Journal of Instruction-Level Parallelism 13 (2011) 1-17 Submitted 3/10; published 1/11

Data Prefetching by Exploiting Global and Local Access Patterns

Ahmad Sharif AHMAD@GATECH.EDU
Hsien-Hsin S. Lee LEEHS@GATECH.EDU
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332–0250

Abstract

This paper proposes a new hardware prefetcher that extends the idea of the Global History Buffer (GHB) originally proposed in [1]. We augment the GHB with several Local History Buffers (LHBs), which keep the memory access information for selective program counters. These buffers can then be queried on cache accesses to predict future memory accesses and enable data prefetching using novel detection schemes. Our trace-driven simulations show that by using approximately a 4KByte (32 Kbits) storage budget, we can obtain an average performance improvement of 22% for SPEC2006 benchmark suite on an ideal out-of-order processor.

1. Introduction

Single-thread performance can be severely impaired by long-latency cache misses. These cache misses can cause the re-order buffer (ROB) to fill up and allocation to stall for tens or even hundreds of cycles, blocking forward progress. To address this issue, a large number of transistors in recent microprocessors is either dedicated to reducing the number of such misses by enlarging the cache capacity and/or reducing the miss latency via an aggressive prefetcher. In this paper, we investigate the design of a hardware prefetcher under the storage budget of 4KB with allowable logic complexity. Our main contributions are as follows:

1. We identify and classify commonly occuring access patterns of memory instructions.

2. We propose a 4KB hardware prefetcher which demonstrates high effectiveness in data prefetching.

The rest of the paper is organized as follows. Section 2 details the processor model used in this study and the performance implications of the said model. Section 3 describes some of the common access patterns in the SPEC2006 benchmark suite. Section 4 describes the design of our data prefetching scheme. Sections 5 and 6 explain our original DPC submission design point and its storage budget. Section 7 discusses of the simulation infrastructure and our simulation methodology. Section 8 presents other design points of our prefetcher. Section 10 describes future work while Section 11 concludes.

2. Performance with Ideal Front-end & Execution Engine

This study focuses on prefetcher design for a processor with a perfect front-end and execution engine. Such a processor will have an IPC[1] equal to its issue width, N, under ideal conditions. Conditions that cause deviation from ideal IPC include:

1. True dependencies between instructions. For example, if there is a chain of dependencies with the result of the previous instruction providing the source for the next one, the IPC can be at most 1.

2. Resource constraints. In the microarchitecture there are hardware resources that instructions need to hold on to for some number of cycles. An example of such a resource is an entry of the ROB. This entry has to be held on to by an instruction from the time allocation takes place till the in-order retirement of the instruction. Standard queuing theory can be applied to each such resource and Little's Law provides an upper-bound on the maximum IPC. Let the average number of cycles a resource needs to be held on to be L and the capacity of such a resource be C. If that is the case, the IPC is bounded by the following equation:

$$IPC \leq \frac{C}{L} \tag{1}$$

   For example, in the simulated processor used in our study, the ROB is 128 instructions in capacity (see Table 3. To maintain an IPC of 4, the average lifetime of a ROB entry must be less than or equal to 32 cycles. The lifetime of a ROB entry is determined by the instruction's own latency and the latency of its predecessors (the in-order retirement requirement forces this). Long-latency cache misses (especially L2 cache misses) can increase the lifetime of critical microarchitecture structures like ROB entries, etc. to more than their design point. This will cause IPC degradation in accordance with Little's Law.

The condition numbered (1) is a characteristic of the workload and as such cannot be overcome without using aggressive speculative techniques. The average latency of an instruction, however, can be decreased considerably by using an accurate and timely hardware prefetcher.

The goal of a hardware prefetcher is to minimize the average latency of instructions thereby increasing IPC. Several metrics can be used to characterize the performance of prefetchers towards this goal, which are the following:

1. Prefetch coverage: This is the percentage of a program's memory accesses that were initiated by the prefetcher. Percentage reduction in cache misses is directly related to prefetch coverage.

2. Prefetch accuracy: This is the percentage of prefetch requests that were later accessed by the program.

1. By instructions we mean the smallest unit of execution in the microarchitecture, which may be micro-ops. IPC is also used in the same context (instructions or micro-ops per cycle).

3. Prefetch timeliness: The prefetcher should issue prefetch requests at a time such that the cache line arrives just in time for the actual access. Timeliness can be measured in terms of the average prefetch-to-use cycles or the median prefetch-to-use cycles or some other metric.

In this work, we study the design of a global and local history based pattern-detecting hardware prefetcher with 4KB of storage regardless of logic complexity. The goal of this prefetcher is to improve the IPC of the single-threaded workloads running on the processor by reducing the number of long-latency cache misses.

3. Motivation & Observed Access Patterns

To gain an understanding of the cache behavior of modern workloads and glean useful information for effective data prefetching, we used traces generated from the SPEC2006 benchmark suite [2]. For each run, we skipped the first 40 billion instructions and used the next 100 million instructions to analyze the memory access characteristics of the benchmarks[2]. As part of an initial study, we investigated the limit of a prefetching scheme by running simulations with a zero-latency L2 and/or DRAM memory[3]. This gives us an approximate upper-limit on what an ideal prefetcher could achieve. Our simulations were run for three different configurations suggested in the DPC-1 infrastructure and recapped below:

1. Configuration 1 simulated an ideal 4-issue, out-of-order processor with no branch hazards. The L2 cache was 2 MB and bandwidth to the caches and memory was unlimited.

2. Configuration 2 simulated the same processor as Configuration 1, but with limited L2 bandwidth of 1 request per cycle and limited DRAM memory bandwidth of one request every 10 cycles.

3. Configuration 3 simulated the same processor as Configuration 2, but with a 512KB L2 cache.

Figure 1 shows the normalized performance results when parts of the memory-hierarchy are replaced by their zero-latency counterparts (but are still subject to bandwidth constraints). This shows that L2 cache misses that take over 200 cycles (200 cycles DRAM latency + 20 cycles L2 latency + latency because of L2 queue waiting) to get back from memory are a major performance bottleneck (in contrast to L1 cache misses that hit the L2 cache). This makes sense because for our ideal out-of-order processor, the L1 misses that are L2 hits are likely to be tolerated due to the nature of out-of-order execution. Long latency L2 misses, however, can cause the ROB to become full and instruction allocation to stall for a large number of cycles.

2. We did not skip the first 40 billion instructions for 998.specrand, 999.specrand, and 481.wrf because these programs did not have enough instructions.
3. Note that bandwidth limitations on the L2 cache and DRAM memory still applied in configurations 2 and 3.

For a more quantitative analysis, we note the following. For configuration 1 where the L2 cache lookup bandwidth is unlimited, in our simulated processor with perfect front-end, L1 misses that are L2 hits will have a ROB entry lifetime equal to 15 (pipeline depth) + 20 (L2 cache hit latency) = 35 cycles. As shown above, this is within 10% of the lifetime required for maximum throughput (32 as shown in section 2). With this re-order buffer entry lifetime, the IPC limit imposed by memory sub-system (by Little's Law) is:

$$\begin{aligned} IPC_{memory\_sub-system\_limit} &= \frac{ROB_{size}}{ROB_{lifetime}} \\ &= \frac{128}{35} \\ &= 3.65 \end{aligned} \tag{2}$$

This is the IPC limit imposed by the memory sub-system for configuration 1 assuming all accesses are L1 misses and L2 hits. With L2 misses converted to hits, SPEC 2006 benchmarks are often limited by non-memory-subsystem constraints.

However, in configurations 2 and 3, for L1 misses that are L2 hits, the lifetime of a ROB entry will be increased because cycles will have to be spent waiting in the L2 queue before the cache look-up is done[4]. The queuing delay is present because the L2 cache can be looked-up only once every 10 cycles. This increased lifetime will result in degradation of IPC in accordance with Little's Law. However, most of the IPC degradation will still be because of long-latency L2 cache misses. The latency of such misses is usually more than 220 cycles and will increase the lifetime of ROB entries of the instruction that caused the miss and the following instructions.

3.1. Observed Access Patterns

During our initial study, we collected memory access traces from the SPEC2006 benchmarks and made the following prefetch-related observations:

1. Using merely the L2 miss addresses observed from the issue stage of an out-of-order processor might not be the best way to train a prefetcher. These addresses may not appear to contain a regular pattern to the prefetcher in certain cases.

2. It is beneficial to use both cache hits and misses of a program to train the predictor's state. By training a prefetcher using only the observed miss addresses, as some prior work did, the prefetcher may not be able to detect any useful pattern.

3. It is beneficial to train the prefetcher with both the local (per PC) and global access information. Some access patterns are not visible at the global level, while others may only be visible if the prefetcher organizes them by the program counter (PC).

4. It might be helpful to classify accesses into regions (i.e., spatially by memory addresses) and detect patterns within their respective regions. For example, in 470.lbm

4. An interesting study would be on the percentage of time spent in the cache lookup vs. in the L2 queue for L1 cache misses. Unfortunately, because of the closed nature of the simulator, we could not perform this study.

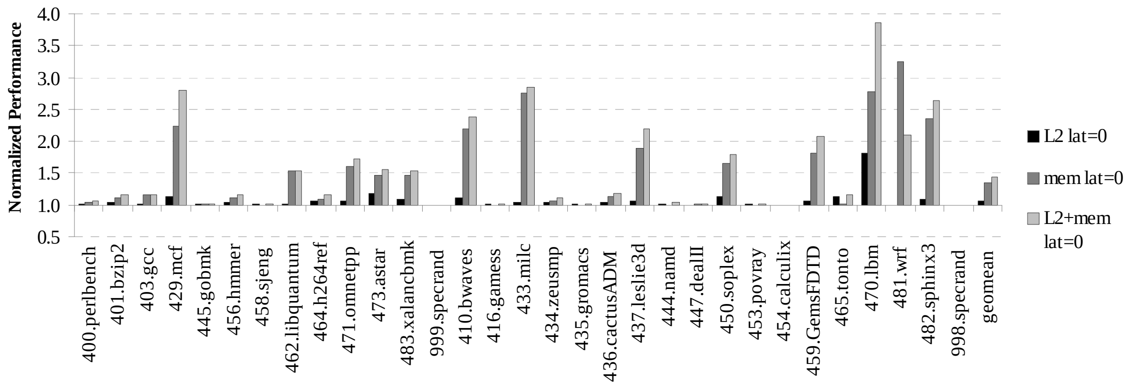

Figure 1: Simulation results showing normalized performance (IPC) when different parts of the memory hierarchy are replaced by their ideal counterparts. The performance here is an average over 3 configurations explained in section 3. An accurate L2 prefetcher can provide a significant portion of the available performance opportunity.

and 462.libquantum, the stride can be easily detected if the accesses are binned according to region. This is because a single instruction can access multiple regions within a period of hundreds of cycles. This might be more visible in a CISC architecture because of complex instructions that generate more than a single memory uop (example: the `movs` family of instructions in x86 decomposes into at least two memory uops that move data from one memory location to another).

5. It might be beneficial to trigger a prefetch request upon each cache *access* (regardless of whether it is a hit or miss) rather than only on a cache *miss*. This is especially useful if the cache access was to a line that the prefetcher brought in.

6. The consecutive memory accesses of some benchmarks follow a exponentially increasing stride. For example,the 429.mcf has accesses with these deltas (in terms of cache lines): 6, 13, 26, 52, etc. This pattern may be observed when the program is chasing pointers that are part of a tree data structure (example: going down a heap data structure laid out as an array).

7. If the logic complexity is not too prohibitive, the prefetcher should detect patterns of memory accesses that might have noise in them. Sometimes programs may not access memory in a completely regular fashion; however, patterns can still be detected if the detection criteria is relaxed.

8. Sometimes because of instruction scheduling optimizations or otherwise, simple patterns may be obscured by "noise". In this case, if the complexity is not too high, a prefetcher may try to do pattern detection using a brute force algorithm. For example, in terms of cache lines a program might access 4, 8, 9, 12, 14, 16, etc. The

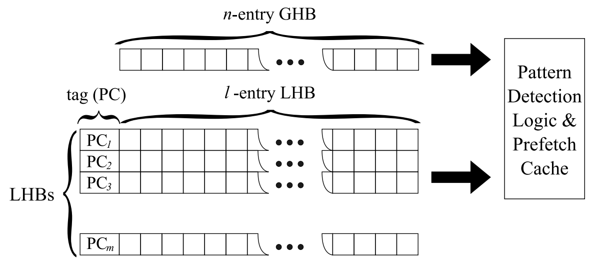

Figure 2: Block diagram of our proposed prefetcher. Each GHB stores n memory access addresses made by the processor. Each LHB stores, along with the PC value, access addresses made by the instruction at that particular PC. The array of LHBs is a fully associative structure, looked up by PC. For our submission, n=128, l=24 and m=32. The prefetch cache is a 32-entry fully associative structure that stores previously issued prefetches.

prefetcher may detect that the program is accessing cache lines with a delta of 4 with some "noise" mixed in and could prefetch lines 20, 24, etc.

4. Prefetcher Design

A hardware prefetcher typically stores some state internally that is derived from past accesses. The current memory accesses are then used along with this state to predict the future access patterns of the program. To train the state of the prefetcher, the simulation framework provided cache line addresses, instruction sequence numbers, PC of the instruction that made the access and whether the access was L1/L2 hit/miss. Typical prefetchers store only a small part of this input as internal state. This is to reduce complexity (which helps lower the prefetcher prediction latency in a real hardware design) and storage requirement. Motivated by the observations from memory-bound benchmark programs in SPEC2006, we decided to store almost all the input provided by the simulation framework in various History Buffers using our 4KB budget. We employ both a single large Global History Buffer (GHB) and multiple smaller Local History Buffers (LHBs) to store the input from the simulator every cycle.

The GHB tracks the most recent n memory accesses (both loads and stores) in a program while each of the LHBs tracks m accesses performed by l a particular PC (each LHB is tagged with a 32-bit PC value). Along with the access address, we also store the hit/miss status of the access in the history buffers. This list of LHBs is maintained as a fully associative cache with LRU replacement. The number of valid entries in the GHB or LHBs is not stored anywhere - instead, an invalid address (CacheAddr_t)(NULL) or (CacheAddr_t)(-1) is inserted after the last valid address. A block diagram of our prefetcher is given in Figure 2.

When an memory access is requested, the processor looks up the GHB and the LHB. An access address is only added if it is not found in these buffers. If the PC is not found in the list of LHBs, the least-recently-used LHB is selected and the access address is stored as the first entry of the LHB (further memory accesses from the same PC will be entered in this

Algorithm 1 The repeating pattern detection algorithm. After a repeated pattern is detected, a score can be assigned based on the length of the repetition (the variable labeled `max` in FindRepeatingPattern).

```
1: procedure FindRepeatingPattern($deltas$, $N$)
2:     $max \leftarrow 0$
3:     for $i \leftarrow N-1, 1$ do
4:         $currentValue \leftarrow CalcRepeatingScore(deltas, i, N)$
5:         if $currentValue > max$ then
6:             $max \leftarrow currentValue$
7:             $maxI \leftarrow i$
8:         end if
9:     end for
10:                ▷ The length of the pattern repeats is now in max. If this greater than a threshold, prefetch is issued.
11: end procedure

12: procedure CalcRepeatingScore($deltas$, $start$, $end$)
13:     $length \leftarrow end - start$
14:     $totalScore \leftarrow 0$
15:     $currStart \leftarrow start$
16:     $currEnd \leftarrow end$
17:     while $currEnd > 0$ do
18:         $currentScore \leftarrow Compare(deltas, start, end, currStart, currEnd)$
19:         $totalScore \leftarrow totalScore + currentScore$
20:         if $curentScore < length$ then
21:             return $totalScore$
22:         end if
23:         $currEnd \leftarrow currEnd - length$
24:         $currStart \leftarrow currStart - length$
25:     end while
26:     return $totalScore$
27: end procedure

28: procedure Compare($deltas, start, end, currStart, currEnd$)
29:     $score \leftarrow 0$
30:     $i \leftarrow 0$
31:     while $deltas[currEnd-i] == deltas[end-(i \% (end-start))]$ && $currEnd-i >= currStart$ do
32:         $score \leftarrow score + 1$
33:         $i++$
34:     end while
35:     return $score$
36: end procedure
```

particular LHB.). The LHB is also tagged with the PC of the memory access instruction. If, however, there is a tag match of the PC in the list of LHBs, the memory access address is simply added to the circular buffer associated with the PC.

Either buffer, the GHB or a particular LHB, provides rich history information that can be used to trigger prefetches when an address is added to it. Various forms of pattern-detection logic can be fed by these buffers, for example:

1. Access Delta Inspection Logic: Inspects the deltas of *accesses* including both hits and misses in the recent past in a 64KB region around the latest access to see if there is a repeatable pattern. If so, prefetches for the next addresses following the pattern might be generated. A brute-force method is employed to find the repeatable pattern in the deltas. This method is shown in Algorithm 1. The algorithm attempts to find repeating sequences of deltas in the accesses and assigns them a score value based on how many times the pattern repeats.

   Example: With accesses (in terms of cache line numbers) of: 0x10, 0x12, 0x13, 0x15, 0x16, the deltas are: 2, 1, 2, 1. The prefetcher will detect this repeating pattern of 2, 1 and prefetch 0x18, 0x19, etc.

2. Miss Delta Inspection Logic: Inspects the deltas in the *L2 misses* in the recent past (if the buffer contains any) in a 64KB region around the latest access. If it can find a repeatable pattern in them, it prefetches the next few addresses.

3. Spatially-Ordered Delta Inspection Logic: Inspects the deltas of *accesses* in a region close to the latest access and orders them with respect to address space (as opposed to time, which is the default order). It then tries to issue prefetches if there is a repeatable pattern within these deltas and issues.

   Example: With accesses (in terms of cache line numbers) of: 0x11, 0x9, 0x10, 0x12, the sorted deltas are in fact (from 0x9) 1, 1, 1. The prefetcher will detect this unit stride and will prefetch 0x13, 0x14, etc.

   Note: This logic also detects patterns with negative deltas in them, i.e., it looks at recent accesses that are (in terms of access address) both less than and greater than the current access to determine if there is a repeatable pattern in the deltas.

4. Exponential-Stride Detection Logic: Inspects the deltas of *accesses* in a region close to the latest access to see if a multiplicative increasing stride is found. If detected, the next few addresses from this pattern are fetched.

   Example: With accesses (in terms of cache line numbers) of: 0x10, 0x12, 0x16, 0x1E, the deltas are: 2, 4, 8. The prefetcher will detect an exponential stride and will prefetch 0x2E, 0x4E, etc.

   Note: We employ slightly inexact pattern detection for this use case. For example, even though deltas (in terms of cache lines) of 6, 13, 26, 53 do not strictly follow an exponentially increasing series, they will be detected as such by the prefetcher. This inexact exponentially increasing stride is found in 429.mcf in SPEC2006.

5. Binary-Search Detection Logic: Inspects the deltas of the *accesses* in a region close to the latest access to see if a binary search pattern is present.

Example: With memory accesses with deltas (in terms of cache line numbers) of: -512, 256, 128, etc. the program is probably doing a binary-search-like pattern. In this particular case, a prefetch for cache line with deltas from the latest access of -64 and 64 can be made by the prefetcher.

Note: We did not implement this in the current prefetcher because of high logic complexity and low returns in terms of performance (since this is a very specialized pattern).

6. Noise-resistant Spatial Pattern Detection: Our regular brute-force pattern detection algorithm recognizes repeating deltas over the last N accesses that are stored in the history buffers. While this gives good performance at the cost of logic, the detection algorithm can be generalized further to detect patterns in the presence of seemingly random accesses that destroy the repeating pattern ("noise"). The source of such noise could be instruction scheduling optimizations that change the order, or perhaps branches into code that adds seemingly random memory accesses to an otherwise repeatable delta pattern. The pattern detection algorithm shown in Algorithm 1 can be generalized by taking the N recent accesses in a region, and then selecting M ($M < N$) accesses from the N. The "regular" brute force algorithm can then be run on the M entries. This is, of course, very costly since all possible M accesses (for all values of M) have to be sent to the pattern detection logic. Therefore we did not implement this and have left it as a future extension of our prefetcher.

   Example: With memory accesses with deltas (in terms of cache line numbers) of: 4, 1, 6, 2, 2, 6, 1, etc. and accesses of: **0**, **4**, 5, **11**, 13, **15**, 21, **22**, etc. if only certain deltas are chosen (corresponding to the bold accesses), a pattern of 4, 7, 4, 7 emerges with seemingly random addresses mixed in.

   Note: Recently, Ishii et al proposed AMPM (Access Map Prefetching Mechanism) in [3], which is a special case of the above pattern detection. They compute the delta between the latest access and all previous accesses and then see if this single delta repeats over and over. The above-described method is a generalization of their delta pattern detector and is costly in terms of logic but detects longer-than-one repeating delta sequences in the presence of random accesses.

In a single cycle, multiple prefetches may be generated (depending on how many memory accesses are made per cycle, among other things). If these prefetches are issued indiscriminately, precious resources such as bandwidth and MSHR entries could be oversubscribed. To prevent multiple prefetches to the same cache line from generating redundant requests, first, the GHB and LHBs are queried to make sure the prefetch about to be issued has not been accessed by the program before. In addition, a fully-associative buffer is checked before issuing the request. This buffer is populated with the most recent prefetches that have been made. Alternatively, the about-to-be-issued prefetch requests can be filtered against a Bloom Filter populated with the most recent cache accesses as Dimitrov and Zhou do [4].

The DPC rules allowed probing the cache to see if a cache line was present. Our submission actually probed the cache before sending in prefetches for filtering purposes (prefetch candidates that were already in the cache were rejected). In Section 8 we show

Table 1: Table listing the parameters of our prefetcher and the description of the said parameters.

Parameter	**Description**
n	Number of entries in the GHB
l	Number of LHBs
m	Number of entries in each LHB
agg	Number of "ahead" prefetches issued
ahd	If the prefetch bit is set, do we issue only the *agg*th prefetch?
cpok	Are prefetches filtered against the current cache contents?
fil	Number of entries in the prefetch filter

other design points of our prefetcher, some of which do not probe the cache before issuing prefetches.

5. DPC Submission Design Point

For our DPC submission design point, we chose [n=128, m=24, agg=4, ahd=0, $cpok$=1, fil=64]. Our prefetch logic contained exact pattern matching for repeating delta patterns and inexact pattern matching for exponentially increasing delta patterns. The prefetcher also detected exact repeating spatial patterns. We did not implement binary-search or noise-resistant pattern matching due to logic complexity. For our prefetcher filter, we chose to use a 64-entry fully associative buffer that stored the most recent cache accesses. In addition to the prefetch filter, only those prefetches were issued that were not present in the cache. The DPC rules allowed probing the cache to see if an address was present or not (by checking the return value of the GetPrefetchBit() function and comparing against -1). This design point gives a 20% performance improvement on average as shown in Figure 3. The L2 cache miss reduction of this prefetcher is shown in Figure 4 - on average 60% of L2 cache misses are reduced by our prefetcher.

6. Parameters and Storage

Our prefetcher design has several tunable parameters, some of which are shown in Table 1[5]. Our DPC submission consisted of a prefetcher at the design point: [n=128, m=24, l=32, agg=8, ahd=8, $cpok$=1, fil=64]. The following paragraph explains how this design point conforms to the DPC design rules in terms of budget cost.

The cost of GHB is 128 $\times$ 32 bits = 4096 bits. The cost of LHBs is 32 $\times$ (24+1) $\times$ 32 = 256000 bits[6]. In addition, there is a prefetch cache of 32 entries (32 $\times$ 32 bits = 1024 bits) to filter out redundant prefetches. Therefore, the total storage cost of the prefetcher is

5. Note: we have more tunable parameters than are shown here. For example, DPC submissions slightly varied the limit on the max variable given in Algorithm 1 before it issues the prefetches

6. Our prefetcher just uses the cache line address and not the full address, so we use 32 - 6 = 28 bits for addresses. 2 extra bits are used for storing the access type (i.e., L1 vs. L2), and storing whether or not it was a hit.

Table 2: Table listing the parameters that identify various design points and the performance results of our prefetcher at those design points.

n	***l***	***m***	***agg***	***ahd***	***cpok***	***fil***	**Perf**	**Cov**	**Notes**
128	24	32	4	0	1	64	20%	60%	Original DPC submission design point
128	24	32	8	0	1	128	22%	62%	Best performaning under DPC rules (temps not counted)
128	24	32	8	0	0	128	8%	61%	Not poking caches
128	24	32	8	1	0	128	17%	60%	Using prefetch bit
128	24	32	8	1	0	128	16%	52%	No LHBs
1024	24	128	8	0	1	128	19%	62%	Large budget without cache poking
1024	24	128	7	0	1	128	19%	61%	Large budget without cache poking
1024	24	128	6	0	1	128	19%	60%	Large budget without cache poking
1024	24	128	5	0	1	128	18%	60%	Large budget without cache poking
1024	24	128	4	0	1	128	17%	60%	Large budget without cache poking

1024 + 25600 + 4096 = 30720 bits. This is under the 4KB storage limit with 2048 bits left for temporary variables. With unlimited hardware complexity, all the temporary variables in the prefetcher can be incorporated as stateless logic[7].

There are three incarnations of the prefetcher that have been submitted. All three have a GHB size of 128, a LHB size of 24 with 32 entries. The cost of GHB is 128*32 bits = 4096 bits. The difference between these submissions is the aggressiveness of the prefetchers. Specifically, these prefetchers have different limits on the `max` variable given in Algorithm

7. We count temporary variables as variables that do not persist across cycles. These variables are created on-the-fly in the software prefetching code, and are cleared every cycle. The storage cost calculation only includes storage that persists across cycles. For example, our code requires a temporary sorted array of a small region of the GHB. This can be done by using a hardware sorting network and feeding it into the stride detector in a single cycle without storing it anywhere. Therefore its cost is not included in the cost calculation.

Table 3: Table detailing parameters of our simulated CPU.

Parameter	Value	Notes
Front End	Perfect	No fetch hazards
Window Size	128	
Issue Width	4	No restriction other than true dependencies
L1 Cache Latency	1	
L2 Cache Latency	20	
Memory Latency	200	
L1 Cache Size	32 KB	
L1 Cache Assoc.	32	
L2 Cache Size	2MB / 2MB / 512KB	Config 1 / 2 / 3
L2 Cache Assoc.	32	
L2 Cache Bandwidth	1000 / 1 / 1 accesses per cycle	Config 1 / 2 / 3
Memory Bandwidth	1000 / 0.1 / 0.1 accesses per cycle	Config 1 / 2 / 3

1 before they issue the prefetches. The cost of LHBs is 32*(24+1)*32=25600 bits[8]. In addition, there is a prefetch cache of 32 entries (32*32bits=1024 bits) to filter out redundant prefetches. Therefore, the total storage cost of the prefetcher is 1024+25600+4096 bits = 30720 bits. This is under the 4KB storage limit not counting the temporary variables. With unlimited hardware complexity, all the temporary variables in the prefetcher can be incorporated as stateless logic[9].

7. Simulation Methodology

For our simulations, we relied on a standard version of the prefetcher kit provided by the organizing committee of DPC-1 [5]. This kit contained PIN [6] and CMP$im[7] for generating and simulating traces. To generate instructional traces, 40 billion instructions were skipped and a trace 100 million instructions long was obtained[10]. An out-of-order model was used for the processor with a 128-instruction window. Table 3 details the other parameters of the performance model of the processor.

8. Our prefetcher just uses the cache line address and not the full address, so we use 32-6=26 bits for addresses. Two extra bits are used for storing the access type (i.e., L1 vs L2), and storing whether or not it was a hit.
9. We count temporary variables as variables that do not persist across cycles. These variables are created on-the-fly in the software prefetching code, and are cleared every cycle. The storage cost calculation only includes storage that persists across cycles. For example, our code requires a temporary sorted array of a small region of the GHB. This can be done by using a hardware sorting network and feeding it into the stride detector in a single cycle without storing it anywhere. Therefore, its cost is not included in the cost calculation.
10. 998.specrand, 999.specrand and 481.wrf did not have their first 40 billion instructions skipped because they did not have enough instructions in the program

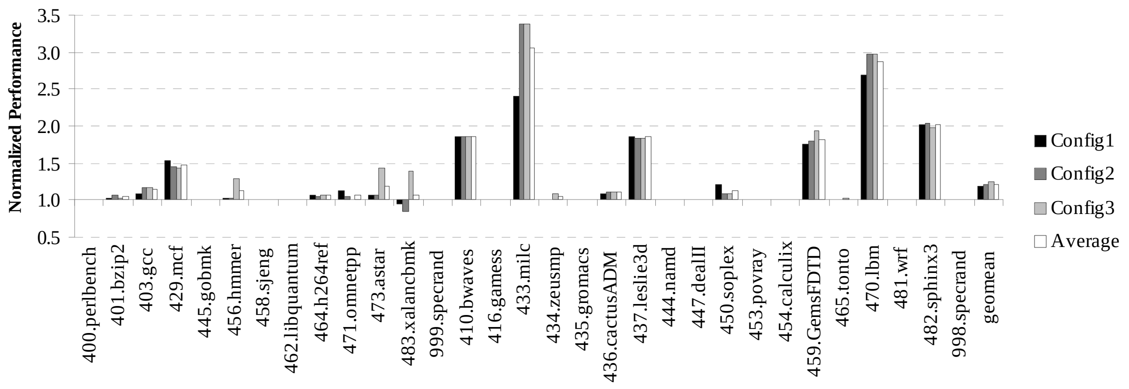

Figure 3: Performance simulation results of our prefetcher at the DPC design point: [n=128, m=24, l=32, agg=4, ahd=0, $cpok$=1]. The average of the three configurations is 20% as shown by the white bar on the right.

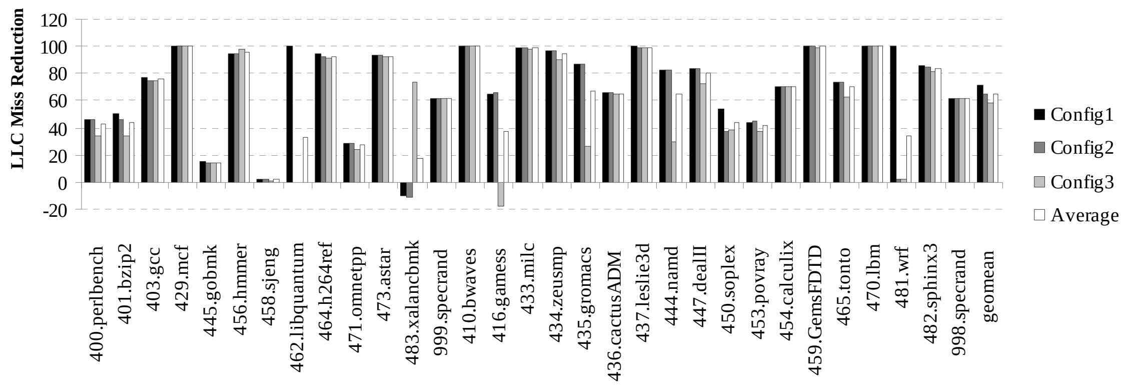

Figure 4: L2 cache miss reduction by using the prefetcher at the DPC design point: [n=128, m=24, l=32, agg=4, ahd=0, $cpok$=1]. The L2 cache misses are reduced by more than 60% on average.

7.1. Benchmarks Not Limited by the Memory Sub-system

In the SPEC2006 suite, there are a few benchmarks that are not limited by the memory sub-system according to our initial study. Benchmarks such as 447.dealII and 453.povray are examples that may be limited by the issue width or the number of floating point units on the simulated machine (the front-end of the machine had perfect branch prediction). For these benchmarks, it is desirable for the prefetcher to not interfere with the current cached contents. Our prefetcher does not degrade the performance of these benchmarks.

7.2. Streaming Benchmarks

Benchmarks like 470.lbm and 462.libquantum have streaming memory behavior. For such applications, since the access patterns are so simple and predictable, our predictor makes accurate predictions and reduces the last-level cache misses by a large percentage for configuration 1. The reason for the high number of LLC misses in configurations 2 & 3 for 470.libquantum is that multiple requests to identical addresses take up multiple slots in the L2 queue, which is a finite resource. This particular benchmark produces a lot of L2 misses to the same address, causing the L2 queue to be full most of the time. Our prefetcher in debug mode reported that most of its prefetch requests were being ignored by the simulator due to the L2 queue being full.

7.3. Pointer-Chasing Benchmarks

429.mcf is an example of a pointer-chasing benchmark. In at least one of its phases, it exhibits accesses whose deltas follow a geometric pattern. Our prefetcher can reduce a large number of LLC misses by correctly predicting the future access addresses as explained in the previous section.

7.4. Benchmarks Negatively Affected

One benchmark in the suite, 483.xalancbmk, was negatively affected by our prefetcher in configuration 1 and 2. The performance was reduced by 10% and the LLC misses were actually increased. This is because our prefetcher is too aggressive and often predicts erroneous addresses. These can pollute the cache and eat up into the fetch bandwidth. Given more storage and access to more microarchitecture modules like MSHRs, etc., we believe that we can improve the performance of benchmarks like 483.xalancbmk.

8. Other Design Points

Using the same simulation framework, we ran other simulations to observe the performance of our prefetcher at other design points. Table 2 summarizes the results at all the design points and the following figures show perforamnce results at selected design points.

Figure 5 shows the performance results at the design point: [n=128, m=24, l=32, agg=8, ahd=0, $cpok$=1]. Figure 6 shows the L2 miss reduction percentages. At this design point, without counting temporary variables, the prefetcher would still be under 4KB of state. This prefetcher also filers prefetch candidates against the cache contents as allowed by the DPC rules.

Figure 7 shows the performance results at the design point: [n=128, m=0, l=0, agg=8, ahd=1, $cpok$=0]. Figure 8 shows the L2 cache miss reduction percentages. This result shows that much of the performance improvement over the baseline can be obtained by using a GHB alone (without using LHBs). The LHBs, however, are required for obtaining the last few performance improvement percentage points.

At the "large-budget" performance point, aggression of the prefetcher from 4 to 8 does not vary the performance by that much as shown by the final few entries of 2.

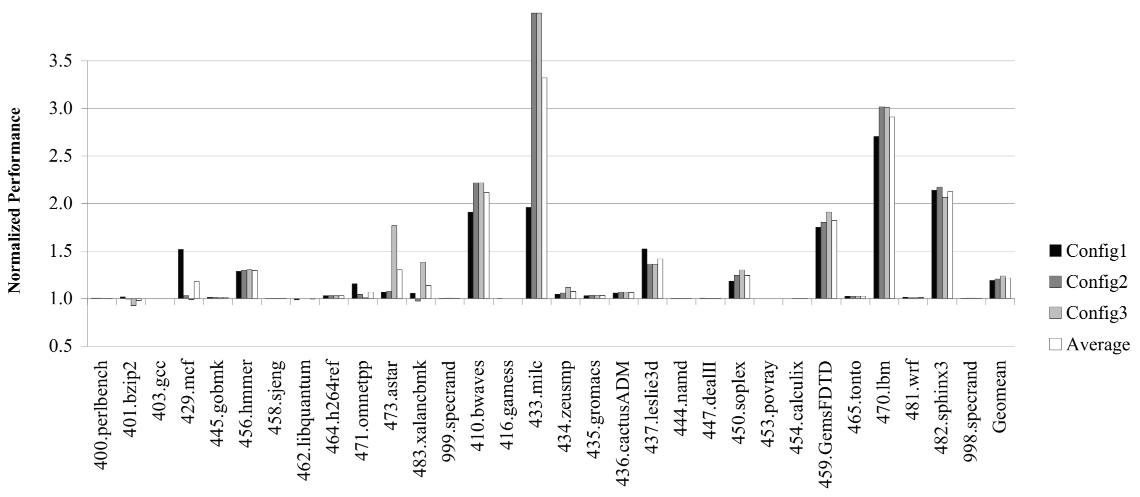

Figure 5: Performance simulation results of another design point with [n=128, m=24, l=32, agg=8, ahd=0, $cpok$=1]. The average of the three configurations is 22%, as shown by the white bar on the right.

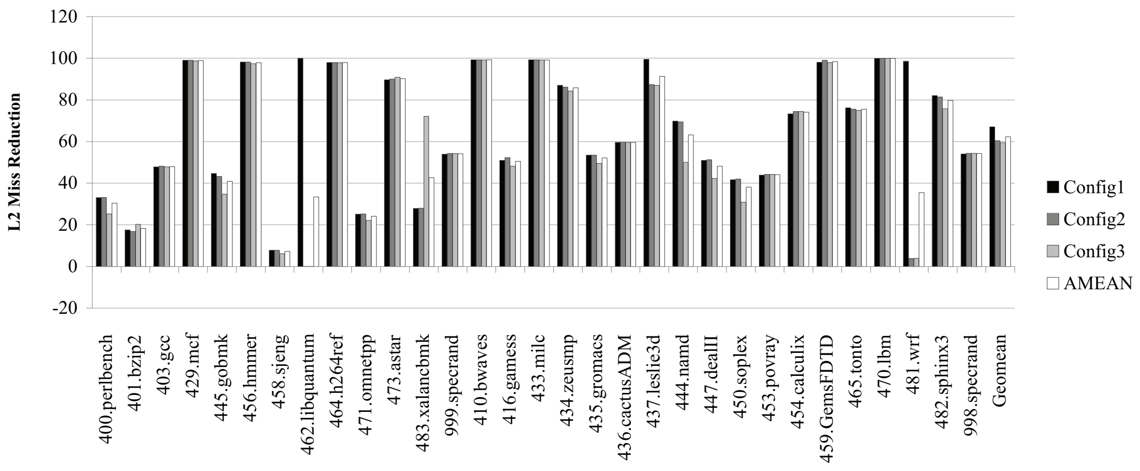

Figure 6: L2 cache miss reduction by using the prefetcher with design point: [n=128, m=24, l=32, agg=8, ahd=0, $cpok$=1].

9. Hardware Realization

Although in its present form, the logic and control part of this prefetcher can be costly to implement as-is in hardware, there could be a few changes to the prefetcher that could make it practical. These changes are beyond the scope of this paper, which only seeks to implement the best performing prefetching algorithm with a fixed budget size (thereby ignoring all hardware complexity). During the design of this prefetcher we came up with a

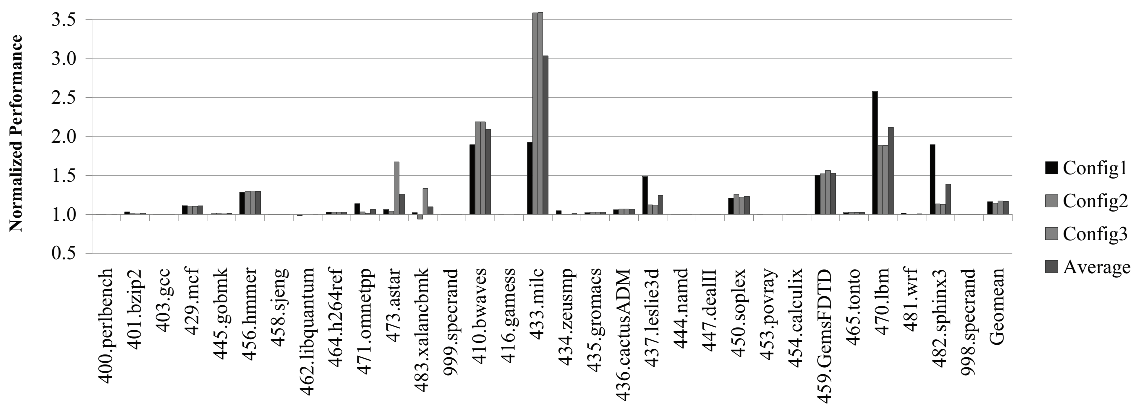

Figure 7: Performance simulation results of another design point with [n=128, m=0, l=0, agg=8, ahd=1, $cpok$=0]. The average of the three configurations is 16% as shown by the white bar on the right.

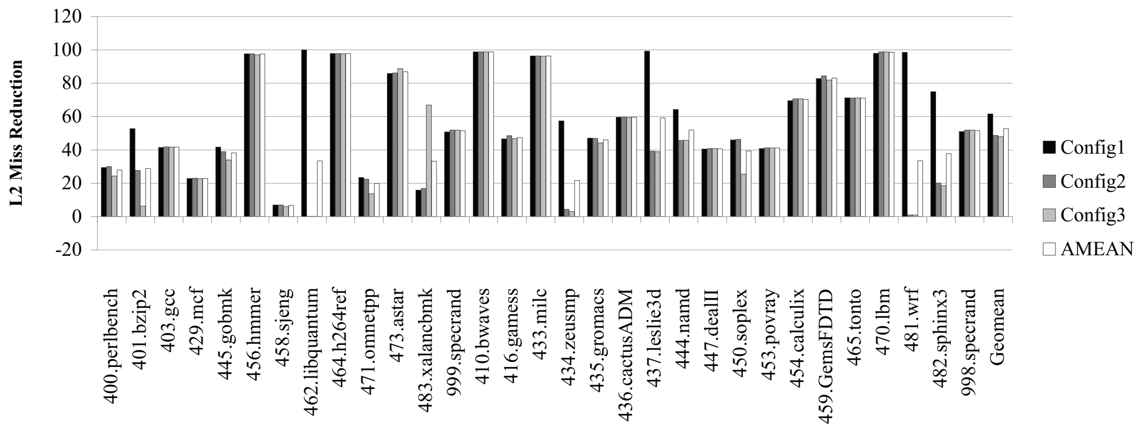

Figure 8: L2 cache miss reduction by using the prefetcher with design point: [n=128, m=0, l=0, agg=8, ahd=1, $cpok$=0]. THe L2 cache misses are reduced by 44% on average.

few ideas for making the prefetcher more hardware-friendly, but all those schemes took up more storage than what our proposed prefetcher currently uses.

10. Future Work

An exhaustive study of the design space for this prefetcher would be an interesting future study. In our paper we have mentioned two other methods for detecting patterns that could be used in future work as well. The binary search pattern could be useful for a few workloads while the noise-resistant repeating delta detector would be useful for branchintensive workloads. Researchers could also try to come up with schemes to efficiently map the logic we propose for our prefetcher into realizable hardware. Another unexplored aspect of this prefetcher would be adding some sort of adaptive control to the aggression of the prefetcher. This might help in benchmarks that are negatively affected by the prefetcher.

11. Conclusion

In this paper, we presented a prefetcher that can significantly reduce the number of last-level cache misses (over 60% on average for SPEC2006) by exploiting and understanding a variety of memory access patterns. It can effectively improve the performance of SPEC suite by about 20% on average using 4KB of bit-budget.

Acknowledgments

The authors would like to thank Dong Hyuk Woo for his valuable feedback during the design of the prefetcher. This work was supported in part by an NSF CAREER Award (CNS-0644096).

References

[1] K. J. Nesbit and J. E. Smith, “Data Cache Prefetching Using a Global History Buffer,” in *HPCA ’04: Proceedings of the 10th International Symposium on High Performance Computer Architecture*, (Washington, DC, USA), p. 96, IEEE Computer Society, 2004.

[2] C. D. Spradling, “SPEC CPU2006 Benchmark Tools,” *SIGARCH Comput. Archit. News*, vol. 35, no. 1, pp. 130–134, 2007.

[3] Y. Ishii, M. Inaba, and K. Hiraki, “Access map pattern matching for data cache prefetch,” in *ICS ’09: Proceedings of the 23rd international conference on Supercomputing*, (New York, NY, USA), pp. 499–500, ACM, 2009.

[4] M. Dimitrov and H. Zhou, “Combining local and global history for high performance data prefetching,” 2009.

[5] “DPC-1 Homepage,” 2008. http://www.jilp.org/dpc/.

[6] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation,” in *PLDI ’05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, (New York, NY, USA), pp. 190–200, ACM, 2005.

[7] A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob., “CMP$im: A Pin-based on-the-fly multi-core cache simulator,” in *Proc. Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS)*, (Beijing, China), pp. 28–36, ACM, 2008.