A Static Filter for Reducing Prefetch Traffic

Viji Srinivasan, Gary S. Tyson Edward S. Davidson
Advanced Computer Architecture Lab, EECS department,
University of Michigan, Ann Arbor
(sviji@eecs.umich.edu, davidson@eecs.umich.edu, tyson@eecs.umich.edu)

Abstract

The growing difference between processor and main memory cycle time necessitates the use of more aggressive techniques to reduce or hide main memory access latency. Prefetching data into higher speed memories is one such technique. However, speculative prefetching can significantly increase memory traffic.

We present a new technique, called Static Filtering (SF), to reduce the traffic generated by a given hardware prefetching scheme while preserving its reduced miss rate. SF uses profiling to select which load instructions should be marked "enabled" to do data prefetching. This is done by identifying which load instructions generate data references that are useful prefetch triggers. SF enables the hardware prefetch mechanism only for the set of references made by "enabled" loads. Our results from applying SF to two well-known hardware prefetching techniques, Next Sequential Prefetching (NSP) and Shadow Directory Prefetching (SDP), shows that SF preserves the decrease in misses that they achieve and reduces the prefetch traffic by 50 to 60% for NSP and by 64 to 74% for SDP. In addition, timing analysis reveals that when finite memory bandwidth is a limiting factor, applying SF does in fact increase the speedup obtained by a baseline hardware prefetching technique.

The other major contribution of this paper is a complete taxonomy which classifies individual prefetches in terms of the additional traffic they generate and the resulting reduction (or increase) in misses. This taxonomy provides a formal method for classifying prefetches by their usefulness. A histogram of the prefetches by category provides a new basis for comparing prefetch techniques.

1 Introduction

Memory access latencies are much larger than processor cycle times, and this gap has been increasing over time. Prefetching has been shown to be an effective approach to hiding large memory latencies. The success of speculative prefetching techniques relies on the accuracy of their address predictions. However, due to misspeculation many useless prefetches may be issued. Such overly aggressive prefetching techniques may expose a more fundamental limit to performance, the memory *bandwidth* bottleneck; the available bandwidth limits the rate of traffic between different levels of the memory hierarchy. As the total traffic of a program run increases, eventually the available bandwidth between the cache and memory becomes a performance bottleneck.

Several software and hardware methods have already been proposed to reduce the number of useless prefetches. Software prefetching techniques 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 can be recognized at compile time. Hardware prefetching techniques, on the other hand, tend to be far more aggressive in triggering prefetches (for example, prefetch the next sequential block on every cache miss). To reduce useless prefetches, these hardware techniques typically rely on simple 1-bit confidence indicators to dynamically turn off prefetching when the prefetch is deemed likely to be useless. These selective techniques increase prediction accuracy by issuing fewer prefetches. However, this increased accuracy may come at a cost of decreased coverage, which increases the misses, thereby defeating the original purpose of prefetching. Effective prefetching thus depends on achieving good miss coverage with sufficient accuracy to avoid polluting the cache and saturating the memory bus with useless prefetches.

In this paper, we propose a new technique, *Static Filtering* (SF), to decrease useless prefetches for a given hardware technique while preserving its ability to reduce cache misses. SF uses profiling to statically select which load instructions should by enabled to trigger data prefetching. Using the profile information, we augment the given hardware prefetching technique to initiate a prefetch associated with a demand load only if the load has been marked "enabled" by SF. We show that by doing the selection statically (via profiling) we can improve upon the prefetch techniques that use purely dynamic confidence mechanisms. In this paper, we apply SF on two well known prefetching techniques: *Next Sequential Prefetching* (NSP) and *Shadow Directory Prefetching* (SDP). Our results indicate that SF is an effective filter to reduce the useless prefetches generated by these techniques while retaining the achieved reduction in misses.

The other major contribution of this paper is a complete taxonomy for classifying prefetches. Consider the traffic and misses of a cache with some prefetch technique relative to a conventional cache that does not prefetch. This taxonomy 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 is causes. More importantly, the sum of the extra traffic (misses) based on the taxonomy and the traffic (misses) of the conventional cache matches the total traffic (misses) of the cache with the prefetch technique, thereby making the taxonomy complete. This taxonomy therefore leads to new metrics to evaluate a given prefetching technique by quantifying the penalty of additional traffic incurred to *potentially* save a cache miss and hide the memory access latency. A histogram of the prefetches by category provides a new basis for comparing prefetch techniques.

The rest of this paper is organized as follows. Section 2 describes the *Static Filter*. Section 3 describes the prefetch taxonomy. In section 4, *Next Sequential Prefetching* and *Shadow Directory Prefetching*, the baseline techniques used in this study are described, followed by the simulation environment and the benchmarks used to evaluation SF applied to these techniques. Section 5 presents and discusses simulation results. We present related prior work in the area of data prefetching in section 6. Section 7 concludes the paper and mentions some promising directions for future work.

2 Trading Misses For Bandwidth

The primary goal of prefetching is to hide memory access latency by eliminating demand misses. However, to maximize the benefits of prefetching, it is important to minimize the number of useless prefetches that degrade the performance by polluting the cache and increasing bus contention. Cache pollution can be mitigated to some extent by adding separate prefetch buffers and probing them in parallel with the cache. However, using prefetch buffers does not decrease the additional traffic generated by useless prefetches and bus saturation due to this traffic can limit system performance. In this paper, we consider prefetching directly into the cache.

Figure 1 sketches typical performance curves as a function of memory bus bandwidth for an

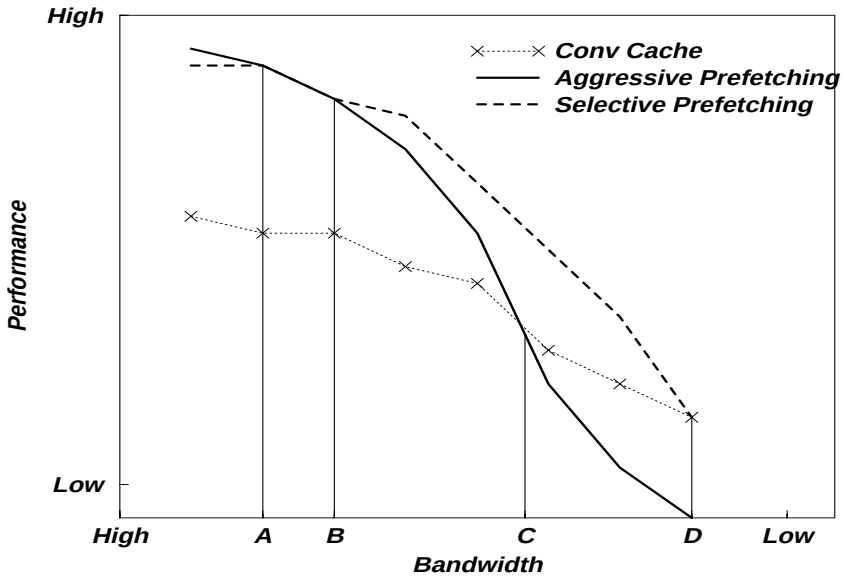

Figure 1: System performance vs Memory bus bandwidth

aggressive data prefetching technique, a selective prefetching technique and a conventional cache with no prefetching. When effectively unlimited bandwidth is available, aggressive prefetching techniques can achieve the best performance, as seen in the Figure 1 prior to point A. As the available bandwidth is decreased, the performance of an aggressive technique deteriorates faster than a more selective prefetching technique, as in the region from point B to point C in Figure 1. As the bandwidth becomes more limiting, the bus becomes a performance bottleneck and aggressive prefetching can result in lower performance than no prefetching at all, as in the C to D region of Figure 1. Eventually, for example past D in Figure 1, even selective prefetching may be worse than none at all. Aggressive techniques perform well with unlimited bus bandwidth; however, in reality, aggressive techniques can easily saturate the bus with excessive traffic. Therefore, it becomes critical to to reduce the number of useless prefetches, while retaining prefetches that do contribute to performance improvement.

2.1 The Static Filter

For a given prefetching technique an access to block x is called a *prefetch trigger* of block y, if a prefetch of y can be initiated when we access x. Most hardware prefetching techniques use data addresses as prefetch triggers. For example, in NSP an access to block x is a prefetch trigger of block $x + 1$. However, most schemes use some additional enable 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 block $x + 1$ if the access to x is not a cache miss. *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 mark 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 can be used to select enabled loads. We now describe the two phases of SF — profile and implementation.

2.1.1 Profile Phase

The profile phase of SF proceeds in two steps. In the first step, we identify the data addresses that were most successful prefetch triggers for the hardware technique. In the second step, we enable the load instructions that generate references to those useful prefetch triggers.

If we had complete knowledge of future data references, we could identify successful prefetch triggers as follows: Suppose the baseline technique **P**, issues a prefetch for cache block x that is next referenced at time $t1$. Suppose further that x replaces the current least recently used block y whose next reference is at time $t2$. Now, there are two possibilities.

- *Case 1:* $t2 < t1$:
  The prefetched block will definitely pollute the cache by replacing a more useful cache block.
- *Case 2:* $t2 > t1$:
  Prefetching x is *potentially useful* because, by evicting y early, we can bring a more useful block into cache.

In reality, even in the profile phase we cannot efficiently gain knowledge of future references to cache blocks. We have therefore implemented an approximate technique to identify successful prefetch triggers in the profile phase based on the history of access patterns.

In the profiling phase of SF, we simulate a conventional cache without prefetching; whenever a block is brought into cache we maintain (or create) an entry in a table (we used a 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 block 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 (say, $t2$) and the load instruction (say, *L1*) that accessed x to begin that cache tour. **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 block chosen for replacement now (to accommodate y) as shown in Figure 2. If $t2 > t1$ the prefetch is deemed *potentially useful*; if $t2 < t1$ it is not.

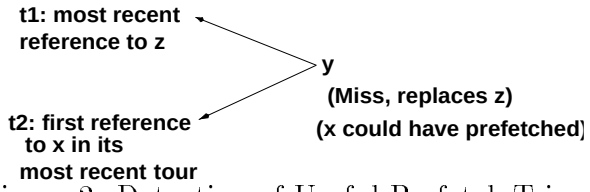

Figure 2: Detection of Useful Prefetch Trigger

For each load instruction we can now find the ratio of the number of misses incurred 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 enabled.

2.1.2 Implementation Phase

In the implementation phase, the baseline technique **P** with SF issues a prefetch *only* if the prefetch triggering data access is caused by an enabled load. Our results show that by so restricting the prefetches we achieve a significant reduction in the traffic requirements of **P** while preserving the reduction in miss rate that **P** achieves without SF.

Static Filter requires the load instruction to be marked "enabled" for data prefetching. In this paper, we use profiling and collect the list of enabled load instructions and use a hardware

[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*.

table at run-time to store these candidate load instructions. However, we can also use 1-bit of the load opcode to indicate whether the load is a candidate for issuing prefetches. This requires a modification to the existing ISA; however, modern processors including Pentium-3, PA-8500 and AMD K6-3 have special instructions for prefetching data into the cache. With the current trend of more ISAs providing support for prefetch instructions, it is to be expected that in the future the compiler will be used to give more prefetch hints to the program using bit(s) of the opcode. We are simply proposing that these prefetch instruction can also be used to improve the performance of the hardware prefetch mechanisms.

3 Prefetch Taxonomy

Prior work in prefetching has generally measured the effectiveness of prefetching techniques using coverage and accuracy. These metrics show the overall performance of a prefetch algorithm. However, they do not provide insight into the usefulness of each prefetch issued by the algorithm. Such insights are essential for providing meaningful comparisons among prefetch techniques, understanding the underlying causes of their performance, and gaining knowledge that helps derive new more effective techniques.

In this section, we introduce a systematic classification of all prefetches according to their usefulness. We analyze the effect of each prefetch on the memory traffic and the number of cache misses. Using the taxonomy we can classify all the prefetches issued into disjoint categories and and analyze their benefits. To achieve this classification, we simultaneously model two identically configured caches, a conventional cache without any prefetching, *conv-cache*, and a cache with some prefetching technique of interest, *pf-cache*. The taxonomy is used to quantify the amount of extra traffic generated by each prefetch and the number of misses saved (or incurred) due to that prefetch.

We measure the traffic between the primary cache and the next level of the memory hierarchy (say L2) as the number of cache blocks that are transferred from L2 to the primary cache.

Suppose block x is prefetched into *pf-cache* and replaces block y. Prefetching x causes one block of traffic for *pf-cache*. The usefulness of this traffic can be assessed by using the cache access outcome (hit/miss) for the next reference to x and to y in both the *pf-cache* and *conv-cache*. The following four categories are obviously disjoint and complete.

- x remains in the *pf-cache* until its next reference and is therefore a hit in the *pf-cache* but y is replaced before its next reference and therefore misses in the *conv-cache*: x is a useful prefetch and replaced a block (y) that is not useful.
- x is a hit in the *pf-cache* and y is a hit in the *conv-cache*: x is a useful prefetch; however, it was prefetched too early because it evicted another useful block, y.
- x is replaced in the *pf-cache* and y is replaced in the *conv-cache*: x is a useless prefetch and caused additional traffic.
- x is replaced in the *pf-cache* and y is a hit in the *conv-cache*: x is not only a useless prefetch; it also caused cache pollution.

However, if x is a hit in the *pf-cache* and y is a hit in the *conv-cache*, it is not straight-forward to quantify the usefulness of the prefetch in terms of the extra traffic generated and the miss saved. It becomes complicated to analyze this case because the outcome of y's access in the *pf-cache* and the outcome of x's access in the *conv-cache* are not considered in the above classification.

Our taxonomy described below is sufficiently refined to be able to precisely account for the extra traffic and the misses saved or incurred that should be attributed to each prefetch.

In the *pf-cache* there are only two possible outcomes for block x; it is either a hit or is replaced before its next reference. When block x is a hit in the *pf-cache*, it can have either a hit or a miss in the *conv-cache*. On the other hand, when block x is replaced before being referenced in the *pf-cache*, the next reference of x may be satisfied by another prefetch or demand fetch. A second prefetch of x (after the replacement of the first prefetched block, but before the next reference to x) is simply classified independently of the first prefetch into one of the above two cases. Observe that independent of the outcome of the next reference to x in the *conv-cache*, we can conclude that an x prefetch that is replaced causes one additional block of traffic in *pf-cache* relative to the *conv-cache*; therefore, we refer to the x reference in *conv-cache* as "don't care." All the possibilities can be enumerated as follows; summary is presented in Table 1.

- The next reference to x is a hit in both *pf-cache* and *conv-cache*. Based on the outcome of y's next reference we have the following 3 cases.
  - **Case 1**: *pf-h-m/conv-h-h*: The next reference of y is a hit in the *conv-cache* and a miss in the *pf-cache*. Therefore, the *pf-cache* has 2 additional blocks of traffic (1 block for prefetching x and 1 block for fetching y) and has 1 additional miss relative to the *conv-cache*.
  - **Case 2**: *pf-h-pfed/conv-h-h*: y is a hit in the *conv-cache* and in the *pf-cache* y is prefetched before the time of its next reference (some other case will begin with that y prefetch, and will account for its cost). Therefore, *pf-cache* has 1 additional block of traffic (for prefetching x) and has no additional misses relative to a *conv-cache*.
  - **Case 3**: *pf-h-dc/conv-h-repl*: y gets replaced in the *conv-cache* before its next reference. As a consequence we need not check the outcome of y's next reference in the *pf-cache*. Therefore, *pf-cache* has 1 additional block of traffic (for prefetching x) and has no additional misses relative to a *conv-cache*.
- The next reference to x is a hit in the *pf-cache* and is a miss in the *conv-cache*. As above, considering all possible outcomes for y's next reference leads to the following 3 cases.
  - **Case 4**: *pf-h-m/conv-m-h*: The next reference to y is a hit in the *conv-cache* and a miss in the *pf-cache*. Therefore, the *pf-cache* has 1 additional block of traffic and has no additional misses relative to the *conv-cache*.
  - **Case 5**: *pf-h-pfed/conv-m-h*: y is a hit in the *conv-cache* and y is prefetched in the *pf-cache* before its next reference (some other case will begin with that y prefetch and account for its cost). Therefore, the *pf-cache* has no additional blocks of traffic and saves a miss relative to the *conv-cache*.
  - **Case 6**: *pf-h-dc/conv-m-repl*: y gets replaced from the *conv-cache* before its next reference. As a consequence we need not check the outcome of y's next reference in the *pf-cache*. Therefore, the *pf-cache* has no additional blocks of traffic and saves a miss relative to the *conv-cache*.
- x gets replaced before its next reference in the *pf-cache* and as mentioned earlier the outcome of x's next reference in the *conv-cache* is irrelevant.
  - **Case 7**: *pf-repl-m/conv-dc-h*: The next reference to y is a hit in the *conv-cache* and a miss in the *pf-cache*. Therefore, the *pf-cache* has 2 additional blocks of traffic and 1 additional miss relative to the *conv-cache*.
  - **Case 8**: *pf-repl-pfed/conv-dc-h*: y is a hit in the *conv-cache* and y is prefetched in the *pf-cache* before its next reference (some other case will begin with that y prefetch and account for its cost). Therefore, the *pf-cache* has 1 additional block of traffic and has no additional misses relative to the *conv-cache*.

– **Case 9**: *pf-repl-dc/conv-dc-repl*: y gets replaced from the *conv-cache* before its next reference. As a consequence we need not check the outcome of y's next reference in the *pf-cache*. Therefore, the *pf-cache* has 1 additional block of traffic and has no additional misses relative to the the *conv-cache*.

We observe that the next reference to block y may be a hit in the *pf-cache* only if it is prefetched later after being replaced now. It is important to note that in each of the 9 cases, *pf-cache* has at least as much traffic as *conv-cache*. The above taxonomy accounts for the extra traffic and extra misses associated with every prefetch and we derive the following 3 groups from the 9 unique case pairs.

- Polluting Prefetches : The prefetches which generate 2 additional blocks of traffic and 1 additional miss relative to the conventional cache clearly cause cache pollution and degrade performance. Cases 1 and 7 above fall into this category.
- Useless Prefetches : The prefetches which generate 1 additional block of traffic and no additional misses relative to the conventional cache are issued too early and cause wasted bandwidth. These are not as harmful as the *polluting prefetches*. Cases 2, 3, 4, 8 and 9 above fall into this category.
- Useful Prefetches: Case 5 and 6 comprise all the useful prefetches. They cause no additional traffic and save a miss relative to the conventional cache. Only these two cases are useful. However, each occurrence of case 5 is clearly linked with some other case involving the later prefetch of y into the *pf-cache*. In addition, since the next reference to y hits in the *conv-cache*, the case for this later y prefetch must be 1, 2 or 3. Such linked chains of cases, beginning with case 5 may not be useful. However, case 6 is always useful.

From the 9 cases we observe that the extra traffic is always one more than the extra miss. Note that when the *pf-cache* incurs one less miss relative to the *conv-cache* we represent it as -1 extra misses. If the 9 cases presented above completely and disjointly accounted for all the extra traffic and all the extra misses, the following 2 equations would be satisfied.

$$\text{Misses}_{\text{pf cache}} = \text{Misses}_{\text{conv cache}} + \text{polluting prefetches} - \text{useful prefetches} \quad (1)$$

$$\text{Traffic}_{\text{pf cache}} = \text{Traffic}_{\text{conv cache}} + 2 * \text{polluting prefetches} + \text{useless prefetches} \quad (2)$$

We do know from the way that the 9 cases and their costs are constructed, that they are disjoint. However, our simulation studies have shown that this 9 case classification, although it completely covers all prefetches, does not completely attribute all extra misses and traffic to them. Specifically, we have not accounted for δm extra misses and δt extra traffic in the *pf-cache*, Intriguingly, the simulations always indicated that $\delta m = \delta t$. A final (non-prefetched) case accounts for δm and δt and completes the taxonomy.

Consider the following example with the *pf-cache* and the *conv-cache* having the same contents (blocks x and y) as shown in Figure 3(a). Block y is the least recently used block.

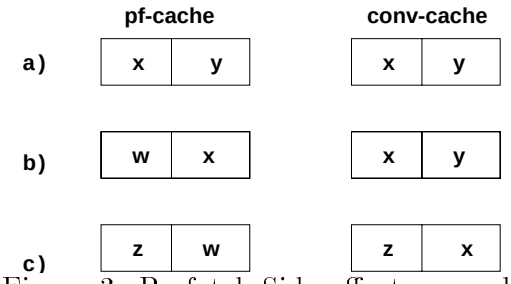

Figure 3: Prefetch Side-effect example

Now if the prefetching technique initiates a prefetch for block w, we obtain the cache state shown in Figure 3(b). The pair *(w, y)* falls into one of the 9 case-pairs identified above. Note that the contents of *conv-cache* does not change. While in this state, suppose the program references block z; we have a miss in both the *pf-cache* and the *conv-cache* and the resulting cache contents are shown in Figure 3(c). Finally, suppose that there is now a reference to x; it is a miss in the *pf-cache* and a hit in the *conv-cache*. This miss leads to 1 extra block of traffic and 1 extra miss in the *pf-cache* relative to the *conv-cache*. An ideal prefetch algorithm would not have prefetched block w and would have instead prefetched block z. This is therefore a consequence of poor speculative prefetching. We refer to the occurrences of this case as *case 10: prefetch side-effects* in our results. 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 3 and
- the block replaced (x in Figure 3) in the *pf-cache* is subsequently hit (before being replaced) in the *conv-cache*.

In every simulation that we have run, the number of case 10 occurrences exactly equals *deltam* extra = δt as shown in equations 3 and 4. Thus we conjecture that the 10 case taxonomy is complete.

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

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

Category	Cases		Extra Traffic	Extra Misses
Polluting Prefetches	**(Case 1)**	*pf-h-m/conv-h-h*	2	1
	(Case 7)	*pf-repl-m/conv-dc-h*	2	1
Useless Prefetches	**(Case 2)**	*pf-h-pfed/conv-h-h*	1	0
	(Case 3)	*pf-h-dc/conv-h-repl*	1	0
	(Case 4)	*pf-h-m/conv-m-h*	1	0
	(Case 8)	*pf-repl-pfed/conv-dc-h*	1	0
	(Case 9)	*pf-repl-dc/conv-dc-repl*	1	0
Useful Prefetches	**(Case 5)**	*pf-h-pfed/conv-m-h*	0	-1
	(Case 6)	*pf-h-dc/conv-m-repl*	0	-1
Prefetch Side-effect	**(Case 10)**		1	1

Table 1: The Prefetch Taxonomy

In our results, we use the above taxonomy to quantify how well SF reduces the useless and polluting prefetches and how well it preserves the useful prefetches. Discriminating between useless and polluting prefetches separates the fraction of prefetches that cause only additional traffic from the fraction that cause additional misses as well as additional traffic.

4 Experimental Framework

We now briefly describe the two baseline techniques, *Next Sequential Prefetching* (NSP) and *Shadow Directory Prefetching*(SDP) that we will use to illustrate the effectiveness of SF. This is followed by a discussion of the simulation environment and the benchmarks used in this study.

4.1 Next Sequential Prefetching

Next sequential prefetching (NSP), proposed in [20], is a simple prefetching technique in which a prefetch for block $(b + 1)$ 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 is also proposed. In this variant, a tag bit is associated with each cache block. This bit is set when the block is brought to cache by a prefetch access. A prefetch to the next sequential block is triggered when there is a cache miss or when there is a hit to a block whose tag bit is set; after initiating the prefetch the tag bit is reset. In essence, this technique exploits the spatial locality exhibited by the applications and uses the data addresses to predict the physically contiguous successor as the prefetch address. In this paper we use only the tagged variant of NSP.

These simple precise rules for *when* to issue a prefetch (on a demand miss or on the first access to the block after being prefetched) and *what* to prefetch (the next consecutive cache block) make NSP easy to implement. However, when the application does not exhibit sequential memory access patterns, NSP wastes memory bandwidth and potentially leads to cache pollution.

4.2 Shadow Directory Prefetching

Shadow Directory Prefetching (SDP), proposed in [15], is a hardware based prefetching technique in which a history of the referencing pattern is maintained in a hardware table. A shadow address is maintained in the L2 cache directory along with the currently resident address. The shadow address refers to the block accessed after the currently resident block was last accessed. Whenever we have a 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 is used. A confirmation bit is added to each directory entry in L2 cache indicating whether the prefetched block was used (1) or not used (0) when it was last prefetched.

SDP relies on the repetition of data access patterns and on a stable relationship between the resident address and the shadow address. SDP exploits the correlation between the resident address and the shadow address, to predict prefetch addresses. The advantage of SDP is that it can a prefetch data that do not exhibit sequential or stride reference behavior. However, if the application has a large working set or if the application's data access patterns are not repetitive, SDP can lead to an excessive number of useless prefetches.

4.3 Simulation Environment

For our simulations, we used the SimpleScalar simulator [2]. The functional cache simulator of the tool set was used for the profile phase and the implementation phase. We used the detailed out-of-order timing simulator to evaluate our technique. We modeled a 4-way superscalar processor with a traditional 5 stage pipeline and a bi-modal branch predictor with 2K entry table. We simulated 4 integer ALUs, 4 floating point adders, 2 integer and floating point multiply/divide units. We assumed a perfect L1 instruction cache. We simulated a 16 KB L1 data cache. We used different block sizes (8, 16 and 32 bytes) and different associativities (1, 2 and 4). We found that the trend of the results was the same for all degrees of associativity, and so we present results only for a 4-way associative cache with 32 byte blocks. The L1 cache was accessed in 1 cycle and the memory was accessed in 30 cycles for the first 8 bytes, and 2 cycles for each 8 bytes thereafter. The processor used 2 read/write ports and a 16-entry load-store queue.

In the implementation phase of SF, we simulated 5 caches — a conventional cache with no prefetching, a cache with a given baseline hardware technique (NSP or SDP), and a cache with the baseline technique enabled by SF (SF-NSP or SF-SDP).

We present results for both integer and floating point benchmarks, namely, six benchmarks from SPECint95 and five benchmarks from SPECfp95 (Due to time constraint we have not been able to collect results for the other benchmarks of the SPEC suite). All these benchmarks were

compiled with gcc -O3 option. We used the *train* input set for our profile phase and the *ref* input set for our implementation phase and limited the execution to 1 billion instructions.

5 Results

We use the following metrics to evaluate the usefulness of SF.

- **Total Traffic :** The total number of cache blocks transferred from L2 to the primary cache. This is measured as the sum of the misses and the total prefetches issued.
- **Prefetch Traffic Analysis:** The usefulness and accuracy of prefetches are illustrated by the distribution of prefetches based on case-pairs identified in Table 1 in section 3.
- **Miss rate:** The ratio of misses to total number of data references. Miss Ratio in conjunction with total traffic illustrates the tradeoff between misses and bandwidth for the baseline techniques with and without SF.
- **Normalized Speedup:** The speedup achieved by the prefetching technique relative to the conventional cache.

5.1 Analysis of Next Sequential Prefetching with SF

5.1.1 Total Traffic

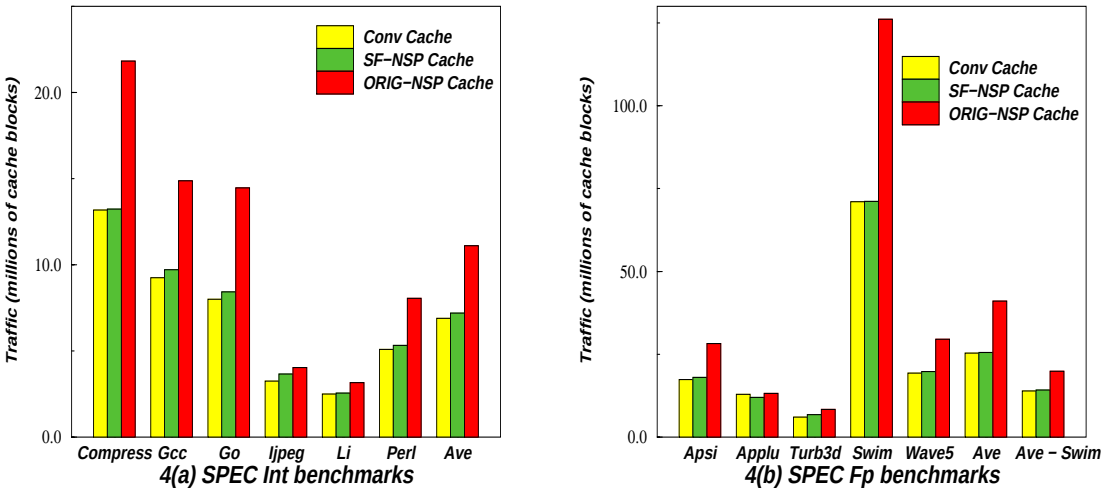

Figure 4: Traffic(millions of blocks): Cache: 16KB, 4-way, 32B blocks

Figure 4(a) and Figure 4(b) present the total traffic for SPECint and SPECfp benchmarks. In each group of bars, the leftmost bar is the traffic in the conventional cache, the second bar is for a cache using SF-NSP technique and the rightmost bar is for a cache using original NSP technique.

Using SF we observe an average of 30% reduction in traffic for integer benchmarks. Floating point benchmarks are known to have sequential access patterns and NSP is a very effective prefetching technique for these benchmarks. However, SF still achieves an average of 25% reduction in traffic for the floating point benchmarks (excluding swim). The last group of bars shows the average traffic 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 6(b).

5.1.2 Prefetch Traffic Analysis

Although total traffic is a measure of the number of blocks of data transferred from L2 to the primary cache, it does not provide insights about the prefetch traffic. Figure 5 shows the prefetch traffic based on our taxonomy. The results are normalized with respect to SF-NSP. For each benchmark, the first bar gives the distribution for SF-NSP, the second bar for NSP. The height of the second bar is proportional to the ratio of prefetches of NSP relative to SF-NSP. The difference in heights is the ratio of the extra prefetch traffic generated by NSP without SF.

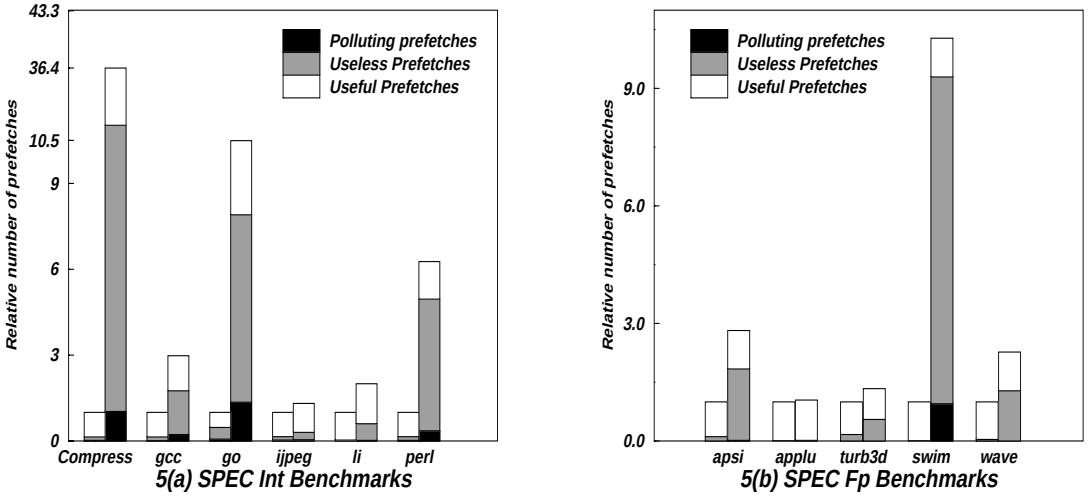

Figure 5: Prefetch Traffic Analysis: Cache: 16KB, 4-way, 32B blocks
(left bar = SF-NSP (normalized to 1), right bar = NSP)

For both the integer and floating point benchmarks, we observe that SF-NSP has only about 2 to 3% *polluting prefetches*. This suggests that using a filter like SF, we may eliminate the need for separate prefetch buffers and instead bring prefetches directly into the cache with more confidence. In the case of integer benchmarks their irregular access patterns cause *polluting prefetches* in NSP; which is eliminated using SF. For example, *go* with NSP has 12% *polluting prefetches*; which is reduced to 2% using SF-NSP.

NSP has a large fraction of *useless prefetches*. We see that *compress* with NSP has 43 times as many prefetches as with SF-NSP and 82% of these prefetches are useless. In the case of *gcc* 51% of the prefetches are *useless prefetches* using NSP and only 13% are useless using SF-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, as we will see in Figure 6. We observe that for *swim* NSP has 10 times as many prefetches as SF-NSP. However, SF's reduction in traffic will not be effective unless SF preserves the decrease in miss ratio achieved by NSP.

Table 2 presents the number of occurrences of the side-effect case identified in section 3. Note that each occurrence of the side-effect case incurs 1 additional block of traffic and 1 additional miss relative to the conventional cache. We see that issuing fewer prefetches using SF we have in general (except for *turb3d* and *wave*) decreased the total number of side-effect cases.

Benchmarks	Prefetch Side-Effects	
	SF-NSP	NSP
compress	14,118	27,190
gcc	45,436	374,931
go	71,774	581,908
ijpeg	23,461	55,390
li	3,509	23,702
perl	21,428	273,032

Benchmarks	Prefetch Side-Effects	
	SF-NSP	NSP
apsi	731	14,771
applu	5,183	5,739
turb3d	9,094	610
swim	48	16418
wave	37,081	35,994

Table 2: Prefetch Side-Effect Cases

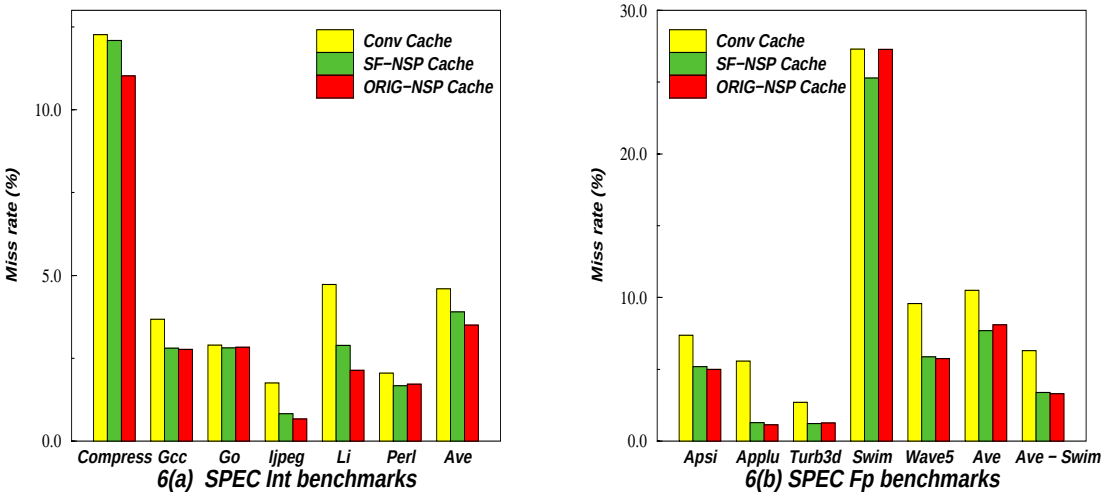

Figure 6: Miss rate: Cache: 16KB, 4-way, 32B blocks

5.1.3 Miss Rate

Figure 6(a) and Figure 6(b) show the miss ratio for SPEC int and SPEC fp benchmarks respectively. In each group of bars, the leftmost bar is the miss rate in the conventional cache, the second bar is for a cache using SF-NSP technique and the rightmost bar is for a cache using original NSP technique.

We observe that relative to a conventional cache, NSP achieves an average of 24% and 46% reduction in misses for integer and floating point benchmarks respectively. On the other hand, SF-NSP reduces the miss rate by 20% and 48% for the integer and floating point benchmarks respectively; which implies that we are not eliminating very many useful prefetches by using SF. However, for *li* and *ijpeg*, NSP performs better than SF-NSP because of their predominantly sequential data access patterns for which SF is too selective, while NSP has many *potentially useful* prefetches. On the other hand for *gcc* and *go* where selectivity is more important, the NSP miss ratio is well preserved. SF-NSP even has a lower miss rate than NSP for *gcc* because the additional *useful prefetches* in NSP were more than offset by the *polluting prefetches*. A clear trade-off between misses and traffic is seen when we compare SF-NSP and NSP results for *compress*; using NSP a 10% reduction is miss rate is achieved with a 40% increase in traffic relative to SF-NSP.

The floating point benchmarks have considerably higher miss rates than the integer benchmarks. For a conventional cache with no prefetching, the average miss ratio is 8%. This implies that a 16KB cache may be too small to capture the working set of these benchmarks. This makes prefetching more critical for improving performance and both NSP and SF-NSP have comparable performance. Since the miss rate of *swim* (27%) is more than twice that of the other benchmarks we have shown the last group of bars excluding *swim* from the average. In fact, for *swim* NSP has more misses than SF-NSP because of cache pollution due to the excessive prefetching in NSP. This is confirmed by our observation in Figure 5(b) that the NSP:SF-NSP prefetch traffic ratio for *swim* is very high (about 10 times).

5.1.4 Normalized Speedup

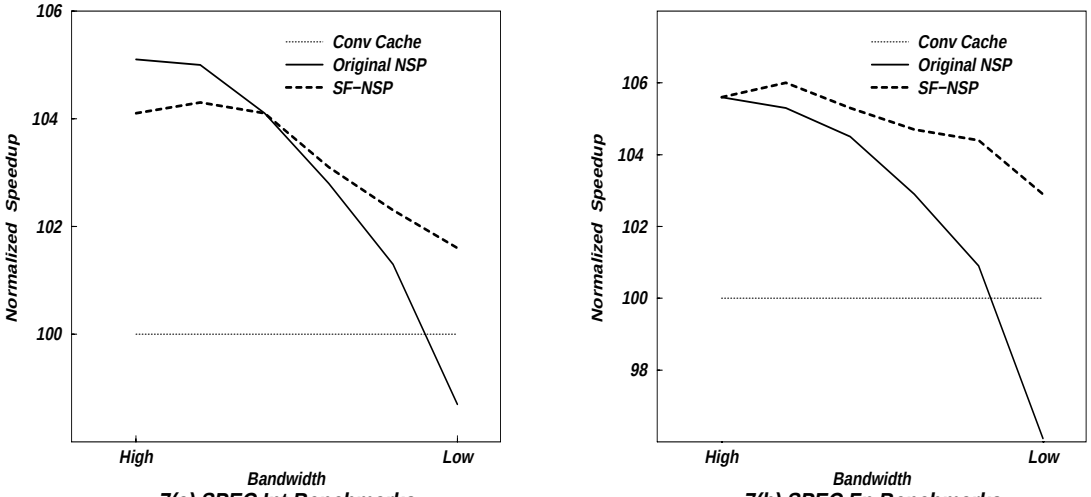

Figure 7: Normalized Speedup (Conventional Cache speedup = 1)

Results from the previous sections show that SF significantly reduces wasted bandwidth of NSP while preserving the achieved reduction in miss rate. Finally, we present the improvement in

execution time using NSP and SF-NSP. Figure 7 shows the speedup normalized with respect to the conventional cache. We present the average speedup for integer and floating point benchmarks. In this experiment, we varied the available bus bandwidth over a wide range: We started with a design where the bus is fully pipelined and can take a new request from the processor each cycle. We refer to this as *high* bandwidth in the X-axis of Figure 7. We modified this design restricting access to the bus to account for bus arbitration overhead, multi-cycle data transfer and more limited pipelining. In the most restrictive configuration, we assume that the bus is non-pipelined with the bus blocked for the duration of the data transfer. We refer to this as *low* bandwidth in the X-axis of Figure 7.

Figure 7(a) shows the average normalized speedup for the integer benchmarks. In the case of a very high bandwidth system we observe that NSP has a 5% speedup relative to the conventional cache; SF-NSP has 4% speedup. As we restrict the bus access, we see that SF-NSP has 3 to 4% speedup and NSP has 2% speedup. As the system becomes limited by the available bandwidth we see that the performance of NSP degrades faster than SF-NSP.

In the case of floating point benchmarks, the original NSP performs very well and by applying SF, we have eliminated almost all the pollution prefetches and hence the overall speedup increases from 5% to 6% with SF.

5.2 Analysis of Shadow Directory Prefetching with SF

To demonstrate that SF can be applied to a more sophisticated hardware prefetching technique we now present results for our second baseline technique SDP. For the results presented in this section, the baseline SDP uses 1-bit confirmation and SF-SDP does not use any dynamic confirmation and relies on SF to eliminate useless prefetches.

5.2.1 Total Traffic

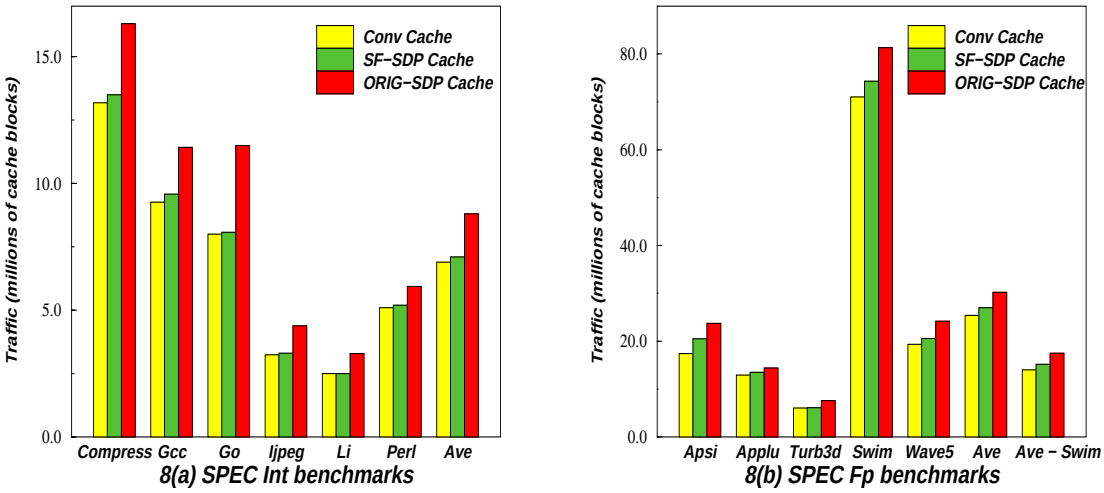

Figure 8: Traffic (millions of blocks): Cache: 16KB, 4-way, 32B blocks

Figure 8(a) and Figure 8(b) present the total traffic. As in the case of NSP, the leftmost bar is the traffic in the conventional cache, the second bar is for a cache using SF-SDP technique and the rightmost bar is for the cache using original SDP technique. SDP is a more sophisticated prefetching technique that achieves similar performance as NSP with fewer prefetches. SDP has

only 79% of NSP traffic for integer benchmarks and 87% for floating point benchmarks. Hence, we do not achieve the same the improvement using SF on SDP as on NSP. Using SF we observe an average of 20% reduction in traffic for integer benchmarks and 15% for floating point benchmarks. As in the case of NSP, *swim* has a very high traffic and hence the last group of bars of Figure 8(b) show the average traffic excluding *swim*.

5.2.2 Prefetch Traffic Analysis

As with NSP, we use our taxonomy to analyze the usefulness of each prefetch. Figure 9 shows results based on our classification of prefetches. The results are normalized with respect to SF-SDP. For each benchmark, the first bar gives the distribution for SF-SDP, the second bar for SDP. The height of the second bar is proportional to the ratio of prefetches of SDP relative to SF-SDP. Again, the difference in heights show the amount of extra prefetch traffic generated by SDP without SF.

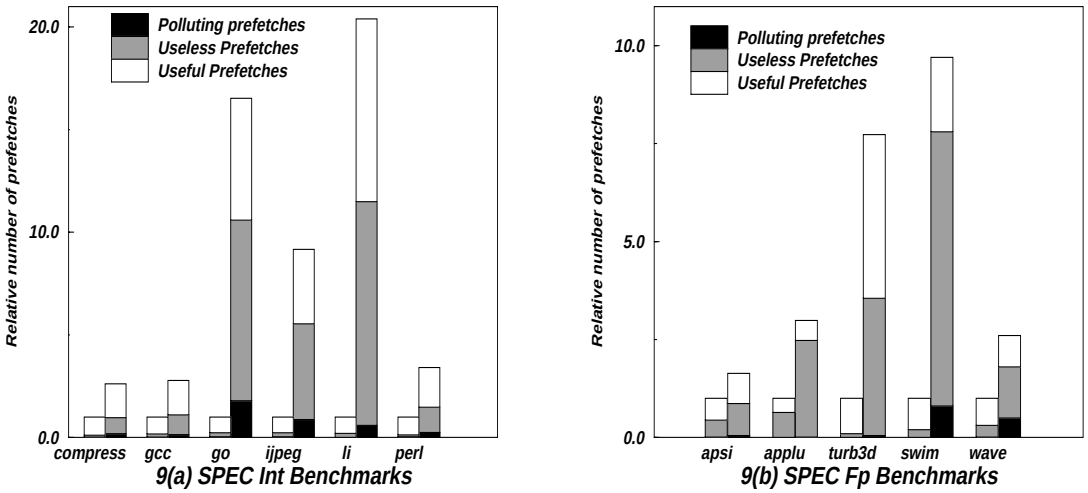

Figure 9: Prefetch Traffic Analysis:Cache: 16KB, 4-way, 32B blocks
(left bar = SF-SDP (normalized to 1), right bar = SDP)

As with NSP, we observe that SF reduces the the *polluting prefetches* of SDP to 2 to 3%. However, in the case of *go* and *li* SF-SDP also decreases the *useful prefetches* and this may decrease the coverage as we will see in Figure 10(a). For floating point benchmarks, we observe that SDP has a very small fraction of *polluting prefetches*. Except for *swim* and *wave*, the other benchmarks do not have any *polluting prefetches*.

SDP has a considerable fraction of *useless prefetches*. In the case of *li* and *applu* using SDP we see 53% and 82% *useless prefetches* respectively; using SF-SDP it is reduced to 19% and 64% respectively. NSP performs well for these benchmarks exploiting their sequential access patterns.

SF-SDP is dominated by *useful prefetches*, indicating the improvement due to the selectivity of SF. However, as opposed to SF-NSP, we see that SF-SDP decreases the total number of *useful prefetches* too and this will increase the miss rate as we will see in Figure 10(b).

Table 3 presents the number of occurrences of the side-effect case identified in section 3. We see that issuing fewer prefetches using SF we have consistently decreased the total number of side-effect cases relative to SDP.

Benchmarks	Prefetch Side-Effects	
	SF-SDP	SDP
compress	67,927	439,852
gcc	40,535	216,086
go	13,993	394,167
ijpeg	10,215	103,897
li	1,184	37,237
perl	17,715	97,386

Benchmarks	Prefetch Side-Effects	
	SF-SDP	SDP
apsi	154,686	218,877
applu	255	4,317
turb3d	1,430	15,473
swim	480	9418
wave	7,021	5,984

Table 3: Prefetch Side-Effect Cases

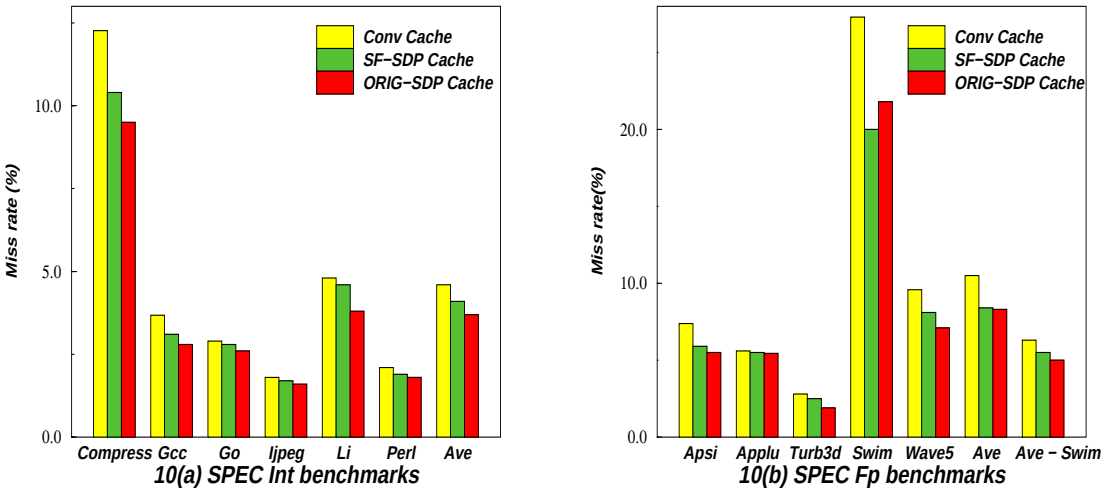

Figure 10: Miss rate: Cache: 16KB, 4-way, 32B blocks

5.2.3 Miss Rate

Figure 10(a) and Figure 10(b) show the miss ratio. Again, in each group of bars, the leftmost bar is the miss rate in the conventional cache, the second bar is for a cache using SF-SDP technique and the rightmost bar is for a cache using original SDP technique. We observe that both SDP and SF-SDP do not show significant improvement in miss rate for these benchmarks. We see that SF-SDP has 10 to 15% increase in the miss rate relative to SDP. However, we see that SDP issues 70 to 80% more prefetches relative to SF-SDP to achieve this reduction in miss rate.

For integer benchmarks SDP performs as well as NSP; however, for floating point benchmarks it does not perform as well as NSP. For floating point benchmarks, the shadow address is not predictable due to the streaming behavior of the applications and hence the prefetches are predominantly not useful. Again we observe that *swim* has a very high miss rate and we use the last group of bars to show the average miss rate excluding *swim*. It is important to note SF performs comparable to the purely dynamic technique of using 1-bit confirmation.

5.2.4 Normalized Speedup

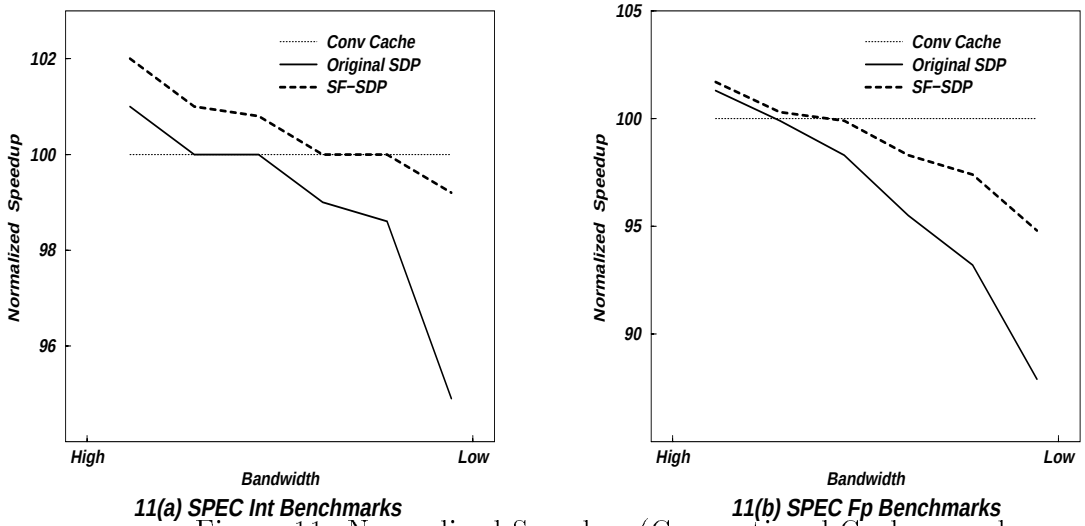

Figure 11: Normalized Speedup (Conventional Cache speedup = 1)

Figure 11 shows the speedup normalized with respect to the conventional cache. The experimental framework is similar to the one used for NSP. Here again, we observe that SF-SDP has better performance relative to SDP when we have a finite bandwidth. However, we see that the speedup using SDP is less than that using NSP. Correspondingly, the speedup using SF-NSP is more than that using SF-SDP.

Our results using NSP and SDP as baseline techniques show that SF is an effective filter in reducing useless and polluting prefetches while retaining the useful prefetches of the baseline techniques.

6 Related Work

Several methods exist for prefetching data into cache. Grouping of consecutive memory words into a cache block is itself one form of prefetching. Caches exploit spatial locality by this implicit prefetching of data. The key tradeoff in prefetching between miss rate and traffic manifests itself

even in this primitive form of prefetching: large cache blocks directly cause more memory traffic as they contain more data that will not be used during a particular cache tour and cache blocks that are too small or too large cause higher miss rates, which in turn leads to increased memory traffic. Furthermore, adapting to application behavior would be helpful as different applications and different parts of an application exhibit different degrees of spatial locality and hence have different optimal block sizes.

The critical issues of prefetching are *when* to issue the prefetch and *what* data to prefetch. In considering the question of *when* to prefetch, we note that data that is prefetched too early might replace a useful block and may even get replaced before it is referenced. On the other hand, a late prefetch hides less of the access latency, or none at all. Regarding *what* to prefetch, clearly the data prefetched should be useful and should not pollute the cache. To do effective prefetching, we need to carefully select both *what* data to prefetch and *when* to issue each prefetch. In addition it is important to evaluate block prefetch strategies over a range of block sizes to assess the degree of spatial locality found in different applications.

Software prefetching uses some special form of *fetch* instruction to preload data into cache. Most software prefetching work ([8], [9], [12], [16], [19]) has focused on scientific applications with regular access patterns. For applications using linked data structures and recursive data structures, techniques have been proposed ([10], [11], [14], [18]) to tolerate latency by speculatively scheduling a load sufficiently in advance. Although software prefetching can be done statically using compiler analysis it does not perform well for applications with access patterns that are less easy to predict. It has the disadvantage of software overhead, both in the number of instructions issued and in code size. This is one of the reasons why we did not apply SF to a software-based baseline technique.

Hardware-based prefetching ([4], [1], [3], [7], [13]) relies on speculation about future memory-access patterns based on previous patterns. Most hardware techniques work well for applications with stride access patterns. More recently techniques have been proposed ([17]) for prefetching linked data structures using dependence analysis and have a prefetch engine run ahead of the execute engine to detect possible prefetch candidates. Markov Prefetching ([6]) predicts cache misses based on previous miss patterns. It is an evolution of shadow-directory prefetching where a miss address has more than one successor miss addresses. Our evaluation of Markov predictors on SPECint benchmarks showed a significant increase in bandwidth requirements relative to shadow directory prefetching. These hardware techniques do not require changes to existing binaries, and hence need no programmer or compiler intervention. However, they do involve considerable hardware logic to detect access patterns at run-time. In addition, hardware prefetching techniques do not have access to global program knowledge and hence cannot control prefetching decisions very well when presented with a variety of data access patterns. We expect a technique like SF can be used to selectively issue prefetches based on the bus contention or available bandwidth. SF can be extended and applied to the above hardware techniques to improve their prefetch accuracy and reduce their bandwidth requirements.

Hybrid prefetching [5] tries to combine the low run time overhead of hardware prefetching with the effectiveness and accuracy of software prefetching, but the only reported evaluations are on scientific applications.

7 Conclusions and Future Work

Our contributions in this paper are two-fold:

- A complete taxonomy for classifying each prefetch, and measuring its *usefulness*.
- A static filter for reducing useless prefetches in any given baseline prefetching technique.

The taxonomy 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. This leads to new metrics to evaluate a given prefetching technique by quantifying the penalty of additional traffic incurred to *potentially* save a cache miss and hide the memory access latency.

The *Static Filter*, introduced in this paper can be used to reduce the bandwidth requirements of any given hardware baseline prefetching technique, while preserving its achieved reduction in miss rate. SF analyzes the data access patterns of an application and uses the result to prefetch *selectively* and reduce wasteful traffic. Prior work in issuing selective prefetches rely on the dynamic confidence mechanisms based on the history between the data addresses of the application. In this paper we have demonstrated that by statically selecting a subset of load instructions to enable data prefetching, we can reduce useless prefetches more effectively without degrading the overall performance. To illustrate the advantages of SF, we chose traditional *Next Sequential Prefetching* and *Shadow Directory Prefetching* as the baseline techniques.

Our results show that SF achieves significant reduction in prefetch traffic of the baseline techniques (50 to 60% for NSP, 64 to 74% for SDP) without overly increasing the miss rate. In addition, timing analysis reveals that when finite memory bandwidth is a limiting factor, applying SF does in fact increase the speedup obtained by a baseline hardware prefetching technique.

In this paper we focused on providing information derived from profiling to a hardware based prefetching technique. Our results indicate that hardware based prefetching techniques benefit from SF. However, the threshold used to determine whether a load is marked "enabled" for data prefetching was fixed statically. In addition, the analysis was done separately for each cache configuration. To adapt SF dynamically to various cache configurations and to use a dynamic threshold, run-time information about the usefulness of a resident block in cache could be used.

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] 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*

[4] 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.*

[5] 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.*

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

[7] 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.*

[8] 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.*

[9] S Vander Wiel, D. Lilja, "When Caches Aren't Enough: Data Prefetching Techniques", *IEEE Computer Magazine, July 1997, pp 23-30.*

[10] 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.*

[11] 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.*

[12] 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.*

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

[14] 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.*

[15] 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.*

[16] 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.*

[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] C. Selvidge, "Compilation-Based Prefetching for Memory Latency Tolerance", *Ph.D Thesis, MIT, May 1992.*

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