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 or co strides) that can be recogni ed at compile time. Hardw are prefetching techniques, on the other hand, tend to be far more aggressive in triggering prefetches for example, prefetc sequential block on every cache miss). To reduce useless prefetches, these hardware techniques typically rely on simple bit con dence indicators to dynamic ally turn off 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. ffec tive prefetching thus depends on achieving good miss coverage with sufficient accuracy to avoid polluting the cache and saturating the memory bus with useless prefetches.

n this paper, a new technique, *Static Filtering* S), to decrease useless p for a given hardware technique while preserving its ability to reduce cache misses. S uses pro to statically select which load instructions should by enabled to trigger data prefetching. sing the pro le information, the given hardware prefetching technique to initiate a prefetch associated with a demand load only if the load has been marked enabled e show. b that by doing the selection statically via pro can improve upon the prefetch techniques that use purely dynamic con dence mec n this paper, S w wo well known prefetching techniques *Next Sequential Prefetching* *Shadow Directory Prefetching* S P). ur results indicate that S to reduce the useless prefetc is designed by these techniques while retaining the achieved reduction in misses.

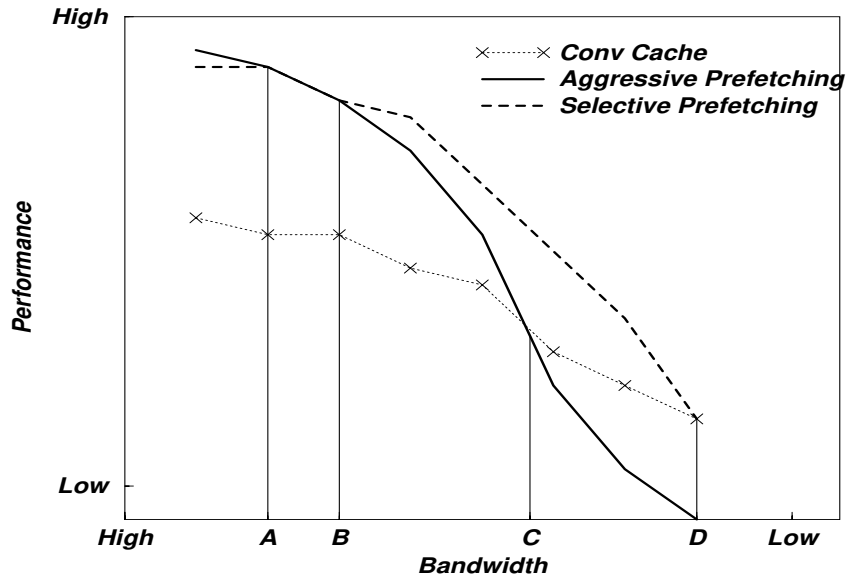
The other major contribution of this paper is a complete taxonomy for classifying prefetches. onsider the traffic and misses of a cache prefetch technique relative to a conventional cache that does not prefetch. This taxonomy analy h prefetches and assigns it ks or of additional traffic and a hange in misses that is or causes. More importantly, net c 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 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 organi wed as follo describes the *Static Filter*. Section describes the prefetch taxonomy. n section *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 S applied to these techniques. Section ts and discusses simulation results. e present related prior work in the area of data prefetching in section . Section concludes the paper and mentions for future work.

Trading i or and idth

The primary goal of prefetching is to hide memory access latency by eliminating demand misses. However, to maximi e the bene hing, it is important to minimi umber of useless prefetches that degrade the performance by polluting the cache and increasing bus contention. he 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. n this paper, prefetching directly into the cache.

eigure typical performance curves as a function of memory bus bandwidth for an



1

aggressive data prefetching technique, a selective prefetching technique and a conventional cache with no prefetching. hen effectively unlimited bandwidth is available, aggressive prefetching techniques can achieve the best performance, as seen in the igure t *A*. As the prior to point available bandwidth is decreased, the performance of an aggressive technique deteriorates faster than a more selective prefetching technique, as in the region from point to point in igure 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 to of igure ntually, for example past *D* in v igure 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

or 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. or example, in SP an access to block x is a prefetch trigger of block x . 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. or example, SP does not prefetc if the access to x is not a cache miss. *Static Filter* S) uses an additional enabling mechanism that must also be satis ed. The pro le is used to determine which loads tend to generate data references that are useful prefetch triggers and mark those loads as *enabled*. At run time S disables the hardw 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, pro ling can be used to select enabled loads. e now describe the two phases of S tation. le and im

2.1.1 Profile Phase

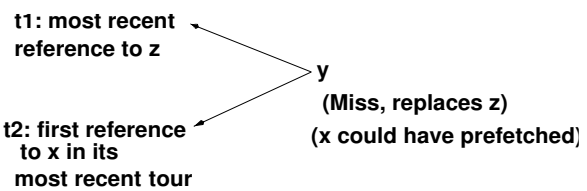
The pro le phase of S wo steps.proceeds in t n the e identify thrst step, w dresses that were most successful prefetch triggers for the hardware technique. n the second step, w enable the load instructions that generate references to those useful prefetch triggers.

e had complete knowledge of future data references, we could identify successful prefetch triggers as follows Suppose the baseline t, issues a prefetch for cache block x that is next referenced at time t . Suppose further that the current least recently used block y whose next reference is at time t w, there are two possibilities.

- $t < t$ ast
  The prefetched block will de nitely pollute by replacing a more useful cache block.
- $t > t$ ast
  Prefetching x is *potentially useful* because, by evicting y early, we can bring a more useful block into cache.

y, even in the pro le phase efficiently gain knowledge of future references to cache blocks. e have therefore implemented an approximate technique to identify successful prefetch triggers in the pro le phase based on the history of access patterns.

n the pro e simulate a conventional cache, without prefetching; whenever a block is brought into cache we maintain or create an table e used a table of entries for our experiments) that records the current time and the P of the load instruction that accessed the data. er 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. x is present[1], we know the time that x was most recently brought into cache y, t sa) and the load instruction that accessed sa x to begin that cache tour. would have prefetched y along with x at time t o determine if that prefetch is *potentially useful*, we check the time t of the most recent , the block chosen for replacement now to accommodate y as shown in igure $t > t$. the prefetc f is deemed *potentially useful*; if $t < t$ it is not.



or each load instruction we can now nd the ratio of the number of misses incurred to the number of *potentially useful* prefetches triggered by the load. tially useful prefetches are more than e mark this load instruction as enabled.

2.1.2 Implementation Phase

n the implementation phase, the baseline technique with S issues a prefetch triggering data access is caused by an enabled load. ur results by so restricting the prefetches we achieve a signi t reduction in the traffic requirements of while preserving the reduction in miss rate that achieves without S .

Static ilter requires the load instruction to be mark enabled hing. for data prefetc this paper, we use pro ling and collect the list of enabled load instructions and use a hardw

[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. We can also use the load opcode to indicate whether the load is a candidate for issuing prefetches. This requires a modification to the existing ver, modern processors including Pentium A , and AM ve special instructions for prefetching data into the cache. ith the curren trend of more viding 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. e are simply proposing that these prefetch instruction can also be used to improve the performance of the hardware prefetch mechanisms.

Prefetch Ta ono

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.

n this section, we introduce a systematic classification of all prefetches according to their usefulness. e analyze the effect of each prefetch on the memory traffic and the number of cache misses. sing the taxonomy we can classify all the prefetches issued into disjoint categories and and analyze their benefits. T e simultaneously model two identically con hes, 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.

e measure the traffic between the primary cache and the next level of the memory hierarchy y sa umber of cache blocks that are transferred from he. to the primary cac

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 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 straightforward to quantify the usefulness of the prefetch in terms of the extra traffic generated and the miss saved. t becomes complicated to analyze this case because the outcome of x and the outcome of x s access in the *conv-cache* are not considered in the above classification. y taxonomy described below is sufficiently refined to be able to precisely account extra traffic and the misses saved or incurred that should be attributed to each prefetch.

pf-cache there are only two possible outcomes for block x; it is either a hit or is replaced before its next reference. When a block is a hit in the *pf-cache*, it can have either a hit or a miss in the *conv-cache*. When the other hand, when a block 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 next block, but before the next reference to x) is simply classified by the independent prefetch of the above two cases. Since 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 the possibilities can be enumerated as follows; summary is presented in Table .

- 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 cases.

  pf-h-m *conv-h-c* as The next reference to y is a hit in the *conv-cache* and a miss in the *pf-cache*. Therefore, the *pf-cache* has additional block of traffic for prefetching x and block for fetching y) and has additional miss relative to the *conv-cache*.

  pf-h-pfed *conv-h-c* as 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 additional block of traffic for prefetching x) and has no additional misses relative to a *conv-cache*.

  pf-h-dc *conv-h-c* as 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 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 cases.

  pf-h-m *conv-m-c* as The next reference to y is a hit in the *conv-cache* and a miss in the *pf-cache*. Therefore, the *pf-cache* has additional block of traffic and has no additional misses relative to the *conv-cache*.

  pf-h-pfed *conv-m-c* as 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*.

  pf-h-dc *conv-m-c* as 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.

  pf-repl-m *conv-dc-h-c* as The next reference to y is a hit in the *conv-cache* and a miss in the *pf-cache*. Therefore, the *pf-cache* has additional block of traffic and additional miss relative to the *conv-cache*.

  pf-repl-pfed *conv-dc-h-c* as 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 additional block of traffic and has no additional misses relative to the *conv-cache*.

pf-repl-dc *onas* *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 additional traffic and has no additional misses relative to the the *conv-cache*.

e 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. t is important that in each of the *pf-cache* cases, 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 groups from the un pairs.

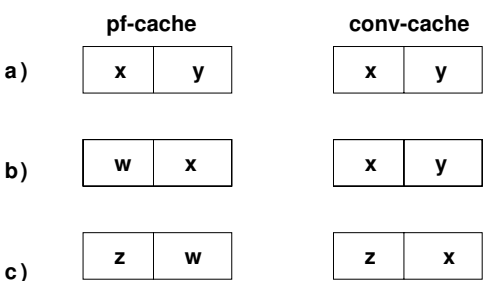
- Polluting Prefetches These prefetches generate additional traffic and additional miss relative to the conventional cache clearly cause cache pollution and degrade performance. ases ve fall into this category. abo
- seless Prefetc The prefetches which generate additional traffic and no additional misses relative to the conventional cache are issued too early and cause wasted bandwidth. These are not as harmful as the *polluting prefetches*. ases , above fall into this category.
- seful Prefetc ase and ns. They cause no additional traffic and save a miss relative to the conventional cache. ny cases are useful. However, each occurrence of case is clearly linked with some other case involving the later prefetch of *y* into the *pf-cache*. n addition, since the next reference to *y* hits in the *conv-cache*, the case for this later *y* prefetch must be , h linked chains of cases, . Suc beginning with case y not being useful. However, case ays useful.

rom the e observe that the extra traffic is always one more than the extra miss. ote that *pf-cache* incurs one less miss relative to the *conv-cache* we represent it as extra misses. f the ted above cases prevent and disjointly accounted for all the extra traffic and all the extra misses, the following equations are ed.

Misses Misses polluting prefetches useful prefetches
Traffic raffic T polluting prefetches useless prefetc

e do know from the way that the cases and their costs are constructed, that they are disjoint. However, our simulation studies have shown that this case classi cation, it completely covers all prefetches, does not completely attribute all extra misses and traffic to them. Speci , we have not accounted for extra misses and extra traffic in the *pf-cache*, triguingly, the simulations always indicated that . A *t* nal hed) case non accounts for and and completes the taxonomy.

onsider the example with the *pf-cache* and the *conv-cache* having the same contents ks *x* and *y*) as shown in igure k *y* is the least recently used. Block a). Block.



w if the prefetching technique initiates a prefetch for block *w*, we obtain the cache state shown in igure *(w, y)* falls into one of the The pair case i ve. pairs ide ote that the contents of *conv-cache* does not change. hile in this state, suppose the program references block ; we have a miss in both the *pf-cache* and the *conv-cache* and the resulting cache contents are shown in igure , suppose that there is now a reference to it; it is a miss in the *pf-cache* and a hit in the *conv-cache*. This miss leads to k extra traffic and 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 . This is therefore a consequence of poor speculative prefetching. e refer to the occurrences of this case as *case efetch side-effects pr* in our results. ase y noting when, is detected b

- t blocks are replaced on a demand miss in both caches. or example, the demand miss to in igure and
- the block replaced *x* in igure *pf-cache* is subsequently hit before being replaced) in the *conv-cache*.

ery simulation that we have run, the number of case occurrences exactly equal extra as shown in equations Thus we conjecture that the y is case tax complete.

Misses Misses polluting prefetches useful prefetches m Traffic raffic T polluting prefetches useless prefetc

at	as r	s ra	ctra	tra
	pf-h-m *onv-h-h* *c* *pf-repl-m* *onv-dc-h* *c*			r
s	*pf-h-pfed* *onv-h-h* *c* *pf-h-dc* *onv-h-repl* *c* *pf-h-m* *onv-m-h* *c* *pf-repl-pfed* *onv-dc-h* *c* *pf-repl-dc* *onv-dc-repl* *c*	r		tc
s	*pf-h-pfed* *onv-m-h* *c* *pf-h-dc* *onv-m-repl* *c*		r	
r	tc	as		

1

n our results, we use the above taxonomy to quantify how well S reduces the useless and polluting prefetches and how well it preserves the useful prefetches. iscriminating bet 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.

tal Fra eri e en

e now brie y describe the techniques, *Next Sequential Prefetching* *Shadow Directory Prefetching* Se will use to illustrate the effectiveness of S . This is followed by a discussion of the simulation environment and the benchmarks used in this study.

.1 e tial t Se hire etc e

text prefetching SP), proposed in simple prefetc ique in which a prefetch for block) is issued t in the cache) whenever block is

accessed. To control the number of useless prefetches, a variant called tagged prefetching is also proposed. ariant, this 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. n essence, this exploits the spatial locality exhibited by the applications and uses the data addresses to predict the physically contiguous successor as the prefetch address. n this paper, we tagged variant of . SP

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

Sha.2 irect r re etc

Shadow irectory Prefetc S P), proposed in a prefetching technique in which a history of the referencing pattern is maintained in a hardware table. A shadow address is maintained in the he directory along with the currently resident address. The shadow address refers to the block accessed after the currently resident block was last accessed. er henev we have a he hit, we issue a prefetch for the corresponding shadow address in the directory. To reduce the number of useless prefetches, a hemb it used. A con rmation rmation bit is added to each directory entry in he indicating whether the prefetched block was used) or as last prefetched.) when it w

S P relies on the repetition of data access patterns and on a stable relationship bet the resident address and the shadow address. S P exploits the correlation between the resident address and the shadow address, to predict prefetch addresses. The advantage of S P 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 repetitiv S P can lead to an excessive of useless prefetches.

Si . lati it e

or our simulations, we used the SimpleScalar simulator . The functional tool set was used for the pro le phase and the implement. e used the detailed out of order ulation to evaluate our technique. e modeled a ay superscalar processor with a traditional stage pipeline and a bi h predictor with branc try table. e en simulated teger A in t adders, teger and in oating ultiply in divide units. e assumed a perfect he. e simulated a cac he. e B used different block si es ytes) and different associativities and found , that the trend of the results was the same for all degrees of associativity, and so we present results only for a ay associative cache with yte blocks. The he was accessed in cac cycle and the memory was accessed in ytes, and for the cycles for each after.b The processor used read write ports and a store queue. en n the implement of S e simulated hes cac ventional cache with no prefetching, a cache with a given baseline hardware technique SP or S P the baseline technique enabled by S S SP or S e present results for both integer and oating mark, namely, six benchmarks from SP t e benchmarks from SP v fp t we have not been able to collect results for the other benchmarks of the SP suite). All these benc

compiled with gcc e used the *train* input set for our pro le phase and the *ref* input set for our implementation phase and limited the execution to billion instructions.

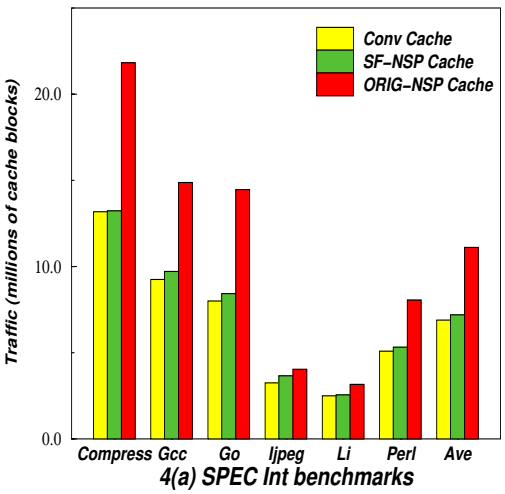
Re ult

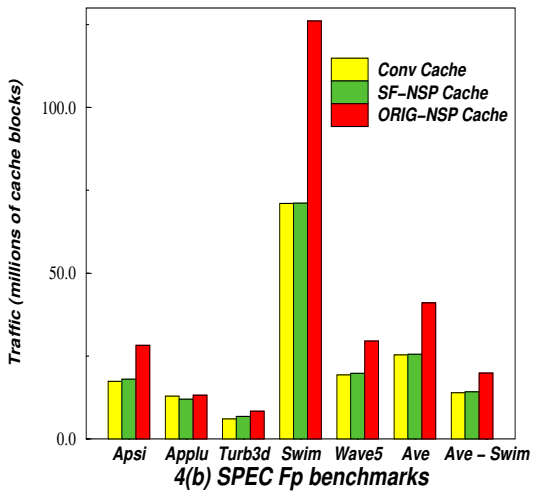
e use the following metrics to evaluate the usefulness of S .

- **ra ta** The total number of cache blocks transferred from he. to the prima This is measured as the sum of the misses and the total prefetches issued.
- **ra c A** The usefulness and accuracy of prefetches are illustrated by the distribution of prefetches based on case pairs identified in T in section
- The ratio of miss rate to total number of data references. Miss atio in conjunction with total traffic illustrates the tradeoff between misses and bandwidth for the baseline techniques with and without S .
- **r** The speedup achieved by the prefetching technique relative to the conventional cache.

.1 al tial hire etc ith SP

otal .1 ra





1

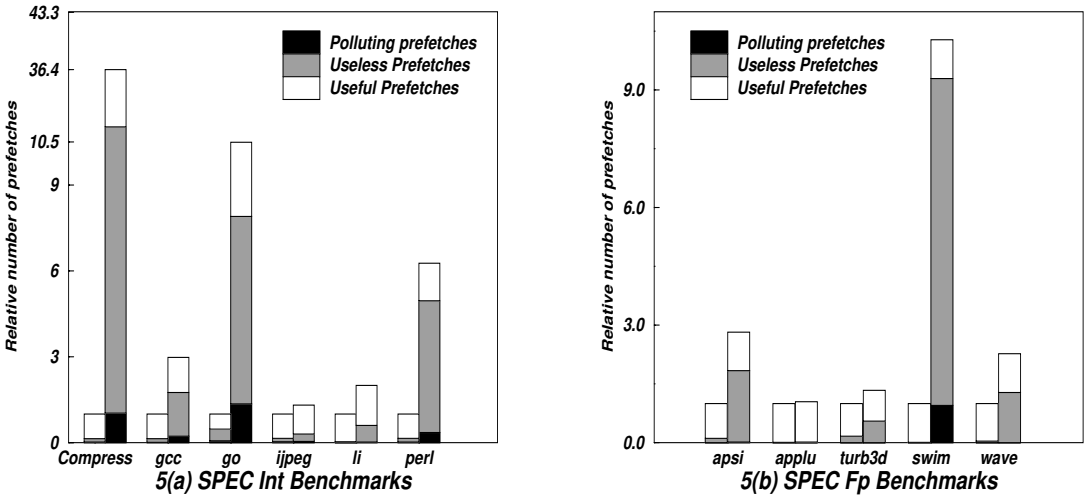
igure t the total a traffic for SP igure and SP hmarks. b) prefetch h group of bars, the leftmost bar is the traffic in the conventional cache, the second bar is for a cache using S hnique and the rightmost bar is for a cache using original SP technique.

esidues S ve an average of teger benchmarks in traffic for oat ing point benchmarks are known to have sequential access patterns and SP is effective prefetching technique for these benchmarks. However, S hieves an average of tion in traffic for the oating point benchmarks excluding swim). The last group of bars sho the average traffic excluding *swim*. or *swim* SP has as much traffic as S .

This leads to more cache pollution as we will see when we compare the corresponding miss rates for SP and S SP in igure

Pre.1.2 h ra et nal sis

Although total traffic is a measure of the number of blocks of data transferred from primary cache, it does not provide insights about the prefetch traffic. ig u e the prefetch sho traffic based on our taxonomy. The results are normali ed with respect to S each benchmark, the est bar is the distribution for S , the second bar for . The height of SP the second bar is proportional to the ratio of prefetches of eS SP S elativ . The difference in heights is the ratio of the extra prefetch traffic generated by SP without S .



1
left bar S t bar SP) n

or both the integer and ting pmarks, we observe that S to *polluting prefetches*. This suggests that using a e S lter may eliminate the need for separate prefetch buffers and instead bring prefetches directly into the cache with more con dence.

n the case of marks their irregular access patterns cause *polluting prefetches* in SP; which is eliminated using S or example, *go* with SP has *polluting prefetches*; which is reduced to . using S SP

SP has a large fraction of *useless prefetches*. e see that *compress* with SP has as many prefetches as with S hes are useless. n the case of hes are *useless prefetches* using SP and only . erall, S v y *useful prefetches*, indicating the improvement due to the selectivity of S wever, for some benchmarks like *go*, *li* and *apsi*, S y be too selective and decrease coverage, as we will see in igure e observe that for *swim* SP has y prefetches as S . However, S SP s reduction in traffic will not be effectiv preserv the decrease in miss ratio achieved by . SP

Table ts the number of occurrences of the side effect case iden ed in section that each occurrence of the side effect case incurs k of traffic additional bloc additional miss relative to the conventional cache. e see that issuing fewer prefetches using S e have in w general *turb* except for *wave*) decreased the total number of side effect cases.

	r ar c	
compress		
gcc		
go		
ijpeg		
li		,
perl		

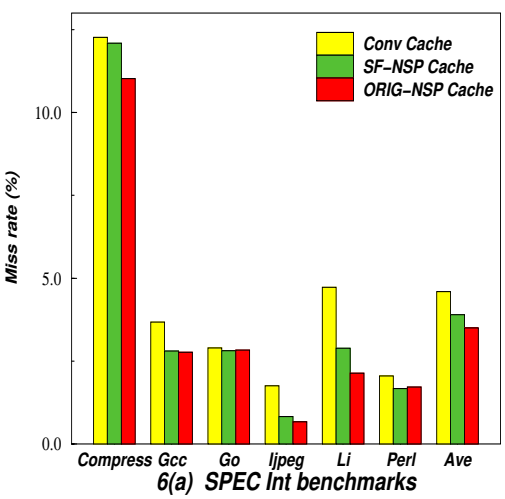
s

,
,
,
,
,

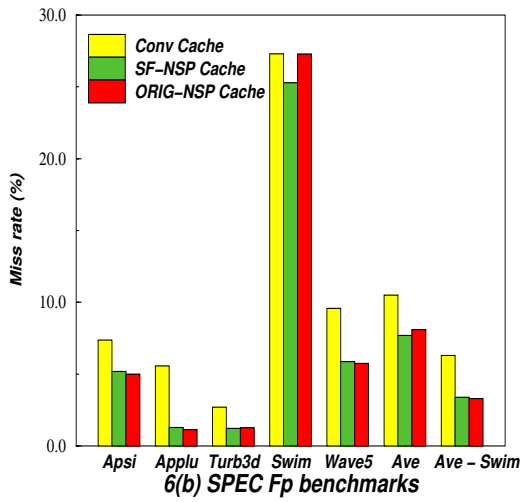
te

	r ar c		s
’ apsi			,
applu ’		,	,
turb ’	d	,	
’ swim			
’ wave			, ,

,



6(a) SPEC Int benchmarks



6(b) SPEC Fp benchmarks

1

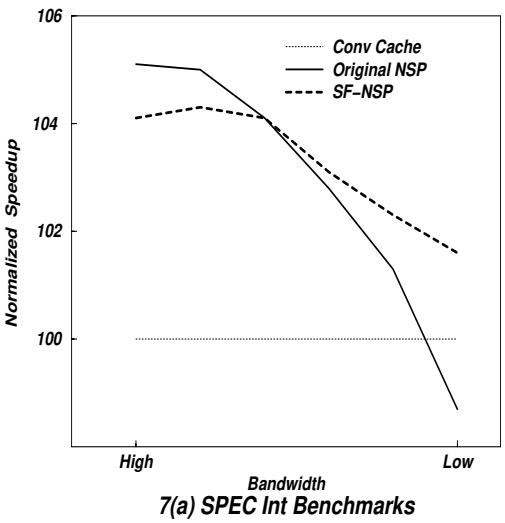
.1. iss ate

igure w the miss ratio for SP t igure SP hmarks respec)fs hoenc tively. h group of bars, the leftmost bar is the miss rate in the conventional cache, the second bar is for a cache using S hnique and the rightmost bar is for a cache using original hnique. SP tec

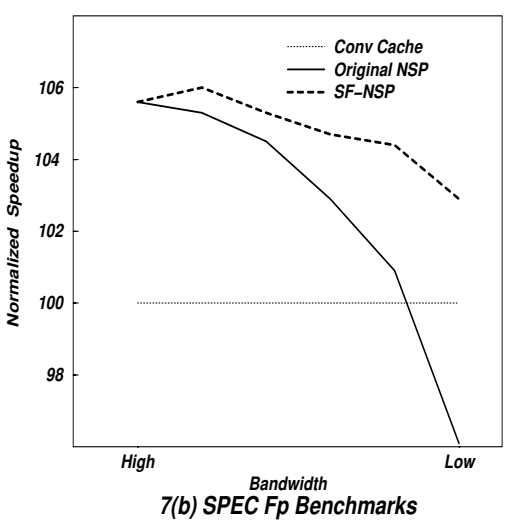
e observe that relative to a conventional cache, hieves an average of reduction in misses for integer and tabing mark respectively. n the other hand, S y SP reduces the miss rate and and tabing mark respectively; which implies that we are not eliminating very many useful prefetches by using S . However, for *li* and *ijpeg*, SP performs better than S tly SP b sequential data access patterns for which S is too selectiv SP *potentially useful* prefetches. n the other hand for *go* where selectivity is more important, the SP miss ratio is ved. S en has a lower miss rate than SP ev *gcc* because the additional *useful prefetches* in ere more than offset by the *polluting prefetches*. A clear trade ween misses and traffic is seen when we compare S S for *compress*; using SP a hieved with a reduction is miss rate is ac relative to S . SP

The tabing mark have considerably higher miss rates than the integer bench marks. or a conventional cache with no prefetching, the average miss ratio is . This that a he may be too small to capture the working set of these benchmarks. This makes prefetching more critical for improving performance and both SP and S ve comparable performance. Since the miss rate of *swim* wice that of the other benchmarks more than t we have shown the last group of bars excluding *swim* from the average. *swim* fact, for SP has more misses than S he pollution due to the excessive prefetching in . SP This is con y our observation in igure h traffic ratio S for *swim* is very high about times).

.1. ormali e pee p



7(a) SPEC Int Benchmarks



7(b) SPEC Fp Benchmarks

esults from the previous sections sho tly reduces wasted bandwidth of SP while preserving the reduction in miss rate. , we really the improvement in

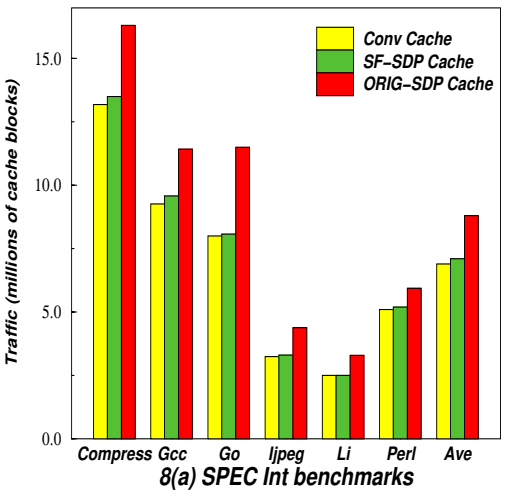
execution time using SP and S igure the speedup normali SP ed with respect to the conventional cache. e present the average speedup for integer and t benchmarks. n this, we varied the available bus bandwidth over a wide range e started with a design where the bus is fully pipelined and can take a new request from the processor each cycle. e refer to this as *high* bandwidth in the e axis of ed this design restricting access to the bus to account for bus arbitration overhead, multi cycle data transfer and more limited pipelining. n the most restrictiv guarantee, that the bus is non pipelined with the bus blocked for the duration of the data transfer. e refer to this as *low* bandwidth in the axis of igure .

igure the average normali a) sho ed speedup for benchmarks. n the case of a very high bandwidth system we observe that SP has a e to the conventional speedup relativ cache; S e restrict the bus access, we see that S speedup. As w speedup and SP has speedup. As the available bandwidth limited b we see that the performance of SP degrades faster than S SP n the case of t benchmarks, the original SP performs and by applying S e have eliminated almost all the pollution prefetches and hence the overall speedup increases from to with S .

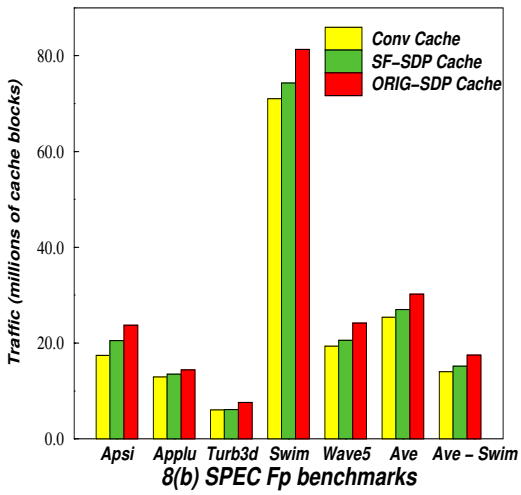
.2 al irect r Shithe

To demonstrate that S can be applied to a more sophisticated prefetching technique we now present results for our second baseline technique S . or the results presented in this section, the baseline S P uses bit con y dynamic con and S rmation and relies on S to eliminate useless prefetc

otal 2.1 ra



8(a) SPEC Int benchmarks



8(b) SPEC Fp benchmarks

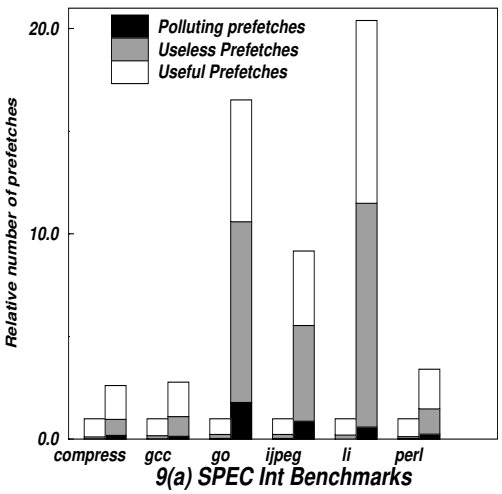
1

igure t the total traffic. As in the case of , the leftmost bar b) presen is the traffic in the conventional cache, the second bar is for a cache using S hnique and S the rightmost bar is for the cache using original S hnique. tsc P is a more sophisticated prefetching technique that achieves similar performance as SP with few. S P has

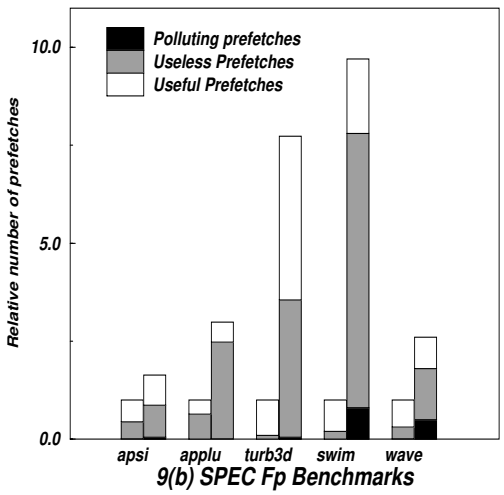
only teger benchmarks and SP traffic for int benchmarks. Hence, oa we do not achieve the same the improvement using S on S . P assumes SP we observe an average of teger benchmarks and traffic for int benchmarks. for As in the case of , *swim* has SP a very high traffic and hence the last group of bars of igure show the average traffic excluding *swim*.

Pre.2.2 h ra et nal sis

As with , we use SP our taxonomy to analy e the useful prefetches. In ure results based on our classi cations of prefetc results are normali ed with respect to S S . or each P benchmark, the est bar distribution for S , the second bar for S P S . The height of the second bar is proportional to the ratio of prefetches of S e to relativ S . Again, the difference in heights P show the amount of extra prefetch traffic generated by S P without S .



9(a) SPEC Int Benchmarks



9(b) SPEC Fp Benchmarks

1

left bar S t bar S S P P) n

As with , we observe that S reduces *polluting prefetches* of S P to However, in the case of *go* and *li* S S *useful prefetches* P also decreases the decrease the coverage as we will see in igure or taking point benchmarks, we observe a). that S P has a very small fraction of *polluting prefetches*. *swim* and *wave*, the other benchmarks do not have any *polluting prefetches*.

S P has a considerable fraction of *useless prefetches*. n the case of *applu* using S P we see *useless prefetches* respectively; using S S P it is reduc respectively. SP performs we benchmarks exploiting their sequential access patterns.

S y *useful prefetches*, indicating the improvement due to the selectivity of S wever, as opposed to S , we see that S SP S umber of *useful prefetches* too and this will increase the miss rate as we will see in igure

Table ts the number of occurrences of the side effect case iden ed in section see that issuing fewer prefetches using S e have consistently decreased the total number of side effect cases relativ P

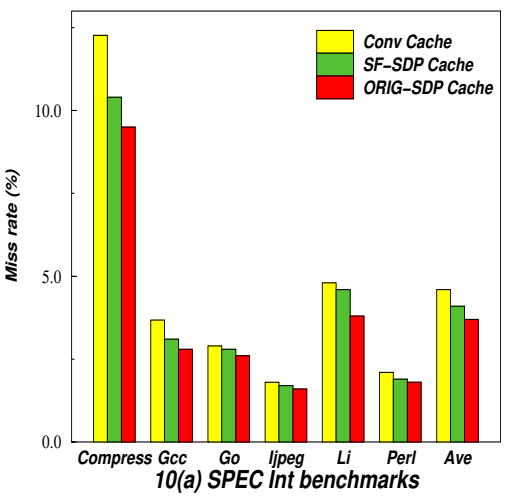
	r ar c	
compress		
gcc		
go		
ijpeg		
li		,
perl		

s

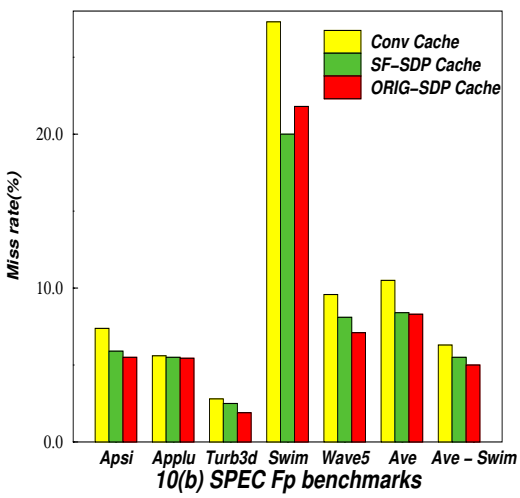
,
,
,
,
,
,

te

	r ar c		s
apsi ’			,
applu ’			,
turb ’	d	,	,
swim ’			
’ wave		,	,
,			



10(a) SPEC Int benchmarks



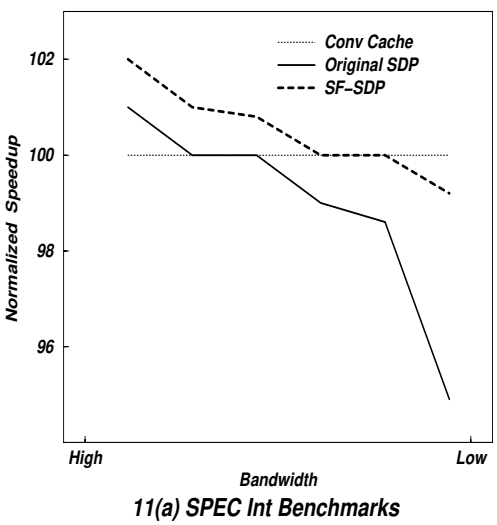
10(b) SPEC Fp benchmarks

1 1

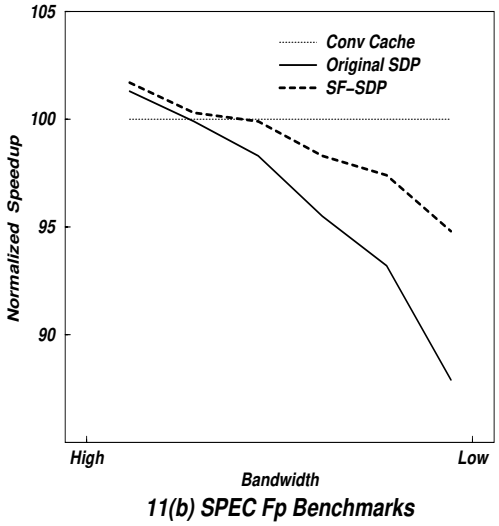
.2. iss ate

igure w 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 S hnique S and the rightmost bar is for a cache using original S hnique. tece observe that both S P and S w signi S t improvement in miss rate for these benchmarks. e see that S S P has e to S . However, we see that S P issues hes relative to S hieve this reduction in miss rate. S more prefetches or integer benchmarks S P performs as w weve, for SP, for tabing poin it does not perform as well as . or SP tabing poin, the shadow address is not predictable due to the streaming behavior of the applications and hence the prefetches are pre dominantly 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*. t is important S performs comparable to the purely dynamic technique of using bit con rmation.

.2. ormali e pee p



11(a) SPEC Int Benchmarks



11(b) SPEC Fp Benchmarks

11

igure the speedup normali sho ed with respect to the conventional cache. The exper imental framework is similar to the one used for . Here again, we observe that S SP better performance relative to S P when a w nite bandwidth. We see that the speedup using S P is less than that using SP, the speedup using S more than that using S . S P

ur results using SP and S P at S baseline tec is an effectiv lter in reducing useless and polluting prefetches while retaining the useful prefetches of the baseline techniques.

Related or

Several methods exist for prefetching data into cache. rouping of consecutiv words into a cache block is itself one form of prefetching. hes 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 he blocks ac 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. urthermore, 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 si es.

The critical issues of prefetching are *when* to issue the prefetch and *what* data to prefetch. n 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. n the other hand, a late prefetch hides less of the access latency, or none at all. *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. n addition it is important to evaluate block prefetch strategies over a range of block si es 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 , , , , ti c applications with regular access patterns. or applications using linked data structures and recursive data structures, techniques have been proposed , , ,) to tolerate 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. t has the disadvantage of software overhead, both in the number of instructions issued and in code si e. This is one of the reasons why we do not apply S to a software based baseline technique.

Hardware based prefetc , , ,) 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) for prefetc linked data structures using dependence analysis and have a prefetch engine run ahead of the execute engine to detect possible prefetch candidates. Markov Prefetching) predicts cac misses based on previous miss patterns. olution is a shadow directory prefetc miss address has more than one successor miss addresses. aluation of Markov predictors on SP t benchmarks showed a signi t 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. are 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. e expect a technique like S can be used to selectively issue prefetches based on the bus contention or available bandwidth. S can be extended and applied to the above hardware techniques to improve their prefetch accuracy and reduce their bandwidth requirements.

Hybrid prefetching bine the low run time overhead of hardware prefetching with the effectiveness and accuracy of software prefetching, but the only reported evaluations are on scienti c applications.

onclu uture or and F

tributions in this paper are two fold

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

The taxonomy analy h prefetches and assigns it ,ks of additional traffic and bloc a hange 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. S analy es the data access patterns of an application and uses the prefetch *selectively* and reduce wasteful traffic. Prior work in issuing selective prefetches rely on the dynamic con denses based on the history between the data addresses of the application. n this paper we 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 S e chose traditional *Next Sequential Prefetching* and *Shadow Directory Prefetching* as the baseline techniques.

ur results sho hieves sign t reduction in prefetch traffic of the baseline techniques , to to verly increasing the miss rate for S SP addition, timing analysis reveals that when nite memory bandwidth is a limiting factor, applying S does in fact increase the speedup obtained by a baseline hardware prefetching technique.

n this paper we providing information derived from pro ling to a hardw prefetching technique. ur results indicate that hardw prefetching techniques bene t from S wever, the threshold used to determine whether a load is marked enabled prefetching was xed statically n addition, the analysis separately for each cache con guration. ST dynamically to ve con gurations and to use a dynamic threshold, run time information about the usefulness of a cache could be used.

Reference

hes for Superscalar Processors *Proceedings of the* Prediction *nnual Int A osium Symp oarchitecture, December* . *7, pp*

ool Set, Burger, T. Austin, *Technical eport* The SimpleScalar, T *niversity of Wisc un* *7*

, T. Pu ali, M. Prefetch behavior of the SP hmark suite *I* , *esearch and Development, of 4 er , Numb* , *ay*

T. hen, king and Baer, Prefetching hes ac edu *Fifth International ence on Architectural Support for Programming anguages and Operating Systems, October* . *, pp 5 -6*

erformance. T. Study of Software and Hardware Baer, ating Prefetc Schemes *Proceedings of the nnual Int osium Symp rchitecture, April* . *4, pp* -

wald, hing Prefetc . oving Mackosph, *Proceedings of the* run *4th Int osium Symp rchitecture, ay* . *7, pp*

ving irect he Performance by the Addition of a Small ully mpro Associative he and Prefetch Buffers *Proceedings of the 7th Int osium on l Symp omputer Architecture, ay* . *0, pp 64-*

, hitecture for Software laiber, Hed ewing Prefetc *Proceedings of the osium on th Int rchitecture, Symp ay* .

ander iel, S. Aren ilja, t hing Techniques I , ac ata
omputer aga . ine, uly

hmidt, S. el, unk M. wapeiger, Prefetching in pointer Spar and Soft
call intensive environments *Proceedings of the nnual Int ostium Symp oar- icr*
chitecture, November . *5, pp* -

wry, hing for based Prefetc structures *Proceed-* , uk, T.
ings of the 7th Int ence on Architectural Support for Programming anguages and
Operating Systems, October . *6, pp*

e, P. hang, . H. wu, hitectures for Superscalar Access Micro
Processors with ompiler hing *Proceedings of the* Prefetc *4th Int*
sium on oarchitecture, November . *, pp 6*

S. Mehrotra, . Harrison, me for
Pointer tensive and n umeric Program *Proceedings of the* , *0th Int ence on Su- l*
percomputing, ay . *6, pp*

wry, M. am, A. alpalo, Tuation of a ompiler Algorithm for Prefetc
ing *Proceedings of the 5th Int ence on Architectural Support for Programming an-*
guages and Operating Systems, October . *, pp 6*

omerene, T. Pu lts, chaffen, . . Spa cio, ing System for Prefetc he ac
Having a Second irectory for Sequen ally Accessed Blocks , *.S. Patent 4,eb 07,*
.

, A. Porter ward, Prefetching Soft ceilings, *Proceedings of the 4th* .
Symposium on Architectural Support for Programming anguages and Operating Systems,
April . *, pp 40-5*

vos, . Sohi, A. hing for Aperiodic Based Prefetc Structures
Proceedings of the 7th Int ence on Architectural Support for Programming anguages
and Operating Systems, October . *7, pp 5-*

e ointer Prefetching for P otl, ink . Sohi, Pro-
ceedings of the 6th Int sium on l Symposium A, ay .

hing for Memory . Selvidge, atency T *Ph.D Thesis,* ompilation
IT,. ay

he Memories *A* , A. Smith, *omputing Surveys, Sept.*