Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers

Santhosh Srinath†‡ Onur Mutlu§ Hyesoon Kim‡ Yale N. Patt‡

†Microsoft
ssri@microsoft.com

§Microsoft Research
onur@microsoft.com

‡Department of Electrical and Computer Engineering
The University of Texas at Austin
{santhosh, hyesoon, patt}@ece.utexas.edu

Abstract

High performance processors employ hardware data prefetching to reduce the negative performance impact of large main memory latencies. While prefetching improves performance substantially on many programs, it can significantly reduce performance on others. Also, prefetching can significantly increase memory bandwidth requirements. This paper proposes a mechanism that incorporates dynamic feedback into the design of the prefetcher to increase the performance improvement provided by prefetching as well as to reduce the negative performance and bandwidth impact of prefetching. Our mechanism estimates prefetcher accuracy, prefetcher timeliness, and prefetcher-caused cache pollution to adjust the aggressiveness of the data prefetcher dynamically. We introduce a new method to track cache pollution caused by the prefetcher at run-time. We also introduce a mechanism that dynamically decides where in the LRU stack to insert the prefetched blocks in the cache based on the cache pollution caused by the prefetcher.

Using the proposed dynamic mechanism improves average performance by 6.5% on 17 memory-intensive benchmarks in the SPEC CPU2000 suite compared to the best-performing conventional stream-based data prefetcher configuration, while it consumes 18.7% less memory bandwidth. Compared to a conventional stream-based data prefetcher configuration that consumes similar amount of memory bandwidth, feedback directed prefetching provides 13.6% higher performance. Our results show that feedback-directed prefetching eliminates the large negative performance impact incurred on some benchmarks due to prefetching, and it is applicable to stream-based prefetchers, global-history-buffer based delta correlation prefetchers, and PC-based stride prefetchers.

1. Introduction

Hardware data prefetching works by predicting the memory access pattern of the program and speculatively issuing prefetch requests to the predicted memory addresses before the program accesses those addresses. Prefetching has the potential to improve performance if the memory access pattern is correctly predicted and the prefetch requests are initiated early enough before the program accesses the predicted memory addresses. Since the memory latencies faced by today's processors are on the order of hundreds of processor clock cycles, accurate and timely prefetching of data from main memory to the processor caches can lead to significant performance gains by hiding the latency of memory accesses. On the other hand, prefetching can negatively impact the performance and energy consumption of a processor due to two major reasons, especially if the predicted memory addresses are not accurate:

- First, prefetching can increase the contention for the available memory bandwidth. Additional bandwidth contention caused by prefetches can lead to increased DRAM bank conflicts, DRAM page conflicts, memory bus contention, and queueing delays. This can significantly reduce performance if it results in delaying demand (i.e. load/store) requests. Moreover, inaccurate prefetches increase the energy consumption of the processor because they result in unnecessary memory accesses (i.e. waste memory/bus bandwidth). Bandwidth contention due to prefetching will become more significant as more and more processing cores are integrated onto the same die in chip multiprocessors, effectively reducing the memory bandwidth available to each core. Therefore, techniques that reduce the memory bandwidth consumption of hardware prefetchers while maintaining their performance improvement will become more desirable and valuable in future processors [22].
- Second, prefetching can cause cache pollution if the prefetched data displaces cache blocks that will later be needed by load/store instructions in the program.[1] Cache pollution due to prefetching might not only reduce performance but also waste memory bandwidth by resulting in additional cache misses.

  Furthermore, prefetcher-caused cache pollution generates new cache misses and those generated cache misses can in turn generate new prefetch requests. Hence, the prefetcher itself is a *positive feedback system* that can be unstable in terms of both performance and bandwidth consumption. Therefore, we would like to augment the prefetcher with a *negative feedback system* to make it stable.

Figure 1 compares the performance of varying the aggressiveness of a stream-based hardware data prefetcher from *No prefetching* to *Very Aggressive prefetching* on 17 memory-intensive benchmarks in the SPEC CPU2000 benchmark suite.[2] Aggressive prefetching improves IPC performance by 84% on average[3] and by over 800% for some benchmarks (e.g. mgrid) compared to no prefetching. Furthermore, aggressive prefetch-

[1] Note that this is a problem only in designs where prefetch requests bring data into processor caches rather than into separate prefetch buffers [13, 11]. In many current processors (e.g. Intel Pentium 4 [6] or IBM POWER4 [24]), prefetch requests bring data into the processor caches. This reduces the complexity of the memory system by eliminating the need to design a separate prefetch buffer. It also makes the large L2 cache space available to prefetch requests, enabling the prefetched blocks and demand-fetched blocks to share the available cache memory dynamically rather than statically partitioning the storage space for demand-fetched and prefetched data.

[2] Aggressiveness of the prefetcher is determined by how far the prefetcher stays ahead of the demand access stream of the program as well as how many prefetch requests are generated, as shown in Table 1 and Section 2.1.

[3] Similar results were reported by [8] and [18]. All average IPC results in this paper are computed as geometric mean of the IPC's of the benchmarks.

1-4244-0805-9/07/$25.00 ©2007 IEEE

ing on average performs better than conservative and middle-of-the-road prefetching. Unfortunately, aggressive prefetching significantly reduces performance on some benchmarks. For example, an aggressive prefetcher reduces the IPC performance of ammp by 48% and applu by 29% compared to no prefetching. Hence, blindly increasing the aggressiveness of the hardware prefetcher can drastically reduce performance on several applications even though it improves the average performance of a processor. Since aggressive prefetching significantly degrades performance on some benchmarks, many modern processors employ relatively conservative prefetching mechanisms where the prefetcher does not stay far ahead of the demand access stream of the program [6, 24].

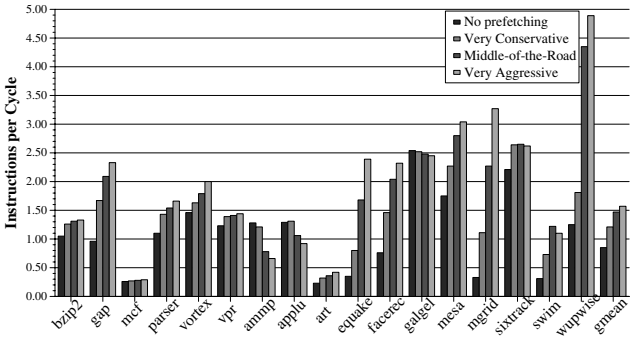

Figure 1. Performance vs. aggressiveness of the prefetcher

The goal of this paper is to reduce the negative performance and bandwidth impact of aggressive prefetching while preserving the large performance benefits provided by it. To achieve this goal, we propose simple and implementable mechanisms that dynamically adjust the aggressiveness of the hardware prefetcher as well as the location in the processor cache where prefetched data is inserted.

The proposed mechanisms estimate the effectiveness of the prefetcher by monitoring the accuracy and timeliness of the prefetch requests as well as the cache pollution caused by the prefetch requests. We describe simple hardware implementations to estimate accuracy, timeliness, and cache pollution. Based on the run-time estimation of these three metrics, the aggressiveness of the hardware prefetcher is decreased or increased dynamically. Also, based on the run-time estimation of the cache pollution caused by the prefetcher, the proposed mechanism dynamically decides where to insert the prefetched blocks in the processor cache's LRU stack.

Our results show that using the proposed dynamic feedback mechanisms improve the average performance of 17 memory-intensive benchmarks in the SPEC CPU2000 suite by 6.5% compared to the best-performing conventional stream-based prefetcher configuration. With the proposed mechanism, the negative performance impact incurred on some benchmarks due to stream-based prefetching is completely eliminated. Furthermore, the proposed mechanism consumes 18.7% less memory bandwidth than the best-performing stream-based prefetcher configuration. Compared to a conventional stream-based prefetcher configuration that consumes similar amount of memory bandwidth, feedback directed prefetching provides 13.6% higher performance. We also show that the dynamic feedback mechanism works similarly well when implemented to dynamically adjust the aggressiveness of a global-history-buffer (GHB) based delta correlation prefetcher [10] or a PC-based stride prefetcher [1]. Compared to a conventional GHB-based delta correlation prefetcher configuration that consumes similar amount of memory bandwidth, feedback directed prefetching provides 9.9% higher performance. The proposed mechanism provides these benefits with a modest hardware storage cost of 2.54 KB and without significantly increasing hardware complexity. On the remaining 9 SPEC CPU2000 benchmarks, the proposed dynamic feedback mechanism performs as well as the best-performing conventional stream prefetcher configuration for those 9 benchmarks.

2. Background and Motivation

2.1. Stream Prefetcher Design

The stream prefetcher we model is based on the stream prefetcher in the IBM POWER4 processor [24] and more details on the implementation of stream-based prefetching can be found in [11, 19, 24]. The modeled prefetcher brings cache blocks from the main memory to the last-level cache, which is the second-level (L2) cache in our baseline processor.

The stream prefetcher is able to keep track of multiple different access streams. For each tracked access stream, a stream tracking entry is created in the stream prefetcher. Each tracking entry can be in one of four different states:

1. Invalid: The tracking entry is not allocated a stream to keep track of. Initially, all tracking entries are in this state.
2. Allocated: A demand (i.e. load/store) L2 miss allocates a tracking entry if the demand miss does not find any existing tracking entry for its cache-block address.
3. Training: The prefetcher trains the direction (ascending or descending) of the stream based on the next two L2 misses that occur +/- 16 cache blocks from the first miss.[4] If the next two accesses in the stream are to ascending (descending) addresses, the direction of the tracking entry is set to 1 (0) and the entry transitions to *Monitor and Request* state.
4. Monitor and Request: The tracking entry monitors the accesses to a memory region from a *start pointer* (address A) to an *end pointer* (address P). The maximum distance between the start pointer and the end pointer is determined by *Prefetch Distance*, which indicates how far ahead of the demand access stream the prefetcher can send requests. If there is a demand L2 cache access to a cache block in the monitored memory region, the prefetcher requests cache blocks [P+1, ..., P+N] as prefetch requests (assuming the direction of the tracking entry is set to 1). N is called the *Prefetch Degree*. After sending the prefetch requests, the tracking entry starts monitoring the memory region between addresses A+N to P+N (i.e. effectively it moves the tracked memory region by N cache blocks).[5]

[4] Note that all addresses tracked by the prefetcher are cache-block addresses.

[5] Right after a tracking entry is trained, the prefetcher sets the start pointer to the the first L2 miss address that allocated the tracking entry and the end pointer to the last L2 miss address that determined the direction of the entry plus an initial start-up distance. Until the monitored memory region's size becomes the same as the *Prefetch Distance* (in terms of cache blocks), the tracking entry increments only the end pointer by the *Prefetch Degree* when prefetches are issued (i.e. the end pointer points to the last address requested as a prefetch and the start pointer points to the L2 miss address that allocated the tracking entry). After the monitored memory region's size becomes the same as *Prefetch Distance*, both the start pointer and the end pointer are incremented by *Prefetch Degree (N)* when prefetches are issued. This way, the prefetcher is able to send prefetch requests that are *Prefetch Distance* ahead of the demand access stream.

Prefetch Distance and *Prefetch Degree* determine the aggressiveness of the prefetcher. In a traditional prefetcher configuration, the values of *Prefetch Distance* and *Prefetch Degree* are fixed at the design time of the processor. In the feedback directed mechanism we propose, the processor dynamically changes *Prefetch Distance* and *Prefetch Degree* to adjust the aggressiveness of the prefetcher.

2.2. Metrics of Prefetcher Effectiveness

We use three metrics (*Prefetch Accuracy*, *Prefetch Lateness*, and *Prefetcher-Generated Cache Pollution*) as feedback inputs to feedback directed prefetchers. In this section, we define the metrics and describe the relationship between the metrics and the performance provided by a conventional prefetcher. We evaluate four configurations: *No prefetching*, *Very Conservative prefetching* (distance=4, degree=1), *Middle-of-the-Road prefetching* (distance=16, degree=2), and *Very Aggressive prefetching* (distance=64, degree=4).

2.2.1. Prefetch Accuracy: Prefetch accuracy is a measure of how accurately the prefetcher can predict the memory addresses that will be accessed by the program. It is defined as

$$Prefetch\ Accuracy = \frac{Number\ of\ Useful\ Prefetches}{Number\ of\ Prefetches\ Sent\ To\ Memory}$$

where *Number of Useful Prefetches* is the number of prefetched cache blocks that are used by demand requests while they are resident in the L2 cache.

Figure 2 shows the IPC of the four configurations along with prefetch accuracy measured over the entire run of each benchmark. The results show that in benchmarks where prefetch accuracy is less than 40% (applu, galgel, and ammp), employing the stream prefetcher always degrades performance compared to no prefetching. In all benchmarks where prefetch accuracy exceeds 40% (except mcf), using the stream prefetcher significantly improves performance over no prefetching. For benchmarks with high prefetch accuracy, performance increases as the aggressiveness of the prefetcher is increased. Hence, the performance improvement provided by increasing the aggressiveness of the prefetcher is correlated with prefetch accuracy.

2.2.2. Prefetch Lateness: Prefetch lateness is a measure of how timely the prefetch requests generated by the prefetcher are with respect to the demand accesses that need the prefetched data. A prefetch is defined to be *late* if the prefetched data has not yet returned from main memory by the time a load or store instruction requests the prefetched data. Even though the prefetch requests are accurate, a prefetcher might not be able to improve performance if the prefetch requests are very late. We define prefetch lateness as:

$$Prefetch\ Lateness = \frac{Number\ of\ Late\ Prefetches}{Number\ of\ Useful\ Prefetches}$$

Figure 3 shows the IPC of the four configurations along with prefetch lateness measured over the entire run of each program. These results explain why prefetching does not provide significant performance benefit on mcf, even though the prefetch accuracy is close to 100%. More than 90% of the useful prefetch requests are late in mcf. In general, prefetch lateness decreases as the prefetcher becomes more aggressive. For example, in vortex, prefetch lateness decreases from 70% to 22% when a very aggresive prefetcher is used instead of a very conservative one. Aggressive prefetching reduces the lateness of prefetches because an aggressive prefetcher generates prefetch requests earlier than a conservative one would.

2.2.3. Prefetcher-Generated Cache Pollution: Prefetcher-generated cache pollution is a measure of the disturbance caused by prefetched data in the L2 cache. It is defined as:

$$Prefetcher\ Generated\ Cache\ Pollution = \frac{Number\ of\ Demand\ Misses\ Caused\ By\ the\ Prefetcher}{Number\ of\ Demand\ Misses}$$

A demand miss is defined to be caused by the prefetcher if it would not have occurred had the prefetcher not been present. If the prefetcher-generated cache pollution is high, the performance of the processor can degrade because useful data in the cache could be evicted by prefetched data. Furthermore, high cache pollution can also result in higher memory bandwidth consumption by requiring the re-fetch of the displaced data from main memory.

3. Feedback Directed Prefetching (FDP)

FDP dynamically adapts the aggressiveness of the prefetcher based on the accuracy, lateness, and pollution metrics defined in the previous section. This section describes hardware mechanisms that track these metrics and the FDP mechanism.

3.1. Collecting Feedback Information

3.1.1. Prefetch Accuracy: To track the usefulness of prefetch requests, we add a bit (*pref-bit*), to each tag-store entry in the L2 cache.[6] When a prefetched block is inserted into the cache, the pref-bit associated with that block is set. Prefetcher accuracy is tracked using two hardware counters. The first counter, *pref-total*, tracks the number of prefetches sent to memory. The second counter, *used-total*, tracks the number of useful prefetches. When a prefetch request is sent to memory, pref-total is incremented. When an L2 cache block that has the pref-bit set is accessed by a demand request, the pref-bit is reset and used-total is incremented. The accuracy of the prefetcher is computed by taking the ratio of used-total to pref-total.

3.1.2. Prefetch Lateness: Miss Status Holding Register (MSHR) [12] is a hardware structure that keeps track of all in-flight memory requests. Before allocating an MSHR entry for a request, the MSHR checks if the requested cache block is being serviced by an earlier memory request. Each entry in the L2 cache MSHR has a bit, called the *pref-bit*, which indicates that the memory request was generated by the prefetcher. A prefetch request is late if a demand request for the prefetched address is generated while the prefetch request is in the MSHR waiting for main memory. We use a hardware counter, *late-total*, to keep track of such late prefetches. If a demand request hits an MSHR entry that has its pref-bit set, the late-total counter is incremented, and the pref-bit associated with that MSHR entry is reset. The lateness metric is computed by taking the ratio of late-total to used-total.

[6] Note that several proposed prefetching implementations, such as tagged next-sequential prefetching [5, 21] already employ *pref-bit*s in the cache.

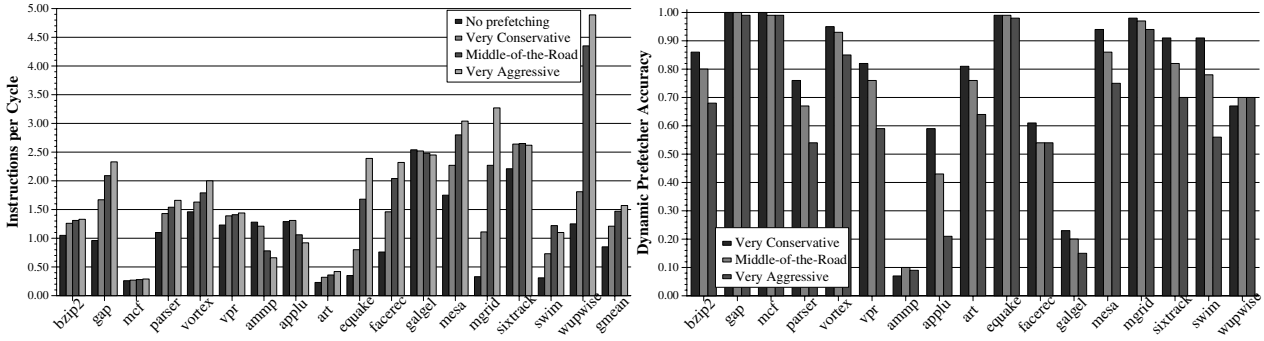

Figure 2. IPC performance (left) and prefetch accuracy (right) with different aggressiveness configurations

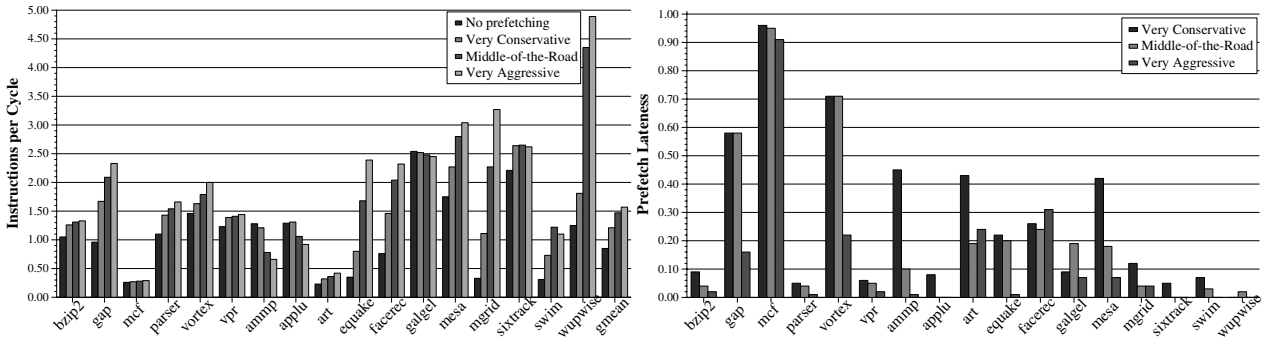

Figure 3. IPC performance (left) and prefetch lateness (right) with different aggressiveness configurations

3.1.3. Prefetcher-Generated Cache Pollution: To track the number of demand misses caused by the prefetcher, the processor needs to store information about all demand-fetched L2 cache blocks dislodged by the prefetcher. However, such a mechanism is impractical as it incurs a heavy overhead in terms of both hardware and complexity. We use the Bloom filter concept [2, 20] to provide a simple cost-effective hardware mechanism that can approximate the number of demand misses caused by the prefetcher.

Figure 4 shows the filter that is used to approximate the number of L2 demand misses caused by the prefetcher. The filter consists of a bit-vector, which is indexed with the output of the exclusive-or operation of the lower and higher order bits of the cache block address. When a block that was brought into the cache due to a demand miss is evicted from the cache due to a prefetch request, the filter is accessed with the address of the evicted cache block and the corresponding bit in the filter is set (indicating that the evicted cache block was evicted due to a prefetch request). When a prefetch request is serviced from memory, the pollution filter is accessed with the cache-block address of the prefetch request and the corresponding bit in the filter is reset, indicating that the block was inserted into the cache. When a demand access misses in the cache, the filter is accessed using the cache-block address of the demand request. If the corresponding bit in the filter is set, it is an indication that the demand miss was caused by the prefetcher. In such cases, the hardware counter, *pollution-total*, that keeps track of the total number of demand misses caused by the prefetcher is incremented. Another counter, *demand-total*, keeps track of the total number of demand misses generated by the processor and is incremented for each demand miss. Cache pollution caused by the prefetcher can be computed by taking the ratio of pollution-total to demand-total. We use a 4096-entry bit vector in our experiments.

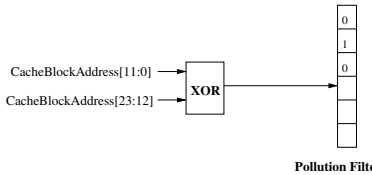

Figure 4. Filter to estimate prefetcher-generated cache pollution

3.2. Sampling-based Feedback Collection

To adapt to the time-varying memory phase behavior of a program, we use interval-based sampling for all counters described in Section 3.1. Program execution time is divided into intervals and the value of each counter is computed as:

$$CounterValue = \frac{1}{2}CounterValueAtTheBeginningOfTheInterval + \frac{1}{2}CounterValueDuringInterval \quad (1)$$

The *CounterValueDuringInterval* is reset at the end of each sampling interval. The above equation used to update the counters (Equation 1) gives more weight to the behavior of the program in the most recent interval while taking into account the behavior in all previous intervals. Our mechanism defines the length of an interval based on the number of useful cache blocks evicted from the L2 cache.[7] A hardware counter, *eviction-count*, keeps track of the number of blocks evicted from the L2 cache. When the value of the counter exceeds a statically-set threshold $T_{interval}$, the interval ends. At the end of an interval, all counters described in Section 3.1 are updated according to Equation 1. The updated counter values are then

[7] There are other ways to define the length of an interval, e.g. based on the number of instructions executed. We use the number of useful cache blocks evicted to define an interval because this metric provides a more accurate view of the memory behavior of a program than the number of instructions executed.

used to compute the three metrics: accuracy, lateness, and pollution. These metrics are used to adjust the prefetcher behavior for the next interval. The eviction-count register is reset and a new interval begins. In our experiments, we use a value of 8192 (half the number of blocks in the L2 cache) for $T_{interval}$.

3.3. Dynamically Adjusting Prefetcher Behavior

At the end of each sampling interval, the computed values of the accuracy, lateness, and pollution metrics are used to dynamically adjust prefetcher behavior. Prefetcher behavior is adjusted in two ways: (1) by adjusting the aggressiveness of the prefetching mechanism, (2) by adjusting the location in the L2 cache's LRU stack where prefetched blocks are inserted.[8]

3.3.1. Adjusting Prefetcher Aggressiveness: The aggressiveness of the prefetcher directly determines the potential for benefit as well as harm that is caused by the prefetcher. By dynamically adapting this parameter based on the collected feedback information, the processor can not only achieve the performance benefits of aggressive prefetching during program phases where aggressive prefetching performs well but also eliminate the negative performance and bandwidth impact of aggressive prefetching during phases where aggressive prefetching performs poorly.

As shown in Table 1, our baseline stream prefetcher has five different configurations ranging from *Very Conservative* to *Very Aggressive*. The aggressiveness of the stream prefetcher is determined by the *Dynamic Configuration Counter*, a 3-bit saturating counter that saturates at values 1 and 5. The initial value of the *Dynamic Configuration Counter* is set to 3, indicating Middle-of-the-Road aggressiveness.

Dyn. Config. Counter	*Aggressiveness*	*Pref. Distance*	*Pref. Degree*
1	Very Conservative	4	1
2	Conservative	8	1
3	Middle-of-the-Road	16	2
4	Aggressive	32	4
5	Very Aggressive	64	4

Table 1. Stream prefetcher configurations

At the end of each sampling interval, the value of the *Dynamic Configuration Counter* is updated based on the computed values of the accuracy, lateness, and pollution metrics. The computed accuracy is compared to two thresholds (A_{high} and A_{low}) and is classified as high, medium or low. Similarly, the computed lateness is compared to a single threshold ($T_{lateness}$) and is classified as either late or not-late. Finally, the computed pollution is compared to a single threshold ($T_{pollution}$) and is classified as high (polluting) or low (not-polluting). We use static thresholds in our mechanisms. The effectiveness of our mechanism can be improved by dynamically tuning the values of these thresholds and/or using more thresholds, but such optimization is out of the scope of this paper. In Section 5, we show that even with untuned threshold values, FDP can significantly improve performance and reduce memory bandwidth consumption on different data prefetchers.

Table 2 shows in detail how the estimated values of the three metrics are used to adjust the dynamic configuration of the prefetcher. We determined the counter update choice for each case empirically. If the prefetches are causing pollution (all even-numbered cases), the prefetcher is adjusted to be less aggressive to reduce cache pollution and to save memory bandwidth (except in Case 2 when the accuracy is high and prefetches are late – we do increase aggressiveness in this case to gain more benefit from highly-accurate prefetches). If the prefetches are late but not polluting (Cases 1, 5, 9), the aggressiveness is increased to increase timeliness unless the prefetch accuracy is low (Case 9 – we reduce aggressiveness in this case because a large fraction of inaccurate prefetches will waste memory bandwidth). If the prefetches are neither late nor polluting (Cases 3, 7, 11), the aggressiveness is left unchanged.

3.3.2. Adjusting Cache Insertion Policy of Prefetched Blocks: FDP also adjusts the location in which a prefetched block is inserted in the LRU-stack of the corresponding cache set based on the observed behavior of the prefetcher. In many cache implementations, prefetched cache blocks are simply inserted into the Most-Recently-Used (MRU) position in the LRU-stack, since such an insertion policy does not require any changes to the cache implementation. Inserting the prefetched blocks into the MRU position can allow the prefetcher to be more aggressive and request data long before its use because this insertion policy allows the useful prefetched blocks to stay longer in the cache. However, if the prefetched cache blocks create cache pollution, having a different cache insertion policy for prefetched cache blocks can help reduce the cache pollution caused by the prefetcher. A prefetched block that is not useful creates more pollution in the cache if it is inserted into the MRU position rather than a less recently used position because it stays in the cache for a longer time period, occupying cache space that could otherwise be allocated to a useful demand-fetched cache block. Therefore, if the prefetch requests are causing cache pollution, it would be desirable to reduce this pollution by changing the location in the LRU stack in which prefetched blocks are inserted.

We propose a simple heuristic that decides where in the LRU stack of the L2 cache set a prefetched cache block is inserted based on the estimated prefetcher-generated cache pollution. At the end of a sampling interval, the estimated cache pollution metric is compared to two thresholds (P_{low} and P_{high}) to determine whether the pollution caused by the prefetcher was low, medium, or high. If the pollution caused by the prefetcher was low, the prefetched cache blocks are inserted into the middle (MID) position in the LRU stack during the next sampling interval (for an n-way set-associative cache, we define the MID position in the LRU stack as the $floor(n/2)$th least-recently-used position).[9] On the other hand, if the pollution caused by the prefetcher was medium, prefetched cache blocks are inserted into the LRU-4 position in the LRU stack (for an n-way set-associative cache, we define the LRU-4 position in the LRU stack as the $floor(n/4)$th least-recently-used position). Finally, if the pollution caused by the prefetcher was high, prefetched cache blocks are inserted into the LRU position during the next sampling interval.

[8] Note that we adjust prefetcher behavior on a global (across-streams) basis rather than on a per-stream basis as we did not find much benefit in adjusting on a per-stream basis.

[9] We found inserting prefetched blocks to the MRU position doesn't provide significant benefits over inserting them to the MID position. Thus, our dynamic mechanism doesn't insert prefetched blocks to the MRU position. For a detailed analysis of the cache insertion policy, see Section 5.2.

Case	*Prefetch Accuracy*	*Prefetch Lateness*	*Cache Pollution*	**Dynamic Configuration Counter Update** (reason)
1	High	Late	Not-Polluting	Increment (to increase timeliness)
2	High	Late	Polluting	Increment (to increase timeliness)
3	High	Not-Late	Not-Polluting	No Change (best case configuration)
4	High	Not-Late	Polluting	Decrement (to reduce pollution)
5	Medium	Late	Not-Polluting	Increment (to increase timeliness)
6	Medium	Late	Polluting	Decrement (to reduce pollution)
7	Medium	Not-Late	Not-Polluting	No Change (to keep the benefits of timely prefetches)
8	Medium	Not-Late	Polluting	Decrement (to reduce pollution)
9	Low	Late	Not-Polluting	Decrement (to save bandwidth)
10	Low	Late	Polluting	Decrement (to reduce pollution)
11	Low	Not-Late	Not-Polluting	No Change (to keep the benefits of timely prefetches)
12	Low	Not-Late	Polluting	Decrement (to reduce pollution and save bandwidth)

Table 2. How to adapt? Use of the three metrics to adjust the aggressiveness of the prefetcher

4. Evaluation Methodology

We evaluate the performance impact of FDP on an in-house execution-driven Alpha ISA simulator that models an aggressive superscalar, out-of-order execution processor. The parameters of the processor we model are shown in Table 3.

4.1. Memory Model

We evaluate our mechanisms using a detailed memory model which mimics the behavior and the bandwidth/port limitations of all the hardware structures in the memory system faithfully. All the mentioned effects are modeled correctly and bandwidth limitations are enforced in our model as described in [16]. The memory bus has a bandwidth of 4.5 GB/s.

The baseline hardware data prefetcher we model is a stream prefetcher that can track 64 different streams. Prefetch requests generated by the stream prefetcher are inserted into the Prefetch Request Queue which has 128 entries in our model. Requests are drained from this queue and inserted into the L2 Request Queue and are given the lowest priority so that they do not delay demand load/store requests. Requests that miss in the L2 cache access DRAM memory by going through the Bus Request Queue. L2 Request Queue, Bus Request Queue, and L2 Fill Queue have 128 entries each. Only when a prefetch request goes out on the bus does it count towards the number of prefetches sent to memory. A prefetched cache block is placed into the MRU position in the L2 cache in the baseline.

4.2. Benchmarks

We focus our evaluation on those benchmarks from the SPEC CPU2000 suite where the most aggressive prefetcher configuration sends out to memory at least 200K prefetch requests over the 250 million instruction run. On the remaining nine programs of the SPEC CPU2000 suite, the potential for improving either performance or bandwidth-efficiency of the prefetcher is limited because the prefetcher is not active (even if it is configured very aggressively).[10] For reference, the number of prefetches generated for each benchmark in the SPEC CPU2000 suite is shown in Table 4. The benchmarks were compiled using the Compaq C/Fortran compilers with the `-fast` optimizations and profile-driven feedback enabled. All benchmarks are fast forwarded to skip the initialization portion and then simulated for 250 million instructions.

[10] We also evaluated the remaining benchmarks that have less potential. Results for these benchmarks are shown in Section 5.11.

4.3. Thresholds Used in FDP Implementation

The thresholds used in the implementation of our mechanism are provided below. We determined the parameters of our mechanism empirically using a limited number of simulation runs. However, we did not tune the parameters to our application set since this requires an exponential number of simulations in terms of the different parameter combinations. We estimate that optimizing these thresholds can further improve the performance and bandwidth-efficiency of our mechanism.

A_{high}	A_{low}	$T_{lateness}$	$T_{pollution}$	P_{high}	P_{low}
0.75	0.40	0.01	0.005	0.25	0.005

In systems where bandwidth contention is estimated to be higher (e.g. systems where many threads share the memory bandwidth), A_{high} and A_{low} thresholds can be increased to restrict the prefetcher from being too aggressive. In systems where the lateness of prefetches is estimated to be higher due to higher contention in the memory system, reducing the $T_{lateness}$ threshold can increase performance by increasing the timeliness of the prefetcher. Reducing $T_{pollution}$, P_{high} or P_{low} thresholds results in reducing the prefetcher-generated cache pollution. In systems with higher contention for the L2 cache space (e.g. systems with a smaller L2 cache or with many threads sharing the same L2 cache), reducing the values of $T_{pollution}$, P_{high} or P_{low} may be desirable to reduce the cache pollution due to prefetching.

5. Experimental Results and Analyses

5.1. Adjusting Prefetcher Aggressiveness

We first evaluate the performance of FDP to adjust the aggressiveness of the stream prefetcher (as described in Section 3.3.1) in comparison to four traditional configurations that do not incorporate dynamic feedback: No prefetching, Very Conservative prefetching, Middle-of-the-Road prefetching, and Very Aggressive prefetching. Figure 5 shows the IPC performance of each configuration. Adjusting the prefetcher aggressiveness dynamically (i.e. *Dynamic Aggressiveness*) provides the best average performance across all configurations. Dynamically adapting the aggressiveness of the prefetcher using the proposed feedback mechanism provides 4.7% higher average IPC over the Very Aggressive configuration and 11.9% higher IPC over the Middle-of-the-Road configuration.

On almost all benchmarks, *Dynamic Aggressiveness* provides performance that is very close to the performance achieved by the best-performing traditional prefetcher configuration for each benchmark. Hence, the dynamic mechanism is

Pipeline	20-cycle minimum branch misprediction penalty; 4 GHz processor
Branch Predictor	aggressive hybrid branch predictor (64K-entry gshare, 64K-entry per-address w/ 64K-entry selector) wrong-path execution faithfully modeled
Instruction Window	128-entry reorder buffer; 128-entry INT, 128-entry FP physical register files; 64-entry store buffer;
Execution Core	8-wide, fully-pipelined except for FP divide; full bypass network
On-chip Caches	64KB Instruction cache with 2-cycle latency; 64KB, 4-way L1 data cache with 8 banks and 2-cycle latency, allows 4 load accesses per cycle; 1MB, 16-way, unified L2 cache with 8 banks and 10-cycle latency, 128 L2 MSHRs, 1 L2 read port, 1 L2 write port; all caches use LRU replacement and have 64B block size
Buses and Memory	500-cycle minimum main memory latency; 32 DRAM banks; 32B-wide, split-transaction core-to-memory bus at 4:1 frequency ratio; 4.5 GB/s bus bandwidth; max. 128 outstanding misses to main memory; bank conflicts, bandwidth, port contention, and queueing delays faithfully modeled

Table 3. Baseline processor configuration

bzip2	crafty	eon	**gap**	gcc	gzip	**mcf**	**parser**	perlbmk	twolf	**vortex**	**vpr**
336K	59K	4969	**1656K**	110K	31K	**2585K**	**515K**	9218	2749	**591K**	**246K**

ammp	**applu**	apsi	**art**	**equake**	**facerec**	fma3d	**galgel**	lucas	**mesa**	**mgrid**	**sixtrack**	**swim**	**wupwise**
1157K	**6038K**	8656	**13319K**	**2414K**	**2437K**	3643	**243K**	1103	**273K**	**2185K**	**292K**	**8766K**	**799K**

Table 4. Number of prefetches sent by a very aggressive stream prefetcher for each benchmark in the SPEC CPU2000 suite

able to detect and employ the best-performing aggressiveness level for the stream prefetcher on a per-benchmark basis.

Figure 5 shows that *Dynamic Aggressiveness* almost completely eliminates the large performance degradation incurred on some benchmarks due to Very Aggressive prefetching. While the most aggressive traditional prefetcher configuration provides the best average performance, it results in a 28.9% performance loss on applu and a 48.2% performance loss on ammp compared to no prefetching. In contrast, *Dynamic Aggressiveness* results in a 1.8% performance improvement on applu and only a 5.9% performance loss on ammp compared to no prefetching, similar to the best-performing traditional prefetcher configuration for the two benchmarks.

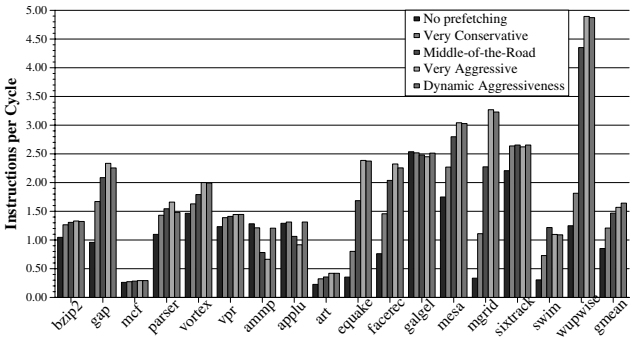

Figure 5. Dynamic adjustment of prefetcher aggressiveness

5.1.1. Adapting to the Program Figure 6 shows the distribution of the value of the *Dynamic Configuration Counter* over all sampling intervals in the *Dynamic Aggressiveness* mechanism. For benchmarks where aggressive prefetching hurts performance (e.g. applu, galgel, ammp), the feedback mechanism chooses and employs the least aggressive dynamic configuration (counter value of 1) for most of the sampling intervals. For example, the prefetcher is configured to be Very Conservative in more than 98% of the intervals for both applu and ammp.

On the other hand, for benchmarks where aggressive prefetching significantly increases performance (e.g. wupwise, mgrid, equake), FDP employs the most aggressive configuration for most of the sampling intervals. For example, the prefetcher is configured to be Very Aggressive in more than 98% of the intervals for wupwise, mgrid, and equake.

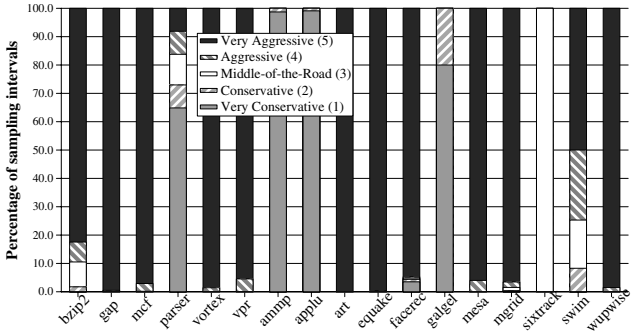

Figure 6. Distribution of the dynamic aggressiveness level

5.2. Adjusting Cache Insertion Policy of Prefetches

Figure 7 shows the performance of dynamically adjusting the cache insertion policy (i.e. *Dynamic Insertion*) using FDP as described in Section 3.3.2. The performance of *Dynamic Insertion* is compared to four different static insertion policies that always insert a prefetched block into the (1) MRU position, (2) MID ($floor(n/2)$th) position where n is the set-associativity, (3) LRU-4 ($floor(n/4)$th least-recently-used) position, and (4) LRU position in the LRU stack. The dynamic cache insertion policy is evaluated using the Very Aggressive prefetcher configuration.

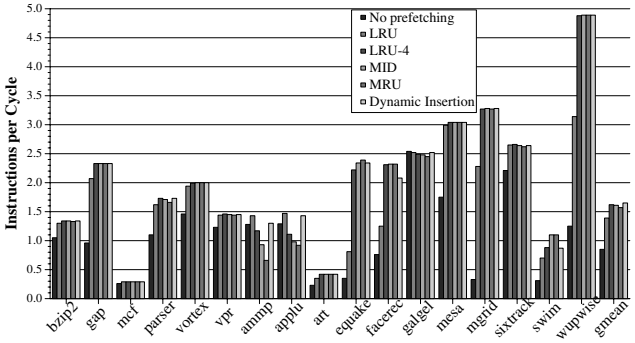

Figure 7. Dynamic adjustment of prefetch insertion policy

The data in Figure 7 shows that statically inserting prefetches in the LRU position can result in significant average performance loss compared to statically inserting prefetches in the MRU position. This is because inserting prefetched blocks

in the LRU position causes an aggressive prefetcher to evict prefetched blocks before they get used by demand loads/stores. However, inserting in the LRU position eliminates the performance loss due to aggressive prefetching in benchmarks where aggressive prefetching hurts performance (e.g. applu and ammp). Among the static cache insertion policies, inserting the prefetched blocks into the LRU-4 position provides the best average performance, improving performance by 3.2% over inserting prefetched blocks in the MRU position.

Adjusting the cache insertion policy dynamically provides higher performance than any of the static insertion policies. *Dynamic Insertion* achieves 5.1% better performance than inserting prefetched blocks into the MRU position and 1.9% better performance than inserting them into the LRU-4 position. Furthermore, *Dynamic Insertion* almost always provides the performance of the best static insertion policy for each benchmark. Hence, dynamically adapting the prefetch insertion policy using run-time estimates of prefetcher-generated cache pollution is able to detect and employ the best-performing cache insertion policy for the stream prefetcher on a per-benchmark basis.

Figure 8 shows the distribution of the insertion position of the prefetched blocks when *Dynamic Insertion* is used. For benchmarks where a static policy of inserting prefetched blocks into the LRU position provides the best performance across all static configurations (applu, galgel, ammp), *Dynamic Insertion* places most (more than 50%) of the prefetched blocks into the LRU position. Therefore, *Dynamic Insertion* improves the performance of these benchmarks by dynamically employing the best-performing insertion policy.

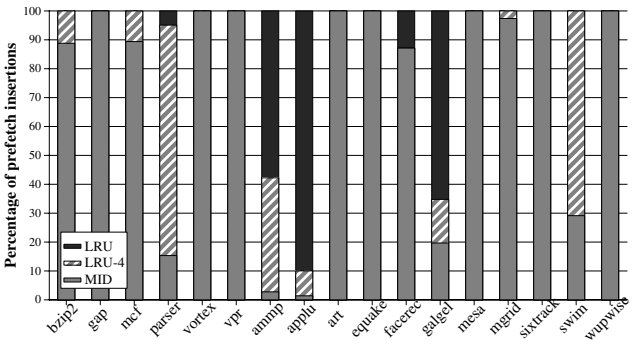

Figure 8. Distribution of the insertion position of prefetched blocks

5.3. Putting It All Together: Dynamically Adjusting Both Aggressiveness and Insertion Policy

This section examines the use of FDP for dynamically adjusting both the prefetcher aggressiveness (*Dynamic Aggressiveness*) and the cache insertion policy of prefetched blocks (*Dynamic Insertion*). Figure 9 compares the performance of five different mechanisms from left to right: (1) No prefetching, (2) Very Aggressive prefetching, (3) Very Aggressive prefetching with *Dynamic Insertion*, (4) *Dynamic Aggressiveness* , and (5) *Dynamic Aggressiveness* and *Dynamic Insertion* together.

Using *Dynamic Aggressiveness* and *Dynamic Insertion* together provides the best performance across all configurations, improving the IPC by 6.5% over the best-performing traditional prefetcher configuration (i.e. Very Aggressive configuration). This performance improvement is greater than the performance

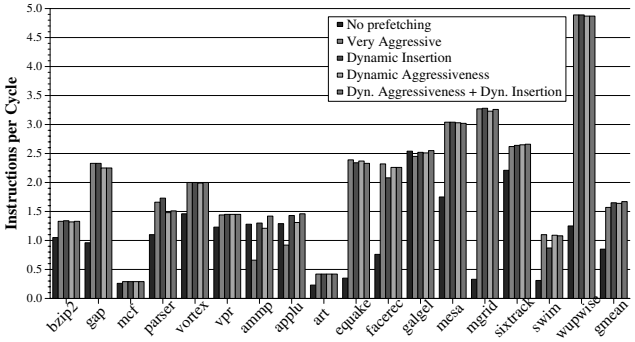

Figure 9. Overall performance of FDP

improvement provided by *Dynamic Aggressiveness* or *Dynamic Insertion* alone. Hence, dynamically adjusting both aspects of prefetcher behavior (aggressiveness and insertion policy) provides complementary performance benefits.

With the use of FDP to dynamically adjust both aspects of prefetcher behavior, the performance loss incurred on some benchmarks due to aggressive prefetching is completely eliminated. No benchmark loses performance compared to no prefetching if both *Dynamic Aggressiveness* and *Dynamic Insertion* are used. In fact, FDP improves the performance of applu by 13.4% and ammp by 11.4% over no prefetching – two benchmarks that otherwise incur very significant performance losses with an aggressive traditional prefetcher configuration.

5.4. Impact of FDP on Bandwidth Consumption

Aggressive prefetching can adversely affect the bandwidth consumption in the memory system when prefetches are not used or when they cause cache pollution. Figure 10 shows the bandwidth impact of prefetching in terms of *Memory Bus Accesses per thousand retired Instructions (BPKI)*.[11] Increasing the aggressiveness of the traditional stream prefetcher significantly increases the memory bandwidth consumption, especially for benchmarks where the prefetcher degrades performance. FDP reduces the aggressiveness of the prefetcher in these benchmarks. For example, in applu and ammp our feedback mechanism usually chooses the least aggressive prefetcher configuration and the least aggressive cache insertion policy as shown in Figures 6 and 8. This results in the large reduction in BPKI shown in Figure 10. FDP (*Dynamic Aggressiveness* and *Dynamic Insertion*) consumes 18.7% less memory bandwidth than the Very Aggressive traditional prefetcher configuration, while it provides 6.5% higher performance.

Table 5 shows the average performance and average bandwidth consumption of different traditional prefetcher configurations and FDP. Compared to the traditional prefetcher configuration that consumes similar amount of memory bandwidth as FDP,[12] FDP provides 13.6% higher performance. Hence, incorporating our dynamic feedback mechanism into the stream prefetcher significantly increases the bandwidth-efficiency of the baseline stream prefetcher.

[11] We use Bus Accesses (rather than the number of prefetches sent) as our bandwidth metric, because this metric includes the effect of L2 misses caused due to demand accesses as well as prefetches. If the prefetcher is polluting the cache, then the number of L2 misses due to demand accesses also increases. Hence, counting the number of bus accesses provides a more accurate measure of the memory bandwidth consumed by the prefetcher.

[12] Middle-of-the-Road configuration consumes only 2.5% less memory bandwidth than FDP.

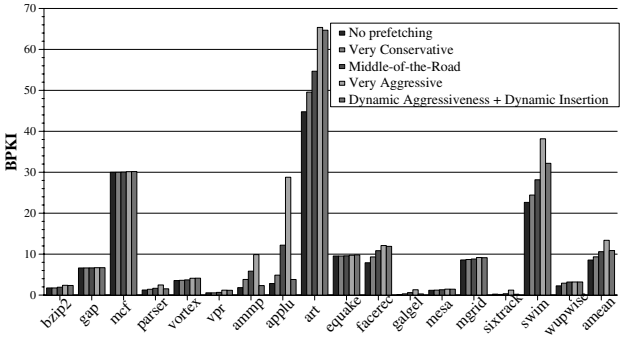

Figure 10. Effect of FDP on memory bandwidth consumption

	No pref.	Very Cons.	Middle	Very Aggr.	FDP
IPC	0.85	1.21	1.47	1.57	1.67
BPKI	8.56	9.34	10.60	13.38	10.88

Table 5. Average IPC and BPKI for FDP vs conventional prefetchers

5.5. Hardware Cost and Complexity of FDP

Table 6 shows the hardware cost of the proposed mechanism in terms of the required state. FDP does not add significant combinational logic complexity to the processor. Combinational logic is required for the update of counters, update of the *pref-bit*s in the L2 cache, update of the entries in the pollution filter, calculation of feedback metrics at the end of each sampling interval, determination of when a sampling interval ends, and insertion of prefetched blocks into appropriate locations in the LRU stack of an L2 cache set. None of the required logic is on the critical path of the processor. The storage overhead of our mechanism is less than 0.25% of the data-store size of the baseline 1MB L2 cache.

5.6. Using only Prefetch Accuracy for Feedback

We use a comprehensive set of metrics –prefetch accuracy, timeliness, and pollution– in order to provide feedback to adjust the prefetcher aggressiveness. In order to assess the benefit of using timeliness as well as cache pollution, we evaluated a mechanism where we adapted the prefetcher aggressiveness based only on accuracy. In such a scheme, we increment the *Dynamic Configuration Counter* if the accuracy is high and decrement it if the accuracy is low. We found that, compared to this scheme that only uses accuracy to throttle the aggressiveness of a stream prefetcher, our comprehensive mechanism that also takes into account timeliness and cache pollution provides 3.4% higher performance and consumes 2.5% less bandwidth.

5.7. FDP vs. Using a Prefetch Cache

Cache pollution caused by prefetches can be eliminated by bringing prefetched data into separate prefetch buffers [13, 11] rather than inserting prefetched data into the L2 cache. Figures 11 and 12 respectively show the performance and bandwidth consumption of the Very Aggressive prefetcher with different prefetch cache sizes - ranging from a 2KB fully-associate prefetch cache to a 1MB 16-way prefetch cache.[13] The performance of the Very Aggressive prefetcher and FDP when prefetched data is inserted into the L2 cache is also shown.

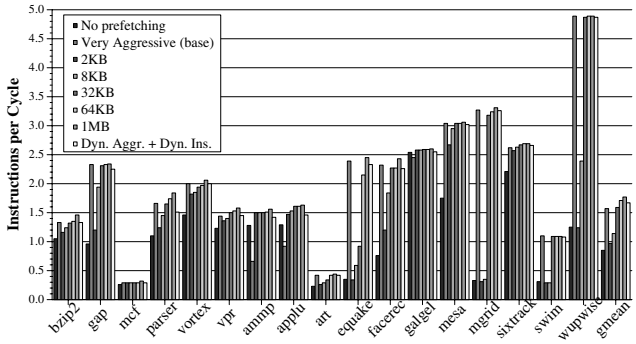

Figure 11. Performance of prefetch cache vs. FDP

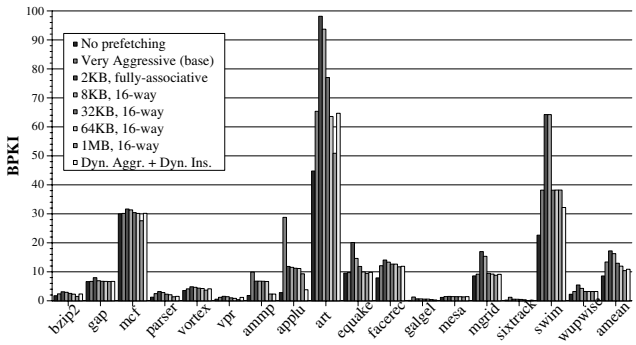

Figure 12. Bandwidth consumption of prefetch cache vs. FDP

The results show that using small (2KB and 8KB) prefetch caches do not provide as high performance as inserting the prefetched data into the L2 cache. With an aggressive prefetcher and a small prefetch cache, the prefetched blocks are displaced by later prefetches before being used by the program - which results in performance degradation with a small prefetch cache. However, larger prefetch caches (32KB and larger) improve performance compared to inserting prefetched data into the L2 cache because a larger prefetch cache reduces the pollution caused by prefetched data in the L2 cache while providing enough space for prefetched blocks.

Using FDP (both *Dynamic Aggressiveness* and *Dynamic Insertion*) that prefetches into the L2 cache provides 5.3% higher performance than that provided by augmenting the Very Aggressive traditional prefetcher configuration with a 32KB prefetch cache. The performance of FDP is also within 2% of the performance of the Very Aggressive configuration with a 64KB prefetch cache. Furthermore, the memory bandwidth consumption of FDP is 16% and 9% less than the Very Aggressive prefetcher configurations with respectively a 32KB and 64KB prefetch cache. Hence, FDP achieves the performance provided by a relatively large prefetch cache bandwidth-efficiently and without requiring as large hardware cost and complexity as that introduced by the addition of a prefetch cache that is larger than 32KB.

5.8. Effect on a Global History Buffer Prefetcher

We have also implemented FDP on the C/DC (C-Zone Delta Correlation) variant of the Global History Buffer (GHB) prefetcher [10]. In order to vary the aggressiveness of this

[13] In the configurations with a prefetch cache, a prefetched cache block is moved from the prefetch cache into the L2 cache if it is accessed by a demand load/store request. The block size of the prefetch cache and the L2 cache are the same and the prefetch cache is assumed to be accessed in parallel with the L2 cache without any adverse latency impact on L2 cache access time.

pref-bit for each tag-store entry in the L2 cache	16384 blocks * 1 bit/block = 16384 bits
Pollution Filter	4096 entries * 1 bit/entry = 4096 bits
16-bit counters used to estimate feedback metrics	11 counters * 16 bits/counter = 176 bits
pref-bit for each MSHR entry	128 entries * 1 bit/entry = 128 bits
Total hardware cost	20784 bits = 2.54 KB
Percentage area overhead compared to baseline 1MB L2 cache	2.5KB/1024KB = **0.24%**

Table 6. Hardware cost of feedback directed prefetching

prefetcher dynamically, we vary the *Prefetch Degree*.[14] Below, we show the aggressiveness configurations used for the GHB prefetcher. FDP adjusts the configuration of the GHB prefetcher as described in Section 3.3.

Dyn. Config. Counter	*Aggressiveness*	*Prefetch Degree*
1	Very Conservative	4
2	Conservative	8
3	Middle-of-the-Road	16
4	Aggressive	32
5	Very Aggressive	64

Figure 13 shows the performance and bandwidth consumption of different GHB prefetcher configurations and the feedback directed GHB prefetcher using both *Dynamic Aggressiveness* and *Dynamic Insertion*. The feedback directed GHB prefetcher performs similarly to the best-performing traditional configuration (Very Aggressive configuration), while it consumes 20.8% less memory bandwidth. Compared to the traditional GHB prefetcher configuration that consumes similar amount of memory bandwidth as FDP (i.e. Middle-of-the-Road configuration), FDP provides 9.9% higher performance. Hence, FDP significantly increases the bandwidth-efficiency of GHB-based delta correlation prefetching. Note that it is possible to improve the performance and bandwidth benefits of the proposed mechanism by tuning the thresholds used in feedback mechanisms to the behavior of the GHB-based prefetcher, but we did not pursue this option.

5.9. Effect of FDP on a PC-Based Stride Prefetcher

We also evaluated FDP on a PC-based stride prefetcher [1] and found that the results are similar to those achieved on both stream and GHB-based prefetchers. On average, using the feedback directed approach results in a 4% performance gain and a 24% reduction in memory bandwidth compared to the best-performing conventional configuration for a PC-based stride prefetcher. Due to space constraints, we do not present detailed graphs for these results.

5.10. Sensitivity to L2 Size and Memory Latency

We evaluate the sensitivity of FDP to different cache sizes and memory latencies. In these experiments, we varied the L2 cache size keeping the memory latency at 500 cycles (baseline) and varied the memory latency keeping the cache size at 1MB (baseline). Table 7 shows the change in average IPC and BPKI provided by FDP over the best performing conventional prefetcher configuration. FDP provides better performance and consumes significantly less bandwidth than the best-performing conventional prefetcher configuration for all evaluated cache sizes and memory latencies. As memory latency increases, the IPC improvement of FDP also increases because the effectiveness of the prefetcher becomes more important when memory becomes a larger performance bottleneck.

[14] In the GHB-based prefetching mechanism, *Prefetch Distance* and *Prefetch Degree* are the same.

5.11. Effect on Other SPEC CPU2000 Benchmarks

Figure 14 shows the IPC and BPKI impact of FDP on the remaining 9 SPEC CPU2000 benchmarks that have less potential. We find that our feedback directed scheme provides 0.4% performance improvement over the best performing conventional prefetcher configuration (i.e. Middle-of-the-Road configuration) while reducing the bandwidth consumption by 0.2%. None of the benchmarks lose performance with FDP. Note that the best-performing conventional configuration for these 9 benchmarks is not the same as the best-performing conventional configuration for the 17 memory-intensive benchmarks (i.e. Very-Aggressive configuration). Also note that the remaining 9 benchmarks are not bandwidth-intensive except for fma3d and gcc. In gcc, the performance improvement of FDP is 3.0% over the Middle-of-the-Road configuration. The prefetcher pollutes the L2 cache and evicts many useful instruction blocks in gcc, resulting in very long-latency instruction cache misses that leave the processor idle. Using FDP reduces this negative effect by detecting the pollution caused by prefetch references and dynamically reducing the aggressiveness of the prefetcher.

6. Related Work

Even though mechanisms for prefetching have been studied for a long time, dynamic mechanisms to adapt the aggressiveness of the prefetcher have not been studied as extensively as algorithms that decide what to prefetch. We briefly describe previous work in dynamic adaptation of prefetching policies.

6.1. Dynamic Adaptation of Data Prefetching Policies

The work most related to ours in adapting the prefetcher's aggressiveness is Dahlgren et al.'s paper that proposed adaptive sequential (next-line) prefetching [4] for multiprocessors. This mechanism implemented two counters to count the number of sent prefetches (*counter-sent*) and the number of useful prefetches (*counter-used*). When counter-sent saturates, counter-used is compared to a static threshold to decide whether to increase or decrease the aggressiveness (i.e. *Prefetch Distance*) of the prefetcher. While Dahlgren et al.'s mechanism to calculate prefetcher accuracy is conceptually similar to ours, their approach considered only prefetch accuracy to dynamically adapt prefetch distance. Also, their mechanism is designed for a simple sequential prefetching mechanism which prefetches up to 8 cache blocks following each cache miss. In this paper, we provide a generalized feedback-directed approach for dynamically adjusting the aggressiveness of a wide range of state-of-the-art hardware data prefetchers by taking into account not only accuracy but also timeliness and pollution.

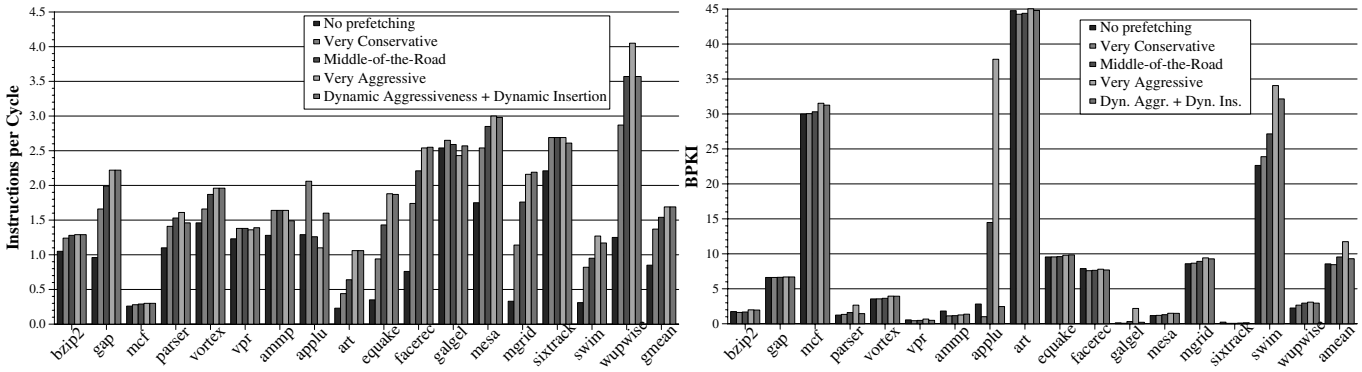

Figure 13. Effect of FDP on the IPC performance (left) BPKI memory bandwidth consumption (right) of GHB-based C/DC prefetchers

L2 Cache Size (memory latency = 500 cycles)						**Memory Latency** (L2 cache size = 1 MB)					
512 KB		1 MB		2 MB		250 cycles		500 cycles		1000 cycles	
Δ IPC	Δ BPKI	Δ IPC	Δ BPKI	Δ IPC	Δ BPKI	Δ IPC	Δ BPKI	Δ IPC	Δ BPKI	Δ IPC	Δ BPKI
0%	-13.9%	6.5%	-18.7%	6.3%	-29.6%	4.5%	-23.0%	6.5%	-18.7%	8.4%	-16.9%

Table 7. Change in IPC and BPKI with FDP when L2 size and memory latency are varied

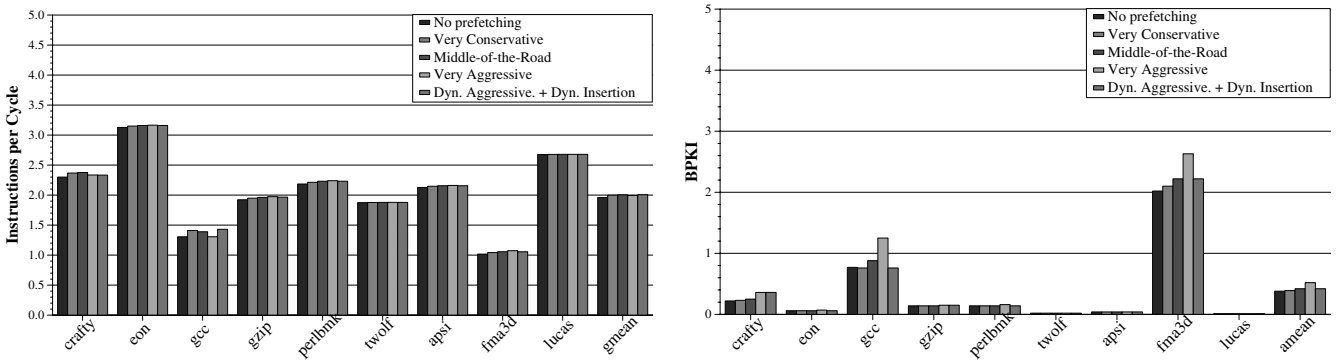

Figure 14. IPC performance (left) and memory bandwidth consumption in BPKI (right) impact of FDP on the remaining SPEC benchmarks

When the program enters a new phase of execution, the prefetcher is tuned based on the characteristics of the phase in Nesbit et al. [10]. In order to perform phase detection/prediction and identification of the best prefetcher configuration for a given phase, significant amount of extra hardware is needed. In comparison, our mechanism is simpler because it does not require phase detection or prediction mechanisms.

Recently, Hur and Lin [7] proposed a probabilistic technique that adjusts the aggressiveness of a stream prefetcher based on the estimated spatial locality of the program. Their approach is applicable only to stream prefetchers as it tries to estimate the a histogram of the stream length.

6.2. Cache Pollution Filtering

Charney and Puzak [3] proposed filtering L1 cache pollution caused by next-sequential prefetching and shadow directory prefetching from the L2 cache into the L1 cache. Their scheme associates a *confirmation bit* with each block in the L2 cache which indicates if the block was used by a demand access when it was prefetched into the L1 cache the last time. If the confirmation bit is not set when a prefetch request accesses the L2, the prefetch request is discarded. Extending this scheme to prefetching from main memory to the L2 cache requires a separate structure that maintains information about the blocks evicted from the L2 cache. This significantly increases the hardware cost of their mechanism. Our mechanism does not need to keep history information for evicted L2 cache blocks.

Zhuang and Lee [25] proposed to filter prefetcher-generated cache pollution by using schemes similar to two-level branch predictors. Their mechanism tries to identify whether or not a prefetch will be useful based on past information about the usefulness of the prefetches generated to the same memory address or triggered by the same load instruction. In contrast, our mechanism does not require the collection of fine-grain information on each prefetch address or load address in order to vary the aggressiveness of the prefetcher.

Other approaches for cache pollution filtering include using a profiling mechanism to mark load instructions that can trigger hardware prefetches [23], and using compile-time techniques to mark dead cache locations so that prefetches can be inserted in dead locations [9]. In comparison to these two mechanisms, our mechanism does not require any software or ISA support and can adjust to dynamic program behavior even if it differs from the behavior of the compile-time profile. Lin et al. [15] proposed using density vectors to determine what to prefetch inside a region. This was especially useful in their model as they used very bandwidth-intensive scheduled region prefetching, which prefetches all the cache blocks in a memory region on a cache miss. This approach can be modified and combined with our proposal to further remove the pollution caused by blocks that are not used in a prefetch stream.

Mutlu et al. [17] used the L1 caches as filters to reduce L2

cache pollution caused by useless prefetches. In their scheme, all prefetched blocks are placed into only the L1 cache. A prefetched block is placed into the L2 when it is evicted from the L1 cache only if it was needed by a demand request while it was in L1. In addition to useless prefetches, this approach also filters out some useful but early prefetches that are not used while residing in the L1 cache (such prefetches are common in very aggressive prefetchers). To obtain performance benefit from such prefetches, their scheme can be combined with our cache insertion policy.

6.3. Cache Insertion Policy for Prefetches

Lin et al. [14] evaluated static policies to determine the placement in cache of prefetches generated by a scheduled region prefetcher. Their scheme placed prefetches in the LRU position of the LRU stack. We found that, even though inserting prefetches in the LRU position reduces the cache pollution effects of prefetches on some benchmarks, it also reduces the positive benefits of aggressive stream prefetching on other benchmarks because useful prefetches–if placed in the LRU position–can be easily evicted from the cache in an aggressive prefetching scheme without providing any benefit. Dynamically adjusting the insertion policy of prefetched blocks based on the estimated pollution increases performance by 1.9% over the best static policy (LRU-4) and by 18.8% over inserting prefetches in the LRU position.

7. Conclusion and Future Work

This paper proposed a feedback directed mechanism that dynamically adjusts the behavior of a hardware data prefetcher to improve performance and reduce memory bandwidth consumption. Over previous research in adaptive prefetching, our contributions are:

- We propose a comprehensive and low-cost feedback mechanism that takes into account prefetch accuracy, timeliness, and cache pollution caused by prefetch requests together to both throttle the aggressiveness of the prefetcher and to decide where in the cache to place the prefetched blocks. Previous approaches considered using only prefetch accuracy to determine the aggressiveness of simple sequential (next-line) prefetchers.
- We develop a low-cost mechanism to estimate at run-time the cache pollution caused by hardware prefetching.
- We propose and evaluate using comprehensive feedback mechanisms for state-of-the-art stream prefetchers that are commonly employed by today's high-performance processors. Our feedback-directed mechanism is applicable to any kind of hardware data prefetcher. We show that it works well with stream-based prefetchers, global-history-buffer based prefetchers and PC-based stride prefetchers. Previous adaptive mechanisms were applicable to only simple sequential prefetchers [4].

Future work can incorporate other important metrics, such as available memory bandwidth, estimates of the contention in the memory system, and prefetch coverage, into the dynamic feedback mechanism to provide further improvement in performance and further reduction in memory bandwidth consumption. The metrics defined and used in this paper could also be used as part of the selection mechanism in a hybrid prefetcher. Finally, the mechanisms proposed in this paper can be easily extended to instruction prefetchers.

Acknowledgments

We thank Matthew Merten, Moinuddin Qureshi, members of the HPS Research Group, and the anonymous reviewers for their comments and suggestions. We gratefully acknowledge the support of the Cockrell Foundation, Intel Corporation and the Advanced Technology Program of the Texas Higher Education Coordinating Board.

References

[1] J. Baer and T. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of Supercomputing '91*, 1991.
[2] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.
[3] M. Charney and T. Puzak. Prefetching and memory system behavior of the SPEC95 benchmark suite. *IBM Journal of Reseach and Development*, 41(3):265–286, 1997.
[4] F. Dahlgren, M. Dubois, and P. Stenström. Sequential hardware prefetching in shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 6(7):733–746, 1995.
[5] J. D. Gindele. Buffer block prefetching method. *IBM Technical Disclosure Bulletin*, 20(2):696–697, July 1977.
[6] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, Feb. 2001. Q1 2001 Issue.
[7] I. Hur and C. Lin. Memory prefetching using adaptive stream detection. In *MICRO-39*, 2006.
[8] S. Iacobovici, L. Spracklen, S. Kadambi, Y. Chou, and S. G. Abraham. Effective stream-based and execution-based data prefetching. In *ICS*, 2004.
[9] P. Jain, S. Devadas, and L. Rudolph. Controlling cache pollution in prefetching with software-assisted cache replacement. Technical Report CSG-462, Massachusetts Institute of Technology, 2001.
[10] K. J.Nesbit, A. S. Dhodapkar, and J. E.Smith. AC/DC: An adaptive data cache prefetcher. In *PACT*, 2004.
[11] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *ISCA-17*, 1990.
[12] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *ISCA-8*, 1981.
[13] R. L. Lee, P.-C. Yew, and D. H. Lawrie. Data prefetching in shared memory multiprocessors. In *ICPP*, 1987.
[14] W.-F. Lin, S. K. Reinhardt, and D. Burger. Reducing DRAM latencies with an integrated memory hierarchy design. In *HPCA-7*, 2001.
[15] W.-F. Lin, S. K. Reinhardt, D. Burger, and T. R. Puzak. Filtering superfluous prefetches using density vectors. In *ICCD*, 2001.
[16] O. Mutlu, H. Kim, D. N. Armstrong, and Y. N. Patt. An analysis of the performance impact of wrong-path memory references on out-of-order and runahead execution processors. *IEEE Transactions on Computers*, 54(12):1556–1571, Dec. 2005.
[17] O. Mutlu, H. Kim, D. N. Armstrong, and Y. N. Patt. Using the first-level caches as filters to reduce the pollution caused by speculative memory references. *International Journal of Parallel Programming*, 33(5):529–559, October 2005.
[18] O. Mutlu, H. Kim, and Y. N. Patt. Techniques for efficient processing in runahead execution engines. In *ISCA-32*, 2005.
[19] S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *ISCA-21*, 1994.
[20] J.-K. Peir, S.-C. Lai, S.-L. Lu, J. Stark, and K. Lai. Bloom filtering cache misses for accurate data speculation and prefetching. In *ICS*, 2002.
[21] A. J. Smith. Cache memories. *Computing Surveys*, 14(4):473–530, 1982.
[22] L. Spracklen and S. G. Abraham. Chip multithreading: Opportunities and challenges. In *HPCA-11*, 2005.
[23] V. Srinivasan, G. S. Tyson, and E. S. Davidson. A static filter for reducing prefetch traffic. Technical Report CSE-TR-400-99, University of Michigan Technical Report, 1999.
[24] J. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Technical White Paper*, Oct. 2001.
[25] X. Zhuang and H.-H. S. Lee. A hardware-based cache pollution filtering mechanism for aggressive prefetches. In *ICPP-32*, 2003.