Sandbox Prefetching: Safe Run-Time Evaluation of Aggressive Prefetchers

Seth H Pugsley[1], Zeshan Chishti[2], Chris Wilkerson[2], Peng-fei Chuang[3], Robert L Scott[3], Aamer Jaleel[4], Shih-Lien Lu[2], Kingsum Chow[3], and Rajeev Balasubramonian[1]

[1]University of Utah, {pugsley, rajeev}@cs.utah.edu
[2]Intel Labs, {zeshan.a.chishti, chris.wilkerson, shih-lien.l.lu}@intel.com
[3]Intel Software and Services Group, {peng-fei.chuang, robert.l.scott, kingsum.chow}@intel.com
[4]Intel Corporation VSSAD, aamer.jaleel@intel.com

Abstract

Memory latency is a major factor in limiting CPU performance, and prefetching is a well-known method for hiding memory latency. Overly aggressive prefetching can waste scarce resources such as memory bandwidth and cache capacity, limiting or even hurting performance. It is therefore important to employ prefetching mechanisms that use these resources prudently, while still prefetching required data in a timely manner.

In this work, we propose a new mechanism to determine at run-time the appropriate prefetching mechanism for the currently executing program, called Sandbox Prefetching. Sandbox Prefetching evaluates simple, aggressive offset prefetchers at run-time by adding the prefetch address to a Bloom filter, rather than actually fetching the data into the cache. Subsequent cache accesses are tested against the contents of the Bloom filter to see if the aggressive prefetcher under evaluation could have accurately prefetched the data, while simultaneously testing for the existence of prefetchable streams. Real prefetches are performed when the accuracy of evaluated prefetchers exceeds a threshold. This method combines the ideas of global pattern confirmation and immediate prefetching action to achieve high performance. Sandbox Prefetching improves performance across the tested workloads by 47.6% compared to not using any prefetching, and by 18.7% compared to the Feedback Directed Prefetching technique. Performance is also improved by 1.4% compared to the Access Map Pattern Matching Prefetcher, while incurring considerably less logic and storage overheads.

1 Introduction

Modern high performance microprocessors employ hardware prefetching techniques to mitigate the performance impact of long memory latencies. These prefetchers operate by predicting which memory addresses will be accessed by a program in the near future and then speculatively issuing memory requests for those addresses. The performance improvement afforded by a prefetcher depends on its ability to correctly predict the memory addresses accessed by a program. Accurate prefetches hide the memory latency of potential demand misses by bringing data earlier to the on-chip caches. In comparison, inaccurate prefetches result in two problems: First, they increase the contention for the available memory bandwidth, which could result in both performance losses and energy overheads. Second, they waste cache capacity, which could result in additional cache misses, contributing to the problem they were intended to solve. In fact, there is often a trade-off between prefetch accuracy and coverage, i.e., to bring in more useful cache lines, the prefetcher must also bring in more useless cache lines. Therefore, before employing a prefetching technique, it is important to weigh the relative benefit of accurate prefetches against the bandwidth and cache capacity concerns of inaccurate prefetches.

One common approach to maximizing prefetcher accuracy is to track streams of accesses in the address space. This is done by observing several memory accesses forming a regular pattern, and then predicting that pattern will continue in subsequent memory accesses. These prefetchers can be accurate, but take some time to confirm streams before any performance benefit can be reaped.

Other techniques, such as next-line prefetchers, or even the use of larger cache lines, harvest fine grained spatial locality, but do so blindly for all memory references without first evaluating the benefits. As a result, these prefetchers may increase bandwidth and power while providing no performance benefit. Kumar et al. [17], took advantage of the benefits of fine grained spatial locality while avoiding the overheads of superfluous prefetches by maintaining bit-vectors indicating which nearby cache lines were most likely to be used after a reference to a particular cache line

978-1-4799-3097-5/14/$31.00 ©2014 IEEE

occurred. The key drawback of this approach is the overhead of storing the bit vector for each footprint.

In this paper, we build on the previous work on prefetchers that exploit fine grained spatial locality. Rather than build and store patterns as in [17], our approach evaluates a few previously defined patterns and identifies at run-time which patterns are most likely to provide benefit. Since not all previously defined prefetch patterns are appropriate for all workloads, we must identify when and where specific patterns will benefit performance.

To address this problem we introduce the concept of a "Prefetch Sandbox." The key idea behind a prefetch sandbox is to track prefetch requests generated by a candidate prefetch pattern, without actually issuing those prefetch requests to the memory system. To test the accuracy of a candidate prefetcher, the sandbox stores the addresses of all the cache lines that would have been prefetched by the candidate pattern. We implement the sandbox using a Bloom filter for its space efficiency and constant lookup time. Based on the number of simulated prefetch hits, a candidate prefetcher may be globally activated to immediately perform a prefetch action after every cache access, with no further confirmation.

Our results show that using the proposed Sandbox Prefetching technique improves the average performance of 14 memory-intensive benchmarks in the SPEC2006 suite by 47.6% compared to no prefetching, by 18.7% compared to the state-of-the-art Feedback Directed Prefetching, and by 1.4% compared to the Access Map Pattern Matching Prefetcher, which has a considerably larger storage and logic requirement compared to Sandbox Prefetching.

2 Background

Hardware prefetchers for regular data access patterns fall into two broad categories: conservative, confirmation-based prefetchers (such as a stream prefetcher), and aggressive, immediate prefetchers (such as a next-line prefetcher). These two varieties of prefetchers have generally opposite goals and methods, but Sandbox Prefetching combines attributes of both.

2.1 Confirmation-Based Prefetchers

A confirmation-based prefetcher is one that performs a prefetch only after it has built up some confidence that the prefetch will be useful. A stream prefetcher is a good example of a confirmation-based prefetcher. When a cache line address *A* is seen for the first time, no prefetches are performed at this time, but the stream prefetcher begins watching for address *A+1*. Even when *A+1* is seen, still no prefetches are made, because there is not yet enough evidence that this is a true stream. Only after *A+2* is also seen, and the stream has been fully "confirmed" will *A+3* (and perhaps further cache lines) finally be prefetched.

Since confirmation prefetchers always wait for some time before issuing any prefetches, they will always leave some performance on the table. Even if a stream prefetcher is perfect at prefetching a long stream in a timely and accurate manner, the fact that it had to confirm the stream before the first prefetch was issued means that its performance will always be limited, because it missed out on prefetching the first three accesses.

Confirmation-based prefetchers have the advantage that once a pattern has been confirmed, many prefetches can be issued along that pattern, far ahead of the program's actual access stream. This improves performance by avoiding late prefetches.

Confirmation-based stream prefetchers operate on the granularity of individual streams. Each address that comes to the prefetcher is considered to be either part of an existing stream, or not a part of any known stream, in which case a new stream will be allocated for it. Once a stream has been confirmed, prefetches may be made along it, but each stream must be confirmed independently of all other streams. Furthermore, a new stream will have to be allocated and confirmed whenever an access stream reaches a virtual page boundary, because the next physical page will not yet be known.

2.2 Immediate Prefetchers

An immediate prefetcher is one that generates a prefetch address and performs a prefetch as soon as it is given an input reference address. The most basic and common of these types of prefetchers is the next-line prefetcher. Every time the next-line prefetcher is given the address of a cache line *A*, it will immediately prefetch the cache line *A+1*. The only requirement for the next-line prefetcher to prefetch address *A+1* is for it to see address *A*. No additional confirmation or input is required.

Immediate prefetchers have the disadvantage that they have a higher probability of being inaccurate, compared to confirmation-based prefetchers. Also, with immediate prefetchers, there is no notion of prefetching "ahead" of a stream of accesses, because an immediate prefetcher takes only a single action each time it is invoked.

Immediate prefetchers have the advantage that they can prefetch patterns which a confirmation-based prefetcher cannot prefetch, because no confirmable pattern exists. For example, consider a linked list of data structures that are exactly the size of two cache lines. A confirmation-based prefetcher would consider the first cache line of each of these linked list nodes to be the beginning of a new pattern, and accessing the second cache line would help build confidence in this pattern, but the third sequential access would never come, because the linked list would jump somewhere else in memory. On the other hand, because immediate prefetchers work on the granularity of individual cache lines, and not streams, a next-line prefetcher would be able to perfectly prefetch the second cache line of these linked list nodes.

3 Related Work

There are numerous studies that have proposed novel prefetching algorithms [13, 3, 15, 19, 7, 20, 14, 5, 18, 10, 11, 22, 6, 9, 23, 2, 1, 17, 16]. Initial research on prefetching relied on the fact that many applications exhibit a high degree of spatial locality. As such, many studies showed these applications can benefit from sequential and stride prefetching [6, 9]. However, applications that lack spatial locality receive very little benefit from sequential prefetching. Therefore, more complex prefetching proposals such as Markov-based prefetchers [14], and prefetchers for pointer chasing applications [7, 20], have also been proposed. While we cannot cover all prefetching related research work, we summarize prior art that closely relates to our Sandbox Prefetching technique.

There have been a few studies that dynamically adapt the aggressiveness of prefetchers. Dahlgren et al. proposed adaptive sequential prefetching for multiprocessors [5]. Their proposal dynamically modulated prefetcher distance by tracking the usefulness of prefetches. If the prefetches are useful, the prefetch distance is increased, otherwise it is decreased. Jiminez et al. present a real life dynamic implementation of a prefetcher in the POWER7 processor [13]. The POWER7 processor supports a number of prefetcher configurations and prefetch distances (seven in all). Their approach exposes to the operating system software the different prefetcher configurations using a Configuration Status Register (CSR) on a per-core basis. The operating system/software time samples the performance of all prefetch configurations and chooses the best prefetch setting for the given core and runs it for several time quanta. In contrast to this work, which uses software to evaluate and program the hardware prefetcher, our proposed scheme is a hardware-only solution.

In this work, we compare Sandbox Prefetching to Feedback Directed Prefetching (FDP) [21] and Address Map Pattern Matching Prefetching (AMPM) [12]. We now describe both of these techniques in some detail.

3.1 Feedback Directed Prefetching

FDP is an improvement on a conventional stream prefetcher, which takes into account the accuracy and timeliness of the prefetcher, as well as the cache pollution generated by the prefetcher, to dynamically vary how aggressively the prefetcher operates.

FDP works by maintaining a structure that tracks multiple different access streams. The FDP mechanism is invoked in the event of an L2 cache miss (which was the last level cache miss in the FDP work). When a new stream is accessed for the first time, a new entry in the tracking structure is allocated. The stream then trains on the next two cache misses that fall within +/-16 cache blocks of the initial miss in order to determine the direction, whether positive or negative, that the stream is heading in.

After a stream and its direction are confirmed, the stream tracker enters monitor and request mode. The tracker monitors a region of memory, between a start and end pointer, and whenever there is an access to that region, one or more prefetches are issued, and the bounds of the monitored region of memory are advanced in the direction of the stream. The size of the monitored memory region, and the number of cache lines which are prefetched with each access, are determined by the current aggressiveness level.

FDP has five levels of aggressiveness that it can switch between, given the current behavior of the program. The least aggressive level monitors a region of 4 cache blocks, and prefetches a single cache line on each stream access. The most aggressive level monitors a region of 64 cache blocks, and prefetches 4 cache lines on each stream access.

3.2 Address Map Pattern Matching

AMPM is the winner of a data prefetching championship whose only limitations were on the number of bits that could be used to store prefetcher state (4 KB). As a consequence, AMPM uses complex logic to make the most of its limited storage budget. The main idea of AMPM is to track every cache line in large 16 KB regions of memory, and then to exhaustively determine if any strides can be discovered through the use of pattern matching, and then to prefetch along those strides.

AMPM tracks address maps for 52 16 KB regions of memory, which maintain 2 bits of state for each cache line in the region, corresponding to whether the line has not been accessed, has been demand accessed, or has been prefetched. This address map is updated on every L2 access and prefetch (L2 was the last level of cache in their work).

Also on every L2 access, the address map corresponding to the current access is retrieved from a fully-associative array of address maps, and is placed in a large shift register to align the map with the current access address. Then it attempts to match 256 separate patterns with this shifted address map, each pattern match requiring two compares to discover series of strided accesses centered around the current access. This generates a list of candidate prefetches, and a number of these are prefetched according to a dynamically changing prefetch degree, and in the order of the smallest magnitude offset to the largest.

4 Sandbox Prefetching

Sandbox Prefetching (SBP) represents another class of prefetcher, and combines the ideas of global confirmation with immediate action to aggressively, and safely, perform prefetches. All of our discussion and evaluations are made in the context of a two-level cache heirharchy, and prefetches happen exclusively from memory to L2.

4.1 Overview

SBP operates on the principle of validating the accuracy of aggressive, immediate offset prefetchers in a safe, sand-

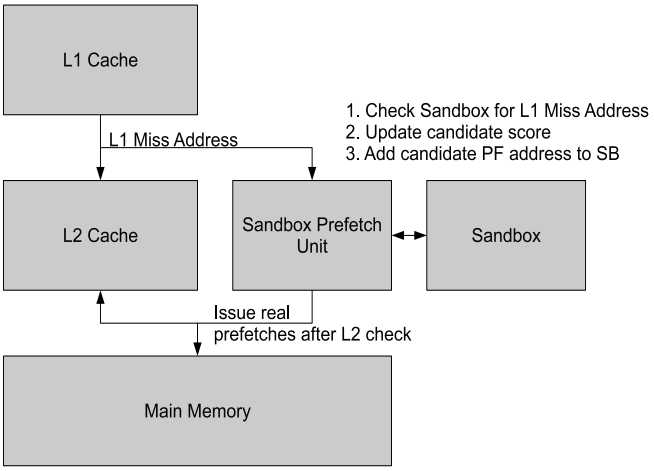


Figure 1. Sandbox Prefetching's place in the memory hierarchy.

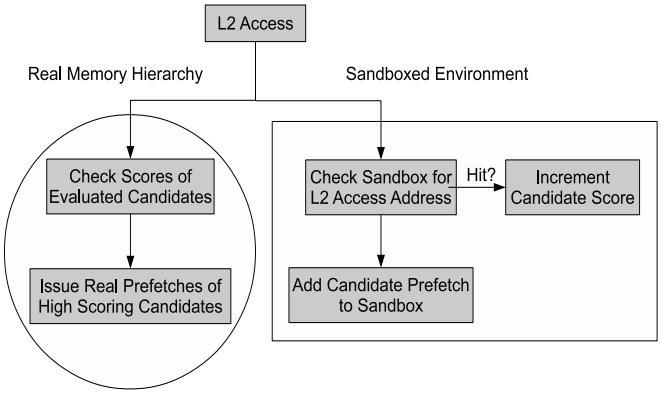


Figure 2. Sandbox Prefetching acts on every L2 access.

boxed environment, where neither the real cache nor memory bandwidth are disturbed, and then deploying them in the real memory hierarchy only if they prove that they can accurately prefetch useful data. A set of candidate prefetchers, each corresponding to a specific cache line offset, are constantly evaluated and re-evaluated for accuracy, and the most accurate of them are allowed to issue real prefetches to main memory.

Immediate prefetchers have one single prefetch action, and they perform this action in all situations they are used. A next-line prefetcher will always fetch the plus-one cache line, regardless of the input it receives. It is the same with the candidate offset prefetchers. Each one of them will perform a prefetch with a specific offset from the current input cache line address. Prefetcher accuracy is a concern, and we therefore cannot allow all candidate prefetchers to issue prefetches all the time.

Candidates are evaluated by simulating their prefetch action and measuring the simulated effect. This is done by adding prefetch addresses into a sandbox, rather than issuing real prefetches to main memory. The sandbox is a structure which implements a set and keeps track of addresses which have been added to it. Subsequent cache accesses test the sandbox to see if their address can be found there. If the address is found there, then the current candidate prefetcher could have accurately prefetched this cache line, and that candidate's accuracy score is increased. The accuracy score is used to tell which, if any, of the candidate prefetchers has qualified to issue real prefetches in the real memory hierarchy. If the address is not found there, then that means the current candidate prefetcher could not have accurately prefetched this line.

Candidate prefetchers are not "confirmed" in the context of a single access stream, as in a stream prefetcher, but rather in the context of all memory access patterns present in the currently executing program. We do not test if a particular offset prefetcher, which prefetches offset O from the current cache line address, is accurate for only a single stream, but we test to see if for *every* access A, that there is a subsequent access to $A+O$. If the pattern holds true for a large enough number of cache accesses, then the candidate prefetcher is turned on in the real memory hierarchy.

Each candidate is evaluated for a fixed number of L2 accesses, and then the contents of the sandbox are reset, and the next candidate is evaluated.

4.2 The Sandbox

The sandbox of Sandbox Prefetching is implemented as a Bloom filter[16]. Each prefetch address generated by the current candidate prefetcher is added to the sandbox Bloom filter, and each time there is a cache access, the Bloom filter is tested to see if the cache line address is contained in it. The sandbox can be thought of as tracking an unordered history of all prefetch addresses the current candidate prefetcher has generated.

Because of the probabilities governing Bloom filters, the size of the Bloom filter is directly related to how many items can be added to it before the false positive rate rises above a desirable level. Because each candidate prefetcher generates only a single prefetch address, we will add a number of items to the Bloom filter equal to the number of L2 accesses in an evaluation period. We experimentally determined that an evaluation period of 256 L2 accesses is optimal for the tested workloads. We chose the size of the Bloom filter to 2048 bits (256 bytes), which for 256 item insertions gives us a maximum expected false positive rate of approximately 1%.

There is only one sandbox per core, and the candidate prefetchers are evaluated one at a time, in a time multiplexed fashion, with the sandbox being reset in between each evaluation. This means there is no opportunity for cross-contamination between mutiple candidate prefetchers sharing a sandbox.

4.3 Candidate Evaluation

Sandbox Prefetching maintains a set of 16 candidate prefetchers, which are evaluated in round-robin fashion. Initially this set of prefetchers is for offsets -8 to -1, and +1 to +8. At the beginning of an evaluation period, the sandbox, the L2 access counter, and the prefetch accuracy score are all reset, along with other counters which track period cache reads, writes, and prefetches, which are used to approximate bandwidth usage.

Each time the L2 cache is accessed, the cache line address is used to check the sandbox to see if this line would have been prefetched by the current candidate prefetcher. If it is a hit, then the prefetch accuracy score is incremented, otherwise, nothing happens. After this, the candidate prefetcher generates a prefetch address, based on the reference cache line address and its own prefetch offset, and adds this address to the sandbox. Finally, the counter that tracks the number of L2 accesses this period is incremented. Once this number reaches 256, the evaluation period is over and the sandbox and other counters are reset, and the evaluation of the next candidate prefetcher begins.

After a complete round of evaluating every candidate prefetcher is over, the bottom 4 prefetchers with the lowest prefetch accuracy score are cycled out, and 4 more offset prefetchers that have not been recently evaluated from the range -16 to +16 are cycled in.

4.4 Prefetch Action

As soon as a candidate prefetcher has finished its evaluation, it may be used to issue real prefetches, if its accuracy score is high enough. In addition to all of the candidate evaluation that is done, each L2 access may result in one or more prefetches to be issued to main memory. We control the number of prefetches that are issued by estimating the amount of bandwidth each core has consumed during its last evaluation period, and then using that to estimate the amount of unused bandwidth available to be used for additional prefetches. Each core in a multi-core setup gets a prefetch degree budget proportional to the number of L2 accesses it performs. This prefetch degree is capped at a minimum of one prefetch per prefetch direction (positive and negative), per core, per L2 access, and at a maximum of eight. The prefetch degree is recalculated at the end of each evaluation period.

Evaluated prefetchers with lower numbered offsets are given preference to issue their prefetches first (and therefore use up some of the prefetch degree budget first). There is an accuracy score cutoff point, below which an evaluated prefetcher will not be allowed to issue any prefetches. Prefetches continue until a number of prefetches equal to the prefetch degree has been issued, and then is repeated for the negative offset prefetchers. The actual offsets of the evaluated prefetchers can change as the less accurate candidate prefetchers are cycled out, so there will need to be some hardware logic to decide the order that evaluated prefetchers will be considered to issue their prefetches. The specific values of the cutoff points will be discussed in the next subsection.

Sandbox Size	2048 bits
Evaluation Period	256 L2 accesses
Total PF Candidates	32
Candidate Offset Ranges	-16 to +16, excluding 0, 16 evaluated per round, then worst 4 cycled out
Candidate Score Storage	16 10-bit counters
Prefetch Accuracy Cutoffs to Issue Multiple Prefetches Per L2 Access	256 (1 PF) 512 (2 PFs) 768 (3 PFs)
Bandwidth Estimation Counters	Read Counter Write Counter Prefetch Counter

Table 1. Sandbox Prefetching parameters and counters.

It is important to keep in mind that there is no additional confirmation before prefetches are issued at this stage. All of the confirmation has already been done globally in the sandbox during the offset prefetcher's evaluation.

4.5 Detecting Streams

So far we have focused on the sandbox's ability to detect the accuracy of offset prefetches, but it can also be used to detect strided access streams. When the sandbox is being probed to see if the current L2 access cache line address could have been prefetched by the current candidate prefetcher, we can also act as though this access is the latest in a strided stream of accesses (where the stride is equal to the offset of the current candidate prefetcher), and test to see if earlier addresses in this strided stream are also found in the sandbox.

For example, if the current candidate prefetcher's offset is +3, whenever we check the sandbox to see if the current cache line address *A* is found in it, we can also check for *A*-3, *A*-6, *A*-9, and so on. If those addresses are also found in the sandbox, then that means that the program is accessing a stream with stride +3. When we were only considering individual offsets that could be prefetched, there was no opportunity to prefetch "ahead," because there was no stream to follow. But now that we can accurately detect strided streams in the access pattern, it makes sense that each candidate prefetcher be allowed to prefetch more than a single line.

We treat the detection of earlier members of a stream in the sandbox the same as we treat the detection of the current access address, by incrementing the sandbox accuracy score for each line found. We probe the sandbox for the current address and the previous three members of the stream, so it's possible that on each L2 access, the prefetch accu-

ISA	UltraSPARC III ISA
CPU configuration	1-4 cores, 3.2 GHz
Core parameters	4-wide out-of-order 128-entry ROB
L1 I-cache	32KB 8-way, private, 4 cycle
L1 D-cache	32KB 8-way, private, 4 cycle
L2 Cache	2-8 MB shared, 12 cycle
Cache line size	64 Bytes
DRAM model	based on USIMM, 12.8 GB/s

Table 2. Simulator parameters.

Single Core Workloads	
401.bzip2, 410.bwaves, 429.mcf, 433.milc, 434.zeusmp, 437.leslie3d, 450.soplex, 459.GemsFDTD, 462.libquantum, 470.lbm, 471.omnetpp, 473.astar, 482.sphinx3, 483.xalancbmk	
Multi Programmed Workloads	
mix1	GemsFDTD, lbm, leslie3d, libquantum
mix2	astar, lbm, libquantum, milc
mix3	astar, milc, soplex, xalancbmk
CloudSuite 1.0 Workloads	
Data Serving, MapReduce, Media Streaming SAT Solver, Web Frontend, Web Search	

Table 3. Evaluation workloads.

racy score may be incremented by up to four. Since there are 256 L2 accesses in an evaluation period, that means the maximum prefetch accuracy score is 1024.

When it is time to issue real prefetches, and an evaluated prefetcher has been found that is not below the accuracy cutoff, then the prefetch accuracy score is examined to see how many prefetches will be issued along the stream that offset prefetcher represents. If the score is greater than 512, then two prefetches along the stream will be done. If the score is greater than 768, then 3 prefetches along the stream will be done. The accuracy cutoff is 256, so if the accuracy score is above this number, but below 512, then only a single prefetch will be done.

4.6 Putting It All Together

We now review the Sandbox Prefetching technique with all its parts as one whole.

There is a set of candidate prefetchers which are evaluated by simulating their prefetch action by adding prefetch addresses to a sandbox Bloom filter, rather than issuing real prefetches, and by testing subsequent cache access addresses to see if they are part of a strided stream. After an entire round of testing candidate prefetchers, the bottom 4 are cycled out to test a broader range of offsets/strides. Each cache access can initiate a number of prefetches, which is based on the amount of available bandwidth. Prefetches are done according to the evalauted prefetchers' prefetch accuracy score, and higher scores mean that more prefetches are issued further down that prefetcher's stream.

5 Methodology

5.1 System Parameters

We evaluate Sandbox Prefetching using the Wind River Simics full system simulator [2], which has been augmented to precisely model a DRAM main memory system by integrating the USIMM DRAM simulator [4]. Each processor core is a 4-wide out-of-order core, using Simics's *ooo-micro-arch* module, with a 128-entry reorder buffer. The parameters of our simulation infrastructure can be seen in Table 2.

We perform both single- and quad-core simulations. Both configurations use 32 KB instruction and data L1 caches, but the single-core simulations use only 2 MB of L2 cache, while the quad-core simulations use a shared 8 MB of L2 cache (the same ratio of 2 MB per core). The L2 cache is the last level of cache, and is inclusive of the L1 caches. The L1 caches use LRU for the cache replacement policy, while the L2 cache uses the PACMan cache replacement policy [23]. Prefetches for all of the tested prefetch methods are performed only at the L2 level. Each core has its own independent prefetching unit, and can have up to 32 simultaneously outstanding prefetches. We do not model an L1 prefetcher of any kind.

Our main memory system is modeled as a single DDR3-1600 memory channel, with up to 12.8 GB/s of memory bandwidth. All DRAM timings, bus and bank contentions, and bandwidth limitations are strictly enforced. The DRAM scheduler is based on first come, first serve, and prioritizes demand read requests over prefetch read requests.

We compare Sandbox Prefetching to two state-of-the-art prefetchers, which were discussed in Section 3, Feedback Directed Prefetching, and Address Map Pattern Matching, as well as a baseline which performs no prefetching. Both FDP and AMPM have been configured according to their respective cited papers.

5.2 Workloads

We evaluate the Sandbox Prefetching method by testing it with a variety of workloads from the SPEC CPU 2006 suite [1]. We selected workloads that exhibit a non-trivial rate of last level cache misses per instruction. Some of these workloads are amenable to regular prefetching, such as *lbm*, *libquantum*, and *milc*. Some of these workloads do not work well with prefetching, such as *mcf*, and *omnetpp*, and are included to show that Sandbox Prefetching does not hurt the performance of applications that are not prefetch-friendly. The list of evaluated applications can be found in Table 3.

We determined the region of program execution to use for our simulations by profiling the last level cache miss rates of each application using our no-prefetching baseline, and then we found a contiguous 500 million instruction region which is representative of overall program execution,

including program phase changes. Once finding these simulation starting points, each experiment was conducted by first warming up the caches for 50 million instructions, and then collecting performance statistics for another 500 million instructions.

We also include results for three mix workloads. These mixes are selected from our single threaded workloads. We chose combinations of applications that would stress the memory bandwidth of the system to different degrees. Our *mix1* workload is comprised of four applications that are all bandwidth intensive. The *mix2* workload is comprised of two bandwidth intensive applications, and two applications of medium bandwidth intensity. Finally, the *mix3* is comprised of four medium bandwidth intensity applications.

Finally, In our sensitivity analysis, Section 6.5, we evaluate 6 workloads from CloudSuite 1.0 [8]. For each CloudSuite application, we began simulation at the beginning of the region of interest. All CloudSuite experiments were conducted using the same 4-core configuration as the SPEC mixes above.

6 Evaluation

6.1 Prefetcher Storage and Logic Requirements

SBP is configured to use a candidate evaluation period of 256 L2 accesses, requiring a sandbox Bloom filter of size 256 B (2048 bits). In addition to this, SBP requires storage to track 16 candidate prefetchers, each requiring 10 bits for accuracy score storage and another 5 bits to store its prefetch offset, and 10 more bytes for various counters. SBP's total storage requirement is 296 B per core. Finally, SBP requires logic that can update counters, update and query the Bloom filter, choose which of the 16 evaluated prefetchers to use, and calculate the prefetch degree, but all of this logic is off the critical path of performance (meaning it is not used in the calculation of prefetch addresses every time the L2 cache is accessed). The only performance-critical logic is used to generate prefetch addresses based on a reference address, and a set of offsets which have been predetermined to have high evaluation scores, which is not unusual for a prefetching mechanism.

FDP is configured to use 2.5 KB of storage per core, which includes bits in the cache tag array to mark prefetched lines, a 4096-bit Bloom filter to track cache pollution (which by itself requires more storage than all of SBP), and more. This quoted storage does not include their baseline 64-entry stream tracking structures, which would add another 600 B. In total, FDP uses 3.1 KB for its prefetching structures, and requires logic which can detect if an address is within an existing stream, allocate new streams, add and remove items from a Bloom filter, and calculate the dynamic settings of the prefetcher based on feedback mechanisms. As with SBP, the more complicated logical components are off the critical path of performance, and the only performance-critical logic calculates the prefetch addresses based on a reference address, the prefetch direction, and prefetch degree.

Sandbox Prefetching	296 B
Feedback Directed Prefetching	3.1 KB
Access Map Pattern Matching	4 KB

Table 4. Prefetcher Storage Overheads

AMPM is configured to use 4 KB of storage per core, most of which is used by the memory access map table, which tracks the use status of every cache line in 850 KB worth of address space using 2-bit counters. In addition to this 4 KB of storage overhead, AMPM requires the ability to pattern match up to 256 stride patterns (up to 512 comparisons of 2-bit numbers) to find prefetch candidates on each L2 access. This would require significantly more logic to do than either FDP or SBP require, and unlike SBP and FDP, AMPM's most complicated logical components *are* on the critical path of performance, and must be invoked every time a prefetch address is generated.

SBP uses considerably less storage than either FDP or AMPM, and its logic requirement is also considerably lower than AMPM, both on and off the critical path of performance.

6.2 Performance

6.2.1 Single Core

Figure 3 shows the performance, measured in IPC normalized to the performance of the no-prefetching baseline (No PF) of our four test configurations. In the single-threaded workloads where AMPM sees its largest performance improvements over No PF, SBP is able to consistently achieve even higher performance. SBP improves upon the performance of AMPM by the greatest amount in GemsFDTD (5.0%), lbm (5.1%), leslie3d (6.8%), milc (7.2%), and sphinx3 (4.6%). On the other hand, AMPM is able to achieve 3.8% higher performance than SBP in bwaves. Overall, SBP's average improvement compared to AMPM across all single-threaded workloads is 2.4%. SBP accomplishes this using only a small fraction of the storage overhead and number of comparison operations per L2 access that AMPM uses to achieve its result.

Compared to FDP, SBP improves performance across single threaded workloads by an average of 18.7%, with a maximum of 68.8% improvement in the lbm workload.

6.2.2 Multi-Core

For the single-programmed workloads, SBP has an entire 12.8 GB/s memory channel to itself, and is able to aggressively prefetch without worrying about bandwidth limitations. For the multi-core situation, we modified SBP's parameters slightly to make it more conservative in prefetching, and conserve bandwidth. This was accomplished by

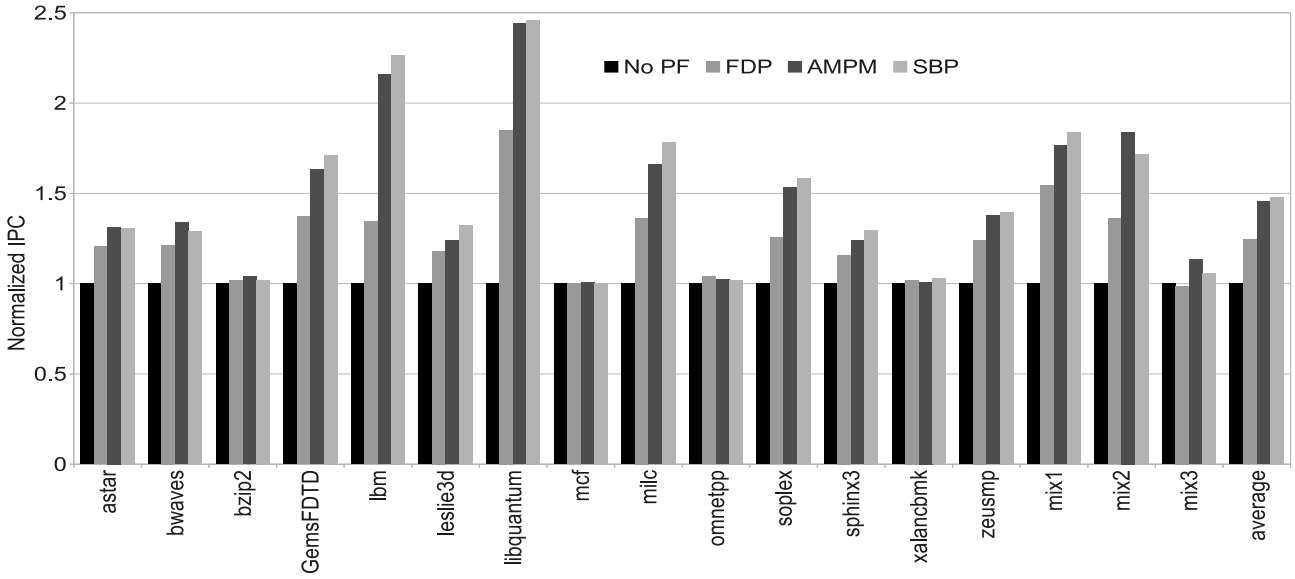


Figure 3. Performance normalized to the no-prefetching baseline.

increasing the accuracy cutoff limit by 50%, meaning that evaluated prefetchers would need more hits in the sandbox Bloom filter before they could qualify to issue real prefetches. This could be a setup-time configuration option which could be set before loading all cores with applications with high bandwidth requirements.

In multi-programmed mix workloads, SBP is able to improve upon the performance of AMPM by 3.9% in mix1, but has lower performance in mix2, and mix3, 6.6% and 6.5% lower, respectively. This performance loss is due to SBP's prefetching still being too aggressive, even after setting a higher accuracy cutoff limit. However, compared to FDP, SBP's performance is strictly higher, averaging 19.2% better, with a maximum of 26.0% improvement in mix3. Again, with lower storage and logic requirements, SBP is able to nearly meet or beat AMPM, and is strictly better than FDP.

6.3 DRAM Channel Usage

Prefetching always increases memory bandwidth usage. If the prefetching technique is effective at reducing cache misses, it will increase IPC and therefore the rate at which read requests are sent to DRAM. Even if the prefetching technique is not effective at accurately predicting what data will be used by the processor next, it will still increase bandwidth, this time by sending superfluous prefetch requests to the DRAM.

Figure 4 shows how much bandwidth was used by each test configuration for each workload. SBP usually uses the most bandwidth of any tested prefetch technique. This is often because SBP has the highest performance, but in cases like bwaves or two of the mix workloads, SBP uses the most bandwidth, but does not have the highest performance.

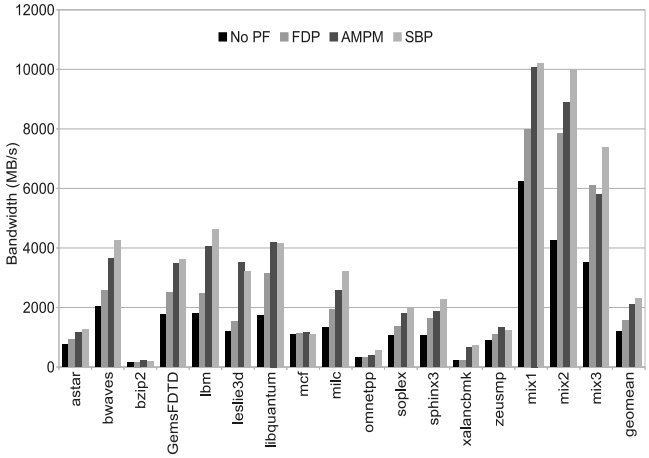


Figure 4. Bandwidth in MB/s. Bandwidth increases as prefetchers become more aggressive, regardless of whether the prefetches are fruitful or not.

Memory bandwidth is a measure of data being transferred per unit time, such as megabytes per second, and is separate from the measure of how much data is transferred across the main memory bus per instruction executed. Figure 5 shows the relative number of Bus Transactions per thousand instructions (BPKI) that were used during each benchmark run, normalized to the BPKI of the no-prefetch baseline. Reads, writes, and prefetches all count toward the number of bus transactions that a benchmark configuration uses.

In this figure, a number close to 1.0 means that the prefetcher did not issue many superfluous prefetches, and

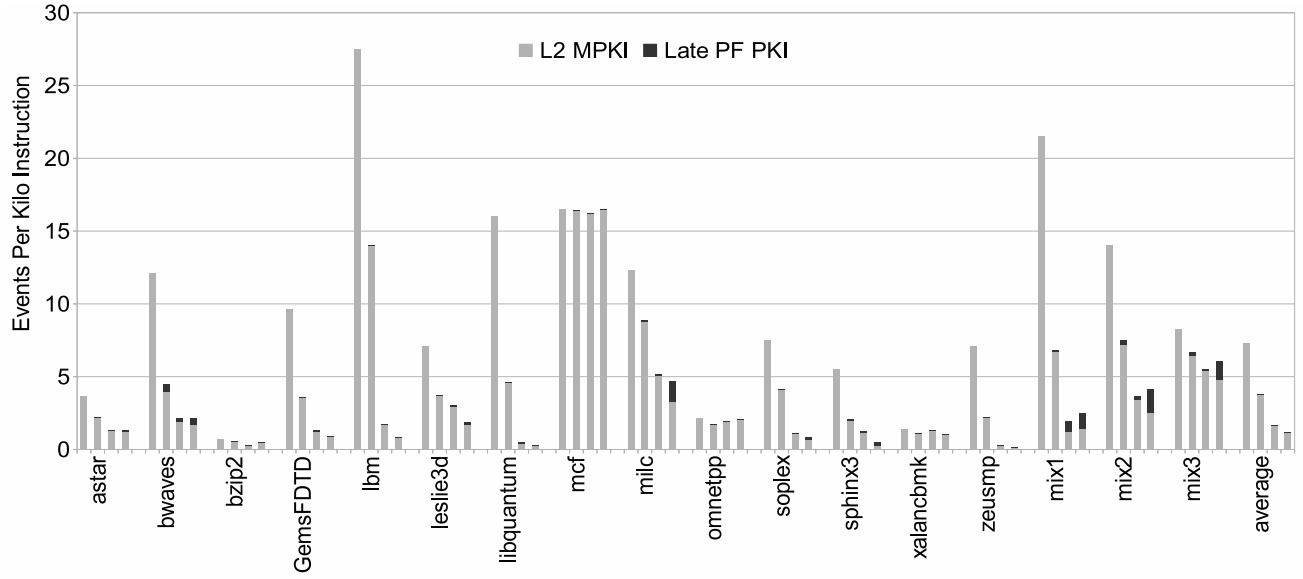


Figure 6. The number of L2 cache misses and late prefetch accesses per thousand instructions. For each workload, the columns are ordered No PF, FDP, AMPM, and SBP.

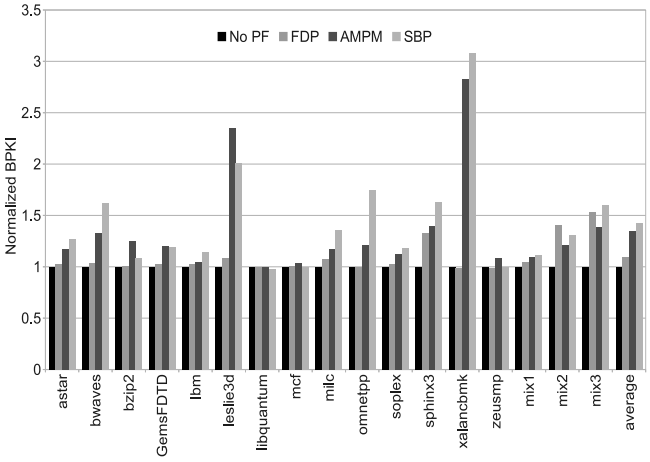


Figure 5. The number of bus transactions per thousand instructions. Bus transactions include DRAM reads, writes, and prefetches.

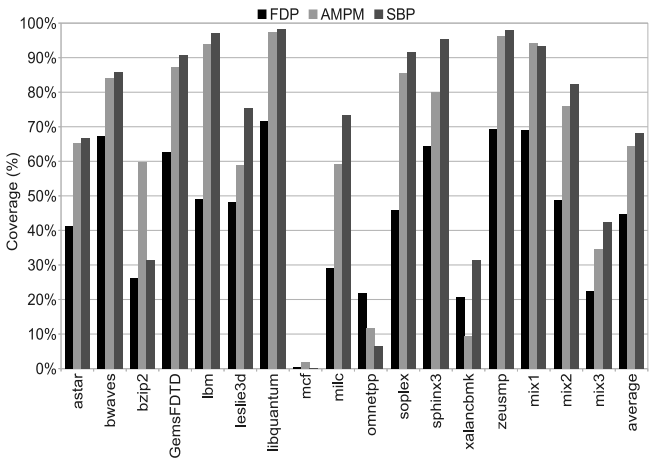


Figure 7. The percent reduction in last level cache misses, compared to the no-prefetching baseline.

that most of the data that was prefetched was consumed by the program. A number much greater than 1.0 signifies that the system configuration generated many unnecessary prefetches, at least wasting memory bandwidth, but possibly also polluting the cache and requiring useful data to be re-fetched from main memory.

Three examples from these two figures highlight important prefetcher behaviors. First, mcf does not see an increase in BPKI, and neither does it see an increase in bandwidth. This is because no prefetcher issues very many prefetches (successful or otherwise) for this workload. Second, libquantum sees a large increase in bandwidth usage, but no increase in BPKI. This is because the prefetchers do large amounts of prefetches, which are almost all fruitful, and would have been fetched by the no-prefetch baseline anyway. Finally, xalancbmk does not see a significant improvement in performance from prefetching, but both AMPM and SBP consume extra memory bandwidth, and greatly increase BPKI. This is because most of the prefetches issued by these prefetchers are unfruitful and pollute the cache, and the extra memory bandwidth usage is just wasted.

6.4 L2 Misses and Prefetcher Coverage

The primary goal of prefetching is to reduce average data access latency by placing data into the cache that otherwise would not be there yet. This is clearly observable when

looking at the last level cache miss rates, as in Figure 6. This figure shows both the L2 cache miss rate per thousand instructions (MPKI), as well as how many late prefetches were seen per thousand instructions.

A late prefetch is defined as a prefetch that has already been issued to the main memory, but before the prefetch data was returned from main memory, the program issued a load for that data. We do not differentiate in this figure between late prefetches that hide only a small amount of the cache miss latency, or late prefetches that hide the majority of the cache miss latency.

SBP sees more late prefetches than either FDP or AMPM, especially for the multi-core workloads. In the single core workloads, even if we were to consider a late prefetch as no better than an outright cache miss, then SBP would still have fewer cache misses than either FDP or AMPM.

Figure 7 shows the prefetch coverage rates for each prefetch technique for each workload. Prefetch coverage is defined as the percentage of cache misses that were present in the no-prefetching baseline that the prefetching technique is able to anticipate and prefetch.

SBP generally has the highest prefetch coverage rate. Looking at the bandwidth usage and BPKI statistics, it might not be surprising that the prefetcher that aggressively uses the most bandwidth and bus transactions also has the highest prefetch coverage, but it is important to note that FDP and AMPM could have used more bandwidth, but their prefetching mechanisms did not identify sufficient opportunities to issue additional useful prefetches, while SBP did.

Prefetch coverage is low for SBP in the bzip2 benchmark. Looking at the IPC, bandwidth, and BPKI metrics shows that SBP was not effective at finding many opportunities for useful prefetches in this workload. Prefetch coverage is also low for SBP in omnetpp. Looking at the other metrics in this case shows that many prefetches are being issued, they just aren't very effective at covering cache misses.

6.5 Sensitivity Analysis

The above results were gathered using the system parameters described in Section 5.1. We now show results for varying the candidate prefetch evaluation period for SBP, and the last level cache size, and for CloudSuite 1.0 workloads.

6.5.1 Candidate Prefetcher Evaluation Period

We tested Sandbox Prefetching using several different durations for the candidate prefetcher evaluation period, as seen in Figure 8. As we varied the evaluation period duration, we also changed the size of the Bloom filter (and therefore storage overhead requirement), to maintain the same expected false positive rate (1%).

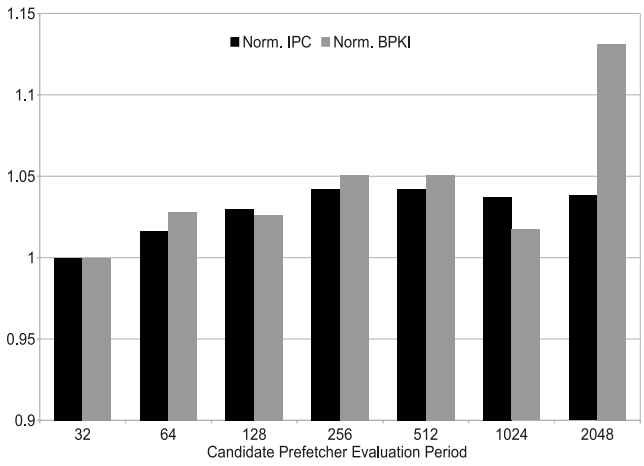


Figure 8. Effect of candidate prefetcher evaluation period length on IPC and BPKI.

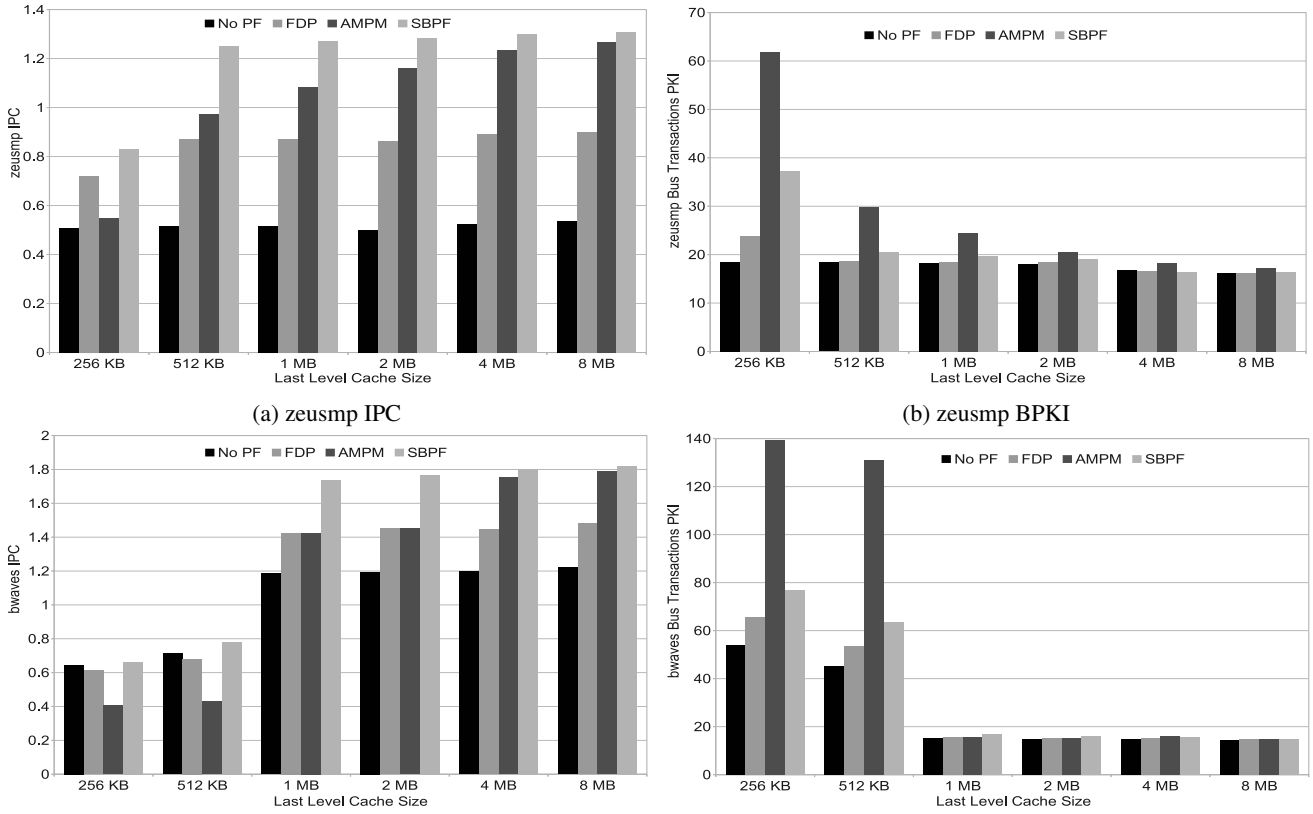
Starting from a duration of 32 L2 accesses, as you increase the candidate prefetcher evaluation period, performance increases because each candidate prefetcher under evaluation has a longer period of time in which to build up its score, because each simulated prefetch that is added to the sandbox has a longer time in which it might be later accessed. Shorter durations might miss out on some accurate prefetchers because they end the evaluation and move on to the next candidate too quickly, before the simulated prefetches can be accessed and the candidate prefetcher can earn a high accuracy score. However, if the evaluation period is too long, then too many prefetchers will rise above the accuracy score cutoff, and there will be too much bandwidth use and cache pollution, and performance suffers.

The maximum for IPC is seen when the evaluation period is set to 256. Any more or less than this and performance degrades.

6.5.2 L2 cache size

For our evaluations we have so far used an industry-typical 2 MB of last level cache per core, and now we investigate what happens if we vary the amount of last level cache. We show results for zeusmp, which in the baseline 2 MB configuration had SBP outperforming AMPM, and bwaves, which in the baseline 2 MB configuration had AMPM outperforming SBP. However, our simulation duration for this experiment was lower than we used for the other performance results (50 million instructions versus 500 million instructions, both after a 50 million instruction warmup), and this simulation window is focused in a region of the highest L2 cache misses for the No PF baseline for these workloads.

We tested for cache sizes ranging from 256 KB to 8 MB (all for a single core), as seen in Figure 9. The baseline no-prefetching configuration sees very little performance ben-



(a) zeusmp IPC

(b) zeusmp BPKI

(c) bwaves IPC

(d) bwaves BPKI

Figure 9. Effect on IPC and BPKI caused by varying the last level cache size.

efit as cache size increases for zeusmp, but its performance nearly doubles for bwaves when the cache size is increased from 512 KB to 1 MB. For both zeusmp and bwaves, when large 4 MB or 8 MB caches are used, SBP and AMPM performance and BPKI are both very similar. For lower cache sizes the differences between the behaviors of AMPM and SBP become more apparent.

In zeusmp, the performance of AMPM gradually increases each time the cache capacity is doubled. SBP, on the other hand, realizes almost all of its performance potential with only 512 KB of cache. AMPM is reliant on high cache capacity to achieve high performance, while SBP is able to deliver the same performance using a 512 KB cache that AMPM can only deliver when using a 4 MB cache.

In addition to having much lower performance, AMPM is also very poorly behaved in zeusmp when using small caches. While AMPM has higher BPKI at every cache size, it is not until cache size reaches 2 MB that its BPKI comes in line with the other prefetch techniques. SBP also has high BPKI when using a 256 KB cache in zeusmp, but this is at least coupled with the highest performance seen for that cache size.

In bwaves, AMPM is even worse behaved for small caches than it was in zeusmp. Performance for AMPM is lower than the no-prefetching baseline for 256 KB and 512 KB caches, and BPKI is nearly tripled. Using a 1 MB or 2 MB cache yields nearly identical results for AMPM, and its performance doesn't increase again until using a 4 MB cache, where it again plateaus. On the other hand, SBP is able to deliver the same performance using a 1 MB cache that AMPM can only deliver when using a 4 MB cache.

6.5.3 CloudSuite 1.0

Prefetching was not effective for some of the six tested CloudSuite workloads. Of these workloads, Media Streaming sees the largest benefit from prefetching, with AMPM improving performance by 28.6%, and SBP improving performance by only 22.4%, compared to No PF. Two other workloads, MapReduce and SAT Solver, see no benefit from AMPM, but SBP improves their performance by 2.4% and 4.3%, respectively. The average performance improvement across the six tested CloudSuite workloads is 6.9% for FDP, 8.3% for AMPM, and 8.0% for SBP, compared to the No PF baseline.

6.6 Performance Review

We will now summarize the results from this section. SBP uses only 9.5% of the storage overhead of FDP, and

only 7.2% of the storage overhead of AMPM, and it also has a much lower logical complexity requirement compared to AMPM. SBP improves upon the performance of FDP by an average of 18.7% in both single and multi-threaded workloads. Although having much lower storage and logical overheads, SBP improves on the performance of AMPM by 2.4% in our evaluated single-threaded workloads, and by 1.4% across all evaluated workloads. We also found that SBP is able to achieve high performance when using a much smaller cache than AMPM requires to achieve the same performance.

7 Conclusions

Modern high performance processors employ a variety of hardware prefetching techniques to mitigate the impact of long memory latencies. We have proposed Sandbox Prefetching, a new mechanism to evaluate the effectiveness of candidate immediate prefetchers in the global context of every memory access a program uses, before they are deployed in the real memory hierarchy. This evaluation is done by simulating prefetches in a Bloom filter-based Sandbox, which avoids wasting real cache space and memory bandwidth on prefetches that have not yet been determined to likely be effective. This mechanism can detect strides in the memory access pattern, as well as fixed offsets which can be accurately prefetched.

Sandbox Prefetching improves performance across a set of memory-intensive SPEC CPU 2006 benchmarks by 47.6% compared to not using any prefetching, and by 18.7% compared to the Feedback Directed Prefetching technique. Performance is also improved by 1.4% compared to the Access Map Pattern Matching Prefetcher, while using considerably less logic or storage overheads.

Acknowledgments

We thank the anonymous reviewers for their many useful suggestions. This work was supported in part by NSF grant CNS-1302663.

References

[1] Standard Performance Evaluation Corporation CPU2006 Benchmark Suite. `http://www.spec.org/cpu2006/`.

[2] Wind River Simics Full System Simulator. `http://www.windriver.com/products/simics/`.

[3] J. Baer and T. Chen. An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty. In *Proceedings of Supercomputing*, 1991.

[4] N. Chatterjee, R. Balasubramonian, M. Shevgoor, S. Pugsley, A. Udipi, A. Shafiee, K. Sudan, M. Awasthi, and Z. Chishti. USIMM: the Utah SImulated Memory Module. Technical report, University of Utah, 2012. UUCS-12-002.

[5] F. Dahlgren, M. Dubois, and P. Stenstrom. Sequential Hardware Prefetching in Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1995.

[6] F. Dahlgren and P. Stenstrom. Evaluation of Hardware-Based Stride and Sequential Prefetching in Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(4):385–395, April 1999.

[7] E. Ebrahimi, O. Mutlu, and Y. Patt. Techniques for Bandwidth-Efficient Prefetching of Linked Data Structures in Hybrid Prefetching Systems. In *Proceedings of HPCA*, 2009.

[8] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *Proceedings of ASPLOS*, 2012.

[9] J. Fu, J. Patel, and B. Janssens. Stride Directed Prefetching in Scalar Processors. In *Proceedings of MICRO-25*, pages 102–110, December 1992.

[10] I. Hur and C. Lin. Memory Prefetching Using Adaptive Stream Detection. In *Proceedings of MICRO*, 2006.

[11] S. Iacobovici, L. Spracklen, S. Kadambi, Y. Chou, and S. Abraham. Effective Stream-Based and Execution-Based Data Prefetching. In *Proceedings of ICS*, 2004.

[12] Y. Ishii, M. Inaba, and K. Hiraki. Access Map Pattern Matching for High Performance Data Cache Prefetch. *The Journal of Instruction-Level Parallelism*, 13, January 2011.

[13] V. Jimenez, R. Gioiosa, F. Cazorla, A. Buyuktosunoglu, P. Bose, and F. O'Connell. Making Data Prefetch Smarter: Adaptive Prefetching on POWER7. In *Proceedings of PACT*, 2012.

[14] D. Joseph and D. Grunwald. Prefetching Using Markov Predictors. In *Proceedings of ISCA*, 1997.

[15] N. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of ISCA-17*, pages 364–373, May 1990.

[16] D. E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1. Addison-Wesley, third edition, 1997.

[17] S. Kumar and C. Wilkerson. Exploiting Spatial Locality in Data Caches Using Spatial Footprints. In *Proceedings of ISCA*, 1998.

[18] K. Nesbit, A. Dhodapkar, and J. Smith. AC/DC: An Adaptive Data Cache Prefetcher. In *Proceedings of PACT*, 2004.

[19] S. Palacharla and R. Kessler. Evaluating Stream Buffers as a Secondary Cache Replacement. In *Proceedings of ISCA-21*, pages 24–33, April 1994.

[20] A. Roth, A. Moshovos, and G. Sohi. Dependence Based Prefetching for Linked Data Structures. In *Proceedings of ASPLOS VIII*, pages 115–126, October 1998.

[21] S. Srinath, O. Mutlu, H. Kim, and Y. Patt. Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers. In *Proceedings of HPCA*, 2007.

[22] J. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. POWER4 System Microarchitecture. In *IBM Technical Whitepaper*, Oct. 2001.

[23] C. Wu, A. Jaleel, M. Martonosi, S. Steely, and J. Emer. PACMan: Prefetch-Aware Cache Management for High Performance Caching. In *Proceedings of MICRO-44*, 2011.