Reducing Cache Pollution via Dynamic Data Prefetch Filtering

Xiaotong Zhuang and Hsien-Hsin S. Lee, *Member*, *IEEE*

Abstract—In order to bridge the gap of the growing speed disparity between processors and their memory subsystems, aggressive prefetch mechanisms, either hardware-based or compiler-assisted, are employed to hide memory latencies. As the first-level cache gets smaller in deep submicron processor design for fast cache accesses, data cache pollution caused by overly aggressive prefetch mechanisms will become a major performance concern. Ineffective prefetches not only offset the benefits of benign prefetches due to pollution but also throttle bus bandwidth, leading to an overall performance degradation. In this paper, we propose and analyze a number of hardware-based prefetch pollution filtering mechanisms to differentiate good and bad prefetches dynamically based on history information. We designed three prefetch pollution filters organized as a one-level, two-level, or gshare style. In addition, we examine two table indexing schemes: Per-Address (PA) based and Program Counter (PC) based. Our prefetch pollution filters work in tandem with both hardware and software prefetchers. As our analysis shows, the cache pollution filters can reduce the ineffective prefetches by more than 90 percent and alleviate the excessive memory bandwidth induced by them. Also, the performance can be improved by up to 16 percent when our filtering mechanism is incorporated with aggressive prefetch filters as a result of reduced cache pollution and less competition for the limited number of cache ports. In addition, a number of sensitivity studies are performed to provide more understandings of the prefetch pollution filter design.

Index Terms—Prefetch, memory subsystems, microarchitecture.

1 Introduction

The growing speed disparity between CPU and main memory poses a great challenge for the scalability and design effectiveness of modern processors. Even though data caches can somehow bridge this gap, data references that miss caches still suffer from long memory lead-off latencies if there is not a large enough number of independent instructions to mask the delay. The problem is exacerbated in static machines, e.g., Intel/HP's Itanium, where missed data dependent instructions will stall the entire pipeline processing. Prefetching, either hardware-based or compiler-assisted, has been extensively studied and shown to be an effective means for hiding memory latency. Instead of waiting for actual memory requests issued by load/store instructions, an effective prefetching scheme can bring data into the memory hierarchy closer to the processor prior to their needs.

1.1 Data Prefetching

Most prefetch techniques are prediction-based, the accuracy and potential performance gain highly depend on the predictability of memory reference behavior. Simple hardware-based prefetching techniques proposed in [1], [2], [3] attempt to identify and capture regular data access patterns with unit strides. More sophisticated hardware-based schemes [4], [5] can issue prefetches for sequential data accesses with arbitrary but constant strides. In [6], [7], Chen and Baer proposed a reference prediction table to monitor data reference patterns and issue prefetches dynamically. Correlation-based prefetching [8], [9], [10] keeps prior cache miss addresses and triggers prefetches by correlating subsequent misses to the history. Recent work by Nesbit et al. [11], [12] enables prefetches with a global history buffer that holds the most recent miss addresses in FIFO order and links entries sharing a common property. In this manner, stale data can be removed from the table, improving the accuracy of correlation-based prefetching. A unified evaluation framework called MicroLib [13] implements hardware prefetchers published in the literature and demonstrates that, under a fair comparison, even the best prefetcher only achieves a very incremental improvement over a decade-old next sequence prefetcher [3] and the next sequence prefetcher is the best among those that prefetch into the L1 cache.

On the other hand, static analysis techniques were applied at compilation time to assist software prefetching [14], [3], which inserts prefetch instructions within the binaries for runtime prefetching. Many contemporary microprocessor instruction sets feature some flavors of cache line fetch instructions that simply move data into the cache without intervening other architectural resources. For example, prefetchnta, prefetcht0/t1/t2 instructions provided in Intel's IA32 processors [15] or, in Alpha ISA, a load instruction can be used to perform data prefetch when specifying the destination register to be $r31, which is hardwired to zero [16]. Since these prefetch instructions are implemented nonblocking, the processor will continue execution without awaiting their completion.

- *X. Zhuang is with the IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598. E-mail: xzhuang@us.ibm.com.*
- *H.-H.S. Lee is with the School of Electrical and Computer Engineering, Georgia Institute of Technology, 777 Atlantic Dr., Atlanta, GA 30332-0250. E-mail: leehs@gatech.edu.*

Manuscript received 8 Mar. 2005; revised 6 Sept. 2005; accepted 7 July 2006; published online 22 Nov. 2006.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0072-0305.

1.2 Aggressive Prefetching

With the advent of billion transistor processors, processor architects continue to dedicate these exponentially increased resources to cache memory on the processor cores with a deeper hierarchy to alleviate the impact of wire delays. Meanwhile, sustainable memory bandwidth also becomes larger between caches and main memory due to improved bus frequency and width. As a result, 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, in fact, is getting smaller in order to guarantee faster L1 accesses, typically in one or two processor cycles for GHz processors. It is also less expensive, in terms of area and power consumption, to build a smaller multiported cache for wide-issue machines which need to process many memory requests simultaneously. Given this design trend, overly aggressive usage of prefetches will not only postpone normal L1 cache accesses but can also pollute the L1 cache, a small and precious memory resource, uninvitingly, leading to an ineffectual use of prefetches.

1.3 Cache Pollution

No data prefetching algorithm can guarantee 100 percent accuracy and effectiveness. A prefetched cache line could turn out to be either completely useless or ineffective. Many implementations place prefetched data in the data cache where they compete for the available cache resources, seriously degrading the performance if the L1 cache is too small. Evicting useful data in the cache due to overly aggressive prefetches causes cache pollution, which unnecessarily reduces the overall performance. Performance can also be significantly degraded when prefetching is imprecise. For example, a stride-based prefetching scheme can be completely ineffective for pointer-based type applications, thereby polluting the data cache. In [17], Luk and Mowry proposed three prefetching schemes for pointer-based applications. Their work shows that prefetch misses can be as high as 80 percent for some benchmark programs, which include those prefetched data that are not eventually accessed by the application or evicted before accessed because they were issued either too early or too late. Srinivasan et al. [18] show that 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 when adopting a prefetching technique. In summary, an inappropriate prefetch can lead to undesirable outcomes by 1) occupying cache space with useless data if the prefetcher is inaccurate and causes more capacity or conflict misses or 2) imposing higher pressure on the competition for finite bandwidth and limited number of cache ports and write-back buffers, especially for aggressive prefetchers on a wide-issue machine. Puzak et al. [19] formulated a method for evaluating prefetching algorithms. They found that, in many cases, prefetching could lose performance if the conditions are not ideal (perfect coverage and accuracy, sufficient timeliness, and ample bandwidth).

In this paper, we first investigate the impact of aggressive prefetching on conventional cache architectures targeting for deep submicron processors. Three different prefetching schemes were evaluated, including software prefetch instructions 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 by showing their high percentage among all prefetches and the L1 traffic. This motivates our endeavor to design a hardware-based cache pollution filter that can effectively prevent the bad prefetches from entering the cache. The filtering is achieved via exploiting historical prefetch behavior. We propose three filtering algorithms, consisting of a one-level, two-level, or gshare style history table incorporating branch history information. We investigate two prediction schemes: The first one is based on the cache line address of the prefetched data called *Per-Address-based* and the second one is based on the program counter value of the prefetch trigger instruction or *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 reviews related approaches. Section 3 gives our motivation. Our proposed filtering hardware designs are described in Section 4. We then evaluate the performance of our filtering scheme in Section 5. Section 6 concludes this work.

2 Related Work

Several previous works were proposed to reduce cache pollution caused by prefetching. These techniques can be classified into three categories—software-based by compilers [20], hardware-based [7], [21], [22], and hybrid [18], [23]. Chen et al. [24] 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 the next level in memory hierarchy. Typically, a prefetch buffer is fully associative. Note that, when accessed in parallel with the L1, the prefetch buffer could become the critical path if it cannot keep up with the speed of the L1, thus limiting the prefetch buffer size.

Lin et al. [25] proposed a technique to filter superfluous prefetches with density vectors. The work is based on a special prefetcher called scheduled region prefetcher. They measure the predictability of spatial locality using density vectors-bit vectors that track the block-level access pattern within a region of memory. The evaluation shows over 70 percent of useless prefetches can be eliminated, whereas, in this work, we intend to develop a more general approach to filter polluting prefetches. We hope the filter can work with a wide range of prefetchers.

In [20], 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. [21] proposed detecting dead cache lines in caches and replacing the dead lines with prefetched data. Their mechanism aims at reducing situations where useful data are evicted from the cache too early. While having a similar goal of reducing cache pollution, our approach focuses on eliminating ineffective or bad prefetches from entering the cache.

Srinivasan et al. presented a comprehensive taxonomy in [18] that classifies prefetches based on traffic and misses generated by each prefetch. They also proposed a static filter in [26] aimed at reducing the number of polluting prefetches. The static filter collects information on the polluting prefetches offline through profiling and uses this profiling information to guide data prefetches. They reported a 2 to 4 percent performance improvement of their static filter scheme combined with Next Sequence Prefetching and Shadow Directory Prefetching. Ideally, the profiling information can provide precise global information for a given input data set; however, it lacks 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 collection is needed. It seems the performance gain of our scheme is higher than their published results.

Recently, [22] proposed using the L1 cache as a filter to filter out useless prefetches and memory accesses on a speculative path. In their scheme, if a speculatively fetched block is referenced by a nonspeculative instruction while it resides in the L1 cache, then the speculatively fetched block is treated as any other nonspeculatively fetched block during eviction; otherwise, the processor will assume that the block is unlikely to be used prior to its eviction from the L2 cache. In this manner, speculative memory accesses bring data only into the L1 cache rather than into all levels of the cache hierarchy.

3 Motivation

We simply classify prefetches into two categories: 1) good or effective—those referenced at least once in the cache prior to their eviction—and 2) bad or ineffective—those never referenced during their lifetime in the cache. A comprehensive prefetch taxonomy [18] requires many additional bits to keep track of the replaced cache line and reference order for both displaced and prefetched cache lines; our simple yet competent classification greatly simplifies the hardware implementation. Fig. 1 shows the distribution of the prefetches based on our classification for 10 benchmark programs selected from the SPEC95, SPEC2000, and Olden benchmark suites. Due to page limitations, we are not able to show the evaluation with all known prefetchers. We select three representative prefetchers. The same set of prefetchers was used by a main related work [18] about the static prefetch filter. The prefetches include both hardware-based (next sequence prefetching—NSP [3]—and shadow directory prefetching—SDP [27]) as well as software-based prefetches. Note

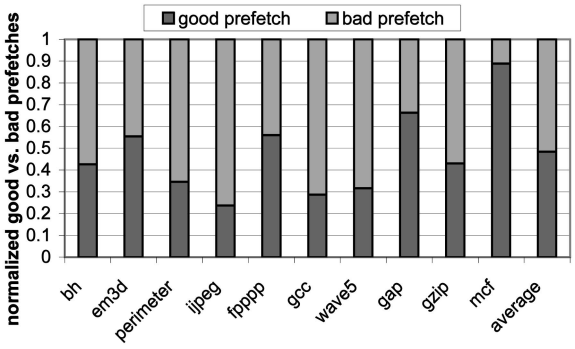

Fig. 1. The effectiveness of prefetches.

that the number of software prefetches is far less than the hardware prefetches but with higher accuracy.

The NSP prefetcher prefetches the next adjacent cache line when a memory access either misses the L1 or hits a cache line in L1. The SDP maintains a shadow line address for 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. The accessing of an L2 cache line would trigger the prefetching of the shadow line into both the L1 and L2 caches. Notice that prior hardware-based prefetchers have attempted to reduce ineffective prefetches. For example, an improved version of the NSP prefetcher employs a tag bit associated with each cache line. When a cache line is prefetched, its corresponding tag bit is set. Then, the prefetcher only prefetches the next cache line when a memory access either misses the L1 or hits a tagged cache line. Similarly, the SDP prefetcher keeps a confirmation bit for each L2 line, indicating if the prefetched line was ever used since it was prefetched last time. Prefetches are only enabled for those that were used after the previous prefetching.

In Fig. 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 indicated, more than half of the prefetches are ineffective or bad in four out of the 10 benchmark programs. Our statistics show, on average, 48 percent of prefetched data are not referenced during their lifetime in the cache.

Fig. 2 shows the traffic distribution in terms of cache lines for the L1 data cache. Obviously, the traffic induced by prefetches accounts for a significant portion of the total traffic to the L1 cache. On average, the prefetch access to normal access ratio is 0.41 with a maximum of 0.57 (ijpeg) and minimum of 0.29 (gzip). In other words, on average, about $\frac{2}{7}$ traffics to the L1 cache are prefetches. Combined with Fig. 1, it implies that 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, leading to performance degradation and unnecessary energy consumption. Our dynamic approach aims at addressing these issues by preventing the overly aggressive prefetches from consuming memory bandwidth and polluting the cache.

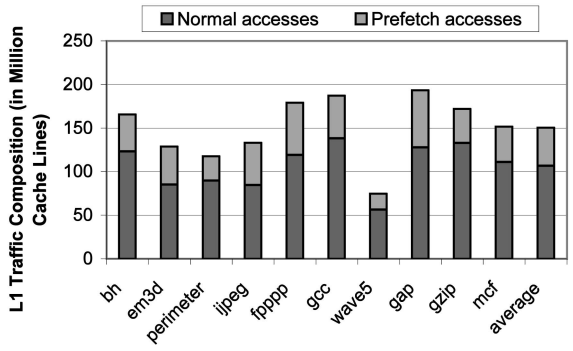

Fig. 2. Traffic distribution of the L1 cache.

4 Prefetch Pollution Filter

In this section, we propose a hardware-based cache pollution filtering scheme for processors with aggressive prefetches enabled. The history-based prefetch pollution filter dynamically determines the effectiveness for each prefetch. The bad (or ineffective) prefetches will be disabled based on the lookup results from the history table to prevent the L1 cache from being polluted. The prefetches under examination include compiler-inserted prefetch instructions and dynamic prefetches generated by prefetch hardware. At runtime, the prefetch pollution filter determines whether an in-flight prefetch should be performed or not. Using such a dynamic implementation, one can maximize the capability of all the hardware and software data prefetching schemes and rely on the prefetch pollution filter to intelligently select the effective prefetches.

4.1 Overview of the Filtering Mechanism

Fig. 3 depicts the overall diagram of our pollution filtering mechanism associated with an out-of-order processor. The prefetch pollution filter is implemented as a standalone module that examines addresses generated from the hardware-based prefetcher, L1 cache controller, and the LD/ST queue (LSQ). 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 prefetch pollution filter to check if the prefetch operation should proceed. Software prefetch instructions are identified from the LSQ and sent to the prefetch pollution filter directly.

Incoming prefetches are sent to the prefetch pollution filter to check whether they should actually be issued. Either the data cache line address or the program counter (PC) of the instruction triggering the prefetch is used for the checking. If the prefetch pollution filter 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. We prioritize demand requests over prefetch requests. As long as the prefetch queue is not full, demand requests are always issued to the cache without being stalled. Moreover, all repeating recent prefetches are disabled. To collect feedback information, each prefetched cache line is associated with two control

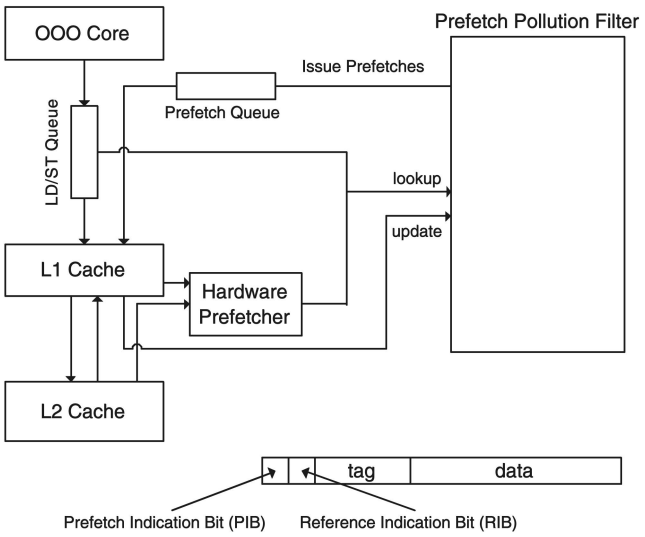

Fig. 3. Overall diagram for the prefetch pollution filtering mechanism.

bits: the *Prefetch Indicator Bit (PIB)* and the *Reference Indication Bit (RIB)*. As shown at the bottom of Fig. 3, these two bits are appended to each cache line tag. 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. 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. This information is then used to update the prefetch pollution filter.

4.2 Three Types of Prefetch Pollution Filters

The critical component is the prefetch pollution filter. In this paper, we study several alternative architectures in constructing the prefetch pollution filter. In Fig. 4, we study three types of prefetch pollution filters. Our design of the prefetch pollution filter bears a similarity to the traditional branch predictors. An array of 2-bit up/down saturation counters is used to record the history of prefetches and to judge if a prefetch should be conducted. They are also updated according to whether the prefetches are actually good or not. We call these 2-bit saturation counters *Prefetch History Table (PHT)*. The number of 2-bit saturation counters approximates the additional on-chip space required.

In our schemes, either the address of the accessed cache line or the program counter (PC) of the triggering instruction can be used as part of the index to the PHT. Here, we use *Per-Address-based (or PA-based)* and *Program-Counter-based (or PC-based)* to represent these two different prefetch pollution filters. The PA-based prefetch pollution filter tracks the cache line address of each prefetch operation issued. Since the same memory instruction may access different cache line addresses at different iterations or from different control flow paths, different prefetches may be triggered. The PA-based filter is capable of discerning these various fetched addresses by the same memory instruction. On the other hand, a PC-based cache pollution filter tracks each instruction's PC that triggers a

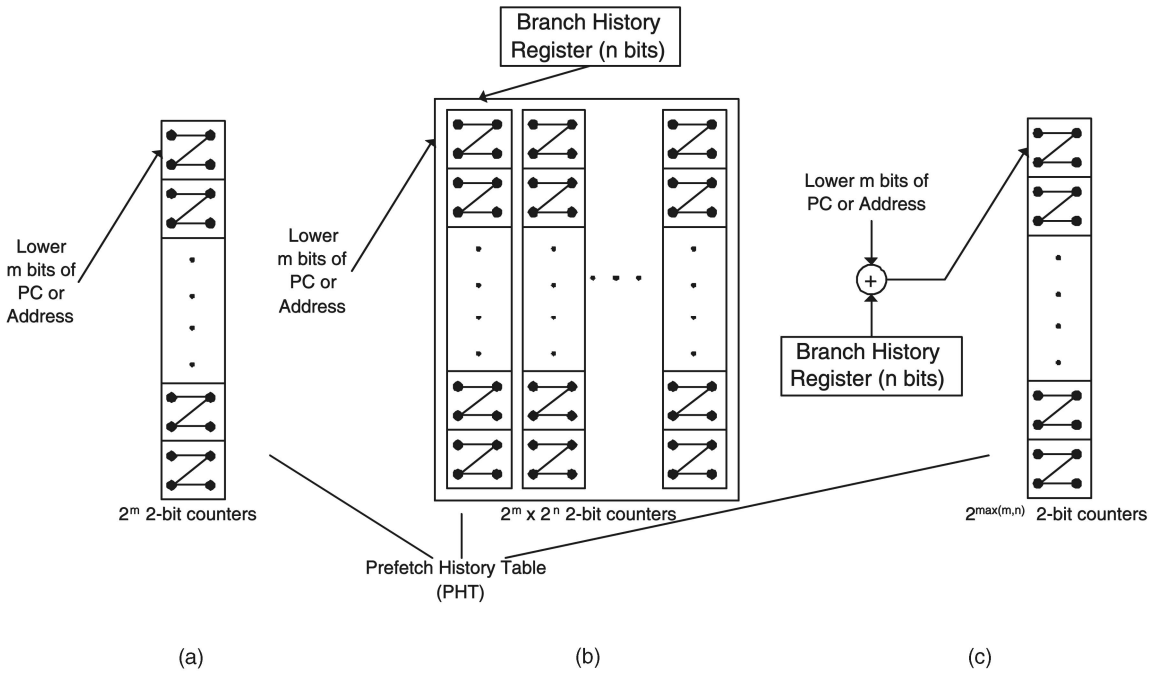

Fig. 4. Three types of prefetch pollution filters.

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.

Fig. 4a shows a single level prefetch pollution filter. It consists of 2^m 2-bit saturation counters. The lower m bits of the accessed address or the PC of the triggering instruction are used as an index for the 2-bit counter array lookup. Also, depending on whether the prefetched cache line is referenced or not, the corresponding two-bit saturation counter is updated. The lookup and update operations to the two-bit counters are the same as those for dynamic branch predictors.

To further improve the accuracy, we can couple it with the branch history register as in branch predictors. By keeping track of the directions of the last n branches encountered along the path, we can determine more precisely whether the prefetch leads to pollution. This is achieved by including an n-bit Branch History Register (BHR). In Fig. 4b, we illustrate a two-level structure for the prefetch pollution filter. The n-bit BHR is globally shared, recording the directions of the last n branches. Together with the lower m bits of the accessed address or the PC of the triggering instruction, a total of 2^{m+n} 2-bit counters are needed. For each accessed address or the PC of the triggering instruction, the lower m bits are used to select the row number, while the BHR indexes the column number of the 2-bit counter array for lookup or update.

Notice that there could be a number of variations to the two level branch predictors [28], [29], [30]. For example, the BHR could be on a per-address or per-set basis, while the second level 2-bit counters could be globally shared by all BHRs to reduce the PHT size. In such a design, the recent n-branch history of each branch instruction or each set of branch instructions is recorded by the corresponding BHR. Yeh and Patt [30] show that such a design could be more area-efficient. However, for prefetch pollution filtering, maintaining per-address or per-set BHRs does not make sense because we are looking at individual prefetches instead of branches. Therefore, we cannot establish a branch history for each prefetch, while, in Fig. 4b, a global BHR tracks the directions of last n branches along the path to the prefetch. In this way, we can filter prefetches differently according to the path taken.

Finally, Fig. 4c illustrates a gshare-style prefetch pollution filter. A gshare branch predictor [31] was proposed to alleviate aliasing issues and reduce the number of two-bit counters without losing the branch history bits. A similar idea is applied to prefetch pollution filtering to reduce on-chip die area. As shown in Fig. 4c, the lower m bits of the prefetch address or the triggering PC are XORed with the branch history register to be used to index into the $2^{max(m,n)}$ 2-bit counter array.

Notice that, due to the limited number of 2-bit counters in the prefetch history table, the aliasing (or interference) issue is inevitable for the PA-based filter. On the other hand, 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 that it reduces the number of items that will be indexed into the filter. Additionally, the PC needs to be passed to the L1 cache and the prefetch pollution filter through a separate datapath.

It is noteworthy that the diagram depicted in Fig. 3 does not use a dedicated fully associative prefetch buffer; instead, data are prefetched into the L1 cache directly since a dedicated prefetch buffer is 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

TABLE 1
System Configuration

Processor		Caches	
Frequency	2GHz	L1 I/D Caches	8KB, 32B line Direct-mapped, 1 cycle
Issue/Retire	8 inst/cycle		
Reorder Buffer	128 entries	L1 D ports	3
Load/Store Queue	64 entries	L2 Cache	512KB, 32B line 4-way, 15 cycles
Branch Predictor	2Bc-gskew, 4*8k entries 10 bit history	L2 port	1
BTB	4-way, 4096 sets	Prefetcher & Filter	
Memory		Queue Len	64 entries
Latency	140-10-10-10 core cycles	History Table	1KB, 4K entries
		BHR size	4 bits

TABLE 2
Properties of the Benchmark Programs

Benchmark	Input data sets	L1 miss	L2 miss
bh	2048 bodies	0.0482	0.0023
em3d	100 nodes 10 arity 10K iter	0.2164	0.0001
perimeter	12 Levels	0.0468	0.2776
ijpeg	penguin.ppm	0.0594	0.0228
fpppp	natoms.in	0.0808	0.0003
gcc	cp-decl.i	0.0562	0.0227
wave5	wave5.in	0.1393	0.0209
gap	ref.in	0.0412	0.2237
gzip	input.graphic	0.0605	0.311
mcf	inp.in	0.0649	0.2457

evaluate and quantify processor architectures for both design options in Section 5.6.

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 an -O4 optimization flag that enables software prefetch instruction generation. The hardware prefetches are assumed to be triggered (if necessary) immediately after a cache access without any delay. All duplicate prefetches are squashed automatically without incurring any penalty. All benchmark programs were 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 8KB direct-mapped L1 cache is employed in exchange for a fast access latency. Similar schemes have been implemented in commercial high performance processors such as the Pentium 4 processor [32]. Configurations are varied in our experiments, e.g., the L1 cache size, history table size, branch history register size, number of L1 ports, etc., for different evaluation purposes.

The default size of the history table has 4,096 entries (1KB). For the two-level and gshare prefetch pollution filters, we set the branch history register (BHR) size to be 4 bits. As mentioned earlier, for the two-level filter, the history table indexing method uses partial bits of the branch history and partial address/PC of the prefetch. In our default model, the total number of bits for the index is 12 bits, i.e., 4K entries, then (12-n) bits, where n is the size of the BHR are from the address/PC of the prefetch. On the other hand, the gshare filter always takes 12 bits (under the default model) from the address/PC, but XORs with the BHR (typically smaller than 12 bits).

We use IPC as the metrics for performance evaluation. Note that the software prefetcher actually adds more instructions for prefetching, which could unfairly increase the IPC values and skew the evaluation results against other types of prefetchers. However, as our data show, the number of executed (runtime) software prefetches is very small (less than 1 percent) among the total number of executed instructions with -O4 optimization; therefore, it should not affect the proper comparison in our results.

Table 2 shows the properties of benchmark programs used. These 10 programs were selected from the Olden [33] (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 select these benchmarks for their high cache miss rates. Otherwise, aggressive prefetching will not be needed for tackling the memory bottleneck.

5.2 Performance Evaluation

5.2.1 Default Processor Model

In Fig. 5, we show the percentage of reduction of bad prefetches for the three prefetch filters and the two indexing methods proposed, which contain six cases—one-level with PA, two-level with PA, gshare with PA, one-level with PC, two-level with PC, and gshare with PC. For clarity, all numbers are normalized to the number of bad prefetches when no filtering is applied. The first three bars are for PA-based prefetch filters and the next three bars are for PC-based prefetch filters. We observe a significant number of bad prefetches are eliminated. Also, both the 2-level and gshare filters show slightly better results than the single level filters, indicating that branch information is helpful for removing bad prefetches. In addition, gshare is more effective in reducing the bad prefetches for more than half of the benchmarks, although the difference is less significant. On average, 95.9 percent of the bad prefetches are removed for PA-based one-level prefetch filter, for PA-based two-level, and, for gshare prefetch filters, the numbers are 97.7 percent and 97.6 percent, respectively. On the other hand, the PC-based prefetch filters perform slightly better than the PA-based ones. On average, 97.7 percent, 98.0 percent, and 98.5 percent, respectively, of bad prefetches are removed by the single level filter, two-level filter, and gshare filter. The reduction in bad prefetches is quite consistent across all benchmarks as we can conclude that at least 90 percent of the bad prefetches are reduced in all cases.

Despite the fact that pollution filters aim to reduce ineffective prefetches, they could be too aggressive and filter out effective prefetches as well due to the unpredictability of cache reference behavior. In Fig. 6, we show the amount of good prefetch reduction. It appears that some benchmarks can preserve most of the good prefetches, such as gap, mcf, while some others eliminate many good prefetches as well (e.g., gcc, perimeter, ijpeg). The simulation results indicate that, on average, the reductions for

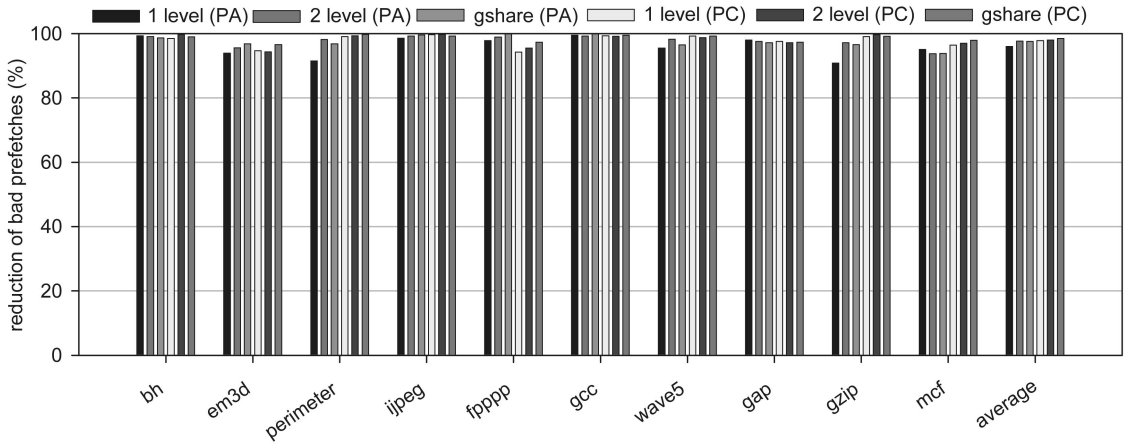

Fig. 5. Comparison of bad prefetches for 8KB data cache.

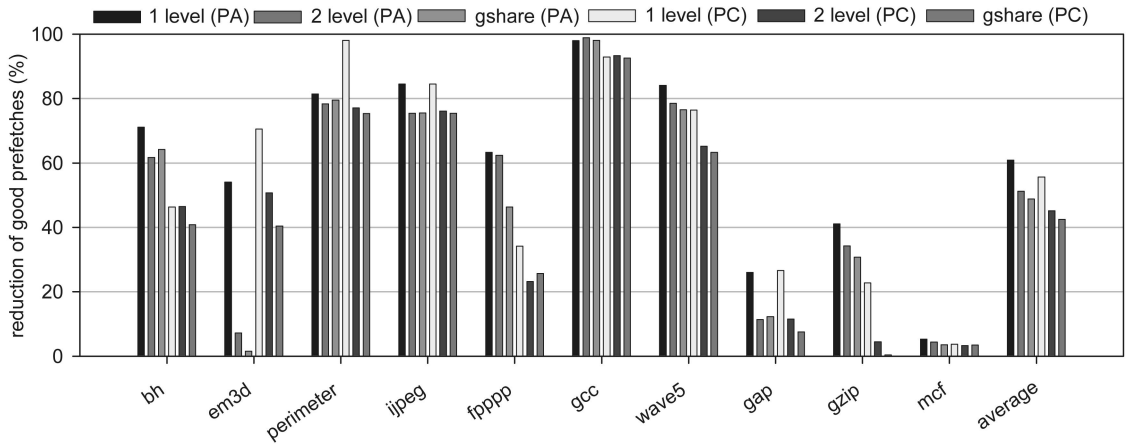

Fig. 6. Comparison of good prefetches for an 8KB data cache.

PA-based one-level, two-level, gshare filters are 60.9 percent, 51.3 percent, and 48.8 percent, respectively. As mentioned before, the PC-based filters reduce bad prefetches more aggressively, but, at the same time, they disable fewer good prefetches. The average reductions for one-level, two-level, and gshare filters are 55.6 percent, 45.1 percent, and 48.8 percent, respectively. Both the two-level and gshare filters remove fewer good prefetches. Both the two-level and gshare filters are quite close in either PA-based or PC-based filtering. The gshare filter reduces fewer good prefetches (by about 3 percent) than the two-level one.

Fig. 5 and Fig. 6 demonstrate that the pollution filters can successfully reduce the bad prefetches dynamically with a tolerable loss of the good ones. Besides, the PA-based filter performs a little bit worse than the PC-based one. Among the three filters, aggressiveness increased from one level, two level to gshare in terms of their ability to disable bad prefetches and preserve good prefetches. Also notice that, for some benchmark programs like the gcc, most of the prefetches are filtered due to their unpredictable nature even though the prefetches are already ineffective for such programs.

From Fig. 7, we notice that, for all benchmark programs, the IPC numbers are improved; apparently the reduction of good prefetches is compensated for by the significant reduction of bad prefetches. Here, the first bar is the IPC numbers of benchmarks without any filtering. The average IPC improvements are 8.2 percent, 12.8 percent, and 13.3 percent for PA-based one-level, two-level, and gshare filters. For PC-based filters, we observe larger improvements primarily due to more bad prefetches being removed. The average speedups are 9.0 percent, 15.1 percent, and 16.2 percent for PA-based one-level, two-level, and gshare filters. Among the three filters, gshare and two-level filters perform much better than the one-level filter. This could be explained by their aggressive reduction of bad prefetches. Although the loss of some good prefetches hampers their effectiveness, the overall prefetch traffic is cut down as well, which brings about the increased speedup due to less cache port competition.

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 will be 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 percent. Reasonably, we can conclude that adding a 1KB history table is more desirable.

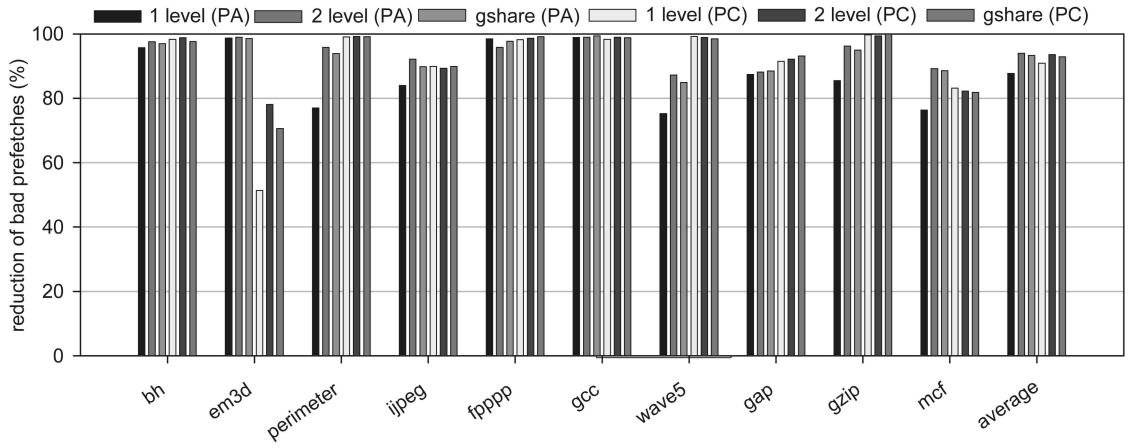

Fig. 7. Comparison of IPC for an 8KB data cache.

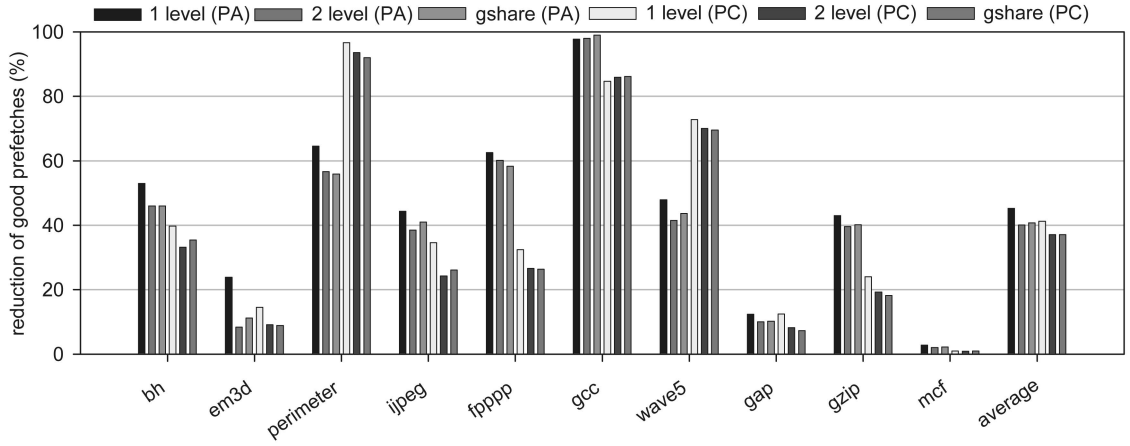

Fig. 8. Comparison of bad prefetches for a 32KB data cache.

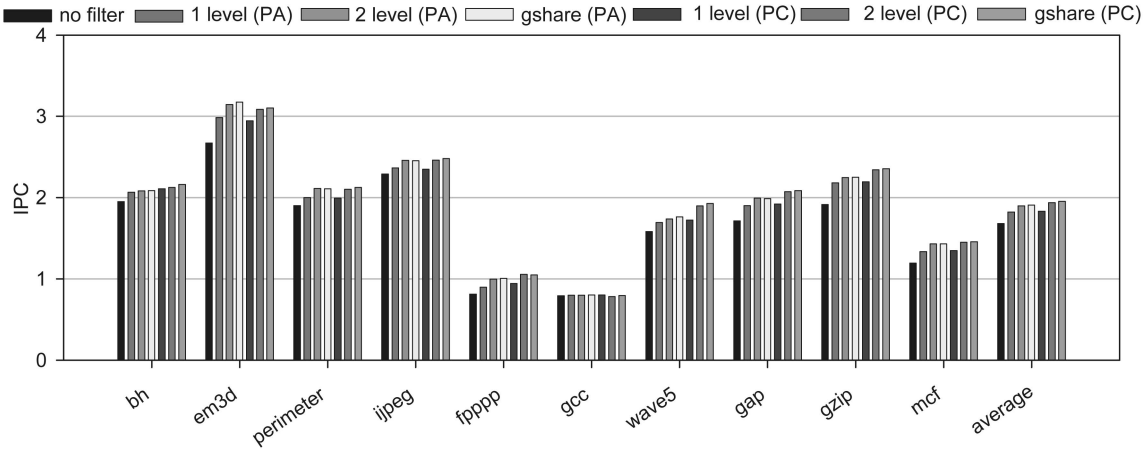

Fig. 9. Comparison of good prefetches for a 32KB data cache.

5.2.2 Processors with 32KB Data Caches

Next, we repeat the same set of experiments and performance analysis by extending the L1 cache to 32KB 4-way set-associative. Due to a larger cache size, the L1 access latency is increased to four cycles in our simulation as precharging the word-lines and signal driving through the bit-lines of the cache now takes longer time for a high frequency processor. Results for bad/good prefetch reduction and IPC comparison are shown in Fig. 8, Fig. 9, and Fig. 10.

In Fig. 8, we present the number of bad prefetches for the six schemes. Similarly, the filters greatly filter bad prefetches. However, with a larger L1 cache, the filters are less aggressive in that a smaller number of bad prefetches was removed compared with the 8KB L1, due in part to reduced conflict and capacity misses for larger caches. On average, the reductions are 87.7 percent, 93.9 percent, and 93.3 percent for PA-based one-level, two-level, and gshare prefetch filters, respectively. For PC-based ones, the numbers are

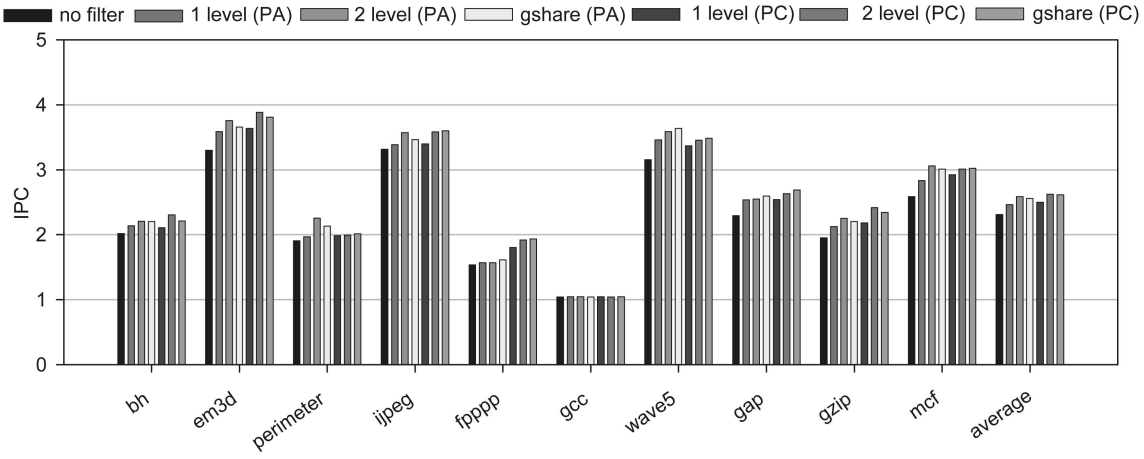

Fig. 10. Comparison of IPC for a 32KB data cache.

90.8 percent, 93.5 percent, and 92.8 percent for single level, two level, and gshare prefetch filters.

In the meantime, we observe a similar trend for the reduction on good prefetches. As shown in Fig. 9, more good prefetches are retained than the cases with 8K L1 cache. The reductions of good prefetches are 45.2 percent (one-level), 40.1 percent (two-level), 40.8 percent (gshare) for PA-based filters, and 41.3 percent (one level), 37.2 percent (two level), and 37.1 percent (gshare) for PC-based ones. Noticeably, the two-level filter and gshare filter perform slightly differently. The two-level filter tends to reduce more bad prefetches, but retain almost an equivalent number of good prefetches. Also, they are much closer, in contrast to the 8KB L1 case. In addition, the amount of traffic reduction also confirms our theory that a larger cache leads to a more effective filtering. For either PA or PC-based filters, roughly 50 percent prefetch bandwidth is reduced for the 32KB L1 cache compared with the 8KB L1 cache.

Fig. 10 shows IPC comparison. The one without filtering is shown as the first bar in the figure. All filters outperform the one without pollution filtering. As shown in the figure, *no filter* always delivers the worst performance among others. On average, the PA-based one-level filter achieves a 6.7 percent speedup and the PC-based one-level filter has an 8.2 percent speedup. The two-level filter is the best among the three, which gets a 11.9 percent speedup with PA-based indexing and a 13.5 percent with PC-based indexing. Finally, gshare is slightly worse than the two-level, with speedups of 10.6 percent (PA-based) and 13.2 percent (PC-based).

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 the 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. The one-level filter is worse than the other two since it reduces fewer bad prefetches but more good prefetches. However, the advantage is much less for a 32KB cache due to the higher prefetch traffic leading to more cache port competition for all filters. The difference between a two-level filter and a gshare filter is more subtle. With a larger cache, the two-level filter is very close to the gshare since more prefetches can be held in cache and this potentially lowers the bar for identifying bad/good prefetches. In other words, some prefetches, when having a longer lifetime in cache, become good prefetches. Overall, the performance difference between these two filters is minimal.

5.3 Impact of the History Table Size

In this section, different history table sizes are evaluated to quantify their impact to the overall performance. We only evaluate the PA-based single level filter since similar results have been observed for other types of filters. The size of the history table is varied from 1,024 entries (256B), 2,048 entries (512 B), 4,096 entries (1KB), 8,192 entries (2KB), up to 16,384 entries (4KB). All the experiments were performed using the default configuration except for the history table size.

Fig. 11 examines the bad/good prefetch ratios. In general, the bad/good prefetch ratio decreases by using larger history tables. It is obvious that more entries will alleviate aliasing in the history table. What was also found

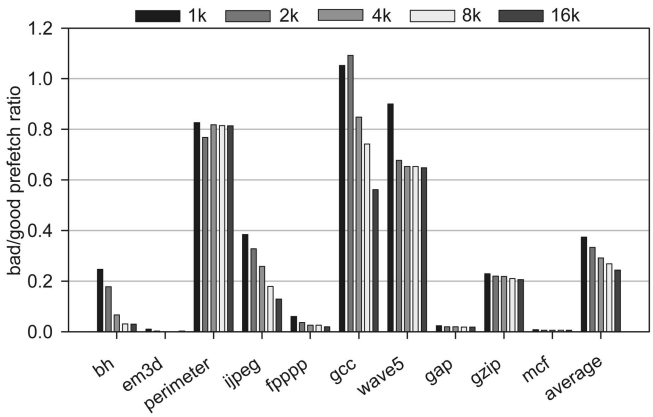

Fig. 11. Comparison of the bad/good prefetch ratio for different history table sizes.

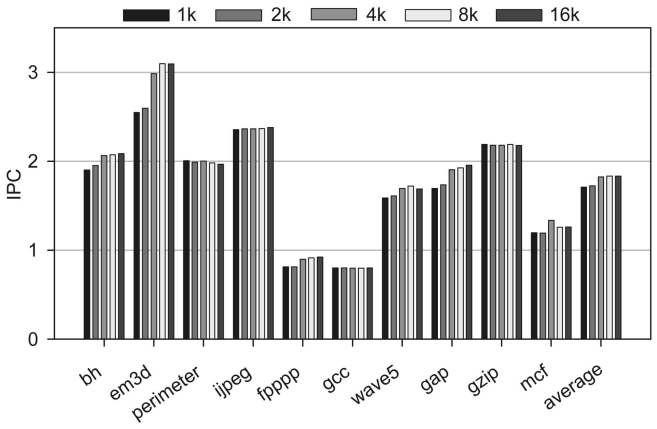

Fig. 12. Comparison of IPC for different history table sizes.

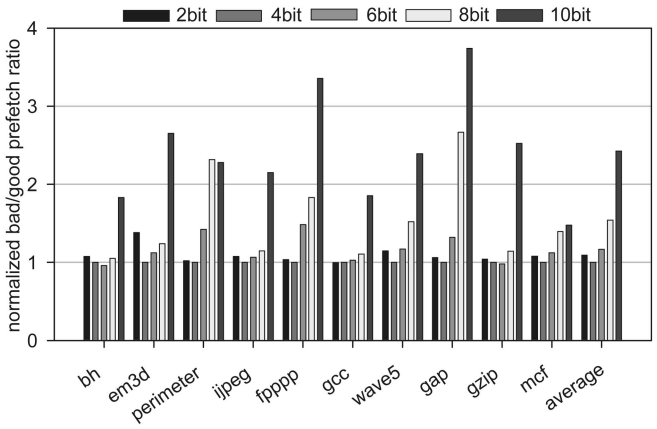

Fig. 13. Comparison of the normalized bad/good prefetch ratio for a two level prefetch filter with different BHR sizes.

is that the number of bad prefetches increases with larger history tables as well. For a few benchmarks such as gap, gzip and mcf, varying the history table size is almost insensitive to the reduction of bad prefetches. Nevertheless, with more good prefetches being preserved, the bad/good prefetch ratio is reduced. On average, the ratio is gradually reduced from 0.37, 0.33, 0.29, 0.27, down to 0.25 for 1K, 2K, 4K, 8K, and 16K entries. As shown in Fig. 11, a 1,024-entry history table is good enough to capture most of the reduction for bh, gcc, and wave5. The ratio is reduced by only 0.04 when the history table is enlarged from 4K entries to 16K entries; in other words, a small history table, e.g., 4K entries, is sufficient to obtain most of the benefits.

Fig. 12 presents the IPC comparison. For most programs, the IPC increases slightly as the history table grows. The harmonic means of the IPCs are 1.70 (1K-entry), 1.72 (2K-entry), 1.82 (4K-entry), and 1.83 (8K-entry and 16K-entry). The harmonic mean shows a 6 percent improvement from the 2K-entry to the 4K-entry. The transitions from the 1K-entry to the 2K-entry, 4K-entry to 8K-entry, and 8K-entry to 16K-entry only result in a less than 1 percent performance improvement. In short, the performance improvement for a history table size larger than 4,096 entries is limited. Moreover, short history tables (1K or 2K entries) can affect the performance to some extent. Hardware implementations should choose the size of the history table based on their cost budget. With 4K entries, the prefetch 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 the Branch History Register (BHR) Size

This section discusses the 2-level and gshare prefetch pollution filters where a branch history register (BHR) is used. We investigate how the size of the BHR affects the effectiveness of our prefetch filtering schemes. Without loss of generality, we only look at the PA-based prefetch filters and all the other configuration parameters are the same as in the default model. The size of the BHR ranges from 2 bit, 4 bit (default), 6 bit, 8 bit, and 10 bit. Notice that a 4,096-entry history table needs 12 bits at most.

Fig. 13 compares the bad/good prefetch ratios, which were normalized to the default with a 4-bit BHR. It appears that the bad/good prefetch ratio increases significantly as a wider BHR is used, especially when it reaches 8 bit and 10 bit. As mentioned earlier, once more branch history bits are counted, fewer bits are used for the address (or PC) of the prefetch. As a result, it could cause severe aliasing. Meanwhile, branch history bits do help to improve the effectiveness as can be seen from the earlier comparison between one level and two level filters. Thus, if the BHR size is too small, the ratio can be bad as well. Fig. 13 shows that a 4-bit BHR is the best for our current setting. The average values of the normalized bad/good prefetch ratios are 1.09 (2-bit), 1.16 (6-bit), 1.54 (8-bit), and 2.42 (10-bit). In other words, except for 2-bit and 4-bit, the other choices lead to much worse ratios.

We now study their impact on the IPC. In Fig. 14, the IPCs are normalized to the default case with a 4-bit BHR. Conceivably, a higher bad/good prefetch ratio leads to a bigger IPC degradation. On average, the normalized IPCs are 0.98 (2-bit), 0.97 (6-bit), 0.96 (8-bit), and 0.92 (10-bit), respectively. In other words, the slowdown could be as large as 8 percent when a 10-bit BHR is used.

Fig. 15 and Fig. 16 repeat the same experiments for the gshare filter. The normalized bad/good prefetch ratios in Fig. 15 imply that the BHR size is affected less radically for the gshare filter. Although the general trend remains the

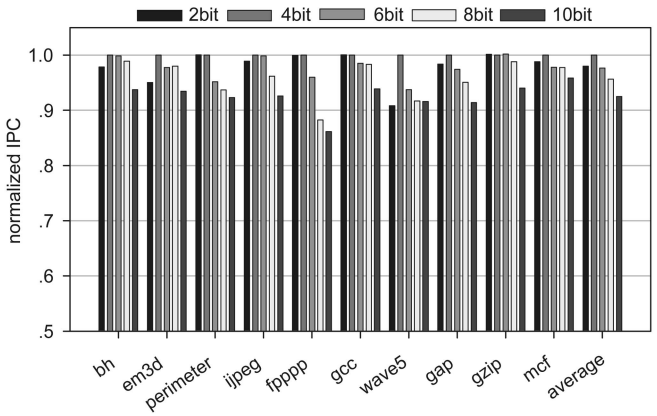

Fig. 14. Comparison of normalized IPC for a two level prefetch filter with different BHR sizes.

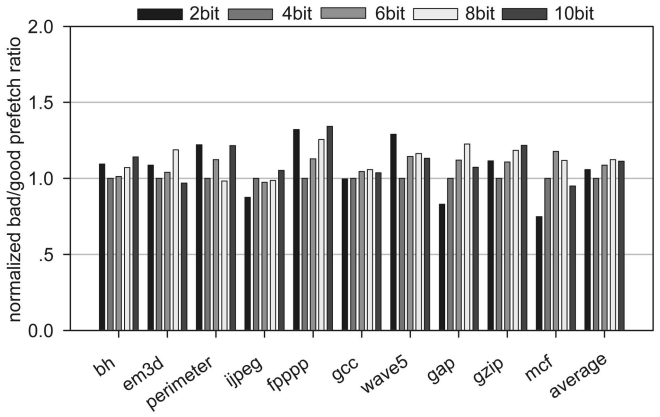

Fig. 15. Comparison of the normalized bad/good prefetch ratio for a gshare prefetch filter with different BHR sizes.

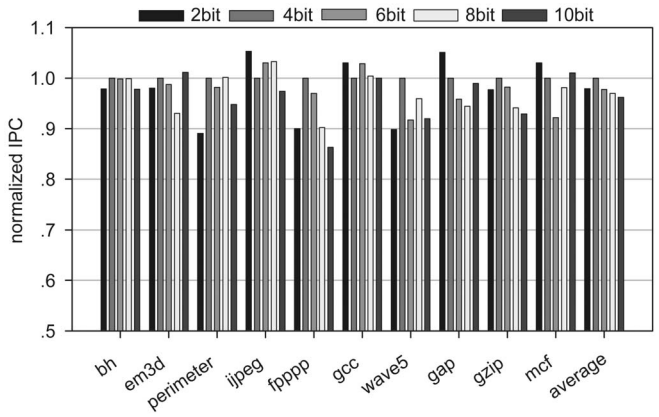

Fig. 16. Comparison of normalized IPC for a gshare prefetch filter with different BHR sizes.

same as in the case for a two level prefetch filter, it becomes less clear for some benchmarks such as perimeter and mcf. The average ratios are 1.06 (2-bit), 1.09 (6-bit), 1.12 (8-bit), and 1.11 (10-bit). It is perhaps because the gshare filter XORs BHR with the indexing address or PC such that the size of the address or PC is still included instead of being reduced like the two level filter. However, it also suggests that counting on more history bits could be detrimental since a few bits for the branch history serve well to distinguish prefetches, while more BHR bits actually bring about inaccuracy.

Fig. 16 shows the IPCs by varying the BHR size, corresponding to the bars in Fig. 15. Again, for individual benchmarks, a particular BHR size could be more suitable; however, the overall performance still suggests that a 4-bit BHR seems to be the best option under our configuration. On average, the normalized IPCs are 0.98 (2-bit), 0.98 (6-bit), 0.97 (8-bit), and 0.96 (10-bit), respectively. As shown, the slowdown for a 10-bit BHR is less severe compared with that of the two-level filter.

5.5 Impact of L1 Cache Ports

Next, the number of L1 cache ports is varied to see how it affects the bad/good prefetch ratio and the IPC. All experiments were performed under the default configuration with the PA-based pollution filter. The number of the L1 ports is increased gradually from three, four to five.[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 in our latency model. The L1 access latency is assumed to be two cycles for a 4-port 8KB cache and three cycles for a 5-port 8KB cache.

Fig. 17 shows the bad/good prefetch ratios for the three PA-based filters. We normalize the ratio to the value of the default machine model with three ports. 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 update must change the setting in the table). For the single level filter, a 4-port cache yields a 5 percent reduction over a 3-port one, then a 4 percent reduction from 4-port to 5-port. For a two-level filter, the reduction from 3-port to 4-port is 1.5 percent. Adding to 5-port gives an extra 1 percent reduction. Finally, the gshare obtains a 8.5 percent ratio reduction with one more port over 3-port and 2.7 percent reduction from 4-port to 5-port. In summary, the ratio reduction decreases as more ports are added. Meanwhile, it appears that more ports are especially helpful for the gshare filter.

Fig. 18 compares the IPC numbers. They are also normalized to the value of the default machine model with three ports. In general, the IPC increases with the port number increased. For a one level filter, it reflects a 3.0 percent speedup with four ports and another 0.8 percent with five ports, while, for a two level filter, 4-port gets a 2 percent speedup and 5-port gets a 0.5 percent speedup over 4-port. For gshare, it is a 2.9 percent speedup with 4-port and another 0.8 percent performance gain with five ports. In short, the speedup differences are quite small; in particular, from 4-port to 5-port, the performance is improved by less than 1 percent.

5.6 Comparison with a Dedicated Prefetch Buffer

In this section, we evaluate the impact of a dedicated prefetch buffer with our baseline machine model. All other configurations are kept intact. The prefetch buffer is suggested by [24] to reduce L1 cache pollution by storing prefetched data in a separate buffer. In our experiments, the prefetch buffer is fully associative with 16 entries. We only show the results using the single level prefetch filter since the other two prefetch filters yield similar results. We consider six scenarios:

1. no filter, no prefetch buffer,
2. no filtering and the prefetch buffer is used for reducing cache pollution,
3. one level PA-based prefetch filter without prefetch buffer,

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.

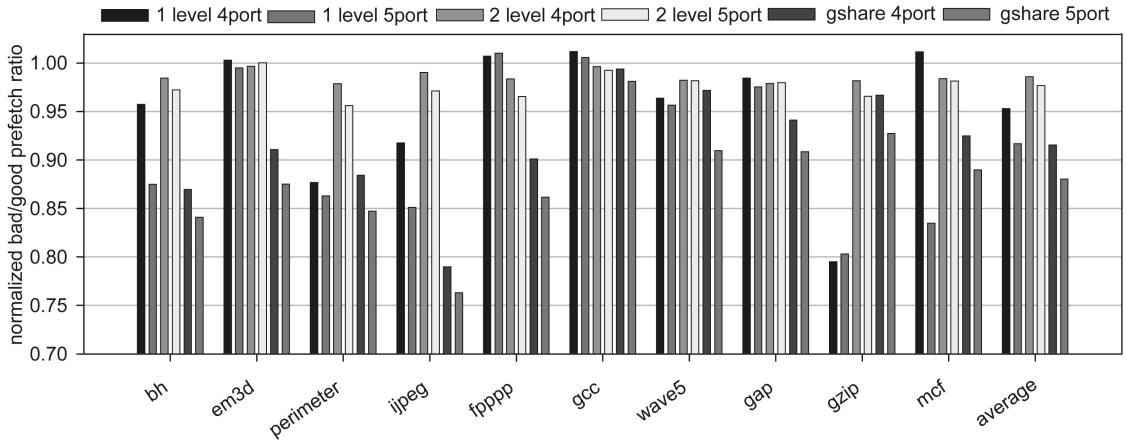

Fig. 17. Comparison of the normalized bad/good prefetch ratio for a different number of L1 ports.

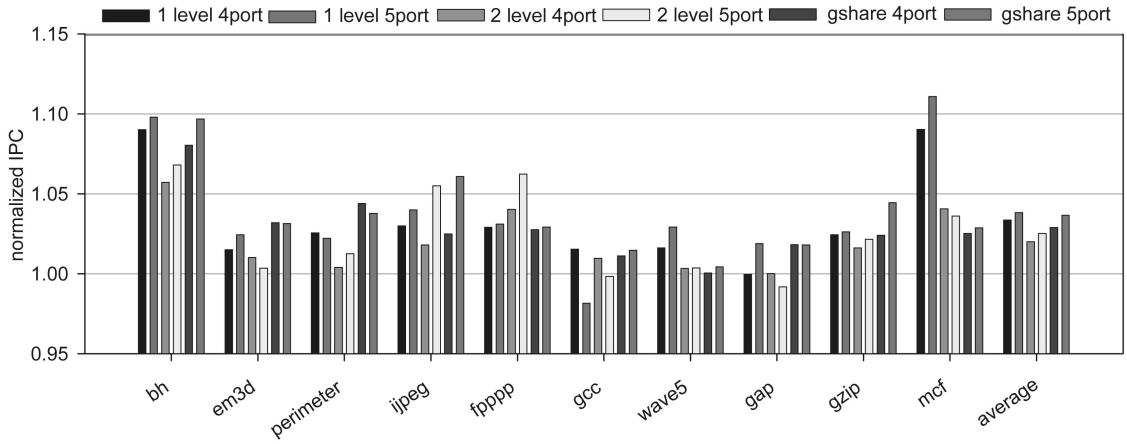

Fig. 18. Comparison of the normalized IPC for a different number of L1 ports.

4. one level PA-based prefetch filter with prefetch buffer, i.e., all prefetches go to the prefetch buffer,
5. one level PC-based prefetch filter without prefetch buffer, and
6. one level PC-based prefetch filter with prefetch buffer.

Fig. 19 shows the bad/good prefetch ratio for the six cases. Surprisingly, with only the prefetch buffer, the ratio is even worse. This is probably because aggressive prefetching can cause severe competition to the prefetch buffer; most prefetches are evicted quickly without actually being accessed. On the other hand, a prefetch buffer is fully associative, so its size cannot be too big, which becomes ineffective for aggressive prefetchers. Besides, unlike prefetch filtering, a prefetch buffer cannot reduce the prefetch traffic, which means the resource competition to the memory subsystem and cache ports remains unchanged.

Moreover, our experiments also show that, for aggressive prefetching, a small dedicated prefetch buffer is less effective if combined with our pollution filters because the prefetch buffer is not able to accommodate a a large number of prefetches and frequently causes early eviction of prefetched blocks. For most of the programs, adding a dedicated prefetch buffer degrades the effectiveness of the pollution filters. In Fig. 19, the bad/good prefetch ratio increases by about 0.05 for PA-based filter and 0.08 for PC-based filter.

In Fig. 20, the IPC numbers concur that no filtering but with prefetch buffer leads to the worst performance. It causes an 11.3 percent slowdown compared with the one without a prefetch buffer. Also, adding a dedicated prefetch buffer on top of our prefetch filters penalizes performance. On average, the IPCs are 1.68, 1.49, 1.82, 1.66, 1.83, and 1.66

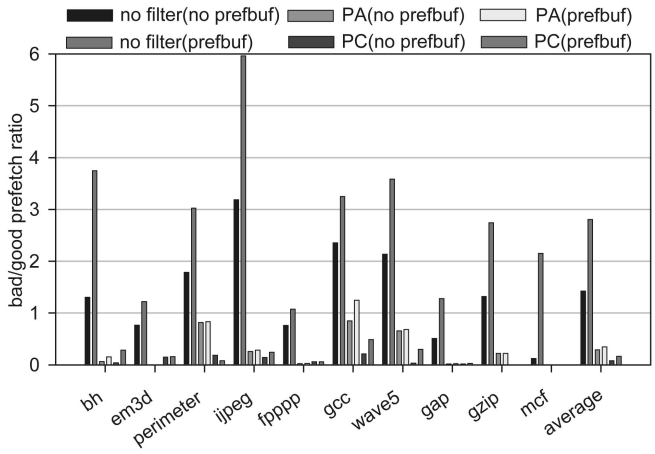

Fig. 19. Bad/good prefetch ratio comparison with prefetch buffer.

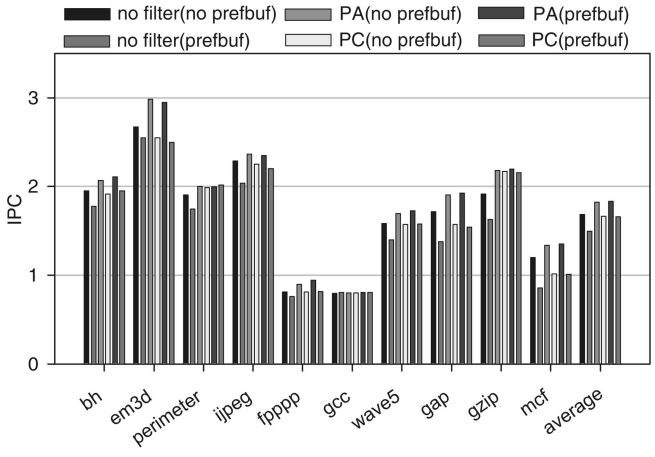

Fig. 20. IPC comparison with prefetch buffer.

for the six cases, respectively. In other words, the performance loss for prefetch buffer only (no filter) ranges from 9 percent to 19 percent, while adding the prefetch buffer on top of the PA-based and the PC-based filter causes a slowdown of 9 percent and 10 percent, respectively. Note that gcc is almost unaffected due to the small absolute numbers of both bad and good prefetches.

5.7 Sensitivity Study for Prefetch Accuracy

In this paper, we include three prefetchers, one software-based prefetcher and two hardware-based ones, i.e., NSP and SDP. Their prefetch accuracies (as a rough estimation, it can be derived from the bad/good prefetch ratios) are quite different. Here, we conduct a sensitivity study for the prefetch accuracy across the three prefetchers. The three prefetchers are separately examined such that we can evaluate the effectiveness of our prefetch filters with regard to the accuracy of each prefetcher. For brevity, we only look at a single level prefetch filter because the other two prefetch filters behave similarly in our evaluation. We first perform all prefetches without filtering, then apply the one level PA-based filter. The bad/good prefetch ratios are separately reported for the three prefetchers.

Fig. 21 illustrates six cases. Due to the wide range of values, the Y-axis is plotted on a log scale. Before delving into the details, it is necessary to report the distribution of prefetches across the three prefetchers. Roughly, when no filtering is applied, software prefetches account for 5 percent, while NSP and SDP consist of 80 percent and 15 percent. From Fig. 21, prefetch filtering only reduces the ratio slightly for a software-based prefetcher (6.2 percent on average). Conversely, the reductions for NSP and SDP are much higher. On average, NSP receives a 91.2 percent reduction and SDP receives 40.2 percent.

Overall, prefetch algorithms with higher accuracy typically lead to worse performance for pollution filtering. This is because a highly accurate prefetcher leaves little space for the prefetch filter to improve the bad/good prefetch ratio. For advanced features, our pollution filter can be made adaptive to start filtering when the prefetching becomes too aggressive (with low accuracy).

6 Conclusions

This paper proposes a number of hardware-based prefetch pollution filtering mechanisms that can significantly reduce the number of bad prefetches (over 90 percent) for processors implementing aggressive hardware and software prefetching. We evaluate three flavors of prefetch pollution filter design, including a single level, a two level, and a gshare style filter. In addition, we also evaluate a different indexing method based on cache line address (Per-Address-based) or program counter (PC-based). The major advantage of employing a cache pollution filter hardware is to enable an architecture to encompass several prefetching techniques, together with a dynamic filtering capability to maintain the performance edge. Excessive but ineffective prefetches causing performance degradation are filtered out by the hardware-based pollution filter. We quantified our approach through simulations and showed that our technique mitigates L1 data cache pollution while reducing the prefetch traffic that compete for the limited number of the L1 cache ports and finite bus bandwidth. As a result, the IPC, on average, could be improved by up to 16 percent for different L1 cache sizes with respect to a machine without any filtering mechanism. We also analyzed and demonstrated the hardware overheads to implement the filter. Basically, the history table size can be kept small (e.g., 1KB

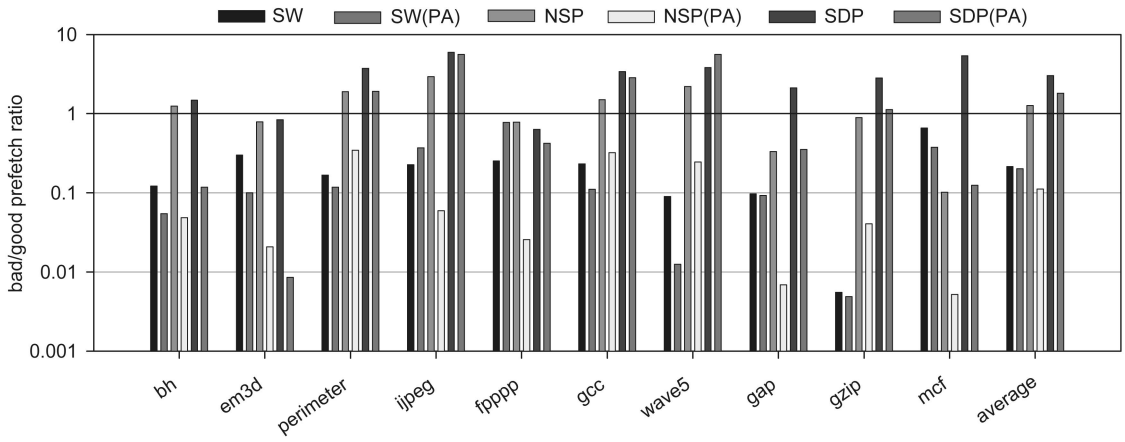

Fig. 21. Bad/good prefetch ratio comparison for prefetchers with different accuracy.

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. In addition, a number of sensitivity tests are experimented with to help understand our prefetch pollution filters thoroughly.

In conclusion, the prefetch pollution filter offers an effective hardware solution with affordable overheads that can improve performance by dynamically controlling the number of bad prefetches generated from overly aggressive prefetching schemes. Given that the L1 cache is getting smaller in the emerging deep submicron processors, our solution provides a means to utilize the limited on-chip resources more intelligently and effectively.

REFERENCES

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

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

[3] A.J. Smith, "Cache Memories," *Computing Surveys,* vol. 14, no. 3, 1982.

[4] F. Dahlgren, M. Dubois, and P. Stenstrom, "Fixed and Adaptive Sequential Prefetching in Shared-Memory Multiprocessors," *Proc. 1993 Int'l Conf. Parallel Processing,* 1993.

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

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

[7] T.-F. Chen and J.-L. Baer, "Effective Hardware-Based Data Prefetching for High Performance Processors," *IEEE Trans. Computers,* vol. 44, no. 5, May 1995.

[8] M.J. Charney and A.P. Reeves, "Generalized Correlation Based Hardware Prefetching," Technical Report EE-CEG-95-1, Cornell Univ., 1995.

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

[10] Y. Solihin, J. Lee, and J. Torrellas, "Using a User-Level Memory Thread for Correlation Prefetching," *Proc. 29th Ann. Int'l Symp. Computer Architecture,* pp. 171-182, 2002.

[11] K.J. Nesbit, A.S. Dhodapkar, and J.E. Smith, "AC/DC: An Adaptive Data Cache Prefetcher," *Proc. Int'l Conf. Parallel Architectures and Compiler Techniques,* pp. 135-145, 2004.

[12] K.J. Nesbit and J.E. Smith, "Data Cache Prefetching Using a Global History Buffer," *Proc. Int'l Symp. High-Performance Computer Architecture,* pp. 96-105, 2004.

[13] D.G. Perez, G. Mouchard, and O. Temam, "MicroLib: A Case for the Quantitative Comparison of Micro-Architecture Mechanisms," *Proc. 37th Int'l Symp. Microarchitecture,* pp. 43-54, 2004.

[14] A.K. Porterfield, "Software Methods for Improvement of Cache Performance on Supercomputer Application," PhD dissertation, Rice Univ., 1989.

[15] *IA-32 Intel Architecture Software Developer's Manual Volume 2B,* http://developer.intel.com/design/Pentium4/manuals/25366714.pdf, Intel Corporation, 2004.

[16] *Alpha Architecture Handbook.* Compaq Computer Corp., Oct. 1998.

[17] C.-K. Luk and T.C. Mowry, "Automatic Compiler-Inserted Prefetching for Pointer-Based Applications," *IEEE Trans. Computers,* vol. 48, no. 2, Feb. 1999.

[18] V. Srinivasan, E.S. Davidson, and G.S. Tyson, "A Prefetch Taxonomy," *IEEE Trans. Computers,* vol. 53, no. 2, pp. 126-140, Feb. 2004.

[19] T.R. Puzak, A. Hartstein, P.G. Emma, and V. Srinivasan, "When Prefetching Improves/Degrades Performance," *Proc. Second Conf. Computing Frontiers (CF '05),* pp. 342-352, 2005.

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

[21] A. Lai, C. Fide, and B. Falsafi, "Dead-Block Prediction and Dead-Block Correlating Prefetchers," *Proc. 28th Int'l Symp. Computer Architecture,* 2001.

[22] O. Mutlu, H. Kim, D.N. Armstrong, and Y.N. Patt, "Cache Filtering Techniques to Reduce the Negative Impact of Useless Speculative Memory References on Processor Performance," *Proc. 16th Symp. Computer Architecture and High Performance Computing,* 2004.

[23] Z. Wang, D. Burger, K.S. McKinley, S.K. Reinhardt, and C.C. Weems, "Guided Region Prefetching: A Cooperative Hardware/Software Approach," *Proc. 30th Ann. Int'l Symp. Computer Architecture,* pp. 388-398, 2003.

[24] 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," *Proc. Int'l Symp. Microarchitecture,* 1991.

[25] W.-F. Lin, S.K. Reinhardt, D. Burger, and T.R. Puzak, "Filtering Superfluous Prefetches Using Density Vectors," *Proc. 19th Int'l Conf. Computer Design,* pp. 124-132, 2001.

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

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

[28] T.-Y. Yeh and Y.N. Patt, "Two-Level Adaptive Training Branch Prediction," *Proc. 24th Int'l Symp. Microarchitecture,* 1991.

[29] T.-Y. Yeh and Y.N. Patt, "Alternative Implementations of Two-Level Adaptive Training Branch Prediction," *Proc. 19th Int'l Symp. Computer Architecture,* 1992.

[30] T.-Y. Yeh and Y.N. Patt, "A Comparison of Dynamic Branch Predictors that Use Two Levels of Branch History," *Proc. 20th Int'l Symp. Computer Architecture,* 1993.

[31] S. McFarling, "Combining Branch Predictors," WRL Technical Note TN-36, Digital Equipment Corp., June 1993.

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

[33] M. Carlisle, "Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines," PhD dissertation, Dept. of Computer Science, Princeton Univ., 1996.

Xiaotong Zhuang received the PhD degree in computer science from the Georgia Institute of Technology in 2006. He received the BS and MS degrees from Shanghai Jiaotong University. He is currently a research staff member at the IBM T.J. Watson Research Center, Yorktown Heights, New York. His main areas of interest are compiler optimization and architectural design for embedded and security systems, and computer architecture. He is also interested in parallel and distributed systems. During his PhD study, he conducted research on several emerging areas such as power-aware computing and network processors. As a result of this endeavor, he has published 30 papers in conferences and journals.

Hsien-Hsin S. Lee (M'96) received the PhD degree in computer science and engineering from the University of Michigan at Ann Arbor. He is an assistant professor in the School of Electrical and Computer Engineering at the Georgia Institute of Technology. His research interests include computer architecture, low power circuits, information security, and 3D ICs. Prior to joining academia, he was a senior computer architect at Intel Corporation and, later, the architecture manager of the StarCore DSP Center at Agere systems. Dr. Lee's doctoral thesis was awarded the Horace H. Rackham School Distinguished Dissertation Award at the University of Michigan. He has coauthored three papers that won Best Paper Awards at MICRO-33, CASES-04, and IBM PAC^2. More recently, Dr. Lee received the US Department of Energy Early CAREER PI Award and was named the recipient of the 2006 ECE Outstanding Junior Faculty Member Award at Georgia Tech. He holds four US patents and is a member of Tau Beta Pi, the ACM, the IEEE, and the IEEE Computer Society.