Best-Offset Hardware Prefetching

Pierre Michaud

Best-Offset Hardware Prefetching

Pierre Michaud

Inria
Campus de Beaulieu, Rennes, France
pierre.michaud@inria.fr

ABSTRACT

Hardware prefetching is an important feature of modern high-performance processors. When the application working set is too large to fit in on-chip caches, disabling hardware prefetchers may result in severe performance reduction. A new prefetcher was recently introduced, the Sandbox prefetcher, that tries to find dynamically the best prefetch offset using the sandbox method. The Sandbox prefetcher uses simple hardware and was shown to be quite effective. However, the sandbox method does not take into account prefetch timeliness. We propose an offset prefetcher with a new method for selecting the prefetch offset that takes into account prefetch timeliness. We show that our Best-Offset prefetcher outperforms the Sandbox prefetcher on the SPEC CPU2006 benchmarks, with equally simple hardware.

1. INTRODUCTION

Hardware prefetching is an important feature of modern high-performance processors. When the application working set is too large to fit in on-chip caches, disabling hardware prefetchers may result in severe performance loss.

Hardware prefetchers try to exploit certain patterns in applications memory accesses. However, there exists a wide diversity of applications and memory patterns, and many different ways to exploit these patterns.

The simplest hardware prefetchers exploit simple memory access patterns, in particular spatial locality and constant strides. Although simple prefetchers do not perform well on all applications, this kind of access pattern occur frequently, and good prefetchers must perform well on these patterns.

Recently, Pugsley et al. introduced a new sort of prefetcher, *offset* prefetchers, and the sandbox method for selecting the prefetch offset dynamically [26]. Offset prefetching is a generalization of next-line prefetching. Unlike a stream prefetcher, an offset prefetcher does not try to detect streams. Pugsley et al.'s offset prefetcher, SBP, requires simple hardware and is very effective on applications having characteristics similar to the SPEC CPU2006 benchmarks. However, the sandbox method does not take into account prefetch timeliness. Issuing accurate prefetches helps only if prefetches hide a substantial fraction of the miss latency.

The Best-Offset (BO) prefetcher we propose is an offset prefetcher with a new method for selecting the prefetch offset, taking into account prefetch timeliness. We show that the BO prefetcher provides significant speedups over SBP on the SPEC CPU2006, with equally simple hardware[1].

The paper is organized as follows. Section 2 discusses related work. Section 3 illustrates offset prefetching with some examples. Section 4 describes the proposed Best-Offset prefetcher. Section 5 gives a detailed description of our baseline microarchitecture, with experimental results. Section 6 presents an experimental evaluation of the BO prefetcher. Finally, Section 7 concludes this study.

2. BACKGROUND

The idea of prefetching is as old as caches, which operate on lines, i.e., blocks of contiguous memory locations, without the guarantee that all the information in a block will actually be used by the program. However, what is called *prefetching* in processors is usually the speculative fetching of cache lines that have not yet been requested by the program. Prefetching has been studied since the 1970's (see [33] for early references).

We consider hardware-only prefetching in this study. Prefetching can be done at any cache level. Level-1 (L1) and level-2 (L2) prefetching lead to different possibilities and tradeoffs, hence different sorts of prefetchers. An L1 prefetcher can use some informations that it would be somewhat costly to propagate to the L2 cache, such as load/store PCs, virtual addresses, and program order. On the other hand, L1 caches have stronger capacity and bandwidth constraints than L2/L3 caches. L1 caches do not tolerate inaccurate prefetches, while L2/L3 caches do to a certain extent. Best-Offset prefetching is intended for L2 prefetching.

One of the simplest prefetching method, next-line prefetching, prefetches line $X+1$ when line X is accessed. A prefetch bit may be added to each cache line to reduce useless prefetches [33]. So and Rechtschaffen proposed to use cache replacement status instead of a prefetch bit [34].

Stride prefetchers try to identify, among load and store instructions, those that access memory with a constant stride [1, 7, 31]. Stride prefetchers usually have a table indexed with load/store PCs. An important feature of stride prefetchers is that they issue a prefetch request only if there is a certain confidence that the prefetch will be useful. However, stride prefetchers are more easily implemented at the L1, as they need to see all the memory instructions, including those that hit in the DL1, and preferably in program order.

[1] A tuned version of the BO prefetcher won the 2015 Data Prefetching Championship [6].

Stream prefetching was introduced by Jouppi [15]. It exploits sequential streams, like next-line prefetching, but tries to improve prefetch timeliness and decrease useless prefetches by prefetching several successive lines into a *stream buffer* when a stream has been detected. Only when a demand access hits on the stream buffer head is the prefetched line moved into the cache (hence reducing cache pollution), and a new line is prefetched to keep the stream buffer full. Several stream buffers are needed for good performance on interleaved streams [15]. Palacharla and Kessler proposed an off-chip stream prefetcher that can prefetch non-unit stride accesses, each stream being identified by the memory region it is accessing [24].

Hagersten proposed ROT, a sophisticated stream prefetcher that can prefetch with non-unit strides [8]. ROT detects streams by maintaining a list of popular strides and a list of recent miss addresses, checking for each recent miss if its distance to the current access equals one of the popular strides. The prefetch stream depth is increased dynamically upon late prefetches. The list of popular strides is updated by comparing the current address with recent miss addresses, incrementing a score for each candidate stride. Candidate strides are promoted to popular strides if their score exceeds a threshold.

A variant of stream prefetching that does not use stream buffers prefetches N consecutive lines at a time (N is the prefetch *degree*) at a certain prefetch distance D in the stream. Some authors have proposed to vary N and/or D dynamically by monitoring prefetch accuracy, cache pollution, prefetch timeliness, which depend on applications characteristics [4, 9, 37]. Hur and Lin have proposed a method for making stream prefetching efficient on short streams [9].

The prefetchers mentioned above exploit simple memory access patterns, such as sequential accesses or constant strides. Many prefetchers have been proposed that try to exploit more complex access patterns. For instance, some prefetchers record in a table some history about past memory accesses and use that history to predict future memory accesses [14, 16, 23, 22, 36, 2, 35, 11, 12, 17] (this list is not exhaustive).

Recently, Pugsley et al. introduced Sandbox prefetching [26]. The Sandbox prefetcher prefetches line $X+D$ when line X is requested at the L2, where D is adjusted dynamically by performing "fake" prefetches in a Bloom filter. This is not a stream prefetcher, but what Pugsley et al. call an *offset* prefetcher. A fundamental difference between offset prefetchers and stream prefetchers is that offset prefetchers do not try to detect streams. Pugsley et al. show that, on the SPEC CPU2006, the Sandbox prefetcher matches or even slightly outperforms the more complex AMPM prefetcher that won the 2009 Data Prefetching Championship [11, 5].

The Sandbox prefetcher, however, does not take into account prefetch timeliness. The Best-Offset prefetcher we propose is an offset prefetcher that takes into account prefetch timeliness.

3. OFFSET PREFETCHING

Offset prefetching is a generalization of next-line prefetching [33]. When a line of address X is requested by the core, an offset prefetcher prefetches line $X+D$, where D is the *prefetch offset*. The case $D=1$ corresponds to next-line prefetching.

The optimal offset value is not the same for all applications. A full-fledged offset prefetcher has an offset selection mechanism for setting dynamically the offset depending on application behavior. To the best of our knowledge, the first published full-fledged offset prefetcher is the Sandbox prefetcher by Pugsley et al. [26]. However, the offset selection mechanism in the Sandbox prefetcher ignores prefetch timeliness. The Best-Offset (BO) prefetcher we introduce takes into account prefetch timeliness.

This section provides some examples illustrating why offset prefetching is an effective prefetching technique. The following examples assume 64 byte lines. For convenience, lines accessed in a memory region are represented with a bit vector, adjacent bits representing adjacent lines. The bit value tells whether the line is accessed ("1") or not ("0"). We ignore the impact of page boundaries and consider only the steady state on long access streams.

3.1 Example 1: sequential stream

Consider the following sequential stream:

```
11111111111111111...
```

That is, the lines accessed by the program are X, X+1, X+2, and so on. A next-line prefetcher yields 100% prefetch coverage and accuracy on this example. However, issuing a prefetch for X+1 just after the access to X might be too late to cover the full latency of fetching X+1 from the last-level cache or from memory, leading to a *late prefetch*. Late prefetches may accelerate the execution, but not as much as timely prefetches. An offset prefetcher yields 100% prefetch coverage and accuracy on sequential streams, like a next-line prefetcher, but can provide timely prefetches if the offset is large enough.

Another factor that may degrade prefetch coverage is *scrambling*, i.e., the fact that the chronological order of memory accesses may not match the program order exactly [11]. In general, on long sequential streams, tolerance to scrambling improves with larger offsets.

3.2 Example 2: strided stream

Consider a load instruction accessing an array with a constant stride of +96 bytes. With 64-byte cache lines, the lines accessed in a memory region are:

```
110110110110110110...
```

If there is no stride prefetcher at the L1 (or if it issues late prefetches), a delta correlation prefetcher observing L2 accesses (such as AC/DC [22]) would work perfectly here, as the sequence of line strides is periodic (1,2,1,2,...). Still, a simple offset prefetcher with a multiple of 3 as offset yields 100% coverage and accuracy on this example.

Offset prefetching can in theory provide 100% coverage and accuracy on any periodic sequence of line strides, by setting the offset equal to the sum of the strides in a period, or to a multiple of that number.

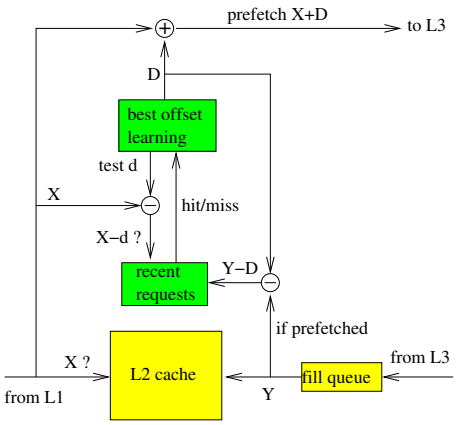

Figure 1: Schematic view of a BO prefetcher.

3.3 Example 3: interleaved streams

Consider two interleaved streams S1 and S2 accessing different memory regions and having different behaviors:

```
S1: 101010101010101010...
S2: 110110110110110110...
```

Stream S1 alone can be prefetched perfectly with a multiple of 2 as offset. Stream S2 alone can be prefetched perfectly with a multiple of 3 as offset. Both streams can be prefetched perfectly with a multiple of 6 as offset.

4. BEST-OFFSET (BO) PREFETCHING

A schematic view of a BO prefetcher is shown in Figure 1. Symbol D in Figure 1 represents the current *prefetch offset*, which is the offset currently used for prefetching. When a read request for line X accesses the L2 cache, if this is a miss or a prefetched hit (i.e., the prefetch bit is set), and if X and $X+D$ lie in the same memory page, a prefetch request for line $X+D$ is sent to the L3 cache.

4.1 Best-offset learning

The prefetch offset D is set automatically and dynamically, trying to adapt to the application behavior, which may vary over time.

The best-offset learning algorithm tries to find the best prefetch offset by testing several different offsets. An offset d is potentially a good prefetch offset if, when line X is accessed, there was in the recent past a previous access for line $X-d$. However, the fact that $X-d$ was recently accessed is not sufficient for guaranteeing that line X would have been prefetched in time. We want prefetches to be timely whenever possible. I.e., for d to be a good prefetch offset for line X, line $X-d$ must have been accessed recently, but not too recently. Ideally, the time between the accesses to lines $X-d$ and X should be greater than the latency for completing a prefetch request.

Our solution is to record in a *recent requests* (RR) table the *base address* of prefetch requests that have been completed. The base address is the address that was used to trigger the prefetch request: if the prefetched line is $X+D$, the base address is X. The base address is obtained by subtracting the current prefetch offset from the address of the prefetched line inserted into the L2[2].

If line $X-d$ is in the RR table, it means that a prefetch request for line $X-d+D$ was recently issued and has been completed. Therefore, if a prefetch request had been issued with offset d instead of D, it would have been a prefetch for the line X currently accessed, and this prefetch would have been timely (assuming that the latency of fetching line X equals the latency of fetching line $X-d+D$).

Several implementions are possible for the RR table. In this study, we choose the simplest implementation: the RR table is direct mapped, accessed through a hash function, each table entry holding a tag. The tag does not need to be the full address, a partial tag is sufficient.

Besides the RR table, the BO prefetcher features an *offset list* and a *score* table. The score table associates a score with every offset in the offset list. The score value is between 0 and SCOREMAX (for instance, SCOREMAX=31 means 5-bit scores).

The prefetch offset is updated at the end of every *learning phase*. A learning phase consists of several *rounds*. At the start of a learning phase, all the scores are reset to 0. On every eligible L2 read access (miss or prefetched hit), we test an offset d_i from the list. If $X-d_i$ hits in the RR table, the score of offset d_i is incremented. During a round, each offset in the list is tested once: we test d_1 on the first access in the round, d_2 on the next access, then d_3, and so on. When all the offsets in the list have been tested, the current round is finished, and a new round begins from offset d_1 again.

The current learning phase finishes at the end of a round when either of the two following events happens first: one of the scores equals SCOREMAX, or the number of rounds equals ROUNDMAX (a fixed parameter). When the learning phase is finished, we search the best offset, i.e., the one with the highest score[3]. This offset becomes the new prefetch offset, and a new learning phase starts.

4.2 Offset list

Nothing prevents a BO prefetcher to use negative offset values. Although some applications might benefit from negative offsets, we did not observe any benefit in our experiments. Hence we consider only positive offsets in this study.

Useful offset values depend on the memory page size, as the BO prefetcher does not prefetch across page boundaries. For instance, assuming 4KB pages and 64B lines, a page contains 64 lines, and there is no point in considering offset values greater than 63. However, it may be useful to consider offsets greater than 63 for systems having superpages.

The choice of offsets to include in the offset list is somewhat arbitrary. For instance, a possibility would be to include all the offsets between 1 and a maximum offset. This maximum offset cannot be too large however, as a big offset list means a big score table and a long learning phase. If we want the list to contain large offsets without the list being too big,

[2]If the prefetch address and the base address lie in different memory pages, the actual base address is unknown, and the RR table is not written.

[3]In practice, the best offset and best score can be computed incrementally every time a score is updated.

we must sample the offsets between 1 and the maximum offset (which requires to implement the offset list as a ROM).

Microarchitects working with thousands of representative benchmarks might want to do an extensive exploration of offsets to put in the list. We do not have enough benchmarks for such exploration to make sense[4].

We propose a method for offset sampling that is algorithmic and not totally arbitrary: we include in our list all the offsets between 1 and 256 whose prime factorization does not contain primes greater than 5. This gives the following list of 52 offsets:

1 2 3 4 5 6 8 9 10 12 15 16 18 20 24 25 27 30 32 36 40 45 48 50 54 60 64 72 75 80 81 90 96 100 108 120 125 128 135 144 150 160 162 180 192 200 216 225 240 243 250 256.

Considering only offsets with small prime factors has two benefits:

- Small offsets are more represented than large ones (small offsets are more likely to be useful).
- The offset list is much smaller than the full offset range.

Moreover, this method is consistent with the example of Section 3.3: if two offsets are in the list, so is their least common multiple (provided it is not too large).

4.3 Prefetch throttling

The BO prefetcher is a degree-one prefetcher: it issues at most one prefetch per access.

One could imagine an offset prefetcher with a prefetch degree greater than one. For instance, a degree-two offset prefetcher would prefetch with two different offsets simultaneously, the best and second best offsets. This might bring some extra performance on applications with irregular access patterns. However, this would increase the number of prefetch requests, putting more pressure on memory bandwidth and cache tags bandwidth[5], unless a prefetch filter is implemented. There is no need for a prefetch filter with a degree-one BO prefetcher. Moreover, prefetching with two offsets may generate many useless prefetches on irregular memory access patterns.

Still, BO prefetching is relatively aggressive compared to some other prefetching methods such as stride prefetching. The useless prefetches issued on irregular access patterns waste energy and memory bandwidth.

The best score obtained at the end of a learning phase gives some information about prefetch accuracy. If the score is very low, it probably means that offset prefetching fails, and we may decide to turn prefetch off.

We define a fixed threshold, BADSCORE, such that when the best score is not greater than BADSCORE, prefetch is turned off. However, best-offset learning never ceases, it continues even when prefetch is off, so that prefetch can be turned on again whenever the application behavior changes and requires prefetching.

Figure 1 illustrates the case where prefetch is on: for every *prefetched* line Y inserted into the L2, we write address $Y-D$ into the RR table (if Y and $Y-D$ lie in the same page). During learning phases when prefetch is off, insertion in the RR table is modified: for every *fetched* line Y, we write address Y in the RR table (i.e., $D=0$).

[4]This would be a case of overfitting.

[5]With two different prefetch offsets D_1 and D_2, redundant prefetch requests are issued when accessing lines X and $X+D_1-D_2$.

4.4 Implementation details

The BO prefetcher features 3 adders, shown in Figure 1. These adders need only produce the position of a line inside a page. For instance, with 4MB pages and 64B lines, each adder is $22-6=16$ bit wide. The page number bits are simply copied from the base address X or from the prefetched line address Y (cf. Figure 1).

The RR table is accessed through a simple hash function. For instance, for a 256-entry RR table, we XOR the 8 least significant line address bits with the next 8 bits to obtain the table index. For 12-bit tags, we skip the 8 least significant line address bits and extract the next 12 bits.

5. BASELINE MICROARCHITECTURE

The microarchitecture simulator used for this study is an in-house simulator based on Pin [20]. The operating system activity is not simulated. The simulator is trace driven and does not simulate the effects of wrong-path instructions.

The benchmarks used for this study are the SPEC CPU 2006. A trace is generated for each benchmark. Each trace consists of 20 samples stitched together. Each sample represents 50 million instructions executed consecutively. The samples are spaced regularly throughout the whole benchmark execution. In total, 1 billion instructions are simulated per benchmark.

Unlike a branch predictor or a cache replacement policy, a hardware prefetcher cannot be evaluated as a stand-alone mechanism, as it interacts with other parts of the microarchitecture in a very complex way. This is one of the reasons why it is sometimes difficult to reproduce published experimental results on prefetching [25].

This section provides a detailed description of our baseline microarchitecture, with a focus on parts that directly impact prefetching effectiveness.

Table 1 summarizes some of the characteristics of our baseline microarchitecture. The superscalar core is loosely modeled after the Haswell microarchitecture, based on information publicly released by Intel [10].

We use 6 variations of the baseline, corresponding to 1,2 and 4 active cores and 2 different memory page sizes (4KB and 4MB). Unless specified otherwise, the simulated microarchitecture parameters are those of Table 1.

5.1 Baseline IPCs

Our baseline microarchitecture is a quad-core with private L2 caches and a shared L3. The caches are non inclusive. We assume a fixed clock frequency, i.e., we do not simulate dynamic voltage and frequency scaling.

This study focuses on single-thread performance. All the performance numbers presented in this paper are from the point of view of a single-thread application running on core 0. However, the IPC (average instructions per cycle) of core 0 depends on what the other cores are doing. If cores 1,2 an

clock freq.	fixed
branch pred.	31KB TAGE & 6KB ITTAGE [30]
I-fetch	1 cache line, 1 taken branch per cycle
branch misp. penalty	12 cycles (minimum), redirect I-fetch at branch execution
decode	8 instructions / cycle
rename	12 micro-ops / cycle
execution ports	4 INT, 2 FP, 3 addr, 2 loads, 1 INT store data, 1 FP store data
retire	12 micro-ops / cycle
reorder buffer	256 micro-ops
issue buffers	INT: 60 micro-ops, FP: 60 micro-ops
phys. registers	128 INT (12 read, 6 write), 128 FP (5 read, 4 write)
ld/st queues	72 loads, 42 stores
MSHR	32 DL1 block requests
store sets [3]	SSIT 2k, LFST 42 stores
cache line	64 bytes
TLB entries	ITLB1: 64, DTLB1: 64, TLB2: 512
IL1	32KB, 8-way assoc. LRU
DL1	32KB, 8-way assoc. LRU, write back, 3-cycle lat., 8 banks, 8 bytes/bank, 2 read + 1 write
L2 (private)	512KB, 8-way assoc. LRU, write back, 11-cycle lat., 16-entry fill queue
L3 (shared)	8MB, 16-way assoc. **5P**, write back, 21-cycle lat., 32-entry fill queue
memory	2 channels, 1 mem. controller / channel
each channel	64-bit bus, bus cycle = 4 core cycles, 1 rank, 8 chips/rank, 8 banks/chip, row buffer: 1KB/chip (8KB/rank)
DDR3 param. (in bus cycles)	tCL=11, tRCD=11, tRP=11, tRAS=33, tCWL=8, tRTP=6, tWR=12, tWTR=6, tBURST=4 (8 beats)
memory controller	32-entry read queue and 32-entry write queue per core
DL1 prefetch	**stride prefetcher**, 64 entries
L2 prefetch	**next line prefetcher**
page size	**4KB / 4MB**
active cores	**1 / 2 / 4**

Table 1: Baseline microarchitecture

3 are idle, core 0 has exclusive access to the whole L3 capacity and memory bandwidth. However, when the other cores are running some application, the IPC of core 0 generally decreases. One reason for this IPC loss is that core 0 experiences more L3 misses because of the competition between cores for the L3 capacity. Another reason is the competition between cores for the memory bus and for DRAM banks, which increases queuing delays (particularly in the memory controller) and degrades DRAM row buffer locality.

To study how these effects impact prefetching effectiveness, we use an artificial micro-benchmark that thrashes the L3 cache by writing a huge array, going through the array quickly and sequentially. We consider 3 configurations in this study:

- **1 active core:** only core 0 is active, cores 1, 2, and 3 are idle.
- **2 active cores:** core 1 runs an instance of the cache-thrashing micro-benchmark, cores 2 and 3 are idle.
- **4 active cores:** cores 1, 2 and 3 each run an instance of the micro-benchmark.

We simulate virtual-to-physical address translation by applying a randomizing hash function on the virtual page number. Hence physical addresses generated by core 0 do not depend upon whether other cores are idle or not. This is an approximation of what happens in real systems, but it makes performance analysis easier.

Figure 2 shows the IPCs of the 29 SPEC CPU2006 benchmarks for the 6 baseline configurations, corresponding to 1, 2, and 4 active cores, 4KB pages and 4MB pages[6].

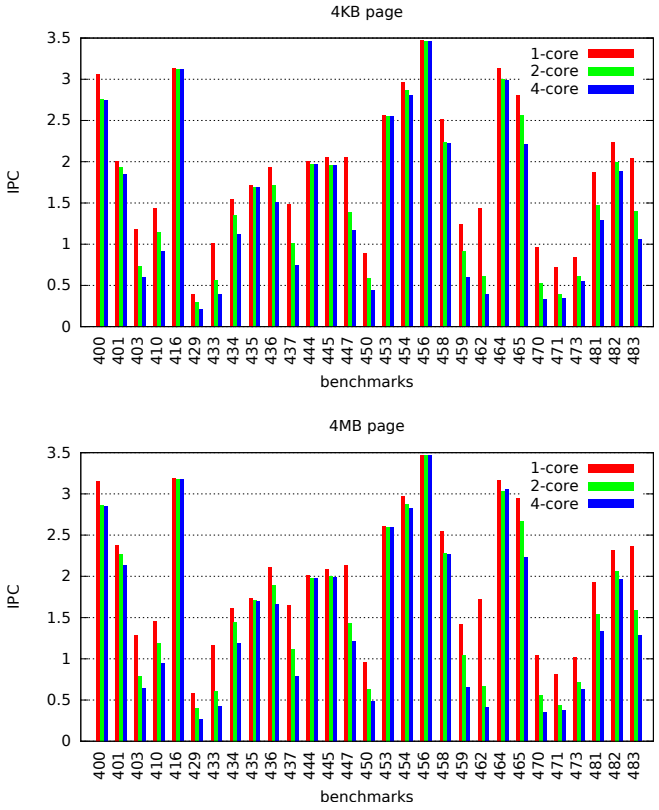

Figure 2: IPC on core 0 for the 6 baseline configurations: 1,2,4 active cores, 4KB page (upper graph), 4MB page (lower graph)

The IPC is generally higher with 4MB pages than with 4KB pages, owing to fewer TLB1 and TLB2 misses.

The IPC of core 0 generally decreases when other cores execute the cache-thrashing micro-benchmark. Nevertheless, some benchmarks are more sensitive than others. Even though we tried to provide fairness (see sections 5.2 and 5.3), activity on other cores reduces the IPC of core 0 quite substantially on certain benchmarks (e.g., 462.libquantum). This allows us to evaluate prefetching when the effective L3 capacity and memory bandwidth available to core 0 is smaller.

5.2 L3 cache replacement policy

Replacement policies such as DIP [27] or DRRIP [13] have been proposed for L2 and L3 caches. We experimented with these policies at the L2 cache but did not observe any significant performance gain over LRU. Still, we found that important performance gains can be obtained by making the L3 replacement policy prefetch-aware (confirming previous studies [19, 38, 37, 39, 29]) and core-aware.

Our L3 baseline replacement policy, called *5P*, uses set sampling and different insertion policies, like the DIP policy [27]. We use five different insertion policies:

- IP1: MRU insertion (i.e., classical LRU replacement).

[6]This is the only graph showing IPCs in this paper, and the only graph where the 6 baselines can be compared with each other.

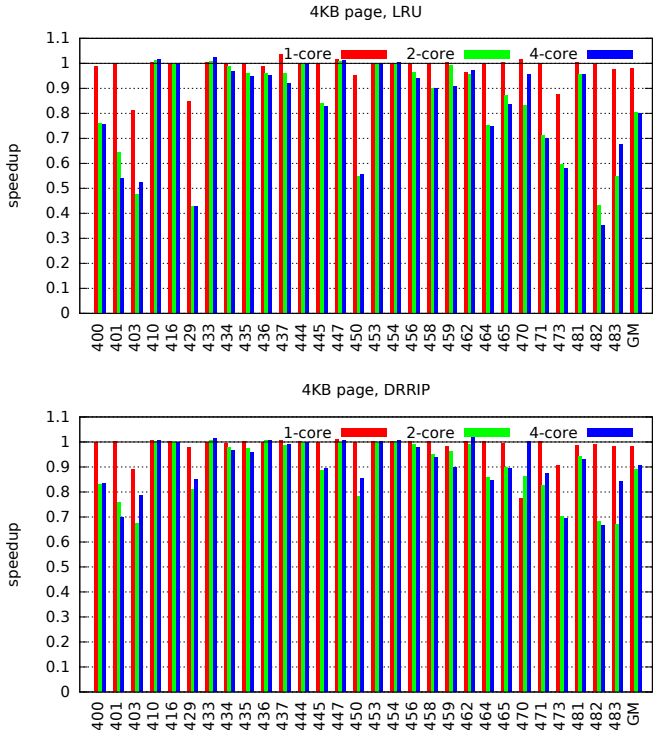

Figure 3: Impact of replacing the baseline L3 replacement policy with LRU (upper graph) and with DRRIP (lower graph). Performance is relative to baselines for 4KB pages. The rightmost cluster of each graph is the geometric mean over the 29 benchmarks.

- IP2: probabilistic LRU/MRU insertion, aka BIP [27].
- IP3: MRU insertion if demand miss, otherwise LRU insertion [32, 34, 38]).
- IP4: MRU insertion if fetched from core with low miss rate, otherwise LRU insertion.
- IP5: MRU insertion if demand miss from core with low miss rate, otherwise LRU insertion.

Upon a cache hit, the hitting block is always moved to the MRU position.

The original DIP policy has only two insertion policies to choose from, and a single counter is sufficient for choosing. With more than two insertion policies, a different mechanism is required. Instead we use a new method. We have one counter C_i per insertion policy IP_i, i.e., 5 counters here. When a demand-miss block is inserted into a set dedicated to policy IP_i, we increment counter C_i. The insertion policy selected in the follower sets is the one with the lowest counter value. However, if the counter value could increase without limitation, this mechanism would be unable to adapt to application behavior changes. Hence we limit the counter value, which cannot exceed CMAX. When *any* counter reaches CMAX, *all* counter values are halved at the same time. This mechanism, which we call *proportional counters*, gives more weight to recent events. For our L3 replacement policy, we use five 12-bit counters and a constituency size of 128 sets.

The IP4 and IP5 insertion policies try to protect from eviction the blocks fetched from cores with a low miss rate [21]. To evaluate the miss rate, we use four 12-bit proportional counters, one counter per core. When a block is inserted into the L3, the counter associated with the corresponding core is incremented. The four counters are halved whenever one of them reaches CMAX. The miss rate is considered low if the counter value is less than 1/4 the maximum of the 4 counter values. It is considered high otherwise.

Figure 3 shows the impact of replacing the 5P policy with LRU and with DRRIP (4KB pages). With a single active core, P5 slightly outperforms both LRU and DRRIP thanks to the prefetch-aware insertion policy IP3, which is particularly effective on 403.gcc and 473.astar.

When several cores are active, the L3 replacement policy has a more dramatic impact. The core-aware insertion policies IP4 and IP5 are effective at providing fairness in the L3 cache utilization.

5.3 Main memory

Our baseline features two memory channels, each with a 64-bit bus clocked at 1/4 the CPU clock frequency. The main memory is loosely modeled after a DDR3 SDRAM (see Table 1). We model neither refresh nor power related parameters (e.g., tFAW). Each DRAM chip has a 1KB row buffer, i.e., a rank of 8 chips has a total 8KB row buffer.

Physical addresses are mapped onto memory as follows. Let $a_{32}\cdots a_6$ be the line address bits ($a_5\cdots a_0$ is the line offset). The mapping for a line is:

Channel (1 bit)	$a_{11} \oplus a_{10} \oplus a_9 \oplus a_8$
Bank (3 bits)	$(a_{16} \oplus a_{13},\ a_{15} \oplus a_{12},\ a_{14} \oplus a_{11})$
Row offset (7 bits)	$(a_{13}, a_{12}, a_{11}, a_{10}, a_9, a_7, a_6)$
Row	$(a_{32}, \cdots, a_{17})$

where $\oplus$ represents the XOR operation.

Each channel has its own memory controller. The two controllers work independently from each other. For fairness, each core has its own read queue and write queue in each controller. The controller selects requests from the queues and schedule them for issue. Once a request is selected, it is issued at the earliest time possible depending on previously selected requests.

For fairness, the scheduler uses a set of four 7-bit *proportional counters* (see Section 5.2). There is one counter per core. A counter is incremented when a read request from the corresponding core is selected for issue.

The scheduler has 2 modes: *steady* and *urgent*. In *steady* mode, a core is first chosen, the *served* core, then read or write requests from that core are selected. The proportional counters are used to determine the served core. However, for a good utilization of row buffer locality, the served core is not changed on every scheduling step but only when one of the write queues is full or when there is no pending read request from the served core hitting in an open row buffer. Writes are performed in batches of 16 writes. Write requests are selected out-of-order from the write queue, trying as much as possible to exploit row buffer locality and bank parallelism. For read requests, an FR-FCFS policy is used [28]. A row is left open after it has been accessed until a subsequent access requires

to close it.

The *urgent* mode is for providing fairness when cores run applications with different miss rates. The urgent mode preempts the steady mode. We define the *lagging* core as the core with the smallest counter value among those with a non-empty read queue. If the L3 fill queue is not full and if the difference between the served core and lagging core counters exceeds 31, a read request from the lagging core is selected for issue.

The scheduler does not distinguish between demand and prefetch read requests. They are treated equally.

5.4 Fill queues and late prefetches

Some microarchitecture simulators implement L2 and L3 MSHRs. MSHRs hold information about pending misses [18]. An MSHR entry is deallocated only after the corresponding miss request has been fulfilled. MSHRs are necessary at the DL1 cache for keeping track of loads/stores that depend on a missing block (and that will have to be rescheduled) and for preventing redundant miss requests. However, MSHRs are not necessary at the L2/L3 caches.

Our baseline microarchitecture does not have L2/L3 MSHRs. Instead, we add associative search capability to the fill queues.

A fill queue is a FIFO holding the blocks that are to be inserted in the cache. An entry is allocated in the fill queue when a miss request is issued to the next cache level (or to memory). For instance, when an L1/L2 miss request is issued to the L3, an entry is reserved at the tail of the L2 fill queue (a request is not issued until there is a free entry).

If the miss request hits in the L3 cache, the fetched block is written into the fill queue. Otherwise, if the block misses in the L3, the fill queue entry is released, and the L1/L2 miss request becomes an L1/L2/L3 miss request. Some metadata (a few bits) is associated with each request as it travels through the memory hierarchy, indicating its type (prefetch or demand miss, instruction or data) and in which cache levels the block will have to be inserted.

Before a block in the fill queue can be inserted into the cache, the replacement policy is invoked to determine the cache way. When the block is inserted into the cache, and if the block was requested at smaller cache levels, it is forwarded to them. For instance, upon an L1/L2/L3 data miss, the block is forwarded to the L2 fill queue while being written into the L3 cache. Later, when the block is written into the L2, it is simultaneously forwarded to the DL1 fill queue[7].

Without L2/L3 MSHRs, we must modify the fill queues for benefiting from late prefetches. When a demand miss hits in a fill queue and the block in the fill queue was prefetched, the miss request is dropped and the block in the fill queue is promoted from prefetch to demand miss. The tag and request type of each fill queue entry are stored in a separate CAM. When a demand miss address is presented to the fill queue, the CAM is searched, and any entry with a matching tag gets promoted.

Before writing a prefetched block in the cache, if the prefetch was generated by the prefetcher at that cache level (i.e., it has not been promoted in the meantime), we must check the cache tags to make sure that the block is not already in the cache. This check is mandatory for correctness[8]. It should be noted that MSHRs would not obviate the need to do such tag check.

[7]Without fill queues, it would be necessary to coordinate all the cache levels to complete an L1/L2/L3 miss, which would be difficult to implement.

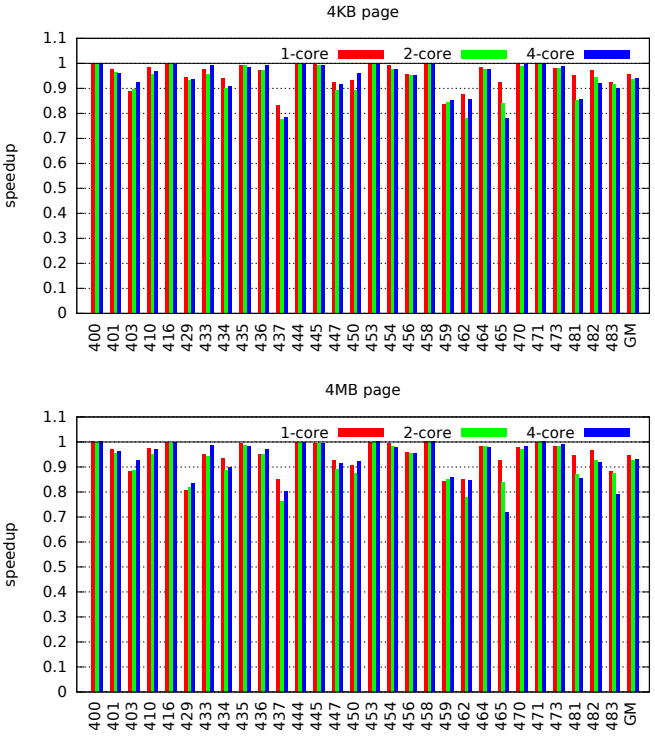

Figure 4: Impact of disabling the DL1 stride prefetcher (performance relative to baselines).

Prefetch requests can be cancelled at any time without taking any specific action [9]. In particular, L2 prefetch requests have the lowest priority for accessing the L3 cache. Prefetch requests wait in an 8-entry prefetch queue until they can access the L3 cache. When a prefetch request is inserted into the queue, and if the queue is full, the oldest request is cancelled.

5.5 DL1 prefetcher

This study focuses on L2 prefetching. The DL1 prefetcher, though not the focus of this study, impacts the L2 prefetcher:

- The L2 prefetcher input stream includes L1 prefetches. The L2 prefetcher does not see addresses in the same order with and without the L1 prefetcher.
- L1 prefetch requests have by definition a better latency tolerance than L1 demand miss requests. When the L1 prefetcher does a perfect job, the L2 prefetcher is superfluous.

Our DL1 prefetcher is a stride prefetcher [7, 31, 1]. It features a 64-entry prefetch table accessed with the PC (instruc-

[8]Blocks must not be duplicated in the cache.

[9]With MSHRs, cancelling a prefetch request would require to update the MSHR state.

tion address) of load/store micro-ops. Each entry contains a tag (the PC), a last address *lastaddr*, a stride, a 4-bit confidence counter and some bits for LRU management. The prefetch table is updated at retirement, not at execution, to guarantee that memory accesses are seen in program order. However, prefetch requests are issued when a load/store accesses the DL1 cache.

When a load/store micro-op retires, it updates its prefetch table entry as follows. If the load/store address *currentaddr* equals $lastaddr + stride$, the confidence counter is incremented, otherwise it is reset to zero. Then, the stride is set equal to $currentaddr - lastaddr$, and *lastaddr* is set to *currentaddr*. Note that *currentaddr* and *lastaddr* are both virtual addresses.

When a load/store accesses the DL1, if this is a miss or a prefetched hit, the prefetch table is looked up. If the load/store has an entry in the table, if the stride is non null and if the confidence counter value equals 15, a prefetch address is computed from the address *currentaddr* of the loads/store: $prefetchaddr = currentaddr + 16 \times stride$ (we determined the fixed factor 16 empirically). Before issuing a prefetch request, we check in a 16-entry filter that a prefetch request to the same line has not been recently issued. If the prefetch address passes the filter, it accesses the second-level TLB (TLB2) and a physical line address is generated. If there is a TLB2 miss, the prefetch request is dropped[10]. Otherwise, it is issued to the uncore (L2 first, then L3 if L2 miss, then memory if L3 miss).

Figure 4 shows the impact of disabling the L1 prefetcher. The L1 prefetcher brings significant speedups on several benchmarks. The maximum speedup is +39% on 465.tonto with 4MB pages and 4 active cores.

5.6 L2 prefetcher

All the different L2 prefetchers considered in this study ignore load/store PCs and work on physical line addresses. They do not have access to TLB or page table information. Prefetch addresses are generated from core request addresses, by modifying the page-offset bits, keeping physical page numbers unchanged.

Our default L2 prefetcher is a simple next-line prefetcher with prefetch bits [33]. Each L2 cache entry contains a prefetch bit, which is set when a prefetched line is inserted into the L2 and which is reset whenever the line is requested by the L1 (L1 miss or L1 prefetch). When a core request for the line of physical address X accesses the L2, if there is a L2 miss or if there is a L2 hit and the prefetch bit is set, a prefetch request for line $X + 1$ is issued to the L3 (or to memory if L3 miss). The L2 prefetchers considered in this study all use prefetch bits. Prefetch bits prevent many useless prefetch requests.

Figure 5 shows the performance impact of disabling the next-line prefetcher. Next-line prefetching is quite effective. Disabling it hurts performance substantially on several benchmarks. All subsequent simulations have the L2 prefetcher enabled. Unless specified otherwise, the speedup numbers provided are relative to the baselines with L2 next-line prefetching.

[10]We did not simulate TLB prefetching.

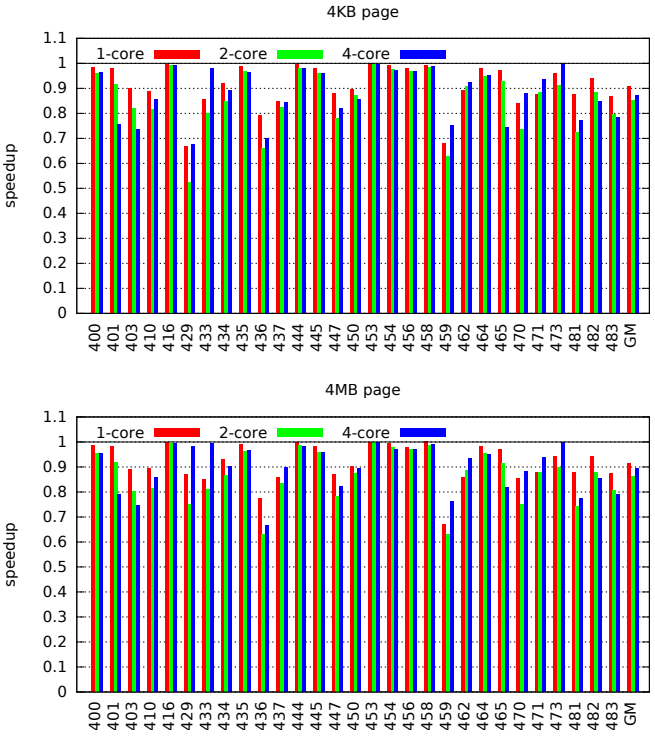

Figure 5: Impact of disabling the L2 next-line prefetcher (performance relative to baselines).

RR table entries	256
RR tag bits	12
SCOREMAX	31
ROUNDMAX	100
BADSCORE	1
scores	52
offset list	cf. Section 4.2

Table 2: BO prefetcher default parameters

6. BEST-OFFSET PREFETCHING

Table 2 provides the default parameters of the BO prefetcher evaluated in this study.

Figure 6 shows the performance of BO prefetching relative to next-line prefetching. BO prefetching brings significant speedup on more than one third of the CPU2006 benchmarks. The speedup peaks at 2.2 for benchmark 470.lbm with 4MB pages and 2 active cores. The speedup over next-line prefetching is on average more important with 4MB pages than with 4KB pages, because with large pages a few benchmarks benefit greatly from large prefetch offsets.

On average, the speedup of BO prefetching over next-line prefetching is more pronounced when 2 cores are active than when only core 0 is active. Our understanding is that, when core 0 competes with another core for L3 space and memory bandwidth, the L2 miss latency increases because of more L3 misses, longer queuing delays in the memory controller and degraded DRAM row buffer locality. The best offset is generally larger with longer L2 miss latencies, and next-line

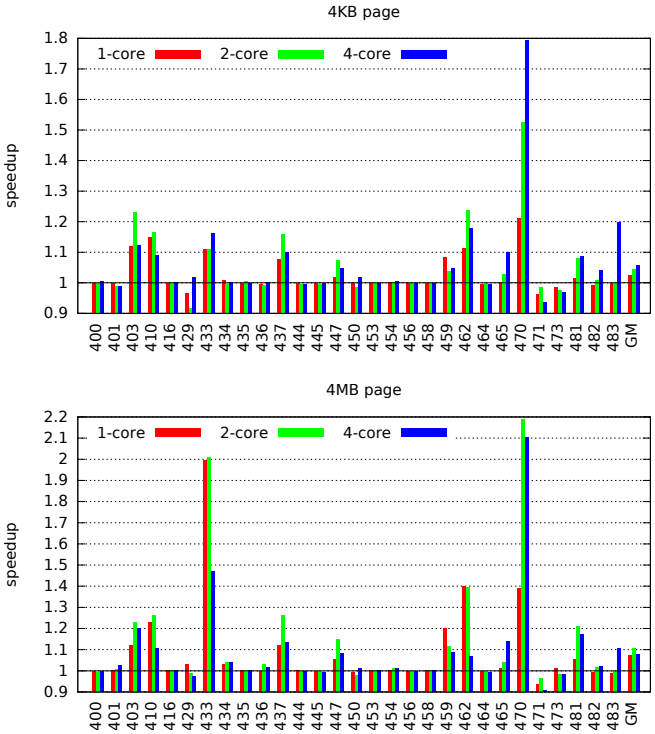

Figure 6: BO prefetcher speedup (relative to next-line prefetching baselines).

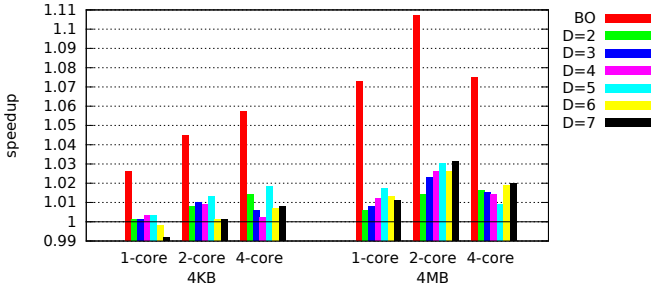

Figure 7: BO prefetcher compared with fixed-offset prefetchers (geometric mean speedup on all benchmarks).

prefetching is relatively less effective. With 4 active cores, the L2 miss latency is even longer, but the reduced memory bandwidth becomes a performance bottleneck for some benchmarks, a problem that prefetching cannot solve.

Figure 7 compares Best-Offset prefetching with fixed-offset prefetching where the fixed offset ranges from 2 to 7. An offset of 1 is clearly not the best fixed offset on the CPU2006 benchmarks. The best fixed offset is 5.

Figure 8 shows, for four chosen benchmarks and a 4MB page size, the speedup of fixed-offset prefetching, where the fixed offset ranges from 2 to 256 (the speedup of BO prefetching is indicated with a horizontal line).

BO prefetching substantially outperform next-line prefetching on these benchmarks. Benchmarks 433.milc, 459.GemsFDTD and 470.lbm exhibit what resembles

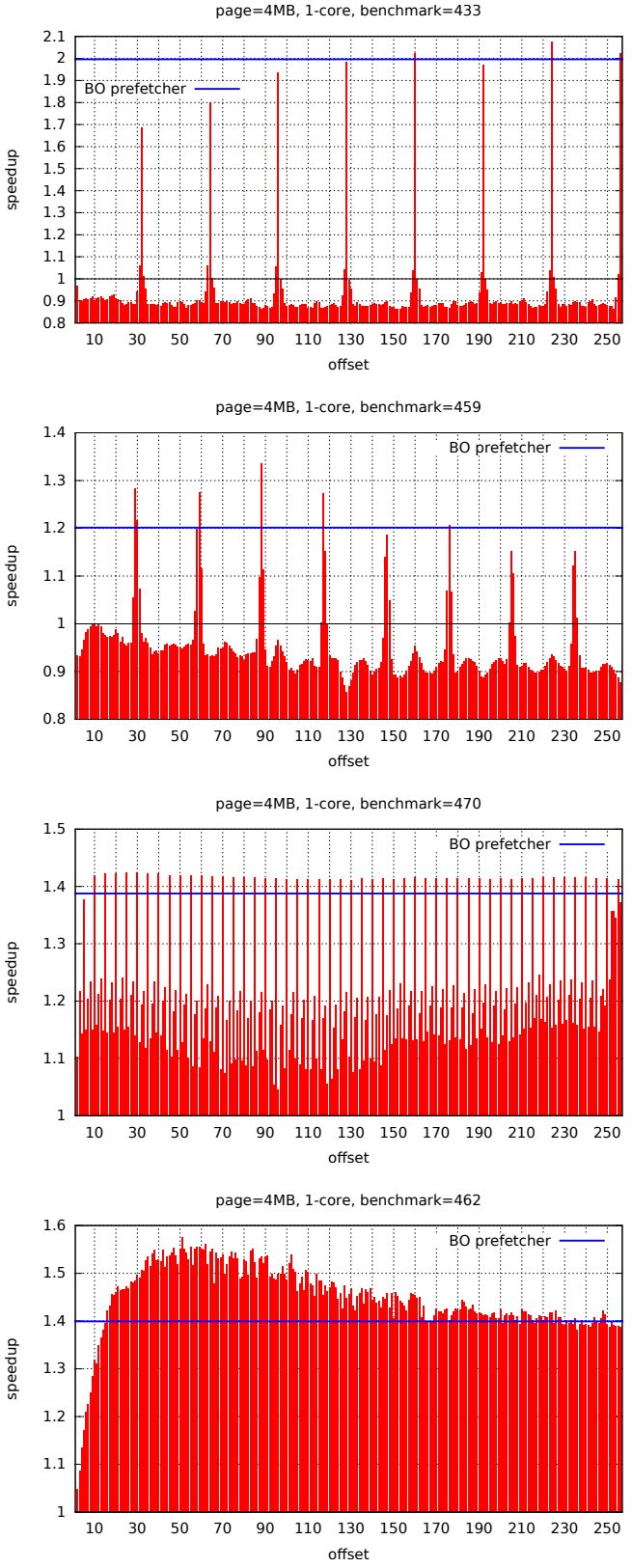

Figure 8: Fixed-offset prefetching with offset ranging from 2 to 256 on benchmarks 433, 459, 470, and 462 (4MB page, 1 active core).

constant-stride accesses[11].

Benchmark 433.milc has its performance peaks at offset values multiple of 32. Note that 433.milc benefits from very large offsets (provided superpages are used).

Benchmark 459.GemsFDTD has its performance peaks at offset values 29, 59, 88, 117, 147, 176, etc., i.e., not exactly multiples of 29, but close. These offset values are not in our fixed list of 52 offsets, and the BO prefetcher is not able to reach the maximum performance. However, the peaks are not isolated. Some offset values close to the peaks, though not optimal, achieve good performance. Luckily, one of these offsets, 30, happens to be in our offset list.

The behavior of 470.lbm is even more unintuitive. Its performance peaks are at offset values multiple of 5. Yet there are secondary (and much smaller) peaks at multiples of 5 plus 3, and there is a cluster of high-performing offsets between 252 and 254.

Benchmark 462.libquantum has a more regular behavior. It generates long sequential streams and can sustain a relatively high IPC if given enough memory bandwidth. However, prefetch timeliness is crucial here, and large offsets are necessary to hide the memory latency. On this benchmark, the BO prefetcher works imperfectly and does not find the best offset. The reason is that the BO prefetcher strives for prefetch timeliness. The offsets yielding the greatest number of timely prefetches have very large values (above 200). Offset values between 30 and 100 yield fewer timely prefetches, but more late prefetches and greater coverage. This example shows that striving for prefetch timeliness is not always optimal. This is one of the main weaknesses of the BO prefetcher, a problem that future work should try to solve[12].

Nevertheless, an important conclusion from Figure 8 is that prefetch timeliness is essential for performance. Recall that speedups are relative to the baseline with L2 next-line prefetching, which is already very effective (Figure 5). Although not apparent in Figure 8, the baseline next-line prefetcher yields a high prefetch coverage on these 4 benchmarks (about 75% coverage for 433.milc and 470.lbm, above 90% for 459.GemsFDTD and 462.libquantum). Yet, the performance of next-line prefetching is quite suboptimal because most prefetches are late.

6.1 Impact of prefetch throttling

Figure 9 shows the impact on mean speedup of varying parameter BADSCORE. A majority of the CPU2006 benchmarks are not impacted by parameter BADSCORE. From this experiment, we believe that prefetch throttling makes a difference only for a minority of applications. On the CPU2006 benchmarks, it occurs that in the few cases where prefetch throttling makes a difference (mostly 429.mcf), it hurts performance. On the SPEC CPU2006, the optimal value of BADSCORE is clearly less than 5% of ROUNDMAX. A much larger set of benchmarks would be needed to draw definitive conclusions. For subsequent simulations, we use BADSCORE=1.

[11]One might expect the L1 stride prefetcher to work well on these benchmarks. This is not the case, for various reasons. See for instance the analysis of 433.milc in [11, 17].

[12]Changing the offset list by removing offsets above 100 would be an ad hoc solution, not a real solution.

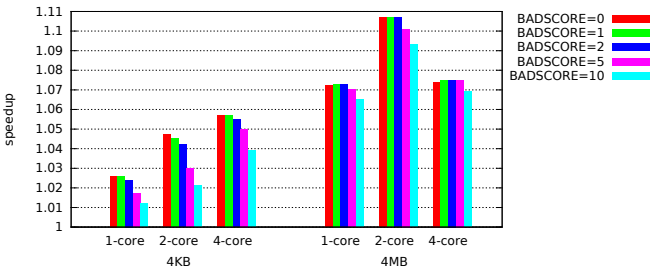

Figure 9: BO prefetcher geometric mean speedup, impact of varying BADSCORE.

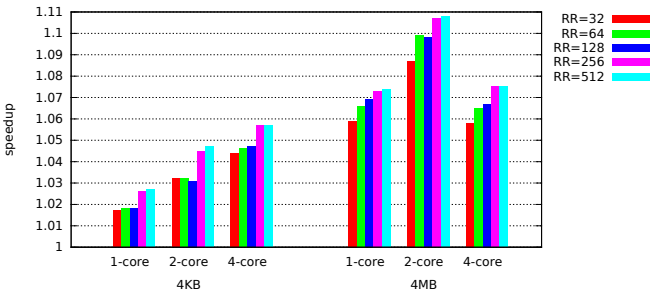

Figure 10: BO prefetcher geometric mean speedup, impact of varying the number of entries of the RR table.

6.2 Impact of the RR table

Figure 10 shows the impact on mean speedup of varying the number of entries of the RR table, from 32 to 512 entries. Prefetching effectiveness increases with the RR table size, up to a certain point. A performance gap can be observed with 4KB pages when going from 128 entries to 256 entries. This gap comes from one benchmark, 429.mcf.

6.3 Comparison with Sandbox prefetching

To the best of our knowledge, the SBP prefetcher of Pugsley et al. is the first published full-fledged offset prefetcher[26]. The SBP prefetcher is cost-effective and was shown to outperform significantly the FDP prefetcher [37] on the SPEC CPU2006 benchmarks, and to match or even slightly outperform the more complex AMPM prefetcher [11].

The SBP prefetcher contains two independent ideas: offset prefetching, and *sandbox* prefetching. Sandbox prefetching is the idea that a prefetcher can be evaluated without it issuing actual prefetch requests. Instead, "fake" prefetch requests are performed by recording them in a structure, which Pugsley et al. proposed to be a Bloom filter. The accuracy of a fake prefetch is evaluated by checking if a subsequent access hits in the Bloom filter. The SBP prefetcher does not take into account prefetch timeliness.

The BO and SBP prefetcher are both offset prefetchers, but they use different offset selection mechanisms. The sandbox is a stand-alone idea that could be used in hybrid prefetching schemes, for choosing dynamically between different sorts of prefetchers, not necessarily offset prefetchers. By contrast, the selection mechanism used in the BO prefetcher is specific

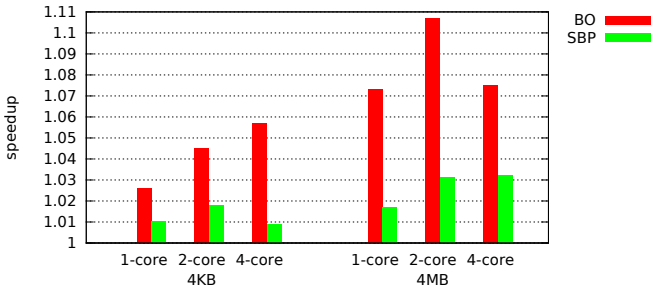

Figure 11: Comparison between the BO and SBP prefetchers (geometric mean speedups relative to baselines).

to offset prefetching. While our RR table may look superficially like a sort of sandbox, it is not: the sandbox is updated with fake prefetches, while the RR table is updated with actual prefetches.

We implemented the SBP prefetcher as described in the original paper [26], but with a few modifications to make the comparison with BO prefetching meaningful. Our SBP uses the same list of offsets as the BO prefetcher (52 positive offsets) and the same number of scores (52). Our SBP uses a 2048-bit Bloom filter indexed with 3 hashing functions. The evaluation period is 256 L2 accesses (miss or prefetched hit). When line X is accessed, we check in the Bloom filter for X, $X-D$, $X-2D$ and $X-3D$, incrementing the score on every hit. The original SBP is a degree-N prefetcher. It can issue prefetch requests with several different offsets, provided their scores are above the accuracy cutoffs. It can also issue 1, 2 or 3 prefetch requests for the same offset depending on the score for that offset. This is how SBP tries to compensate for not being able to evaluate prefetch timeliness. However, degree-N prefetching may generate a lot of redundant prefetches. Some redundant prefetches are removed automatically by the mechanisms already in place in our baseline for dealing with late prefetches [13]. Nevertheless, for SBP, we also look up the L2 tags before issuing a prefetch[14]. Apart from this extra L2 tag check and replacing the BO prefetcher with SBP, everything else is unchanged, and the two prefetchers operate in the same conditions.

Figure 11 compares the BO and SBP prefetchers, giving the geometric mean speedups relative to baselines. SBP does outperform the default next-line prefetcher on average, but not as much as the BO prefetcher. Figure 12 gives for each benchmark the speedup of the BO prefetcher relative to the SBP prefetcher. SBP outperforms the BO prefetcher on some benchmarks, but never by a big margin (always within 10%). However, on some benchmarks (429.mcf, 433.milc,...), BO prefetching brings substantial speedups over SBP.

Figure 13 gives the number of DRAM accesses (read or write) per 1000 instructions, comparing 4 configurations: no L2 prefetch, next-line prefetch, BO prefetch, and SBP. The next-line and BO prefetchers generate approximately the same memory traffic. The BO prefetcher generates significantly more memory traffic than SBP on benchmarks

[13]Fill queues are associatively searched. So are the prefetch queue and the memory controller read queue before insertion.

[14]We assume that tag lookups do not impact performance.

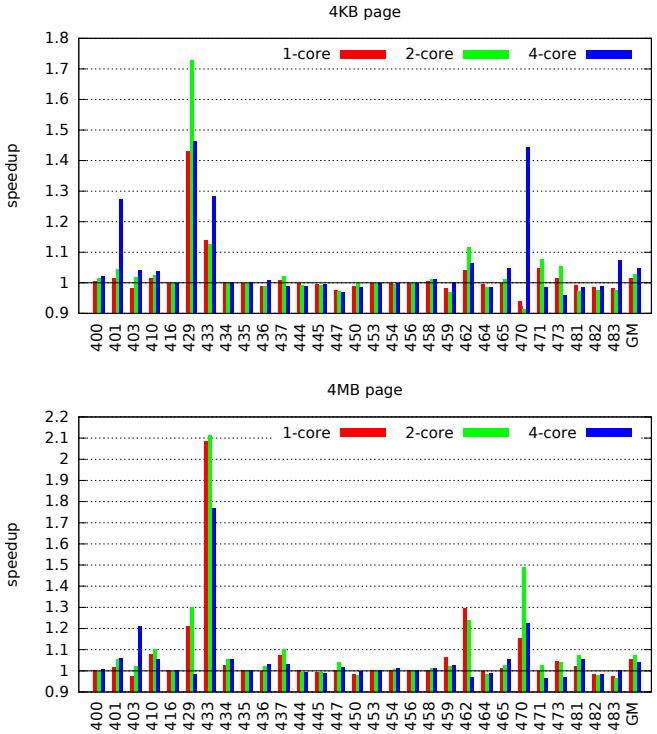

Figure 12: BO prefetcher speedup relative to SBP.

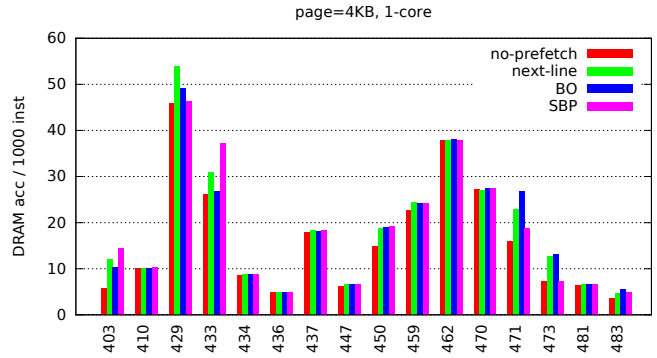

Figure 13: Number of DRAM accesses per 1000 instructions (4KB page, 1 active core). Omitted benchmarks access the DRAM infrequently.

471.omnetpp and 473.astar. On benchmarks 403.gcc and 433.milc, it is the other way around. On other benchmarks, the BO and SBP prefetchers have similar prefetch aggressiveness.

The performance differences observed in Figure 12 mostly come from the ability of BO prefetching to take into account prefetch timeliness. SBP tends to select small offsets yielding high prefetch coverage, but with late prefetches and suboptimal performance (cf. Figure 8).

7. CONCLUSION

The Sandbox prefetcher of Pugsley et al. defines a new class of prefetchers, *offset* prefetchers, that are very effective despite requiring simple hardware. However, the Sandbox

prefetcher does not take into account prefetch timeliness. We propose an offset prefetcher with a new method for selecting the prefetch offset, different from the sandbox method. We showed that our BO prefetcher outperforms the Sandbox prefetcher on the SPEC CPU2006 by taking into account prefetch timeliness.

Future work on offset prefetching will have to solve two problems that we did not address in this study. One problem is the fact that striving for prefetch timeliness is not always optimal, i.e., the Best-Offset prefetcher we presented is imperfect, it does not always find the best prefetch offset. The second problem is more general and concerns most studies about prefetching. Prefetching effectiveness is strongly dependent on application characteristics. We found empirically that, on the SPEC CPU2006, our prefetch throttling parameter BADSCORE could be set very low. There is no guarantee that this is true on a larger set of applications. Future work may try to adjust dynamically the throttling parameter.

Acknowledgment

This work is partially supported by the European Research Council Advanced Grant DAL No 267175.

8. REFERENCES

[1] T.-F. Chen and J.-L. Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers*, 44(5), May 1995.

[2] Y. Chou. Low-cost epoch-based correlation prefetching for commercial applications. In *MICRO*, 2007.

[3] G. Z. Chrysos and J. S. Emer. Memory dependence prediction using store sets. In *ISCA*, 1998.

[4] F. Dahlgren, M. Dubois, and P. Stenström. Fixed and adaptive sequential prefetching in shared memory multiprocessors. In *ICPP*, 1993.

[5] DPC-1. 1st JILP Data Prefetching Championship, 2009. http://www.jilp.org/dpc/.

[6] DPC-2. 2nd Data Prefetching Championship, 2015. http://comparch-conf.gatech.edu/dpc2/.

[7] J. W. C. Fu, J. H. Patel, and B. L. Janssens. Stride directed prefetching in scalar processors. In *MICRO*, 1992.

[8] E. Hagersten. *Toward scalable cache only memory architectures*. PhD thesis, Royal Institute of Technology (KTH), Stockholm, Sweden, 1992.

[9] I. Hur and C. Lin. Memory prefetching using adaptive stream detection. In *MICRO*, 2006.

[10] Intel. *Intel 64 and IA-32 architectures optimization reference manual*, September 2014.

[11] Y. Ishii, M. Inaba, and K. Hiraki. Access map pattern matching for high performance data cache prefetch. *Journal of Instruction-Level Parallelism*, 13, January 2011.

[12] A. Jain and C. Lin. Linearizing irregular memory accesses for improved correlated prefetching. In *MICRO*, 2013.

[13] A. Jaleel, K. B. Theobald, S. C. Steely Jr., and J. Emer. High performance cache replacement using re-reference interval prediction (RRIP). In *ISCA*, 2010.

[14] D. Joseph and D. Grunwald. Prefetching using Markov predictors. In *ISCA*, 1997.

[15] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully associative cache and prefetch buffers. In *ISCA*, 1990.

[16] G. B. Kandiraju and A. Sivasubramaniam. Going the distance for TLB prefetching: an application-driven study. In *ISCA*, 2002.

[17] T. Kim, D. Zhao, and A. V. Veidenbaum. Multiple stream tracker: a new hardware stride prefetcher. In *Computing Frontiers*, 2014.

[18] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *ISCA*, 1981.

[19] W.-F. Lin, S. K. Reinhardt, and D. Burger. Reducing DRAM latencies with an integrated memory hierarchy design. In *HPCA*, 2001.

[20] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa Reddi, and K. Hazelwood. Pin : building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[21] P. Michaud. The 3P and 4P cache replacement policies. In *1st JILP Workshop on Computer Architecture Competitions (JWAC-1): Cache Replacement Championship*, 2010. http://www.jilp.org/jwac-1/.

[22] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith. AC/DC: an adaptive data cache prefetcher. In *PACT*, 2004.

[23] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. In *HPCA*, 2004.

[24] S. Palacharla and R. Kessler. Evaluating stream buffers as a secondary cache replacement. In *ISCA*, 1994.

[25] D. Gracia Pérez, G. Mouchard, and O. Temam. MicroLib: a case for quantitative comparison of micro-architecture mechanisms. In *MICRO*, 2004.

[26] S. H. Pugsley, Z. Chishti, C. Wilkerson, P. f. Chuang, R. L. Scott, A. Jaleel, S.-L. Lu, K. Chow, and R. Balasubramonian. Sandbox prefetching: safe run-time evaluation of aggressive prefetchers. In *HPCA*, 2014.

[27] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely Jr., and J. Emer. Adaptive insertion policies for high-performance caching. In *ISCA*, 2007.

[28] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *ISCA*, 2000.

[29] V. Seshadri, S. Yekdar, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. Mitigating prefetcher-caused pollution using informed caching policies for prefetched blocks. *ACM Transactions on Architecture and Code Optimization*, 11(4), January 2015.

[30] A. Seznec and P. Michaud. A case for (partially) tagged geometric history length branch prediction. *Journal of Instruction Level Parallelism*, February 2006.

[31] I. Sklenar. Prefetch unit for vector operations on scalar computers. *Computer Architecture News*, 20(4), September 1992.

[32] A. J. Smith. Sequentiality and prefetching in database systems. *ACM Transactions on Database Systems*, 3(3):223–247, September 1978.

[33] A. J. Smith. Cache memories. *ACM Computing Surveys*, 14(3), 1982.

[34] K. So and R. N. Rechtschaffen. Cache operations by MRU change. *IEEE Transactions on Computers*, 37(6):700–709, June 1988.

[35] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi. Spatio-temporal memory streaming. In *ISCA*, 2009.

[36] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Spatial memory streaming. In *ISCA*, 2006.

[37] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback directed prefetching: improving the performance and bandwidth-efficiency of hardware prefetchers. In *HPCA*, 2007.

[38] Z. Wang, D. Burger, K. S. McKinley, S. K. Reinhardt, and C. C. Weems. Guided region prefetching: a cooperative hardware/software approach. In *ISCA*, 2003.

[39] C.-J. Wu, A. Jaleel, M. Martonosi, S. C. Steely Jr, and J. Emer. PACMan: prefetch-aware cache management for high performance caching. In *MICRO*, 2011.