A Prefetch Taxonomy

Viji Srinivasan, Edward S Davidson, Gary S Tyson
e-mail: sviji@eecs.umich.edu, davidson@eecs.umich.edu, tyson@eecs.umich.edu

Abstract

The difference in processor and main memory cycle time has necessitated the use of aggressive prefetching techniques to reduce or hide main memory access latency. However, prefetching can significantly increase memory bandwidth and unsuccessful prefetches may even pollute the primary cache. Although the current metrics, coverage *and* accuracy*, do provide an impression of the overall quality of a prefetching algorithm, they are simplistic and do not unambiguously and precisely explain the performance of a prefetching algorithm. In this paper, we introduce a new and complete taxonomy for classifying prefetches, accounting for the difference in traffic and misses generated by each prefetch. Our classification of individual prefetches is very useful in identifying the specific strengths and weaknesses of an existing prefetch implementation and we demonstrate it by applying our taxonomy to two implementable prefetching techniques. Based on the histogram of prefetches by category we developed a static filter to reduce/eliminate polluting prefetches of any given prefetching technique.*

1 Introduction

As processor speeds have increased over the past few decades, the gap between memory access latency and the processor cycle time has steadily increased. Use of multi-level cache hierarchies has been a traditional solution to bridging this gap. However, with superscalar processors now executing multiple instructions per cycle, even greater demand is placed on the cache system. Prefetching is a technique that reduces memory access latency by fetching lines into cache before a demand 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 prefetching techniques are speculative; they must predict the prefetch addresses. Hence the following issues arise:

- Some addresses are not accurately predicted limiting the effectiveness of speculative prefetching techniques.
- Even when the prediction is accurate and the prefetch is issued early enough to cover the nominal latency, the full memory latency may not be hidden due to additional delays caused by limited available memory bandwidth.

- Prefetched lines, whether correctly predicted or not, may be issued too early and pollute the cache by replacing more desirable lines.

Effective prefetching depends on achieving good miss coverage with sufficient accuracy to avoid saturating the memory bus with useless prefetches and polluting the cache. Traditionally *coverage* and *accuracy* have been used as metrics to evaluate the effectiveness of a prefetching algorithm. Prefetch *coverage* is the fraction of misses that are eliminated by the prefetching algorithm. Prefetch *accuracy* estimates the quality of the prefetching algorithm as a fraction of all prefetches that are useful. To measure coverage and accuracy, prefetches are broadly classified into two categories : *good* and *bad*. Good prefetches are referenced by the application before they are replaced and bad prefetches are replaced before they are referenced. If a conventional cache has M misses without using any prefetching technique and has G good prefetches and B bad prefetches with some prefetching algorithm, we can measure prefetch coverage and accuracy as follows:

$$\text{Coverage} = \text{G/M}, \tag{1}$$

$$\text{Accuracy} = \text{G/(G + B)}. \tag{2}$$

Although *coverage* and *accuracy* do provide an impression of the overall quality of a prefetching algorithm, they are simplistic and do not unambiguously and precisely explain the performance of a prefetching algorithm. Using the simple example in Figure 1 we show that maximizing good prefetches, or even getting reasonable coverage with high accuracy is not the true objective of a prefetching technique.

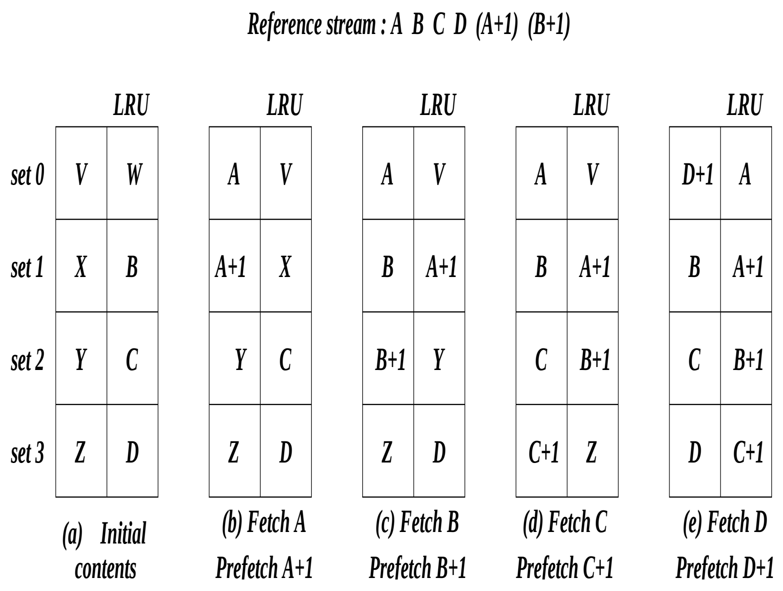


Figure 1: Cache access outcomes with NSP

The cache used in this example has 4 sets with 2-way associativity and uses an LRU replacement policy. The initial contents of the cache are given in Figure 1(a). The reference stream consists of the 6 unique line addresses, A, B, C, D, $(A + 1)$, and $(B + 1)$, which map to sets 0, 1, 2, 3, 1, and 2 respectively. The total misses for this cache without any prefetching technique is 3 (missing lines : A, $(A + 1)$ and $(B + 1)$). We now consider the cache access outcomes using the *Next Sequential Prefetching* (NSP) technique [19]. Figure 1(b) shows the cache contents after the miss to line A is resolved. In NSP the next sequential line $(A + 1)$ is prefetched along with A. Line $(A + 1)$ replaces the LRU line B of set 1. Figures 1(c,d,e) show the contents of the cache after the references to B, C and D, respectively, are resolved. Of the 4 prefetches, 2 are *bad* and 2 are *good*. Therefore, the coverage is 66.7% and the accuracy is 50%. Although NSP does eliminate 2 of the 3 misses, it introduces 3 new misses. Therefore the total misses with NSP is increased from 3 to 4. Furthermore, NSP increases the total traffic from 3 to 8 cache lines. This example illustrates that a prefetch technique, even with 66.7% coverage and 50% accuracy can actually increase the miss ratio and more than double the memory traffic.

Prefetch side-effects on the misses and traffic are thus critical in designing an efficient prefetching technique. Coverage and accuracy, as seen above, do not provide a direct indication of performance, and can be very misleading metrics. Furthermore, even direct measurements of the total change in misses and total traffic do not provide much insight on the underlying causes of that net change and how the performance of a prefetching technique may be improved. There is a need for a more refined classification of individual prefetches that gives more insights into how a given prefetching technique works.

In this paper we introduce a new, accurate and complete taxonomy, called the Prefetch Traffic and Miss Taxonomy (PTMT), for classifying prefetches and precisely accounting for the difference in traffic and misses generated by each prefetch. PTMT quantifies the usefulness of a prefetch in terms of the difference in misses and the net traffic increase that are due to the outcome of both the prefetch and the line replaced to accommodate the prefetch. PTMT's classification of individual prefetches has proven to be very useful in identifying the specific strengths and weaknesses of an existing prefetch implementation. Furthermore, the taxonomy helps answer the following questions:

- How can a prefetch technique eliminate those prefetches that pollute the cache?
- How can a prefetch technique reduce useless prefetches and thus relieve the pressure on the available memory bandwidth?
- What new insights does the prefetch classification provide for improving the performance of the prefetching technique?

The key to PTMT is the simultaneous simulation of a conventional cache with no prefetching and another cache that implements some prefetching technique. Using this model we analyze each prefetch and assign it a -1, 0, or +1 net change in misses and a cost of 0, 1 or 2 lines of additional traffic that it causes relative to the conventional cache. The completeness of PTMT assures that the extra traffic and the difference in misses based on PTMT plus the traffic and misses of the conventional cache always exactly matches the total traffic and misses of the cache with the prefetch technique. PTMT therefore provides new and precise metrics for evaluating a given prefetching technique. Furthermore, by separately tabulating the strengths and weaknesses of a prefetch technique, PTMT provides insights for improving it.

The rest of this paper is organized as follows. Section 2 describes the prefetch traffic miss taxonomy. Section 3 uses PTMT to measure the effectiveness of two common prefetching algorithms, *Next Sequential Prefetching* and *Shadow Directory Prefetching*. In section 4 we describe insights derived from the taxonomy to improve the performance of these prefetching techniques. We conclude in section 5.

2 Prefetch Traffic and Miss Taxonomy

The Prefetch Traffic and Miss Taxonomy (PTMT) categorizes each prefetch according to 2 factors, whether it remains in the cache until its next reference or is replaced before then, as well as the outcome of the next reference to the line evicted to accommodate this prefetch.

2.1 Cache Configuration

Classifying prefetches using PTMT fundamentally requires the simultaneous simulation of two identically configured caches, a conventional cache without any prefetching, *conv-cache*, and one with some prefetching technique of interest, *pf-cache*. As our goal is to study the effect of prefetching on the cache performance, we prefetch directly into the cache and do not consider using a separate prefetch buffer [10]. There is no restriction on the size, associativity, or line size of these two caches, except that they are the same.

In this paper, we refer to lines in each cache as *prefetched* or *regular* based on the following:
A line that is prefetched into the *pf-cache* is called a *prefetched line* until it is either replaced or referenced; after being referenced it is called a *regular line* until it is replaced. All other lines in either cache are called *regular lines*. A 1-bit tag is maintained in the *pf-cache* to distinguish between regular and prefetched lines.

As our focus is on the number of cache lines fetched from the next level of memory, we assume that the caches are write-through and no-write-allocate, and the traffic to the next level of memory due to stores is ignored. LRU replacement is used. Each set of the cache is implemented in the simulation as an LRU stack. All new lines entering the cache via a miss or a prefetch are pushed onto the top of the LRU stack (the most recently used position); lines leave the set from the bottom of the stack. If there are n lines in a set, the top of the LRU stack is referred to as position 0 and the bottom of the stack is referred to as position *(n - 1)*. A prefetch is squashed if the line is already present in the cache.

2.2 Prefetch Classification

The references to lines in the *pf-cache* can be broadly classified into two types – those that are associated with a prefetch (i.e., references to a prefetched line or to a line that was replaced to accommodate a prefetch), and those that are not. PTMT accounts for all extra misses and traffic associated with both categories of references.

2.2.1 References to Lines Associated With a Prefetch

Suppose line x is prefetched into *pf-cache* and replaces line y. In the *pf-cache* there are only two possible next events for line x : a hit, i.e., it remains in the *pf-cache* until its next reference, or a replacement, i.e., x is replaced before its next reference. In the *conv-cache*, the next reference to line x is either a hit or a miss.

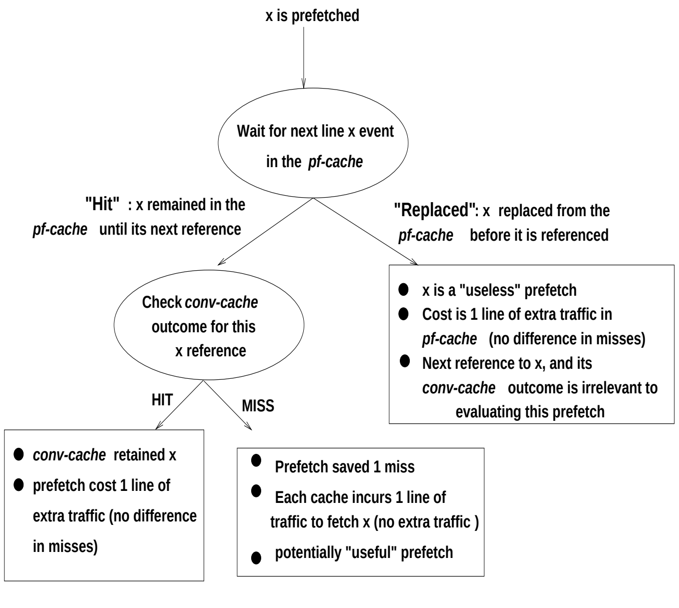

x is prefetched
Wait for next line x event in the pf-cache
"Hit" : x remained in the pf-cache until its next reference
"Replaced": x replaced from the pf-cache before it is referenced
Check conv-cache outcome for this x reference
x is a "useless" prefetch
Cost is 1 line of extra traffic in pf-cache (no 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

Figure 2: Events in *pf-cache* and *conv-cache* for line x

Figure 2 shows all the possible events for line x in the *pf-cache* and the *conv-cache* and Table 1 summarizes the extra traffic and misses associated with each combination.

When line x is a hit in the *pf-cache*, the following two possibilities arise:

- *hit-hit* : Line x is a hit in both caches. The *pf-cache* incurs 1 cache line of extra traffic to prefetch x.

- *hit-miss* : Line x is a hit in the *pf-cache* and a miss in the *conv-cache*. This appears to be a useful prefetch. However, the usefulness of this prefetch depends on what happens next to the replaced line y. Note that when the *pf-cache* incurs one fewer miss relative to the *conv-cache* we represent it as -1 extra misses.

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

Table 1: Cost of events in *pf-cache* and *conv-cache* for line x

When x is replaced from the *pf-cache* before being referenced, it does not matter whether x is present or absent in the *conv-cache* and we refer to the outcome as "don't care". Hence the only possible combination when x is replaced from the *pf-cache* is:

- *replaced-don't care* : The next reference to x in the *pf-cache* may be satisfied by another prefetch or by a demand miss. A second prefetch of x (after the replacement of the first prefetched line, but before the next reference to x) is simply classified independently of the first prefetch. Therefore, the only cost associated with the replaced x prefetch is 1 cache line of additional traffic in the *pf-cache*.

Similarly, let us consider all the possible outcomes of line y (the line that the x prefetch replaced in the *pf-cache*). In the *pf-cache*, line y is either a miss or is prefetched before its next reference. In the *conv-cache*, line y is either a hit, i.e., it remains in the cache until its next reference, or is replaced before its next reference. Figure 3 shows all the possible outcomes for line y in the *pf-cache* and the *conv-cache*; Table 2 summarizes the extra traffic and misses associated with each combination.

For ease of explanation we discuss the possible outcomes of y in the *conv-cache* prior to the possible outcomes of y in the *pf-cache*. When line y is a hit in the *conv-cache*, the following possibilities arise:

- *miss-hit* : Line y is a miss in the *pf-cache* and a hit in the *conv-cache* due to early (premature) replacement of y in *pf-cache*.
- *prefetched-hit* : If y is a hit in the *conv-cache* and is prefetched back into the *pf-cache* (after being replaced to accommodate x), the outcome of this y prefetch falls into one of the 3 cases in Table 1. So we must not attribute the cost of that y prefetch to this x prefetch.

When y is replaced from the *conv-cache* before being referenced, whether the next reference to

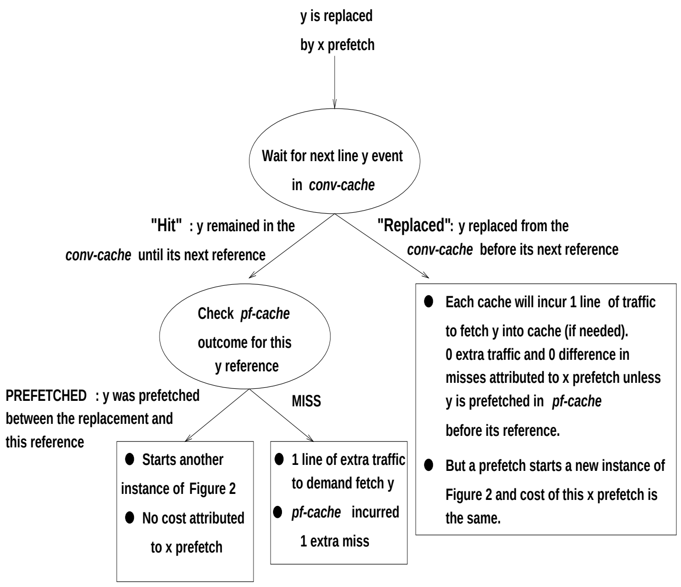

y is replaced by x prefetch
Wait for next line y event in *conv-cache*
"Hit" : y remained in the *conv-cache* until its next reference
"Replaced": y replaced from the *conv-cache* before its next reference
Check *pf-cache* outcome for this y reference
PREFETCHED : y was prefetched between the replacement and this reference
MISS
● Starts another instance of Figure 2
● No cost attributed to x prefetch
● 1 line of extra traffic to demand fetch y
● *pf-cache* incurred 1 extra miss
● 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 in *pf-cache* before its reference.
● But a prefetch starts a new instance of Figure 2 and cost of this x prefetch is the same.

Figure 3: Events in *pf-cache* and *conv-cache* for replaced line y

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

Table 2: Cost of events in *pf-cache* and *conv-cache* for replaced line y

y in the *pf-cache* is satisfied by a prefetch or demand fetch is irrelevant in evaluating this x prefetch. Hence the only possible combination is:

- *don't care-replaced* : We refer to the y reference in the *pf-cache* as "don't care" since the cost attributed to the x prefetch is based on a demand fetch/miss of y in the *pf-cache*. If y is prefetched back into the *pf-cache* the change in cost will be correctly calculated and attributed to that prefetch.

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

Table 3: Cost in traffic and miss for the 9 case pairs

Category	**Cases**
Useful Prefetches	**5, 6**
Useless Prefetches	**2, 3, 4, 8, 9**
Polluting Prefetches	**1, 7**

Table 4: Three categories of prefetches derived from the 9 case-pairs

To quantify the extra traffic and misses incurred due to a prefetch we combine the cases in Table 1 with the cases in Table 2. The 9 combined cases are presented in Table 3. These are further grouped into 3 categories based on the amount of extra traffic and misses generated and presented in Table 4.

From Table 3 we see that cases 5 and 6 save a miss and yet cause no additional traffic. These are the only *Useful Prefetches*. *Useless Prefetches* waste bandwidth (1 additional line of traffic) without saving any cache miss. Finally, *Polluting Prefetches* decrease performance by requiring 2 lines of additional traffic and causing an additional miss.

2.2.2 References to Lines Not Associated With a Prefetch

In section 2.2.1 the possible outcomes of prefetched lines and the lines they replaced were described and the associated extra traffic and misses were derived. But in an associative cache (associativity > 1) those may not be the only altered outcomes due to prefetching; the re-ordering of the LRU stack may induce side-effects on some of the regular lines that remain in the LRU stack. In this section, we derive the extra traffic and misses associated with these lines which are not directly associated with a prefetch.

Table 5 lists the possible combinations of the outcomes in the *pf-cache* and the *conv-cache* for a line z that is not associated with a prefetch.

Outcomes for z in		**Category**
pf-cache	*conv-cache*	
Hit	Hit	*PH-CH*
Miss	Miss	*PM-CM*
Miss	Hit	*PM-CH*
Hit	Miss	*PH-CM*

Table 5: Possible outcomes in both caches for lines not associated with a prefetch

For *PH-CH* and *PM-CM* outcomes of line z the *pf-cache* clearly incurs no additional traffic or misses relative to the *conv-cache*. If only these two cases occur for lines not associated with prefetches, then the 9 case pairs of Table 3 would fully account for the cost in traffic and misses of any prefetching technique. We now show that although the *PM-CH* outcomes can occur, they can easily be detected and their costs are then incorporated in the taxonomy. Finally we show that the *PH-CM* outcomes cannot occur if z is a regular line.

First consider *PM-CH*. This case is possible only when prior to this reference of z the following events occurred at some time since the last reference to z: (i) z became the LRU line in the *pf-cache* and was then replaced to accommodate a regular line; and (ii) z was retained in the *conv-cache* until this reference. This reference to z would then cost 1 additional miss and 1 cache line of additional traffic in the *pf-cache* relative to the *conv-cache*. The following simple example illustrates an occurrence of this case.

In this example the *pf-cache* and the *conv-cache* are 2-way associative. We focus on one set of these caches. Initially, as shown in Figure 4(a), suppose the contents of the set are the same in both caches (regular lines x and y, where y is the LRU line). Now if the prefetching technique initiates a prefetch for line w, we obtain the cache state shown in Figure 4(b). The *pf-cache* prefetches w and replaces y; the traffic and misses due to the prefetch can be accounted from Table 3. Note

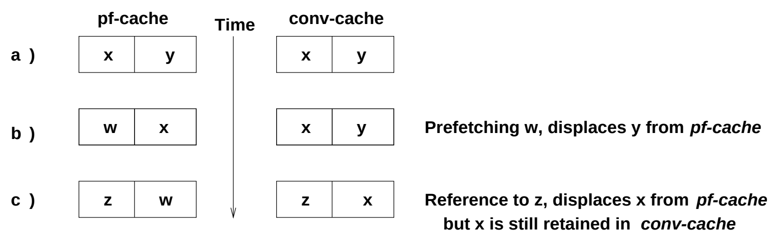

pf-cache
Time
conv-cache
a)
x
y
x
y
b)
w
x
x
y
Prefetching w, displaces y from *pf-cache*
c)
z
w
z
x
Reference to z, displaces x from *pf-cache* but x is still retained in *conv-cache*

Figure 4: Prefetch Side-effect example

that the contents of *conv-cache* do not change. While in this state, suppose the program references line z; we have a miss in both the *pf-cache* and the *conv-cache* and the resulting cache contents are shown in Figure 4(c). Finally, suppose that there is now a reference to x, which is a regular line. The outcome is a miss in the *pf-cache* and a hit in the *conv-cache*, which leads to 1 cache line of additional traffic and 1 extra miss in the *pf-cache* relative to the *conv-cache*. An ideal prefetch algorithm would not have prefetched line w and would have instead prefetched line z. The *PM-CH* case therefore an indirect consequence of poor speculative prefetching. We refer to this case, case 10, as a *Side-effect* of prefetching. An occurrence of case 10 is detected by noting when:

- Different blocks are replaced on a demand miss in both caches. For example, the demand miss to z in Figure 4 and
- the block replaced in the *pf-cache* (x in Figure 4) remains in the *conv-cache* until its next reference.

Finally, we analyze the case when the outcome of line z is *PH-CM*. The proof of Theorem 1, which depends on Lemmas 1 and 2 shows that this case cannot occur if x is a regular line. An *PH-CM* outcome is thus possible only when x is a prefetched line in the *pf-cache*.

Lemma 1 *For a given set S the number of regular lines in the* pf-cache *is never more than 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, each set of the *pf-cache* has at most the same number of regular lines as the *conv-cache*.

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

Proof: 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*. This condition satisfies the lemma. Hence it remains to show that the lemma is satisfied until the next reference to l. We do this by showing that every type of cache event either preserves $pf(l) \geq conv(l)$ or causes l to no longer be a regular line in the *pf-cache* (which satisfies the lemma until the next reference to l, since l again becomes a regular line in the *pf-cache* only upon its next reference). Let A be the associativity of set S. Consider any non-reference interval of l and let T be the initial portion of that interval during which l is a *regular* line in the *pf-cache*. Consider the following exhaustive list of possible events in T.

(a) *References to or prefetches in other sets*: These have no effect on the position of l in set S.

(b) *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.

(c) *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 of the caches, its position is defined to be A (the associativity itself).) The following possibilities arise:

1. *$pf(m) > pf(l), conv(m) > conv(l)$:* This increases the position of line l in both caches by 1, and hence $pf(l) \geq conv(l)$ is preserved after this reference to m. (Note that $pf(m)$ and/or $conv(m)$ could be A.)
2. *$pf(m) > pf(l), conv(m) < conv(l)$:* Since $conv(m) < conv(l)$, line m was referenced earlier in T. However, $pf(m) > pf(l)$ implies that l was replaced and prefetched in the *pf-cache* after the m reference. Since l is not referenced during T, l must be a prefetched line which contradicts the assumption that l is a regular line.
3. *$pf(m) < pf(l), conv(m) < conv(l)$:* This does not alter the position of l in either cache, and hence $pf(l) \geq conv(l)$ is preserved after this reference to m.
4. *$pf(m) < pf(l), conv(m) > conv(l)$:* Since $conv(m) > conv(l)$, line m was not referenced earlier in T. Considering also that $pf(m) < pf(l)$, it must be that m was prefetched earlier in T. Thus m is a prefetched line in the *pf-cache*. This is the only case in which $conv(l)$ increases by 1 and $pf(l)$ remains unchanged. However, this event occurs only after a corresponding occurrence during T of event (b) above, in which $pf(l)$ increases by 1 and $conv(l)$ remains 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.

■

Theorem 1 *The outcome that line z is a hit in the pf-cache (PH) and a miss in the conv-cache (CM) can occur only if z is a prefetched line in the pf-cache.*

Proof: Suppose line z is a hit in the *pf-cache* and a miss in the *conv-cache*. This implies that, prior to this reference of z, the stack position of line z in the *pf-cache* was less than its position in the *conv-cache* (which is, by definition, A). From Lemmas 1 and 2, this is not possible if x is a regular line. Hence the proof. ■

Theorem 1 shows that the *PH-CM* outcome cannot occur for regular lines. Thus the *PM-CH* outcome (case 10) and the 9 case pairs of Table 3, constitute a complete set of cases for the taxonomy. The PTMT case histogram thus provides a new and accurate basis for comparing prefetch techniques. Table 6 summarizes the cost in traffic and misses for the 10 cases.

Category	**Cases**	**Extra**	
		Traffic	**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

Table 6: The Prefetch Taxonomy

From Table 6 we observe that, except for case 10, the extra traffic is always one more than the extra misses. Since the 10 cases of PTMT completely and disjointly account for all the extra traffic and the extra misses of a prefetch technique, equations 3 and 4 are satisfied.

$$\text{Misses}_{\text{pf cache}} = \text{Misses}_{\text{conv cache}} - \text{useful prefetches} + \text{polluting prefetches} + \delta\text{m} \tag{3}$$

$$\text{Traffic}_{\text{pf cache}} = \text{Traffic}_{\text{conv cache}} + \text{useless prefetches} + 2 * \text{polluting prefetches} + \delta\text{t} \tag{4}$$

Since each occurrence of the only remaining case 10, causes 1 additional line of traffic and 1 extra miss relative to *conv-cache*, $\delta m = \delta t =$ number of case 10 occurrences.

2.3 Prefetch Chains

From Table 3 we observe that in cases 2, 5 and 8, the line that was replaced to accommodate a prefetch is subsequently prefetched into the *pf-cache* before its next reference. For example, let line x be prefetched and let it replace line y (referred to as *(x, y)*). If *(x, y)* falls into one of cases 2, 5 or 8, then line y must have been prefetched back into the *pf-cache* prior to its next reference, and replaced some line, say z, forming *(y, z)*. Since the *conv-cache* outcome for line y in *(x, y)* is a hit, the *conv-cache* outcome for line y in *(y, z)* has to be a hit too. Thus, *(y, z)* has to be one of cases

1, 2, or 3 only. Since cases 2, 5, and 8 require at least one more prefetch for completion, they form a part of a chain of at least 2 prefetches. Therefore, the total cost associated with a case 2, 5, or 8 prefetch should include the cost of all prefetches in the chain. Since cases 2 and 8 are useless prefetches we focus our discussion on useful chains which start with case 5 followed by case 1, 2, or 3 prefetches.

- Case 1 has a cost of 2 blocks of extra traffic and 1 additional miss which makes chain of cases (5 and 1) not favorable.
- Case 2 has a cost of 1 additional block of traffic and does not cost any extra miss, which when linked with case 5 costs a total of 1 block of extra traffic to save 1 miss. Furthermore, we showed above that case 2 must be followed by another prefetch of cases 1, 2 or 3. Thus no linked chain can end with a case 2 prefetch and the linked chain's cost depends on its length. Note that a case 2 followed by a case 1 prefetch is clearly not favorable. However, case 2 followed by cases 2 or 3 may or may not be favorable depending on the relative importance of traffic and misses. For example, if case 5 is followed by n case 2 prefetches, which is in turn followed by a chain-ending case 3 prefetch, the *pf-cache* incurs a total cost of $(n + 1)$ blocks of extra traffic to save 1 miss. This linked list thus becomes progressively less desirable as n grows.
- Case 3 has a cost of 1 additional block of traffic and does not cost any extra miss. Since case 3 replaced a block that is also replaced in the *conv-cache* before its next reference, case 3 ends the linked chain.

Therefore, the prefetch that is always beneficial is case 6. An ideal prefetching algorithm should maximize the occurrence of case 6 prefetches. It should never generate prefetches belonging to cases, 4, 7, 8, or 9. Depending on the length of the resultant chains, the prefetching technique may or may not benefit from case 5 prefetches.

3 Experimental Framework

In section 2 we discussed our classification and demonstrated that PTMT accounts for the direct and indirect costs of all the prefetches in the program. We now apply PTMT to variants of two common prefetching techniques and derive some insights into how to improve their performance.

3.1 Implementable Prefetching Algorithm

Many software and hardware methods have been proposed to predict addresses and issue prefetches. Software prefetching techniques [3, 6, 7, 11, 12, 13, 15, 18] derive hints from global program analysis and insert explicit prefetch instructions only when they are deemed likely to be useful. But existing software-based selection methods are limited to the kind of data access patterns (constants or strides) that are easiest to recognize at compile time. On the other hand, hardware prefetching techniques [1, 4, 5, 9, 10, 14, 16, 17, 19], rely on speculation about future memory-access patterns based on an observed run-time pattern history.

To illustrate the use of PTMT we chose two well-known hardware prefetching algorithms from among the above techniques.

- **Next Sequential Prefetching**

  Next sequential prefetching (NSP), [8, 19], is a simple prefetching technique in which a prefetch for line $(b + 1)$ is issued (if not already present in the cache) whenever block b is accessed. To control the number of useless prefetches, a variant called tagged prefetching [8] has been proposed. In this A tag bit associated with each cache line is set when the line is brought to cache by a prefetch access. A prefetch to the next sequential line is triggered whenever there is a cache miss or when there is a hit to a block whose tag bit is set (prefetched line). After initiating the prefetch, the tag bit is reset. In essence, this technique exploits an application's spatial locality and uses the data addresses to predict the next sequential address as the prefetch address. Our experiments (omitted due to space constraints) using NSP with and without tags indicate that the tagged variant generally has higher accuracy, but lower coverage than NSP without tags. The selective prefetching afforded by the tags does help reduce the useless and the polluting prefetches, but the overall miss rate is similar for both techniques. In this paper we use only the tagged variant of NSP.

- **Shadow Directory Prefetching**

  Shadow Directory Prefetching (SDP), proposed in [16], is a hardware based prefetching technique in which a history of the referencing patterns is maintained in a hardware table. For each line in L2, a shadow address is maintained in the L2 cache directory along with the address of the currently resident line. The shadow address refers to the next line accessed in L2 after the currently resident line was last accessed. Whenever we have an L2 cache hit, we issue a prefetch for the corresponding shadow address in the directory. To reduce the number of useless prefetches, a 1-bit confirmation scheme was proposed. A confirmation bit was added to each shadow entry in L2 cache directory indicating whether that line was referenced (1) or not referenced (0) while still in L1 after its last prefetch.

3.2 Simulation Environment and Results

The SimpleScalar simulator [2] and its functional cache simulator were used to apply PTMT to six benchmarks from SPECint95 and five from SPECfp95. The benchmarks were compiled with gcc -O3 option and run up to 1 billion instructions with the *reference* input set. As the trends of the results were similar for the different cache configurations, we present results only for data prefetching using a 16KB, 4-way associative data cache with 32 byte lines.

Benchmark	**Miss rate** (%)		**Coverage**		**Accuracy** (%)		**Misses** (10^6 **lines**)		**Prefetches** (10^6 **lines**)	
	NSP	**SDP**	**NSP**	**SDP**	**NSP**	**SDP**	**NSP**	**SDP**	**NSP**	**SDP**
compress	11.0	9.5	0.20	0.32	26.3	69.9	11.8	10.3	10.0	6
gcc	2.8	2.8	0.40	0.30	47.2	64.4	7.0	7.0	7.9	4.4
go	2.8	2.7	0.25	0.22	30.3	42.0	7.8	7.4	6.7	4.1
ijpeg	0.7	1.6	0.69	0.20	80.2	42.0	1.2	2.9	2.8	1.5
li	2.1	3.8	0.59	0.23	71.8	45.4	1.2	2.0	2	1.3
perl	1.7	1.8	0.35	0.18	47.4	63.2	4.3	4.4	3.8	1.5
apsi	5.0	5.5	0.35	0.31	36.6	49.5	11.8	12.9	16.5	10.8
applu	1.2	5.5	0.80	0.02	98.1	17.2	2.7	12.6	10.5	1.8
turb3d	1.3	1.9	0.54	0.29	59.6	54.8	2.9	4.4	5.5	3.2
swim	27.3	21.8	0.08	0.25	10.3	73.1	70.8	56.9	55.2	24.4
wave	5.7	7.1	0.45	0.31	48.6	61.4	11.6	14.3	18	9.9

Table 7: Miss rate, coverage, accuracy, misses and prefetches for NSP and SDP

Table 7 shows the miss rate, coverage, accuracy, misses, and prefetches for NSP and SDP. On average, NSP has higher coverage and lower accuracy than SDP. For benchmarks with sequential access patterns (e.g., *ijpeg, li, applu, turb3d*), NSP outperforms SDP. SDP is generally far more selective than NSP as seen from the total traffic ($misses\ +\ prefetches$) in Table 7, (SDP has only 79% of NSP traffic for SPECint and 87% for SPECfp benchmarks) implying that most of the NSP traffic is useless. For example, in *perl*, the miss rate is down to 1.7% for NSP from 1.8% for SDP, but the total traffic is up from 5.9 for SDP to 8.1 million lines for NSP. These metrics do not separately quantify the extra traffic and extra misses due to polluting and useless prefetches and hence they do not provide much insights for improving the techniques.

Figure 5 shows the distribution of prefetches into the 3 categories identified using our taxonomy. From Figure 5 we see that SDP has a higher fraction of useful prefetches than NSP (except for *ijpeg, li, applu, turb3d*). confirming the lower accuracy for NSP. The additional traffic of NSP is wasted on polluting and useless prefetches (e.g., in *go* the polluting prefetches are up from 10% for SDP to 13% for NSP). *Swim* exhibits a classic example of polluting and useless prefetches

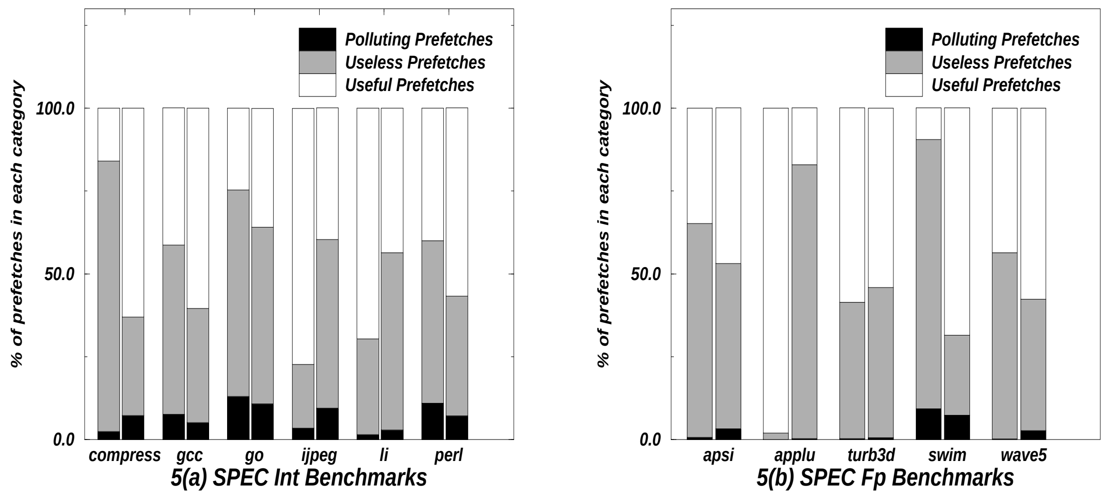


Figure 5: Classification of prefetches for NSP and SDP

(left bar = NSP, right bar = SDP)

(Total Prefetches in each scheme normalized to 100)

outweighing the effects of the useful prefetches (useless prefetches are up from 24% for SDP to 81% for NSP).

Benchmark	**Prefetch Side-Effects**	
	NSP	**SDP**
compress	27,190	439,852
gcc	374,931	216,086
go	581,908	394,167
ijpeg	55,390	103,897
li	23,702	37,237
perl	273,032	97,386

Benchmark	**Prefetch Side-Effects**	
	NSP	**SDP**
apsi	14,771	218,877
applu	5,739	4,317
turb3d	610	15,473
swim	16,418	9,418
wave	35,994	5,984

Table 8: Prefetch Side-Effects (Case 10)

Table 8 presents the number of side-effect cases identified in section 2 showing indirectly the effect of speculative prefetching on the LRU stack. There are very few case 10 occurrences compared to the total prefetches (0.03 to 0.04 per prefetch) as seen from Table 7.

From the histogram of the prefetches by category we identified the need for a filter to decrease the polluting prefetches. We have proposed such a filter and use PTMT to evaluate its effectiveness in the next section.

4 Static Filter

We have proposed a Static Filter (SF) [20] which attempts to reduce polluting prefetches so as to minimize the impact of the prefetching technique on the bandwidth requirements and at the same time to retain the achieved reduction in miss rate.

For a given prefetching technique an access to line x is called a *prefetch trigger* of line y, if the access to x may initiate a prefetch of y. Most hardware prefetching techniques use data addresses as prefetch triggers (e.g., NSP uses an access to line x as a trigger to prefetch line $x + 1$). However, many of these schemes use additional enabling mechanisms to qualify the trigger – i.e., to actually perform the prefetch when its trigger occurs *only* if the enable condition is set. For example, NSP does not prefetch line $x + 1$ if the access to x is not a cache miss. Our *Static Filter* (SF) uses profiling to provide an additional enable mechanism that must also be satisfied. The profile is used to determine which load instructions tend to generate data references that are useful prefetch triggers, and marks those loads as *enabled*. At run time SF disables the hardware prefetch mechanism unless the load making a trigger access is enabled. Since the characteristics of the load instructions of the program tend to be stable across runs, profiling is an effective way to select enabled loads.

In the profiling phase of SF, we simulate a conventional cache without prefetching; whenever a line is brought into cache we maintain (or create) an entry in a table (we used an LRU table of 2K entries for our experiments) that records the current time and the PC of the load instruction that accessed the data. Whenever there is a demand miss to cache line y we check if at least one of its *prefetch triggers*, x, is present in the table. If x is present[1], we know the time that x was most recently brought into cache, $t2$, and the load instruction, $L1$, that accessed x to begin that cache tour. The prefetching technique **P**, would have prefetched y along with x at time $t2$. To determine if that prefetch is *potentially useful*, we check the time $t1$ of the most recent access to z, the line chosen for replacement now (to accommodate y): If $t2 > t1$ the prefetch is deemed *potentially useful*; if $t2 < t1$ it is not.

We use a heuristic to select loads to be enabled. For each load instruction we find the ratio of the number of misses it incurs to the number of *potentially useful* prefetches triggered by the load. If the potentially useful prefetches are more than 50% of the misses we mark this load instruction as enabled.

In the implementation phase, the baseline technique **P** with SF initiates a prefetch *only* if the prefetch triggering data access is issued by an enabled load. Our results have shown that SF achieves

[1] If x is not present, we do not have the necessary information and assume that by default the prefetch of y is *not potentially useful*.

a significant reduction in the traffic requirements of the prefetching techniques while preserving the reduction in miss rate that they achieve without SF. In the context of this paper, SF shows how our metrics can be used to improve the quality of existing prefetch techniques.

4.1 Results using Static Filter

We present the miss rate in conjunction with the total traffic to illustrate the tradeoffs between misses and bandwidth. The the usefulness and accuracy of prefetches are assessed via the PTMT prefetch histogram.

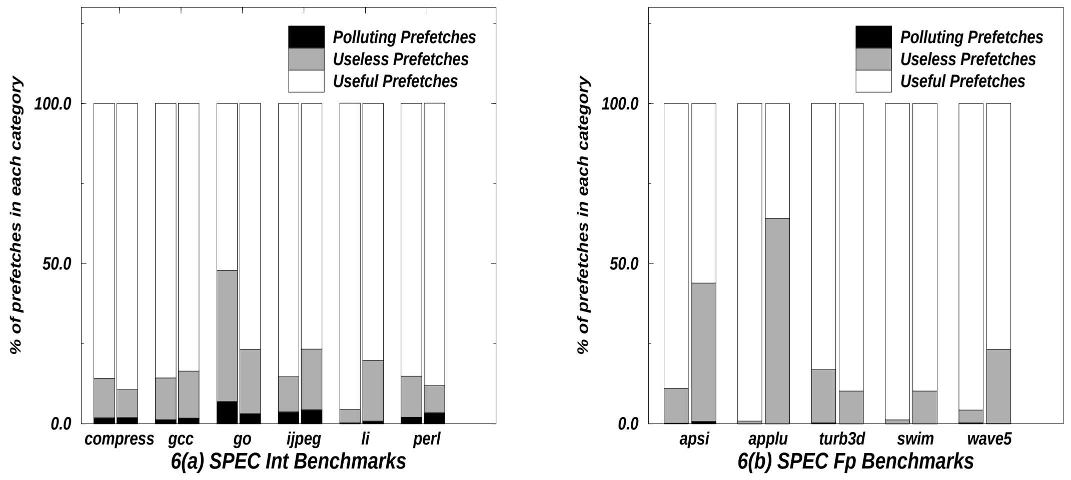


Figure 6: Classification of prefetches for SF-NSP and SF-SDP
(left bar = SF-NSP, right bar = SF-SDP)
(Total Prefetches in each scheme normalized to 100)

- **Prefetch Traffic Analysis**

  Figure 6 shows the improved prefetch histogram when SF is applied to NSP and SDP. Furthermore, as the total prefetches issued by SF-NSP and SF-SDP differ significantly, each is far less than those issued using NSP and SDP alone, the improvement in the histogram is even more significant.

  For both SPECint and SPECfp benchmarks, SF-NSP has only about 2 to 3% *polluting prefetches* (e.g., in *go*, the polluting prefetches 2% for SF-NSP, vs. 13% for NSP). Overall, SF-NSP is dominated by *useful prefetches*, indicating the improvement due to the selectivity of SF. However, for some benchmarks like *go*, *li* and *apsi*, SF may be too selective and decrease coverage.

Similarly, SF-SDP has only 2 to 3% *polluting prefetches*. Unlike SF-NSP, we see that SF-SDP, unfortunately, also decreases the total number of *useful prefetches*, and this will increase the miss rate. (as seen in Figure 10). SF's reduction in *polluting prefetches* and hence in total traffic will not be effective unless SF preserves the decrease in miss ratio achieved by NSP or SDP alone.

- **Total Traffic**

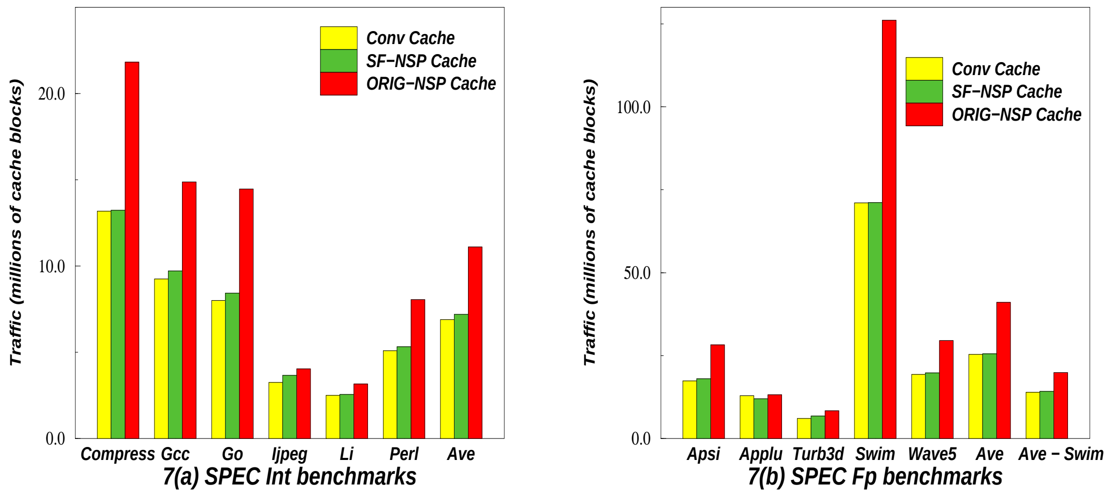


Figure 7: NSP : Traffic(millions of blocks): Cache: 16KB, 4-way, 32B blocks

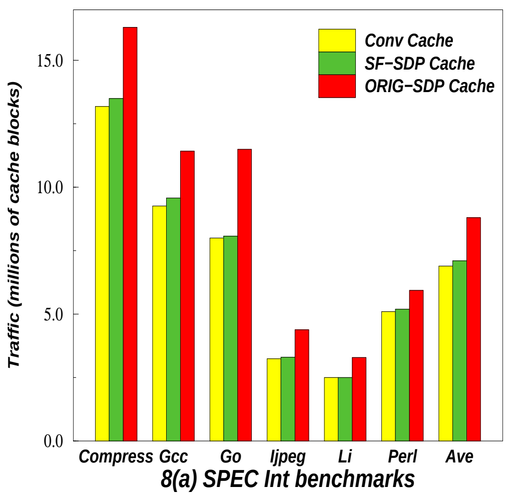
In all the subsequent figures the leftmost bar is the result for conventional cache, the second bar is for a cache using SF on a baseline technique and the rightmost bar is for a cache using baseline technique alone.

Figures 7(a) and 7(b) present the total traffic for the SPECint and SPECfp benchmarks using NSP as the baseline technique. Using SF we observe an average of 30% reduction in traffic for SPECint benchmarks. Although SPECfp benchmarks are known to have sequential access patterns, SF still achieves an average of 25% reduction in traffic (excluding *swim*). Here and in the subsequent figures, the last group of bars shows the average results excluding *swim*. For *swim* NSP has almost twice as much traffic as SF-NSP. This leads to more cache pollution, as we will see when we compare the corresponding miss rates for NSP and SF-NSP in Figure 9(b).

Figures 8(a) and 8(b) present the total traffic using SDP as the baseline technique. Since SDP is more selective than NSP, we do not achieve the same the improvement using SF on SDP as on NSP. SF reduces traffic by 20% for SPECint and 15% for SPECfp benchmarks.

- **Miss Rate**

Figures 9(a) and 9(b) show the miss rate for SPECint and SPECfp benchmarks using SF on



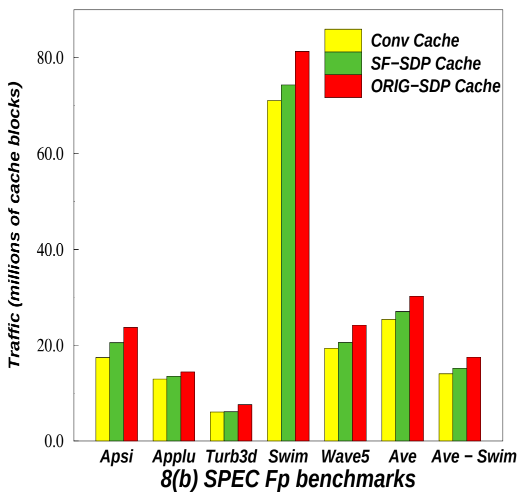
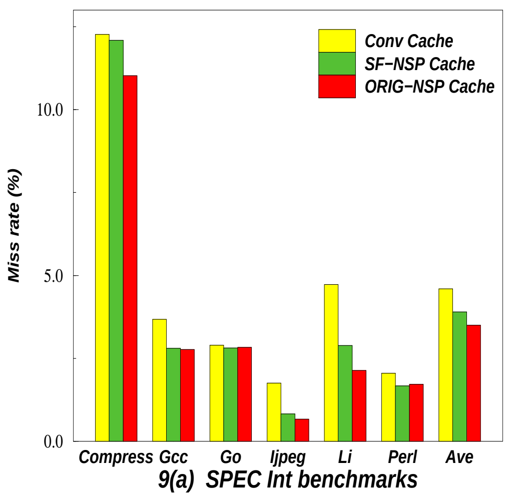


Figure 8: SDP : Traffic(millions of blocks): Cache: 16KB, 4-way, 32B blocks



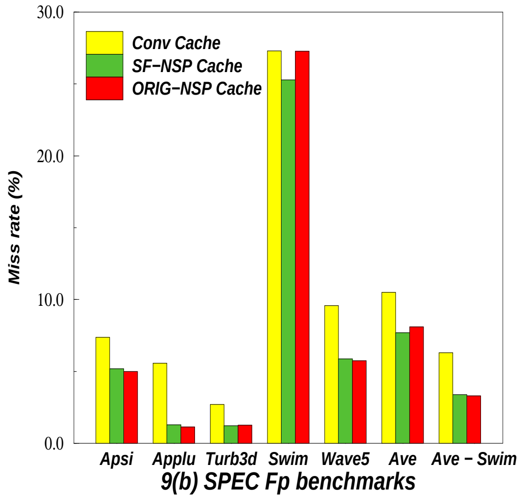
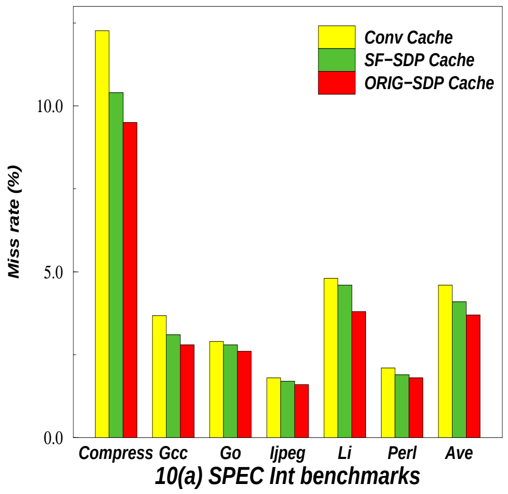


Figure 9: NSP: Miss rate: Cache: 16KB, 4-way, 32B blocks

NSP. NSP achieves an average miss rate reduction of 24% for SPECint and 46% for SPECfp relative to a conventional cache. SF-NSP reduces the miss rate by 20% and 48% for SPECint and SPECfp respectively, indicating that we are not eliminating very many useful prefetches by using SF. From Figures 5 and 6, average useful prefetches for SPECint are up from 45% for NSP to 81% for SF-NSP. However, for benchmarks with predominantly sequential access patterns, (e.g., *li* and *ijpeg*) SF may be too selective On the other hand for *gcc* and *go* where selectivity is more important, the NSP miss ratio is well preserved. A clear trade-off between misses and traffic is seen for *compress*; using NSP a 10% reduction in miss rate is achieved with a 40% increase in traffic relative to SF-NSP; this may or may not be desirable depending on the available bandwidth.

The high miss rate (average of 8% without prefetching) of SPECfp, vs. the SPECint benchmarks, suggests that a 16KB cache may be too small to capture the working set of these benchmarks. This makes prefetching more critical for improving performance; NSP and SF-NSP have comparable performance. For *swim*, NSP has more misses than SF-NSP due to cache pollution caused by excessive prefetching. This is confirmed by our observation in Figure 7(b) that the NSP:SF-NSP prefetch traffic ratio for *swim* is very high (about 10 times).



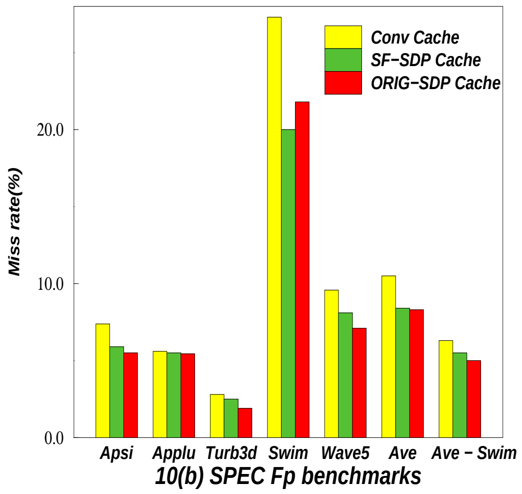


Figure 10: SDP : Miss rate: Cache: 16KB, 4-way, 32B blocks

Figures 10(a) and 10(b) show the miss ratio using SDP as the baseline technique. Neither SDP nor SF-SDP shows a significant improvement in miss rate for these benchmarks. SF-SDP has a 10 to 15% increase in the miss rate relative to SDP. However, SDP issues 70 to 80% more prefetches than SF-SDP in order to achieve this reduction in miss rate. SF-SDP performs as well as SF-NSP for SPECint benchmarks, but not as well for SPECfp benchmarks.

Our results using NSP and SDP as baseline techniques show that PTMT did identify the useless and polluting prefetches and that SF did significantly reduce these harmful prefetches while retaining nearly all of the useful prefetches of the baseline techniques. Thus SF does, as claimed, reduce the bandwidth requirement of a baseline technique while preserving its reduction in miss rate.

5 Conclusion

PTMT analyzes each prefetch and assigns it 0, 1 or 2 blocks of additional traffic and a -1, 0, or +1 net change in misses that it causes relative to a conventional cache. The key to this classification is the simultaneous simulation of two caches – a cache with prefetching and a conventional cache. The three major contributions in this paper are,

- A classification that completely describes the 9 possible outcomes that encompass a prefetched line and the line it evicts from the cache.
- A classification which further describes the effect of prefetching on the LRU ordering within a cache set (case 10) and the indirect effect this has on the miss rate and traffic.
- An Analysis of the chaining effects among prefetches so as to understand how even a useful prefetch (case 5) may not be beneficial in a broader context.

Using the above classification we derived new metrics to evaluate a given prefetching technique by quantifying the additional traffic penalty that each prefetch incurs in order to *potentially* save a cache miss and hide the memory access latency.

We have demonstrated the usefulness of PTMT by applying it to two implementable prefetching techniques (NSP and SDP). Based on the prefetch classification observed for NSP and SDP we developed a static filter to reduce/eliminate polluting prefetches. Our results show that SF eliminates nearly all the polluting prefetches and achieves a significant reduction in the prefetch traffic of the baseline techniques with only a modest increase in miss rate.

References

[1] J. Bennett, M. Flynn, "Prediction Caches for Superscalar Processors", *Proceedings of the 30th Annual Int'l Symposium on Microarchitecture, December 1997, pp 81-91.*

[2] D. Burger, T. Austin, "The SimpleScalar Tool Set, Version 2.0," *Technical Report TR 1342, University of Wisconsin, Jun 1997.*

[3] D. Callahan, K. Kennedy, A. Porterfield, "Software Prefetching," *Proceedings of the 4th Symposium on Architectural Support for Programming Languages and Operating Systems, April 1991, pp 40-52.*

[4] M. Charney, T. Puzak, "Prefetching and memory system behavior of the SPEC95 benchmark suite," *IBM Journal of Research and Development, Vol 41, Number 3, May 1997.*

[5] T. Chen, J. Baer, "Reducing Memory Latency via Non-blocking and Prefetching Caches," *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, October 1992, pp 51-61.*

[6] T. Chen, J. Baer, "A Performance Study of Software and Hardware Data Prefetching Schemes," *Proceedings of the 21st Annual Int'l Symposium on Computer Architecture, April 1994, pp 223-232.*

[7] W. Chen, S. Mahlke, P. Chang, W. Hwu, "Data Access Microarchitectures for Superscalar Processors with Compiler-Assisted Data Prefetching," *Proceedings of the 24th Int'l Symposium on Microarchitecture, November 1991, pp 69-73.*

[8] J.D. Gindele, "Buffer Block Prefetching Method," *IBM Tech. Disclosure Bull. 20, 2, July 1977, pp 696-697.*

[9] D. Joseph, D. Grunwald, "Prefetching Using Markov Predictors," *Proceedings of the 24th Int'l Symposium on Computer Architecture, May 1997, pp 252-263.*

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

[11] A Klaiber, H Levy, "An Architecture for Software-Controlled Data Prefetching," *Proceedings of the 18th Int'l Symposium on Computer Architecture, May 1991, pp 43-53.*

[12] M. Lipasti, W. Schmidt, S. Kunkel, R.Roediger, "Spaid: Software Prefetching in pointer and call intensive environments," *Proceedings of the 28th Annual Int'l Symposium on Microarchitecture, November 1995, pp 231-236.*

[13] C-K Luk, T. Mowry, "Compiler based Prefetching for recursive data structures," *Proceedings of the 7th Int'l Conference on Architectural Support for Programming Languages and Operating Systems, October 1996, pp 222-233.*

[14] S. Mehrotra, L. Harrison, "Examination of a Memory Access Classification scheme for Pointer-Intensive and Numeric Programs," *Proceedings of the 10th Int'l Conference on Supercomputing, May 1996, pp 133-139.*

[15] T. Mowry, M. Lam, A. Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching," *Proceedings of the 5th Int'l Conference on Architectural Support for Programming Languages and Operating Systems, October 1992, pp 62-73.*

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

[17] A. Roth, A. Moshovos, G. Sohi, "Dependence Based Prefetching for Linked Data Structures," *Proceedings of the 7th Int'l Conference on Architectural Support for Programming Languages and Operating Systems, October 1997, pp 115-126.*

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

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

[20] reference removed for double blind reviewing