Path Confidence based Lookahead Prefetching

Jinchun Kim*, Seth H. Pugsley†, Paul V. Gratz*, A. L. Narasimha Reddy*, Chris Wilkerson† and Zeshan Chishti†

*Texas A& M University
cienlux@tamu.edu, pgratz@gratz1.com, reddy@tamu.edu

†Intel Labs
{seth.h.pugsley, chris.wilkerson, zeshan.a.chishti}@intel.com

***Abstract*—Designing prefetchers to maximize system performance often requires a delicate balance between coverage and accuracy. Achieving both high coverage and accuracy is particularly challenging in workloads with complex address patterns, which may require large amounts of history to accurately predict future addresses. This paper describes the Signature Path Prefetcher (SPP), which offers effective solutions for three classic challenges in prefetcher design. First, SPP uses a compressed history based scheme that accurately predicts complex address patterns. Second, unlike other history based algorithms, which miss out on many prefetching opportunities when address patterns make a transition between physical pages, SPP tracks complex patterns across physical page boundaries and continues prefetching as soon as they move to new pages. Finally, SPP uses the confidence it has in its predictions to adaptively throttle itself on a per-prefetch stream basis. In our analysis, we find that SPP improves performance by 27.2% over a no-prefetching baseline, and outperforms the state-of-the-art Best Offset prefetcher by 6.4%. SPP does this with minimal overhead, operating strictly in the physical address space, and without requiring any additional processor core state, such as the PC.**

I. Introduction

The "Memory Wall" [1], the vast gulf between processor execution speed and memory latency, has led to the development of large and deep cache hierarchies over the last twenty years. Although processor frequency is no-longer on the exponential growth curve, the drive towards ever greater DRAM capacities and off-chip bandwidth constraints have kept this gap from closing significantly. Cache hierarchies can improve average access times for data references with good temporal locality, however, they do not help to decrease the latency of the first reference to a particular memory address.

Prefetching is a well-studied technique which can provide an efficient means to improve the performance of modern microprocessors. The aim of prefetching is to proactively fetch useful cache lines from further down in the memory hierarchy, ahead of their first demand reference. Typically, prefetching techniques predict future access patterns based on past memory accesses. In essence, prefetching hardware speculates on the spatial and temporal locality of memory references, based on past program behavior.

In some earlier proposed prefetching techniques, the prefetching opportunity is limited to waiting until a cache miss occurs, and then prefetching either a set of lines sequentially following the current miss [2], a set of lines following a strided pattern with respect to the current miss [3], or a set of blocks spatially around the miss [4], [5]. More recent prefetchers attempt to predict complex, irregular access patterns [6], [7], [8], [9], [10]. While these methods show significant benefit, because they are inherently reactive, the depth of their speculation is limited, which can lead to untimely prefetches.

978-1-5090-3508-3/16/$31.00 © 2016 IEEE

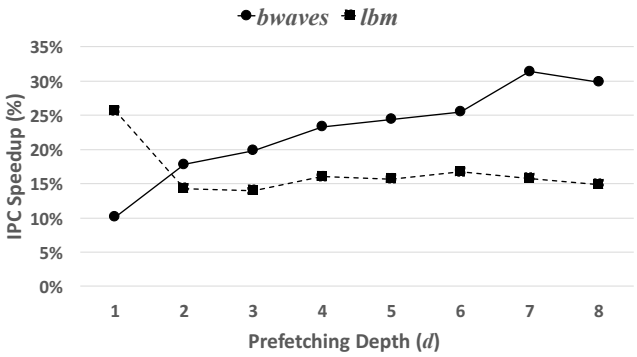


Fig. 1: Impact of prefetching depth on a simple PC-based delta prefetcher.

Increasing prefetch depth is one way to speculate deeper. For example, if a processor has a simple next line prefetcher that prefetches a (+1) delta ahead of the current demand cache line, we can build a more aggressive next *d* line prefetcher that prefetches $d*$(+1) deltas ahead of the current miss (*e.g.,* $+1_1$, $+1_2$, ..., $+1_d$). To generalize, a stride prefetcher whose delta distance (stride) is *N* and whose prefetching depth is *d* can be represented by (N_1, N_2, ..., N_d).

Naively increasing the prefetching depth, however, does not always improve overall system performance, because often the predicted reference stream and the actual reference stream will eventually diverge. Figure 1 shows the effect of prefetching depth on two different SPEC CPU 2006 benchmarks using a simple Program Counter (PC) based delta prefetcher [11]. As the prefetching depth *d* grows, the PC delta prefetcher brings more cache lines that are *dN* distance away from base address. In this figure, *bwaves* benefits from deeper prefetching until $d = 7$, because its memory access pattern is a predictable series of (+1) or (-1) deltas, up to seven steps ahead of the current demand access. After that point, the performance benefit drops as further speculative requests serve only to consume memory bandwidth, without contributing additional cache hits. On the other hand, *lbm* suffers from performance degradation as the prefetching depth grows. Since *lbm* has a variety of memory access patterns [9], deeper prefetching with a simple delta predictor wastes bandwidth, and pollutes

the cache. While deeper speculation is useful for *bwaves*, *lbm* shows the greatest benefit from a speculative depth of only $d = 1$. Thus, as Figure 1 illustrates, achieving performance across many workloads requires adapting speculation relative to its accuracy.

To address both prefetching coverage and accuracy, prior work has adopted lookahead mechanisms [9], [12], [13]. These studies, however, suffer from high hardware complexity [12], [13], or do not implement adaptive throttling [9]. Moreover, most hardware prefetchers work in the physical address space [4], [9], [10], [11], where the mapping between virtual and physical memory is not known. As a result, it is often difficult to predict patterns across 4KB physical page boundaries.

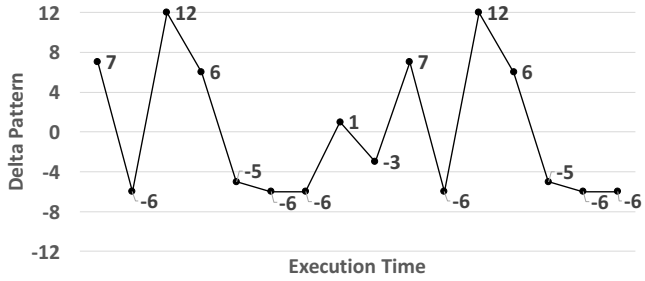
In this work, we propose a simple but powerful path confidence-based lookahead prefetcher, the Signature Path Prefetcher (SPP). The contributions of this work are:

- We introduce a signature mechanism that stores memory access patterns in a compressed form and initiates the lookahead prefetching process. Up to four small deltas can be compressed in this 12-bit signature without aliasing. By correlating the signature with future likely delta patterns, SPP learns both simple and complicated memory access patterns quickly and accurately. The signature can be also used to detect the locality between two physical pages, and continue the same prefetching pattern off the end of one physical page and onto the next.
- We develop a path confidence-based prefetch throttling mechanism. As lookahead prefetching goes deeper, a series of signatures builds a signature path. Each signature path has a different confidence value based on its previous delta history, prefetching accuracy, and the depth of prefetching. The path confidence value is used to throttle prefetching depth dynamically in order to balance prefetch coverage with accuracy.
- Unlike prior lookahead based prefetchers [12], [13], SPP does not require deep hooks into the core microarchitecture and is purely based on the physical memory access stream.

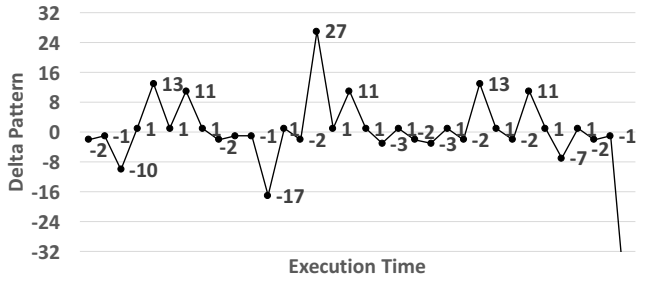
We evaluate SPP with a combination of SPEC CPU 2006 and commercial workloads and find it achieves an average 27.2% performance improvement compared to a baseline without prefetching. Moreover, SPP outperforms recent, best of class, lookahead and non-lookahead prefetchers [8], [9], [10], including the winner of the most recent data prefetching competition, by 6.4% on average. The remaining sections are organized as follows. Section II discusses the motivation for path confidence based prefetching. Section III describes the detailed hardware implementation of SPP. A detailed performance evaluation is presented in Section IV. Finally, we conclude the paper in Section V.

II. Motivation and Prior Work

In this section, we examine previously proposed prefetchers with an eye toward improvement. In particular, we note that complex data access patterns are difficult to predict without the use of recursive lookahead mechanisms, while existing lookahead-based prefetchers do not consider path confidence, instead prefetching to an arbitrary degree.



(a) *GemsFDTD*: Complex but predictable pattern



(b) *mcf*: Complex and random patterns

Fig. 2: Complex memory access pattern.

A. Prefetching complex access patterns

An optimal prefetching algorithm should cover a wide range of memory access patterns. Simple stride prefetching techniques only detect sequences of addresses that differ by a constant value and fail to capture diverse delta patterns [14]. For example, Figure 2 shows two examples of complex memory access patterns, taken from *GemsFDTD* and *mcf*, which cannot be captured by a simple prefetcher. Though both show complicated patterns, *GemsFDTD* (Figure 2a) has a repeating sequence of strides (+7, -6, +12, +6, -5, -6, -6), that should be predictable assuming the prefetcher can store a long delta history. Concatenating such a long sequence in a simple pattern table, however, could result in huge storage overhead.

On the other hand, Figure 2b shows that *mcf* has a random (though biased) access pattern that is difficult to predict. In this particular case, it is better to use a simple next-line prefetcher, because (+1) is the most commonly seen delta pattern. Offset based prefetchers such as the Best Offset prefetcher [10] and the Sandbox prefetcher [15] evaluate multiple offsets at runtime and issue prefetches with an offset that maximizes likelihood of use. These offset prefetchers do not, however, account for temporal ordering between delta patterns, and suffer from low accuracy on complex, yet predictable address patterns. In addition, if there are multiple offsets that are commonly observed during program execution, offset prefetchers take longer to train or fail to select the optimal offset. Lookahead prefetchers [9], [12] efficiently encode the relationship between accesses to yield future predictions, enabling further speculative lookahead accesses.

B. Adapting Aggressiveness

Lookahead prefetchers learn patterns by collecting histories of observed data access patterns, and correlating these with the

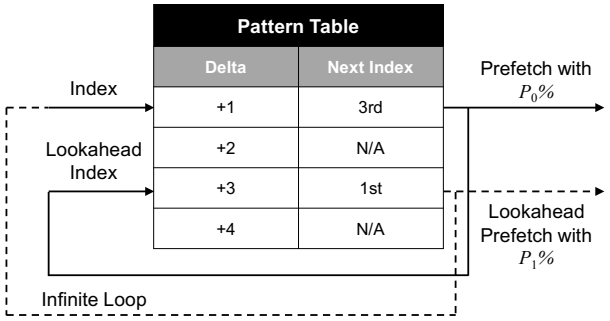


Fig. 3: A case of runaway lookahead prefetching caused by an infinite loop in the prediction pattern table.

next expected delta in the pattern. Figure 3 shows an example lookahead prefetcher that recursively refers to a pattern table to generate future prefetches. In this example, the prefetcher indexes into the pattern table to find the next predicted delta for prefetching. Once this prefetch is issued, the prefetcher recursively uses that prediction to again index into the pattern table and generate further predictions. This recursion allows lookahead prefetchers [9], [12] to prefetch far ahead of the current program execution, and generate timely prefetches for as long as their predictions remain accurate.

In principle, the lookahead process can be repeated as long as the predicted pattern is found in the pattern table. As shown in Figure 3, delta (+1) predicts the 3rd index (+3), and delta (+3) predicts the 1st index (+1), forming a loop. While the loop here may persist for many iterations, it is unlikely to persist forever, thus the true desired prefetching depth must be limited. To avoid over-prefetching, existing lookahead prefetchers [12], [9] globally and statically limit the depth to which lookahead is pursued ahead of the current demand access stream. Unfortunately, it is often the case that the ideal prefetch depth varies from application to application, as shown in Figure 1, and even varies between prefetch streams within the same application. Thus a per-prefetch stream throttling mechanism is critical to adapt prefetch aggressiveness to the stream's prediction confidence [16].

C. Prefetching and Page Boundaries

Virtual memory is a vital tool in the operation of modern computers, but it creates unique challenges when designing data prefetchers which work in the lower-levels of the cache. Two addresses which are contiguous in the virtual address space might be separated by great distances in the physical address space. By extension, patterns which are trivially easy to detect in the virtual address space might be nearly impossible to fully detect in the physical address space.

Prefetchers are often located alongside caches where they have no access to address translation hardware, and hence they do not have knowledge of the relationship between physical pages in the virtual address space. Thus, data access patterns which cross physical page boundaries are difficult to exploit. Physical address prefetchers often try to learn intra-page or global patterns, applying these patterns to new pages that are encountered, or they discover simple patterns, like streams or strides, by considering each new page in a vacuum. This has the effect of either ignoring complex patterns altogether, as in the case of the Best Offset prefetcher [10], or requiring long, per-page, warmup times, as in Access Map Pattern Matching [8] and conventional stream prefetchers. All of these prefetchers must stop prefetching once a page boundary is reached, because it is impossible to know the physical mapping of the next page in the virtual address space. Although challenging to implement, an effective prefetcher should be able to seamlessly continue complex patterns detected in one page as they cross page boundaries, without having re-detect the pattern from scratch in the new page.

D. Other Prior Prefetchers

Somogyi *et al.* proposed Spatial Memory Streaming (SMS) [4] which leverages the correlation between memory request instruction addresses (PC) and access patterns spatial near the current memory request. This purely spatial pattern ignores the temporal ordering between future demand accesses. Later efforts extended this approach to detect the temporal order between delta patterns [6], [7]. While these prefetchers achieve good performance, they require megabytes of hardware state storage, which is orders of magnitude more than the other prefetchers considered in this paper.

III. Design

To address the deficiencies outlined in Section II, we propose the Signature Path Prefetcher (SPP), a novel, low-overhead and accurate lookahead prefetcher.

A. Design Overview

SPP uses a speculative mechanism we refer to as "lookahead" to increases its prefetching degree and improve timeliness. Once a delta prediction has been made by SPP, and a prefetch request has been issued, the accumulated history used to make the prediction(*i.e.,* the signature) is speculatively extended to include the predicted delta, thereby generating a new lookahead signature (the mechanism is discussed in detail in Section III-B). This signature can then be used to make a new delta prediction, which leads to producing yet another signature. Thus, we use predicted deltas, with no confirmation of their accuracy, to recursively speculate down a "*signature path*." These speculative signatures resemble techniques used in branch prediction to deeply speculate beyond unresolved branches. Relying on this mechanism, SPP can continue to speculate much further than the most recently confirmed delta. In fact, with this mechanism in place, the challenge shifts from generating new delta predictions to deciding when to stop.

From the discussion in Section II-B of Figure 3, we can conclude that lookahead prefetches will tend to be less accurate than initial prefetches. In general, as the number of speculative deltas increases, our confidence in the prefetch requests should decrease as the product of the confidence of each speculative prefetch along the path. To illustrate this, assume the probability of the first $delta_0$ prediction being accurate is p_0 and the probability of the second $delta_1$ is p_1. During speculative lookahead, the prefetch resulting from $delta_1$ has a probability of being accurate of $p_0 * p_1$. This

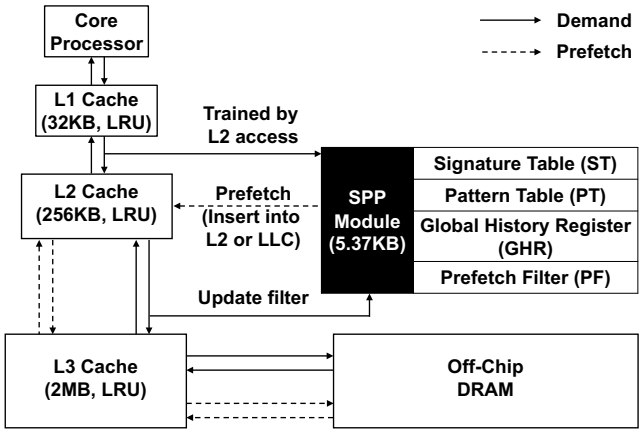


Fig. 4: Overall SPP architecture.

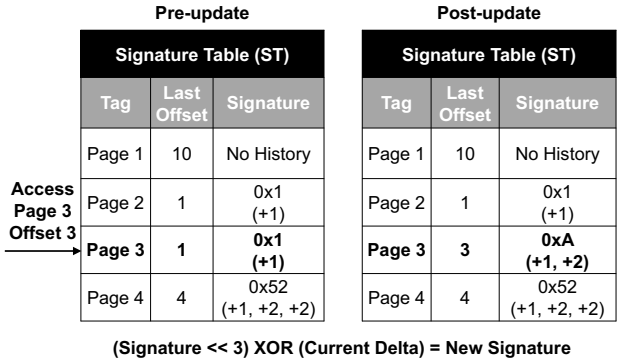
is because the path used to select $delta_1$, contains $delta_0$ which itself is only accurate with probability p_0. In fact, the probability that the delta predicted at any lookahead depth of d will produce a useful prefetch is a product of the probabilities of all prior speculative deltas ($\prod p = p_0 * p_1 * p_2 * ... * p_d$). We label this path probability P_d as the product of the probability of all constituent deltas to depth d such that $P_d = \prod p$. Further, the probability of any path of depth d can be expressed as a function of the probability of the most recent delta and the probability of the previous path, as shown in Equation 1:

$$P_d = p_d \cdot P_{d-1} \tag{1}$$

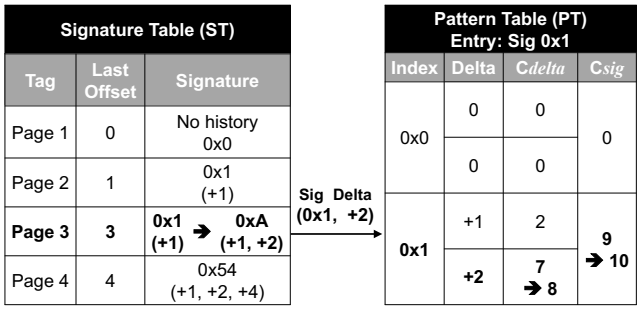
P_d can be compared against a given confidence threshold to adaptively determine when prefetching for this particular lookahead stream should be stopped. Further, this confidence, P_d, can also be used to determine which cache level to insert prefetched lines, with more accurate/confident prefetches sent to a higher cache level and less accurate/confident prefetches sent to a lower cache level.

To gracefully handle the transition between physical pages, SPP shares learning across page boundaries. As discussed in Section II-C, SPP operates in the physical address space, without any access to address translation. Thus, the spatial relationships between physical pages are largely unknown and assumed to be random. Despite this, SPP leverages learning from one physical page to make timely predictions in other pages. SPP does this in two ways, first, while delta history signatures are maintained on a per-physical page basis, as will be described in Section III-B, those signatures index into a global table for predicting deltas, which is shared by all pages. Second, as described in Section III-D, when the first demand access is made to a new physical page, the delta signature patterns from a previous physical page whose delta predictions crossed a page boundary can be inherited and used to bootstrap predictions in this new page. This gives SPP the advantage of not requiring long, per-page warmup periods to start prefetching complex patterns, which leads to higher prefetch coverage.

The high level design of the SPP engine is illustrated in Figure 4. The SPP module consists of three main structures



(a) Update signature



(b) Update delta pattern

Fig. 5: SPP table update operations.

(Signature Table, Pattern Table, and Prefetch Filter) and a small Global History Register (GHR), which is used for cross-page boundary bootstrapped learning. SPP is trained by L2 cache accesses, and can fill prefetch requests into either the L2 or LLC (depending on the confidence of the predicted prefetch). The Signature Table (ST) tracks the 256 most recently used pages, and stores the history of previously seen delta access patterns in each page as a compressed 12-bit signature. The Pattern Table (PT) is indexed by the history signatures generated by the ST and stores predicted delta patterns. The PT also estimates the path confidence that a given delta pattern will yield a useful prefetch. If the delta generated by the PT is found to have sufficient confidence (above a configured threshold), then it is passed as a prefetch candidate to the Prefetch Filter (PF), which checks for redundant prefetches. If the predicted delta crosses a 4KB physical page boundary, SPP does not issue the prefetch but instead redirects the request to the GHR for page boundary learning. In the remainder of this section we describe each stage of the SPP in detail.

B. Learning Memory Access Patterns

The ST, shown in Figure 5a, is designed to capture memory access patterns within 4KB physical pages, and to compress the previous deltas in that page into a 12-bit history signature. The ST tracks the 256 most recently accessed pages, and stores the last block accessed in each of those pages in order to calculate the delta signature of the current memory access,

which is then used to access and update the PT. Whenever there is an L2 cache access, its physical address is passed to the ST to find a matching entry for the corresponding physical page. Figure 5a shows an example of accessing the ST with a physical page number of 3 and block offset 3 from the beginning of the page. In this case, the ST finds a matching entry for this page and is able to read a stored signature 0x1. This signature is a compressed representation of a previous access pattern to that page, which was generated via a series of XORs and shifts as shown in Equation 2. In this case, the signature 0x1 represents a single previous delta access to Page 3, which was (+1).

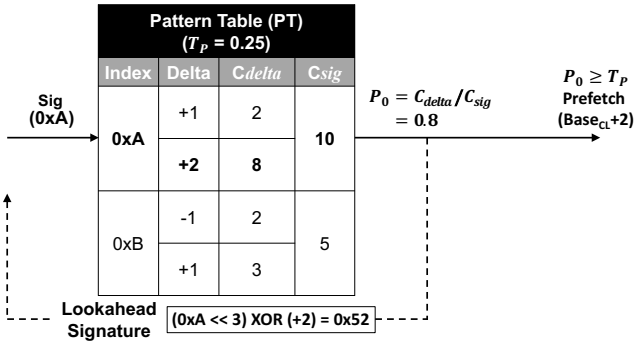
$$\begin{aligned} &\textit{New Signature} \\ &= (\textit{Old Signature} << \textit{3-Bit}) \text{ XOR } (\textit{Delta}) \end{aligned} \tag{2}$$

Since the ST stores the last block offset 1 accessed in Page 3, we know that the current delta in Page 3 is (3 - 1) = (+2). This delta is non-speculative, because it is based on a demand request to the L2. Therefore, we can infer that a given set of accesses (signature 0x1 in this instance) will lead to a delta of (+2). Figure 5b shows how this correlation is used to update the delta pattern in the PT.

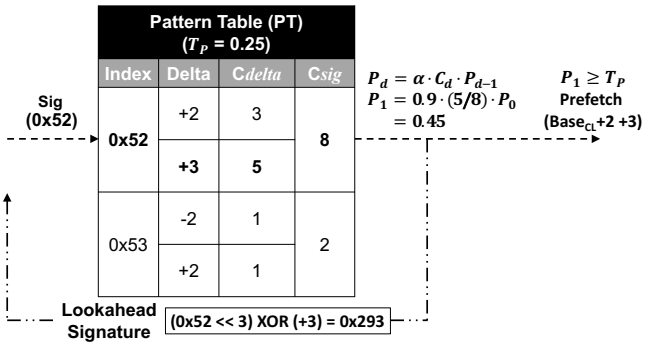
The probability of each delta occurring is approximated using a per-delta confidence value C_d, which is derived from counters stored in the PT. Since a matching delta (+2) that corresponds to 0x1 is found in the PT, the corresponding C_{delta} counter increases by one. In order to estimate the prefetch accuracy probability for each delta, we also maintain an separate counter C_{sig}, which tracks the total occurrences of the signature itself. If either C_{delta} or C_{sig} saturates, all counters associated with that signature are right shifted by 1. In doing so, SPP is able to continue updating its counters with the most recent information without ever completely losing all previously collected history. If the PT contains no matching delta, we simply replace the existing delta with the lowest C_{delta} value.

Unlike the ST, whose entries correspond to individual physical pages, each PT entry is shared globally by all pages. If Page A and Page B, for example, share the same access pattern, they will generate the same signature, which indexes to the same entry in the PT, and updates the same delta pattern in the PT. Sharing patterns between pages in the PT minimizes SPP training time as well as the number of entries needed to store predictions. Each entry in the PT can hold up to four different deltas so that multiple different deltas can be prefetched by a single signature. Each of the deltas in a PT entry can be prefetched if its corresponding probability ($C_d = C_{delta}/C_{sig}$) is above the given prefetching threshold T_P.

After updating the PT, the ST is also updated with a new signature based on the current delta (+2). Equation 2 shows how the SPP generates a new history signature. The old signature 0x1 is left-shifted 3-bits and XORed with the current delta (+2). In this way, a 12-bit signature can represent the last four memory accesses in Page 3. Note that we refer to this signature as being "compressed" because deltas greater than 7 have the potential to overlap and cause aliasing with existing



(a) Prefetch (Lookahead depth d=0)



(b) Lookahead prefetch (Lookahead depth d=1)

Fig. 6: Path confidence-based lookahead prefetching.

history. At this point, the new signature (0x1<<3) XOR (+2) = 0xA represents the current access pattern, (+1,+2), in Page 3. Assuming 4KB pages with 64B cache lines, all possible deltas fall in the range of (-63) to (+63). We use a 7-bit sign+magnitude representation for both positive and negative deltas. Thus, negative and positive deltas produce different signatures, pointing to different entries in the pattern table, and ultimately different prefetch targets.

C. Path Confidence-based Prefetching

After updating the ST, as in Figure 5a, SPP accesses the PT so that it can predict the next delta following signature 0xA. As shown in Figure 6a, the path confidence P_d is calculated for deltas associated with signature 0xA. The initial path confidence P_0 is simply set by (C_{delta} / C_{sig}) since there is no prior path confidence value. In this example, (+2) delta has $P_0 = 0.8$ which is greater than the prefetching threshold T_P. Therefore, the PT adds the delta (+2) to the current cache line's base address and issues a prefetch request.

In addition, the PT initiates the lookahead process by building a speculative *lookahead signature*. As shown in Figure 6a, the lookahead signature 0x52 is generated from 0xA and the predicted delta (+2) using Equation 2. While there could be multiple candidates for prefetching, SPP only generates a single lookahead signature, choosing the candidate with the highest confidence. The lookahead signature is used to index the PT again so that SPP can search for further prefetch and lookahead candidates down the signature path. If the lookahead signature 0x52 finds more prefetch candidates in the PT (Figure 6b), the process will be repeated and

prefetch requests will be issued. The lookahead mechanism is used recursively until the signature path confidence P_d falls below the prefetching threshold T_P. Note, a separate, greater confidence threshold T_f determines which level of the cache a given prefetch will be fill into, either the L2 if P_d is greater than T_f or the LLC if P_d is less than T_f.

SPP also uses a global accuracy scaling factor α based on the observed global prefetching accuracy to further throttle down or increase the aggressiveness of the lookahead process. Thus, Equation 1 is modified to include α according to Equation 3:

$$P_d = \alpha \cdot C_d \cdot P_{d-1} \quad (\alpha < 1) \tag{3}$$

If the prefetching accuracy is generally high across all prefetches, α will decrease the path confidence P_d very slowly, allowing deeper lookahead prefetching. On the other hand, if global prefetching accuracy is low, α will quickly throttle down lookahead prefetching. The prefetching accuracy is tracked by the PF described in section III-E. The global α and the local, per-delta C_d work together to provide a throttling mechanism which globally adapts to the general prefetching confidence of a given program phase, while simultaneously favoring some signature paths over others based on their relative confidence.

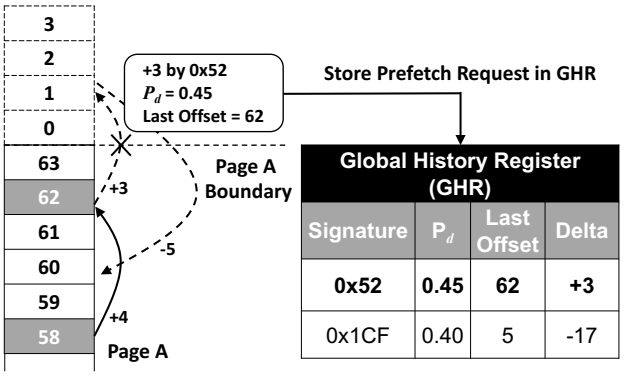
Note that for simplicity, in the discussion above we describe path confidence calculations in the context of floating point numbers. However, in a real implementation (and in our simulator), we use 7-bit fixed point numbers to represent path confidence values between 0∼100, and perform multiplication and division on those fixed point numbers. In addition, since C_{delta} and C_{sig} are 4-bit saturating counters, we can use a simple 16x16=256 entry lookup table that stores all possible division results, which allows us to completely remove the expensive divider modules. Moreover, the extra computational latency can be hidden, because SPP can calculate the path confidence in the background while the L2 cache is waiting for the DRAM to service demand misses.

In addition to throttling when P_d falls below the prefetch threshold T_p, SPP also stops prefetching if there are not enough L2 read queue resources. Reserving L2 read queue entries is desirable because even accurate, useful prefetches can take resources away from even more performance-critical demand misses from the L1 cache. Therefore, SPP does not issue prefetches when the number of empty L2 read queue entry becomes less than the number of L1 MSHRs. To summarize, SPP stops prefetching if the prefetcher observes one of following conditions:

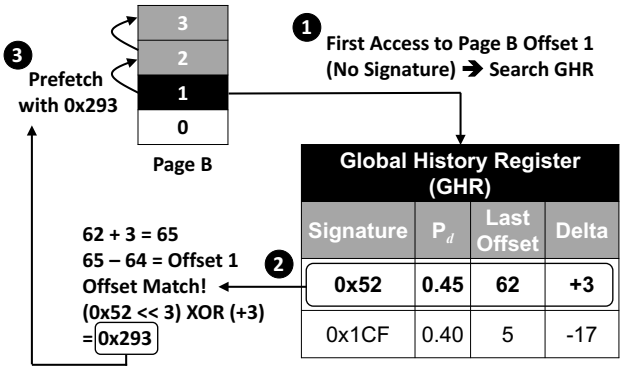
1) Low path confidence P_d.
2) Too few L2 read queue resources.

D. Page Boundary Learning

One of the challenges for history-based prefetchers like SPP is the loss of history information that occurs during transitions to new physical pages. Pages that are adjacent in the virtual address space may not be adjacent in the physical address space. As a result, patterns that are very easy to learn and follow in the virtual address space may be very hard to detect and predict when they are broken up by a physical page



(a) GHR stores prefetch request that crosses a page boundary



(b) Immediately start prefetching for new page

Fig. 7: Learning delta patterns across page boundaries.

transition. Previously proposed history-based techniques [4], [9] address this by making predictions using the first offset in a page, or with very short delta histories. Although these predictions may increase coverage, they do so at the expense of accuracy because they are made with little or no information about what happened in the previously accessed page.

SPP addresses this problem with a novel mechanism that allows histories to be maintained and tracked across physical page transitions. It does this by augmenting the per-page ST with a global history that is updated when the prefetcher makes a prediction off the end of a page, and is checked against when accessing new pages for the first time.

Figure 7 shows how SPP connects a signature path across physical page boundaries without the need to relearn any delta history patterns, or do any other warmup in the new page. As shown in Figure 7a, when there is a prefetch request that goes beyond the current Page A, a conventional streaming prefetcher must stop prefetching, because it is impossible for the prefetcher to predict the next physical page number. However, this boundary-crossing prediction can still be useful when the next page is accessed for the first time, and we find that the page offset of that first access matches the out-of-bounds offset previously predicted by SPP.

In order to track this behavior, the boundary crossing prediction is stored in a small 8-entry GHR. The GHR stores the current signature, path confidence, last offset, and delta used for the out-of-page prefetch request, which is everything necessary to bootstrap SPP prefetching in a new page. If we

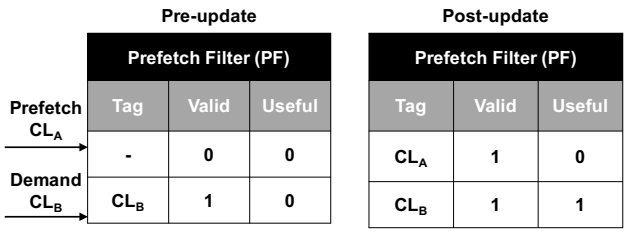


Fig. 8: Prefetching Filter.

access a new page that is not currently tracked (*i.e.,* a miss in the ST), SPP searches for a GHR entry whose last offset and delta match the current offset value in the new page. Figure 7b shows that Page B is accessed with an initial offset of 1. SPP checks if any GHR entry's last offset and delta value match the current offset of 1. In this case, the signature 0x52 has the last offset and delta whose sum (62 + 3 = 65) matches the offset of 1 in Page B, since there are only 64 64B blocks in 4KB physical page. We can now predict that Page B will produce a delta pattern that is a continuation of the signature 0x52.

Since the pattern predicted by signature 0x52 is now continuing in a new page, we need to connect the signature 0x52 with delta (+3) by generating a new signature. Using the same Equation 2, we generate a new signature 0x293 for Page B that can begin prefetching immediately, without needing to learn any additional delta history. The new signature 0x293 is entered into the ST for future use by Page B. Thus, SPP does not suffer from long, per-page warmup periods, and it can prefetch complex patterns in new physical pages sooner, resulting in higher prefetch coverage. Unlike the Global History Buffer [17], which records all observed delta patterns, the GHR only stores delta patterns that cross page boundaries. Note that SPP does not stop looking even further ahead for more prefetch candidates after coming to the end of a page and updating the GHR. As shown in Figure 7a, if the (+3) delta, which lands in Page B, is predicted to be followed by a (-5) delta, which comes back to Page A, then SPP can still exploit this behavior and prefetch an offset of 60 in Page A.

E. Prefetch Filter

The main objectives of the PF, shown in Figure 8, are to decrease redundant prefetch requests, and to track prefetching accuracy. The PF is a direct-mapped filter that records prefetched cache lines. SPP always checks the PF first, before it issues prefetches. If the PF already contains a cache line, this means that line has already been prefetched, and SPP drops the redundant prefetch request. Entries in the filter get cleared by resetting a *Valid* bit when the corresponding cache line is evicted from the L2 cache.

Due to collisions, a filter entry may already be occupied by another prefetched cache line. In this case, SPP simply replaces the old cache line, stores the current prefetch request in the filter, and issues the current prefetch. Note that this simple replacement policy might erase cache lines from the filter before they get evicted from the L2 cache, which could lead to re-prefetching, but we find in practice that this happens very infrequently.

By adding a *Useful* bit to each filter entry, the PF can also approximate prefetching accuracy. SPP has two global counters, one which tracks the total number of prefetch requests (C_{total}), and the other which tracks the number of useful prefetches (C_{useful}). The C_{total} counter increases whenever SPP issues a prefetch that is not dropped by the filter. Useful prefetches are detected by actual L2 cache demand requests hitting in the PF, which increments the C_{useful} counter. To avoid increasing C_{useful} more than once per useful prefetched line, we set a used bit in the PF entry which keeps it from being double counted. The global prefetching accuracy tracked by this filter is used for α in Equation 3 to throttle the path confidence value.

IV. Evaluation

In this section, we evaluate the SPP prefetch engine. We first present the evaluation methodology, followed by single core and multi-core performance. Finally, we present a sensitivity study of SPP's design parameters.

A. Methodology

We evaluate SPP using the ChampSim simulator, which is an updated version of the simulation infrastructure used in the 2nd Data Prefetching Championship (DPC-2) [18]. We model 1-4 out-of-order cores, whose parameters can be found in Table I. ChampSim is a trace-based simulator, and we collect SimPoint [19] traces from 18 memory intensive SPEC CPU2006 [20] applications using PinPlay [21]. We also collect single thread traces from 3 server workloads (Data Caching, Graph Analytics, SAT Solver) from CloudSuite [22]. Since our SimPoint methodology does not work with the server workloads, we instead collect the server workload traces after fast-forwarding at least 30B instructions to pass through the benchmark's initialization phase. For performance evaluation, we warm up each core for 200M instructions and collect results over an additional 1B instructions.

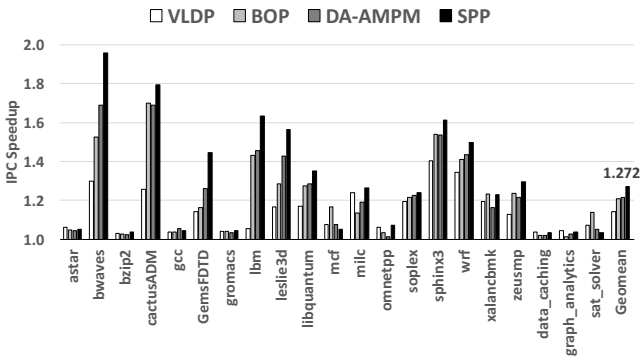
Single core simulations use a single DRAM channel. In multi-core simulations, all cores share a single L3 cache and main memory system, with two DRAM channels. Instruction caching effects are not modeled in this simulation infrastructure. In ChampSim, all prefetching actions are initiated by an L2 access, but prefetches can be directed to fill in either the L2 or the L3 cache. Our simulation infrastructure uses a 4KB page size when mapping virtual to physical addresses. In ChampSim, virtual to physical page mappings are arbitrarily randomized. All of the prefetchers we evaluate in this work were designed to operate strictly in the physical address space with no knowledge of the relationship between physical and virtual address spaces.

We compare SPP against three top performing recently proposed prefetching algorithms: the Variable Length Delta Prefetcher (VLDP) [9], the Best Offset Prefetcher (BOP) [10], and the DRAM-Aware Access Map Pattern Matching (DA-AMPM) [23] prefetcher[1] . We use the original code for

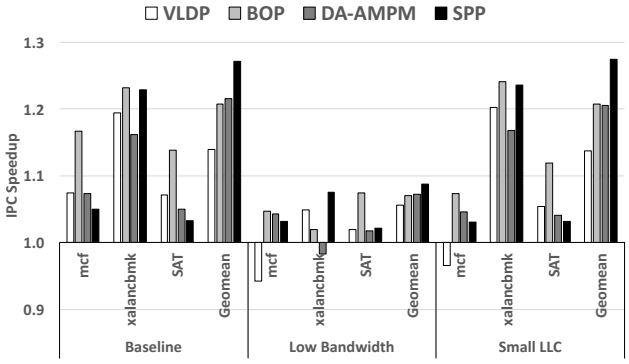
[1]Note: DA-AMPM is an extended version of AMPM [8] which accounts for DRAM row buffer locality.

Core Parameters	1-4 Cores, 3.2 GHz 256 entry ROB, 4-wide 64 entry scheduler 64 entry load buffer
Branch Predictor	16K entry gshare 20 cycle mispredict penalty
Private L1 Dcache	32KB, 8-way, 4 cycles 8 MSHRs, LRU
Private L2 Cache	256KB, 8-way, 8 cycles 16 MSHRs, LRU, Non-inclusive
Shared L3 (LLC)	2MB/core, 16-way, 12 cycles, LRU, Non-inclusive
Main Memory	1-2 64-bit channels 2 ranks/channel, 8 banks/rank 1600 MT/s

TABLE I: Simulator parameters.



(a) IPC speedup versus no prefetching.



(b) IPC speedup for Low BW and small LLC.

Fig. 9: Single-core IPC speedup.

each of these prefetchers submitted to DPC-2. In each case, their design parameters have been re-tuned to attain their highest performance in ChampSim running these traces. SPP does not use any feature of ChampSim that was not also available to the designers of the other evaluated prefetchers. SPP's threshold configurations were empirically derived. The prefetching threshold T_P = 0.25 and fill level threshold T_F = 0.9 were found to provide good performance improvement and accuracy, and they are used throughout these results except where otherwise noted.

B. Single Core Performance

Figure 9 shows the IPC speedup of all four evaluated prefetchers over a no-prefetching baseline. Overall, **SPP** outperforms or matches all other prefetchers on nearly every benchmark. On average, **SPP** achieves a 27.2% geometric mean speedup, which is 6.4% and 5.6% more than **BOP** and **DA-AMPM** respectively. Also, **SPP** outperforms **VLDP** (a recent lookahead prefetcher) by 13.2%. **SPP** shows particularly significant improvement in benchmarks that have complex access patterns. For example, both *GemsFDTD* and *lbm* show substantial performance improvement with **SPP**. Further, **SPP** shows remarkable performance gains in *bwaves, cactusADM, leslie3d,* and *libquantum*. These applications are highly predictable by SPP, and gain significantly from SPP's ability to adaptively look ahead very deeply.

As Ferdman *et al.* [22] noted in a prior study, data prefetching for large scale workloads is not as effective as prefetching for general purpose workloads. Performance with **SPP** only improves by 2∼4% for this class of benchmark. However, **SPP** and **VLDP** show the best performance for Data Caching and Graph Analytics, because both prefetchers are able to capture the complex memory access patterns found in those workloads. **BOP** achieves a 13.8% performance gain in *SAT solver* by aggressively prefetching on every L2 cache access. The impact of aggressive prefetching by **BOP** will be discussed further in the next section. We also modeled and compared the performance of the SMS [4] prefetcher originally designed for server workloads, however, the overall performance of SMS on both SPEC and CloudSuite was less than that of DA-AMPM. To simplify the performance analysis, we only include the results from DA-AMPM.

Figure 9b shows the performance improvement under different memory resource constraints. The baseline configuration used for Figure 9a has 12.8GB/s of DRAM bandwidth and 2MB LLC. The low bandwidth test is configured with only 3.2GB/s memory bandwidth, and the small LLC configuration has only 512KB of last level cache. Along with a geomean across all benchmarks, Figure 9b also shows performance for three benchmarks where **SPP** has lower performance than other prefetchers. As shown in Figure 9b, due to aggressive prefetching, **VLDP** and **BOP** show similar or worse performance improvement for *mcf* and *xalandbmk* under the low bandwidth configuration. Meanwhile, **BOP** always shows the best performance for *mcf* and *SAT solver* regardless of resource constraints. We find that these benchmarks have random access patterns (*i.e.,* pointer chasing) that cannot be easily captured in the physical address space. Therefore, offset-based prefetchers [10], [15] work better than **SPP** for these, because they are trained by individual offset occurrence frequency, and do not rely on longer delta patterns repeating. However, the benefit of aggressive offset prefetching without pattern matching can be nullified by resource constraints when multiple workloads fight for limited shared LLC and DRAM bandwidth [24]. The performance degradation of **BOP** with *mcf* and *SAT solver* is discussed in the multi-core analysis section.

1) Prefetching Coverage and Accuracy: The substantial performance benefit of **SPP** versus the other prefetchers shown in Figure 9 is a direct result of **SPP's** prefetching accuracy and coverage. Figure 10 shows the prefetching coverage for each benchmark. In this figure, each prefetcher is indicated

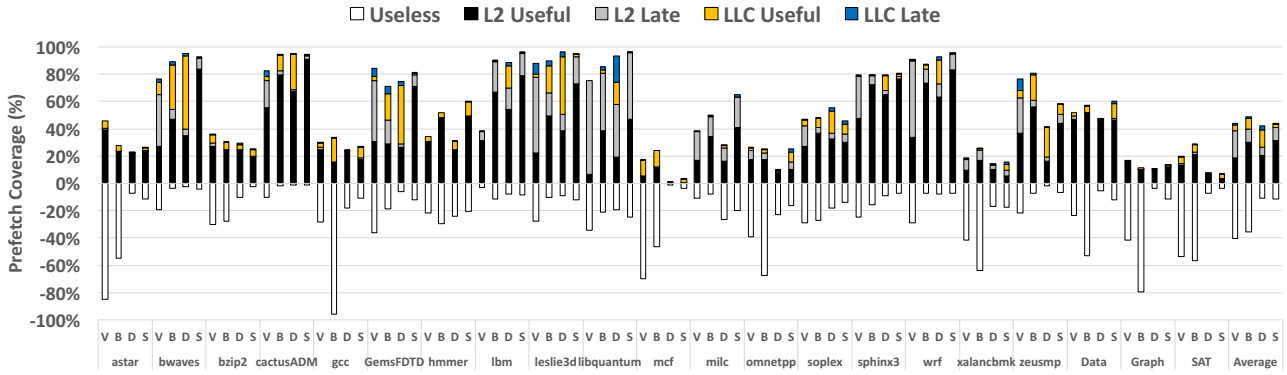


Fig. 10: Prefetching coverage and useless prefetches.

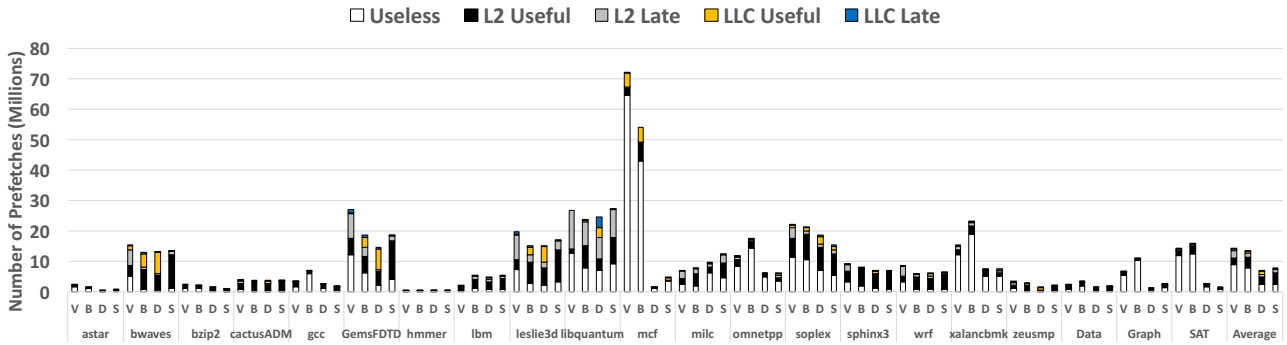


Fig. 11: Raw prefetch request breakdown. **VLDP** and **BOP** both generate significant numbers of useless prefetches.

by its first letter (*e.g.*, "V" for **VLDP**, "D" for **DA-AMPM**, etc.). Prefetching coverage is measured by the number of useful prefetches divided by the number of cache misses without prefetching. Because prefetching can be useful in different ways, we further break down the useful prefetches into four different categories. **Useful** represents the portion of prefetched cache lines that are filled into the cache before their first demand access, eliminating all cache miss penalty. On the other hand, **Late** represents the portion of prefetched cache lines that were issued to DRAM but were not filled before their first demand access. Thus, **Late** prefetches have the effect of accelerating the demand fetch compared to a regular cache miss. **Useful** and **Late** are measured for both the L2 cache and the LLC. We also plot the fraction of useless prefetches on the bottom of the figure, showing the effect of over-aggressive, low-confidence prefetching. On average, **BOP** shows the highest prefetching coverage, however, it also generates a large number of useless prefetches (mostly to the L2 cache), which consume additional DRAM bandwidth and energy. Alhough SPP has slightly lower coverage compared to BOP, it has the largest number of L2 **Useful** prefetches due to its highly accurate dynamic path confidence-based throttling.

Because prefetching coverage alone does not show the aggressiveness of each prefetcher, we also plot the raw number of prefetch requests in Figure 11. As discussed in Section IV-B and shown in Figure 9a, **BOP** and **VLDP** outperform **SPP** in three workloads (*mcf*, *xalancbmk* and *SAT solver*) in a single-core environment. Figure 11 shows that for these workloads,

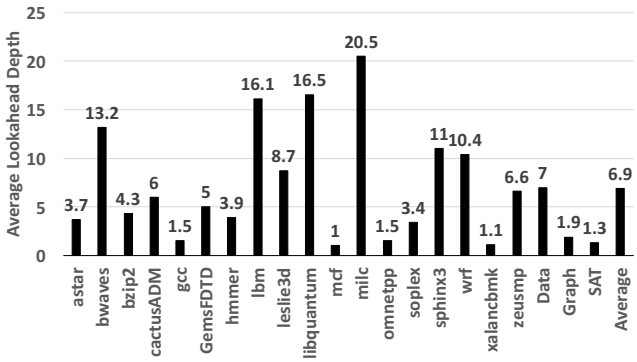


Fig. 12: Average Lookahead Depth.

BOP and **VLDP** generate a huge number of useless prefetches to achieve this performance advantage, aggressively consuming DRAM bandwidth and LLC capacity. In a single core environment, this over-prefetching does not negatively impact performance, however as we will show, performance does degrade when the LLC and DRAM bandwidth are shared by multiple cores.

2) Average Lookahead Depth: Because the length of memory access patterns is different in each application, the optimal lookahead prefetching depth also varies. Figure 12 shows the diversity of lookahead depths SPP uses across the 21 benchmarks we examined. For example, *libquantum* is well known to have very stable and long delta patterns. Figure 12

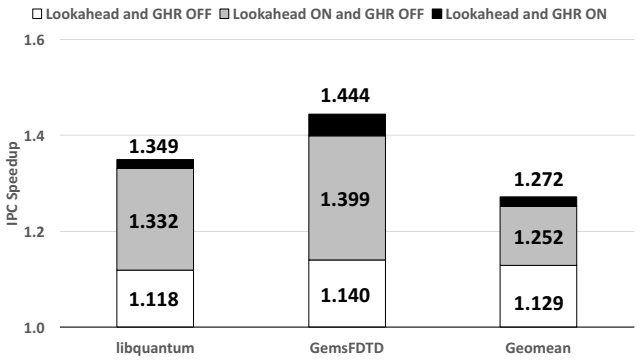
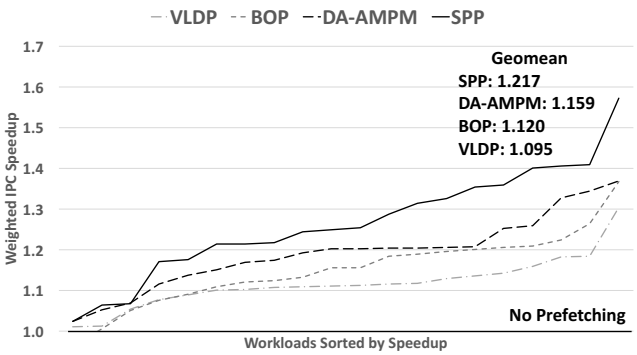


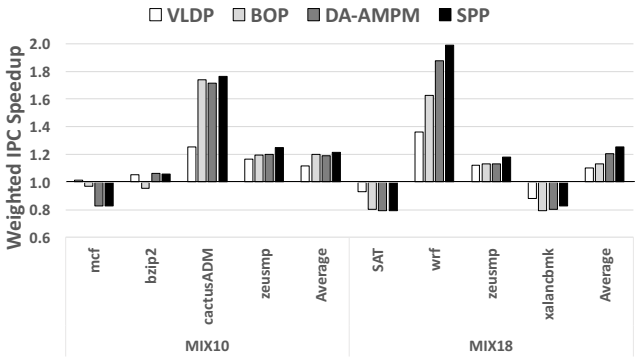
Fig. 13: Contribution of SPP components. Stacked graph represents accumulated speedup from each component.

confirms that SPP uses a very deep lookahead depth for *libquantum*. Meanwhile, *milc* also uses a deep lookahead depth, but this case is very different from *libquantum*. We find that many delta patterns in *milc* do not fit within a single 4KB page. In these cases, SPP will predict that the access pattern will leave the current page, and then, several deltas later, return to the current page, where prefetching can continue. SPP will continue looking further ahead until its confidence falls below the threshold. On the other hand, **SPP** does not prefetch deeply on *xalancbmk*, because this benchmark is not amenable to prefetching, due to a low degree of spatial and temporal locality in each page. In this case, **SPP** detects that prefetching is inaccurate by using the PF, and lowers the global scaling factor α, which serves to limit the lookahead depth. The average lookahead depth of **SPP** across all traces is 6.9.

3) Contribution to Performance Improvement: Figure 13 shows the relative contribution of lookahead and page boundary learning (via the GHR), compared to a basic version of **SPP** without either of these features. To highlight the impact of each feature, we select two benchmarks (*libquantum* and *GemsFDTD*), and break down the performance benefit. A basic **SPP** algorithm without lookahead or the GHR shows a performance improvement of 11.8% on *libquantum* and 14.0% on *GemsFDTD*. Adding lookahead prefetching achieves a significant additional speedup of 21.4% for *libquantum*, because the benchmark has simple memory access patterns that can be accurately covered with very deep lookahead prefetching. On the top of that, page boundary learning provides little benefit for *libquantum*, because each page takes very little time to train, even withouth the GHR. For *GemsFDTD*, lookahead prefetching improves performance by 25.9%. However, page boundary learning provides a substantial boost of 4.5% to *GemsFDTD* because many delta patterns do not fit inside a single 4KB page. In this application, a common delta pattern is (+30) followed by (+1), thus many of the deltas cross page boundaries. The GHR structure captures this behavior in *GemsFDTD* and provides a substantial performance improvement. On average, basic **SPP**, without lookahead provides a 12.9% speedup; adding lookahead prefetching provides 12.3% more speedup, and finally adding the page boundary learning gains another 2.0% improvement.



(a) Workloads sorted by normalized weighted speedup



(b) Performance alaysis for MIX10 and MIX18.

Fig. 14: Normalized speedup for mixes of 4 workloads.

C. Multi-programmed Mix Performance

To show results for a multi-core system, we generate 20 multi-programmed mixes consisting of traces from different benchmarks, and assign each trace to a different core. The mixes are randomly generated in order to fairly represent the characteristics of multi-programmed workloads. Figure 14a shows the performance improvement of four-workload mixes, measured by normalized weighted speedup. The graph is sorted by speedup order. Out of the 20 random mixes, **SPP** achieves the best performance on 19 mixes. For the remaining mix, **DA-AMPM** beats **SPP** by less than 0.5%. On average, **SPP** achieves a 21.7% speedup compared to the baseline.

Note that the **BOP** prefetcher, which nearly ties for second place on single core benchmarks, shows particularly poor performance in multicore systems. As shown in Figure 10, although **BOP** has good coverage, it also fetches a high number of useless lines. In a single core system this does not hurt performance, because the LLC is not over-committed. In a multicore system, however, the LLC pressure is greater, which leads to performance loss. Figure 14b shows two examples of multi-programmed mixes that suffer from **BOP**'s aggressive prefetching. Each mix contains benchmarks in which **BOP** outperforms **SPP** in the single core environment. In MIX10, **VLDP** and **BOP** show the least performance degradation on *mcf*. However, this performance in *mcf* comes from aggressive prefetching, which prevents other workloads from getting better performance. In particular, **BOP** shows performance degradation on *bzip2* while the other prefetchers

Structure	Entry	Component	Storage
Signature Table	256	Valid (1 bit) Tag (16 bit) Last offset (6 bit) Signature (12 bit) LRU (6 bit)	11008 bits
Pattern Table	512	C_{sig} (4 bit) C_{delta} (4*4 bit) Delta (4*7 bit)	24576 bits
Prefetch Filter	1024	Valid (1 bit) Tag (6 bit) Useful (1 bit)	8192 bits
Global History Register	8	Signature (12 bit) Confidence (8 bit) Last offset (6 bit) Delta (7 bit)	264 bits
Accuracy Counter	1 1	C_{total} (10 bit) C_{useful} (10 bit)	10 bits 10 bits
11008 + 24576 + 8192 + 264 + 20 = 44060 bits ≈ 5.37 KB			

TABLE II: SPP storage overhead.

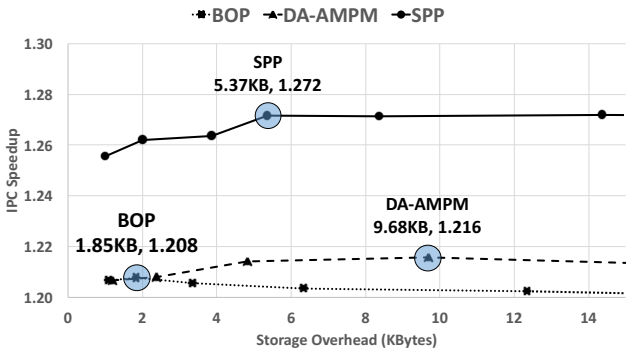


Fig. 15: SPP storage sensitivity.

show performance improvement. In MIX18, all prefetchers gain performance in *wrf* and *zeusmp* at the cost of degrading *SAT solver* and *xalancbmk*. Similarly, **BOP**'s advantage is lower than **SPP**'s due to its aggressive prefetching on *SAT solver* and *xalancbmk*. Note that in single core experiments, **BOP** showed performance improvement in *xalancbmk* and *SAT solver*. However, when resources are shared among multiple cores, **BOP**'s aggressive prefetching hurts overall system performance.

D. Storage Sensitivity

The total storage used by **SPP** in the preceding experiments is 5.37KB, with the storage requirement for each individual component shown in Table II. As the table shows, **SPP**'s largest component is the PT (3KB). This table has multiple delta predictions and counters for each tracked signature. Figure 15 shows a performance sensitivity analysis between **BOP**, **DA-AMPM**, and **SPP** when scaling storage size. Each point represents the storage configuration used for the performance evaluation. As expected, the overall performance of **SPP** and **DA-AMPM** does not increase with greater storage capacity. Generally, **SPP** always outperforms the other prefetchers at a given storage budget. Note that, counterintuitively, **BOP** does not benefit from greater storage, because it takes a longer time

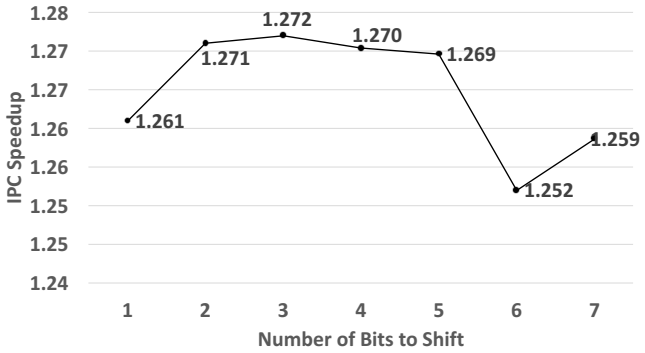


Fig. 16: Performance sensitivity with respect to the number of bits to shift.

to find the best offset value when more cache lines are in its recent request table [10].

E. Signature Sensitivity

SPP uses a 12-bit history signature built by a series of 3-bit shifts and XORs. This shifting and XORing represents a form of lossy information compression. The more bits that are shifted, the less compression, and vice-versa. Here we examine the performance impact of this compression. Figure 16 shows the performance sensitivity to the number of shifted bits. Since each delta pattern within a 4KB page can be represented with 7 bits (-63 $\sim$ 63), there is no value in shifting more than 7 bits. The figure shows that a 12-bit signature prefers 3-bit shifting. The goal of the history signature is to track as many useful deltas as possible within a small 12-bit signature. This is best achieved by giving small deltas, which are the most common, all the precision they need, while still keeping some information from larger deltas. By shifting 3-bits prior to XORing, SPP compresses four deltas into 12-bits. Deltas from 0-7 are represented at full precision. Larger deltas (>7) cause some aliasing, but still contribute information to the final signature.

V. Conclusion

Signature Path Prefetching offers compelling solutions for three major prefetching challenges. First, it is able to learn and prefetch complex data access patterns by using a compressed history signature. Second, it is able to detect when a data access pattern crosses a page boundary, and quickly resume prefetching on the new page. Third, it is able to balance aggressive prefetching with accuracy by using path confidence. SPP is able to do all this without using the program counter or other core registers, and while operating strictly in the physical address space. SPP improves performance versus a no-prefetching baseline by 27.2%, and improves performance by 6.4% versus the next best competing technique.

Acknowledgment

We thank the National Science Foundation, which partially supported this work through grants CCF-1320074 and I/UCRC-1439722, and Intel Corp. for their generous support.

REFERENCES

[1] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *SIGARCH Comp. Arch. News*, vol. 23, pp. 20–24, March 1995.

[2] A. J. Smith, "Sequential program prefetching in memory hierarchies," *Computer*, vol. 11, pp. 7–21, December 1978.

[3] T. Chen and J. Baer, "Effective hardware-based data prefetching for high-performance processors," *IEEE Transactions on Computers*, vol. 44, pp. 609–623, 1995.

[4] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial memory streaming," in *ACM SIGARCH Computer Architecture News*, vol. 34, pp. 252–263, IEEE Computer Society, 2006.

[5] I. Hur and C. Lin, "Memory prefetching using adaptive stream detection," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 397–408, IEEE Computer Society, 2006.

[6] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi, "Spatio-temporal memory streaming," in *ISCA*, pp. 69–80, 2009.

[7] A. Jain and C. Lin, "Linearizing irregular memory accesses for improved correlated prefetching.," in *MICRO*, pp. 247–259, 2013.

[8] Y. Ishii, M. Inaba, and K. Hiraki, "Access map pattern matching for high performance data cache prefetch," *Journal of Instruction-Level Parallelism*, vol. 13, pp. 1–24, 2011.

[9] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, "Efficiently prefetching complex address patterns," in *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture*, 2015.

[10] P. Michaud, "A best-offset prefetcher," in *High Performance Computer Architecture (HPCA), 2016 IEEE 20th International Symposium on*, IEEE, 2016.

[11] J. W. Fu, J. H. Patel, and B. L. Janssens, "Stride directed prefetching in scalar processors," *ACM SIGMICRO Newsletter*, vol. 23, no. 1-2, pp. 102–110, 1992.

[12] D. Kadjo, J. Kim, P. Sharma, R. Panda, P. Gratz, and D. Jimenez, "B-fetch: Branch prediction directed prefetching for chip-multiprocessors," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 623–634, IEEE Computer Society, 2014.

[13] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: An alternative to very large instruction windows for out-of-order processors," in *Proceedings of the 9th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 129–140, 2003.

[14] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith, "Ac/dc: An adaptive data cache prefetcher," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pp. 135–145, IEEE Computer Society, 2004.

[15] 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 *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pp. 626–637, IEEE, 2014.

[16] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pp. 63–74, IEEE, 2007.

[17] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer," in *Software, IEE Proceedings-*, pp. 96–96, IEEE, 2004.

[18] S. H. Pugsley, A. R. Alameldeen, C. Wilkerson, and H. Kim, "The 2nd Data Prefetching Championship (DPC-2)."

[19] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using simpoint for accurate and efficient simulation," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, pp. 318–319, ACM, 2003.

[20] "Standard Performance Evaluation Corporation CPU2006 Benchmark Suite.."

[21] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie, "Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs," in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pp. 2–11, ACM, 2010.

[22] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," in *ACM SIGPLAN Notices*, vol. 47, pp. 37–48, ACM, 2012.

[23] Y. Ishii, M. Inaba, and K. Hiraki, "Unified memory optimizing architecture: memory subsystem control with a unified predictor," in *Proceedings of the 26th ACM International Conference on Supercomputing*, pp. 267–278, ACM, 2012.

[24] N. D. Enright Jerger, E. L. Hill, and M. H. Lipasti, "Friendly fire: understanding the effects of multiprocessor prefetches," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 177–188, 2006.