A Hardware-based Cache Pollution Filtering Mechanism for Aggressive Prefetches

Xiaotong Zhuang
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
xt2000@cc.gatech.edu

Hsien-Hsin S. Lee
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332
leehs@ece.gatech.edu

Abstract

Aggressive hardware-based and software-based prefetch algorithms for hiding memory access latencies were proposed to bridge the gap of the expanding speed disparity between processors and memory subsystems. As smaller L1 caches prevail in deep submicron processor designs in order to maintain short cache access cycles, cache pollution caused by ineffective prefetches is becoming a major challenge. When too aggressive prefetching are applied, ineffective prefetches not only can offset the benefits of benign prefetches due to pollution but also throttle bus bandwidth, leading to overall performance degradation.

In this paper, a hardware based cache pollution filtering mechanism is proposed to differentiate good and bad prefetches dynamically using a history table. Two schemes — Per-Address (PA) based and Program Counter (PC) based — for triggering prefetches are proposed and evaluated. Our cache pollution filters work in tandem with both hardware and software prefetchers. As shown in the analysis of our simulated results, the cache pollution filters can significantly reduce the number of ineffective prefetches by over 90%, alleviating the excessive memory bandwidth induced by them. The IPC is improved by up to 9% as a result of reduced cache pollution and less competition for the limited number of cache ports.

1. INTRODUCTION

The speed disparity between CPU and main memory continues to increase that poses a major obstacle for performance scalability of modern processors. Although data caches can somehow bridge this gap, yet initial data references that miss caches still suffer from long memory lead-off latencies if there is no enough number of independent instructions to mask the delay, the problem aggravates for static machines, e.g. Intel/HP's Itanium. Prefetching has become an essential technique for hiding memory latency. Instead of waiting for actual memory instructions' requests for data accesses, prefetching brings data into the memory hierarchy closer to the processor before they are demanded.

1.1 Data Prefetching

Most prefetch techniques are prediction-based, the accuracy and potential performance gain highly rely on the predictability of memory reference behaviors. Simple hardware-based prefetching techniques proposed in [10, 15, 16] attempt to identify and capture regular data access patterns with unit strides. More sophisticated hardware-based schemes [7, 8] can issue prefetches for sequential data accesses with arbitrary but constant strides. In [3, 4], Chen and Baer proposed a reference prediction table to monitor data reference patterns and issue prefetches dynamically. Correlation-based prefetching [2] keeps prior L1 cache miss addresses and triggers prefetches by correlating subsequent misses to the history.

On the other hand, static analysis techniques were applied at compilation time to perform software prefetching [14, 16]. They embed prefetch instructions within the binaries for runtime prefetching. Many contemporary microprocessor instruction sets feature some flavors of fetch instructions that simply move data into the cache without intervening other architectural resources. For example in Alpha ISA, the load instruction can perform data prefetch if the destination register is $r31 which is hardwired to zero [6]. Since these prefetch instructions are non-blocking, CPU can continue execution without awaiting their completion.

1.2 Aggressive Prefetching

As the IC feature size continues to miniaturize, computer architects are dedicating more transistors to cache memory on the processor cores, however, with more hierarchies. At the same time, as additional memory bandwidth among caches and main memory become available, more aggressive prefetching schemes were proposed to utilize it. Current design trend shows that even though the overall cache size is getting larger, the first level (L1) cache is, in fact, getting smaller in order to guarantee an expeditious L1 access, typically in one or two core cycles. For example, Intel's Pentium 4 processor employs an 8KB first level data cache. Recent research results [20] also suggest to shorten cache memory latency by engaging a 1KB micro-cache for future Itanium architectures. It is also less expensive to build a smaller multi-ported cache for wider machines which need to process more memory requests at the same cycle. Based on this design trend, overly aggressive use of prefetches will not only postpone normal L1 cache accesses but also lead to severe L1 cache pollution.

1.3 Cache Pollution

No data prefetching algorithms can guarantee 100% accuracy and effectiveness. A prefetched cache line could be either completely useless or ineffective when it is displaced before consumed. These prefetched data are allocated in the data cache and compete for the available cache resources, seriously degrading the performance when the L1 cache size is small. Evicting useful data in the cache by ineffective prefetches causes cache pollution which unnecessarily reduces the overall performance due to overly aggressive prefetching schemes. Performance is also significantly degraded for some benchmarks when prefetching cannot be done precisely. For example, stride-based prefetching schemes can easily become ineffective for the pointer-based type applications, thereby polluting the data cache. Luk and Mowry in [12] proposed 3 prefetching schemes for pointer-based applications. Their work shows prefetch miss can be very high (80%) for some benchmark programs, which includes those prefetches that are not fi-

Proceedings of the 2003 International Conference on Parallel Processing (ICPP'03)
0190-3918/03 $ 17.00 © 2003 IEEE

IEEE COMPUTER SOCIETY

nally accessed by the application or evicted without access because they are issued either too early or too late. Srinivasan et al. in [17] shows even a prefetcher with a high coverage and accuracy may still lead to low performance (high total miss rate and low IPC). Therefore, the side-effects of the prefetcher are also critical for a prefetching technique.

In summary, an inappropriate prefetch can cause undesirable outcomes by (1) occupying cache space with useless data if the prefetcher is inaccurate and causing more capacity or conflict misses. (2) imposing higher pressure on the competition for finite bandwidth and limited number of ports of the cache, especially for aggressive prefetchers on a wide issue machine.

In this paper, we first investigate the impact of aggressive prefetching on conventional cache architectures targeting for deep submicron processes. Three different prefetching schemes are evaluated including software prefetches inserted by the Alpha compiler and two aggressive hardware-based prefetch algorithms. We then examine all the prefetches together with the runtime footprint of given programs to identify the effective prefetches, i.e. prefetched data that are referenced by issued memory instructions prior to eviction. These prefetches are classified as *good* prefetches. In contrast, those never referenced prefetches are classified as *bad*. Then, we evaluate the impact of bad prefetches toward the overall performance by artificially eliminating those bad ones. This motivates our endeavor to design a hardware-based cache pollution filter that can effectively prevent the bad prefetches from entering the cache by exploiting historical information. We propose two filtering algorithms, which either make prediction based on the cache line address of the prefetched data (Per-Address based) or on the program counter value of the prefetch instruction (Program-Counter based). Performance improvement, bus traffic reduction, and design options are quantified in our simulations and analysis.

The rest of this paper is organized as follows. Section 2 describes related approaches. Section 3 gives motivation. Our filtering hardware designs are described in Section 4. We evaluate the performance of our filtering scheme in Section 5. Section 6 concludes this work.

2. RELATED WORK

Several previous works have addressed the problem of reducing the cache pollution caused by prefetching. These techniques can be classified into three categories — software-based by compiler [19], hardware-based [4, 11], and hybrid [17]. Chen et al. in [5] proposed a dedicated prefetch buffer for data prefetching. Instead of bringing prefetched data into caches, the software data prefetch instructions allocate prefetched data into a dedicated prefetch buffer. The data cache and the prefetch buffer are probed either in parallel or in sequence for each data item accessed. If both are missed, the data will be fetched to the cache from next level memory hierarchies. Typically, a prefetch buffer is fully associative. When accessed in parallel with the L1, the prefetch buffer can become the critical path if it cannot keep up with the speed of the L1, thus limiting the prefetch buffer size.

In [19], Wang et al. introduced a compiler's approach that checks the data in the cache to see if the next reuse distance is twice the cache size. It is shown that this scheme can reduce the pollution of prefetched data if the data are unused or the prefetch distance is too long to keep the data in the cache. Data being marked as *evict-me* have the highest priority to be displaced from the cache. Lai et al. [11] proposed to detect dead cache lines in caches and replace the dead lines with prefetched data. Their mechanism aims to reduce the situations where useful data are evicted from the cache too early. While having the similar goal to reducing cache pollution our approach focuses on eliminating ineffective prefetches from entering the cache. Srinivasan et al. presented a comprehensive taxonomy in [17] that classifies prefetches based on traffic and misses generated by each prefetch. They also proposed a static filter in [18] aiming at reducing the number of polluting prefetches. The static filter collects information of the polluting prefetches off-line through profiling and uses this profiling information to guide prefetches. They reported a 2 to 4% performance improvement of their static filter scheme combined with Next Sequence Prefetching and Shadow Directory Prefetching. In theory, the profiling information can provide precise global information for a given input data set, however, it lacks the dynamic adaptivity during runtime when the working set changes. In contrast to their work, our technique solely relies on hardware to evaluate each prefetch dynamically. No profiling information collection is needed. Our results show that our dynamic mechanism can deliver better performance than their static filter.

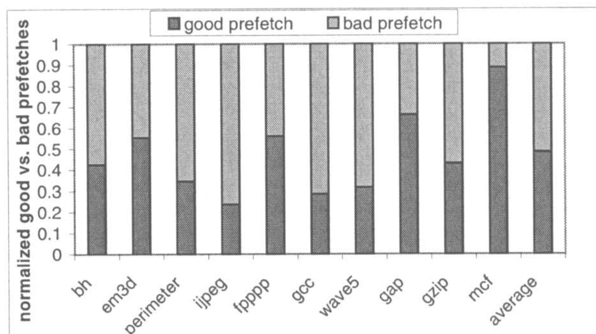

Figure 1: Effectiveness of prefetches.

3. MOTIVATION

In this paper, prefetches are simply classified into two categories: 1) good or effective — those referenced in the cache before they are evicted; 2) bad or ineffective — those never referenced during their lifetime in the cache. As a comprehensive prefetch taxonomy [17] requires many additional bits to keep track of the replaced cache line and reference order for both replaced and prefetched cache line, our simple yet competent classification simplifies the hardware implementation. Figure 1 shows the distribution of the prefetches based on our classification for 10 benchmark programs selected from the SPEC95, SPEC2000 and Olden benchmark suites. The prefetches include both hardware-based (next sequence prefetching — NSP [16] and shadow directory prefetching — SDP [13]) as well as software-based prefetches. Note that the number of software prefetches are far less than hardware prefetch but more accurate. The hardware-based prefetchers can reduce ineffective prefetches dynamically. For example, the NSP employs a tag bit associated with each cache line. When a cache line is prefetched, its corresponding tag bit is set. The next adjacent cache line is automatically prefetched when a memory access either misses the L1 or hits a tagged cache line. Similarly, the SDP maintains a shadow line address in each L2 cache line for prefetching purposes along with its resident address. The shadow line is the next line missed after the currently resident line was last accessed. A confirmation bit is added to each L2 cache line indicating if the prefetched line was ever used since it was prefetched last time.

In Figure 1, the number of "Good Prefetches" and the number of "Bad Prefetches" are normalized to the total number of prefetches for each benchmark program. As

IEEE COMPUTER SOCIETY

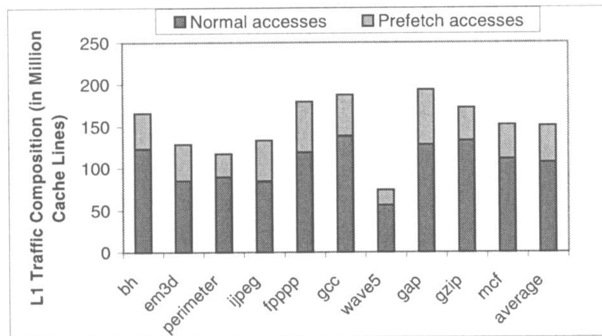

Figure 2: Traffic distribution of L1 cache.

indicated, more than half of the prefetches are ineffective or "bad" in 4 out of the 10 benchmarks. Our statistics show, on average, 48% prefetches are not referenced during their lifetime in cache.

Figure 2 shows the traffic distribution for the L1 cache in terms of cache lines for the data cache. Obviously, the traffics induced by prefetches take a significant portion of the total traffics to the L1 cache. On average, the prefetch access to normal access ratio is 0.41 with a maximum of 0.57 (ijpeg) and minimal of 0.29 (gzip). In other words, on average, about 2/7 traffics to the L1 cache are prefetches. Combined with Figure 1, it implies the aggressive and/or excessive prefetches generated by state-of-the-art processors could be ineffective, polluting caches, and thrashing resources such as buses and caches, which lead to performance loss and unnecessary energy consumption. Our dynamic approach attempts to address these issues and prevent the over-aggressive prefetches, i.e. those never referenced in L1, from consuming the memory bandwidth and polluting the L1 cache.

4. THE PREFETCH POLLUTION FILTER

In this section, we propose a hardware-based cache pollution filter for processors with aggressive prefetches enabled. The cache pollution filter dynamically determines the effectiveness of a prefetch instruction by employing a history table. The "bad" (or ineffective) prefetches will be discarded based on the lookup results from the history table, thereby preventing L1 cache from being polluted. As discussed in Section 3, a processor with an aggressive prefetching mechanism is under examination. The prefetches include both compiler-inserted prefetch instructions and dynamic prefetches generated by the hardware. At runtime, the prefetch pollution filter determines whether an in-flight prefetch should be performed or not. With such a dynamic implementation, one can maximize the capability of data-prefetching, with both hardware and software techniques, while reducing cache pollution simultaneously.

Figure 3 depicts the anatomy of our prefetch pollution filter design and its relation to an out-of-order processor and its associated cache hierarchy. The prefetch pollution filter is implemented as a stand-alone module that examines data addresses generated from the hardware-based prefetcher, L1 cache, and the LD/ST queue. The hardware prefetch generator is triggered by data accesses to the L1 or L2 cache depending on the prefetch algorithms (the trigger may come from other sources. In our cases, however, the two hardware-based prefetchers are triggered by L1 or L2 cache accesses). The hardware prefetch generator accepts the trigger and reroutes it to the pollution filter to check if the prefetch should be conducted. For software prefetch instructions, they are identified from the LSQ and sent to the pollution filter directly.

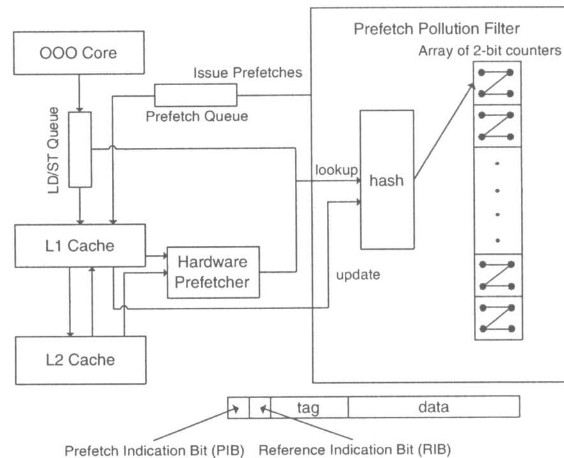

Figure 3: Cache Pollution Filter.

The prefetch filter consists of a single level history table, a hash function and the mechanism to lookup and update the history table. The incoming prefetches are sent to the pollution filter to check against the history table. Either the data cache line address or the program counter (PC) of the instruction triggering the prefetch is hashed and indexed into the corresponding 2-bit counter value of the history table, which indicates whether this prefetch should be performed. If the history table rejects the prefetch, this prefetch operation will be terminated and no prefetch will be issued to the L1 cache; otherwise the prefetch is issued to the prefetch queue. As illustrated in Figure 3, the prefetch queue contends the L1 cache ports with normal L1 memory references issued by the processor.

To collect feedback information, each prefetched cache line is associated with two control bits called *Prefetch Indicator Bit (PIB)* and *Reference Indication Bit (RIB)*. As shown at the bottom of Figure 3, two bits are added to the tag for each cache line. PIB is used to indicate whether this line is brought in by the prefetcher (1 for prefetched lines; 0 for demand misses) and RIB indicates whether this line is ever referenced during its lifetime in L1. RIB is valid only if PIB is set. The overhead of our scheme is insignificant as these additional bits for each cache line are typically found in hardware-based prefetcher for controlling the number of ineffective prefetches. For example, both the NSP and SDP need a bit in each L1 cache line to keep track of the prefetched line while the SDP also includes another reference indication bit, similar to ours, to indicate whether a line is accessed in the L1. Given these shared bits, the overhead of our scheme is primarily in the history table.

Whenever a cache line is replaced and evicted from the L1, its corresponding PIB is checked to see if the line was brought in by prefetching. If yes, its RIB is further checked to see if it was ever referenced. The address of the cache line or the PC together with the RIB are passed to the pollution filter. The history table is then updated accordingly.

Each history table entry uses a two-bit saturation counter. Either the address of the prefetch or the PC of the triggering instruciton is used to index into the history table and depending on whether the prefeched cache line is referenced or not, the two-bit saturation counter is updated. The lookup and update operations to the two-bit counter are the same as those for branch predictors. We will discuss the impact of the length of the history table with respect to the performance in Section 5.3.

Note that the scheme depicted in Figure 3 does not use a dedicated fully-associative prefetch buffer, instead, data are prefetched into the L1 cache directly. Since a dedi-

IEEE COMPUTER SOCIETY

cated prefetch buffer could be more complex and expensive to build due to additional buses, routing, and layout issues, etc. Most of the contemporary microprocessors implemented their data prefetch mechanism in the cache hierarchy in lieu of dedicating a prefetch buffer. Nevertheless, we also evaluate and quantify processor architectures for both design options in Section 5.5.

4.1 PA-based Cache Pollution Filter

Per-Address-based (or PA-based) cache pollution filter tracks the cache line address (address with cache line offset bit stripped) of each prefetch operation issued. Since the same memory instruction may lead to different cache line addresses at different iterations, thus different prefetches could be triggered. The PA-based filter is capable of discerning these various fetched addresses by the same memory instruction. Due to the limited length of the history table, however, the aliasing (or interference) problem could be severe for the PA-based filter.

4.2 PC-based Cache Pollution Filter

A PC-based cache pollution filter tracks the program counter (offset by the instruction size) of each instruction that triggers a prefetch. For prefetches enabled by a software prefetch instruction, the PC is identical to the PC of the software prefetch instruction. For hardware-based prefetch algorithms, the PC of the memory instruction that triggers the prefetch is used. The PC-based filter may not be as precise as the PA-based filter due to sharing among different prefetch addresses from the same trigger, notwithstanding it saves the history table space. Additionally, the PC needs to be passed to the L1 cache and the cache pollution filter through a separate data path.

5. EXPERIMENTAL RESULTS

5.1 System Configuration and Benchmarks

Our experimental infrastructure is based on Simplescalar 3.0 using Alpha binaries. All benchmark programs were compiled using gcc targeting Alpha ISA with -O4 optimization flag which generates software prefetch instructions. The hardware prefetches are assumed being triggered (if necessary) immediately after a cache access without any delay. All duplicate prefetches are squashed automatically with no penalty. All benchmark programs are run up to 300 million instructions. The default configuration parameters are detailed in Table 1. In this study, we target a deep-submicron high performance processor, in which a small L1 is typically employed in exchange of a fast access latency. Hence we assume a default processor with an 8KB direct-mapped L1 cache. Similar schemes have been implemented in commercial high performance processors such as the Pentium 4 processor [9]. Configurations are varied in our experiments, e.g. the L1 cache size, history table size, number of L1 ports, etc. for different evaluation purposes. The default size of the history table has 4096 entries (1KB).

Table 2 shows the properties of benchmark programs used. These 10 programs were selected from the Olden [1] (bh, em3d, perimeter), SPEC95 (ijpeg, fpppp, gcc, wave5) and SPEC2000 (gap, gzip, mcf) benchmark suites. Their input sets, L1 data cache miss rates and L2 data cache miss rates with prefetch turned off are shown in the table.

We first evaluate the performance with an 8KB L1 cache, then we compare the performance results with those of a 32KB L1 cache; in Section 5.3, we study the performance sensitivity of the history table size; and in Section 5.4, we take the number of L1 ports into account; finally, we evaluate our scheme with a dedicated prefetch buffer in Section 5.5.

Processor	
Target Frequency	2 GHz
Issue/Retire	7 inst/cycle
Reorder Buffer	128 entries
Load/Store Queue	64 entries
Branch Predictor	Bimodal, 2048 entries
BTB	4-way, 4096 sets
Caches	
L1 I/D	8KB, 32b line Direct-mapped, 1 cycle
L1 D ports	3
L2 I/D	512KB, 32b line 4-way, 15 cycles
L2 I/D ports	1
Memory	
Latency	150 core cycles
Bus	64-byte wide
Prefetcher	
Queue Length	64 entries
Pollution Filter	
History table	1KB, 4K entries

Table 1: System Configuration

Benchmark	Input data sets	L1 miss%	L2 miss%
bh	2048 bodies	0.0464	0.0026
em3d	100 nodes 10 arity 10K iter	0.2161	0.0001
perimeter	12 Levels	0.0478	0.2709
ijpeg	penguin.ppm	0.0565	0.0235
fpppp	natoms.in	0.0807	0.0003
gcc	cp-decl.i	0.0551	0.0221
wave5	wave5.in	0.1387	0.0209
gap	ref.in	0.0409	0.2247
gzip	input.graphic	0.0597	0.3176
mcf	inp.in	0.0648	0.2426

Table 2: Properties of the benchmark programs

5.2 Performance Evaluation

5.2.1 Default processor model

In Figure 4, we compare the number of prefetches that are bad (ineffective) and good (effective) in the L1 cache for 3 scenarios - (1) without pollution control (no filtering), (2) the PA-based pollution filter, and (3) the PC-based pollution filter. For clarity, all numbers are normalized to the number of good prefetches in case (1). The first 3 bars show the number of bad prefetches. An average of 97% bad prefetches are eliminated with the PA-based filter while nearly 98% of bad prefetches can be removed by the PC-based filter. The next 3 bars shows the number of good prefetches. Despite the pollution filters aim at reducing ineffective prefetches, both of the pollution filters could be too aggressive and filter out effective prefetches as well due to the unpredictability of cache reference behavior. The simulated results of our 10 benchmark programs show that an average of 51% of good prefetches are disabled for the PA-based filter and about 48% for the PC-based filter. With the drastic reduction for bad prefetches, there is a 75% reduction in total prefetch bandwidth for the PA-based filter and a 74% reduction in the PC-based filter. Figure 4 demonstrates that the pollution filters can successfully reduce bad prefetches dynamically with tolerable loss of good ones. Besides, the PA-based filter performs almost on par with the PC-based filter as shown in Figure 4. Also notice that, for some benchmark programs, like the gcc, most of the prefetches are filtrated due to their unpredictable nature even though the prefetches are already ineffective for such programs.

Figure 5 shows the reduction of bad/good prefetch ratio. On average, this number is reduced by 70% for PA-based filtering and 91% for PC-based filtering. Figure 6 compares the simulated Instruction Per Cycle (IPC) numbers for our filters versus the baseline. For all benchmark programs, the IPC numbers are improved, apparently, the reduction

IEEE COMPUTER SOCIETY

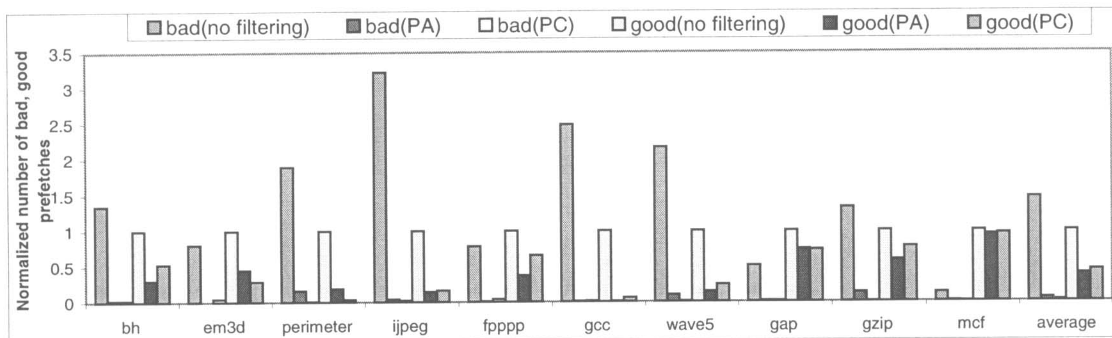

Figure 4: Prefetch miss/hit ratios for 8KB D-cache.

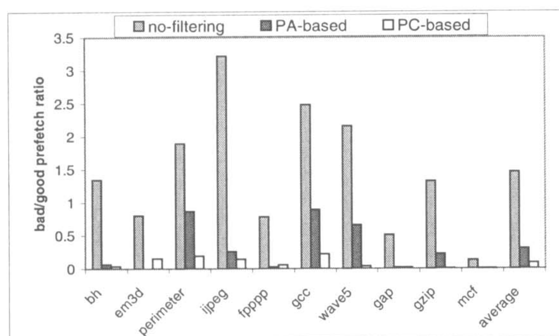

Figure 5: Bad/good prefetch ratios for 8KB D-cache.

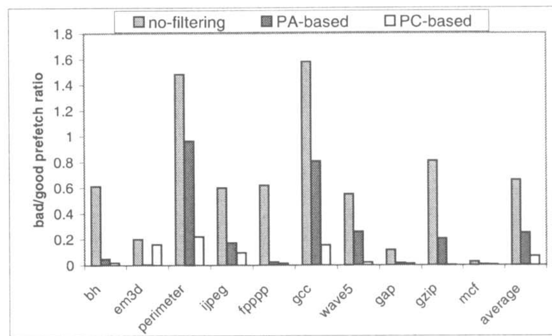

Figure 8: Bad/good prefetch ratios for 32KB D-cache.

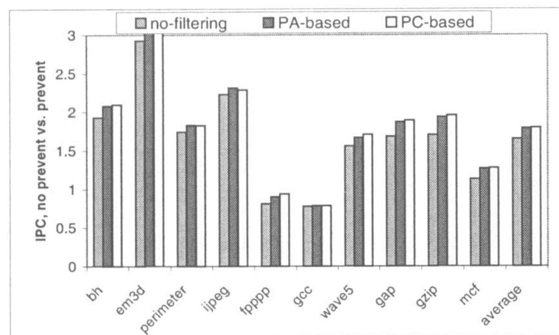

Figure 6: IPC comparison for 8KB D-cache.

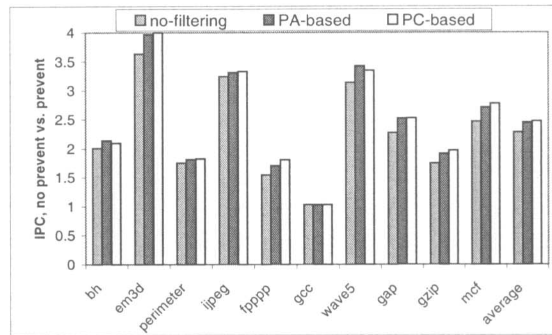

Figure 9: IPC comparison for 32KB D-cache.

of good prefetches is compensated by elimination of bad prefetches. The last column of the table shows the mean for all benchmarks. On average, we achieve 8.2% increase for PA-based pollution filer while 9.1% for the PC-based pollution filter.

We also notice that adding a 1KB history table for cache pollution filtering is actually more effective than simply increasing the cache size. Due to implementation difficulty (a 9KB cache in terms of access speed is less cost-effective due to the 9-way management), we only compare our default model with the one with 16KB L1 cache (other configurations are identical). The speedup for 16KB L1 is about 20%. Reasonably, we can conclude adding a 1KB history table is more desirable.

In addition, we have experimented with the two hardware prefetch algorithms separately (due to the small number of software prefetches and their higher accuracy, the effectiveness of our pollution filter is less conspicuous). For NSP, without pollution filtering, the good/bad prefetch ratio is 1.8 on average. The pollution filter reduces bad prefetches by 97.5% and good prefetches by 48.1%. On the other hand, without pollution filtering, the good/bad ratio for SDP is 11.7. The pollution filter reduces bad prefetches by 68.3% and good prefetches by 61.9%. In conclusion, prefetch algorithm with higher accuracy seems to cause the pollution filtering to perform worse. For advanced features, our pollution filter can be made adaptive to start filtering when the prefetching becomes too aggressive (with low accuracy).

5.2.2 *Processors with 32KB Data Caches*

Figure 7 to Figure 9 repeat the same performance analysis by enlarging the L1 cache size to 32KB. Due to the larger cache size, the L1 access latency is increased to 4 cycles in our simulation as pre-charging the word-lines and signal driven through the bit-lines of the cache now takes

IEEE COMPUTER SOCIETY

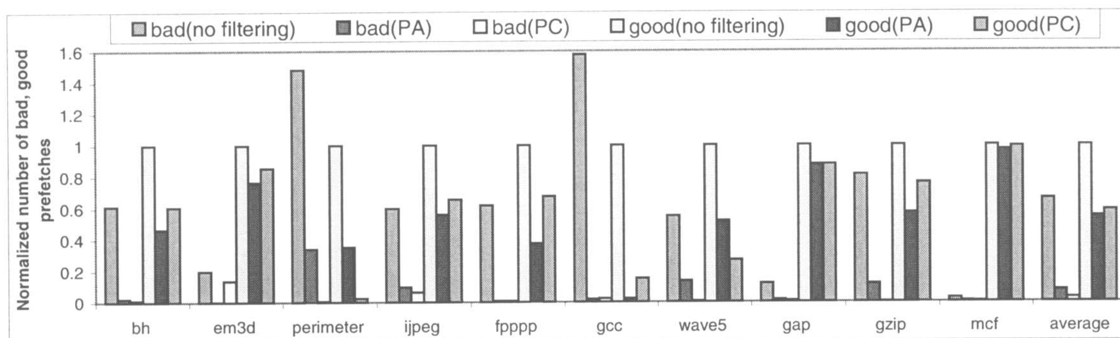

Figure 7: Prefetch miss/hit ratios for 32KB D-cache.

longer for a high frequency processor.

In Figure 7, we present the amount of bad and good prefetch traffics for the 3 scenarios. As expected, the filters greatly filtrated bad prefetches. We observe a 91% reduction of bad prefetches for the PA-based filter and 92% reduction for the PC-based filter. In the meantime, more good prefetches are preserved for the 32KB L1 cache. Only 35% good prefetches are removed for the PA-based filter and 27% for the PC-based filter. Due to reduced conflict and capacity misses for larger caches, our pollution filters are more effective in removing bad prefetches compared against the scenarios of smaller caches as described in Section 5.2.1. In addition, the amount of traffic reduction also confirms our theory that a larger cache leads to a more effective filtering. The PA-based filter reduces 52% prefetch bandwidth requirement; on the other hand, 47% is reduced by the PC-based scheme, which are well below the cases for 8KB L1 cache (75% and 74%). Figure 8 gives reduction for bad/good prefetch ratio. On average, this number is reduced by 75% for PA-based filtering and 93% for PC-based filtering, which are slightly better than the 8KB cache. Figure 9 gives the results of the IPC comparison. Both of the PA-based and the PC-based filters outperform the one without pollution filtering. As shown in the figure, "no filtering" always delivers the worst IPC number. On average, the PA-based filter shows a 7.0% speedup while the PC-based filter improves performance by 8.1%.

In summary, a smaller L1 cache size, also the trend of deep submicron processors, results in a more aggressive filtering. Although a less aggressive pollution filtering preserves more good prefetches, at the same rate, it retains more bad prefetches, hence more bandwidth is consumed by prefetch traffic. The performance largely depends on the trade-off between prefetch traffic reduction and cache pollution reduction. Once prefetch traffic is reduced too much to introduce enough useful prefetches, the performance degrades. As for gcc, the good prefetches are reduced to the extent that it offsets the benefits of traffic reduction.

5.3 Impact of the History Table Size

In this section, different history table sizes are evaluated to quantify their impacts to the overall performance. The size of the history table is varied from 1024 entries (256B), 2048 entries (512 B), 4096 entries (1KB), 8192 entries (2KB) up to 16384 entries (4KB). All the experiments with variable history table sizes were performed using the default configuration. Only the PA-based filter is evaluated.

As the first performance metric, we examine the number of good prefetches in Figure 10. All the numbers shown in this figure are normalized to that of a 4096-entry history table - the default configuration. In general, the number of good prefetches increases as a longer history is employed, indicating effective prefetches are better preserved. A few outliers in Figure 10, such as gap, gzip and mcf, however, show that varying the history table size is almost insensitive in preserving the number of good prefetches. It implies that a small history table, e.g. 1024 entries, is good enough for capturing most of the good prefetches in these benchmarks.

Figure 11 shows the trend of the number of bad prefetches with different history table sizes. It may be surprising that the number of bad prefetches also increases in some benchmarks such as gcc when the history table gets longer. Actually, as shown earlier, these numbers are already small, the absolute numbers of increased bad prefetches are still less than the increase in the number of the good prefetches.

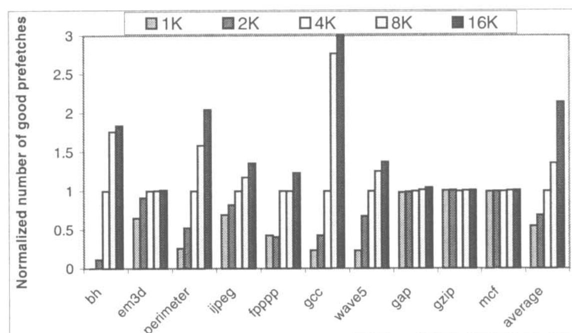

Figure 10: Number of good prefetches for different history table sizes (normalized to 4K entries).

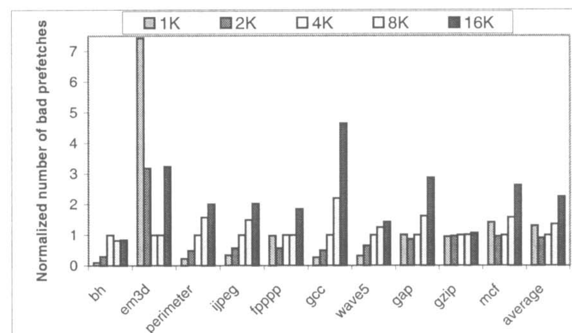

Figure 11: Number of bad prefetches for different history table sizes (normalized to 4K entries).

IEEE COMPUTER SOCIETY

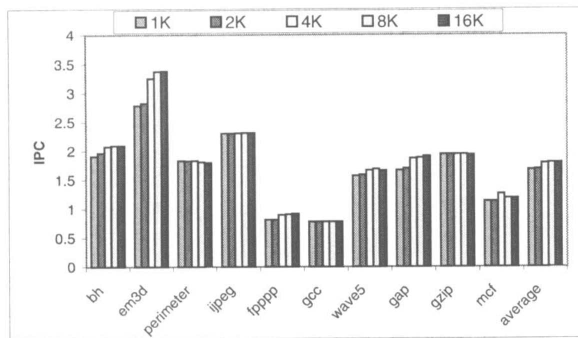

Figure 12: IPC for different history table sizes.

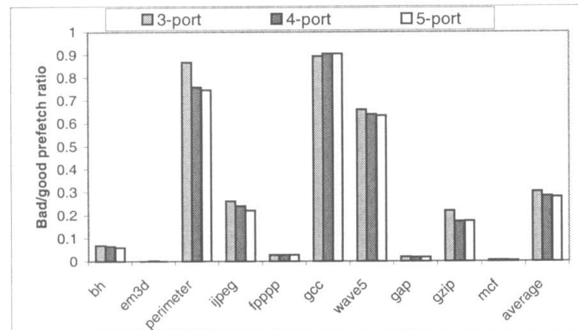

Figure 13: Bad/good prefetch ratios for different numbers of L1 ports.

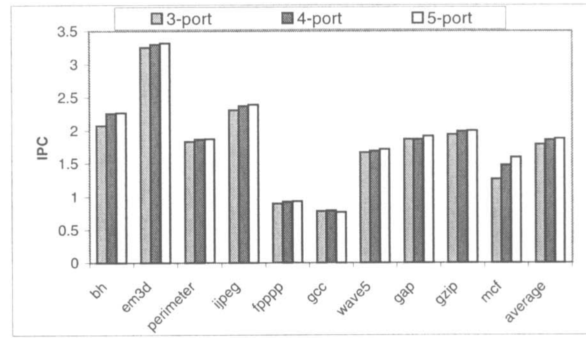

Figure 14: IPC for different numbers of L1 ports.

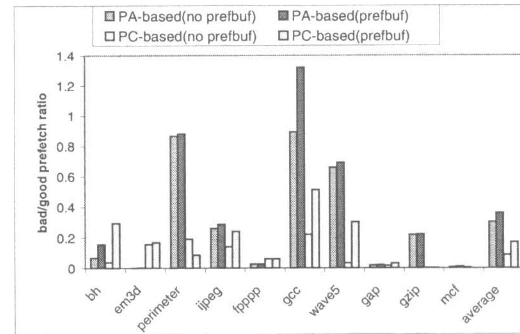

Figure 15: Bad/good prefetch ratio comparison with prefetch buffer.

Another possibility is that all prefetches first mapped to the history table are assumed to be good and issued, if the working set of the program is less than the table size, more bad prefetches may result. For some of the benchmarks, like em3d, gap, and mcf, the minimum is reached for mid-size tables, therefore, a history table too short or too long may not be good.

Figure 12 presents the IPC comparison for different sizes of the history tables. For most programs, the IPC increases slightly with longer tables. The mean shows a 6% improvement from 2048-entry to 4096-entry. Further increase in the table sizes makes little difference in performance, mostly within 1%. In summary, the performance improvement for a history table size over 4096 entries is limited. Moreover, short history tables (1024 or 2048 entries) can affect the performance to some extent. Hardware implementations should choose the size of the history table based on their cost budget. With 4096 entries, the pollution filter will take only 1KB space with direct indexing, a small overhead in future performance processors with one billion transistors available to explore.

5.4 Impact of L1 Cache Ports

Next, the number of L1 cache ports is varied to see how it affects bad/good prefetch ratio and the IPC. All experiments are performed with the default configuration and the PA-based pollution filter. The number of the L1 ports is increased gradually from 3, 4 to 5[1]. Note that additional cache ports lead to a bigger cache design, thus elongating the access latency. We take these physical design constraints into account. For a 4-port 8KB cache, the L1 access latency is assumed 2 cycles and 3 cycles for a 5-port 8KB cache.

[1]Our processor model does not differentiate read ports and write ports. All ports are universal for either reads or writes. The prefetch queue competes for these L1 ports.

The bad/good prefetch ratio is used in this study. Figure 13 shows the bad/good prefetch ratios. For most benchmark programs, this value decreases as more L1 ports are provided. With fewer L1 ports, the competition for the ports is more intense. Consequently, prefetches to the L1 are postponed as they are lined up waiting for the L1 cache ports to become available. This procrastination turns potential good prefetches into bad, if they reach the L1 cache too late. However, our pollution filter should try to adjust the history table for the increased misses (a previously good feedback turns bad, and the table updater must change the setting in the table.) The ratios for 4-port and 5-port L1 caches are quite close. On average, there is a 6% drop from 3-port to 4-port caches and only a 2% drop from 4-port to 5-port.

Figure 14 compares the IPC numbers. In general, the IPC increases with the port number increased. The mean of IPC reflects a 4% speedup from 3-port to 4-port, and less than 1% gain from 4-port to 5-port.

In summary, the port number of L1 cache has a direct impact on the performance of the pollution filter. As shown in the figures, the impact diminishes quickly with the increase of the number of ports, due in part to longer access latency. Therefore, adding more L1 ports, which is expensive in terms of die area, gains marginal benefits when the number of universal ports is over 4, even with an 8-wide issue processor.

5.5 Comparison with a Dedicated Prefetch Buffer

In this section, we evaluate the impacts of a dedicated prefetch buffer with our baseline machine model. All other configurations are kept intact. The prefetch buffer is fully associative with 16 entries. Both PA-based and PC based filters are evaluated and quantified.

IEEE COMPUTER SOCIETY

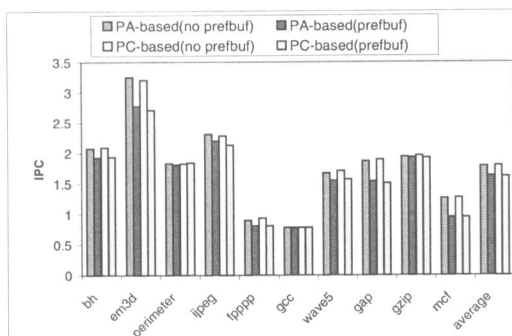

Figure 16: IPC comparison with prefetch buffer.

Prefetch buffer is suggested by [5] to reduce L1 cache pollution by storing prefetched data in a separate buffer. As observed from our experiments, a single prefetch buffer is ineffecitve in reducing bad prefetches when prefetching is done aggressively. This is because 1)Prefetch buffer cannot reduce the prefetch traffic. 2)Distinguish bad/good prefetches simply by restricting their lifetime in the prefetch buffer is not precise. 3)Prefetch buffer is fully associative, so its size cannot be big, which causes some prefetches to be evicted earlier. Our results show that prefetch buffer can only improve IPC by 1 to 2% when all the software and hardware prefetches are enabled.

However, our experiments in the following show that for aggressive prefetching, a small dedicated prefetch buffer is less effective if combined with our pollution filters. In Figure 15 and Figure 16, four schemes were investigated including the PA-based filter with or without a dedicated prefetch buffer and the PC-based filter with or without a dedicated prefetch buffer. In Figure 15, the bad/good prefetch ratio is again used as a metric for comparison. In most of the programs, adding a dedicated prefetch buffer degrades the effectiveness of pollution filters.

In Figure 16, the IPC numbers concur that a dedicated prefetch buffer causes performance penalty. On average, the IPC loses by 9% for the PA-based filter and 10% for the PC-based filter. Note that gcc is almost unaffected, probably due to the small absolute numbers of both bad and good prefetches.

6. CONCLUSIONS

This paper proposes two hardware-based prefetch pollution filtering mechanisms that can significantly reduce the number of bad prefetches (over 98% for an 8KB L1 cache and 92% for a 32KB cache) for architectures with aggressive hardware and software prefetching. The major advantage of employing a cache pollution filter hardware is to enable architectures to encompass several prefetching techniques altogether with dynamic filtering capability to maintain the performance edge. Excessive but ineffective prefetches causing performance degradation are filtered out by the hardware-based filter. We quantified our approach through simulations and showed that our technique mitigates L1 data cache pollution while reducing the prefetch traffics that compete for limited number of the L1 cache ports and finite cache bandwidth. As a result, the IPC, on average, is improved by 7% to 9% for different L1 cache sizes with respect to a machine without any filtering mechanism. We also analyzed and demonstrated that the hardware overheads for implementing the filter. Basically, the history table size can be kept small (1KB or 512B for some benchmarks) while the overhead for the L1 cache is very insignificant as the flags for enabling other hardware prefetching algorithms can be reused. Next, we analyzed the impact of different L1 cache ports and noticed that the improvements are decreased when more cache ports are added. Finally, we compared our baseline machine with a machine featuring a dedicated prefetch buffer.

In conclusion, the prefetch pollution filter offers an effective hardware solution with affordable overheads that improves performance by dynamically controlling the number of bad prefetches generated from aggressive prefetching schemes. For small L1 caches emerging in deep submicron processors, this solution provides a more efficient utilization for the limited resources.

7. ACKNOWLEDGEMENTS

We would like to thank Prof. Santosh Pande, Mr. Weidong Shi and Mr. Tao Zhang from Georgia Tech for their support and help for this work.

8. REFERENCES

[1] M. Carlisle. *Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines.* PhD thesis, Princeton University, Dept. of Computer Science, 1996.

[2] M. J. Charney and A. P. Reeves. Generalized Correlation Based Hardware Prefetching. Technical report, Cornell University, 1995.

[3] T.-F. Chen and J.-L. Baer. Reducing Memory Latency via Non-blocking and Prefetching Caches. In *Proc. of the 5th Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, 1992.

[4] T.-F. Chen and J.-L. Baer. Effective Hardware-Based Data Prefetching for High Performance Processors. *IEEE Trans. on Computers*, Vol. 44, No.5, 1995.

[5] W. Y. Chen, S. A. Mahlke, P. P. Chang, and W.-M. W. Hwu. Data Access Microarchitectures for Superscalar Processors with Compiler-Assisted Data Prefetching. In *Proc. of Int'l Symp. on Microarchitecture*, 1991.

[6] Compaq Computer Corporation. *Alpha Architecture Handbook*, October 1998.

[7] F. Dahlgren, M. Dubois, and P. Stenstrom. Fixed and Adaptive Sequential Prefetching in Shared-memory Multiprocessors. In *Proc. of the 1993 Int'l Conf. on Parallel Processing*, 1993.

[8] J. W. C. Fu, J. H. Patel, and B. L. Janssens. Stride Directed Prefetching in Scalar Processors. In *Proc. of the 25th Int'l Symp. on Microarchitecture*, 1992.

[9] G. Hinton, D. Sagar, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, Q1 Issue, 2001.

[10] K. Chan K, C. C. Hay, J. R. Keller, G. P. Kurpanek, F. X. Schumacher, and J. Zheng. Design of the HP PA 7200 CPU. *Hewlett-Packard Journal*, Vol. 47, No.1, 1996.

[11] A. Lai, C. Fide, and B. Falsafi. Dead-block Prediction and Dead-block Correlating Prefetchers. In *Proc. of the 28th Int'l Symp. on Computer Architecture*, 2001.

[12] C.-K. Luk and T. C. Mowry. Automatic Compiler-Inserted Prefetching for Pointer-Based Applications. *IEEE Trans. on Computers*, February 1999.

[13] J. Pomerene, T. Puzak, R. Rechtschaffen, and F. Sparacio. Prefetching System for a Cache Having a Second Directory for Sequentially Accessed Block. U.S.Patent #4,807,110, February 1989.

[14] A. K. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Application.* PhD thesis, Rice University, 1989.

[15] S. Przybylski. The Performance Impact of Block Sizes and Fetch Strategies. In *Proc. of the 17th Int'l Symp. on Computer Architecture*, 1990.

[16] A. J. Smith. Cache Memories. *Computing Surveys*, Vol. 14, No. 3, 1982.

[17] V. Srinivasan, E. S. Davidson, and G. S. Tyson. A Prefetch Taxonomy. *To appear in IEEE Trans. on Computers.*

[18] V. Srinivasan, G. Tyson, and E. Davidson. A Static Filter for Reducing Prefetch Traffic. Technical Report CSE-TR-400-99, University of Michigan, 1999.

[19] Z. Wang, K. S. McKinley, A. L. Rosenberg, and C. C. Weems. Using the Compiler to Improve Cache Replacement Decisions. In *Proc. of Int'l Conf. on Parallel Architectures and Compiler Techniques*, 2002.

[20] Y. Wu, R. Rakvic, L.-L. Chen, C.-C. Miao, G. Chrysos, and J. Fang. Compiler Managed Micro-cache Bypassing for High Performance EPIC Processors. In *Proc. of the 35th Int'l Symp. on Microarchitecture*, 2002.

0190-3918/03 $ 17.00 © 2003 IEEE

IEEE COMPUTER SOCIETY