considered. The essence of our main contribution can be found in Section 4, where we discuss in detail the operation of the prefetch policies. Section 5 describes our experimental methodology and further details our system assumptions. Experimental results can be found in Section 6, with conclusions in Section 7. In the Appendix (Section 8), we include a detailed description of the experimental tracing methodology used in the collecting and processing of the traces.

Our main conclusions are the following:

- The use of prefetch feedback combined with on-chip directories for off-chip caches can yield effective prefetch performance.
- The use of a directory extension is effective for prefetching in systems with no L3.
- The combined use of victim caching and prefetching can provide miss ratios corresponding to substantially larger caches with only modest increases in transfer traffic.

As mentioned above, our results are largely restricted to determining prefetch opportunities and prefetch accuracies as defined here (namely, whether a prefetched line will be referenced before it is discarded). The resulting performance improvements, if any, are further determined by the order and timing in which items are actually referenced, as well as by the detailed architecture of the buses and memory subsystems. We include some simple examples to quantify the scale of the improvements.

2. Description of prior work

The most common way to exploit spatial locality consists of employing large line sizes in a cache, a technique that could be regarded also as a form of prefetching. For many workloads of interest, it is a generally accepted empirical fact that increasing the line size often results in improved cache-miss rates, in spite of the resulting reduction of the number of lines that may be stored in the cache. However, large lines are also associated with increased data movement as well as increased coherency interference between processors.

In techniques that use information from prior references, the data transferred is varied on the basis of earlier recorded accesses. Perhaps the first example of such a technique can be found in the work of Franaszek and Bennett on adaptive variation of the transfer unit size [2]. They introduced the notion of storing information (for each block of several pages in a database) pertaining to pages accessed within this data block; the information was then used to control the data transferred (when doing I/O) as that block was referenced; prefetches were placed in a side buffer. See also the work of Van Vleet et al. [3]. In the work of Alexander and Kedem [4], prefetches are determined using a table that stores potentially multiple addresses of groups of cache lines that were referenced after a given reference. The SRAM buffers of Charney and Puzak [5] are placed between two caches to improve the apparent performance of the one closer to the processor. The prefetching technique considered is a variant of next sequential prefetching; one of the features of this work relevant to ours is that a confirmation bit is employed for improving prefetch accuracy. The work of Johnson and Hwu [6] proposes to segment memory in regions called macroblocks for which statistics for reference patterns are then kept in a memory access table. This table has a counter per macroblock that denotes the frequency of access of this block; on the basis of this counter, it is decided which line to keep during a replacement decision. A possibility is to prefetch subsets of a memory page, not necessarily contiguous, on certain accesses to the page (for example, L2 cache misses). The work of Kumar and Wilkerson [7, 8] exploits this general idea through a Spatial Footprint Predictor, which tracks which portions of a block have been accessed in the past. In his Ph.D. thesis [9], Burger discusses the notions of dual-size fetches and sub-block prefetching, which correspond roughly to the ideas of adaptive transfer unit size and prefetching of lines that are noncontiguous but spatially close. Yu and Kedem [10] propose the use of a prediction table cache with entries representing historical access information for cache lines within a given page; see also Kedem et al. [11]. The work of Lai et al. on dead block correlating prefetching [12] is a prefetching scheme that links a candidate prefetch to a line that is evicted as a result of a prediction that it will no longer be used; the link to the present paper is that in contrast in our work, lines that are evicted become potential prefetch candidates. Lin et al. [13] (see also the earlier work of Lin et al. [14]) use the notion of density vectors, which is closely aligned with the work of Kumar and Wilkerson [7] and Burger [9]. Temam [15] extended Belady's MIN algorithm to a setting in which prefetching is done with no memory latency. One of the scenarios considered by Temam is that of prefetching lines within the same page to which a cache access is made. Moreno et al. [16] describe a general table-based mechanism that stores accesses to pages and uses this information in preparing suitable prefetch candidates. Hu et al. [17] describe a technique using predictions based on correlations between sequences of accesses in different cache congruence sets. The use of meta-data structures to track reference patterns of regions in memory can also be found in other work; for example, in the recent work of Moshovos [18] and Cantin et al. [19], the authors introduce filters that allow them to prevent remote directory queries when executing a cache coherency protocol.

Another idea exploited here is the notion of victim lines, that is, lines that are ejected from a cache because of a buffer-management action. Jouppi [20] introduced *victim caching*, which in his original design is used to improve the performance of a direct-mapped cache with

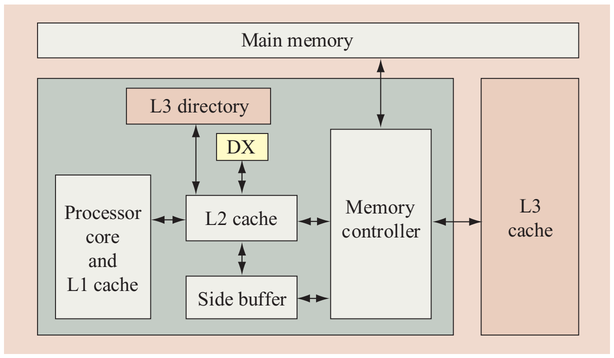

Figure 1

Overview of system architectures considered. We consider separately the settings in which there is an on-chip L3 directory and those in which there is none; in the latter we include a new construct that we call a Directory eXtension (DX).

the addition of a small, fully associative cache that receives victims from the primary cache. The idea of victim caching has other attractive properties, which led for example to its use in some IBM PowerPC* microprocessors, in which L3 is configured as a victim cache of the L2 as a complement to a policy that directly inserts into L2 lines requested from memory on L3 misses.

Some ideas we consider here are similar to those mentioned above. However, the access information we maintain in the DX is on pages with recently evicted lines, rather than pages with recently referenced lines, thus enabling what we term *victim prefetching* without the maintenance of redundant cache data. Victim prefetching is further combined with victim caching, which is shown to yield additional advantages. Another difference from previous work is our utilization of the directories of large L3s, which can be used as repositories of useful reference information. We combine these structures with feedback mechanisms that improve prefetch accuracy. Another aspect of our results is that they are based on trace analyses that permit the study of very large traces drawn from commercial applications.

3. System structures

As mentioned above, our main focus here is the use of pattern locality for reducing the effective L2 miss ratio. The configuration we use includes a relatively small side buffer in which we place or replace carefully chosen data. In our work, the side buffer can be used in one of two ways: In the first, it is the recipient of all prefetches and all victims of the L2 cache; in the second, it only receives all prefetches. In the first approach, we assume one additional bit per entry in the side buffer in order to distinguish prefetches from L2 victims.

As it is often true of ideas that can be stated at the policy level, the present work may be applicable to other cache settings. For example, the results for the combination of L2 cache and side buffer may carry over to caches with no side buffer but with larger associativity.

In contrast to the best known successful prefetch strategies such as wide line prefetch or stride prefetch, the methods described in this work rely on a significantly larger amount of past learned information, thereby raising the associated storage and data-management issues. We consider two approaches to this problem that reflect whether or not an L3 is present in the system.

L3 present in the system

For the purpose of this work, the relevance of the presence of an L3 is related to whether the associated L3 directory is easily accessible by the prefetch mechanism. In many systems with off-chip L3 caches, the L3 directory is on-chip for performance reasons (see **Figure 1**). If an L3 is present, we assume that lines evicted from the processor chip (either the L2 cache or side buffer) are immediately placed in the L3, and that this is the only way in which a line can be installed in the L3 (i.e., the L3 is a victim cache of the processor chip). Moreover, a line that is transferred from the L3 into the processor chip (either the L2 cache or side buffer) is deleted from the L3. This is largely for the purpose of discussion, as the results also pertain to standard L3s which have the inclusion property, namely that the contents of L2 and the side buffer are a subset of those in L3. In some architectures considered in our experimental section, evictions from the L2 cache are routed to the side buffer, not the L3 cache; then all evictions of the processor chip are from the side buffer, and these are incorporated in the L3 cache.

We expect an L3 to hold a much larger number of entries than an L2. As a consequence, L3 directories are generally rich information sources of past data references and are also a natural target for the storage of other useful statistics. To support this statement, note that by searching the L3 directory one can answer queries such as “Give me all lines within a given page that have been evicted recently from L2.” By appropriately augmenting the L3 directory, one can support a more elaborate query which further demands that no line whose last fetch was an unsuccessful prefetch be reported. We show that incorporating such feedback information in a prefetch policy can yield improvements in its effectiveness.

The L3 directory queries mentioned above could require a significant number of accesses (the L2 directory need not be scanned because of the assumption of exclusivity between the L2 and L3 contents). Since these directory queries are for the purpose of initiating prefetches, they can be regarded as low-priority requests in comparison with queries that are being originated by actual demands from the processor. This can be implemented with priority queues; note that basic query arbitration is generally already implemented in situations in which two or more L2 caches share one L3.

No system L3

If an L3 is not included in the system, the associated L3 directory will obviously not normally be considered in a chip design. Here we store all prefetch-related information in a structure we term a Directory eXtension (DX). A DX is a table organized on a page basis; each entry of the DX has a bit for every line in the associated page. The management of the entries of a DX is as in a standard cache (with a least-recently-used eviction rule applied on equivalence classes of the entries). The relevance of the page notion is that it is the largest storage unit for which we may generally expect spatial locality to be present (in systems with multiple page sizes we might choose the smallest such, generally 4 KB). Figure 1 shows the logical placement of a DX if it is incorporated in the design. The figure shows that such a system is obtained by omitting the L3 directory and the L3 cache. (We postpone a full description of the operation of the DX to the next section.)

4. Description of policies

We now consider management policies associated with the system structures described above. We define a *failed prefetch* in the obvious way, as a line that was prefetched but not referenced before eviction from the processor chip. We define a *prefetch window* as the number of contiguous lines from a page that are considered for a prefetch. In our experiments, we restrict our windows to be aligned fractions of pages (e.g., a half or a quarter of a page).

In systems with an L3 directory, on every L2 miss that also misses in the side buffer, we prefetch all lines of the same page that are valid in L3 from the prefetch window, except for lines whose last retrieval from L3 (if any) was a failed prefetch. Prefetches are placed in the side buffer, which is managed like a standard cache. If a miss from the L2 is found in the side buffer, the line is deleted from the side buffer and inserted in the L2 cache. As mentioned above, in some configurations we consider, the side buffer may also receive evictions from the L2. If a line is in the L2 cache, it is not in the side buffer. Conversely, if a line is inserted in the side buffer, by definition it was not in the L2 at insertion time, and thus the contents of the L2 and the side buffer are mutually exclusive.

The above policy may be implemented by including an extra bit in the directory for each line stored in L3. By default, this bit is set to 1 when the line is placed in the L3 after eviction, unless the line joined the side buffer as a prefetch from L3 but was never requested, in which case the bit is set to zero. Thus, on misses of the processor chip one then scans the L3 directory to find possible prefetches. We reiterate that such scanning is not in any critical path and can be assigned low priority with respect to normal directory queries.

For systems with no L3, we introduce a DX, structured as described above. The DX is managed according to the following policy:

1. *On evictions from L2 (in systems in which these are not placed in the side buffer)*. The corresponding page is searched in the DX; a new entry is created in case it is not found. In the DX entry, the bit corresponding to the evicted line is set to 1, denoting an "on" bit. Those lines associated with "on" bits we term DX *candidate prefetches*.
2. *On eviction from the side buffer*. If the evicted line joined the side buffer as an L2 eviction, the procedure is performed as described in step 1. Otherwise, it joined as a prefetch, and no further action is taken.
3. *On misses from L2 that also miss the side buffer*. The corresponding entry in the DX is looked up, and in case of a match, the lines from this page within the prefetch window which have their prefetch bits set to 1 are fetched and inserted into the side buffer, with the exception of the demand miss that triggered the prefetch event, which is placed in L2 directly. The prefetch bits for the fetched lines are then set to 0, and if all bits for this page are 0, the entry is deleted from the DX.
4. When a new entry is created in the DX and there is no space in the corresponding DX equivalence class, the LRU entry in this class is deleted.

We now briefly discuss the impact of the above policies on critical paths of the design of the L2 and L3 directories and caches. The eviction events in steps 1 and 2 simply result in the associated DX update with no further consequence. In step 3 we note that the memory demand fetch is scheduled at the earliest opportunity (as in a standard architecture) regardless of whether or not the DX lookup is finished; the prefetches are issued at the time their identity is learned. Note that a relevant performance issue is whether the memory subsystem is operating with an open or closed page policy, a detail beyond the scope of this paper.

5. Experimental methodology

Description of traces

We show simulation results using traces captured from buses of specially configured IBM machines. These traces further undergo a certain postprocessing phase to be described in the Appendix. Because of a number of physical limitations, the references reported in the trace consist of transactions observed in a bus only after they have been filtered through an L2 cache with 128-byte lines, which is 1 MB in size and direct-mapped in all cases. This corresponds exactly to 8,192 sets, each consisting of a single line. The specifics on how these traces were obtained can be found in the Appendix, which also includes a discussion on how one can reconstruct the contents of the directory of a hypothetical larger cache using only the trace information.

The conclusions of the Appendix are next summarized. It is feasible to recreate exactly the directory of the hypothetical larger cache if[1]

1. The number of lines of the larger cache is an integer multiple of the number of sets in a 1-MB direct-mapped cache.
2. LRU replacement is employed.
3. Exactly the same function that maps lines to sets is employed. For these traces, the sets are selected by address bits 7–19, assuming a numbering starting from 0.
4. For every one of the 8,192 sets, the number of LOAD misses in the trace mapped to the set is at least equal to the number of entries in the set. This is simply because otherwise there will be one or more entries in the cache with unknown contents.

A note on notation

When referring to the geometry of a memory (a cache, a directory extension, etc.), we use the notation *ambw*, where *a* and *b* are positive integers and the naming means that the cache has a megabyte of capacity and *b* ways. Thus, for example, 2m2w is a 2-MB, two-way cache. We also use other standard notation: B = byte, KB = kilobyte, MB = megabyte.

Overview of the experiments

The specifications of the computer system considered are as follows:

- A single-threaded uniprocessor with one L2 cache, one L3 cache, and a side buffer searched on misses of the L2.
- Lines evicted from the chip are immediately inserted into L3, which is a 32-MB, eight-way set-associative cache.
- L2 is 2 MB, two-way set-associative
- The side buffer is 256 KB and four-way set-associative.
- Prefetches are scheduled only in L2 misses, and prefetched lines must be within the same 4-KB page of the L2 demand miss associated with them.
- The line size is 128 bytes. Thus, a 4-KB page has 32 lines.
- The workloads are TPC-C and Trade2 (similar results for CPW and Notesbench are also discussed, but not in detail).

The strategy that shows the most promise among those considered here prefetches exclusively from L3 and uses feedback information to decrease the probability of an unsuccessful prefetch. Moreover, it employs the side buffer for storage of victims of the L2 as well, which are inserted at the most recently used (MRU) position.

We term the strategy above *victim caching and prefetching with feedback*. The alternative with no feedback has a corresponding name. Another method also considered is as above but without storing L2 victims in the side buffer; we call this technique *victim prefetching* with *feedback*/*no feedback* labels appended as appropriate.

For all prefetching techniques (novel or not) demonstrated here, including the ones described below, we consider restricting the potential prefetches to be within a page sub-block of a given size, with 2, 4, 8, 16, or 32 lines being the possibilities. We assign lines 1–16 to the first sub-block of length 16, and lines 17–32 to the second sub-block. For sub-blocks of length 8, the partitioning is (1–8) (9–16) (17–24) (25–32), and so forth. The sub-block selected is exactly the one to which the demand miss belongs.

Of primary concern is to contrast the technique with more conventional ones, and to address this issue we also simulate

1. *Simple cache (no side buffer)*. We report on statistics for various cache sizes.
2. *Victim caching*. The side buffer stores all evictions of the L2, and nothing else.
3. *Victim caching + contiguous block prefetching*. Here we prefetch a contiguous block of lines of a preset size (2, 4, 8, 16, or 32 lines). As in victim caching and prefetching, we also insert L2 evictions into the side buffer in the MRU position. Prefetches are always from the L3 and are inserted into the side buffer; we never prefetch into the L2. As described above, the address of the sub-block is obtained directly from the address of the line by setting to zero the appropriate number of least-significant bits.

In presenting our measurements, we take as a reference point the simple cache system for the smallest cache size

[1]In fact, simulations for more general situations are feasible; nevertheless, we do not explore them in this work.

simulated, in this case a 2-MB two-way set-associative cache. The parameters that we present for each prefetching technique are as follows:

1. The reduction of L2 LOAD misses relative to the smallest simple cache simulation. Note that L2 misses can also be caused by STORE events, but it may be argued that the LOAD miss reduction figure represents possible system performance improvements more accurately than a LOAD/STORE compound figure if one assumes a processor that continues execution past a store miss.
2. The total number of lines transferred from L3 to L2, relative to the same number for the smallest simple cache. These transfers could be due to misses on LOAD, STORE events, or prefetches. The rationale behind presenting the compound figure is that this better represents the net stress imposed on the communication bus from L3 to L2.
3. The same as in part 2 above, but considering the total number of lines transferred from either memory or L3 to L2. This data is mainly relevant to compare against the contiguous block prefetching possibility.
4. The ratio between the number of prefetched lines evicted from the side buffer divided by the number of prefetches added to the side buffer. Because every successful prefetch is deleted from the side buffer when demanded after an L2 miss, this statistic corresponds exactly to the probability of a failed prefetch.
5. The same as in part 4 above, but instead of prefetched lines, we consider lines that enter the side buffer because they are victims of the L2. This is relevant only in the victim caching and prefetching setting.

Description of the statistics-gathering procedure
In order to ensure that simulation start-up edge effects are negligible in our results, we let every simulation run until half of the records in the trace have been processed. A principal indicator of the relevance of the measurements presented is the fraction of L3 lines not initialized at this moment, because a poorly utilized L3 would indicate that the behavior of the prefetch algorithm would not yet be sufficiently stable to be measured. The corresponding L3 utilization in our simulations is at least 90% at the time tracking of the prefetching behavior begins. At this point we reset the performance counters, which are the source for the statistics reported in this paper. The final numbers are collected after all records have been processed. For a fixed workload, exactly the same number of records are processed for each experiment.

6. Experimental results
Our performance results are offered in two types of plots, the first describing tradeoffs between the L2 miss rate and data transfer traffic for a given policy and the second describing the probabilities of failed prefetch and failed victim caching events. The translation of these results into actual computer system performance results is a delicate matter because issues of prefetch timeliness and performance penalties due to additional queuing delays must be taken into consideration. In understanding the results, it may be advantageous to consider a simple observation relating traffic, prefetch performance, and traffic increases due to prefetching. Let T_r be the traffic without the fetches that will be saved by prefetching, T_p be the number of successful prefetches, and yT_p be the number of unsuccessful prefetches. The total traffic is then $T_r + T_p$ without prefetching, and $T_r + T_p + yT_p$ with prefetching. Suppose that the read traffic is four times the write traffic, that prefetching eliminates half the read misses, and that $y = 0.5$. Then the increase in traffic due to prefetching is 20%.

The first type of plot is exemplified by **Figure 2**, where the horizontal axis denotes the relative load miss-rate reduction compared with that of a simple 2m2w L2 cache. The vertical axis refers to the total number of lines transferred from L3 into L2 (load + store + prefetch), again normalized against the statistic for the simple L2 cache. Note that lines that cannot be serviced from the L3 are brought from memory. In Figure 2, no prefetching is enabled; our goal is to illustrate the relative improvements in L2 miss rates that are attainable by increasing the cache size along with the associated increase in associativity implied by our simulation restrictions. In what follows, we use these statistics to contrast performance improvements due to prefetching with those due to increased cache sizes. The second type of plot illustrates some other important properties of the prefetch algorithm, such as the number of prefetches executed but never referenced by the processor.

We offer results for two workloads, TPC-C and Trade2. We provide relevant statistics for our prefetch algorithm for a given workload, L2 cache size, and prefetch buffer size through a set of six plots. For purposes of intelligibility, we introduce the plots for TPC-C first on an individual basis, then bundle the plots for Trade2 in a separate figure; thus, after interpreting a particular plot type for the first workload, one may cross-check the parallel results for the second workload.

Prefetching with feedback enabled
Refer to **Figure 3**, where all statistics are normalized to those of a 2m2w L2 cache with no prefetching and with a disabled side buffer. For comparison purposes, we have also included the performance of a 3m3w L2 cache and

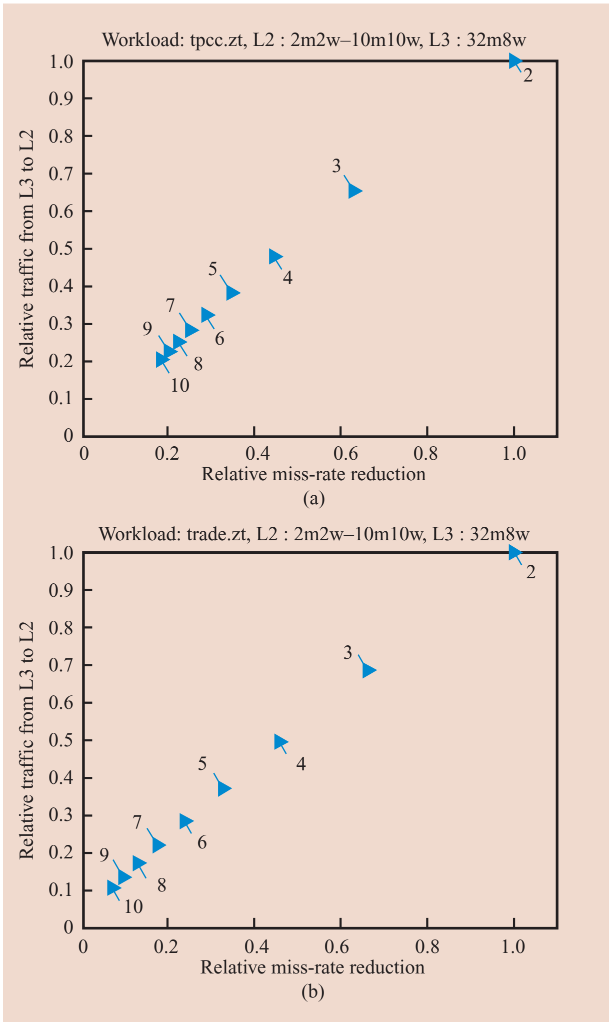

Figure 2

Performance of various L2 cache sizes for two workloads normalized to that of the 2m2w L2 cache; in these simulations there is no prefetching and no side buffer enabled. The integer next to each symbol denotes the size and associativity of the L2 cache.

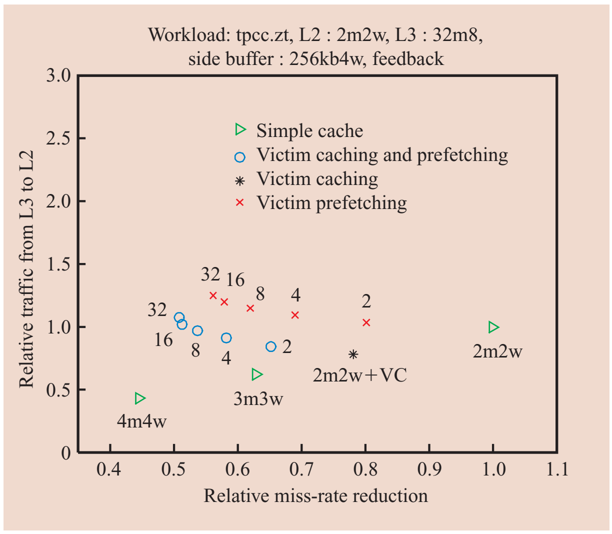

Figure 3

Victim prefetching with feedback for the TPC-C workload. The integer next to each of the blue circles and red cross marks specifies the number of lines that comprise the prefetch window. One sample conclusion is that victim prefetching gives significant miss-rate reductions and moderate traffic increases, and that further allowing for L2 victim caching gives an even better tradeoff.

a 4m4w L2 cache (green triangles). If the side buffer functions as a victim cache of the L2, one obtains the performance indicated by the asterisk. The number of misses removed by the victim cache is approximately half of the number of misses removed by the additional megabyte (and overall associativity) of the 3m3w cache.

When the side buffer is employed as a repository for prefetches only, and when the prefetching scheme is victim prefetching from L3 into L2 with feedback, we obtain the simulation points denoted by the red cross marks. Each of these represents a different choice for the prefetch window span. If the side buffer functions both as a prefetch buffer and as a victim cache, one obtains the points denoted by the blue circles. As can be seen, for this workload the latter strategy, which allows for both victim prefetching and caching, is a better tradeoff than the former.

The point labeled "16 ○" offers a reduction of misses, with respect to the simple L2 cache, of about 50%. The total traffic from L3 onto the chip remains approximately the same as the L2 cache alone. Furthermore, this prefetching technique is a significant improvement over the miss-rate reduction of the 3m3w cache. However, obtaining the actual overall performance benefit of these prefetches would require a more detailed analysis, involving issues such as prefetch timeliness and queuing penalties of the failed prefetches.

The above discussion is relevant for a designer who is contemplating the possible addition of a side buffer to support prefetching. Where a side buffer already exists or no side buffer will be considered, it is more appropriate to contrast a given prefetching technique against the performance of the L2 cache, with the side buffer functioning as a victim cache. Note that in the case in which no side buffer is considered, the above corresponds roughly to a situation in which the L2 has a sufficient degree of associativity to permit flexible management of prefetches versus demand fetches.

Compared with the L2 cache with a victim cache (the asterisk), the prefetching technique reduces the miss rate by approximately 35%, at a cost of about 32% increase in traffic from L3.

We now examine in more detail the efficiency of the side buffer (we refer the reader to **Figure 4**), in particular the probability of a failed prefetch (the *y* axis) and the probability that an L2 victim was cached in the victim cache but not used (the *x* axis), whenever the prevailing policy allows for victim caching. When there is no victim caching and only victim prefetching, we obtain the horizontal lines given by the vp(2) through vp(32) labels; here it can be seen that the probability of a failed prefetch increases with the size of the prefetch window span. Nevertheless, this probability is at most approximately 1/3. If only victim caching is allowed, we obtain the vertical line which indicates that the probability that a line inserted in the victim cache is not used before eviction is about 4/5. This probability is much higher than the previous one, but note that L2 victims are much less costly than prefetches in that they need not be transferred from memory. If one allows for both victim caching and prefetching (blue circles), both undesired probabilities increase; in particular, the probability of a failed prefetch now ranges from 1/5 to 2/5. Nevertheless, the blue circles still indicate a better performance tradeoff than the horizontal lines (as deduced from Figure 3), because the relatively large number of victim cached lines improve the miss rate significantly more than the degradation due to the increase of the undesired probabilities.

Prefetching with no feedback

We now contrast a system with no feedback with one which uses feedback information. Refer to Figure 3 and **Figure 5**, in particular to the points labeled with blue circles (victim caching and prefetching), which we indicate by using the shorthand notation "vcap." Compare the vcap(8) point in the no-feedback case with vcap(16) in the feedback case. The miss rate reductions relative to the 2m2w L2 cache are approximately the same, yet when there is no feedback the traffic increases about 37% for the no-feedback case and about 3% for the feedback setting.

Feedback may also prevent prefetches that would have been successful otherwise: Consider vcap(32) of the case with no feedback. Although the traffic measurement is about 2.7 relative to the 2m2w L2 cache, the reduction in misses is similar to that of a 4m4w cache, in contrast to the case with feedback. This suggests that there may be better ways to incorporate feedback information into the prefetch policy.

As in the case of feedback, in **Figure 6** we show a plot that includes the probabilities of unsuccessful prefetches for the policies considered. Both the horizontal lines (associated with victim prefetching) and the blue circles (victim prefetching and caching) indicate significantly

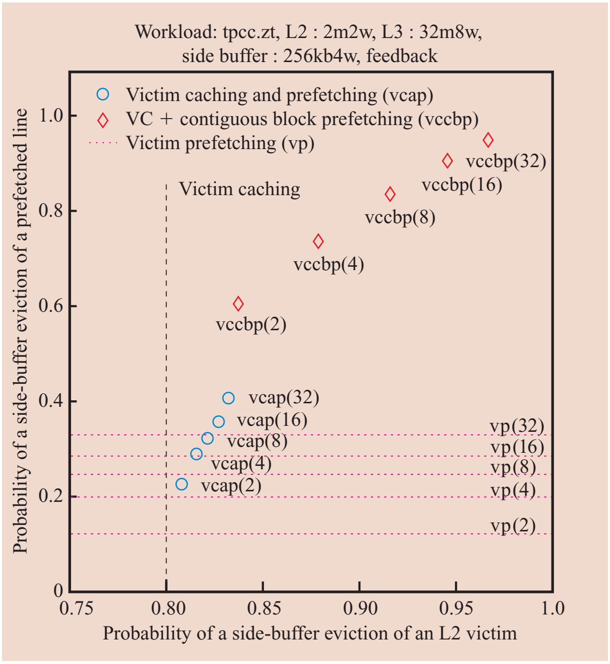

Figure 4

Side-buffer efficiency statistics for the TPC-C workload and prefetching with feedback. In both axes, less is better.

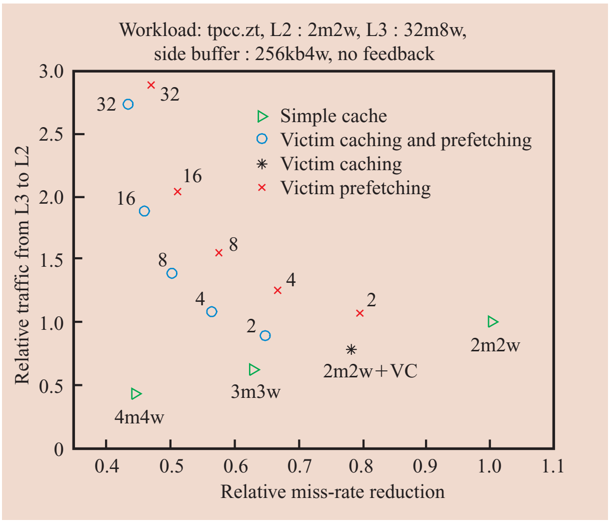

Figure 5

Victim prefetching with no feedback for the TPC-C workload.

worse statistics for the no-feedback case, as expected. For example, for vcap(16) the probability of a failed prefetch is around 2/3, in contrast to the same for the feedback case (Figure 3), which is about 1/3.

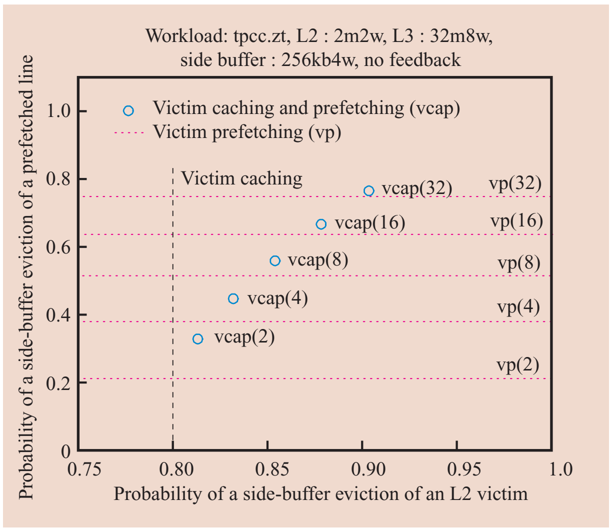

Figure 6

Side-buffer efficiency statistics for the TPC-C workload in the setting in which there is no feedback (see in contrast Figure 4).

Contiguous block prefetching

It is also of interest to consider the performance of a simple prefetcher, which makes no use of the L3 directory and does not employ feedback information. In this case, prefetches could come from memory or from the L3. The policy considered is one which prefetches a contiguous block of lines aligned, for example, on full, half, or quarter pages. The address of the block is obtained from the demand miss. In addition to prefetching, the side buffer also functions as a victim cache of the L2 (hence, the overall policy label is victim caching and contiguous block prefetching). For this policy, we present two plots in **Figure 7**. The difference between these is in the quantity represented by the y axis. In part (a), we present the number of lines transferred from L3 onto the chip, relative to the same measurement for the 2m2w L2 cache. In part (b), we present the total number of lines transferred either from memory or L3, relative to the total number of lines transferred for the simple 2m2w L2 case. The general conclusion we draw from these figures is that simple contiguous block prefetching is not a competitive policy; note the unreasonable traffic requirements and the high probability of an unsuccessful prefetch in the red diamonds of Figure 7.

Effectiveness of victim prefetching for a larger L2 cache

In **Figure 8** we find the counterpart of Figure 3 for the TPC-C workload, but in this case for a 4m4w L2 cache. The miss-rate reduction of victim caching and prefetch with a window span of 16 with respect to the victim-caching-only point (the asterisk) is approximately 30%, which is slightly less than the 35% that we previously observed for the 2m2w L2 case. The traffic increase with respect to the victim-caching-only point is about 40%, in contrast to the 32% observed in the 2m2w L2 setting. Another noticeable difference is that the simulation points in which full page prefetch was a possibility were strictly worse than the half-prefetch setting, likely due to side-buffer pollution by the additional prefetches. Thus, the benefits of the combined prefetch/victim side buffer are largely preserved when going to a larger cache with higher associativity. This may be partially because the

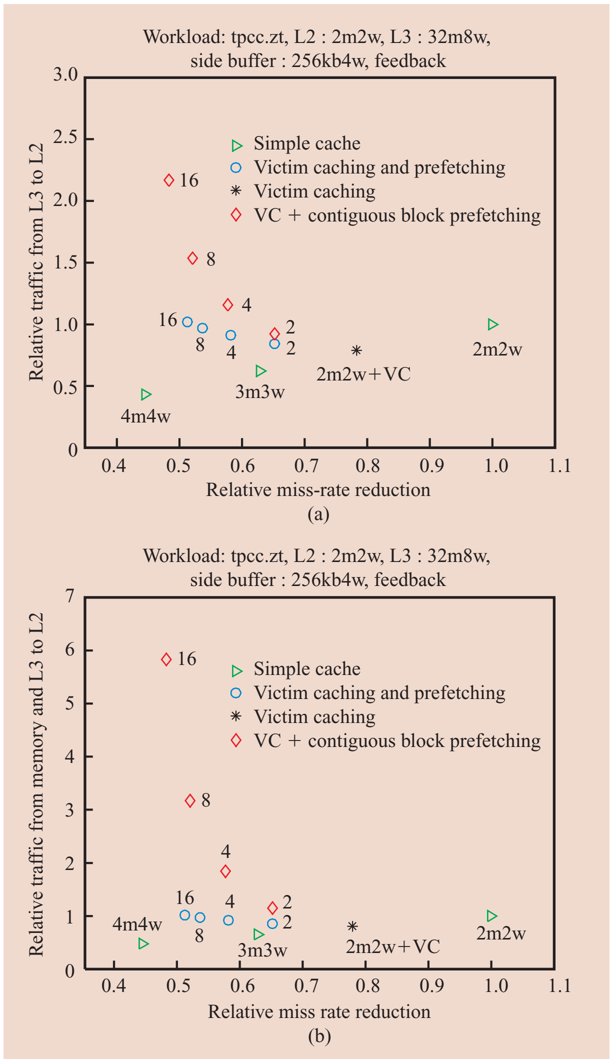

Figure 7

Victim caching and contiguous block prefetching for the TPC-C workload. The difference between the two plots is in the y-axis, which in (a) indicates the traffic from L3 and in (b) the traffic from both the memory and the L3.

degree of associativity in the side buffer masks that of the cache.

Directory extension effectiveness

We conclude by considering the effectiveness of adding a DX to the processor chip. As previously discussed, a DX is suggested for implementing the prefetch policy when there is no L3 cache (and hence no access to an L3 directory). We refer the reader to **Table 1**, where we report DX miss-rate statistics (recall that a DX is interrogated on L2 misses) in a system where no prefetching is enabled. For comparison purposes, we also report in **Table 2** the miss rate of an L3 that is 32 MB and eight-way.

Assume a 2m2w L2 cache. On every L2 miss we compute the prefetch candidates. With a DX with 1,024 entries, only about 20–25% of those misses are associated with a successful inquiry at the DX level. A DX that is eight times as large has miss rates ranging from approximately 23% to 37%, which is a considerable improvement over the DX of 1,024 entries. The optimal DX size depends on the available budget and desired performance benefit. If the L2 size is doubled, the misses become more difficult to predict, and the size desired for a DX approximately doubles, as suggested by the empirical results.

In this work we have chosen to simulate the L3 directory option instead of the directory extension version of the policy. In order to determine the size of a directory extension that is needed to match the prefetch information stored in the L3 directory, one may use the following simple observation: One should estimate the average number of different pages that have at least one line in the L3 directory, and then allocate that number of entries to the directory extension. We have observed that on average a page has four to eight lines in a cache; assuming the former, it means that the directory extension should have approximately a quarter of the number of entries of the L3 cache.

System performance implications

We now briefly discuss some performance implications for a highly simplified system model.

We assume a system with no L3, such that an L2 cache fault which is not serviced by the side buffer causes a delay D in machine cycles, and that a hit to the buffer causes no additional delay. Suppose that the trace represents N instructions. Typically, some subset of these are idle instructions, representing for example times when a page fault was being processed and no other work could be scheduled immediately. Each such instruction is executed in one cycle.

For the TPC-C trace described here, a representative fraction for idle instructions is 0.1. The number of infinite L2 cycles required per instruction is approximately 1.7.

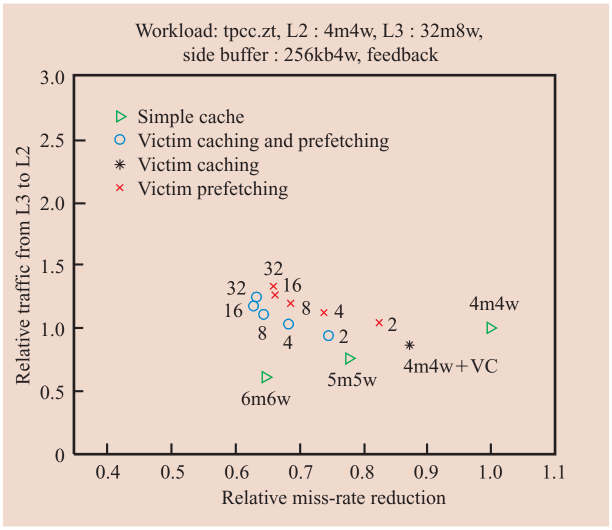

Figure 8

Victim prefetching for the TPC-C workload with feedback, with a 4m4w L2 cache (to be contrasted with the 2m2w L2 in Figure 3).

The percentage of non-idle instructions resulting in L2 misses (for the 1-MB direct-mapped cache) is approximately 0.02. The number of cycles required for a machine to process the set of N instructions is then

$$T = 0.1N + 0.9N(1.7 + 0.02RD),$$

where R is the relative miss ratio, compared to the 1-MB direct-mapped cache. The number of cycles per instruction (CPI) of the machine for non-idle instructions is about 3.27. For the machine generating the trace, D is of the order of 90. Thus, about half the processing time in this case is due to 1m1w L2 cache fault delays. Reducing the number of faults by a factor of 2 can thus improve performance by as much as 25%.

The miss rates of the prefetching methods studied here are given in comparison to the miss rate of a simple cache of the appropriate size. Therefore, to obtain performance numbers it is necessary to learn the miss rate of the simple cache compared with the miss rate of the 1m1w cache used for capturing the traces. **Table 3** gives this statistic for three workloads and cache sizes.

A sample calculation follows: Figure 3 shows that for TPC-C the miss rate of a 2m2w L2 with the "16 ○" strategy is approximately half that of a 2m2w L2 with no prefetching and no side buffer. The miss rate of a 2m2w L2 is ~0.49 of that of a 1m1w cache, as shown in Table 3. Compounding these numbers then yields about 0.25. This gives a CPI of 2.44 with a 2m2w L2 cache with no prefetching and a CPI as low as 2.03 for the same cache but with prefetching into the side buffer, which is a reduction of about 16% of the original CPI. The result is a

Table 1 Directory extension miss-rate statistics for various L2 cache sizes and workloads (no prefetching enabled, no victim caching). The notation *a*e*b*w implies a memory with *a* entries and with associativity *b*.

L2 geometry	*DX geometry*	*DX miss rates*			
		TPC-C	*TRADE*	*CPW*	*NOTES*
2m2w	8192e4w	0.237318	0.250004	0.320904	0.372551
	4096e4w	0.370400	0.452307	0.423462	0.513887
	2048e4w	0.560952	0.652450	0.579140	0.671960
	1024e4w	0.759551	0.814356	0.752682	0.808705
4m4w	16384e4w	0.278221	0.133674	0.512374	0.416528
	8192e4w	0.420749	0.321932	0.600887	0.556574
	4096e4w	0.579302	0.539234	0.703534	0.702265
	2048e4w	0.738021	0.727691	0.813192	0.830273
8m8w	32768e4w	0.281420	0.098430	0.579822	0.461278
	16384e4w	0.431931	0.183081	0.731036	0.585343
	8192e4w	0.566722	0.348746	0.832160	0.710878
	4096e4w	0.742797	0.563857	0.903598	0.823781

Table 2 L3 miss-rate statistics for various L2 cache sizes and workloads (no prefetching enabled, no victim caching).

L2 geometry	*L3 32mb8w miss rate*			
	TPC-C	*TRADE*	*CPW*	*NOTES*
2m2w	0.0907	0.007	0.2228	0.1412
4m4w	0.2255	0.015	0.3887	0.217

Table 3 Miss rate of an L2 cache of a given size with respect to the miss rate of a 1m1w L2. Statistics for three different workloads are presented.

Miss rate wrt L2 1m1w	*TPC-C*	*TRADE*	*NOTES*
L2 2m2w	0.488	0.622	0.515
L2 4m4w	0.302	0.315	0.285
L2 8m8w	0.123	0.110	0.164

bound for the actual CPI benefit, as the benefit is further reduced by factors such as the timeliness of the prefetches and additional queuing delays caused by the increased traffic.

To obtain a better estimate of the true CPI performance improvement, we would need to refine our model to take into account issues of prefetch timeliness and queuing penalties due to the failed prefetches; we next give a brief overview of a type of analysis one might undertake to account for these. To support our discussion, we refer the reader to the time diagram in **Figure 9**.

Often it is true that the latency to memory is determined mainly by the time required for the memory subsystem to react to a request, rather than the time required to transfer the line across the memory bus; the ratio of the former to the latter could be typically in the range of 8:1 to 16:1. In our simplified model, the processor always stalls waiting for a line to be serviced from memory. We assume that the prefetches associated with an L2 miss can be brought back-to-back across the memory bus after the initial missed line is transferred from memory; this can be done because the prefetches are coming from the same page and because of an assumption that the DRAM devices are functioning with an open-page policy. If after processing the data requested in the first miss the processor immediately issues a request for a second line, and if such a line was actually prefetched by the policy, then on average (assuming that we do not have special prefetch ordering strategies in place) the second miss will be found in the middle of the sequence of prefetched lines that come after the initial miss.

The observation above can be used to provide an initial estimate of the prefetch benefit; note that here we have made a pessimistic assumption that the second miss comes immediately after the first miss is serviced. The prefetch benefit can be further discounted to take into account an increased queuing penalty due to the prefetches and other issues such as processor multithreading. A complete analysis is beyond the scope of this paper; we simply mention here that in realistic settings we have computed the prefetch CPI benefit to be at least half of that predicted by a simple miss-rate reduction computation at the beginning of this section; thus, in that example the CPI improvement would be 8% instead of 16%.

Discussion on other workloads

The qualitative behavior of the prefetching technique considered in this discussion remains constant across a variety of different workloads; in particular, for comparison we refer the reader to **Figure 10**, which bundles the results for Trade2 in a single figure. In particular, in the plot at the top left (to be contrasted with Figure 3), the "16 ○" point in fact meets the miss-rate reduction of the 4m4w cache, again with approximately the same number of fetches from L3 as in the simple L2 cache situation. The miss-rate improvement, when compared to that obtained by TPC-C, is slightly greater: In this case, a miss-rate reduction of about 53% is experienced. When compared to the performance of the L2 cache with a victim cache, the miss-rate reduction is about 42% at a cost of a 25% total traffic increase.

The interpretation of the other plots of Figure 10 follows along the same lines as the methodology described above. A feature of reasonable interest is that the results hold qualitatively across both workloads; in fact, the same is true of the other two workloads studied but not presented here (CPW, Notes). The best strategy, using a reasonable criterion, is consistently that of victim caching and prefetching with feedback, and the gains in L2 miss-rate reductions are within the same order of magnitude.

7. Conclusion

Memory latency is a factor with increasing performance impact as processor speeds outdistance those of the memory subsystem. Two approaches to this problem are to a) have more effective policies for the retention of required data in the on-chip caches, and b) initiate the transfer of such data before it is actually requested (prefetching). In this paper, we have considered some techniques for prefetching based on the observation that access patterns appear to be significantly page-specific. We have considered two basic approaches: the use and augmentation of access pattern information implicit in directories for off-chip caches, and the use of a special construct, which we term a *directory extension*, for holding such information in the absence of such a directory. We have shown that page-specific access information, coupled with feedback to improve prefetching accuracy, can yield significant miss-ratio reductions at the cost of what may be acceptable traffic increases for a variety of commercial workloads. The use of a directory for an off-chip cache may be particularly attractive, as it is associated with a modest additional cost in hardware complexity. Directions for future work may include more detailed performance evaluations for systems with multiple processors and studies of issues at the hardware–software interface. An example of the latter is the consideration of the inclusion of thread information with DX entries.

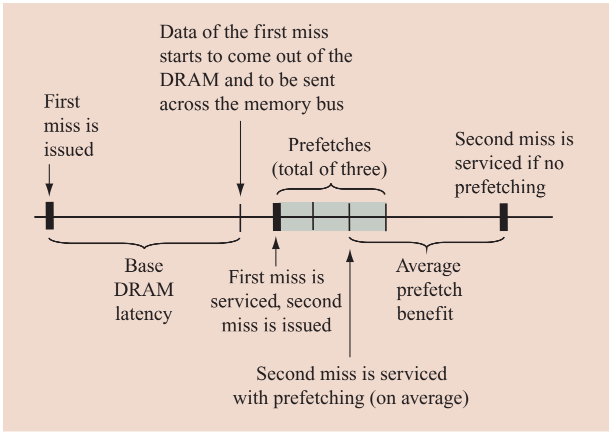
Data of the first miss starts to come out of the DRAM and to be sent across the memory bus
First miss is issued
Prefetches (total of three)
Second miss is serviced if no prefetching
Base DRAM latency
First miss is serviced, second miss is issued
Average prefetch benefit
Second miss is serviced with prefetching (on average)

Figure 9

Time diagram to illustrate the timeliness attributes of our prefetch strategies. In this example there are three prefetches; on average, the desired prefetch will be serviced after the second prefetch is received.

8. Appendix: Tracing methodology

The traces are collected by attaching a probing device to the bus between the L2 cache and the rest of the system. There is no L3 cache. Captured by this probe are all L2 cache misses that go to memory and all commands to and from the I/O (DMA and MMIO commands). This method is used because the benchmarks traced are large, commercial applications with a great deal of interaction with various types of I/O that makes them difficult to capture using other methodologies. The processor is in-order, so there are no speculative cache misses. The processor supports multithreading, but it was turned off. For all of the traces, the operating system was OS/400* and the database was DB2*.

Because the probing device is placed at the bus, there are some restrictions on the type of information that can be extracted. For example, clearly every processor load instruction which hits in the L2 cache is not observed. The events that can be observed that are of importance to our study are

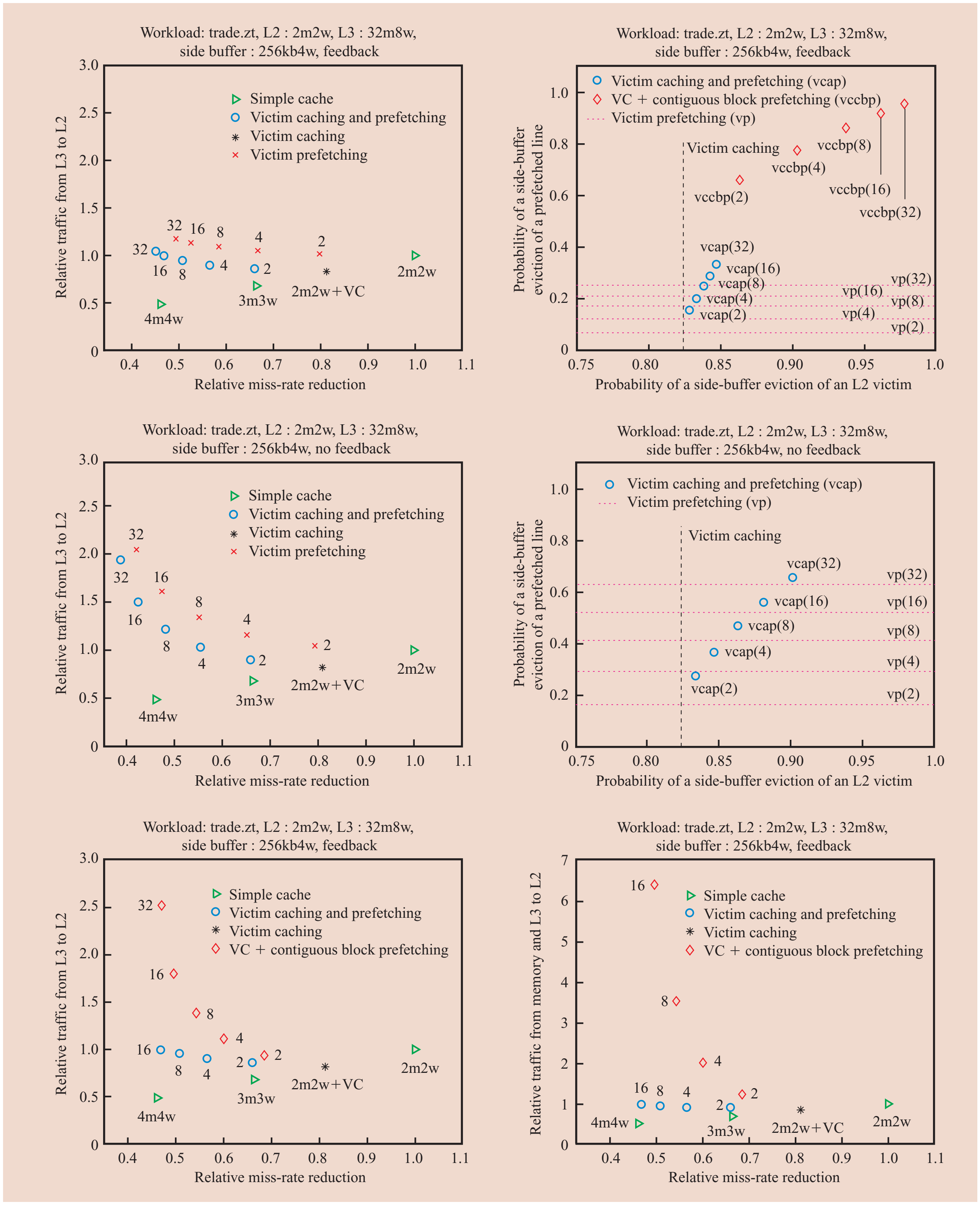

Figure 10

Prefetch performance results for Trade2, presented from left to right and top to bottom in the same order as they were presented for the TPC-C workload. For interpretation of these plots, we refer the reader to the main text of the paper.

- L2 load misses.
- Read-with-intent-to-modify (rwitm) events. These are placed on the bus when a processor store instruction misses in the L2 cache. The associated line is placed in exclusive state in the L2 cache. The store then takes place and the modified flag of the line is set.
- L2 cast-outs of modified lines.
- I/O DMA.

Note that for a given set from the direct-mapped L2 cache, after the first L2 load miss mapped to the set appears on the bus, one can determine the address of the line held in the set after the miss is serviced from memory. Every subsequent processor load to the same line that misses the L1 is not observed, but these load misses, if any, obviously do not change the contents and identity of the line stored in the set. Nevertheless, it is possible that a processor store instruction to the same line may be executed that may not result in a bus rwitm event, because the associated line may already be held in exclusive state. These store instructions are also immediately invisible to our probing, suggesting that modified bits cannot be accurately tracked under this set of assumptions. Interestingly, this is not the case, as we shall shortly explain.

Continuing our discussion, the next transaction observed for this set in the bus must be either a processor load or store to a different line, but one which maps to the same set.

In either case, the identity of the line that resides in the set after the completion of the load/store is known. Furthermore, in the case of a store it is known that the line is currently modified. The identity of the associated evicted line, if any, is also known; nevertheless, it is not always known whether the evicted line was modified. To learn this, one must examine the next event in the trace mapped to the set in question. If this event is an L2 cast-out of the evicted line, we know that the evicted line was modified. If the next event to the set is to a different line, the evicted line was unmodified. The above discussion illustrates that it is possible to recreate the identities of the lines stored in the 1-MB direct-mapped cache along with their modified status, said status known *a posteriori*, after the L2 modified cast-out, if any, is observed.

For the purpose of cache simulation, it is desirable to postprocess the bus trace in a manner that removes the inconvenient non-causal element exposed above. A solution is to insert a STORE immediately before the load/store instruction that caused the cast-out and to remove the cast-out from the trace. Since it is technically impossible to recover the exact time at which the store happened, the events as ordered in the postprocessed trace do not necessarily correspond to reality. However, the exact location of the store is not necessary, since the resulting effect on the LRU information of the simulated cache will be identical because the cache is direct-mapped. This means that there is no other access to another cache line for this set, since the line was brought into the cache. In rebuilding the contents of the direct-mapped cache, the modified bit will be set by the time the offending load/rwitm is received, and thus the L2 cast-out event can be reproduced.

We now turn to the problem of building the contents of the directory of a hypothetical larger cache. If one assumes that the larger cache employs an LRU rule and that the number of sets remains unchanged (but the associativity is increased, of course), the task of constructing the MRU position of each set has already been accomplished. However, the second MRU can be obtained by tracking ejections or modified cast-outs of the MRU and so forth, and the entire cache contents can be built this way. A specific set of sufficient conditions to be met for the simulation of larger caches to be successful was given in Section 5.

Acknowledgment
This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. NBCH3039004.

*Trademark, service mark, or registered trademark of International Business Machines Corporation.

**Trademark, service mark, or registered trademark of Transaction Processing Performance Council or Sun Microsystems, Inc.

References
1. P. G. Emma, A. Hartstein, T. R. Puzak, and V. Srinivasan, "Exploring the Limits of Prefetching," *IBM J. Res. & Dev.* **49**, No. 1, 127–144 (2005).
2. P. A. Franaszek and B. T. Bennett, "Adaptive Variation of the Transfer Unit in a Storage Hierarchy," *IBM J. Res. & Dev.* **22**, No. 4, 405–412 (1978).
3. P. Van Vleet, E. Anderson, L. Brown, J. L. Baer, and A. Karlin, "Pursuing the Performance Potential of Dynamic Cache Line Sizes," *Proceedings of the International Conference on Computer Design,* 1999, pp. 528–537.
4. T. Alexander and G. Kedem, "Distributed Prefetch-Buffer/Cache Design for High Performance Memory Systems," *Proceedings of the 2nd International Symposium on High-Performance Computer Architecture,* 1996, pp. 254–263.
5. M. J. Charney and T. R. Puzak, "Prefetching and Memory System Behavior of the SPEC95 Benchmark Suite," *IBM J. Res. & Dev.* **41**, No. 3, 265–286 (1997).
6. T. L. Johnson and W. W. Hwu, "Run-Time Adaptive Cache Hierarchy Management via Reference Analysis," *Proceedings of the 24th International Symposium on Computer Architecture,* 1997, pp. 315–326.
7. S. Kumar and C. Wilkerson, "Exploiting Spatial Locality in Data Caches Using Spatial Footprints," *Proceedings of the 25th Annual International Symposium on Computer Architecture,* 1998, pp. 357–368.
8. C. B. Wilkerson and S. Kumar, "Spatial Footprint Prediction," U.S. Patent 6,535,961, B2, March 18, 2003.
9. D. Burger, "Hardware Techniques to Improve the Performance of the Processor/Memory Interface," Technical Report, Computer Science Department, University of Wisconsin at Madison, 1998.

10. H. Yu and G. Kedem, "DRAM-Page Based Prediction and Prefetching," *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 2000, p. 267.
11. G. Kedem, R. Ronen, and A. Yoaz, "Method and Apparatus for Cache Line Prediction and Prefetching Using a Prefetch Controller and Buffer and Access History," U.S. Patent 6,134,643, October 17, 2000.
12. A.-C. Lai, C. Fide, and B. Falsafi, "Dead-Block Prediction and Dead-Block Correlating Prefetchers," *Proceedings of the 28th Annual International Symposium on Computer Architecture,* 2001, pp. 52–62.
13. W.-F. Lin, S. K. Reinhardt, D. Burger, and T. R. Puzak, "Filtering Superfluous Prefetches Using Density Vectors," *Proceedings of the IEEE International Conference on Computer Design,* 2001, pp. 124–132.
14. W.-F. Lin, S. K. Reinhardt, and D. Burger, "Reducing DRAM Latencies with an Integrated Memory Hierarchy Design," *Proceedings of the 7th International Symposium on High-Performance Computer Architecture,* 2001, pp. 301–312.
15. O. Temam, "An Algorithm for Optimally Exploiting Spatial and Temporal Locality in Upper Memory Levels," *IEEE Trans. Computers,* **48**, No. 2, 150–158 (1999).
16. J. H. Moreno, J. A. Rivers, and J. D. Wellman, "Method and Apparatus for Memory Prefetching Based on Intra-Page Usage History," U.S. Patent 6,678,795, January 13, 2004.
17. Z. Hu, M. Martonosi, and S. Kaxiras, "TCP: Tag Correlating Prefetchers," *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture,* 2003, pp. 317–326.
18. A. Moshovos, "RegionScout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence," *Proceedings of the 32nd Annual International Symposium on Computer Architecture,* 2005, pp. 234–245.
19. J. F. Cantin, M. H. Lipasti, and J. E. Smith, "Improving Multiprocessor Performance with Coarse-Grain Coherence Tracking," *Proceedings of the 32nd Annual International Symposium on Computer Architecture,* 2005, pp. 246–257.
20. N. P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," *Proceedings of the 17th International Symposium on Computer Architecture,* 1990, pp. 364–373.

Received August 10, 2005; accepted for publication February 23, 2006; Internet publication August 8, 2006

Peter A. Franaszek *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (paf@us.ibm.com).* Dr. Franaszek received the Ph.D. degree in electrical engineering from Princeton University in 1965. From 1965 to 1968, he was employed by Bell Laboratories; he joined the IBM Research Division in 1968. During the academic year 1973–1974, he was on sabbatical leave at Stanford University as Consulting Associate Professor of Computer Science and Electrical Engineering. His interests are in the general area of information representation and management, and computer system organization. Dr. Franaszek has received two IBM Corporate Awards for his work on codes for magnetic recording, an IBM Corporate Patent Portfolio Award for his contribution to the ESCON* architecture, and IBM Outstanding Innovation Awards for fragmentation-reduction algorithms, for network theory, for concurrency-control algorithms, for run-length-limited codes, for the 8B/10B code used in ESCON, Fibre Channel, and Gigabit Ethernet, and for compressed-memory machines. He is a member of the IBM Academy of Technology and a Master Inventor. He is a Fellow of the IEEE, and received the 1989 IEEE Emmanuel R. Piore Award for his contributions to the theory and practice of constrained channel coding in digital recording. In 2003, he received the ACM Paris Kanellakis Theory and Practice Award for his contributions to the theory and practice of such coding. Dr. Franaszek holds more than fifty patents and has published more than forty-five technical papers.

Luis A. Lastras-Montaño *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (lastrasl@us.ibm.com).* Dr. Lastras-Montaño received the Ph.D. degree in electrical engineering from Cornell University in 2000; he received the B.Sc. degree from the School of Sciences (UASLP, Mexico). He joined the IBM Thomas J. Watson Research Center after receiving his graduate degree. His academic interests are in the field of information theory, specifically network lossy and lossless data compression. Other interests include large deviations, statistical inference, communication signal design, and foundational issues at the intersection of information theory and computing systems architecture. At IBM his contributions have included theoretical topics such as the performance analysis of multiple description codes, universal lossless compression algorithms, and non-asymptotic large deviations theory, as well as practical topics such as algorithms for low-complexity lossless compression, prefetching in microarchitectures, and the design and analysis of compressed memory systems.

Steven R. Kunkel *IBM Systems and Technology Group, 3605 Hwy. 52 N, Rochester, Minnesota 55901 (srkunkel@us.ibm.com).* Dr. Kunkel received his Ph.D. degree from the University of Wisconsin at Madison in 1987. He then joined IBM in Endicott, New York, doing performance analysis of a vector facility for a mid-range System/390* product. In 1989, he transferred to the IBM Rochester, Minnesota, site where he currently works. During most of his years in Rochester, he did architecture and performance analysis for AS/400* and RS/6000* (now called iSeries* and pSeries*) products. This included such areas as NUMA, VLIW, caches, MP cache coherency, multithreading, and converting AS/400 to PowerPC* architecture processors. Dr. Kunkel is currently a Senior Technical Staff Member; he continues to do architecture and performance analysis for iSeries, pSeries, and zSeries* servers.

Aaron C. Sawdey *IBM Systems and Technology Group, 3605 Hwy. 52 N, Rochester, Minnesota 55901 (sawdey@us.ibm.com).* Dr. Sawdey received his Ph.D. degree from the University of Minnesota in 1997. From 1997 to 1999 he worked for SGI/Cray in

Eagan, Minnesota, on debuggers, application performance analysis software, and parallel processing libraries. In 1999 he joined IBM in Rochester, Minnesota, where he does cache and SMP interconnect analysis for PowerPC processors used in the iSeries and pSeries products.