A Prefetch Taxonomy

Viji Srinivasan, *Member*, *IEEE*, Edward S. Davidson, *Fellow*, *IEEE*, and
Gary S. Tyson, *Member*, *IEEE*

Abstract—The growing difference between processor and main memory cycle time demands the use of aggressive prefetch algorithms to reduce the effective memory access latency. However, prefetching can significantly increase memory traffic and unsuccessful prefetches may pollute the cache. Metrics such as coverage and accuracy result from a simplistic classification of individual prefetches as "good" or "bad." They do not capture the full effect of each prefetch and, hence, do not accurately reflect the quality of the prefetch algorithm. Gross statistics such as changes in the number of misses, total traffic, and IPC are not attributable to individual prefetches. Such gross metrics are therefore useful only for ranking existing prefetch algorithms; they do not evaluate the effect of individual prefetches so that an algorithm might be tuned. In this paper, we introduce a new, accurate, and complete taxonomy, called the Prefetch Traffic and Miss Taxonomy (PTMT), for classifying each prefetch by precisely accounting for the difference in traffic and misses it generates, either directly or indirectly. We illustrate the use of PTMT by evaluating two data prefetch algorithms.

Index Terms—Prefetch algorithms, cache memory systems.

1 Introduction

Prefetching reduces the effective memory access time by fetching lines into cache before they are demanded by a reference. In theory, we could prefetch each missing cache line so that it arrives in cache just before its next reference. However, in reality, aggressive prefetch algorithms ([3], [8], [9], [13], [14], [15], [1], [5], [7], [10], [11], [12], [16], [17], [18], [19]) determine the addresses for prefetching speculatively. Hence, the following issues arise:

- Some addresses are not accurately predicted, thereby limiting the effectiveness of the speculative prefetch algorithm by increasing memory traffic without benefit, or worse by polluting the cache by replacing desirable lines with unwanted lines.
- Even when its address is accurate, a prefetch may not be issued early enough to cover the full nominal miss (or "memory") latency, let alone any additional delays above the nominal that may arise due to limited available bandwidth to memory or other contention effects.
- A prefetch may be issued too early and pollute the cache by replacing a line that will be referenced before the prefetched line; furthermore, the prefetched line may be replaced before it is referenced.

Five metrics commonly used to evaluate the effectiveness of a prefetch algorithm consist of three gross measurements of the achieved performance (number of misses, total traffic, and instructions per cycle (IPC)) and two metrics (coverage and accuracy) which are attributable to individual prefetches. Specifically, each prefetch is broadly classified as *good* if the prefetched line is referenced by the application before it is replaced or *bad* otherwise. If a conventional cache has M misses without using any prefetch algorithm, the prefetch coverage and accuracy of a given prefetch algorithm that yields G good prefetches and B bad prefetches are calculated as:

$$\mathrm{Coverage} = \mathrm{G/M}; \quad \mathrm{Accuracy} = \mathrm{G/(G+B)}.$$

Effective prefetching is said to depend on achieving good miss coverage (implicitly assuming that each good prefetch eliminates one miss) with sufficient accuracy to avoid saturating the memory bus with useless prefetches and polluting the cache (implicitly assuming that a bad prefetch is useless or polluting, but a good prefetch is not).

The simple example in Fig. 1 shows that these implicit assumptions are not valid and that even a very poor prefetch algorithm can have both high coverage and high accuracy. Consider a set, s, of two identically configured 2-way set associative caches with LRU replacement, one (on the left) using a prefetch algorithm and the other (on the right) being a conventional cache with no prefetching. The reference stream consists of sequence $ywyx$, where w, x, and y are in distinct lines that map to set s; references to other sets may be interspersed.

Fig. 1 shows at (a) the contents of set s after the initial y and w references which set up identical set s contents and replacement stacks in both caches and likewise at (b) after the algorithm, apparently too early, prefetches x (as is customary in prefetching, a prefetched line entering the cache is designated most recently used (MRU) in its set), at (c) after referencing y again, and, finally, at (d) after referencing x, at which point the set s contents are once again identical.

In this example, the coverage is 100 percent since there was one miss (to x in the cache without prefetching) and

- *V. Srinivasan is with the IBM T.J. Watson Research Center, Yorktown Heights, NY 10598. E-mail: viji@us.ibm.com.*
- *E.S. Davidson and G.S. Tyson are with the Electrical Engineering and Computer Science Department, University of Michigan, Ann Arbor, MI 48109. E-mail: {davidson, tyson}@eecs.umich.edu.*

Manuscript received 26 Oct. 2001; revised 19 Sept. 2002; accepted 22 Apr. 2003.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 115265.

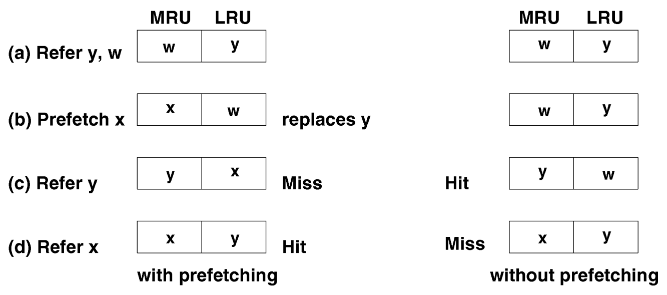


Fig. 1. Cache access outcomes for the reference sequence to set $s = \ldots y \ldots w \ldots y \ldots x$, with and without prefetching x after referencing w.

there is one good prefetch (of line x). The accuracy is also 100 percent (no bad prefetches). But, the number of misses is unchanged (although the good prefetch does eliminate the miss to x, the replaced line y now has a miss) and the cache with prefetching has one more line of traffic than the cache without prefetching. This simple example illustrates that, even a prefetch algorithm that appears to be perfect, with 100 percent coverage and 100 percent accuracy, may actually be worse than doing no prefetching at all. Note further that, if y were prefetched back into the cache between (b) and (c), we would also eliminate the miss to y and would have two good prefetches, resulting in 200 percent coverage. The absurd possibility of coverage greater than 1 was also mentioned in [6], which observed that such an extreme case does not actually occur in practice since, in a full application, there are many misses that a prefetch algorithm will not attempt to cover.

Coverage and accuracy are thus simplistic and misleading estimators of miss reduction and extra traffic; secondary effects of prefetching on misses and traffic are critical to the design of an efficient prefetch algorithm. Gross statistics, like the net change in misses, traffic, and IPC do not attribute those changes to the effects of particular prefetches; by not illuminating the underlying causes of the net changes, they provide little insight regarding how the performance of a prefetch algorithm might be improved.

A more refined classification of individual prefetches is needed to provide more insight into how a given prefetch algorithm works. In this paper, we address this need by introducing a new, accurate, and complete taxonomy, called the Prefetch Traffic and Miss Taxonomy (PTMT). The key to PTMT is the simultaneous simulation of the same reference stream on a prefetch cache (using some prefetch algorithm of interest) and on an identically configured conventional cache (with no prefetching). Prefetches are not simply classified as good or bad (according to the hit/miss outcome of the prefetched line in the prefetch cache), but as one of nine cases according to the next events associated not just with the prefetched line, but also with the line it replaces in the prefetch cache, as well as the next events associated with those lines in the conventional cache.

Each of these nine cases has a particular benefit and cost that is attributed to each prefetch of that case, namely, a net change of -1, 0, or +1 misses and an additional 0, 1, or 2 lines of traffic, relative to the conventional cache. Furthermore, each case has a unique associated syndrome of events that are the underlying cause of its benefit and cost. PTMT thus precisely accounts for and explains the difference in traffic and misses generated both directly by the prefetched lines themselves and indirectly by the lines they replace. One other effect can occur in associative caches, namely, some additional misses and traffic due to the prefetch-induced side-effect of reordering within the LRU replacement stacks; a 10th case in PTMT captures this side-effect. PTMT's 10 cases are disjoint and complete and their costs and benefits are exact, i.e., the difference in traffic and misses between the cache with prefetching and the conventional (no prefetch) cache is exactly equal to the extra traffic (always nonnegative) and extra misses (hopefully negative) as determined from the 10 case population histogram generated by PTMT. PTMT has proven to be useful in identifying and understanding the specific strengths and weaknesses of existing prefetch algorithms and their underlying causes. Applying PTMT helps answer questions like the following:

- Which prefetches replaced a more desirable cache line?
- Which prefetches actually saved a cache miss?
- What are the best opportunities for improving the performance of this prefetch algorithm?

PTMT thus provides an accurate understanding of the usefulness of each prefetch by precisely quantifying and explaining its direct and indirect contributions to the total change in traffic and misses. Such a classification helps a prefetch algorithm developer to understand and tune the algorithm by focusing on eliminating individual prefetches that degrade the performance or on finding ways to convert them to performance enhancing prefetches.

In the rest of this paper, Section 2 describes PTMT in detail. Section 3 describes the chosen prefetch algorithms and the benchmarks used to evaluate them. Section 4 uses PTMT to analyze the working of a near-optimal (unimplementable) prefetch algorithm [4]. The effectiveness of *Next Sequential Prefetching* (NSP) is analyzed using PTMT in Section 5. We summarize and present conclusions in Section 6.

2 Prefetch Traffic and Miss Taxonomy

The simple example in Fig. 1 showed that a *good* prefetch (x) may replace a useful line (y), thereby saving no misses and eventually incurring one extra line of traffic to refetch the replaced line. We were able to quantify the true and complete effect of each prefetch on the net change in traffic and misses by simultaneously simulating a cache that employs some prefetch algorithm of interest, *pf-cache*, and a cache of the same structure with no prefetching, *conv-cache*, and comparing their behavior. The Prefetch Traffic and Miss Taxonomy (PTMT) uses such a comparison to classify each prefetch.

2.1 The PTMT Cache Model

There is no restriction on the size, associativity, or line size of the *pf-cache* and the *conv-cache*, except that they be the same; both caches are assumed to be write-through (i.e., prefetching can affect write-back traffic, but this is not modeled). Although it is possible to develop such a taxonomy for any replacement policy that preserves some

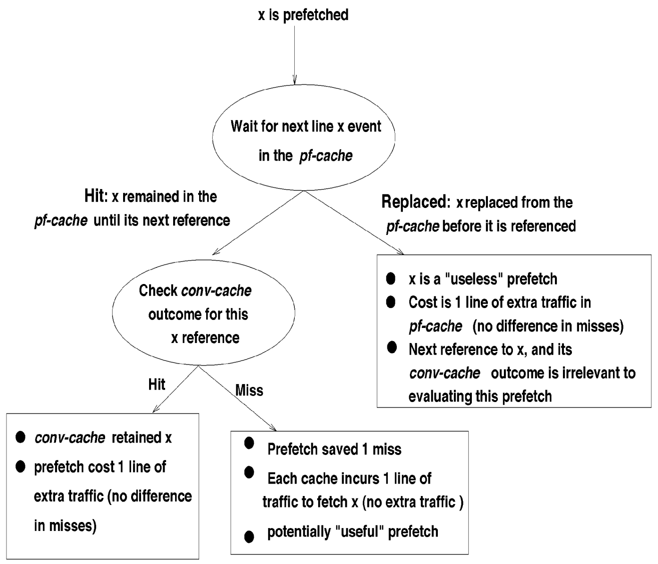

x is prefetched
Wait for next line x event in the pf-cache
Hit: x remained in the pf-cache until its next reference
Replaced: x replaced from the pf-cache before it is referenced
Check conv-cache outcome for this x reference
x is a "useless" prefetch
Cost is 1 line of extra traffic in pf-cache (no difference in misses)
Next reference to x, and its conv-cache outcome is irrelevant to evaluating this prefetch
Hit
Miss
conv-cache retained x
prefetch cost 1 line of extra traffic (no difference in misses)
Prefetch saved 1 miss
Each cache incurs 1 line of traffic to fetch x (no extra traffic)
potentially "useful" prefetch

Fig. 2. Next event on prefetched line *x* in *pf-cache* and in *conv-cache*.

order (i.e., not a random replacement policy) between the *conv-cache* and the *pf-cache*, PTMT is developed for LRU replacement, which is by far the most common replacement strategy in use today. Each set of the cache is implemented within the simulation as an LRU stack. As is commonly done when prefetching, each new line that enters a cache set, whether via a miss or a prefetch, is pushed onto the top of that set's LRU stack (the most recently used position); lines leave the set from the bottom of the stack. For an n-way associative cache, the top of the LRU stack is referred to as position *0*, and the bottom as $(n-1)$. As our goal is to study the effect of prefetching on the cache performance, we prefetch directly into the *pf-cache* and do not consider using a separate prefetch buffer [12]. Moreover, a prefetch buffer is wasteful because the better prefetch algorithms do not cause much cache pollution, even without a buffer. Finally, a prefetch is squashed if the line is already present in the *pf-cache*; such squashed prefetches do require a lookup in the cache directory, but do not incur any cost or benefit in terms of the number of misses or the number of lines of traffic to memory. Although it would be easy to add, PTMT does not tabulate squashed prefetches.

2.2 Prefetch Classification

The prefetch classification in PTMT is particularly concerned with a pair of events associated with the prefetch of line x: 1) The next event on line x after line x is prefetched and 2) the next event on line y after line y is replaced to accommodate the prefetch of x. These events, their effect on traffic and misses, and the resulting classification of the prefetches in PTMT are discussed in Section 2.2.1. Some types of prefetches lead to a succession of prefetches that are linked in a chain. These chains are discussed in Section 2.2.2. Finally, for associative caches, prefetches do have an indirect effect on traffic and misses that is due to the LRU stack reordering that is induced as a side-effect of the prefetching. Side-effect misses and traffic, which are not attributed to particular prefetches, complete the taxonomy and are discussed in Section 2.2.3. Note that PTMT focuses only on the effect of individual prefetches on traffic and misses and does not directly address the issue of prefetch timeliness.

TABLE 1
Cost of the Next Event on Prefetched Line *x* in *pf-cache* and in *conv-cache*

Next event on x in		**Extra**	
pf-cache	*conv-cache*	**Traffic**	**Misses**
hit	hit	1	0
hit	miss	0	-1
replaced	don't care	1	0

2.2.1 Events Associated With a Prefetch

Suppose line x is prefetched into the *pf-cache* and replaces line y. In the *pf-cache*, there are only two possible next events on line x: a hit, i.e., x remains in the *pf-cache* until its next reference, or a replacement, i.e., x is replaced in the *pf-cache* before its next reference. In the *conv-cache*, that same next reference to line x is either a hit or a miss. Fig. 2 shows all the possible next events on prefetched line x in the *pf-cache* and in the *conv-cache*. Table 1 summarizes the extra traffic and misses associated with each combination.

From Fig. 2, we observe that, when the next reference to line x is a hit in the *pf-cache*, the following two possibilities arise:

- *hit-hit*: The next reference to line x after the prefetch of x is a hit in both caches. This implies that 1) x was retained in the *conv-cache* from its prior reference until that next reference and 2) although x was replaced in the *pf-cache* sometime after that prior reference, x was subsequently prefetched back into the *pf-cache* prior to its next reference and remained in the *pf-cache* until that next reference. In this set of events, the *pf-cache* incurs one cache line of traffic to prefetch x, whereas the *conv-cache* incurs none, i.e., there is one extra line of traffic in the *pf-cache*.
- *hit-miss*: The next reference to line x is a hit in the *pf-cache* and a miss in the *conv-cache*. This appears to be a useful prefetch; however, its usefulness depends on the next event on the replaced line y. Note that the *pf-cache* incurs one fewer miss relative to the *conv-cache*; we represent this as -1 extra misses. There is no extra traffic since the line of traffic due to the prefetch of x in the *pf-cache* is offset by the line of traffic due to the miss in the *conv-cache*.

Now, consider the case where x is replaced from the *pf-cache* before being referenced (i.e., the third row of Table 1). The next reference to x will eventually be satisfied in the *pf-cache* either by another prefetch or by a demand miss. If a second or subsequent prefetch of x (after the replacement of the first prefetched line, but before the next reference to x) results in a hit in the *pf-cache* at that next reference to x, then the outcome of that next reference to x will be charged to that subsequent prefetch of x, not to the current x prefetch. If, on the other hand, that next reference to x is a miss in the *pf-cache*, that outcome is the same as if the current x prefetch were simply not done, so again, in

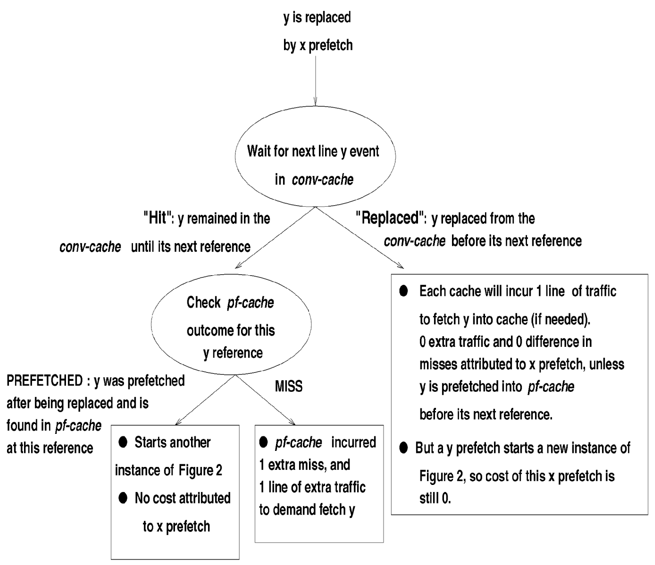

y is replaced
by x prefetch
Wait for next line y event
in conv-cache
"Hit": y remained in the
conv-cache until its next reference
"Replaced": y replaced from the
conv-cache before its next reference
Check pf-cache
outcome for this
y reference
PREFETCHED : y was prefetched
after being replaced and is
found in pf-cache
at this reference
MISS
● Starts another
instance of Figure 2
● No cost attributed
to x prefetch
● pf-cache incurred
1 extra miss, and
1 line of extra traffic
to demand fetch y
● Each cache will incur 1 line of traffic
to fetch y into cache (if needed).
0 extra traffic and 0 difference in
misses attributed to x prefetch, unless
y is prefetched into pf-cache
before its next reference.
● But a y prefetch starts a new instance of
Figure 2, so cost of this x prefetch is
still 0.

Fig. 3. Next event on replaced line *y* in *pf-cache* and in *conv-cache*.

this case, that outcome should not be charged to the current *x* prefetch. Since the *pf-cache* outcome is not charged to the current *x* prefetch (regardless of whether it is ultimately a miss or a hit due to some subsequent prefetch), the outcome of that next reference in the *conv-cache* is likewise not charged to the current prefetch. The cache tour associated with the current *x* prefetch is thus complete as soon as *x* is replaced in the *pf-cache*; we classify this *x* prefetch as *replaced* and charge it with the one line of traffic incurred for prefetching *x*. Note that this one line of traffic is incurred regardless of the presence or absence of *x* in the *conv-cache*. Thus, when line *x* is prefetched and replaced before being referenced, the interval of events on *x* charged to this prefetch begins at the time of prefetch and ends at the replacement of the prefetched line. As there are no events on *x* in the *conv-cache* during this interval, the *conv-cache* outcome of the next event to *x* is not within this interval and is thus not charged to this prefetch of *x*. Hence, when *x* is replaced from the *pf-cache* before its next reference, we refer to the outcome of the *x* reference in the *conv-cache* as *don't care* and the resulting event combination is called *replaced-don't care*. The only cost associated with such a prefetch is thus one line of extra traffic (due to the prefetch) and no change in misses since there are no demand references to *x* during the interval whose *x* event outcomes are charged to this prefetch.

Now, consider line *y* (the line that the *x* prefetch replaced in the *pf-cache*). We focus on the possible next events on *y* in the *conv-cache* and the corresponding events on *y* in the *pf-cache*. Line *y* either remains in the *conv-cache* until its next reference (resulting in a *hit*) or is replaced there before its next reference (resulting in a *miss*). In the *pf-cache*, either line *y* results in a miss at its next reference or it results in a hit by being prefetched back into the *pf-cache* at some time before its next reference and remaining there until that next reference. Fig. 3 shows all the possible next relevant events on replaced line *y* in the *pf-cache* and in the *conv-cache*; Table 2 summarizes the extra traffic and misses associated with each combination.

TABLE 2
Cost of the Next Event on Replaced Line *y* in *pf-cache* and in *conv-cache*

Events for y in		**Extra**	
pf-cache	*conv-cache*	**Traffic**	**Misses**
miss	hit	1	1
prefetched	hit	0	0
don't care	replaced	0	0

Note that, regardless of whether *y* results in a *pf-cache* miss or hit at its next reference, there may be some intervening events that are irrelevant to this case. Namely, between the prefetch of *x* that replaces *y* and the next reference to *y*, *y* may be prefetched and replaced in the *pf-cache* any number of times. However, all such useless *y* prefetches, if any, will be independently classified and charged as one line of extra traffic, as shown in row 3 of Table 1; none of those useless prefetches affect the cost of this *x* prefetch.

From Fig. 3, we observe that, when the next reference to line *y* is a hit in the *conv-cache*, the following possibilities arise:

- *miss-hit*: The next reference to line *y* is a hit in the *conv-cache*, but a miss in the *pf-cache* due to the replacement of *y* in the *pf-cache* by the prefetch of *x*.
- *prefetched-hit*: The next reference to line *y* is a hit in the *conv-cache* as well as the *pf-cache*. Since the next reference to *y* is a hit in the *pf-cache*, *y* must have been prefetched back into the *pf-cache* at some time before this next reference and remained there until this next reference. As shown in row 1 of Table 1, that *y* prefetch will be charged with the traffic for bringing *y* back into the *pf-cache*; it is important not to double count that traffic by also charging it to this *x* prefetch. Consequently, in Table 2, no additional *y* traffic is charged to this *x* prefetch as that traffic, as well as the outcome for *y* at its next reference, will be charged to some later *y* prefetch. To stress this relationship to a later prefetch, this case is called *prefetched-hit* in Table 2.

We now consider the case where *y* is replaced from the *conv-cache* before its next reference (i.e., the third row in Table 2). In this case, the next reference to *y* in the *conv-cache* will be a miss, causing a demand fetch from the next level of the memory hierarchy. The next reference to *y* in the *pf-cache* may be satisfied either by a successful prefetch or by a demand fetch from the next level of the memory hierarchy, which we denote as "*don't care*" since, in either case, no *y* cost in traffic or misses is charged to this *x* prefetch. In particular, if *y* is prefetched back into the *pf-cache*, the next reference to that prefetched line will fall into one of the three cases in Table 1 and the cost of that *y* prefetch must not be charged to this *x* prefetch. On the other hand, if the next reference to *y* is a miss in the *pf-cache*, each cache will incur a miss and one line of traffic to demand fetch *y* and there is no net difference in misses and traffic that needs to be charged to this *x* prefetch. Hence, the only possible combination when *y* is replaced from the *conv-cache* is *don't*

TABLE 3
Cost in Traffic and Misses for the Nine (x, y) Case Pairs

Cases	*pf-cache* outcomes		*conv-cache* outcomes		Extra	
	x (prefetched)	y (replaced)	x (prefetched)	y (replaced)	Traffic	Misses
1	hit	miss	hit	hit	2	1
2	hit	prefetched	hit	hit	1	0
3	hit	don't care	hit	replaced	1	0
4	hit	miss	miss	hit	1	0
5	hit	prefetched	miss	hit	0	-1
6	hit	don't care	miss	replaced	0	-1
7	replaced	miss	don't care	hit	2	1
8	replaced	prefetched	don't care	hit	1	0
9	replaced	don't care	don't care	replaced	1	0

care-replaced, with no extra y traffic or misses charged to this x prefetch.

To quantify the extra traffic and misses incurred due to the x prefetch we combine the three cases in Table 1 with the three cases in Table 2 in all possible ways. The nine combined cases are presented in Table 3. Note that, in each case, the increase in traffic is one more than the change in misses.

2.2.2 Prefetch Chains

From Table 3, we observe that cases 2, 5, and 8 list "prefetched" as the *pf-cache* outcome for the replaced line y, indicating that the line that was replaced by the prefetch is subsequently prefetched into the *pf-cache* before its next reference and is found in *pf-cache* at that next reference. Thus, cases 2, 5, and 8 must be chained to some next case in a chain of prefetches and their cost and benefit is best understood by considering the entire interrelated chain of prefetches that is started by one of these cases. Note, however, that, for cases 3, 6, and 9, the replaced line, y, may or may not be prefetched back into the *pf-cache* before its next reference, but cases 3, 6, and 9 do not require such a prefetch, so no chain of interdependent prefetches is started by case 3, 6, or 9. Furthermore, for any case (including 1, 4, and 7), y may be prefetched back into the *pf-cache* and replaced any number of times before its next reference, but the original x prefetch clearly does not depend on such useless prefetches and, thus, does not chain to them.

Fig. 4 illustrates the occurrence of a prefetch chain in a 2-way set associative cache. As in Fig. 1, we focus only on one set of these caches and that set has the same contents in both caches at (a) and again at (e). Suppose the *pf-cache* then prefetches m at (b) and prefetches n at (c). The program then references line n at (d). Since this reference to n is a hit in both the *pf-cache* and the *conv-cache* and since it is the next reference to n for the (n, p) prefetch, (n, p) can only be case 1, 2, or 3. Finally, m is referenced at (e). The hit-miss outcome of this reference to m means that the (m, n) prefetch at (b) must be case 4, 5, or 6; however, when combined with the fact that n is prefetched back into the *pf-cache* at (c), which results in a hit-hit outcome of the reference to n at (d), we see that (m, n) must be a case 5 prefetch. Since p is replaced in the *conv-cache* by the reference to m at (e), the next reference to p must be a miss in the *conv-cache*; hence, the (n, p) prefetch must be case 3. Fig. 4 thus shows an example of a prefetch chain that begins with a case 5 prefetch, followed by a case 3 prefetch. Note from Fig. 4 that the *conv-cache* experiences one miss resulting in one line of traffic, and the *pf-cache* experiences two prefetches and two hits, resulting in two lines of traffic and no misses, for a net savings of one miss at the cost of one extra line of traffic, just as calculated by adding up the cost of a case 5 and a case 3 prefetch from Table 3.

As explained in Section 2.2.1, n could be prefetched multiple times between the m prefetch at (b) and the next reference to n at (d). However, all but the last of those prefetches would result in replacements of n before that

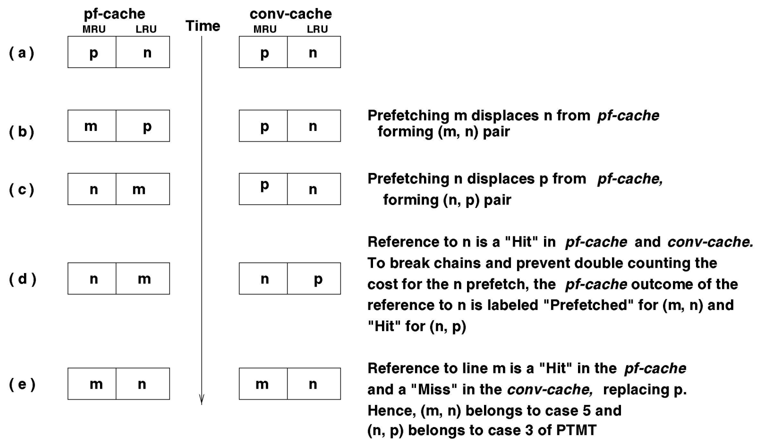

pf-cache
MRU
LRU
Time
conv-cache
MRU
LRU
(a)
p
n
p
n
(b)
m
p
p
n
Prefetching m displaces n from pf-cache forming (m, n) pair
(c)
n
m
p
n
Prefetching n displaces p from pf-cache, forming (n, p) pair
(d)
n
m
n
p
Reference to n is a "Hit" in pf-cache and conv-cache. To break chains and prevent double counting the cost for the n prefetch, the pf-cache outcome of the reference to n is labeled "Prefetched" for (m, n) and "Hit" for (n, p)
(e)
m
n
m
n
Reference to line m is a "Hit" in the pf-cache and a "Miss" in the conv-cache, replacing p. Hence, (m, n) belongs to case 5 and (n, p) belongs to case 3 of PTMT

Fig. 4. Prefetch chain example.

next reference (otherwise, there would be no later prefetch of *n* before its next reference). As the construct in Fig. 3 shows, only the last of these prefetches (the one that causes the *pf-cache* to hit at the next reference to *n*) is chained to the *m* prefetch at (b). The other intervening *n* prefetches, if any, would be independently classified as case 7, 8, or 9 of Table 3 and would not be chained to this (*m, n*) pair.

Thus, a prefetch belonging to case 2, 5, or 8 is part of a chain of at least two prefetches. As noted in the discussion of the example above, the next prefetch in the chain must be case 1, 2, or 3. Moreover, these prefetch chains can terminate only with a case 1 or a case 3 prefetch because a case 2 prefetch also has a *y* outcome of "prefetched" in the *pf-cache* and thus continues the chain. Therefore, every chain begins with a case 2, 5, or 8 prefetch, which is followed by zero or more occurrences of case 2 prefetches, and is finally terminated with a case 1 or a case 3 prefetch. The set of all possible prefetch chains can thus be denoted by the regular expression $(2,5,8)(2)^*(1,3)$. Note that, in a prefetch chain, the line replaced by one prefetch is prefetched back by the next prefetch of the chain before its next reference and that prefetch causes the line to experience a hit in the *pf-cache* at that next reference.

In order to make an accurate and disjoint assignment to each prefetch of its effect on traffic and misses, PTMT breaks these prefetch chains by classifying the *n* outcome of each (*m, n*) pair in a chain (except for the last) as *prefetched* in the *pf-cache* and by not attributing any line *n* traffic or line *n* misses to that (*m, n*) pair. Therefore, while evaluating the usefulness of a case 2, 5, or 8 prefetch, it is important to take into account the overall cost of the entire prefetch chain. Since a prefetch belonging to case 2 or 8 already incurs extra traffic without saving a miss and since the following case 1, 2, and 3 prefetches only add more traffic and possibly one additional miss to the chain, chains starting with case 2 or case 8 prefetches (i.e., chains of the form $(2,8)(2)^*(1,3)$) will always increase memory traffic and will never decrease misses. Thus, a useful prefetch chain must start with a case 5 prefetch.

Now, consider chains that start with a case 5 prefetch, i.e., chains of the form $5(2)^*(1,3)$. Note that each case 2 or case 3 prefetch costs one additional line of traffic without saving a miss, while a case 1 prefetch costs two additional lines of traffic and causes one additional miss. Chains of the form $5\ (2)^*\ 3$ may or may not be useful depending on the relative importance of traffic and misses. A case 5 prefetch followed by n (≥ 0) case 2 prefetches and then a chain-ending case 3 prefetch causes the *pf-cache* to incur a total cost of $(n+1)$ lines of extra traffic in order to save one miss. This chain thus becomes progressively less desirable and, eventually, undesirable, as n grows. Similarly, a $5\ (2)^*\ 1$ chain incurs a total cost of $(n+2)$ lines of extra traffic without saving a miss; chains of the form $5\ (2)^*\ 1$ are clearly undesirable. A case 5 prefetch may thus have a positive benefit, but only if the chain that it begins is not too long and ends in a case 3 prefetch.

2.2.3 Side-Effects of Prefetching

The nine prefetch cases shown in Table 3 completely quantify the extra traffic in prefetched lines as well as the extra traffic and misses associated with the next reference to each prefetched line and the next reference to the line it replaces. Furthermore, these nine cases disjointly attribute this extra traffic and misses to particular prefetches. It remains to analyze references to lines that are not explicitly associated with a prefetch as above. To distinguish lines not associated with a prefetch from those that are associated with a prefetch, we classify the lines in the cache as either *prefetched* or *regular* lines. A line that has been brought to the cache by a prefetch (rather than by a demand fetch from the next level of memory hierarchy) is called a *prefetched* line from the time that it is prefetched to the cache until it is either referenced or replaced. Once a prefetched line is referenced before being replaced, it is no longer considered to be a prefetched line and is called a *regular* line from immediately after such a reference until it is replaced; all lines brought to the cache by a demand fetch from the next level of memory hierarchy are also *regular* lines.

TABLE 4
Outcomes for a Reference to a Line *z* That Is Not Associated with a Prefetch

Outcome for reference to z in		Category	Resolution
pf-cache	*conv-cache*		
Hit	Hit	*PfHit-ConvHit*	OK
Miss	Miss	*PfMiss-ConvMiss*	OK
Miss	Hit	*PfMiss-ConvHit*	PTMT case 10
Hit	Miss	*PfHit-ConvMiss*	Impossible

In Section 2.2.1, we derived the change in traffic and misses due to prefetched lines and the lines they replaced. But, in an associative cache (associativity > 1), there may be other references with altered outcomes due to prefetching. In associative caches with LRU replacement, prefetching has the side-effect of reordering the LRU stack of any set in which prefetching occurs. In this section, we derive the extra traffic and misses due to this side-effect.

Table 4 lists the possible outcomes in the *pf-cache* and the *conv-cache* of a reference to a line, *z*, that is not associated with a prefetch, i.e., line *z* is neither a prefetched line in the cache nor did it end its most recent tour in the *pf-cache* by being replaced to accommodate a prefetch. We use the labels *PfHit*, *PfMiss*, *ConvHit*, and *ConvMiss* to refer to the results of a reference to *z*, namely, a hit or miss in the *pf-cache* or the *conv-cache*, respectively. Thus, *PfHit-ConvHit* refers to a reference to *z* that resulted in a hit in both caches.

PfHit-ConvHit and *PfMiss-ConvMiss* references clearly incur no additional traffic or misses relative to the *conv-cache*. If only these two cases could occur for lines not associated with prefetches, then the nine case pairs of Table 3 would fully account for the cost in traffic and misses of any prefetch algorithm. Unfortunately, this is not the case; we now show that a reference to *z* can belong to the *PfMiss-ConvHit* category. However, this can easily be detected. The costs associated with *PfMiss-ConvHit* are then incorporated into the taxonomy as case 10. Finally, we show that a reference to *z* cannot belong to *PfHit-ConvMiss* category unless *z* is a prefetched line, in which case, Table 3 already accounts for the extra traffic and misses.

Now, consider the *PfMiss-ConvHit* category. This case is possible only when, prior to this reference to *z*, the

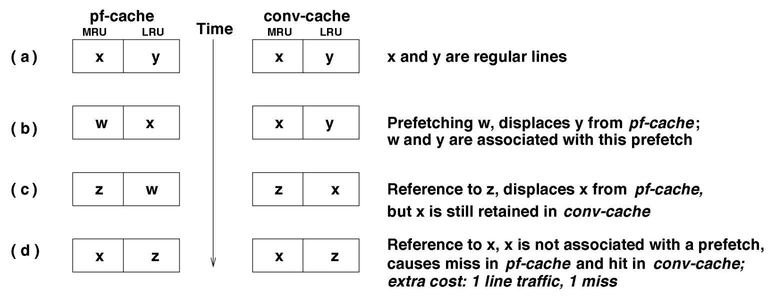

pf-cache
MRU LRU
Time
conv-cache
MRU LRU
(a) x y x y
x and y are regular lines
(b) w x x y
Prefetching w, displaces y from pf-cache; w and y are associated with this prefetch
(c) z w z x
Reference to z, displaces x from pf-cache, but x is still retained in conv-cache
(d) x z x z
Reference to x, x is not associated with a prefetch, causes miss in pf-cache and hit in conv-cache; extra cost: 1 line traffic, 1 miss

Fig. 5. Prefetch side-effect example.

following events occurred at some time after the last reference to z:

1. Since this reference to z is a miss in the *pf-cache*, z must have been replaced from the *pf-cache*. Note that z could not have been replaced to accommodate a prefetch because z would then have been associated with that prefetch and already considered in Section 2.2.1. Therefore, the only possibility here is that z was replaced from the *pf-cache* to accommodate the demand fetch of a line from the next level of the memory hierarchy.
2. Since this reference to z is a hit in the *conv-cache*, z must have been retained in the *conv-cache* until this reference.

This reference to z costs the *pf-cache* one additional miss and one cache line of additional traffic relative to the *conv-cache*. Fig. 5 shows a simple example of the occurrence of this case with a 2-way set associative cache. Again we focus on one set and the contents of the two caches are the same at (a) and again at (d). The traffic and misses resulting from the prefetch of w at (b) are accounted for by one of the nine cases in Table 3. The miss reference to line z at (c) replaces different lines in the two caches. Now, there is a reference to line x which PTMT has not associated with any prefetch. The x reference misses in the *pf-cache* and hits in the *conv-cache*, causing one cache line of additional traffic and one extra miss, relative to the *conv-cache*. We refer to this new case, case 10, as a *Side-effect* of prefetching. Note that an ideal prefetch algorithm would have prefetched line z rather than line w at (b), which would have avoided the occurrence of case 10 in this example. In more complex examples, it is difficult to associate an occurrence of case 10 with a particular prefetch.

In general, an occurrence of case 10 is detected by noting when:

- There is a demand fetch from the next level of memory hierarchy (i.e., a miss) in the *pf-cache* (e.g., the demand fetch of z at Fig. 5c), *and*
- the line replaced in the *pf-cache*(e.g., x at Fig. 5c) is missing in the *pf-cache* at its next reference, but remains in the *conv-cache* until its next reference.

Note that, in a direct-mapped cache, each set has only one line; therefore, if a line x being referenced is found to be present in the *conv-cache*, but not in the *pf-cache*, then x must be the most recently referenced line in its set and therefore must have been replaced in the *pf-cache* to accommodate some prefetch that must have occurred after that most recent reference; x would thus be associated with that prefetch and the extra cost of this reference to x would be covered by one of the nine cases of Table 3. Therefore, case 10 cannot occur for direct-mapped caches.

We now analyze the final case of a *PfHit-ConvMiss* reference to line z. We show, using Theorem 1 below, that this case cannot occur if z is a regular line; it can only occur when z is a prefetched line in the *pf-cache*. Thus, all *PfHit-ConvMiss* cases are covered by the first nine PTMT cases. The proof of Theorem 1 depends on Lemmas 1 and 2.

Lemma 1. *For a given set S, the number of regular lines in the pf-cache never exceeds the number of regular lines in the conv-cache.*

Proof. In the *conv-cache*, all the lines of set S are regular. However, in the *pf-cache*, set S might contain some prefetched lines. Since both caches have the same associativity, A, each set of the *pf-cache* has at most A regular lines, whereas each set in the *conv-cache* has exactly A regular lines. □

Lemma 2. *For a regular line, l, in a set, S, of the pf-cache, its LRU replacement stack position, $pf(l)$, is always greater than or equal to its stack position, $conv(l)$, in the conv-cache.*

Proof. Let t_b and t_e be the times at which two successive references to l occur. Note that, whenever line l is referenced, it moves to stack position 0 in both caches, and remains or becomes a regular line in the *pf-cache*. Therefore, the lemma is satisfied at times t_b and t_e. Hence, it remains to show that the lemma is satisfied in the interval (t_b, t_e), referred to as a nonreference interval of l.

Consider the following exhaustive list of possible events in the interval (t_b, t_e):

1. *References to or prefetches of lines in other sets*: These have no effect on the position of l in set S.
2. *A prefetch of a line, m ($\neq l$), in set S*: In this case, the position of line l increases by 1 in the *pf-cache*, while it remains unchanged in the *conv-cache*. Thus, if $pf(l) \geq conv(l)$ prior to this event, $pf(l) > conv(l)$ after this event.
3. *A reference to some line, m ($\neq l$), in set S*: Prior to this reference, let $pf(m)$ and $conv(m)$ be the position of m in the *pf-cache* and the *conv-cache*, respectively. (If line m is absent in either cache, its position in that cache is defined to be A, the associativity of set S.) The following possibilities arise:

a. $pf(m) > pf(l)$ and $conv(m) > conv(l)$: The position of line l is increased by 1 in both caches; hence, $pf(l) \geq conv(l)$ is preserved after this reference to m. (Note that $pf(m)$ and/or $conv(m)$ could be A.)

b. $pf(m) > pf(l)$ and $conv(m) < conv(l)$: Since $conv(m) < conv(l)$, line m must have been referenced since the last reference to l at time t_b. Since l is not referenced in the interval (t_b, t_e), $pf(m)$ can be greater than $pf(l)$ only if l was replaced and then prefetched back into the *pf-cache* after the reference to m. In such a case, however, l would be a prefetched line, which contradicts our assumption that l is a regular line. Therefore, this case cannot occur.

c. $pf(m) < pf(l)$ and $conv(m) < conv(l)$: This case does not alter the position of l in either cache; hence, $pf(l) \geq conv(l)$ is preserved after this reference to m.

d. $pf(m) < pf(l)$ and $conv(m) > conv(l)$: Since $conv(m) > conv(l)$, line m has not been referenced since the last reference to line l at time t_b. However, since $pf(m) < pf(l)$, it must be that m was prefetched after time t_b. Thus, m must be a prefetched line in the *pf-cache*. Such a reference to a prefetched line m is the only case in which $conv(l)$ increases relative to $pf(l)$: $conv(l)$ increases by 1 and $pf(l)$ remains unchanged. However, as noted above, this event can occur only after a corresponding prefetch of m after t_b, i.e., a prior occurrence of event 2 above, in which $pf(l)$ was increased by 1 while $conv(l)$ remained unchanged. Therefore, this case can only arise if $pf(l) > conv(l)$. Hence, even though $conv(l)$ is now increased by 1, $pf(l) \geq conv(l)$ is preserved after this reference to m.

4. *A replacement of line l from the pf-cache*: This case occurs whenever $pf(l)$ is incremented to A. For a regular line l, this case occurs always and only as an immediate result of either case 2 with $pf(l) = A - 1$ or case 3.a with $pf(m) = A$ and $pf(l) = A - 1$. Note that the replacement itself does not alter $pf(l)$ and that, as long as $pf(l) \geq conv(l)$, line l cannot be replaced from the *conv-cache* unless it is already missing from the *pf-cache* or is being replaced from the *pf-cache* at that same time.
5. *A prefetch of line l into the pf-cache*: Since a prefetch of l will not be issued if line l is already present in the *pf-cache*, this event can occur only after a prior occurrence of event 4. Event 5 makes l a prefetched line. Thus, line l will be either a prefetched line or not present (i.e., not a regular line) in the *pf-cache* from the first occurrence of case 4 until t_e, causing the lemma to be vacuously satisfied from the time that case 4 occurs until we reach time t_e.

We have shown that each possible type of cache event in the interval (t_b, t_e) either preserves $pf(l) \geq conv(l)$ or causes l not to be a regular line in the *pf-cache* for the remainder of the (t_b, t_e) interval. Hence, the lemma is satisfied at t_b, at t_e, and during the entire nonreference interval (t_b, t_e); it is therefore satisfied at every reference and during every nonreference interval between references, i.e., always. □

TABLE 5
The Prefetch Traffic and Miss Taxonomy (PTMT)

Category	Cases	Extra Traffic	Extra Misses
Useful Prefetches	5, 6	0	-1
Useless Prefetches	2, 3, 4, 8, 9	1	0
Polluting Prefetches	1, 7	2	1
Prefetch Side-effect	10	1	1

Theorem 1. *The next reference to line z is a hit in the pf-cache (PfHit) and a miss in the conv-cache (ConvMiss) only if z is a prefetched line in the pf-cache.*

Proof. Consider a PfHit-ConvMiss reference to line z. Prior to this reference, the stack position of line z in the *pf-cache* must have been less than its position in the *conv-cache* (which is A, by definition, since the reference is a miss in the *conv-cache*). From Lemmas 1 and 2, this is not possible if z is a regular line. Hence, z must be a prefetched line in the *pf-cache*. □

Theorem 1 shows that the *PfHit-ConvMiss* category cannot occur for regular lines. Thus, the *PfMiss-ConvHit* category (case 10) and the nine case pairs of Table 3 constitute a complete set of cases. Table 5 summarizes the cost in traffic and misses for the 10 cases. For each case, except for case 10, the extra traffic is always one more than the extra misses.

2.2.4 Summary of PTMT Observations

In Table 5, a prefetch of x that belongs to case 1 or 7 is called *polluting* because it causes two lines of extra traffic and one additional miss. A prefetch may be a polluting prefetch simply because it was issued too early. If a polluting prefetch of x can be delayed until after the next reference to y, it might do less damage and might become *useful*.

Similarly, a case 2, 3, 4, 8, or 9 prefetch is classified as a *useless* prefetch for one of the following reasons:

1. Case 2 or 3 prefetch: At some time after its last reference, the *pf-cache* replaced x because of contention in the cache set due to prefetching, while the *conv-cache* retained x until its next reference; therefore, the *pf-cache* had to prefetch x, incurring a line of traffic. However, such a useless prefetch differs from a polluting prefetch in that it does not cause an extra miss for the replaced line, y, i.e., y is either prefetched back into the *pf-cache* (case 2) or y is replaced from the *conv-cache* before its next reference (case 3). Although cases 2 and 3 are deemed useless when viewed in isolation, they may be needed to continue or complete a chain that evicted x from the *pf-cache* (as described in Section 2.2.2). Case 3 is more

desirable than case 2 as it completes the chain with minimal extra cost. Note that a case 2 prefetch adds the same cost as case 3, but only continues the chain without completing it. A case 2 prefetch might be converted to a better case if the prefetch can be delayed until after the next reference to y.

2. Case 4 prefetch: Prefetching x saved a miss at the next reference to x, but caused a miss at the next reference to y; furthermore, it costs one line of traffic to do this shift. As with case 2, a case 4 prefetch might be converted to a better case by delaying it until after the next y reference if possible.
3. Case 8 or 9 prefetch: Line x did not remain in the *pf-cache* long enough to be referenced; it is possible that x is not needed by the application (perhaps this was a bad prefetch address prediction) or that x was simply prefetched too early or that x was replaced by some other prefetch that should not have been issued (at least at that time).

Finally, when x belongs to case 5 or 6, it is a *useful* prefetch because it incurs no extra traffic and the *pf-cache* would have incurred a miss had it failed to issue a prefetch for x. Although a case 5 prefetch starts a chain of prefetches, it is still called *useful* because the prefetch algorithm prefetches the replaced line, y, back into the *pf-cache* before its next reference. However, the usefulness of a case 5 prefetch decreases as the length of the chain it starts grows and more and more extra traffic is added for the purpose of saving this one miss. Case 6 is the ideal prefetch case as it prefetches a line that causes a miss in the *conv-cache* and successfully converts that miss to a hit in the *pf-cache*; furthermore, it does no damage since it replaces a line whose next reference is a miss in the *conv-cache*.

The 10 cases of PTMT completely and disjointly account for all the extra traffic and misses of a prefetch algorithm. Equations (1) and (2), which are derived from the costs in Table 5, therefore exactly calculate the number of misses and lines of traffic seen by the *pf-cache*.

$$\begin{aligned}\text{Misses}_{\text{pf cache}} = \text{Misses}_{\text{conv cache}} &- (\#\text{ useful prefetches})\\ &+ (\#\text{ polluting prefetches}) \qquad (1)\\ &+ (\#\text{ prefetch side effects})\end{aligned}$$

$$\begin{aligned}\text{Traffic}_{\text{pf cache}} = \text{Traffic}_{\text{conv cache}} &+ (\#\text{ useless prefetches})\\ &+ 2 * (\#\text{ polluting prefetches}) \qquad (2)\\ &+ (\#\text{ prefetch side effects}).\end{aligned}$$

3 Benchmarks

In this section, we illustrate the use of PTMT to understand the working of a near-optimal prefetch algorithm [4] and Next Sequential Prefetching (NSP) [19], [10] when used for prefetching data into the L1 data cache.

To evaluate these algorithms on a wide range of current workloads that stress the D-cache, we selected those integer applications from the SPEC CPU2000 [21] suite that exhibit higher D-cache miss ratios (*bzip2, crafty, gcc, gzip, twolf*), as well as Windows-NT traces for Doom, Explorer, and Netscape, and traces of the commercial database applications, TPCC and TPCD. Characteristics of these benchmarks are summarized in Table 6.

TABLE 6
Benchmark Characteristics

Name	Instructions (in millions)	Mem Refs. (in millions)
tpcc	172	95
tpcd	58	20
doom	751	509
explorer	407	247
netscape	864	500
bzip2	2,000	753
crafty	2,000	818
gcc	2,000	844
gzip	2,000	753
twolf	2,000	719

We used traces of the database workloads, TPCC and TPCD [22], that were collected on an RS/6000 machine by the microprocessor research group at the IBM T.J. Watson Research Center. For the Windows-NT applications, we used traces collected by Stevan Vlaovic at the University of Michigan, using a PC simulator based on Bochs [23]. Bochs models the entire machine platform in detail to support the complete execution, including operating system events in addition to the standard instruction, data, and branch traces. Out-of-the-box Windows-NT 4.0 (Build 1381) was used as the operating system for the virtual PC simulator. The three Windows-NT applications studied are: Id's Doom, Microsoft Explorer 5.0, and Netscape 4.0. Doom is a first-person type combat game; the run of Doom included recording a session of a Doom game and then replaying it on the PC simulator. Both Microsoft Explorer 5.0 and Netscape 4.0 are Web browsers; our input is a set of three HTML pages: the CNN Web page, an ESPN Web page, and the University of Michigan's homepage.

4 Evaluating a Near Optimal Prefetch Algorithm

Cao et al. [4] proposed an unimplementable near-optimal prefetch algorithm, called the "aggressive algorithm," with the goal of establishing a lower bound on the total elapsed time required to access a known set of references. Their chosen machine model, unless stalled, executes one memory reference per time unit using a fully associative cache with a capacity of k lines and a one cycle access time. The system has an F cycle L2 access (miss or prefetch) latency and allows only one outstanding access to the next level of memory at any point in time. The aggressive algorithm uses the following four rules to issue prefetches:

- *Rule 1.* Each prefetch should bring into the cache the next line in the reference stream that is not already in the cache.
- *Rule 2.* The line evicted to accommodate the prefetch should be the one with its next reference farthest in the future.
- *Rule 3.* A prefetch of y must never replace any line x that will be referenced before y.

- *Rule 4.* A prefetch is issued at the first opportunity that satisfies the three rules above.

It is shown in [4] that, on any reference string, R, the elapsed time of the aggressive algorithm on R is within a factor of $\min(1 + F/k, 2)$ of the optimal elapsed time. Note that this near-optimal algorithm is unimplementable due to its use of future knowledge. It therefore serves only to approximate a theoretical upper bound on the achievable performance of prefetch algorithms. We now show how PTMT can be used to identify which aggressive prefetch decision points of this algorithm result in extra traffic.

Since Rule 1 eliminates all cache misses, the aggressive algorithm generates no prefetches belonging to cases 1, 4, or 7 of PTMT in which the next reference to the line evicted by the prefetch is a miss in the *pf-cache*. Likewise, case 10 cannot occur since it has a miss in the *pf-cache*. Furthermore, since Rules 1 and 3 ensure that a prefetched line is never replaced before being referenced, cases 7, 8, and 9 of PTMT cannot occur.

Thus, each prefetch of the aggressive algorithm must be case 2, 3, 5, or 6 in PTMT. Case 6 prefetches are the most useful because they save a miss without costing any extra traffic. In Section 2.2.2, we showed that case 5 prefetches start a prefetch chain and case 2 prefetches start or continue a prefetch chain that becomes progressively less desirable as the length of the chain grows, causing more and more extra traffic and, possibly, an extra miss. The aggressive algorithm, however, does not add extra misses since cases 2 and 3 do not add extra misses and cases 5 and 6 each save one miss.

Since the aggressive algorithm cannot generate case 8 prefetches and since, as we now show, no prefetch chain in the aggressive algorithm can start with a case 2 prefetch, each prefetch chain in the aggressive algorithm must start with a case 5 prefetch.

Suppose the aggressive algorithm issues a prefetch for x at time t, replacing line y, and suppose the pair (x, y) belongs to case 2 of PTMT. We show, by contradiction, that (x, y) cannot begin a prefetch chain:

1. Since (x, y) belongs to case 2, the next references to x and y both hit in the *conv-cache*, both x and y are present in the *conv-cache* at time t.
2. Since no prefetching occurs in the *conv-cache*, x is present in the *conv-cache* throughout its nonreference interval, (t_b, t_e), that contains t; note that x is referenced at t_b ($< t$) and t_e ($> t$), but not referenced in (t_b, t_e).
3. Since x is present in the *pf-cache* at t_b, but absent at t, it must have been replaced to accommodate some other prefetch in the interval (t_b, t). Let z be the line brought in by that prefetch.
4. Since x is replaced in the interval (t_b, t) and subsequently prefetched prior to its next reference at t_e, the (z, x) pair can only belong to cases 2 or 5. (Case 8, which never occurs in the aggressive algorithm, is the only other case in which the replaced line hits in the *conv-cache* and is prefetched in the *pf-cache* before its next reference.)
5. Thus, we have shown that every case 2 (x, y) pair has an earlier (z, x) pair chained to it that belongs to case 2 or 5. By applying this argument inductively, we see that no chain in the aggressive algorithm can begin with a case 2 prefetch.

Therefore, each prefetch chain of the aggressive algorithm must start with a case 5 prefetch. Since the aggressive algorithm performs only case 2, 3, 5, and 6 prefetches, all its prefetch chains are of the form $(5\ (2)^*\ 3)$. Each such chain of length L saves one miss, but incurs $L - 1$ lines of extra traffic. Furthermore, since there are no *pf-cache* misses in the aggressive algorithm, a case 3 prefetch to line y can occur only if y was replaced by some (x, y) prefetch. However, that (x, y) prefetch cannot be either case 3 or 6 since in those cases (see Table 3), the replaced line, y, is also replaced in the *conv-cache*, whereas y is also the prefetched line in this case 3 and hits in the *conv-cache* at that same next reference. Thus, that (x, y) prefetch must be either case 2 or 5 and this case 3 prefetch of y must therefore be chained to it. Hence, the only case 3 prefetches in the aggressive algorithm are chain-terminating prefetches. The prefetches of the aggressive algorithm therefore consist entirely of unchained case 6 prefetches and $(5\ (2)^*\ 3)$ chains.

We can now address why this near-optimal algorithm is not in fact optimal. Note that the replaced line of a case 5 prefetch remains in the *conv-cache* until its next reference, whereas, in a case 6 prefetch, it is replaced. Thus, a case 5 prefetch could be converted to a case 6 prefetch by delaying it until the *pf-cache* can replace a line which will also be replaced from the *conv-cache* before its next reference. However, Rule 4 of the aggressive algorithm does not permit delaying a prefetch for this reason; therefore, each case 5 prefetch represents an aggressive prefetch decision point at which the algorithm may generate extra traffic due to the resulting prefetch chains. Nevertheless, unless that extra traffic causes congestion that induces significant runtime delays, this algorithm may be able to reduce the total elapsed time by not delaying the prefetch. The aggressive prefetch algorithm does not use a strategy that is complex enough to resolve these situations optimally.

Fig. 6a presents the PTMT histogram of the prefetches issued by this aggressive near-optimal algorithm on the commercial and Windows-NT applications. These simulations used a 32KB, fully associative cache with 32 byte lines and a miss penalty of 20 cycles. Since the algorithm always prefetches the next line in the reference stream that is not already in the cache, there are no cache misses. However, since the algorithm delays its prefetches until it can issue the prefetch without replacing a line that is referenced before the next reference to the prefetched line, there will be "delayed hits" which occur whenever a prefetch cannot be issued early enough and the prefetched line fails to arrive in cache before its next reference, therefore incurring some fraction of a full miss penalty. For the purpose of classification using PTMT, we treat "delayed hits" as hits. However, whenever we run a detailed timing simulation, the delayed hits are separated from the other (0 penalty) hits.

Fig. 6a shows a breakdown by case of the number of prefetches per 1,000 references for the commercial and Windows-NT applications. As explained above, the aggressive near-optimal algorithm never generates cases 1, 4, 7, 8, 9, and 10. On average, 69 percent of the prefetches belong to

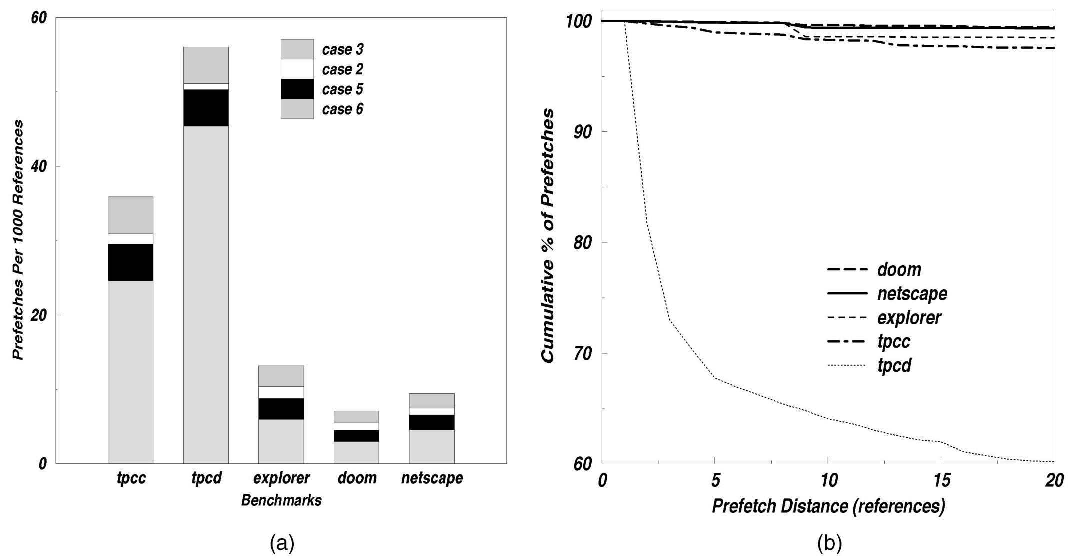


Fig. 6. Performance of the near-optimal algorithm. (a) PTMT classification. (b) Prefetch distance.

case 6.[1] The length of a prefetch chain is at most 3, i.e., only $(5\ 3)$ and $(5\ 2\ 3)$ chains actually occurred. On average, 13 percent of the prefetches belong to case 5 (and, hence, another 13 percent belong to case 3), while only 5 percent belong to case 2; therefore, 38 percent of the prefetch chains are of the form $(5\ 2\ 3)$ and 62 percent are $(5\ 3)$.

PTMT classifies prefetches based only on extra traffic and misses; it does not directly account for the timeliness of the prefetches. In Fig. 6b, we show the prefetch distance, as measured using the aggressive algorithm's machine model, as defined in [4]. In this model, machine stalls are due only to waiting for a memory operand and, in the absence of a machine stall, the time between two successive references is constant and is called "one time-unit." Furthermore, the machine model allows only one outstanding access to the next level of the memory hierarchy at any point in time. Therefore, the prefetch distance is measured as the number of references from when the prefetch is actually issued until its first use.

Fig. 6b shows that, for all the applications (except *tpcd*) almost all the prefetches have a prefetch distance of 20 or more references. Since the fetch latency to the next level of memory hierarchy was set to 20 in these simulations, almost all prefetches do arrive in a timely fashion, i.e., sufficiently ahead of their next reference time. We therefore conclude that the extra traffic generated by the prefetch chains does not cause congestion, i.e., it does not induce significant runtime delays in this timing model even though 18 percent of the prefetches are case 2 or 3.

In *tpcd*, however, nearly 40 percent of the prefetches arrive with a distance of less than 20 references. Our analysis of the *tpcd* references showed that its misses occur in bursts and, consequently, the prefetch requests of the aggressive algorithm also occur in bursts. Since the algorithm allows only one outstanding prefetch to the next level of memory, these prefetches do result in delayed hits.

1. Note that here and in the rest of this paper, the "average" is based on weighting the applications by taking an equal number of instructions from each application.

5 Evaluating Next Sequential Prefetch Algorithm

Next Sequential Prefetching (NSP) [10], [19] is a simple prefetch algorithm in which, whenever line b is accessed, a prefetch for line $(b + 1)$ is issued unless line $(b + 1)$ is already present in the cache. To control the number of useless prefetches, a variant called tagged prefetching [10] is usually employed. In this variant, a tag bit associated with each cache line is set whenever the line is brought to cache by a prefetch. A prefetch to the next sequential line is triggered whenever there is a cache miss and whenever there is a hit to a line whose tag bit is set (i.e., on the first reference to a prefetched line). After initiating the prefetch of line $(b + 1)$, the tag bit of line b is reset. In essence, this algorithm exploits an application's sequential spatial locality. In this paper, we use only tagged NSP.

To limit the number of useless and polluting prefetches further, we may also use a "confirmation bit" filter [17]. A confirmation bit is associated with each line in the L2 cache. It is initially set to 1, reset to 0 whenever the line is prefetched, and then replaced in L1 without being used (a "bad" prefetch, i.e., case 7, 8, or 9 of PTMT), and set to 1 again only when the line experiences a demand fetch from the L2 cache (i.e., an L1 miss). Prefetch requests are squashed if the line's confirmation bit is found to be 0 in the L2 directory. We evaluate NSP with and without this filter.

5.1 Simulation Environment

To evaluate the five applications from the CPU2000 suite, detailed cycle level simulations were run by incorporating PTMT into the out-of-order simulator from the SimpleScalar toolset [2] and using the microarchitectural parameters in Table 7. As we did not have access to cycle level timing simulators that could run the Windows-NT or commercial applications, we developed a functional cache simulator that incorporates PTMT and used it to evaluate these applications.

TABLE 7
Microarchitectural Parameters for the SimpleScalar Simulation

Fetch, Decode & Issue Width	4
Inst Fetch & L/S Queue Size	16
Reservation stations	64
Functional Units	4add/2mult
Memory system ports to CPU	4
L1 I cache	16KB, 2-way, 32byte
L1 D cache	32KB, 4-way, 32byte
L1 access time (cycles)	1
Unified L2 cache	256KB, 2-way, 32byte
L2 cache latency (cycles)	15
Mem latency (cycles)	30
Branch Predictor	2-lev, 2K-entry

5.2 Simulation Results

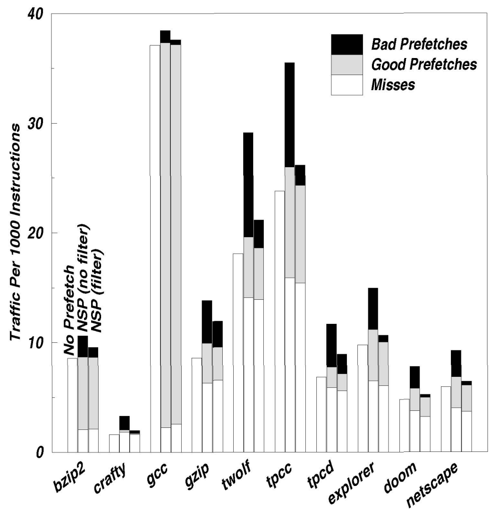
We first discuss NSP algorithm performance using gross traditional metrics, namely, coverage, accuracy, number of misses, good prefetches and bad prefetches, number of prefetched lines that failed to arrive in cache before their next use, and IPC. We next evaluate NSP using PTMT.

5.2.1 Traditional Metrics

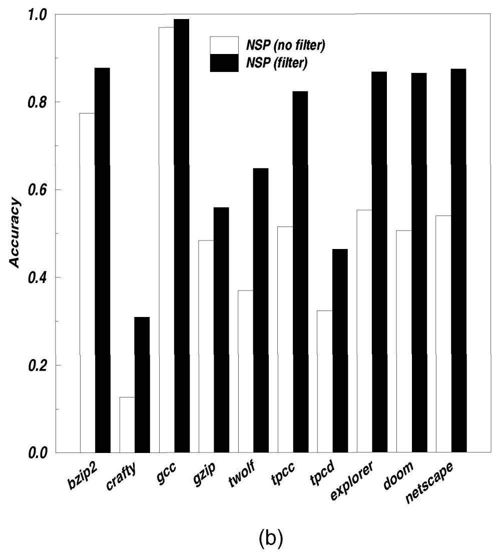
Fig. 7a presents the traffic per 1,000 instructions with no prefetching and for NSP with and without the confirmation bit filter. Traffic is measured as the total number of lines fetched from the L2 cache, due to both demand fetches and prefetches. Each bar shows the number of misses, the number of *good* prefetches, and the number of *bad* prefetches.

Using NSP without the confirmation bit filter increases traffic by 29 percent for the CPU2000 applications and 55 percent for the commercial and Windows-NT applications. The confirmation bit significantly reduces the number of bad prefetches, decreasing average traffic by 18 percent for the CPU2000 applications and by 28 percent for the commercial and Windows-NT applications.

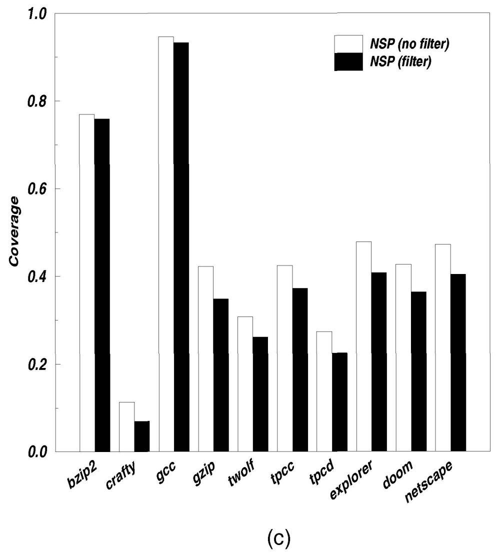
Fig. 7a shows that NSP does not reduce the misses for *crafty*; one reason may be the small number of misses



(a)

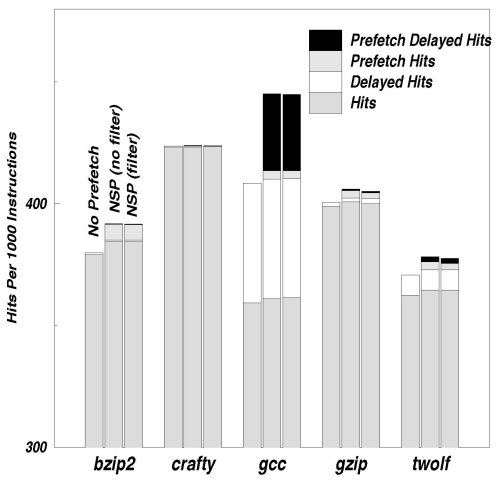


(b)

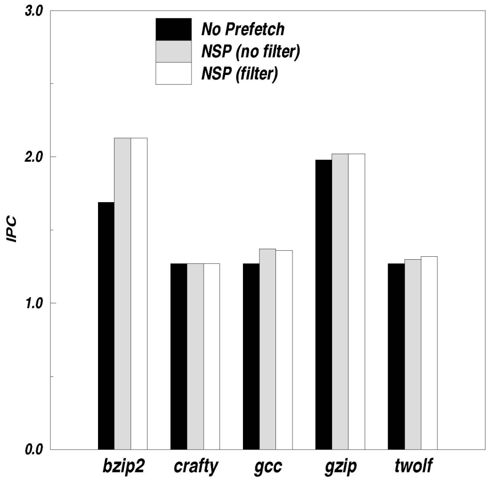


(c)

Fig. 7. Traffic, accuracy, and coverage using NSP. (a) Traffic per 1,000 instructions. (b) Accuracy. (c) Coverage.



(a)



(b)

Fig. 8. Hits and IPC for CPU2000 applications. (a) Hits per 1,000 instructions. (b) IPC.

experienced by *crafty*. Since NSP uses cache misses as prefetch triggers, applications with fewer misses will have fewer opportunities to trigger prefetches. For the other CPU2000 applications, NSP without the filter reduces the number of misses by 66 percent; adding the confirmation bit filter actually increases the average number of misses by 1 percent. For the commercial and Windows-NT applications, NSP reduces the misses by 29 percent without the filter and by a further 8 percent with the filter. This further reduction may be due to the predominantly nonsequential L2 access patterns of these applications, which increases NSP's need for filtering.

Fig. 7b and Fig. 7c show the accuracy and coverage for NSP without and with the filter. Among the CPU2000 applications, the accuracy varies from $> 90\%$ (*gcc*) to $< 50\%$ (*crafty*). For the commercial applications, applying the confirmation bit filter improves the accuracy to $> 80\%$ (except for *tpcd*). However, the confirmation bit filter decreases the coverage, implying that it also eliminates some good prefetches.

In Fig. 7a, except for *bzip2*, *gcc*, and *gzip*, NSP eliminates more misses when the confirmation bit filter is used. However, Fig. 7c shows that the coverage decreases for all applications when the confirmation bit filter is applied, which implies a reduction in the number of good prefetches. With such gross metrics we can only surmise that, maybe, without the filter, the bad prefetches were polluting the cache, and, by reducing the number of bad prefetches, the filter reduced the number of misses even though the number of good prefetches was also reduced. Using PTMT will clarify the underlying causes.

In addition to miss reduction and traffic increase, it is important to know whether these algorithms issue their prefetches in a timely fashion. When a prefetch is not triggered sufficiently ahead of time, the prefetched line does not arrive in cache before it is referenced, resulting in a "delayed hit" that suffers some fraction of the full miss penalty. Timeliness of the prefetches can be measured by quantifying the number of prefetched lines that are referenced before arriving at the cache. As we did not have access to cycle level timing simulators for the Windows-NT or commercial applications, we present prefetch timeliness and overall IPC results for the CPU2000 applications only. However, we have seen in Fig. 6b that nearly all the prefetches for the Windows-NT and commercial applications, except for *tpcd*, had a prefetch distance of 20 or more instructions with the near-optimal algorithm; nevertheless, when the functional simulator evaluated NSP for these applications, over 80 percent of the prefetches were not timely (distances $<$ 20 instructions).

Fig. 8a shows the hits per 1,000 instructions for the CPU2000 applications with no prefetching, NSP without filter, and NSP with filter, respectively. Each bar distinguishes hits to *regular* lines from hits to *prefetched* lines. Recall that a line brought to the cache by a prefetch is considered a *prefetched* line until it is either referenced or replaced; otherwise, all lines in cache are *regular*. In each bar, "Prefetch Delayed Hits" refers to accesses to prefetched lines that have not yet arrived at the cache at the time of the reference. Similarly, "Delayed Hits" refers to accesses to regular lines that are delayed due to a trailing edge effect of some prior miss or prefetch. Finally, accesses to regular and prefetched lines that are present in the cache at the time of their reference are referred to as "Hits" and "Prefetch Hits," respectively.

Fig. 8a shows that, for *gcc*, 90 percent of the references to lines prefetched by NSP are delayed hits, but only 19 percent are delayed hits for the other CPU2000 applications. Furthermore, prefetching generally increases the number of delayed hits to *regular* lines, presumably because those demand fetches from the L2 cache that still remain may take a longer time to arrive since they have to wait for any outstanding prefetch requests to complete. Adding the confirmation bit filter has very little effect on the hits or delayed hits. Thus, although the filter does help reduce the overall traffic to the L2 cache by reducing the bad prefetches, it does not appear to either improve or degrade overall prefetch timeliness.

Fig. 8b shows the overall IPC with and without prefetching. On average, NSP increases the average IPC by 6.6 percent; poor prefetch timeliness may be one limiting factor. Applying the filter has little effect on IPC. The main goal of the confirmation bit filter is to reduce the useless and polluting prefetches; therefore, when bandwidth to the

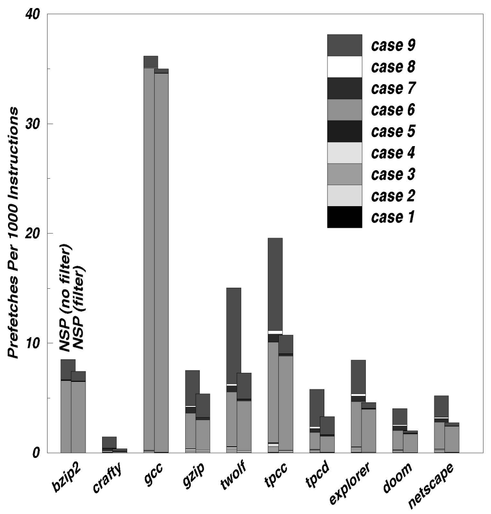


Fig. 9. Prefetch classification using PTMT.

L2 cache is a bottleneck, filtering will improve the IPC by reducing overall traffic. However, in the SimpleScalar model used for these applications, bandwidth to L2 is not a bottleneck and, therefore, the reduction in useless prefetches did not improve either effective memory access time or IPC.

5.2.2 PTMT Classification

Fig. 9 shows the detailed PTMT classification for NSP on each application both without (left bar) and with the filter (right bar).

The average fraction of polluting prefetches is 2.6 percent. Moreover, over 90 percent of the polluting prefetches belong to case 7. The confirmation bit filter reduces the average fraction of polluting prefetches to 1.3 percent. We observed in Section 5.2.1 that the confirmation bit filter reduced overall misses for *twolf* and the commercial and Windows-NT applications, even though the number of good prefetches decreased. We now clearly see that this reduction in misses is mainly due to the reduction in the number of polluting prefetches.

The average fraction of useless prefetches is 27 percent without the confirmation bit filter, but only 14 percent with the filter. Although traditional metrics classify case 2, 3, and 4 prefetches as *good*, PTMT showed that these prefetches incur an extra line of traffic without any net savings in cache misses; PTMT thus classifies case 2, 3, and 4 prefetches as useless, along with the *bad* prefetches of cases 8 and 9. Nevertheless, from Fig. 9, we observe that the average fraction of prefetches belonging to cases 2, 3, and 4 is only 2.0 percent (without filter); hence, these prefetches are not responsible for a significant portion of the increase in the overall traffic seen in Fig. 7.

The useful prefetches (cases 5 and 6) account for the substantial reduction in misses observed in Fig. 7. Here, we see that almost all the useful prefetches are case 6; fewer than 1 percent of all prefetches are case 5. Thus, almost all the useful prefetches replaced a line that would also be replaced before its next reference in a conventional cache (without any prefetching).

Recall that lines replaced by case 3, 6, and 9 prefetches are also replaced from the *conv-cache*. Thus, these prefetches do not prematurely replace a line that would otherwise have been retained in the cache. However, among these cases, only the case 6 prefetches are truly useful. The average fraction of case 3, 6, and 9 prefetches in Fig. 9 is 92 percent.

Further, recall that lines replaced to accommodate case 2, 5, and 8 prefetches are subsequently prefetched into the *pf-cache* before their next reference and remain there until that next reference, implying that these prefetches were probably issued too early and replaced a line that was more desirable than the prefetched line at that time. However, in Fig. 9, the average fraction of case 2, 5, and 8 prefetches is only 2.1 percent.

Case 10 side-effects (not shown in Fig. 9) are negligible, ranging from 0.023 occurrences per thousand instructions in *gcc* to 0.548 in *tpcc* without the filter and from 0.011 (*gcc*) to 0.353 (*gzip*) with the filter. The LRU stack reordering due to NSP thus has no significant effect on the number of misses and extra traffic.

To summarize, the above PTMT analysis shows that NSP without any filter does have a large number of useless prefetches, almost all of which are case 9. Polluting prefetches constitute a very small fraction of all prefetches and are therefore not a serious concern. Among the useful prefetches, case 6 dominates; thus, undesirable chaining due to case 5 prefetches is minimal. The confirmation bit filter was effective in reducing overall traffic and did not increase misses significantly. The filter greatly reduced useless prefetches while usually retaining almost all the useful prefetches. Nevertheless, by using the detailed timing simulator on the CPU2000 applications, we observed that applying the filter did not significantly change their IPC. However, L2 cache bandwidth was not a bottleneck in the SimpleScalar timing model used for these simulations; IPC will benefit more from filtering whenever L2 bandwidth is a serious concern.

6 Conclusions

PTMT analyzes each prefetch and assigns it 0, 1, or 2 lines of additional traffic and a -1, 0, or +1 net change in misses that it causes relative to a conventional cache. The key to being able to carry out this classification is the simultaneous simulation of a cache with some prefetch algorithm of interest and a conventional cache without prefetching. The three major contributions of this paper are:

- A classification that completely describes the nine possible outcomes that encompass a prefetched line as well as the line it evicts from the cache, and disjointly attributes an increase in traffic and a change in the number of misses to each prefetch according to these outcomes.
- This classification further describes the effect (case 10) that prefetch induced LRU stack reordering (viewed as an indirect side-effect of prefetching) has on the miss rate and traffic. This 10 case classification and the cost attributed to each instance of a case disjointly and completely accounts for the net difference in traffic and misses between a cache with some prefetch algorithm and the same cache with no prefetching.
- An analysis of the chaining effects among prefetches which reveals, for example, that even an apparently

useful prefetch (case 5) may not be beneficial in the broader context of the total cost of the chain of other prefetches on which it depends.

We have illustrated the usefulness of PTMT by applying it to a near optimal prefetch algorithm that assumes future knowledge and to the next sequential prefetch algorithm with and without a confirmation bit filter. Although it generally did not cause much cache pollution or chaining, NSP did not improve IPC very much because many misses remained which NSP did not attempt to cover and many of its prefetches were not timely enough to eliminate delayed hits (we refer the reader to [20] for an analysis of the remaining misses and further PTMT detail and evaluations). The confirmation bit filter did eliminate most of NSP's useless prefetches, but had little effect on IPC because bandwidth to the L2 cache was not a bottleneck in our NSP simulations.

ACKNOWLEDGMENTS

The authors would like to thank the reviewers who helped clarify and strengthen the presentation of this paper.

REFERENCES

[1] J. Bennett and M. Flynn, "Prediction Caches for Superscalar Processors," *Proc. 30th Ann. Int'l Symp. Microarchitecture,* pp. 81-91, Dec. 1997.

[2] D. Burger and T. Austin, "The SimpleScalar Tool Set, Version 2.0," Technical Report TR 1342, Univ. of Wisconsin, June 1997.

[3] D. Callahan, K. Kennedy, and A. Porterfield, "Software Prefetching," *Proc. Fourth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 40-52, Apr. 1991.

[4] P. Cao, E. Felton, A. Karlin, and K. Li, "A Study of Integrated Prefetching and Caching Strategies," *Proc. ACM SIGMETRICS,* pp. 188-196, June 1995.

[5] M. Charney, "Correlation-Based Hardware Prefetching," PhD dissertation, Cornell Univ., Aug. 1995.

[6] M. Charney and T. Puzak, "Prefetching and Memory System Behavior of the SPEC95 Benchmark Suite," *IBM J. Research and Development,* vol. 41, no. 3, pp. 265-286, May 1997.

[7] T. Chen and J. Baer, "Reducing Memory Latency via Non-Blocking and Prefetching Caches," *Proc. Fifth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 51-61, Oct. 1992.

[8] T. Chen and J. Baer, "A Performance Study of Software and Hardware Data Prefetching Schemes," *Proc. 21st Ann. Int'l Symp. Computer Architecture,* pp. 223-232, Apr. 1994.

[9] W. Chen, S. Mahlke, P. Chang, and W. Hwu, "Data Access Microarchitectures for Superscalar Processors with Compiler-Assisted Data Prefetching," *Proc. 24th Int'l Symp. Microarchitecture,* pp. 69-73, Nov. 1991.

[10] J.D. Gindele, "Buffer Block Prefetching Method," *IBM Technical Disclosure Bull.,* vol. 20, no. 2, pp. 696-697, July 1977.

[11] D. Joseph and D. Grunwald, "Prefetching Using Markov Predictors," *Proc. 24th Int'l Symp. Computer Architecture,* pp. 252-263, May 1997.

[12] N. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," *Proc. 17th Int'l Symp. Computer Architecture,* pp. 364-373, May 1990.

[13] A. Klaiber and H. Levy, "An Architecture for Software-Controlled Data Prefetching," *Proc. 18th Int'l Symp. Computer Architecture,* pp. 43-53, May 1991.

[14] M. Lipasti, W. Schmidt, S. Kunkel, and R. Roediger, "Spaid: Software Prefetching in Pointer and Call Intensive Environments," *Proc. 28th Ann. Int'l Symp. Microarchitecture,* pp. 231-236, Nov. 1995.

[15] C.-K. Luk and T. Mowry, "Compiler Based Prefetching for Recursive Data Structures," *Proc. Seventh Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 222-233, Oct. 1996.

[16] S. Mehrotra and L. Harrison, "Examination of a Memory Access Classification Scheme for Pointer-Intensive and Numeric Programs," *Proc. 10th Int'l Conf. Supercomputing,* pp. 133-139, May 1996.

[17] J. Pomerene, T. Puzak, R. Rechtschaffen, and F. Sparacio, "Prefetching System for a Cache Having a Second Directory for Sequentially Accessed Blocks," US Patent 4,807,110, Feb. 1989.

[18] A. Roth and G. Sohi, "Effective Jump-Pointer Prefetching for Linked Data Structures," *Proc. 26th Int'l Symp. Computer Architecture,* pp. 111-121, May 1999.

[19] A. Smith, "Cache Memories," *ACM Computing Surveys,* pp. 473-530, Sept. 1982.

[20] V. Srinivasan, "Hardware Solutions to Reduce Effective Memory Access Time," PhD dissertation, Univ. of Michigan, Jan. 2001.

[21] Standard Performance Evaluation Corp., "CPU2000 documentation," http://www.spec.org/osg/cpu2000/docs, 2000.

[22] "Transaction Processing Performance Council," http://www.tpc.org, 2003.

[23] S. Vlaovic, E.S. Davidson, and G.S. Tyson, "Improving BTB Performance in the Presence of DLLs," *Proc. 33rd Int'l Symp. Microarchitecture,* pp. 77-86, Dec. 2000.

Viji Srinivasan received the BS degree in physics from the University of Madras in 1990, the MS degree in computer science and engineering from the Indian Institute of Science in 1994, and the PhD degree in computer science and engineering from the University of Michigan in 2001. Currently, she is with the IBM Research Division, T.J. Watson Research Center as a research staff member in the Computer Architecture Department. Her research interests include computer architecture and performance analysis. She is a member of the IEEE.

Edward S. Davidson received the BA degree in mathematics (Harvard University, 196l), the MS degree in communication science (University of Michigan, 1962), and the PhD degree in electrical engineering (University of Illinois, 1968). He joined the University of Michigan as a professor of electrical engineering and computer science in 1988, served as its chair through 1990, as associate chair for computer science and engineering (1997-2000), and was appointed professor emeritus in June 2000. He managed the hardware design of the Cedar parallel supercomputer at the Illinois Center for Supercomputing Research and Development (1984-1987) and directed Michigan's Center for Parallel Computing (1994-1997). His research interests include computer architecture, parallel and pipeline processing, performance modeling, intelligent caches, supercomputing, and application code assessment and tuning. His recent research has focused on analyzing and improving the performance of application codes on parallel, vector, and workstation architectures and on intelligent cache design and management, pipeline design, and prefetching. He was elected a fellow of the IEEE in 1984, was chair of ACM-SIGARCH (1979-1983), and received the IEEE Computer Society's Harry M. Goode Memorial Award in 1992, its Taylor L. Booth Education Award in 1996, and the IEEE Computer Society/ACM Eckert-Mauchly Award in 2000.

Gary S. Tyson received the PhD degree in computer science from the University of California-Davis in 1997. His research interests include computer architecture, high performance memory systems, and compiler optimization. He is an assistant professor in the Advanced Computer Architecture Lab (ACAL) in the Department of Electrical Engineering and Computer Science at the University of Michigan. Prior to this, he was an acting assistant professor in the Computer Science Department at the University of California-Riverside from 1995-1997. He is a member of the IEEE.