The Dissertation Committee for Akanksha Jain
certifies that this is the approved version of the following dissertation:

Exploiting Long-Term Behavior for Improved Memory System Performance

Committee:

Calvin Lin, Supervisor

Doug Burger

Donald S. Fussell

Yale N. Patt

Keshav Pingali

Exploiting Long-Term Behavior for Improved Memory System Performance

by

Akanksha Jain, B.Tech.; M.Tech.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

May 2016

Acknowledgments

My PhD experience has been a valuable intellectual and personal journey for me, and for all that I have learned in the past seven years, I have many people to thank.

First and foremost, I am grateful to my advisor, Calvin Lin, who has ensured that my graduate education helped me grow intellectually. He has helped me structure my thought process and see bigger questions behind our research efforts. His optimism and enthusiasm for research is infectious, and it has always encouraged me to try harder. I especially appreciate his thorough feedback on my writing, which has helped me think and write more systematically. Finally, in all our interactions, he has maintained an environment where I felt free to express myself creatively, which has shaped me as a researcher.

I would also like to thank my committe members for their valuable inputs over the course of my PhD. Dr. Fussell has provided kind words of encouragment and important feedback on numerous occassions. Dr. Patt got me excited about computer architecture through his graduate architecture course. My interactions with Dr. Keshav Pingali pointed me to important challenges in optimizing irregular programs, and my interactions with Dr. Doug Burger encouraged me to think about some important issues in cache replacement.

I am thankful to my group members Curtis, Jeff, Ashay, Hao, Jia, Oswaldo and Dong for their support, feedback, and great company.

Last but not the least, I am fortunate to have a great support system in my

family and friends. My parents and my brother have supported me unconditionally throughout my graduate education, and they have cheered for my every endeavor, even when they had no idea what I was doing. My husband, Vikram has not only provided encouragement and support, but he has also acted as a senior graduate student providing helpful guidance and always being available to bounce ideas off. All my friends in Austin (Avni, Pooja, Suyog, Esha, Sucheta, Ramya, Vinay, Rachit), and those who took time to visit me in Austin (Ankita, Dharik, Aru, Purvi, Radha), I couldn't have made it without you.

Akanksha Jain

The University of Texas at Austin
May 2016

Exploiting Long-Term Behavior for Improved Memory System Performance

Publication No. _____

Akanksha Jain, Ph.D.
The University of Texas at Austin, 2016

Supervisor: Calvin Lin

Memory system performance is a key bottleneck for many programs. Caching and prefetching are two popular hardware mechanisms to alleviate the impact of long memory latencies, but despite decades of research, significant headroom remains. In this thesis, we show how we can significantly improve caching and prefetching by exploiting a long history of the program's behavior. Towards this end, we define new learning goals that fully exploit long-term information, and we propose history representations that make it feasible to track and manipulate long histories.

Based on these insights, we advance the state-of-the-art for three important memory system optimizations. For cache replacement, where existing solutions have relied on simplistic heuristics, our solution pursues the new goal of emulating the

optimal solution for past references to predict caching decisions for future references. For irregular prefetching, where previous solutions are limited in scope due to their inefficient management of long histories, our goal is to realize the previously unattainable combination of two popular learning techniques, namely address correlation and PC-localization. Finally, for regular prefetching, where recent solutions learn increasingly complex patterns, we leverage long histories to simplify the learning goal and to produce more timely and accurate prefetches.

Our results are significant. For cache replacement, our solution reduces misses for memory-intensive SPEC 2006 benchmarks by 17.4% compared to 11.4% for the previous best. For irregular prefetching, our prefetcher obtains 23.1% speedup (vs. 14.1% for the previous best) with 93.7% accuracy, and it comes close to the performance of an idealized prefetcher with no resource constraints. Finally, for regular prefetching, our prefetcher improves performance by 102.3% over a baseline with no prefetching compared to the 90% speedup for the previous state-of-the-art prefetcher; our solution also incurs 10% less traffic than the previous best regular prefetcher.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

For many programs, long memory latencies are a critical bottleneck for overall system performance. To alleviate this bottleneck, computer architects employ optimizations, such as, caching and prefetching. Caches reduce the average memory latency by storing a few data items close to the processor, and prefetchers hide long memory latencies by predicting and fetching memory locations ahead of time.

Because of their significant potential, there has been a long line of work that improves both caching and prefetching [90, 53, 81, 95, 41, 68, 42, 49, 96, 50, 55, 73, 29, 78, 28, 16, 91, 93, 17, 59, 86, 85, 52, 4, 26, 76, 87, 35, 79], but despite decades of research, significant headroom remains. For example, the left side of Figure 1.1 shows that there is a considerable gap between state-of-the-art replacement policies and an unrealistic optimal cache replacement solution [5], and the right side of Figure 1.1 shows that existing irregular prefetchers achieve a small proportion of the performance of an unconstrained irregular prefetching algorithm.

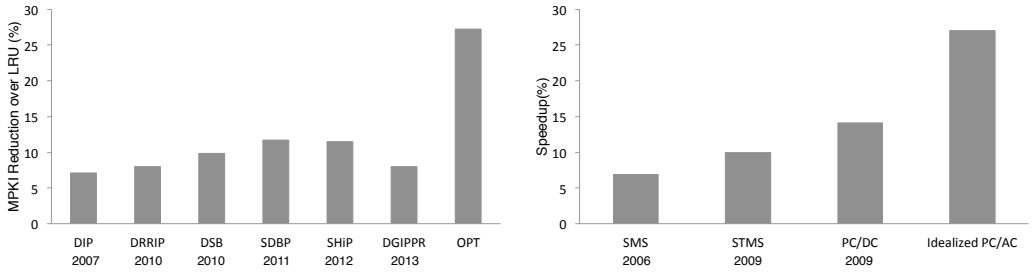

Figure 1.1: Significant headroom exists for cache replacement and irregular prefetching.[1]

1.1 Goals

In this thesis, our goal is to significantly advance the state-of-the-art in prefetching—both regular and irregular—and caching. To achieve this goal, we observe that most memory system optimizations can be viewed as learning problems because they learn from past behavior to predict future behavior. As learning problems, these optimizations typically have a well-defined *learning goal*, which describes the behavior that they intend to learn. For example, a regular prefetcher's learning goal is to find sequences of accesses that are a constant stride apart. To advance the state-of-the-art in caching and prefetching, this thesis explores the following hypothesis:

> **Hypothesis**: Memory system optimizations can be improved by utilizing long-term history information to enable learning goals that are infeasible with short-term history information.

To understand the importance of long histories, consider the problem of cache replacement. For cache replacement, Belady's algorithm [5] is optimal, but it is impractical as it assumes knowledge of the future. Therefore, existing solutions use

[1]Due to techical reasons, we cannot accurately quantify the headroom for regular prefetchers.

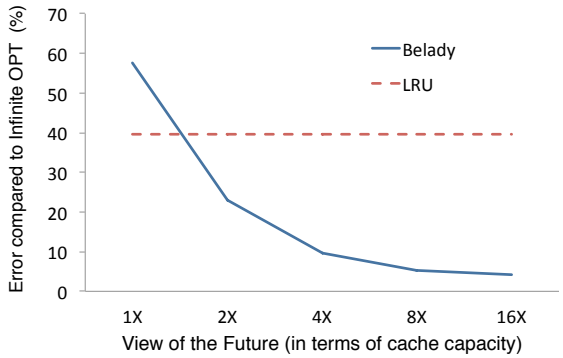

Figure 1.2: Belady's algorithm requires a long view of the future.

heuristics, such as LRU and MRU, but these heuristics are limited because they are tailored to work for specific access patterns and do not perform well in more complex scenarios. Instead of using heuristics, we observe that while it is impossible to look into the future, we can apply Belady's optimal algorithm to past references, and if past behavior indicates future behavior, we can learn the optimal solution for the past to emulate Belady's solution for the future. One concern with this idea is that Belady's algorithm looks infinitely far into the future, and therefore our solution would require an infinitely long history. In fact, Figure 1.2 shows that Belady's algorithm benefits from long windows and that it approaches the performance of the true optimal solution (OPT) when its window is 8× the size of the cache. Thus, to effectively learn the OPT solution for past memory references, we would need to track a long—but not unbounded—history.

As another example, consider the problem of irregular prefetching. Irregular memory accesses—such as those caused by pointer-chasing and complex traversals of array-like structures—are common, but they are hard to prefetch because they cannot be predicted by learning a few strides. The notion of *address correlation* is a promising technique that aims to find repeating pairs of correlated memory accesses [13, 15]. To learn address correlation (AC), irregular prefetchers need to

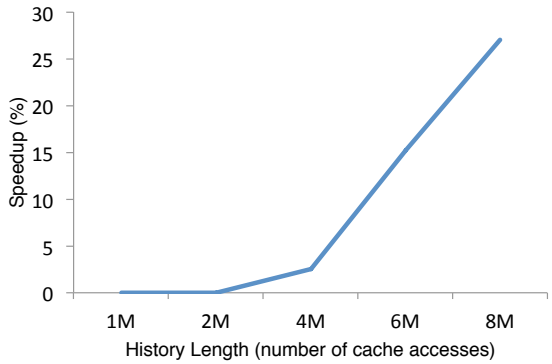

Figure 1.3: Irregular Prefetching requires a long history.

remember a long history of memory references because correlated pairs repeat across many last-level cache accesses [91, 16]. Figure 1.3 shows how the performance of an idealized address correlation-based prefetcher declines with shorter histories. To use long histories, existing irregular prefetchers use large off-chip meta-data structures [92, 93], but they perform poorly because sequences of correlated memory references (also known as *temporal streams*) become interleaved and appear unpredictable at the microarchitectural level. We observe that temporal streams can be predicted accurately if we combine address correlation with the notion of PC-localization [59]. Unfortunately, existing history representations make this combination infeasible because PC-localization further increases the large penalty of accessing the off-chip meta-data. To effectively utilize long-term repetition for irregular prefetching, we explore history organizations that offer fundamentally better trade-offs. In particular, our history organization has two key benefits over previous solutions: (1) it enables an irregular prefetcher that is the first to combine address correlation with PC-localization; (2) it enables a meta-data caching scheme which dramatically reduces the cost of managing the large off-chip meta-data.

These examples illustrate three points. First, long histories can help improve memory system optimizations. Second, to fully exploit long histories, we need new

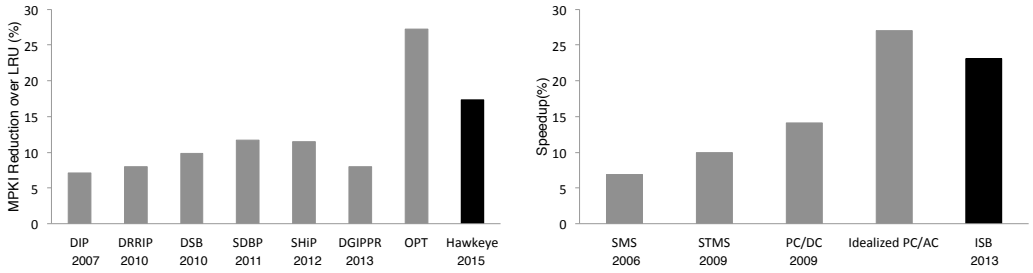

Figure 1.4: Our solutions offer significant improvements.

learning goals, such as the idea of learning OPT's behavior for cache replacement and remembering PC-localized temporal streams for irregular prefetching. Finally, we need efficient mechanisms to track and manipulate long histories so that we can meet our desired learning goals.

With these insights in mind, we now describe our solutions for cache replacement, irregular prefetching, and regular prefetching. For cache replacement and irregular prefetching, we explain how long histories can be leveraged to realize ambitious learning goals, and for regular prefetching, we show that long-term information, in fact, simplifies the learning goal, which allows us to produce a more robust and timely prefetcher. To manage long histories, we propose new history representations that are both informative and efficient. The result is that our solutions are indeed able to advance the state-of-the-art for each of these problems as shown in Figure 1.4.

1.2 Our Solution

Cache Replacement. For cache replacement, our goal is to learn Belady's optimal solution for past memory references and use that information to predict the caching behavior of future accesses. Our use of OPT introduces two technical chal-

lenges. First, we need an efficient mechanism of reconstructing OPT. Second, a long history is needed to compute OPT. We solve the first problem by using the notion of liveness intervals (see Section 3.1), which leads to a simple and efficient solution. The use of liveness intervals is novel for cache replacement, because it explicitly conveys information about both reuse distance and the demand on the cache, which are both essential for making proper eviction decisions. We solve the second problem by using Set Dueling [68] to sample a small subset of all cache lines.

The result is that with 16 KB of additional storage (plus tag storage), our replacement policy, which we call *Hawkeye* [37] can compute OPT's solution for past accesses with 95% accuracy. Of course, past behavior does not always model future behavior, so Hawkeye's performance does not match OPT's. Nevertheless, as shown in Figure 1.4, Hawkeye performs significantly better than previous policies on a memory-intensive SPEC CPU 2006 benchmarks.

Irregular Prefetching. For irregular prefetching, our goal is to efficiently combine address correlation with PC-localization. A related goal is to reduce the cost of managing and retrieving prefetcher meta-data, which can be as large as a few MBs. To realize both goals, we create a new level of indirection by mapping correlated physical addresses to consecutive addresses in a new address space, which we call the *structural address space*. As a result of this indirection, irregular prefetching in the physical address space reduces to regular prefetching in the structural address space. Thus, our prefetcher is called the *Irregular Stream Buffer*, or the ISB [36].

The structural address space is unique because unlike previous solutions, it is both spatially and temporally organized, which provides several benefits. First, it enables the combination of PC-localization and address correlation, which is prohibitively expensive for purely temporal organizations [92, 59] (see Section 4.1 for more details). Second, it can predict long temporal streams using a single lookup, which is highly inefficient with purely spatial organizations [44]. Finally, ISB's

mappings from physical to structural address space—which can be as large as the memory footprint of the application—can be managed in the same way that virtual to physical mappings are managed in hardware. In particular, we use a small on-chip cache to store physical to structural mappings for TLB-resident pages, and we synchronize this cache with the TLB. Thus, the movement of meta-data between on-chip and off-chip memory is hidden by long latency TLB misses. With this caching scheme, ISB is the first address correlation-based prefetcher that relies only on on-chip meta-data for its training and prediction operations.

The result is that with 32KB of on-chip storage, the ISB approaches the performance of an idealized combination of address correlation and PC-localization (see Figure 1.4), and it achieves 93.7% accuracy while incurring only only 8.4% traffic overhead for its meta-data accesses. By contrast, the previous best irregular prefetcher achieves 64.2% accuracy with approximately 35% traffic overhead.

Regular Prefetching. Regular prefetchers are simple, effective, and commercially viable because they detect simple patterns of constant strides using a relatively small history of memory accesses. To improve performance, recent regular prefetchers [79, 59, 21] learn complex stride patterns such as, 2, 3, 4, 2, 3, 4. We observe that instead of learning the precise sequence of strides, stride patterns can be inferred by finding a single aggregate stride–9 in this case–that works for every reference in the stream.

By pursuing a simpler learning goal, our Aggregate Stride Prefetcher (ASP) enjoys several advantages over previous regular prefetchers[2]: (1) ASP's pattern matching algorithm is robust to reordering in the memory system; (2) ASP can use an accurate scheme to find the most timely stride irrespective of how individual streams are interleaved at the microarchitectural level; (3) ASP trains quickly be-

[2]The recently published Best Offset Prefetcher [57], which was developed concurrently with our work, is quite similar to ASP (see Chapter 2 for details).

cause it can use the stride patterns learned in one region to predict strides in other unseen regions.

While the notion of aggregate strides is conceptually simple, it can be challenging to find aggregate strides because the exact length of stride patterns are highly variable and because aggregating strides over incorrect lengths can result in both poor coverage and accuracy. For example, in the previous example, aggregating strides over a length of 2 results in an aggregate stride of 5, which is correct for only 1 in 3 accesses. To find aggregate strides over any pattern length, we find that long histories are helpful, as aggregate strides tend to repeat more often when we observe a long history of pairwise strides. Thus, for regular prefetching, long histories enable a simpler learning goal, which offers attractive design trade-offs.

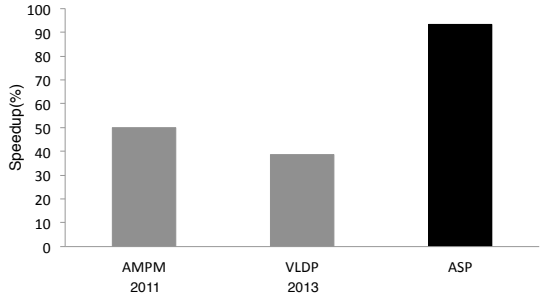

Figure 1.5: ASP outperforms previous solutions (degree 1 prefetching).

As shown in Figure 1.5, ASP significantly outperforms state-of-the-art regular prefetchers with a prefetch degree of 1. With a prefetch degree of 4, ASP achieves a speedup of 102.3%, while the previous best solution achieves a speedup of 90%. At higher degrees and in multi-core environments, ASP's traffic overhead is significantly less than other prefetchers.

1.3 Contributions

The main contribution of this thesis is the confirmation of the hypothesis presented in Section 1.1 for three different memory system optimizations, namely, cache replacement, irregular prefetching and regular prefetching. Moreoever, we show that exploiting long-term behavior enables qualitatively better solutions.

In particular, this thesis makes the following conceptual contributions:

1. For cache replacement, our goal is to learn from the optimal solution rather than relying on heuristics. To compute the optimal solution, we introduce a new online algorithm that emulates Belady's algorithm for past memory references.

2. For irregular prefetching, our goal is to combine address correlation with PC-localization. To achieve this goal, we introduce the structural address space that logically arranges irregular streams in the order of their traversal rather than the order of their allocation.

3. For regular prefetching, our goal is to find aggregate strides instead of learning complex delta patterns, and we show that aggregate strides, in fact, subsume delta correlation while being more robust and timely.

This thesis makes the following concrete contributions:

1. We introduce the Hawkeye cache replacement policy that improves cache replacement by applying Belady's optimal solution to a long history of past accesses to train a predictor that is consulted for future caching decisions. For the SPEC CPU 2006 benchmark suite, Hawkeye obtains a 17.0% miss rate reduction over LRU, which results in a 8.4% improvement in performance. On a 4-core system, Hawkeye's speedup over LRU increases to 15.0%.

2. We introduce the ISB, the first data prefetcher that combines the use of PC localization and address correlation to prefetch irregular memory access patterns effectively and efficiently. In particular, the ISB obtains 23.1% speedup and 93.7% accuracy, while incurring an average of 8.4% memory traffic overhead due to meta-data accesses.

3. We introduce the Aggregate Stride Prefetcher(ASP), a regular data prefetcher that leverages a long history of memoy references to learn complex delta patterns using the simpler notion of aggregate strides. On single core systems, ASP with a prefetch degree of 1 improves performance by 93% over a baseline with no prefetching, and with a prefetch degree of 4, ASP's performance benefit increases to 102.3%. On a 4-core system, ASP improves performance by 64%.

Chapter 2

Related Work

This chapter places our work in the context of prior research in cache replacement, irregular prefetching, and regular prefetching. For cache replacement, we categorize existing work based on the nature of information that various policies use, such as, short-term and long-term information. For irregular prefetching, we focus on related work that uses long-term behavior, and we briefly discuss prefetchers that use short-term behavior to find short regular streams in irregular programs. For regular prefetching, we categorize existing work based on their history representations, and in Chapter 5, we explain how ASP's history representation allows it to leverage long-term behavior.

2.1 Cache Replacement

Since Belady's optimal cache replacement algorithm [5] was introduced in 1966, there has been considerable work in the development of practical replacement algorithms [11, 98, 72, 27, 90, 53, 81, 95, 41, 68, 42, 49, 96, 50, 55, 73, 29, 78, 28]. Here, we focus on work that is most closely related to Hawkeye, organized based on the type of information that they use to make decisions.

2.1.1 Short-Term History Information

Many solutions use short-term information that reflects the current state of the cache, ignoring any information about cache lines that have been evicted.

Such solutions typically uses heuristics that cater to specific types of cache access patterns. For example, *recency-friendly* policies such as LRU and its variations [42, 95, 81] prioritize recently used lines under the assumption that they will soon be used again. Other policies favor lines with high access-frequency under the assumption that frequently used lines will soon be used again [73, 29, 71, 60, 51]. Jaleel et al. enhance recency-friendly policies by cleverly using 2-bits of re-reference interval prediction (RRIP) [41] to eliminate cache pollution due to streaming accesses. Hawkeye uses RRIP's idea of aging to adapt to changes in phase behavior.

To avoid the pathological behavior of recency-friendly policies on workloads that exhibit large reuse distances, *thrash-resistant* policies [68, 78] discard the most recently used line instead of the least recently used line, thereby retaining a portion of the active working set. Unfortunately, thrash-resistant policies perform poorly in the presence of recency-friendly or streaming accesses.

Because different replacement policies favor different cache access patterns, hybrid solutions have been developed [88, 69, 68] to dynamically select among competing policies. The key challenges with hybrid replacement are the management of additional information and the high hardware cost for dynamic selection. Qureshi et al. introduce Dynamic Set Sampling (DSS) [68], an inexpensive mechanism that chooses the best policy by sampling a few dedicated sets to assess the efficacy of the desired policy. Thus, DSS allows the policy to change over time, but it selects a single policy for all cache lines. By contrast, Hawkeye can use different policies for each load instruction.

2.1.2 Long-Term History Information

Recent work exploits long-term information, including information about lines that have been evicted from the cache. For example, some policies [48, 23] predict reuse distances for incoming lines based on past reuse distributions, but such policies are expensive. Moreover, unlike Hawkeye's liveness intervals, reuse distance alone leads to inaccurate decisions because it does not account for the demand on the cache. For example, a line with a long reuse interval can remain in the cache if there is low demand on the cache, while at some other point in time, a line with a short reuse distance can be evicted from the cache if there is high demand for the cache. (See Section 3.1.1).

Hawkeye builds on recent work that learns the caching behavior of past load instructions to guide future caching decisions: SHiP [96] uses a predictor to identify instructions that load streaming accesses, while SDBP [49] uses a predictor to identify lines that are likely to be evicted by the LRU policy. Thus, SHiP and SDBP improve cache efficiency by not dedicating cache resources to lines that are likely to be evicted. However, these policies can be inaccurate because they learn the behavior of heuristic-based replacement policies (LRU and RRIP), which perform well for a limited class of access patterns. By contrast, Hawkeye simulates and learns from the past behavior of OPT, which makes no assumptions about access patterns.

Hawkeye considers a longer history of operations than either SHiP or SDBP, maintaining a history that is 8 times the size of the cache. To simulate OPT's behavior, we introduce an algorithm which bears resemblance to the Linear Scan Register Allocator [65] but solves a different problem.

Another class of predictors called dead-block predictors observe past behavior to identify cache lines that are unlikely to be reused before they are evicted from the cache. Dead block prediction has been used to drive replacement decisions, where dead blocks are identified by remembering sequences of instructions that re-

sult in last touch to a cache line [54], by learning upper bounds on the time gap between cache reuses [31, 2], or by learning the number of accesses to a block before it becomes dead [50]. While Hawkeye can be viewed as a dead block predictor, the key difference between Hawkeye and existing dead-block predictors is that Hawkeye identifies blocks that are likely to be dead with OPT, while other dead-block predictors learn the behavior of heuristics such as LRU. Liu et al. propose the use of cache bursts [55] rather than cache accesses to drive dead block prediction for L1 caches, but we observe that cache bursts offer limited benefit for last-level caches.

2.1.3 Future Information

Another class of replacement policies takes inspiration from victim caches [45] and defers replacement decisions to the future when more information is available. For example, the Shepherd Cache [72] emulates OPT by deferring replacement decisions until future reuse can be observed, but it cannot emulate OPT accurately because it uses an extremely limited window into the future; larger windows would be expensive because unlike Hawkeye's history of past references, the Shepherd Cache must store the contents of the lines that make up its window into the future. Other solutions [67, 78] use slightly longer windows ($2\times$) into the future, but these solutions do not model OPT. In general, these solutions make a tradeoff between the window size and the precision of their replacement decisions.

2.1.4 Other Types of Information

Cache performance can be improved by not only reducing the number of misses but by selectively eliminating expensive misses. For example, MLP [69] and prefetch-friendliness [97] can be used to reduce the overall performance penalty of LLC misses. The Hawkeye policy focuses on cache misses and is complementary to these techniques.

2.2 Irregular Prefetching

We now review prior research in optimizing irregular memory accesses, first discussing techniques for improving spatial locality, and then discussing the considerable prior work in prefetching.

2.2.1 Improving Spatial Locality

Spatial locality can be improved by re-ordering the layout of pointer-based data structures during memory allocation [14] or garbage collection [33], but both techniques involve expensive memory copying, and the former relies on programmer hints. Carter, et al., re-order memory accesses in a shadow address space [10] to improve locality and initiate prefetching, but their technique is limited to statically allocated data structures and requires both OS and programmer intervention.

2.2.2 Stride Prefetching

Stride prefetchers can detect short-term patterns in irregular programs. In general, they target regular memory accesses, building on Jouppi's next-line prefetcher [82, 45] by adding non-unit strides [61] and by predicting strides [4, 26]. Ishii, et al., introduce a clever data structure that compactly captures information about multiple stride lengths [35]. Sair, et al., support irregular streams by introducing a stride length predictor [76].

Hur and Lin enhance stream prefetchers by adding a small histogram of the stream lengths of recently seen memory accesses [34]. These histograms allow stream buffers to accurately prefetch tiny "streams" that might be as short as two cache lines, thereby providing some coverage for irregular memory accesses that stream buffers alone cannot prefetch.

2.2.3 Prefetching Based on Spatial Locality

Irregular memory accesses can also be prefetched by detecting spatial locality [43, 52, 8, 12]. Variations of the *Spatial Locality Detection Table* [43] track accesses to different regions of memory so that spatially correlated data can be prefetched together. These approaches typically need large tables to detect locality, but Somogyi, et al. [86] show how smaller tables can be used by correlating spatial locality with the program counter in addition to parts of the data address. As a result, *Spatial Memory Streaming (SMS)* can use tables as small as 64 KB, while achieving good performance improvements for commercial workloads.

2.2.4 Pointer-based Prefetching

Pointer-based data structures are an important source of irregular memory accesses, so many techniques focus on prefetching pointers.

Compilers can insert prefetch instructions—known as *jump pointers*—for all children of a visited node of a linked data structure [56, 74]. The key issue with compiler-based solutions is poor timeliness; to hide long memory latencies, the software prefetches need to be inserted far from their use.

Hardware solutions, such as pointer caches [18] and hardware jump pointer tables [75], can issue timely prefetches but incur storage overheads of up to 1 MB; some also require compiler support and modifications to the ISA. Content Directed Prefetching (CDP) [19] is a stateless mechanism that searches through cache lines for pointer addresses that are then greedily prefetched. CDP is attractive in terms of storage requirements, but it wastes memory bandwidth because of its low accuracy.

Cooperative hardware-software approaches can combine the accuracy of software prefetching and the timeliness of hardware prefetching [75]. Guided Region Prefetching [89] uses static analysis to annotate load instructions with hints to the hardware prefetcher. Ebrahimi, et al. use compiler-guided filtering mechanisms to

inform a CDP prefetcher about the pointers that are most likely to be fetched [25].

There are two key differences between the ISB and pointer-based approaches: (1) The ISB does not give special treatment to pointers, so it can exploit other sources of irregular memory accesses; (2) pointer-based approaches can prefetch compulsory misses, while the ISB cannot.

2.2.5 Prefetching Based on Temporal Locality

Joseph and Grunwald introduce the notion of correlation-based prefetching with their Markov Prefetcher [44], which uses a table to record possible successors of a given memory address. The presence of address correlation in applications has been studied both quantitatively [13] and qualitatively [91] for scientific and commercial workloads. Studies find that the length of correlated streams can vary from two to several hundred [13, 94], which implies that large amounts of storage are needed to prefetch these workloads effectively. While some designs reduce this on-chip table requirement [32], the table size still grows in proportion to the application's active memory footprint. Thus, a variety of solutions store the Markov table off-chip and optimize the memory bandwidth requirements and prefetch look-ahead distance for off-chip table access [16, 84].

Nesbit and Smith introduce the GHB as a general structure for prefetching streams of temporally correlated memory requests [59]. However, when used to record address correlation [92], the GHB is quite large, requiring about 4 MB of off chip storage for scientific workloads and about 48 MB for commercial server workloads. Thus, Wenisch, et al.'s STMS prefetcher introduces latency and memory traffic optimizations for reading and updating the off-chip history buffer and index table [93]. These techniques reduce the memory traffic from 3× [16, 84, 94] to 1.05-1.75× [93] for long streams. Rather than use address correlation, other GHB-based prefetchers use delta correlation [59, 58], whose space requirements are dramati-

cally smaller, but we show that for irregular accesses, delta correlation leads to low coverage and accuracy.

PC localization has been used to improve the accuracy and coverage of correlation-based prefetchers [61, 52, 59, 86, 85], but until now, the combination of PC localization and address correlation has been too expensive to be practically considered.

Finally, Diaz et al. propose a method of chaining PC-localized streams for better prefetch timeliness [20]. The ISB is orthogonal to these ideas, so it is possible to use stream chaining to link various PC-localized streams in an ISB design, but we do not explore this option in this paper.

2.2.6 Spatial-Temporal Prefetching

The best known irregular prefetcher, Somogyi, et al.'s STeMS prefetcher [85], exploits temporal correlation at a coarse granularity and spatial correlation at a finer granularity, essentially learning temporal sequences of spatial regions. The ISB could be employed in a similar fashion to identify the coarse-grain temporal stream, but we do not explore this idea in this paper.

2.3 Regular Prefetching

Next-line prefetchers [82] fetch the next line on every cache access, and Jouppi's stream buffers [45] confirm sequential streams before prefetching. Stride prefetchers build on stream buffers by adding non-unit strides [61] and by predicting strides [4, 26, 76]. To improve accuracy, Hur and Lin use histograms to predict stream lengths [34], and to improve accuracy, timeliness and robustness, Srinath, et al use a feedback-based scheme [87] to modulate aggressiveness.

Although stream buffers are simple and efficient, they are limited to sequential and strided access patterns and perform poorly in complex scenarios which arise

due to recurring delta patterns, interleaving streams or memory reordering due to out-of-order execution.

Improvements to stream buffers can be broadly classified in the three categories based on their history representations and pattern matching algorithms: Order-based Prefetchers compute strides between consecutive accesses to a region and look for recurring stride sequences, Spatial Prefetchers construct a spatial bitmap and look for recurring spatial patterns, and Offset Prefetchers maintain a recent history of prefetches and look for offsets that would achieve maximum coverage. By contrast, the ASP presents a new history representation that maintains strides between many pairs of accesses over a relatively long history.

2.3.1 Order-Based Prefetchers

The use of delta correlation can find repeating stride patterns for TLB prefetching [46], and Nesbit, et al's global history buffer (GHB) based prefetcher, PC/DC [59], can learn delta correlation for data accesses. The key limitations of PC/DC are (1) its complex GHB traversal algorithm, (2) a limited history of one delta for each prediction, and (3) its inability to find stride patterns across multiple load instructions.

The Variable Length Delta Prefetcher(VLDP) [79] addresses the shortcomings of PC/DC by employing three prediction tables, each of which tracks longer delta histories (the largest table tracks a delta history of 3). For prefetching, the table with the longest delta match makes a prediction for the next most likely stride.

In general, Order-based delta prefetchers such as PC/DC and VLDP are beneficial because they can learn complex patterns, and they can generalize patterns across spatial regions. Unfortunately, their utility is limited to short delta patterns because their overhead increases with larger delta histories (VLDP would require a new prediction table for every additional match in a longer delta history). Moreover,

any perturbation in the memory access ordering can severely impact these prefetchers because they rely on a strict ordering of memory accesses to infer the sequence of deltas.

2.3.2 Spatial Prefetchers

To avoid performance loss due to out-of-order execution and to learn complex access patterns, spatial locality can be exploited [43, 52, 8, 12] to find recurring footprints within spatial regions. These approaches typically need large tables to detect locality, but Somogyi, et al. [86] show how smaller tables can be used by correlating spatial locality with the program counter in addition to the data address.

Ishii, et al's AMPM [35] uses a spatial data structure called access map to compactly capture information about multiple stride lengths [35].

In general, Spatial Prefetchers suffer from two limitations: (1) They ignore the relative order of memory accesses within a region, which allow temporally distant accesses to be viewed as related if they are co-located spatially; (2) they can learn a limited class of complex delta patterns (see Section 5.4).

2.3.3 Offset-Based Prefetchers

Instead of learning strides or spatial patterns, offset-based prefetchers find an offset or stride that results in the highest prefetch coverage. The Sandbox Prefetcher [66] uses bloom filters to choose among a few pre-defined aggressive offset prefetchers. Many contestants in the Data Prefetching Championship 2015 [1] propose modifications to the Sandbox Prefetcher to improve timeliness. In particular, the Best Offset Prefetcher [57] uses a delay queue to evaluate offsets that are more likely to be timely.

Conceptually, the Best Offset Prefetcher is very similar to the ASP, which was developed concurrently. The two prefetchers share the same goals as the aggregate

stride is likely to emerge as the best offset for any delta pattern. However, the mechanisms used by the two prefetchers to achieve this goal are different. ASP's algorithm has the small advantage that it can detect and prefetch any offset, whereas the Best Offset Prefetcher is limited to a fixed set of pre-defined offset values. But for many scientific applications, this benefit is not significant as these applications tend to use a small number of strides. Thus, with the right set of candidate strides, the Best Offset Prefetcher can be tuned to replicate ASP's results.

We also note that the presentation of ASP in this thesis includes an in-depth analysis and evaluation of aggregate strides. In particular, we present detailed insights and the first quantitative and qualitative comparison of aggregate strides (or offset prefetching) with both order-based and spatial stride prefetchers. Our study is the first to comprehensively demonstrate why offset-based prefetchers are fundamentally superior to order-based and spatial prefetchers(see Section 5.4).

Chapter 3

The Hawkeye Cache

Caches are important mechanisms for reducing the long latencies of DRAM memory accesses, and their effectiveness is significantly influenced by their replacement policy. Unfortunately, cache replacement is a difficult problem. Unlike problems such as branch prediction, in which the definitive answer to the question, "Will this branch be taken?", will be readily available in a few cycles, it is difficult to get the definitive answer to the question, "Which cache line should be evicted?"

In the absence of definitive feedback, existing replacement policies build on heuristics, such as Least Recently Used (LRU) and Most Recently Used (MRU), which each work well for different workloads. However, even with increasingly clever techniques for optimizing and combining these policies, these heuristic-based solutions are each limited to specific classes of access patterns and are unable to perform well in more complex scenarios. As a simple example, consider the naive triply nested loop algorithm for computing matrix multiplication. As depicted in Figure 3.1, the elements of the C matrix enjoy short-term reuse, while those of the A matrix enjoy medium-term reuse, and those of the B matrix see long-term reuse. Figure 3.1 shows that existing replacement policies can capture some subset of the available reuse, but only Belady's algorithm [5] can effectively exploit all three forms of reuse.

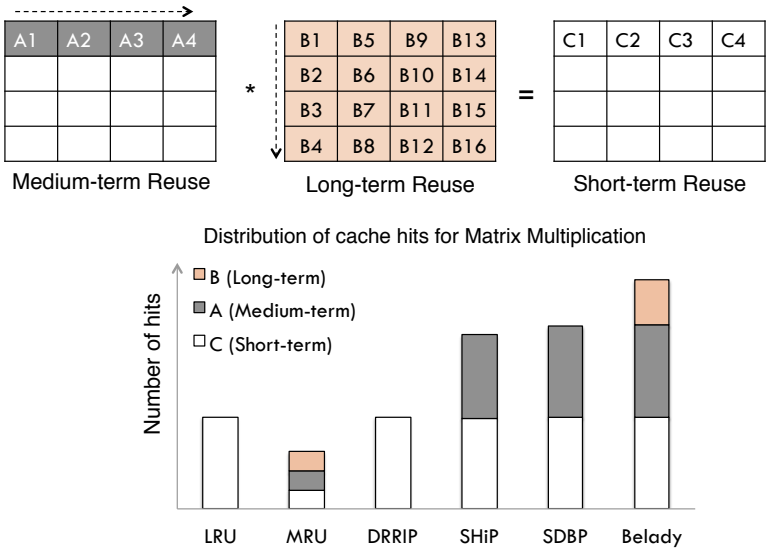

Figure 3.1: Existing replacement policies are limited to a few access patterns and are unable to cache the optimal combination of A, B and C.

In this thesis, we present a fundamentally different approach, one that is not based on LRU, on MRU, or on any heuristic that is geared towards any particular class of access patterns. Our algorithm is instead based on Belady's algorithm: While Belady's algorithm is impractical because it requires knowledge of the future, we show that it is possible to apply a variant of Belady's algorithm to the history of past memory accesses. If past behavior is a good predictor of future behavior, then our replacement policy will approach the behavior of Belady's algorithm. We refer to the decisions made by Belady's algorithm as OPT. To learn the past behavior of Belady's algorithm, we observe that if with the OPT solution a load instruction has historically brought in lines that produce cache hits, then in the future, the same load instruction is likely to bring in lines that will also produce cache hits.

Our new cache replacement strategy thus consists of two components. The first reconstructs Belady's optimal solution for past cache accesses. The second is a predictor that learns OPT's behavior of past PCs to inform eviction decisions for

future loads by the same PCs.

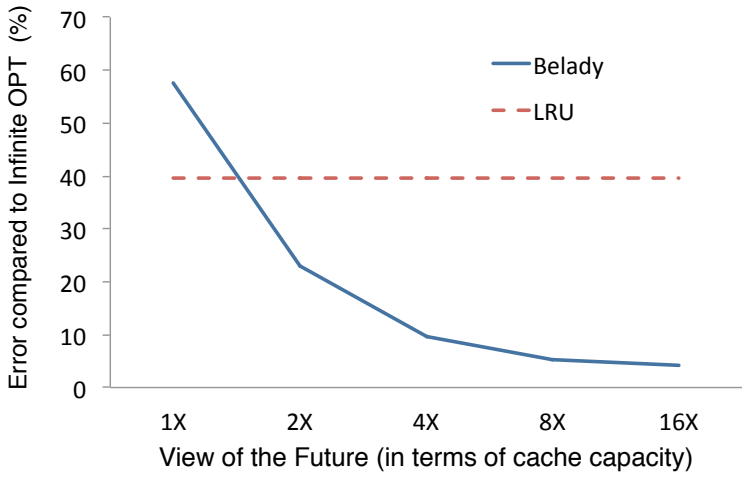

Figure 3.2: Belady's algorithm requires a long view of the future.

One concern with this idea is that Belady's algorithm looks arbitrarily far into the future, so our solution would theoretically need to remember an arbitrarily long history of past events. However, Figure 3.2 shows the impact of limiting this window of the future. Here, $1\times$ represents a window that consists of accesses to k cache lines, where k is the capacity of the cache. We see that while Belady's algorithm performs better when it can see farther into the future, it approaches the performance of a true OPT policy when given a reuse window of $8\times$ the cache size. Thus, we dub our new replacement policy Hawkeye[1].

Our use of OPT introduces two technical challenges. First, we need an efficient mechanism of reconstructing OPT. Second, a long history is needed to compute OPT. We solve the first problem by using the notion of liveness intervals (see Section 3.1), which leads to a simple and efficient solution. The use of liveness intervals is novel for cache replacement, because it explicitly conveys information

[1]Hawks are known for their excellent long-range vision and can see up to $8\times$ more clearly than the best humans.

about both reuse distance and the demand on the cache, which are both essential for making proper eviction decisions. We solve the second problem by using Set Dueling [68] to sample a small subset of all cache lines.

The result is that with 16 KB of additional storage (plus tag storage), Hawkeye can compute OPT's solution for past accesses with 99% accuracy. Of course, past behavior does not always model future behavior, so Hawkeye's performance does not match OPT's. Nevertheless, as shown in Figure 3.3, Hawkeye performs significantly better than previous policies on a memory-intensive SPEC CPU 2006 benchmarks.

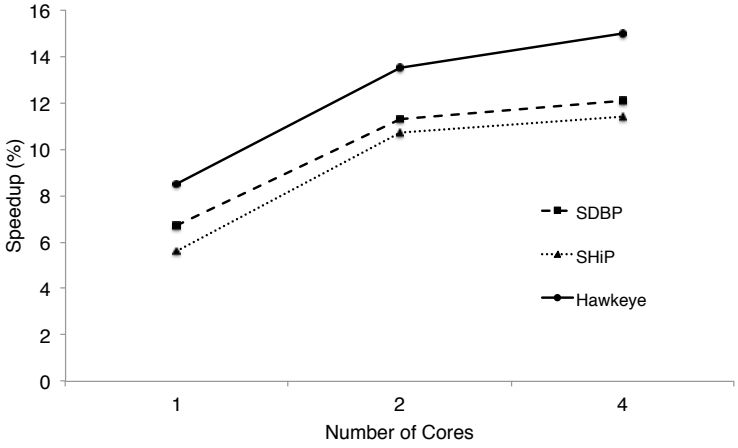

Figure 3.3: Speedup over LRU for 1, 2, and 4 cores.

To summarize, this chapter makes the following contributions:

- We introduce the Hawkeye cache replacement policy, which learns Belady's optimal solution (OPT) for past accesses to guide future replacement decisions.
- We introduce the OPTgen algorithm for efficiently computing OPT for a history of past cache accesses. OPTgen builds on three critical insights: (1) OPT's decision depends not only on a cache line's reuse interval but also on the overlap of reuse intervals, which represents the demand on the cache; (2)

OPT's decision for a past access can be determined at the time of its next use; (3) a reuse window of 8× is necessary to generate OPT's solution accurately.

- To allow Hawkeye to practically simulate OPT, we use Set Dueling [68] to capture long-term behavior with a small 12KB hardware budget.

- We evaluate Hawkeye using the Cache Replacement Championship simulator [3] and show that Hawkeye substantially improves upon the previous state-of-the art. On the SPEC CPU 2006 benchmark suite, Hawkeye obtains a 17.0% miss rate reduction over LRU, compared with 11.4% for Khan, et al.'s SDBP policy [49]. In terms of performance, Hawkeye improves IPC over LRU by 8.4%, while SDBP improves IPC by 6.2%. On a 4-core system, Hawkeye improves speedup over LRU by 15.0%, while SDBP improves speedup by 12.0%.

This chapter is organized as follows. Section 2.1 discusses related work. Sections 3.1 and 3.2 describe and evaluate our solution. We conclude in Section 3.3.

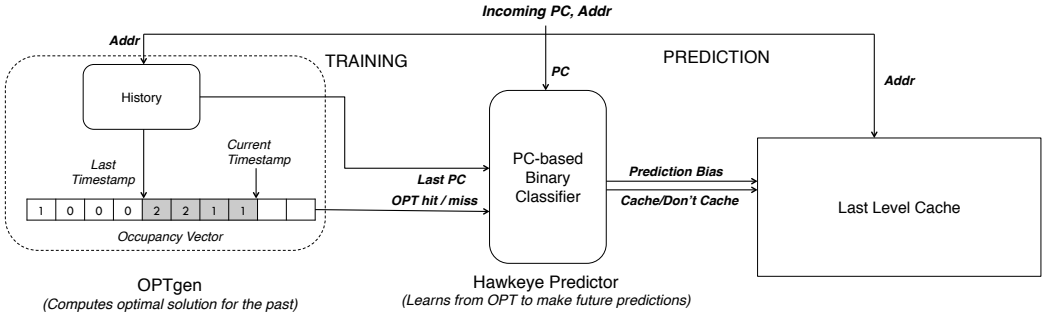

Figure 3.4: Block diagram of the Hawkeye replacement algorithm.

3.1 Our Solution

Conceptually, we view cache replacement as a binary classification problem, where the goal is to determine if an incoming line is *cache-friendly* or *cache-averse:* Cache-

friendly lines are inserted with a high priority, while cache-averse lines are marked as eviction candidates for future conflicts. To determine how incoming lines should be classified, Hawkeye reconstructs Belady's optimal solution for past accesses to learn the behavior of individual load instructions.

Figure 3.4 shows the overall structure of Hawkeye. Its main components are the Hawkeye Predictor, which makes eviction decisions, and OPTgen, which simulates OPT's behavior to produce inputs that train the Hawkeye Predictor. The system also includes a Sampler (not shown), which reduces the amount of state required to reconstruct OPT's behavior. We now describe each component in more detail.

3.1.1 OPTgen

OPTgen determines what would have been cached if the OPT policy had been used. Starting from the oldest reference and proceeding forward in time, OPTgen assigns available cache capacity to lines in the order that they are reused. To assign cache capacity to old references, OPTgen repeatedly answers the following question: Given a history of memory references that includes a reference to cache line X, would the next reference to the same line, which we refer to as X', be a hit or a miss under the OPT policy?

To answer this question, we observe that OPT's decision for any past reference X can be determined at the time of its next reuse X' because any later reference is farther into the future than X', so Belady's algorithm would favor X' over that other line [6]. Thus, we define the time period that starts with a reference to X and proceeds up to (but not including) its next reference X' to be X's *usage interval*. Intuitively, X's usage interval represents its demand on the cache, which allows OPTgen to determine whether the reference to X' would result in a cache hit.

If we further define a cache line's *liveness interval* to be the time period during which that line resides in the cache under the OPT policy, then X would be a cache miss if at any point in its usage interval the number of overlapping liveness intervals matches the cache's capacity. Otherwise, X would be a cache hit.

For example, consider the sequence of accesses in Figure 3.5, which includes X's usage interval. Here, the cache capacity is two. We assume that OPTgen has already determined the liveness intervals of A, B, and C, and since these intervals never overlap, the maximum number of overlapping liveness intervals in X's usage interval never reaches the cache capacity; thus there is space for line X to reside in the cache, and OPTgen infers that X' would be a hit.

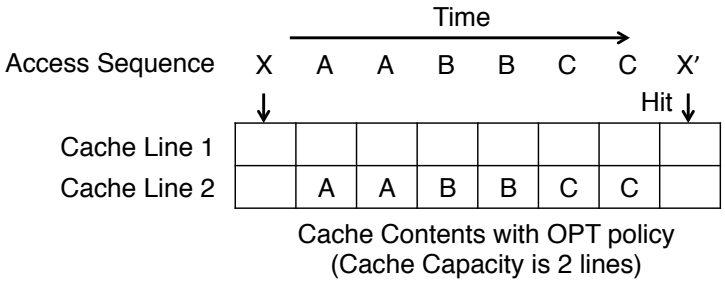

Figure 3.5: Intuition behind OPTgen.

OPTgen uses an *occupancy vector* to record the occupied cache capacity over time; each entry of this vector contains the number of liveness intervals that overlap at a particular time. To understand OPTgen's use of the occupancy vector, consider the example access stream in Figure 3.6(a) and OPT's solution for this access stream in Figure 3.6(b). Figure 3.6(c) shows how the occupancy vector is computed over time. In particular, the top of Figure 3.6(c) shows the sequence of lines that is accessed over time. For example, line B is accessed at Times 1 and 2. Each row in Figure 3.6(c) represents the state of the occupancy vector at a different point in time, so, for example, the third row (T=2) illustrates the state of the occupancy

vector after Time 2, i.e., after the second access of line B and after OPTgen has determined that OPT would have placed B in the cache at Time 1.[2]

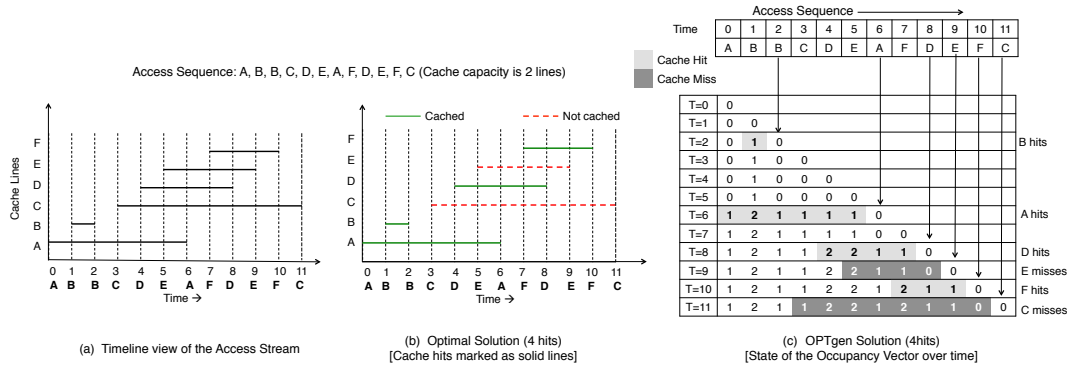

Figure 3.6: Example to illustrate OPTgen.

For an access to X, the occupancy vector for the usage interval (shown in gray) is updated as follows:

- The most recent entry of the occupancy vector (corresponding to this access to X) is set to 0.
- When line X is loaded for the first time, no further changes to the occupancy vector are made, reflecting the fact that OPT makes decisions based on the next reuse of the line.
- If X is not a first-time load, OPTgen checks to see if every element corresponding to the usage interval is less than the cache capacity: If so, then OPT would have placed X in the cache, so the shaded portions of the occupancy vector are incremented; if not, then X would have been a cache miss, so the occupancy vector is not modified.

[2]In this discussion, we will use "T=1" to refer to rows of the figure, and we will use "Time 1" to refer to events in the Access Sequence, ie, columns in the figure.

For example, in Figure 3.6(c), consider the access of D at Time 8. Using the occupancy vector before T=8 (same as the occupancy vector at T=7 with a 0 added for Time 8), OPTgen sees that the elements in the usage interval (the values at positions 4 through 7) are all less than the cache capacity (2), so it concludes that D would be found in the cache at Time 8, and it increments the elements in positions 4 through 7.

As another example, consider the access to C at Time 11; some of the shaded elements have value 2, so OPTgen concludes that this access to C would have been a cache miss, so it does not increment the shaded elements of the occupancy vector. We see that by not incrementing any of the shaded elements of the occupancy vector for cache misses, OPTgen assumes that misses will bypass the cache. If we wanted OPTgen to instead assume a cache with no bypassing, then the most recent entry (corresponding to the current access) would have been initialized to 1 instead of 0.

The example in Figure 3.6 highlights two important points. First, by reconstructing OPT, OPTgen is able to recognize both long-term and short-term reuse that is cache friendly. For example, both A and B hit in the cache even though the reuse interval of A far exceeds the cache capacity. Second, OPTgen can be implemented in hardware with very little logic because the occupancy vector can be maintained with simple read, write, and compare operations.

OPTgen for Set-Associative Caches For set-associative caches, OPTgen maintains one occupancy vector for each cache set such that the maximum capacity of any occupancy vector entry never exceeds the cache associativity. Occupancy vectors measure time in terms of cache accesses to the corresponding set, and they include enough entries to model 8× the size of the set (or the associativity). Thus, for a 16-way set-associative cache, each occupancy vector has 128 entries (corresponding to 8× the capacity of the set), and each occupancy vector entry is 4 bits wide, as its value cannot exceed 16.

3.1.2 Reducing the Size of OPTgen

So far, our discussion of OPTgen has not considered resource constraints, as we have assumed that the occupancy vector measures time in terms of individual cache accesses. We have also assumed that OPTgen has knowledge of liveness intervals that extend back 8× the size of the cache, which for a 16-way 2MB cache requires OPTgen to track over 260K entries in both the occupancy vectors and the history. This section describes two techniques that reduce these hardware requirements.

Granularity of the Occupancy Vector

To reduce the size of the occupancy vector, we increase its granularity so that each element represents a *time quantum*, a unit of time as measured in terms of cache accesses. Our sensitivity studies (Section 3.2) show that a time quantum of 4 cache accesses works well, which for a 16-way set-associative cache reduces the size of the occupancy vector from 128 to 32 entries.

Since occupancy vector entries for 16-way set-associative caches are 4 bits wide, the occupancy vector for each set requires 16 bytes of storage, which for a 2MB cache would still amount to 32KB storage for all occupancy vectors (2048 sets × 16 bytes per set).

Set Dueling

To further reduce our hardware requirements, we use the idea of Set Dueling [68], which monitors the behavior of a few randomly chosen sets to make predictions for the entire cache. To extend Set Dueling to Hawkeye, OPTgen reconstructs the OPT solution for only 64 randomly chosen sets. Section 3.2.3 shows that reconstructing the OPT solution for 64 sets is sufficient to emulate the miss rate of an optimal cache and to train the Hawkeye Predictor appropriately.

Set Dueling reduces Hawkeye's storage requirements in two ways. First, since

OPTgen now maintains occupancy vectors for 64 sets, the storage overhead for all occupancy vectors is only 1 KB (64 occupancy vectors $\times$ 16 bytes per occupancy vector). Second, it dramatically reduces the size of the history, which now tracks usage intervals for only 64 sampled sets.

To track usage intervals for the sampled sets, we use a Sampled Cache. The Sampled Cache is a distinct structure from the LLC, and each entry in the Sampled Cache maintains a 2-byte address tag, a 2-byte load instruction PC, and a 1-byte timestamp. For 64 sets, the Sampled Cache would need to track a maximum of 8K addresses to capture usage intervals spanning a history of 8$\times$ the size of the cache, but we find that because of repeated accesses to the same address, 2400 entries in the Sampled Cache are enough to provide an 8$\times$ history of accesses. Thus the total size of the Sampled Cache is 12KB, and we use an LRU policy for eviction when the Sampled Cache is full.

3.1.3 The Hawkeye Predictor

The second major component of Hawkeye is a predictor that classifies the lines loaded by a given PC as either cache-friendly or cache-averse. This predictor builds on the observation that the majority of OPT's decisions for loads by a given PC are similar and therefore predictable. Figure 3.7 quantifies this observation, showing that for SPEC2006, the average per-PC *bias*—the probability that loads by the same PC have the same caching behavior as OPT—is 90.4%.

Thus, the Hawkeye Predictor learns whether loads by a given instruction would have resulted in hits or misses under the OPT policy: If OPTgen determines that a line X would be a cache hit under the OPT policy, then the PC that last accessed X is trained positively; otherwise, the PC that last accessed X is trained negatively. The Hawkeye Predictor has 8K entries, it uses 3-bit counters for training, and it is indexed by a 13-bit hashed PC.

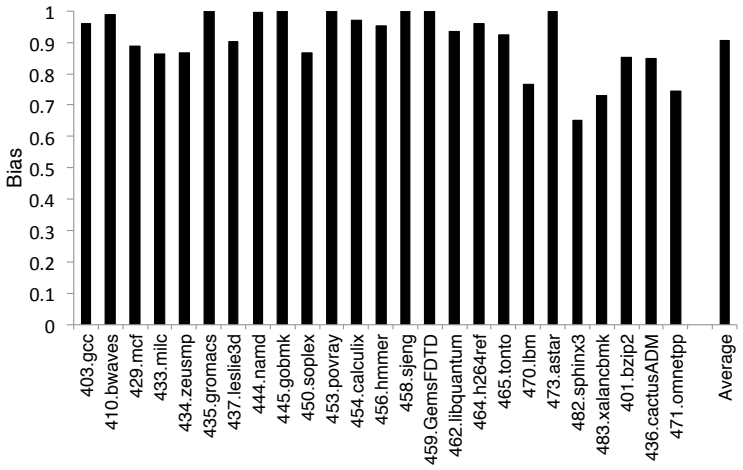

Figure 3.7: Bias of OPT's decisions for PCs.

For every cache access, the predictor is indexed by the current load instruction, and the high-order bit of the corresponding 3-bit counter indicates whether the line is cache-friendly (1) or cache-averse (0). As we explain in Section 3.1.4, this prediction determines the line's replacement state.

Occasionally, load instructions will have a low bias, which will result in inaccurate predictions. Our evaluation shows that we can get a small performance gain by augmenting Hawkeye's predictions to include confidence, but the gains are not justified by the additional hardware complexity, so we do not evaluate this feature.

3.1.4 Cache Replacement

Our overall cache replacement goal is to use Hawkeye's predictions within a phase and to use an LRU strategy at phase change boundaries, when Hawkeye's predictions are likely to be incorrect. Thus, Hawkeye first chooses to evict cache-averse lines, as identified by the Hawkeye Predictor. If no lines are predicted to be cache-averse, then the oldest cache-friendly line (LRU) is evicted, allowing Hawkeye to adapt to phase changes. This scheme is likely to evict cache-averse lines from the new

Hawkeye Prediction \ Hit or Miss	Cache Hit	Cache Miss
Cache-averse	RRIP = 7	RRIP = 7
Cache-friendly	RRIP = 0	RRIP = 0; Age all lines: if (RRIP ¡ 6) RRIP++;

Table 3.1: Hawkeye's Update Policy.

working set before evicting cache-friendly lines from the old working set, but this behavior is harmless because cache-averse lines from the new working set are likely to be evicted anyway. To correct the state of the predictor after a phase change, the predictor is detrained when cache-friendly lines are evicted. In particular, when a cache-friendly line is evicted, the predictor entry corresponding to the last load PC of the evicted line is decremented if the evicted line is present in the sampler.

To implement this policy efficiently, we associate all cache-resident lines with 3-bit RRIP counters [41] that represent their eviction priorities; lines with a high eviction priority have a high RRIP value, and lines with low eviction priorities have a low RRIP value. Conceptually, the RRIP counter of a line combines information about Hawkeye's prediction for that line and its age. On every cache access (both hits and misses), the Hawkeye predictor generates a binary prediction to indicate whether the line is cache-friendly or cache-averse, and this prediction is used to update the RRIP counter as shown in Table 3.1. In particular, rows in Table 3.1 represent Hawkeye's prediction for a given access, and columns indicate whether the access was a cache hit or miss. For example, if the current access hits in the cache and is predicted to be cache-averse, then its RRIP value is set to 7. As another example, when a newly inserted line (cache miss) is predicted to be cache-friendly, its RRIP value is set to 0, and the RRIP values of all other cache-friendly lines are

incremented to track their relative age. In general, cache-friendly lines are assigned an RRIP value of 0, and cache-averse lines are assigned an RRIP value of 7.

On a cache replacement, any line with an RRIP value of 7 (cache-averse line) is chosen as an eviction candidate. If no line has an RRIP value of 7, then Hawkeye evicts the line with the highest RRIP value (oldest cache-friendly line) and detrains its load instruction if the evicted line is present in the sampler.

Hawkeye's insertion policy differs in three ways from other RRIP-based policies [41, 96]. First, lines that are predicted to be cache-friendly are never saturated to the highest value, which ensures that cache-averse lines are always prioritized for eviction. Second, lines that are predicted to be cache-friendly are always assigned an RRIP value of 0 regardless of whether they were hits or misses. And finally, cache hits are promoted to an RRIP value of 0 only if they are predicted to be cache-friendly. These differences are designed to give the Hawkeye Predictor greater influence over the RRIP position than cache hits or misses.

3.2 Evaluation

3.2.1 Methodology

We evaluate Hawkeye using the simulation framework released by the First JILP Cache Replacement Championship (CRC) [3], which is based on CMP$im [39] and models a 4-wide out-of-order processor with an 8-stage pipeline, a 128-entry reorder buffer and a three-level cache hierarchy. The parameters for our simulated memory hierarchy are shown in Table 3.2. The infrastructure generates cache statistics as well as overall performance metrics, such as IPC.

Benchmarks. We evaluate Hawkeye on the entire SPEC2006 benchmark suite.[3] For brevity, Figures 3.8 and 3.9 show averages for all the benchmarks but only include

[3]We currently cannot run perl on our platform, leaving us with 28 benchmark programs.

L1 I-Cache	32 KB 4-way, 1-cycle latency
L1 D-Cache	32 KB 4-way, 1-cycle latency
L2 Cache	256KB 8-way, 10-cycle latency
Last-level Cache	2MB, 16-way, 20-cycle latency
DRAM	200 cycles
Two-core	4MB shared LLC (25-cycle latency)
Four-core	8MB shared LLC (30-cycle latency)

Table 3.2: Baseline configuration.

bar charts for the 20 replacement-sensitive benchmarks that show more than 2% improvement with the OPT policy. We compile the benchmarks using gcc-4.2 with the -O2 option. We run the benchmarks using the reference input set, and as with the CRC, we use SimPoint [63, 30] to generate for each benchmark a single sample of 250 million instructions. We warm the cache for 50 million instructions and measure the behavior of the remaining 200 million instructions.

Multi-Core Workloads. Our multi-core results simulate either two benchmarks running on 2 cores or four benchmarks running on 4 cores, choosing all combinations of the 12 most replacement-sensitive SPEC2006 benchmarks. For 2 cores, we simulate all possible combinations, and for 4 cores, we randomly choose one tenth of all the workload mixes, resulting in a total of 136 combinations. For each combination, we simulate the simultaneous execution of the SimPoint samples of the constituent benchmarks until each benchmark has executed at least 250M instructions. If a benchmark finishes early, it is rewound until every other application in the mix has finished running 250M instructions. Thus, all the benchmarks in the mix run simultaneously throughout the sampled execution. Our multi-core simulation methodology is similar to the methodologies used by recent work [41, 96, 49] and the Cache Replacement Championship [3].

To evaluate performance, we report the weighted speedup normalized to LRU

for each benchmark combination. This metric is commonly used to evaluate shared caches [49, 40, 83, 98] because it measures the overall progress of the combination and avoids being dominated by benchmarks with high IPC. The metric is computed as follows. For each program sharing the cache, we compute its IPC in a shared environment (IPC_{shared}) and its IPC when executing in isolation on the same cache (IPC_{single}). We then compute the weighted IPC of the combination as the sum of $IPC_{shared}/IPC_{single}$ for all benchmarks in the combination, and we normalize this weighted IPC with the weighted IPC using the LRU replacement policy.

Evaluated Caching Systems. We compare Hawkeye against two state-of-the-art cache replacement algorithms, namely, SDBP [49] and SHiP [96]; like Hawkeye, both SHiP and SDBP learn caching priorities for each load PC. We also compare Hawkeye with two policies that learn global caching priorities, namely, Dueling Segmented LRU with Adaptive Bypassing [27] (DSB, winner of the 2010 Cache Replacement Championship) and DRRIP [41].

DRRIP and SHiP use 2-bit re-reference counters per cache line. For SHiP, we use a 16K entry Signature Hit Counter Predictor with 3-bit counters. For SDBP, we use a 1-bit dead block prediction per cache line, 8KB sampler, and 3 prediction tables, each with 4K 2-bit counters. Our SDBP and SHiP implementation is based on the code provided by the respective authors with all parameters tuned for our execution. For DSB, we use the code provided on the CRC website and explore all tuning parameters. To simulate Belady's OPT, we use an in-house trace-based cache simulator. Hawkeye's configuration parameters are listed in Table 3.3.

For our multi-core evaluation, the replacement policies use common predictor structures for all cores. In particular, Hawkeye uses a single occupancy vector and a single predictor to reconstruct and learn OPT's solution for the interleaved access

stream from the past.

Power Estimation. We use CACTI [80] to estimate the dynamic energy consumed by the various replacement policies. Our energy estimates are limited to the additional components introduced by the replacement policy and do not consider the impact of improved cache performance on system-wide energy consumption.

3.2.2 Comparison with Other Policies

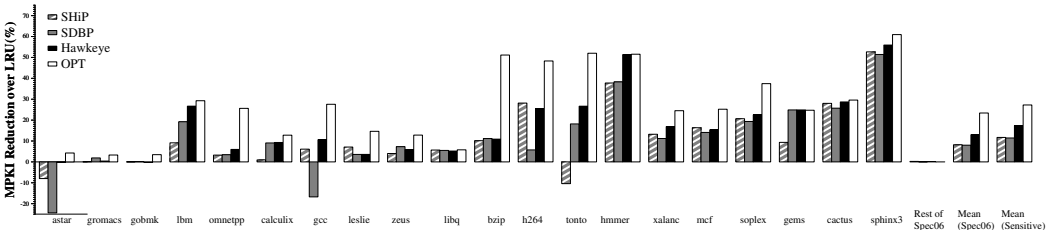

Figure 3.8: Miss rate reduction for all SPEC CPU 2006 benchmarks.[4]

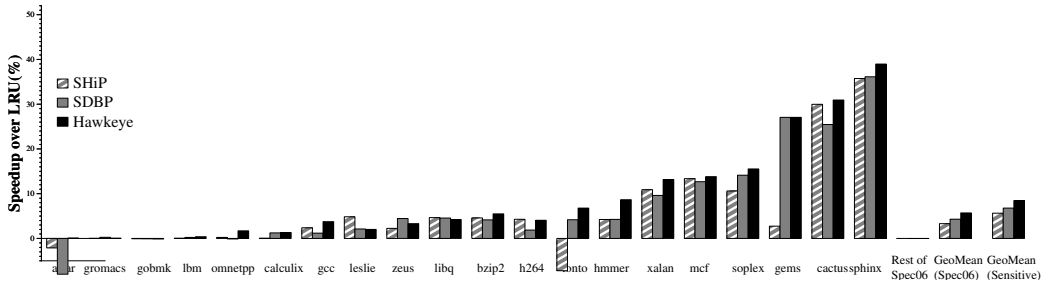

Figure 3.9: Speedup comparison for all SPEC CPU 2006 benchmarks.

Figure 3.8 shows that Hawkeye significantly reduces the LLC miss rate in comparison with the two state-of-the-art replacement policies. In particular, Hawkeye achieves an average miss reduction of 17.0% on the 20 memory-intensive SPEC benchmarks, while SHiP and SDBP see miss reductions of 11.7% and 11.4%, respectively. Figure 3.9 shows that Hawkeye's reduced miss rate translates to a speedup

[4]The results for the replacement-insensitive benchmarks (bwaves, milc, povray, dealII, sjeng, wrf, gamess and namd) are averaged in the a single bar named "Rest of SPEC".

of 8.4% over LRU. By contrast, SHiP and SDBP improve performance over LRU by 5.6% and 6.2%, respectively.

Figure 3.9 demonstrates two important trends. First, SDBP and SHiP each perform well for different workloads, but their performance gains are not consistent across benchmarks. For example, SHiP achieves the best performance for cactus, mcf, and sphinx but performs poorly on gems and tonto. By contrast, Hawkeye performs consistently well on all the workloads. Second, in contrast with the other replacement policies, Hawkeye does not perform worse than the LRU baseline on any of the benchmarks. For example, SHiP and SDBP both slow down astar, and they increase the miss rates of tonto and gcc, respectively. These results reinforce our claim that previous replacement policies are geared to specific classes of access patterns, whereas by learning from OPT, Hawkeye can adapt to any workload.

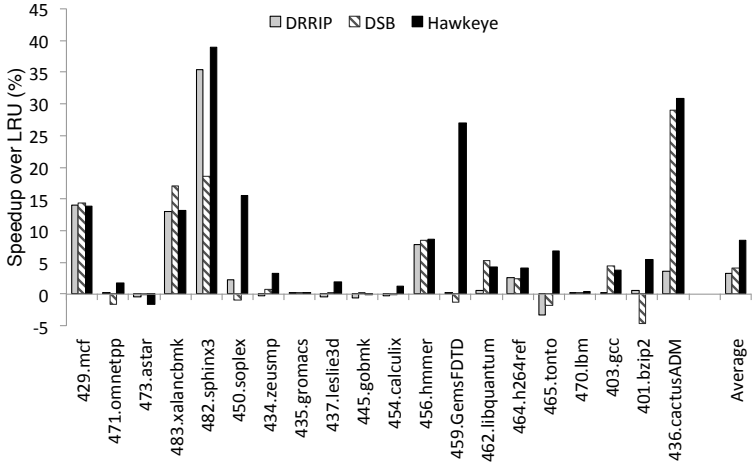

Figure 3.10: Comparison with DRRIP and DSB.

Finally, Figure 3.10 shows that Hawkeye's performance improvement over LRU is much greater than DRRIP's (3.3% vs. 8.4%) and almost twice as large as DSB's (4.2% vs 8.4%). To understand Hawkeye's benefit, we observe that DRRIP learns a single policy for the entire cache, while DSB learns a single bypassing priority. By contrast, Hawkeye can learn a different priority for each load PC. Since

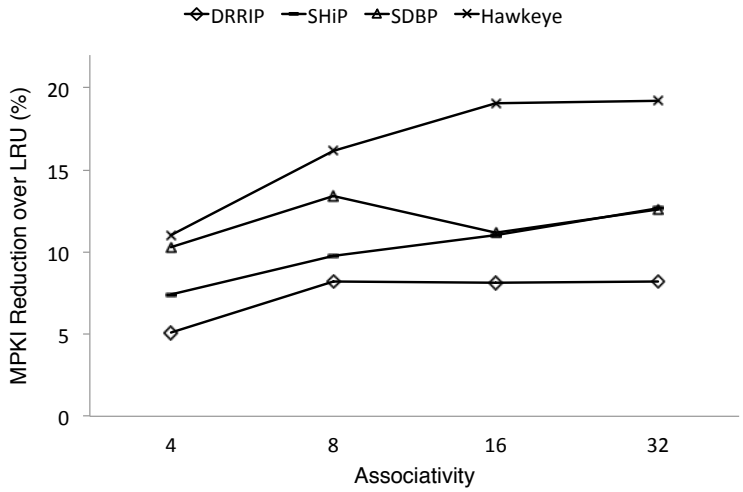

Figure 3.11: Sensitivity to cache associativity.

it is common for cache-friendly and cache-averse lines to occur simultaneously, any global cache priority is unlikely to perform as well as Hawkeye.

Sensitivity to Cache Associativity. Higher associativity gives a replacement policy more options to choose from, and since Hawkeye is making more informed decisions than the other policies, its advantage grows with higher associativity, as shown in Figure 3.11.

Except for SDBP, all of the replacement policies benefit from higher associativity. SDBP deviates from this trend because it uses a decoupled sampler[5] to approximate LRU evictions from the cache itself, and its performance is sensitive to a mismatch in the rate of evictions from the cache and the sampler. Hawkeye's decoupled sampler prevents this by employing a strategy that is independent of the cache configuration.

Hardware Overhead.

Table 3.3 shows the hardware budget for Hawkeye's three memory compo-

[5] For each data point in Figure 3.11, we choose the best performing sampler associativity.

Component	Parameters	Budget
Sampler	2400 entries; 5-byte entry	12KB
Occupancy Vector	64 vector, 32 entries each 4-bit entry Quantum=4 accesses	1KB
Hawkeye Predictor	8K entries; 3-bit counter	3KB
Replacement State per line	3-bit RRIP value	12KB

Table 3.3: Hawkeye hardware budget (16-way 2MB LLC)

nents, namely, the sampler, the occupancy vector, and the PC-based predictor. For a 2MB cache, Hawkeye's total hardware budget is 28KB, including the per-cache-line replacement state in the tag array. Table 3.4 compares the hardware budgets for the evaluated replacement policies. We note that Hawkeye's hardware requirements are well within the 32KB budget constraint for the Cache Replacement Championship [3].

Finally, we also observe that like other set dueling based replacement policies, such as SDBP and SHiP, Hawkeye's hardware budget for meta-data storage (Occupancy Vectors and Sampled Cache) does not increase with additional cores or larger caches.

Hardware Complexity. Because every occupancy vector update can modify up to 32 entries, OPTgen would appear to perform 32 writes on every cache access, which would consume significant power and complicate queue management. In fact, the number of updates is considerably smaller for three reasons. First, the occupancy vector is updated only on sampler hits, which account for only 5% of all cache accesses. Second, we implement the occupancy vector as an array of 32-bit lines, such that each line contains eight 4-bit entries of the occupancy vector. On a sampler

Policy	Predictor Structures	Cache Meta-data	Hardware Budget
LRU	None	16KB	16KB
DRRIP	8 bytes	8KB	8KB
SHiP	4KB SHCT 2KB PC tags	8KB	14KB
SDBP	8KB sampler 3KB predictor	16KB	27KB
Hawkeye	12KB sampler 1KB OPTgen 3KB predictor	12KB	28KB

Table 3.4: Comparison of hardware overheads.

hit, all eight entries in the cache line can be updated in parallel using a modified 32-bit adder (which performs addition on 4-bit chunks of the line). As a result, a sampler hit requires at most 4 writes to the occupancy vector. Third, Figure 3.12 shows the distribution of the number of entries that are updated on each sampler hit: 85% of these accesses update 16 entries or fewer, which means that they modify no more than 2 lines and can complete in 2 cycles. Moreover, 65% of the accesses update 8 entries or fewer, so they modify no more than 1 line. Because occupancy vector updates are not on the critical path, these latencies do not affect performance.

Energy Consumption. Hawkeye does not increase the energy consumption of cache lookups or evictions, but it consumes extra energy for the sampler, the predictor, and the occupancy vector. We compute the dynamic energy consumption of each of these components by computing the energy per operation using CACTI and by computing the number of probes to each component as a fraction of the total number of LLC accesses. While the predictor is probed on every LLC access, the sampler is only probed for LLC accesses belonging to the sampled cache sets, and the occupancy vector is accessed only for sampler hits. As shown in Figure 3.12, the great majority of the occupancy vector updates modify no more than 4 lines.

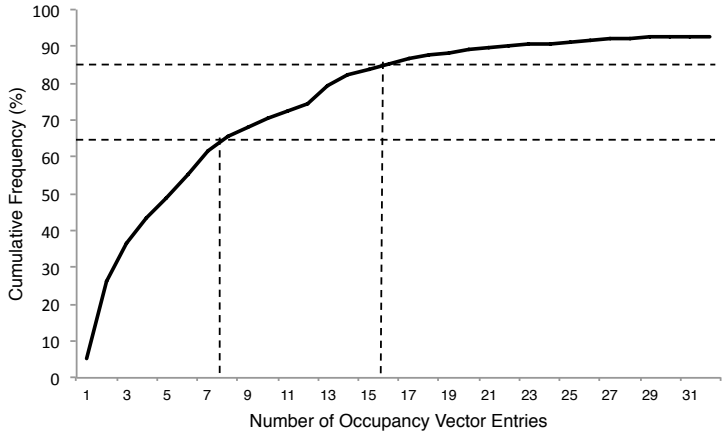

Figure 3.12: CDF of number of Occupancy Vector entries updated on sampler hits. 85% of the accesses update 16 entries or fewer, so they modify no more than 2 lines.

We find that the Hawkeye Predictor, sampler, and occupancy vector consume 0.4%, 0.5% and 0.1%, respectively, of the LLC's dynamic energy consumption, which results in a total energy overhead of 1% for the LLC. Thus, Hawkeye's energy overhead is similar to SDBP's (both Hawkeye and SDBP use a decoupled sampler and predictor), while SHiP's energy overhead is 0.5% of the LLC because it does not use a decoupled sampler.

3.2.3 Analysis of Hawkeye's Performance

There are two aspects of Hawkeye's accuracy: (1) OPTgen's accuracy in reconstructing the OPT solution for the past, and (2) the Hawkeye Predictor's accuracy in learning the OPTgen solution. We now explore both aspects.

OPTgen Simulates OPT Accurately. Recall from Section 3.1 that OPTgen maintains occupancy vectors for only 64 sampled sets, and each entry of the occupancy vector holds the number of cache-resident lines that the OPT policy would retain in a given time quantum.

Figure 3.13 shows that OPTgen is accurate when it models occupancy vectors for all sets and uses a time quantum of 1. When sampling 64 sets, OPTgen's accuracy decreases by 0.5% in comparison with the true OPT solution, and with a time quantum of 4 cache accesses, its accuracy further decreases by only 0.3%. Thus, when using a time quantum of 4, OPTgen can achieve 99% accuracy in modeling the OPT solution with 64 occupancy vectors.

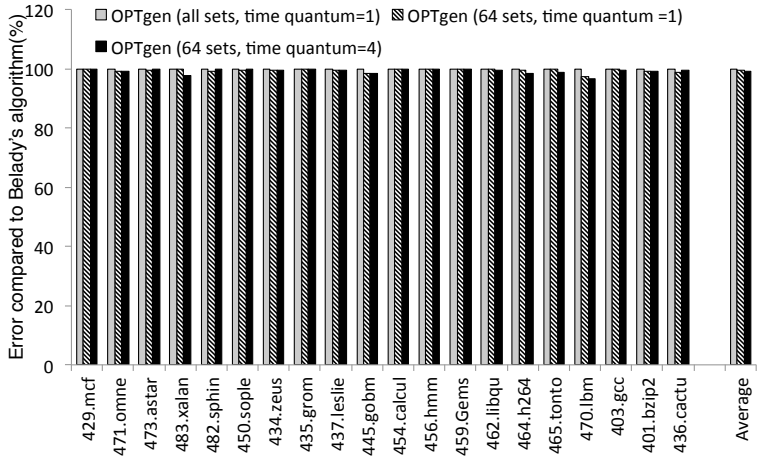

Figure 3.13: OPTgen simulates OPT accurately.

Predictor Accuracy. Figure 3.14 shows that the Hawkeye Predictor is 81% accurate in predicting OPT's decisions for future accesses. There are two sources of inaccuracy: (1) Optimal decisions of the past may not accurately predict the future; (2) the predictor may learn slowly or incorrectly due to resource limitations and training delay. Since the average bias of the OPT solution for load instructions is 91%, we conclude that the predictor contributes to the remaining loss.

Sampler Accuracy. Figure 3.15 shows that on average, a sampled history has little impact on Hawkeye's performance. However, the impact of sampling on Hawkeye's performance varies with benchmark. For benchmarks such as bzip2, calculix, and tonto, sampling actually improves performance because the sampled history not only

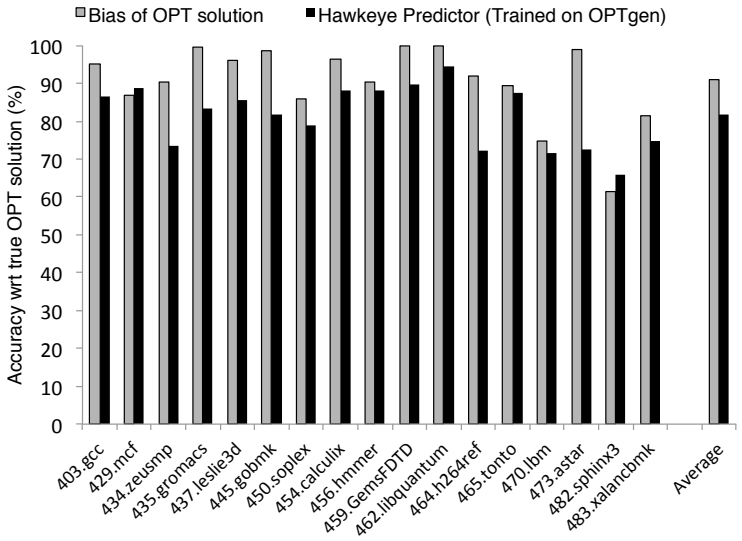

Figure 3.14: Accuracy of the Hawkeye Predictor.

filters out noisy training input, but it also accelerates training by aggregating many training updates into a single event. For zeusmp, soplex, and h264, performance is degraded with a sampled history because the samples are unable to adequately represent the past behavior of these benchmarks.

Distribution of Eviction Candidates. Recall that when all eviction candidates are predicted to be cache-friendly, Hawkeye evicts the oldest line (LRU). Figure 3.16 shows the frequency with which the LRU candidate is evicted. We see that the Hawkeye Predictor accounts for 71% of the overall evictions, though the distribution varies widely across benchmarks. For benchmarks that have a cache-resident working set, such as astar, gromacs, and gobmk, Hawkeye learns that most accesses are cache hits, so it typically defaults to the LRU policy. By contrast, for benchmarks that have a complex mix of short-term and long-term reuse, such as mcf, xalanc, and sphinx, the Hawkeye Predictor accounts for a majority of the evictions, and the LRU evictions occur only during infrequent working set transitions.

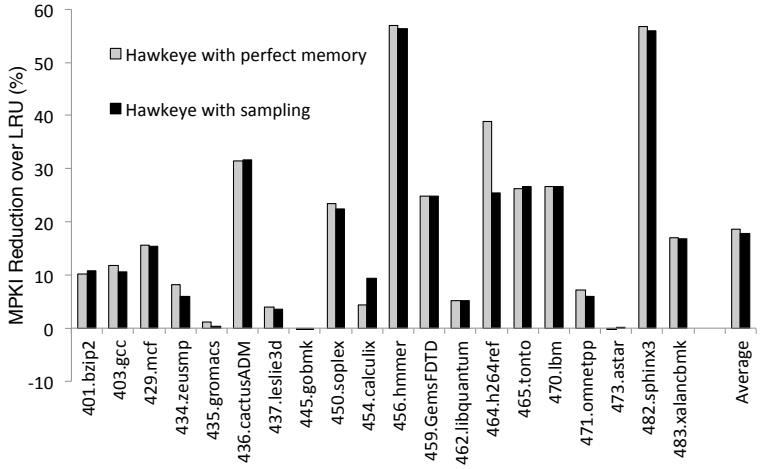

Figure 3.15: Impact of sampling on performance.

3.2.4 Multi-Core Evaluation

The top graph in Figure 3.17 shows that on a 2-core system with a shared LLC, Hawkeye's advantage over SDBP and SHiP increases, as Hawkeye achieves a speedup of 13.5%, while SHiP and SDBP see speedups of 10.7% and 11.3%, respectively. The bottom graph in Figure 3.17 shows that Hawkeye's advantage further increases on a 4-core system, with Hawkeye improving performance by 15%, compared with 11.4% and 12.1% for SHiP and SDBP, respectively. On both 2-core and 4-core systems, we observe that while SHiP outperforms both SDBP and Hawkeye on a few workload mixes, its average speedup is the lowest among the three policies, which points to large variations in SHiP's performance.

Figures 3.18 and 3.3 summarize Hawkeye performance as we increase the number of cores from 1 to 4. We see that Hawkeye's relative benefit over SDBP increases with more cores. We also see that the gap between SHiP and SDBP diminishes at higher core counts.

Scheduling Effects. A key challenge in learning the caching behavior of multi-core systems is the variability that can arise from non-deterministic schedules. Thus, the

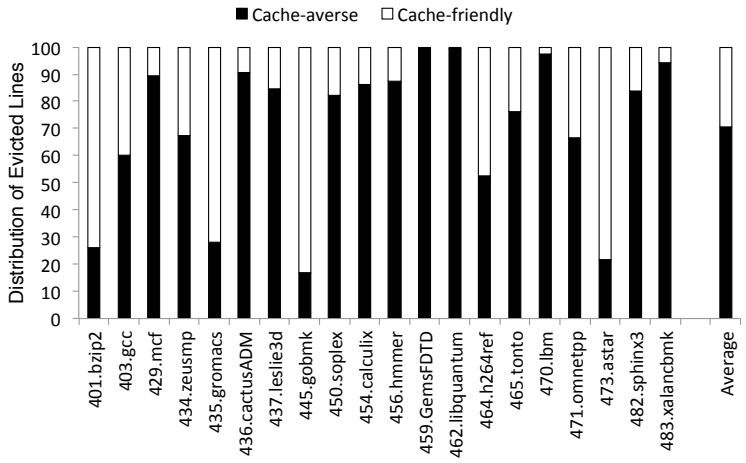

Figure 3.16: Distribution of evicted lines.

optimal solution of the past memory access stream may not represent the optimal caching decisions for the future memory access stream. However, our evaluation shows that for multi-programmed workloads, the average bias of OPT's decisions is 89%, which explains why Hawkeye is beneficial for shared caches.

3.3 Summary

In this chapter, we have introduced the Hawkeye cache replacement policy and shown that while it is impossible to look into the future to make replacement decisions, it *is* possible to look backwards over a sufficiently long history of past memory accesses to learn and mimic the optimal behavior.

The advantage of learning from OPT is that OPT can exploit reuse for any workload, so unlike existing policies, it is not focused on certain types of reuse—e.g., short-term and medium-term. This claim is supported by our empirical results: Unlike other policies, which for some workloads increase the number of cache misses (when compared against LRU), Hawkeye does not increase the number of cache misses for any of our evaluated workloads.

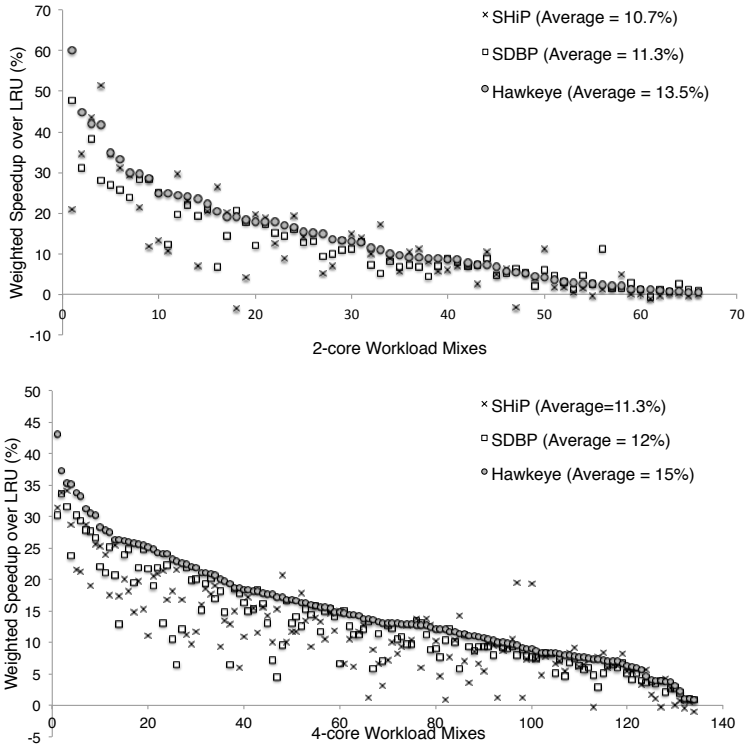

Figure 3.17: Weighted speedup for 2 cores with shared 4MB LLC (top) and 4 cores with shared 8MB LLC (botttom).

Conceptually, Belady's algorithm is superior to work that focuses on reuse distance, because Belady's algorithm directly considers both reuse distance *and* the demand on the cache. Concretely, by learning from OPT, Hawkeye provides significant improvements in miss reductions and in speedup for both single-core and multi-core settings.

More broadly, we have introduced the first method of providing an oracle for training cache replacement predictors. As with the trend in branch prediction, we expect that future work will enhance cache performance by using more sophisticated predictors that learn our oracle solution more precisely. Indeed, given the 99% accuracy with which OPTgen reproduces OPT's behavior, the greatest potential for improving Hawkeye lies in improving its predictor. Finally, we believe that

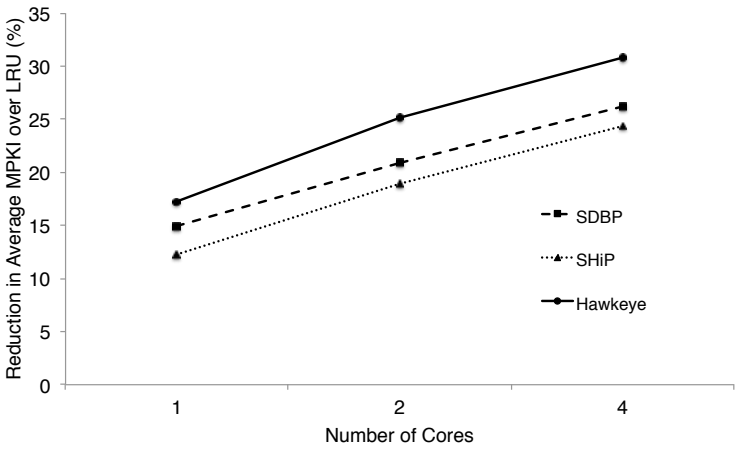

Figure 3.18: Miss reduction over LRU for 1, 2, and 4 cores.

Hawkeye's long history provides information that will be useful for optimizing other parts of the memory system, including thread scheduling for shared memory systems, and the interaction between cache replacement policies and prefetchers.

Chapter 4

Irregular Stream Buffer

Prefetching is an important technique for hiding the long memory latencies of modern microprocessors. For regular memory access patterns, prefetching has been commercially successful because stream and stride prefetchers are effective, small, and simple. For irregular access patterns, prefetching has proven to be more problematic. Numerous solutions have been proposed [3-4, 8-15, 17, 19, 22-23, 25-28, 32-34, 37-44], but there appears to be a basic design tradeoff between storage and effectiveness, with large storage required to achieve good coverage and accuracy [89].

For example, prefetchers based on *address correlation,* the subject of this chapter, identify sequences of correlated memory addresses—also known as *temporal streams*—by learning the most likely successor for a given memory reference. Because this correlation information grows in proportion to the application's memory footprint, the fundamental challenge for these prefetchers is the management of megabytes of off-chip correlation information [16, 91, 93]. Access to this off-chip meta-data increases prediction latency and memory traffic, which reduces the effectiveness of prefetching.

Recent solutions use the Global History Buffer (GHB) [59], which organizes correlation information by storing recent memory accesses in a time-ordered circular

history buffer; a spatially organized index table is used to find addresses within the history buffer (see Figure 4.1). With the temporally ordered history buffer, temporal streams can be efficiently prefetched because each stream is stored contiguously. For address correlation, GHB-based prefetchers can amortize the cost of off-chip meta-data access by fetching long temporal streams [93, 17]. Unfortunately, temporal organizations cannot effectively hide the latency of fetching meta-data for short streams, and even the most optimized implementations incur an average memory traffic overhead of 35% on commercial and scientific workloads [92].

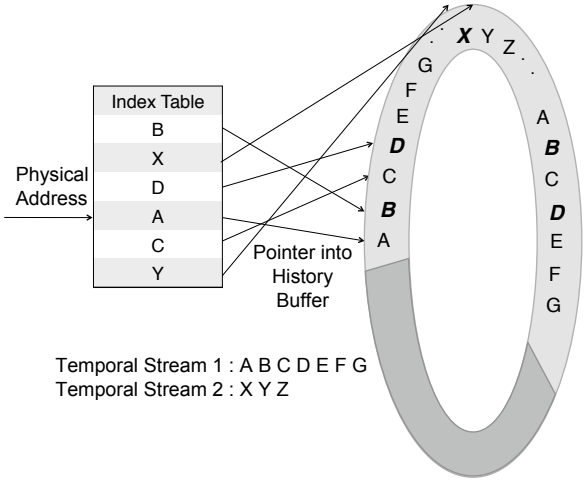

Figure 4.1: Address correlation using the GHB.

One way to reduce the cost of these off-chip accesses would be to cache only the meta-data that correspond to the TLB-resident pages of memory. The movement of this cached meta-data to and from DRAM could then be synchronized with expensive TLB evictions, largely hiding the latency of these off-chip accesses.

Unfortunately, this proposed caching scheme is ill-suited to temporally organized structures such as the GHB. For example, assume in Figure 4.1 that physical addresses `B`, `X`, and `D` reside on the same page; we see that these addresses are scattered throughout the history buffer and are likely to appear multiple times in the

history buffer, so there is no efficient way to evict these entries from the history buffer when their TLB entry is evicted, nor is it easy to reuse the scattered evicted entries of the history buffer.

This thesis introduces the Irregular Stream Buffer (ISB), a new correlation-based prefetcher that employs just such a caching scheme and that provides other significant benefits with respect to coverage and accuracy. The main idea is to introduce an extra level of indirection to create a new *structural address space* in which correlated physical addresses are assigned consecutive structural addresses. The key point is that in this structural address space, streams of correlated memory addresses are both temporally ordered and spatially ordered. For example, we see in Figure 4.2 that a sequential traversal of the structural address space visits the elements of the irregular temporal stream—`A`, `B`, `C`, `D` and `E`—in temporal order. Thus, the problem of prefetching irregular streams is reduced to sequential prefetching in the structural address space. The mapping to and from structural addresses is performed at a cache line granularity by two spatially indexed on-chip address caches whose contents can be easily synchronized with that of the TLB.

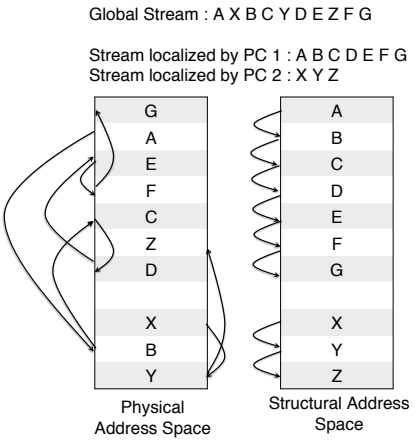

Figure 4.2: Structural address space.

In addition to the reduced memory traffic provided by our caching scheme,

the ISB enjoys several other significant benefits:

- *Improved Prediction Capability:* Unlike GHB-based solutions (see Section 3), the ISB can use PC localization, a technique that segregates the prefetcher input into multiple streams based on the PC of the loading instruction, which is known to improve coverage and accuracy [59, 86, 85, 52]. In particular, the ISB can combine PC localization and address correlation because any PC-localized temporal stream is stored consecutively in the on-chip address cache (see Figure 4.2), i.e., the localization is performed before physical addresses are translated to structural addresses.

- *Training on the Reference Stream:* Because the vast majority of its meta-data accesses are on-chip, the ISB can train on the LLC (last level cache) access stream instead of its miss stream, which significantly improves the predictability of the reference stream. By contrast, most previous prefetchers that use address correlation train on the LLC miss stream to avoid the significant off-chip traffic that would be generated by accessing off-chip meta-data on every LLC access.[1]

- *Support for Short Streams:* The ISB's caching scheme greatly reduces memory traffic overhead for all streams, not just for long streams.

This chapter makes the following contributions:

1. We introduce the ISB, the first prefetcher to combine the use of PC localization and address correlation.

2. We show—using the irregular, memory-intensive subset of the SPEC 2006 benchmarks—that the ISB significantly advances the state-of-the-art in temporal stream prefetching. The ISB obtains 23.1% speedup and 93.7% accuracy,

[1]The STeMS prefetcher can train on the access stream because it searches for coarse-grained temporal streams, relying on a complex spatial prefetcher to fill in the gaps [85].

while an idealized STMS prefetcher, which over-approximates the previous state-of-the-art (STMS) [92], obtains 9.9% speedup and 64.2% accuracy. We also show that the ISB is superior to two other recent prefetchers, SMS [86], which exploits spatial locality, and PC/DC [59, 21], which uses delta correlation instead of address correlation.

3. We introduce a method of organizing data that synchronizes the movement of prefetcher meta-data with TLB misses to reduce memory traffic overhead. For a single core with DDR2 memory, the ISB incurs an average of 8.4% memory traffic overhead due to meta-data access. As a point of comparison, Wenisch, et al. report that the STMS prefetcher produces an average memory traffic overhead of roughly 35% for a mix of commercial and scientific workloads [92].

4. We show that the ISB performs well when combined with a state-of-the-art stride prefetcher (AMPM) [35]. A hybrid that uses an 8 KB ISB achieves a 40.8% speedup over a baseline with no prefetching.

This chapter is organized as follows. Section 2.2 places our work in the context of prior work. Section 4.1 motivates our solution by describing the technical issues with pure spatial and purely temporal organizations of correlation information. Section 4.2 then describes our solution, and Section 4.3 evaluates our solution, before we conclude.

4.1 The Problem

To motivate the benefits of the ISB's structural address space, this section explains the problems caused by purely spatial and purely temporal organizations of correlation information.

Early solutions organize correlated address pairs *spatially* in a *Markov table,* which is indexed by memory address [44]. Unfortunately, Markov tables require

multiple table lookups to prefetch temporal streams. To reduce the number of table lookups, each table entry could store a fixed-length stream [16], but because temporal stream lengths vary widely from two to several hundred [13, 94], it is difficult to optimize for any single stream length. Thus, fixed-length stream entries lead to inefficient use of on-chip storage, with short streams wasting space (see the entries for Tag X and Y in Figure 4.3), and long streams storing data redundantly (see the entries for Tags A, B, C in Figure 4.3).

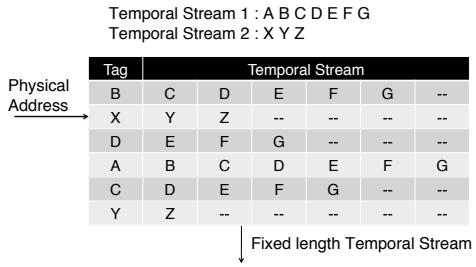

Figure 4.3: Markov Table with fixed length temporal streams.

The GHB instead stores correlation information temporally, which supports efficient temporal stream prefetching. Unfortunately, this temporal organization makes it prohibitively expensive for GHB-based solutions to combine PC localization with address correlation, because linked list traversals are needed to find past occurrences of the triggering memory request (see Figure 4.4). Alternatively, we could imagine allocating a separate fixed-size GHB for each PC, but this solution has issues similar to those of Markov tables: Short streams would waste space, while long streams would require us to chain together multiple GHBs and to follow multiple pointer dereferences to traverse the entire chain. As a result, GHB-based designs forsake either PC localization [93] or address correlation [58, 59, 20], sacrificing significant coverage for design simplicity.

The ISB's structural address space allows the correlation information to be organized both spatially and temporally to provide the advantages of both ap-

proaches: (1) Temporal streams can be efficiently prefetched; (2) the ISB can combine PC localization and address correlation; and (3) the ISB can cache correlation information for just the TLB-resident pages and synch the management of this correlation information with TLB misses.

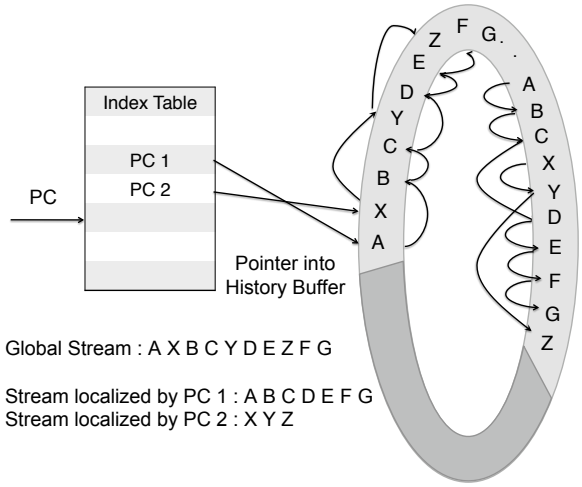

Figure 4.4: PC-localized address correlation using the GHB.

4.2 Our Solution

This section describes our solution by first summarizing the overall ISB design and then providing technical details.

The ISB prediction mechanism mimics the simplicity of stream buffers. Just as stream buffers predict regular memory access patterns, the ISB predicts sequences of memory addresses that are consecutive in the structural address space. Thus, the ISB's prediction step is much simpler than that of other correlation-based prefetchers, which can involve traversals through the GHB.

To enable these predictions, the ISB training mechanism translates correlated physical memory addresses to consecutive structural addresses. The mapping from

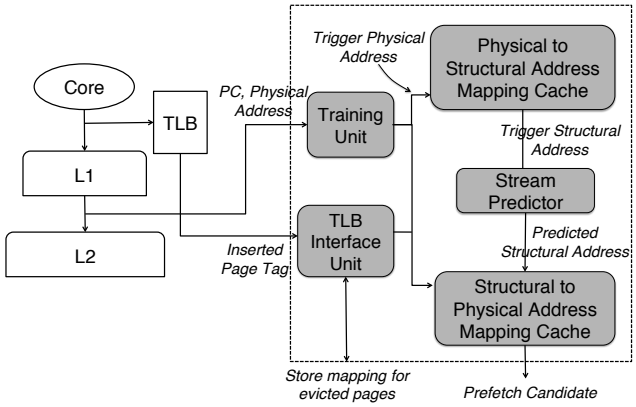

Figure 4.5: Block diagram of the Irregular Stream Buffer.

the physical address space to the structural address space is cached on-chip only for pages that are resident in the TLB, and the prefetcher updates these caches during long latency TLB misses to effectively hide the latency of accessing off-chip meta-data.

4.2.1 ISB Components

The key components of the ISB are shown in Figure 4.5 and are described below.

Training Unit The training unit takes as input the load PC and the load address, and it maintains the last observed address in each PC-localized stream. It learns pairs of correlated physical addresses and maps these to consecutive structural addresses.

Address Mapping Caches (AMCs) The ISB uses two on-chip caches to maintain the mapping between physical and structural addresses. The Physical-to-Structural AMC (PS-AMC) stores the mapping from the physical address space to the structural address space; it is indexed by physical addresses. The Structural-to-Physical AMC (SP-AMC) stores the inverse mapping as the PS-AMC and is

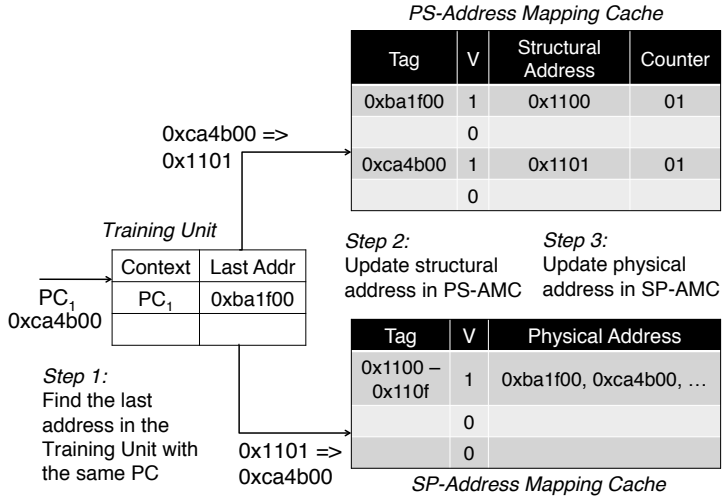

Figure 4.6: ISB training mechanism.

indexed by structural addresses. While the SP-AMC is not strictly necessary, it enables efficient temporal stream prediction because each cache line in the SP-AMC can yield in a single lookup 16 prefetch candidates from the current temporal stream.

Stream Predictor The stream predictor manages streams in the structural address space. It is analogous to stream buffers that are used for prefetching regular memory accesses [45]. Each entry in the stream predictor stores the starting structural address of the temporal stream, a counter to indicate the length of the observed stream, and a counter to indicate the current prefetch look-ahead distance. Like a stream buffer, the stream predictor can be configured for various prefetch degrees and look-ahead distances.

4.2.2 Prefetcher Operation

We now discuss in more detail each of the ISB's three key functions—training, prediction, and TLB eviction.

Training The training process assigns consecutive structural addresses to the correlated physical addresses that are observed by the training unit. When a correlated pair (A,B) is observed, the PS-AMC is queried to see if A and B have previously been assigned structural addresses. If A and B already have consecutive structural addresses, the ISB increments the confidence counter for B's entry in the PS-AMC. If instead A and B have previously been assigned non-consecutive structural addresses, then the confidence in B's mapping is decremented. When the confidence counter hits 0, B is assigned the structural address following A's structural address. If there is no existing mapping for A in the PS-AMC, the ISB generates a new structural address for A and assigns B the subsequent structural address.

Structural addresses are allocated in fixed size chunks of size c to facilitate temporal streams. To keep track of unassigned structural addresses, the ISB maintains a 64-bit counter and increments it by c after every new allocation. Structural addresses are not de-allocated for future reuse, because the 32-bit structural address space is large enough to map 256 GB of physical address space. Fixed size allocation allows every temporal stream to grow up to length c in the structural address space. Temporal streams of length greater than c must request a new allocation in the structural address space for every $(c+1)$th element. Shorter temporal streams, on the other hand, can lead to internal fragmentation of the structural address space. Our experiments show that $c = 256$ is a good choice that supports efficient temporal stream prediction without suffering from excessive internal fragmentation.

As an example of the training process, consider a localized stream as shown in Figure 4.6, where the Training Unit's last observed address is `0xba1f00`, whose structural address is `0x1100`. When the Training Unit receives the physical address `0xca4b00` in the same localized stream, it performs three steps. (1) It assigns `0xca4b00` the structural address following `0xba1f00`'s structural address, namely `0x1101`. (2) It updates the PS-AMC entry indexed by physical address `0xca4b00`,

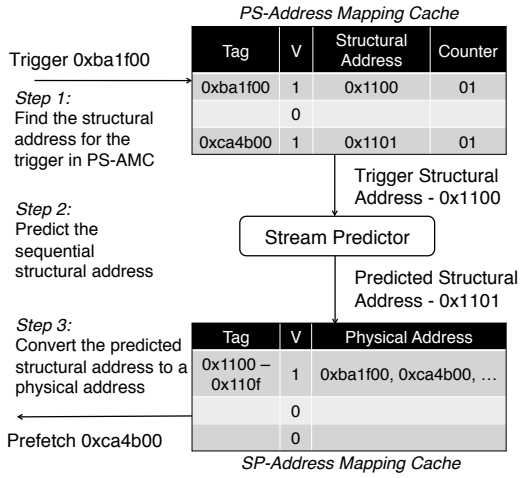

Figure 4.7: ISB prediction mechanism.

and it updates the SP-AMC entry indexed by structural address `0x1101`. (3) It changes the last observed address in the Training Unit to `0xca4b00`.

Prediction One goal of the ISB design is to keep the prediction process (Figure 4.7) as simple as possible. There are three steps. (1) Each L2 cache access becomes a *trigger address* for the prefetcher, causing the PS-AMC to retrieve the trigger address' structural address. In our above example, an access to physical address `0xba1f00` is translated to structural address `0x1100` by the PS-AMC. (2) The Stream Predictor predicts the next consecutive structural addresses to prefetch, which for degree 1 prefetching is `0x1101`. For degree k prefetching, the prediction would include the next k structural addresses, which in this example would be `0x1102`, `0x1103`, `0x1104` and so forth. (3) The SP-AMC retrieves the physical addresses for each of the predicted structural addresses to prefetch. So, `0x1101` is mapped back to `0xca4b00`, and a prefetch request is initiated for this physical address. This mechanism, which consists of two cache lookups, can be used to predict temporal streams efficiently since a single cache line in the SP-AMC contains the translation for 16 consecutive structural addresses.

TLB evictions During a TLB eviction, the ISB writes to DRAM any modified mappings for the evicted page, and it fetches from DRAM the structural mapping for the incoming page. The writeback mechanism invalidates the PS-AMC cache lines corresponding to the evicted page, and it initiates a write to memory if the dirty bit is set. Since the PS-AMC and SP-AMC store redundant information, the contents of the SP-AMC need not be written to memory on an eviction. The fetch mechanism initiates a read request for the structural mapping of the newly inserted page and updates both caches appropriately. Since a TLB miss is a long latency operation involving multiple cache and DRAM accesses, these main memory reads and writes are off the critical path and small enough to not interfere with the core-initiated memory requests. In particular, the ISB is able to overlap its off-chip access with the latency of a TLB miss.

4.2.3 Details of the Address Mapping Caches

To optimize the use of on-chip storage, the ISB uses a compressed representation of the physical/structural addresses in its AMCs. Because the AMCs hold only TLB-resident cache lines, the ISB can use the 7-bit index in the TLB to replace the high order 42 bits of the physical address. The SP-AMC can then store the 13-bit physical address formed by concatenating the 7-bit physical page index and the 6-bit offset in the physical page. Similarly, the PS-AMC can store the 13-bit structural address formed by concatenating the 7-bit structural page index and the 6-bit offset in the structural page. The structural page indices are maintained in a CAM which is updated on a TLB miss or on a new allocation in the structural address space. This compressed representation is used for all internal ISB operations, such as training and prediction. The 13-bit physical address is expanded to the original 64-bit address only when the ISB schedules a prefetch request, and the 13-bit structural page index needs to be expanded only when the off-chip structural

mapping is updated on a TLB eviction.

The PS-AMC and SP-AMC are organized as set-associative caches with 32-byte cache lines. Each cache line in the PS-AMC contains the structural mapping for 16 consecutive physical addresses, with each mapping using 2 bytes to store a 13-bit structural address, a 2-bit confidence counter, and a valid bit. Similarly, each 32-byte cache line in the SP-AMC contains the physical address maps for 16 consecutive structural addresses. If we were to fully provision each cache to map all pages in a 128 entry data TLB, the SP-AMC and PS-AMC would store 8K mapping entries, requiring a total of 32 KB of storage. However, our evaluation shows that in a hybrid setting, provisioning for more than 2K entries has diminishing performance gains and that an 8 KB ISB provides an attractive trade-off between on-chip storage and performance.

4.2.4 Off-chip Storage

To organize the ISB's off-chip meta-data, we use the Predictor Virtualization framework proposed by Burcea at al [7]. In particular, we use a dedicated region of physical memory to maintain the mapping from the physical to the structural address space, which precludes the need for virtual address translation or OS intervention for meta-data accesses.

For our workloads, it suffices to reserve for the ISB 8 MB of off-chip storage. By contrast, the GHB-based prefetchers that we simulate require up to 128 MB of off-chip storage for the same workloads. This discrepancy in off-chip storage arises because the ISB's meta-data grows with the application's memory footprint, whereas the GHB's meta-data is proportional to the number of memory requests made by the application.

Core	Out-of-order, 4 Int/2 Mem/4 FP Func Units, 128-entry ROB, 4-wide dispatch/commit, 80-entry LSQ, 256 physical registers
Front-End	4-wide Fetch, 32-entry Fetch Queue, 4K entry BTB, 1K entry RAS, Hybrid Two-Level Branch Predictor, 128 KB 8-way L1 I-Cache
L1	64 KB 8-way, 2-cycle latency
L2	2 MB 8-way, 18-cycle latency, 64 MSHRs
DTLB	128 entries per core
DRAM	50 ns latency
Two-core	Private L1 cache, 4 MB shared L2 cache
Four-core	Private L1 cache, 8 MB shared L2 cache

Table 4.1: Baseline configuration.

4.3 Evaluation

4.3.1 Methodology

We evaluate the ISB using Marss, a cycle accurate full-system x86 simulator [62], to model single-core, 2-core, and 4-core systems (see Table 4.1 for details). Our simulation infrastructure faithfully models cache queue contention, port conflicts and memory traffic due to prefetch requests. Our TLB simulation allows page entries to be cached in the last-level cache and accurately accounts for the latency of TLB misses. For single-core simulations, we disable timer interrupts. For multi-core simulations, we account for the occasional variation in IPC due to kernel interrupts by taking the median of five runs.

Benchmarks Because we are interested in irregular memory accesses, our evaluation uses the memory-intensive benchmarks from SPECint2006, which generally use irregular pointer-based data structures. We consider a benchmark to be memory-intensive if it has a CPI > 2 and an L2 miss rate $> 50\%$, according to Jaleel's careful

characterization of SPEC2006 [38]. We also use two benchmarks from SPECfp2006, soplex and sphinx3, which contain a mix of both regular and irregular memory accesses. The benchmarks are compiled using gcc-4.2 with the -O2 option. We compile the benchmarks disabling SSE3/4 instructions because our simulator lacks SSE support. All benchmarks are run using the reference input set. We use the SimPoint sampling methodology, generating for each benchmark multiple SimPoints of 250 million instructions to accurately capture all phases of the benchmark. The SimPoints are generated using the SimPoint Tool [63, 30]. We choose a SimPoint length of 250 million instruction because it is large enough to capture long-range behavior, including multiple L2 cache misses on a given address.

Multi-programmed Workloads We simulate multi-programmed workloads by choosing different combinations of our existing benchmarks, simulating two benchmarks at a time on our 2-core configuration and four benchmarks at a time on our 4-core configuration. For each benchmark, we fast-forward to a single SimPoint of 250 million instructions. We then simulate the simultaneous execution of the SimPoint regions for the particular benchmark combinations.

Evaluated Prefetchers In addition to the ISB, we simulate four other prefetchers that target irregular memory accesses.

First, we simulate Idealized STMS, an idealized version of Wenisch, et al's Sampled Temporal Memory Streaming (STMS) prefetcher [92]). Rather than implement all of the STMS optimizations, we simply simulate an idealized G/AC prefetcher,[2] which represents an upper bound on STMS' performance. In particular, the performance of STMS has been shown to approach that of an idealized G/AC prefetcher for long streams [92]. Idealized STMS uses a 64 MB GHB with 8M in-

[2]Using Nesbit and Smith's terminology [59], in which the name before the slash describes the reference scheme and the name after the slash describes the type of correlation that is used, a G/AC prefetcher trains on a Global reference stream and uses Address Correlation.

dex table entries and optimistically assumes that its accesses to the DRAM-resident GHB are free in terms of access latency, DRAM traffic, and memory controller contention. In terms of accuracy and coverage, Idealized STMS primarily differs from STMS in two ways. First, Idealized STMS performs well for short streams, while STMS does not. Second, Idealized STMS trains on the L2 access stream instead of the L2 miss stream.

Second, we simulate an idealized PC/AC prefetcher that represents an upper bound for what any GHB-based prefetcher could achieve, because it uses the combination of PC localization and address correlation. This idealized PC/AC prefetcher is completely unrealistic. In addition to the optimistic assumptions that we make for Idealized STMS, we give PC/AC—when possible—an infinite number of linked list traversals per prediction, which is essential to its speedup. For example, when limited to 10,000 linked list traversals per prediction, coverage falls by 50%. However, for mcf and libquantum, we limit the linked list traversals per prediction to 10,000 to allow our simulations to finish within 3 days.

Third, we simulate Nesbit and Smith's PC/DC prefetcher, which which learns the deltas, or differences, between consecutive memory addresses [59]. Delta correlation allows PC/DC to store all meta-data on chip, so this prefetcher can realistically train on the L2 access stream. We tune PC/DC using all of the optimizations described by Dimitrov and Zhou [21], who submitted the best GHB-based prefetcher in the 2009 Data Prefetching Competition. As with Dimitrov and Zhou's design, our PC/DC prefetcher uses the GHB to exploit delta correlation in both the local and global streams.

Fourth, we simulate the Spatial Memory Streaming (SMS) prefetcher [86], the best known prefetcher that purely exploits spatial locality. The SMS prefetcher realistically trains on the L2 access stream.

We also study the benefit of using irregular prefetchers in conjunction with

regular stride prefetchers. For this study, we use Ishii et al.'s AMPM prefetcher [35], the winner of the 2009 Data Prefetching Competition. AMPM identifies hot zones in memory and stores a bitmap to infer strided patterns in the access stream. AMPM is extremely effective and aggressive because it can detect regular memory accesses independent of the order in which they are observed. We give the AMPM 4 KB of storage and tune it by adjusting its threshold and associativity parameters to produce the best coverage.

For the hybrid experiments, we use an 8 KB ISB, because a 32 KB ISB provides only a small speedup improvement. For the non-hybrid experiments, we use a 32 KB ISB, which contains a 16 KB direct-mapped PS-AMC with 32-byte cache lines, and which uses an 8-way set-associative SP-AMC with 32-byte cache lines.

4.3.2 Single-Core Results

Figure 4.8 compares the speedup, accuracy, and coverage of our five prefetchers on a single core. We see that the two PC/AC-based prefetchers—ISB and idealized PC/AC—achieve significantly better speedup and accuracy than the others. In particular, the speedups over a baseline with no prefetching are 26.9% for idealized PC/AC, 23.1% for ISB, 14.1% for PC/DC, 9.97% for Idealized STMS, and 6.9% for SMS. Idealized PC/AC and ISB also see impressive accuracies of 88.0% and 93.7%, respectively, while the other irregular prefetchers observe less than 65% accuracy on average. These results indicate that PC-localized address correlation is superior to the other techniques—global address correlation (STMS), delta correlation (PC/DC), and spatial footprints (SMS)—for prefetching irregular accesses.

These isb-graphs also show that a practically provisioned ISB approaches the performance of an idealized PC/AC. By contrast, STMS, the previous state-of-the-art in correlation prefetching [92], approaches the performance of idealized

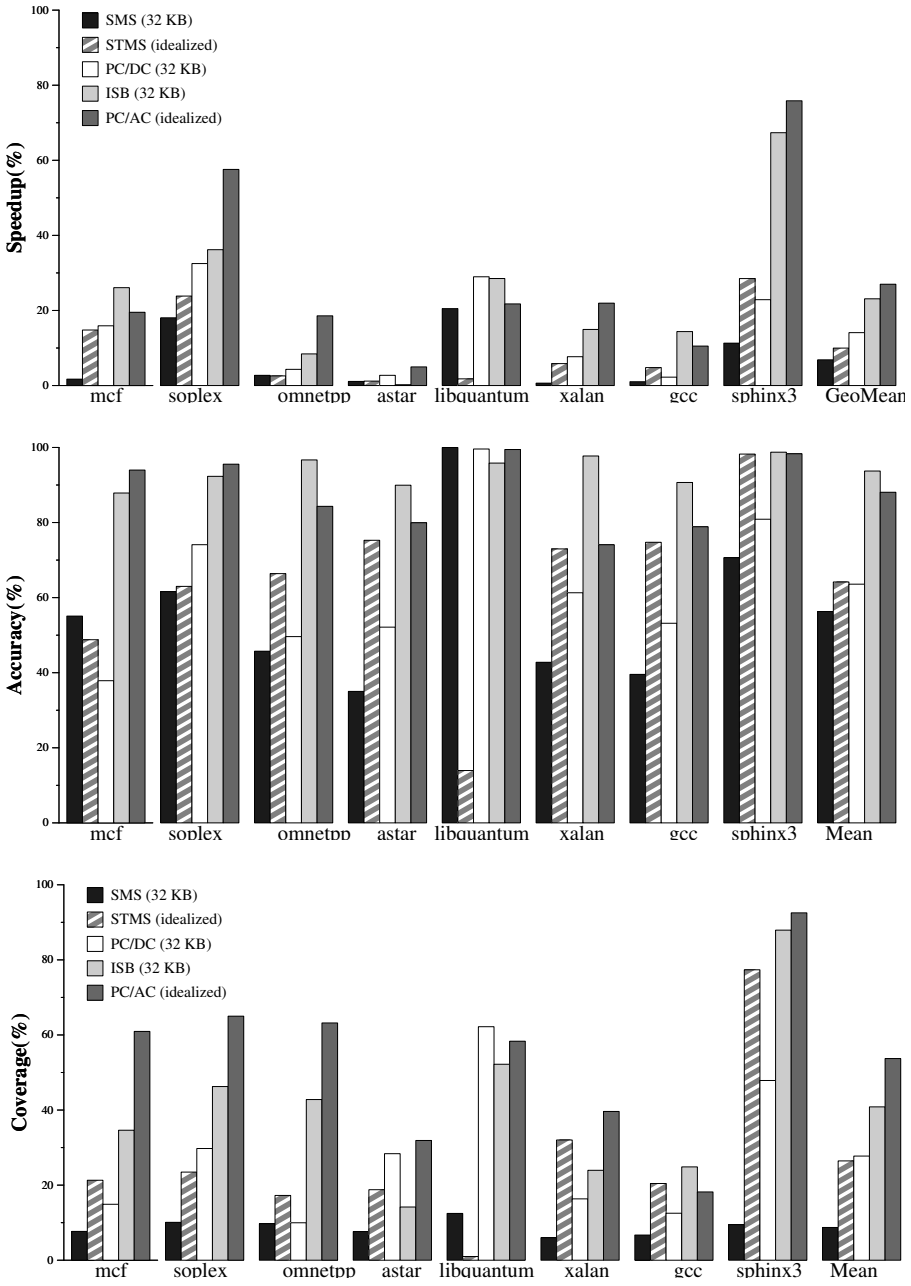

Figure 4.8: Comparison of irregular prefetchers on single core (degree 1)

	mcf	soplex	omnet	astar	libq	xalan	gcc	sphinx	Mean
Useless prefetches	2.8%	5.3%	9.5%	5%	0.05%	5.2%	16%	4.1%	6.3%
Meta-data accesses	5.7%	3.8%	5.7%	12%	1.6%	12.6%	11.3%	3.9%	8.4%

Table 4.2: Memory traffic overhead of the ISB with DDR2.

G/AC. Figure 4.8 shows two anomalies, namely, that the ISB performs better than the idealized PC/AC on mcf and libquantum. Idealized PC/AC performs poorly on these two benchmarks because it is prohibitively expensive to completely idealize the PC/AC prefetcher for these two benchmarks due to their large memory footprint, so for these two benchmarks, we limited the number of linked list iterations to 10,000 and used the largest possible GHB that allowed the simulations to complete in 3 days.

If we make Idealized PC/AC a bit more realistic by letting it train on the L2 miss stream instead of the L2 access stream, its speedup falls to 10.4% and its accuracy falls to 86.3%. Similarly, if Idealized STMS trains on the L2 miss stream, its speedup falls to 8.3% and its accuracy to 58.6%.

Finally, we note that the ISB sees speedup of just 2.3% on the remaining SPEC FP benchmarks, because the ISB cannot predict compulsory misses, whereas many stride prefetchers can. The ISB does not slow down any of the benchmarks.

4.3.3 Memory Traffic Overhead

The ISB's memory traffic overhead approaches that of prefetchers, such as SMS and PC/DC. that store all of their meta-data on chip. In particular, the ISB incurs an average of 14.7% memory traffic overhead, while Dimitrov and Zhou's PC/DC prefetcher [21] incurs 12.6% overhead and SMS just 10.5% overhead. The highly accurate ISB incurs just 6.3% overhead due to useless prefetches. The ISB accesses off-chip meta-data only during a TLB miss, reading at most 256 bytes of mapping

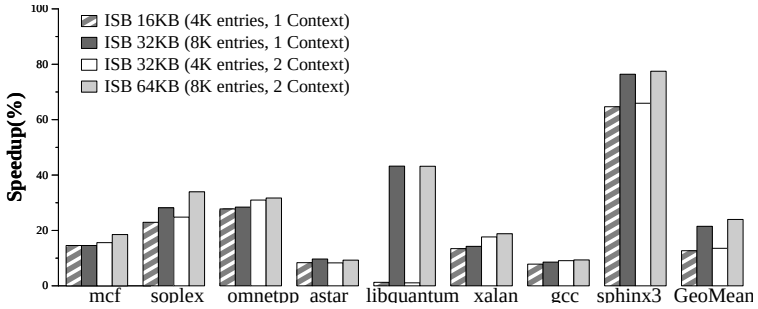

Figure 4.9: Design Space Exploration of ISB.

information per page. Assuming a bus width of 64-bytes with DDR2, this information can be fetched from DRAM in four requests. Since not all cache lines in a page are necessarily mapped, the actual traffic per page can vary from one to four requests. As seen in Table 4.2, the ISB's access to off-chip correlation data increases memory traffic by an average of 8.4%. With DDR3's 128-byte bus width, the traffic would be reduced to 4.2% because the information for the entire page could be fetched in a single DRAM request. By contrast, the STMS prefetcher incurs about 35% overhead due to meta-data access [92].

4.3.4 Design Space Exploration of ISB

We explore the ISB design space by varying three key parameters: the width of the training counter, the number of contexts stored per memory address, and the size of the Address Mapping Cache.

To explore the impact of counter width on ISB's performance, we vary the counter width from 2-8 bits and also consider a 32-bit counter, which approximates an infinitely long history. We find that the performance is largely insensitive to the width of the training counter, but smaller counters are 26% more accurate than 32-bit counters. Smaller counters are more accurate because they are more conservative in issuing prefetches for unstable/unpredictable streams. Therefore, we use a counter width of 2 bits.

Next, we vary ISB's storage budget by varying the number of entries in the Address Mapping Cache and the number of *contexts* per cache entry. Larger contexts enable ISB to map each physical address to more than one structural address when the physical address occurs with multiple load instructions. For example, a context of 2 implies that a physical address can be mapped to at most two structural addresses if it occurs with two or more different load instructions. For our study, we consider four configurations: 16 KB budget (two sub-caches each with 4K entries and 1 context), 32 KB budget (two sub-caches each with 8K entries, 1 context), 32 KB budget (two sub-caches each with 4K entries, 2 contexts), and 64 KB budget (two sub-caches each with 8K entries, 2 contexts). Figure 4.9 shows the speedup of the four configurations over a baseline with no prefetching. We see that reducing the number of entries has a much larger impact on performance than reducing the number of contexts per entry. Therefore, we choose a 32 KB cache with 8K entries and 1 context because it provides a good tradeoff between size and performance.

4.3.5 Degree Evaluation

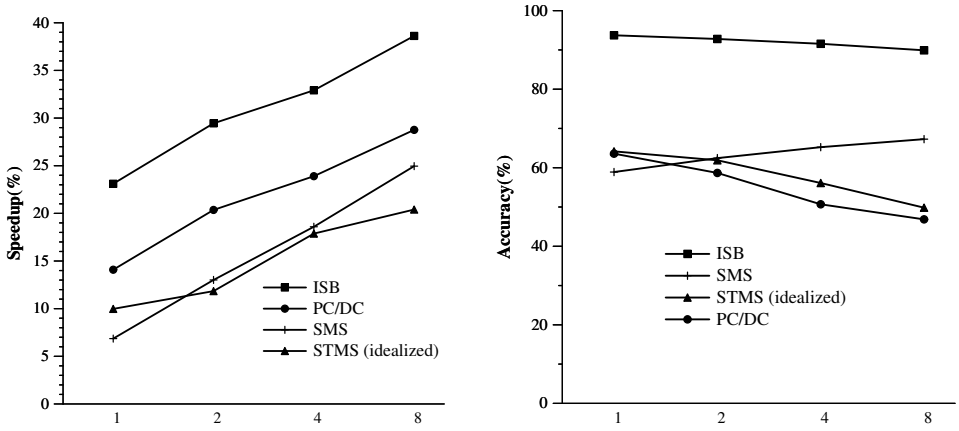

Figure 4.10: Impact of prefetch degree on speedup and accuracy.

Figure 4.10 shows how the speedup and accuracy of four prefetchers—ISB,

PC/DC SMS, and Idealized STMS—vary as the prefetch degree is increased from 1 to 8, revealing several trends:

- The ISB performs well as the degree increases: With degree 8, its speedup rises from 23.1% to 38.6%, and its accuracy decreases by just 3.8%.
- PC/DC has the most severe tradeoff between speedup and accuracy: With degree 8, its speedup almost doubles to 28.8%, but its accuracy falls down to 46.8%, which is the worst among all prefetchers.
- By contrast, the SMS prefetcher has the best tradeoff between speedup and accuracy, as it improves in both speedup *and* accuracy as the degree is increased, indicating that prefetches from higher density spatial regions are more accurate, but even for degree 8, the ISB exhibits significantly better speedup and accuracy than SMS.
- Finally, except at degree 1, Idealized STMS has the worst performance of all of the prefetchers, and its accuracy curve closely matches that of PC/DC.

4.3.6 Hybrid Design with AMPM

Vendors that implement an irregular prefetcher will undoubtedly also implement a regular prefetcher, so we now consider hybrid designs that combine an irregular prefetcher with an AMPM stride prefetcher. Here, we only consider the three practical prefetchers, namely, ISB, PC/DC, and SMS.

When combined with a regular prefetcher, the ISB is much less sensitive to the size of the AMC. As a result, in a hybrid setting with AMPM, an ISB with 8 KB of storage sees speedup of 40.8%, whereas an ISB with 32 KB of storage sees an additional speedup of only 6.3%. This behavior can be understood by observing that in our workloads, phases of regular and irregular accesses see little overlap and that the ISB requires large on-chip memory to prefetch long regular streams. In a

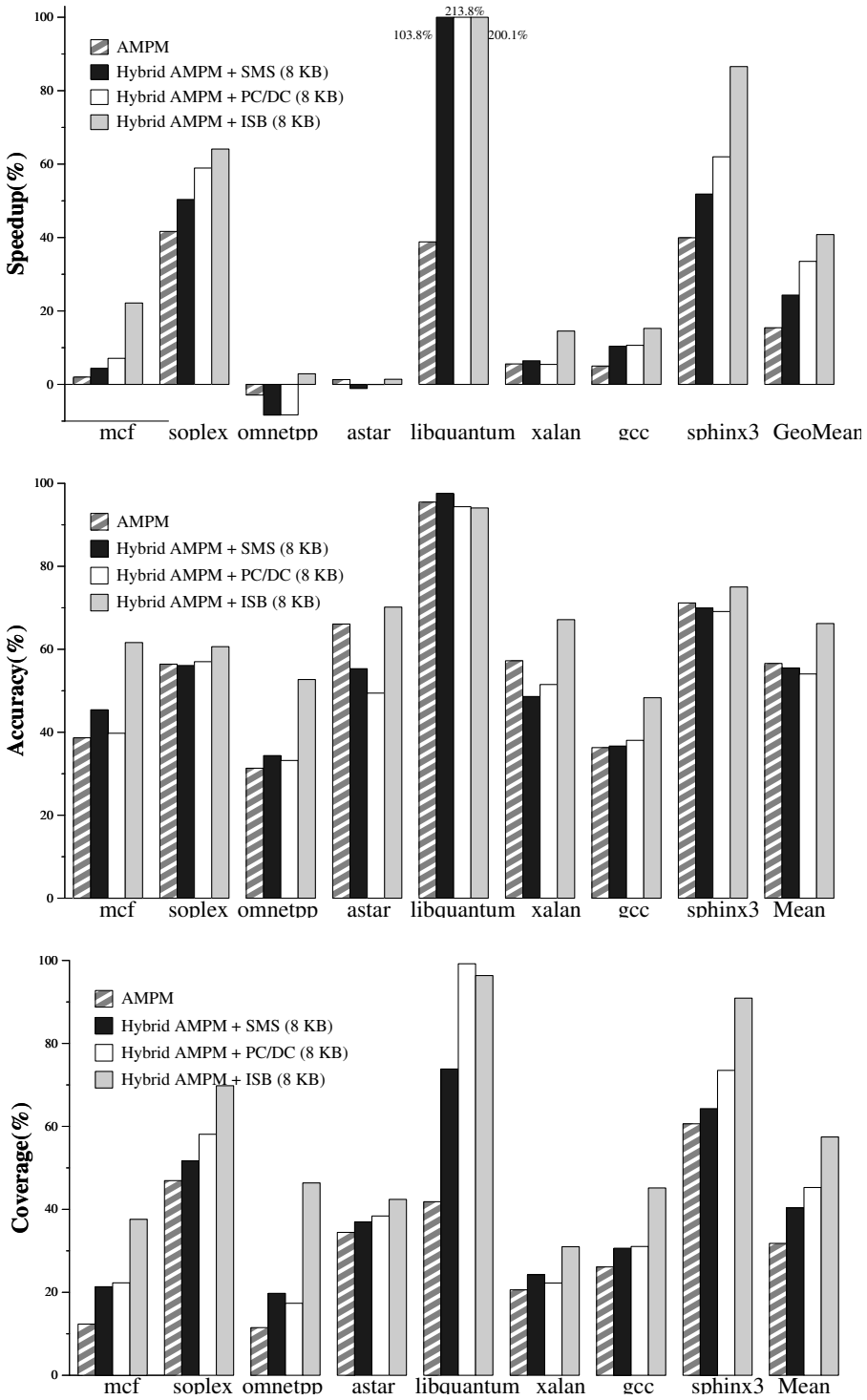

Figure 4.11: Comparison of hybrid prefetchers

hybrid setting, 8 KB is sufficient for the ISB to prefetch the irregular phases, while AMPM can prefetch the regular phases.

Figure 4.11 compares AMPM against hybrid prefetchers that combine AMPM with an 8 KB ISB, an 8K SMS, and an 8K PC/DC, respectively. The AMPM + SMS hybrid achieves a 24.3% speedup over a baseline with no prefetching, the AMPM + PC/DC achieves a 33.5% speedup, while AMPM alone achieves 15.4% speedup. The AMPM + ISB hybrid achieves a speedup of 40.8% over a baseline with no prefetching, which is an improvement of 25.4% over AMPM. The coverage graph shows that SMS achieves just 4.5% coverage and PC/DC only 9.4% additional coverage over AMPM, while ISB achieves an extra 21.6% coverage over AMPM. A closer inspection of Figure 4.11 indicates several other key points.

1. For libquantum, the AMPM + PC/DC hybrid outperforms the AMPM + ISB hybrid because the ISB is not capable of prefetching cold misses, while PC/DC is.

2. The three benchmarks that contain both regular and irregular accesses—soplex, sphinx, and gcc—see good speedups over AMPM with all hybrids.

3. For four of the benchmarks—mcf, omnetpp, astar, and xalan—only the AMPM + ISB hybrid achieves a significant improvement over AMPM. These benchmarks are dominated by pointer-based accesses to a graph, a graph, a tree, and a tree, respectively. This indicates that delta correlation and spatial footprints are not very effective for irregular accesses. Moreover, poor coverage combined with poor accuracy causes the AMPM + SMS hybrid and the AMPM + PC/DC hybrid to slow down omnetpp and astar.

4. The AMPM + ISB hybrid has the highest accuracy among the hybrids at 66.2%. This accuracy is significantly lower than the ISB's accuracy of 93.7% because of AMPM's poor accuracy of 56.6%. For chips with a larger number

of cores, a less aggressive stride prefetcher than AMPM would probably be wise.

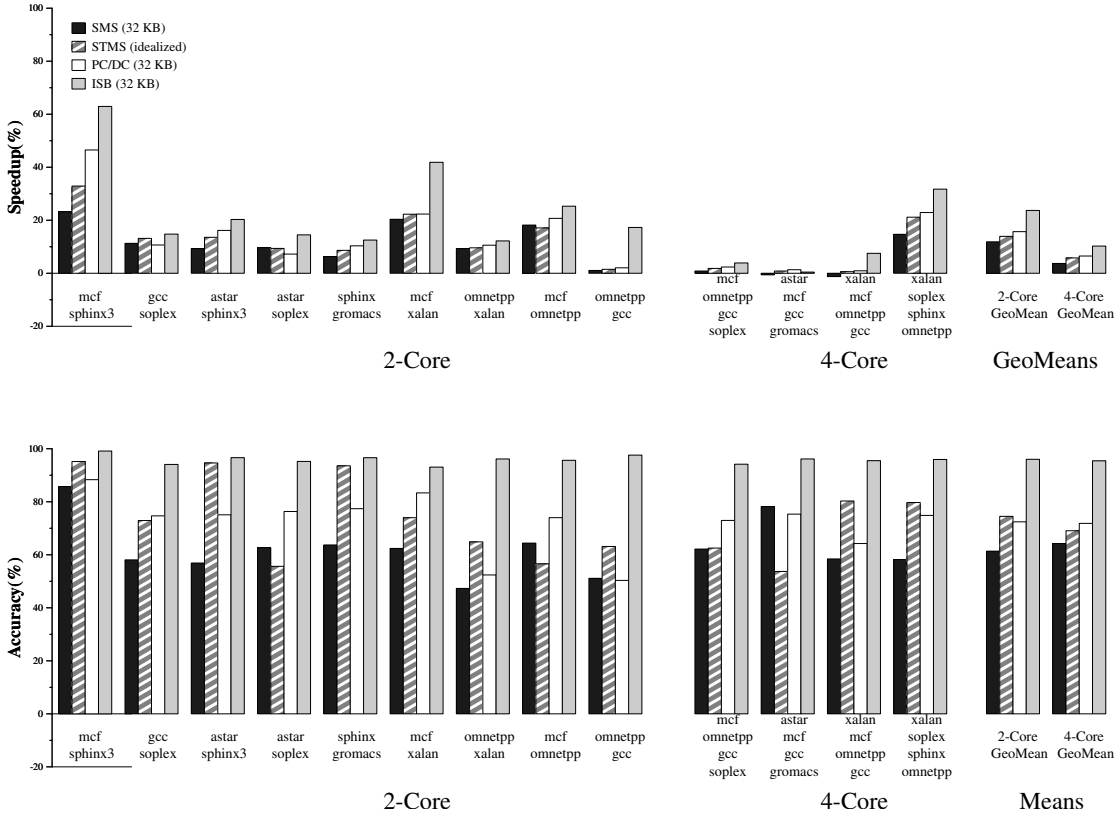

Figure 4.12: Comparison of irregular prefetchers on 2-core (left) and 4-core (right) systems.

4.3.7 Multi-Core Results

Figure 4.12 compares the ISB with SMS, PC/DC, and Idealized STMS on a multi-core system using multi-programmed workloads as described in Section 4.3.1. We see that the ISB outperforms the three prefetchers on both the 2-core and 4-core machines. On the 2-core machine, the ISB sees a speedup of 23.69%, whereas SMS, PC/DC and Idealized STMS see average speedups of 11.9%, 13.9% and 15.7%, re-

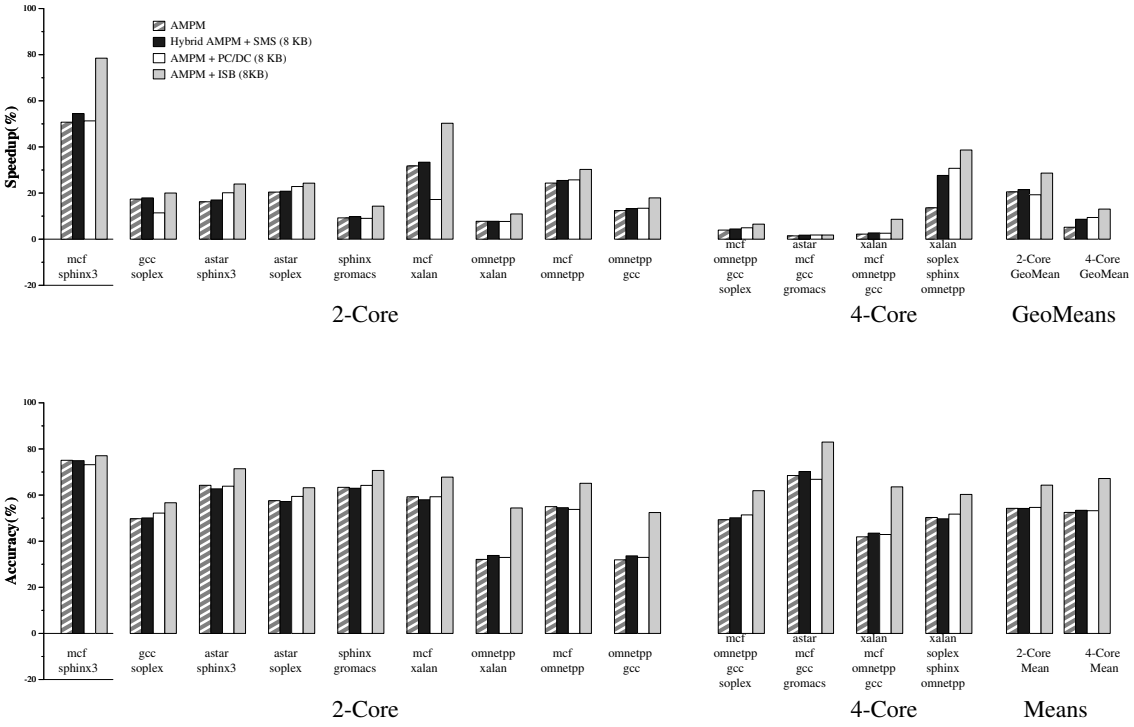

Figure 4.13: Comparison of hybrid prefetchers on 2-core (left) and 4-core (right) systems.

spectively. The average speedup for all prefetchers is lower on the 4-core machine, with the ISB observing a 10.3% speedup, and with SMS, PC/DC, and Idealized STMS achieving 3.7%, 5.8% and 6.5% speedup, respectively. The ISB's accuracy is consistently above 95% for both configurations, which makes it attractive in a multi-core setting, since useless prefetches increase both memory traffic and cache pollution. As the number of cores increase, prefetching accuracy can have a significant bearing on system performance. For example, Ebrahimi, et al. show that in a multi-core environment with 4 cores, any prefetcher whose accuracy is below 40% needs to be throttled down to preserve overall system performance [25].

Figure 4.13 evaluates hybrid prefetchers by combining AMPM with the ISB, SMS and PC/DC. In a hybrid setting, only the ISB is able to significantly outperform

AMPM on both machines, which supports our claim that the ISB is more effective at irregular prefetching than PC/DC and SMS. The AMPM + ISB hybrid observes speedups of 28.6% and 13% on 2-core and 4-core machines, respectively, which is much less than the 40.8% speedup that it achieved on the single-core machine. This decline can be attributed to AMPM, which generates considerable useless traffic due to its poor accuracy.

4.3.8 Power Evaluation

While training on the L2 reference stream provides significant coverage and accuracy benefits, its increased activity increases the prefetcher's power consumption. We thus evaluate the power and energy consumption of the ISB by comparing them against that of the GHB-based PC/DC prefetcher that trains on the L2 miss stream. This discussion will not consider the power impact of useless prefetches on cache and memory subsystem behavior. We use CACTI [80] to estimate the energy consumed by the prefetching hardware per read/write operation, and we then multiply that cost by the activity counters of the prefetching hardware. We find that the ISB consumes 0.77 times the energy of PC/DC but 1.07 times the power. The increase in average power consumption can be attributed to the faster execution time with the ISB. The ISB generates more activity by training on the L2 access stream, but uses a simple training and prediction logic. By contrast, PC/DC consumes far more energy per input due to its linked list traversals through the GHB, so for the same energy budget, the ISB is able to use localization and exploit the information available in the entire L2 access stream with minimal power overhead.

4.4 Summary

In this chapter, we have introduced the Irregular Stream Buffer, which represents a significant milestone in the long quest to build prefetchers for irregular memory

accesses: The ISB is the first practical prefetcher that combines address correlation with PC localization. While the previous state-of-the-art in temporal stream prefetching, STMS, approaches the behavior of an idealized G/AC prefetcher for long streams, the ISB approaches the superior coverage and accuracy of an idealized PC/AC prefetcher for all streams.

The key idea behind the ISB is an extra level of indirection that translates correlated physical addresses to consecutive addresses in a new structural address space. Thus, in the structural address space, the elements of a temporal stream appear in sequential order, which greatly simplifies prediction.

The structural address space provides three important benefits.

1. It allows the ISB to manage meta-data efficiently by caching TLB-resident meta-data on chip and synchronizing the contents of this cache with TLB misses. The result is just 8.4% memory traffic overhead for accessing off-chip meta-data, significantly lower than the overheads reported for other address correlation-based prefetchers, such as STMS [92], which itself represented an order of magnitude improvement over its predecessors [16].

2. It improves coverage and accuracy by supporting the combination of PC localization and address correlation. For example, on a single core, an idealized PC/AC prefetcher obtains 26.9% average speedup and 88% accuracy, compared with 14.1% speedup and 65% accuracy for PC/DC; an idealized G/AC prefetcher (ie, Idealized STMS) sees 9.97% speedup and 65% accuracy.

3. Our caching scheme improves coverage and accuracy by allowing the ISB to train on the LLC reference stream instead of the LLC miss stream, which in our experiments more than doubles the observed speedup. For example, the idealized PC/AC prefetcher sees 26.9% speedup when trained the L2 access stream, as opposed to just 10.4% when trained on the L2 miss stream.

Looking to the future, we plan to evaluate the ISB on commercial workloads. We expect that the ISB will perform well on these workloads, because unlike the GHB, the ISB's on-chip storage and memory traffic overhead depend only on the size of the TLB, not the application's memory footprint. To extend the ISB's benefits to TLBs with large pages, including superpages, we plan to explore a two-level ISB design that can synchronize with pages of any size without undermining the ISB's small on-chip budget. We also plan to evaluate the use of ISB as the temporal component of spatial-temporal prefetchers similar to Somogyi, et al.'s STeMS prefetcher [85]. More broadly, we believe that the use of a linearized structural address space can be used to drive other micro-architectural optimizations for irregular programs.

Chapter 5

Aggregate Stride Prefetching

In 1990, Jouppi [45] introduced the notion of a stream buffer to hide the long latencies of DRAM accesses. The idea was to identify and prefetch sequences of memory references whose successive addresses differ by some constant stride. Since then, researchers have taken two different approaches to increasing the coverage of stride-based prefetchers.

One approach completely relaxes temporal order by exploiting spatial locality [35]. The idea is to keep a spatial footprint—a bitmap—which is then insensitive to order. Thus, the set of memory references 1, 2, 3 produces the same footprint as 3, 2, 1. This idea was cleverly used by Ishii, et al in the AMPM prefetcher to identify strides from possibly complex patterns of memory usage [35]. Unfortunately, AMPM loses temporal information, which restricts its ability to learn complex delta patterns. Moreover, because each footprint is associated with a particular region of memory, training time can be slow as AMPM cannot learn patterns from other regions.

The second approach generalizes the notion of a stream to include patterns of strides [59, 79]. For example, the reference stream 1, 2, 5, 6, 9, 10, ... consists of alternating strides of 1 and 3. The recently proposed Variable Length Delta

Prefetcher (VLDP) [79] uses different amounts of context to learn these complex stride patterns. In particular, it uses three prediction tables that use stride history lengths of 1, 2, and 3, respectively. So the above reference stream would be detected by the second table. Unfortunately, VLDP is highly sensitive to the sequence of memory references, and it suffers from poor timeliness as it can only learn strides for a few subsequent accesses.

In this thesis, we present a third approach: Instead of learning a completely spatial pattern (no temporal order) or a precise complex pattern (complete temporal order), our goal is to learn the *aggregate stride* of a possibly complex stride pattern, where the aggregate stride is the sum of the strides in a stride pattern. Thus, when our Aggregate Stride Prefetcher (ASP) sees the reference stream 1, 2, 5, 6, 9, 10, whose stride pattern is 1,3,1,3, it will learn the aggregate stride of 4. Because this reference stream can be viewed as two interleaved reference streams that each has a stride length of 4, the aggregate stride can be used for both of the sequences, as shown below:

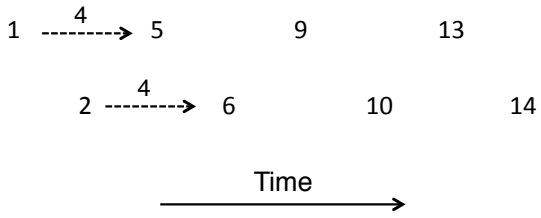

Figure 5.1: An aggregate stride of 4 can learn the stride pattern 1,3,1,3.

The benefit of aggregate strides may be somewhat counter-intuitive. For example, for patterns that consist of multiple strides, say k strides, the aggregate stride will miss the first $k-1$ prefetches in a stream. However, aggregate strides are much easier to learn than complex stride patterns. To see this point, consider the following three stride patterns:

```
Pattern 1:  3, 2, 2

Pattern 2:  2, 3, 2

Pattern 3:  2, 2, 3
```

While ASP learns the same aggregate stride of 7 for all three patterns, PC/DC and VLDP instead try to learn each of these three stride patterns separately and then need to learn when to apply each of these three stride patterns. But there is little benefit to solving this more difficult prediction problem because the aggregate stride will work for all three streams.

Moreover, because addition is commutative, the aggregate stride is less sensitive to perturbations in the stride pattern that can arise from from out-of-order execution. At the same time, the simpler learning goal allows ASP to examine a much longer history to find strides that will both be accurate and timely. For example, we evaluate an instance of ASP that uses a history of 256 past memory references, as opposed to VLDP, which keeps a history of at most three recent strides, and we show how ASP's performance decreases as its history length decreases. Finally, as with VLDP, ASP's use of temporal information allows it to learn patterns from other regions, which speeds up training time.

This chapter makes the following contributions.

- We introduce the notion of aggregate stride prefetching, which is a simpler form of delta correlation, and we design a prefetcher that detects aggregate strides.
- We evaluate the Aggregate Stride Prefetcher using a set of memory-intensive SPEC2006 benchmarks and show that ASP significantly outperforms both AMPM and VLDP on single core and multi-core configurations. On single core systems, ASP with a prefetch degree of 1 improves performance by 93%(vs.

AMPM's 50% and VLDP's 38.8%). With a prefetch degree of 4, ASP's performance benefit increases to 102.3%(vs. AMPM's 90.4% and VLDP's 87.6%). On a 4-core system, ASP improves performance by 64%(vs. AMPM's 43% and VLDP's 56%) while incurring 13% less traffic.

- We show that ASP's superior performance is a result of its ability to leverage a long history of memoy references to learn complex stride patterns and issue timely and accurate prefetches.

- We provide insights as to why aggregate stride prefetching is more effective than Delta Correlation and spatial-based prefetchers, and we show that both these classes of prefetchers have fundamental limitations which ASP does not suffer from.

5.1 Design Goals

Before we describe our solution, we discuss our design goals and their importance for effective prefetching.

5.1.1 Complex Patterns

Our first goal is to capture a wide range of regular access patterns, including constant strides and recurring delta patterns. Section 5.4 presents a detailed analysis of different access patterns and characterizes the ability of different prefetchers to learn those patterns.

5.1.2 Reordering

Memory accesses can be re-ordered due to out-of-order scheduling by the core or due to reordering by the cache controller. Thus, our second design goal is to be

robust to perturbations of the memory access sequence. Section 5.4 illustrates the impact of reordering on different prefetchers with an example.

5.1.3 Fast Training

Prefetchers usually observe a few memory accesses to learn the dominant access pattern, but region-based prefetchers such as AMPM [35] incur a training penalty at the start of every new region and are unable to apply what they've learned from one spatial region to another. Thus, our third design goal is to train our prefetch quickly by leveraging the access patterns that are being learned globally.

5.1.4 Timeliness

Finally, the benefit of prefetchers can be improved by ensuring that they are timely and arrive before the demand request. One way to improve timeliness is to learn larger strides. In particular, AMPM tends to choose larger strides than VLDP, which helps improve timeliness. For example, for the stride pattern 1,3,1,3, AMPM learns a stride of 4, while VLDP will learn alternating strides of 1 and 3; the timeliness could potentially be further improved by choosing a stride of 8.

Long strides can hide more latency, but they also present two issues. First, they can lead to lower coverage. In the previous example, by prefetching strides of 8, we miss prefetches to the first few elements of the stream, i.e., A, A+1, A+4 and A+5. Second, it can reorder memory requests such that prefetch requests that are used later are requested before critical demand loads. For example, Figure 5.2 shows that with a stride of 8, the prefetches to A+8, A+9, A+12 will be requested and possibly serviced before the demand requests to A+1, A+4 and A+5, thereby increasing the latency of these demand accesses. However, as Figure 5.2 also shows, once the initial cost of large strides is covered, the timeliness benefits of larger strides continue for the duration of the stream. Therefore, long streams are better able to

overcome the performance loss due to this reordering of memory requests, since they can amortize the *start-up cost* over many timely prefetches.

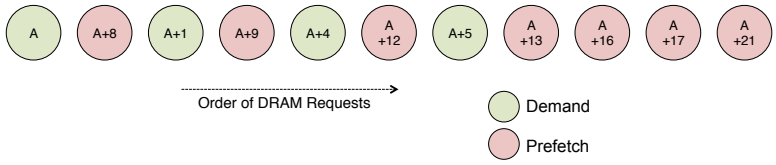

Figure 5.2: Large strides can reorder memory requests and delay demand requests.

Thus, our final goal is to design a timely prefetcher that balances the stream start-up cost with timeliness benefits over the length of the stream.

We now describe ASP and explain how it fulfills all of our three design goals.

5.2 Our Solution

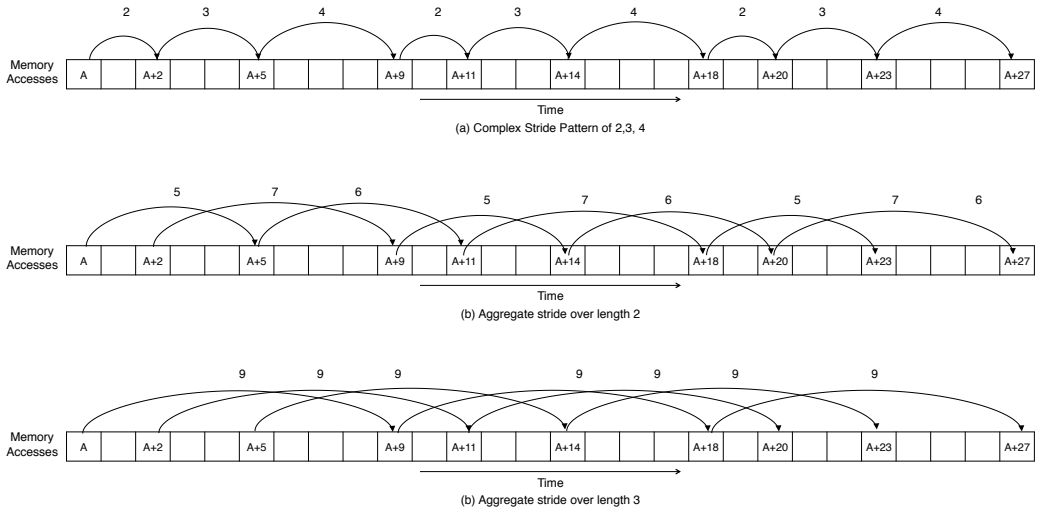

Figure 5.3: The same memory access pattern can be learned with a complex stride pattern or with an aggregate stride.

To learn complex stride patterns, the ASP looks for aggregate strides that

tend to repeat. For example, Figure 5.3(a) shows a memory access pattern with repeated stride pattern of (2, 3, 4), and Figure 5.3(b) shows that the same access pattern can be learned with an aggregate stride of 9 (sum of 2, 3 and 4) that repeats across all accesses.

Learning aggregate strides is a relatively simpler task than learning precise delta patterns, and in this section, we will show how this simpler view provides several benefits over delta prefetchers including robustness to reordering of memory accesses and ease of learning long stride patterns. Moreover, we will show that ASP also enables new optimizations that greatly improve the timeliness of prefetch requests.

We start by describing our algorithm for finding the best aggregate stride and follow that with a new proposal to improve the timeliness of prefetch requests. Finally, we present a detailed micro-architectural design of the Aggregate Stride Prefetcher.

5.2.1 Computing Aggregate Strides

The idea of finding aggregate strides is conceptually simple, but the main challenge lies in finding the aggregate stride without knowing the length over which strides must be aggregated. Aggregating strides over an incorrect length will result in both poor coverage and accuracy. For example, in Figure 5.3, the correct length for stride aggregation is 3. Aggregating strides over a length of 1 (Figure 5.3(a)) will result in strides of 2, 3 and 4, and aggregating strides over a length of 2 (Figures 5.3(b)) will result in strides of 5, 6 and 7. Each of those strides occur for only one in three memory accesses and will result in 33% coverage and accuracy.

Our solution builds on the insight that the correct aggregate stride S will occur more commonly than all other aggregate strides because every memory reference will participate in a stride of S. Figure 5.3(c) shows that aggregating strides

over the correct length of 3 results in a single stride of 9 that repeats for all memory accesses. Using a stride of 9, every memory reference in Figure 5.3 can be prefetched with 100% coverage and accuracy. This insight can be generalized to stride patterns of any length, including length 1 (constant stride accesses).

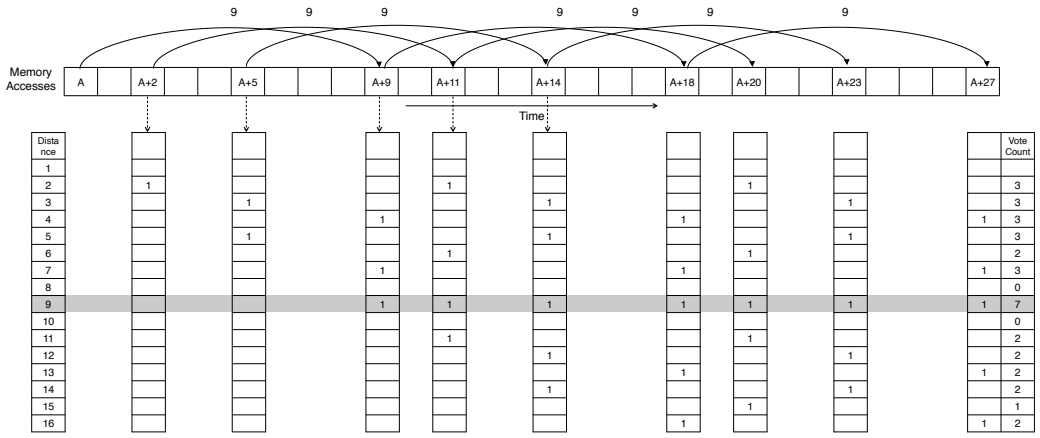

Figure 5.4: ASP uses a voting-based algorithm to find the most common aggregate stride.

To build on this insight, the ASP computes all possible aggregate strides within a certain window of time, which we call the *temporal window* and uses a voting-based confirmation algorithm to arrive at the most frequently occurring aggregate stride. Figure 5.4 illustrates this algorithm for the example in Figure 5.3. Each column in the table in Figure 5.4 represents for the memory access at the top of the column all possible distances to previous accesses, and each row represents a potential aggregate stride; a table entry is marked as 1 if the memory access in the column has seen a stride corresponding to the row. For example, the column corresponding to A+5 has 1's for strides 3 and 5 because it has a stride of 3 with A+2 and and a stride of 5 with A. The last column in the table sums up the *votes* for each stride, where the stride 9 is a clear winner.

To compute strides that populate the table, we consider all accesses that

are within the temporal window, which in our case varies from 4 to 32. While computing 32 strides for every memory access is seemingly expensive, realize that the column values for a new access can be derived by shifting the column values for the previous access. For example, in Figure 5.4, the column for A+36 can be derived from the column corresponding to A+32 by shifting the 1s down by 4 (stride between A+32 and A+36). As a result, computing all strides within the temporal window is inexpensive.

This voting-based algorithm gives ASP several advantages. ASP issues prefetch requests in the order of the access stream, and it can learn aggregate stride patterns from other pages because we take a global vote. Additionally, the ASP has two benefits: (1) It is insensitive to small perturbations in the memory access stream because the correct aggregate stride will always collect the maximum number of votes irrespective of the order in which the references arrive within the temporal window considered; (2) It can find delta patterns of any length by expanding the temporal window with a linear growth in hardware cost. Finally, ASP is able to leverage a long history of memory accesses to find strides that are both accurate and timely.

5.2.2 Improving Timeliness

Higher coverage is one method of improving performance, but improving the prefetcher timeliness also improves performance by hiding more memory latency. To improve timeliness, the prefetched stream should be sufficiently ahead of the access stream so that the prefetch request can hide all the off-chip memory latency. The gap between the prefetched stream and the access stream, also known as the *prefetch distance*, can be chosen statically at design time depending on the overall memory latency.

One way to achieve the desired prefetch distance is to uniformly increase

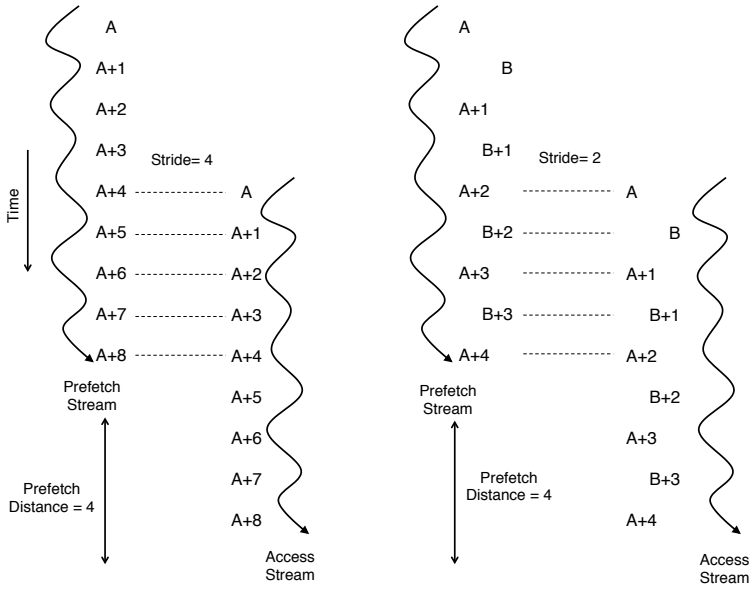

Figure 5.5: The choice of strides for timely prefetching depends on stream interleaving.

stride length of all prefetch streams by the desired prefetch distance. [1] For example, the left side of Figure 5.5 shows that to achieve a prefetch distance of 4, we can increase the stride of a sequential stream (stride 1) to 4; this would result in prefetching A+4 with a trigger access of A, A+5 with a trigger access if A+1, and so on. One problem with increasing stride lengths uniformly for all streams is that it does not account for the effects of interleaving streams. For example, the right side of Figure 5.5 shows that when two sequential streams (A, A+1,...,A+8) and (B, B+1,...,B+8) are interleaved, the stride length should be increased to 2 to realize an overall prefetch distance of 4. Thus, to find the best stride that maintains the desired prefetch distance and maximizes the timeliness of all prefetch streams, we need a global view of stream interactions.

ASP addresses this problem by incorporating the desired prefetch distance

[1] The notion of increasing stream lookahead is equivalent to increasing stride length.

in its stride learning algorithm. In particular, ASP re-positions its learning window such that the entire prefetch stream (not individual streams) is ahead of the access stream by the desired prefetch distance. Figure 5.6 shows how we can pick appropriate strides by moving ASP's temporal window by the desired prefetch distance. In particular, Figure 5.6(a) shows that when there is a single sequential stream, moving the temporal window by the desired prefetch distance of 4 results in predicting a stride of 4 which is equivalent to predicting a stride of 1 with a lookahead of 4. But Figure 5.6(b) shows that when streams are interleaved, moving the temporal window by 4 allows ASP to pick a stride of 2. Thus, the aggregate strides learned by ASP are more timely because they account for the effects of interleaving streams.

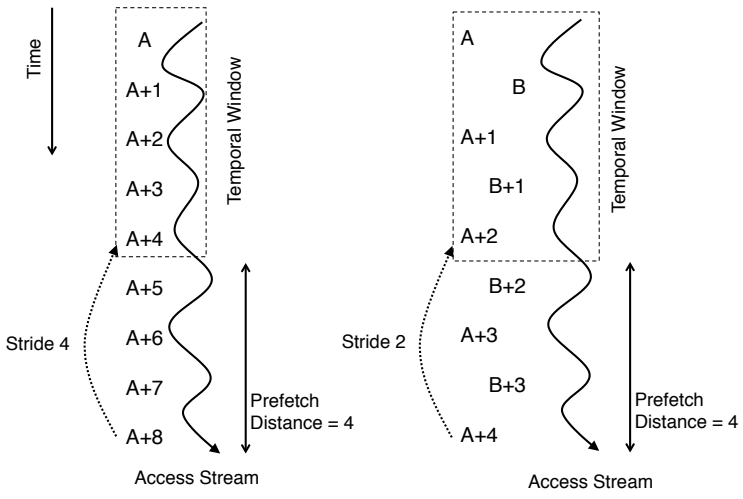

Figure 5.6: Prefetch distance can be maintained by choosing strides from the appropriate temporal window.

5.2.3 Detailed Design and Operation

Figure 5.7 shows the overall design of the ASP. A fixed size *history buffer* (size 256 in our design) represents ASP's temporal window and holds the most recent cache

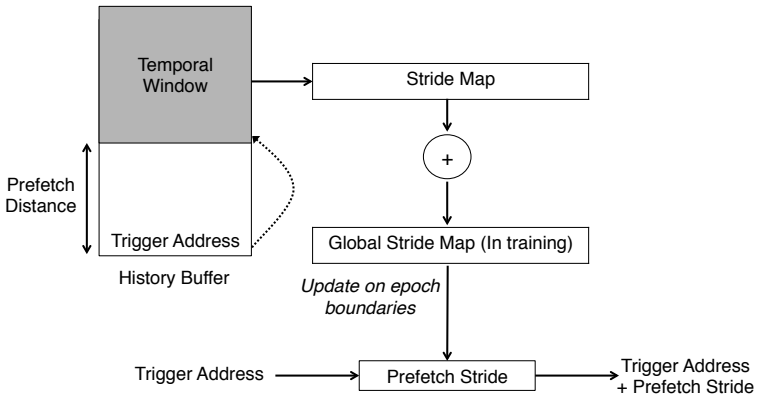

Figure 5.7: Hardware Design for ASP.

line accesses. Each entry in this buffer consists of a 16-bit cache tag and a *stride map*, which is a list of all strides between the cache line and all previous accesses in its temporal window. The stride map is represented as a 48-bit bitmap with a bit marked as 1 if that particular stride has been observed (the first 16 bits represent negative strides from -16 to -1, and the latter 32 bits represent positive strides from 1 to 32). The history buffer is organized as an 8-way cache with 64 sets. Each set represents a sequence of consecutive entries in the history buffer, with consecutive sets representing different time chunks.

Training. When a new cache access X occurs, ASP chooses the most recent access Y in the temporal window with which it can compute an appropriate stride (between -16 and 32 in our case). We find that the the neighbor can usually be located within the 16 most recent accesses in the temporal window, which requires at most 2 lookups in the 8-way history cache. The stride map for this new access is computed by shifting the stride map of Y by $X - Y$. The entry for X is then added to the history buffer with its newly constructed stride map. For prefetch distances greater than 1, the search for the most recent access is restricted to the temporal window as shown in Figure 5.7.

Observing a shifted temporal window has benefits for timeliness, but it can result in slow learning at the start of new streams. In fact, for short streams, this technique may result in signficant coverage loss. Therefore, if ASP is unable to find any spatial neighbor in its shifted temporal window, we pick the closest neighbor from the most recent entries in the history buffer.

Prediction. For prediction, we would like to sum all the stride maps in the history buffer to find the most popular stride. However, for simplicity, we maintain a *Global Stride Map* which aggregates the stride maps of all access within a fixed epoch (2048 cache accesses in our design). Updating the Global Stride Map is simple: before writing the stride map of X in the history buffer, we also perform a bitwise addition between X's stride map and the Global Stride Map. When an epoch finishes, the trained Global Stride Map is used to pick the most popular stride, which is then used for prefetching in the subsequent epoch.

5.3 Evaluation

5.3.1 Methodology

We evaluate ASP using the Sniper multi-core simulator [9], modeling a 4-wide out-of-order processor with an 8-stage pipeline, a 128-entry reorder buffer and a three-level cache hierarchy. The parameters for our simulated memory hierarchy are shown in Table 5.1. Sniper is unable to provide reordered memory references to the prefetcher (reordering effects are limitied to timing simulation) so our results do not account for re-ordering effects. Finally, our memory controller makes no distinction between prefetch and demand requests.

L1 D-Cache	128 KB 4-way, 1-cycle latency
L2 Cache	256KB 8-way, 10-cycle latency
Last-level Cache	2MB, 16-way, 18-cycle latency
DRAM	58.8ns latency, 12.8GB/s bandwidth
Two-core	4MB shared LLC (22-cycle latency)
Four-core	8MB shared LLC (26-cycle latency)

Table 5.1: Baseline configuration.

Benchmarks. We evaluate ASP on the all memory-intensive SPEC2006 benchmarks.[2] We consider a benchmark to be memory-intensive if it has a CPI > 2 and an L2 miss rate $> 50\%$, according to Jaleel's careful characterization of SPEC2006 [38]. All benchmarks are compiled using gcc-4.2 with the -O2 option and are run using the reference input set. We use SimPoint [63, 30] to generate for each benchmark as many as 20 samples of 250 million instructions.

Multi-Core Workloads. Our multi-core results simulate either two benchmarks running on 2 cores or four benchmarks running on 4 cores, choosing 100 random combinations of the 20 most memory-intensive SPEC2006 benchmarks. For each combination, we simulate the simultaneous execution of the a single SimPoint sample of the constituent benchmarks until each benchmark has executed at least 250M instructions. If a benchmark finishes early, it continues to run until every other application in the mix has finished running 250M instructions. Thus, all the benchmarks in the mix run simultaneously throughout the sampled execution.

Evaluated Prefetchers. We compare ASP against two state-of-the-art regular prefetchers, namely, AMPM [35] and VLDP [79]. For both prefetchers, we perform extensive tuning to find the parameters that maximize their performance. The final

[2]We currently cannot run perl and bzip2 on our platform.

Prefetcher	Parameters	Budget
AMPM	52 access maps; 16KB region	4 KB
VLDP	3 DPTs, 1 OPT with 64 entries	1 KB
ASP	256-entry History Buffer, 48 strides	2 KB

Table 5.2: Prefetcher Configuration.

configuration for all prefetchers is listed in Table 5.2. For ASP, we use a 256-entry history buffer and a prefetch distance of 8.

All prefetchers are allowed to cross page boundaries because previous work [64, 22] has shown that 50-70% of consecutive virtual pages are allocated contiguously in the physical address space. This contiguity allows for greater opportunity to prefetch but results in useless prefetches to unmapped pages.

5.3.2 Comparison with Other Prefetchers

Figure 5.8(a) shows that ASP significantly outperforms both AMPM and VLDP with degree 1 prefetching. In particular, ASP achieves a speedup of 93.3% while AMPM and VLDP see speedups of 48.9% and 38.5%. If we exclude lbm and libquantum (to avoid skewing the averages), AMPM sees a speedup of 43.9%, VLDP sees a speedup of 45.3% and ASP a speedup of 74.3%. For lbm, VLDP sees negative speedup even though it has 18% coverage because prefetches for lbm are not timely and result in significant backpressure on the L1 cache.

ASP's advantage stem from coverage and timeliness.

Coverage. Figure 5.8(b) shows that ASP and VLDP have the same coverage, but a closer look reveals that ASP tends to match the coverage of the best of AMPM or VLDP for individual benchmarks. For example, for cactus, gems, zeusmp and leslie, ASP matches the coverage of VLDP, and for bwaves, lbm, soplex and xalancbmk, ASP matches the superior coverage of AMPM. For milc and mcf, ASP gets better

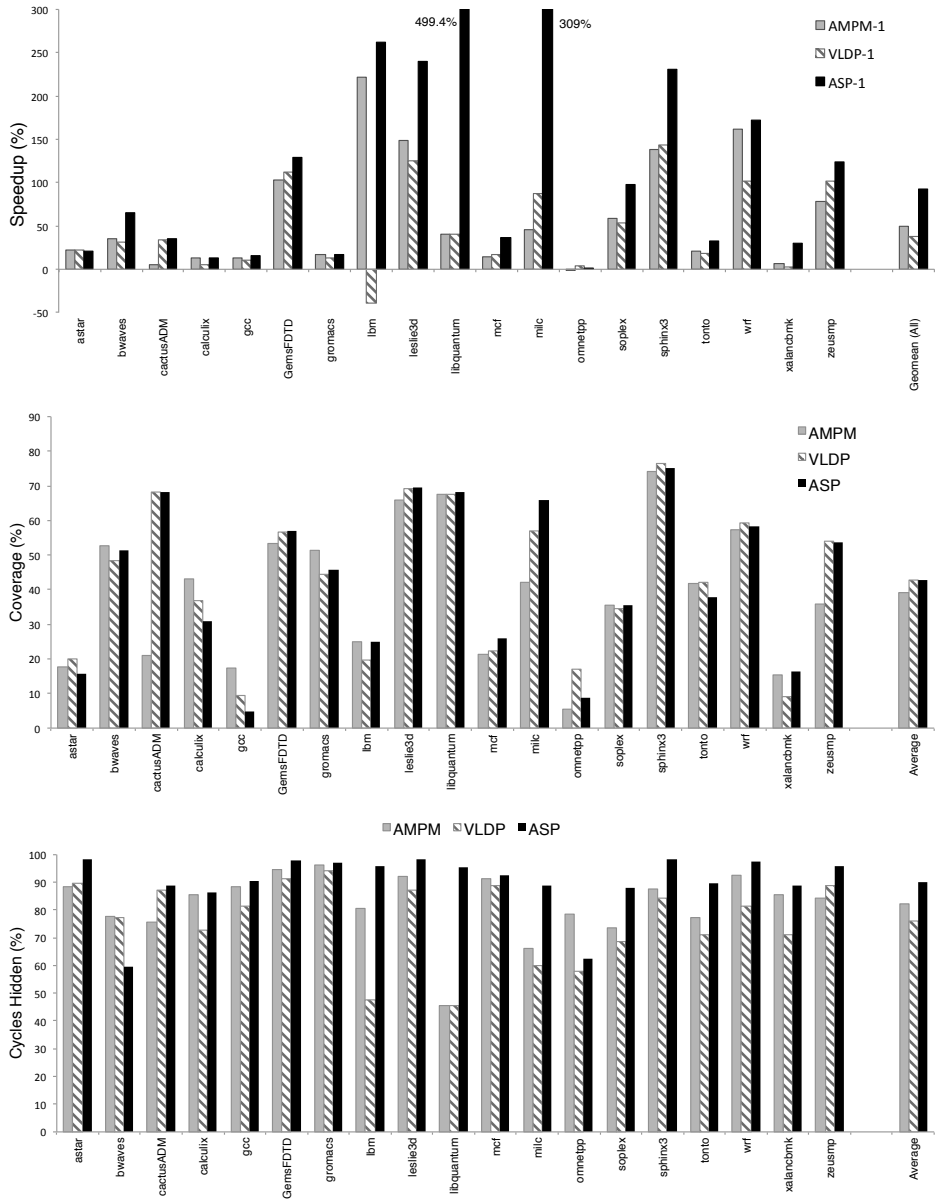

Figure 5.8: Comparison between AMPM, VLDP and ASP with degree 1 prefetching.

coverage than both AMPM and VLDP. AMPM has a coverage advantage over VLDP because (1) it can learn delta patterns of any length, and (2) it does not miss prefetching the first two deltas in every page. Finally, we also observe that ASP gets lower coverage than AMPM and VLDP for benchmarks such as astar, calculix, gcc and omnetpp because these benchmarks are dominated by short streams, which do not favor ASP's tendency to pick long strides.

Timeliness. The biggest gains for ASP come because it hides more latency for each prefetch as shown in Figure 5.8(c). On an average, ASP hides 90% of the memory latency for all successful prefetches, while prefetches from AMPM and VLDP hide only 82% and 76% of the memory latency. ASP's huge performance wins for milc and libquantum can be explained by the fact that ASP's prefetches hide 88.6% and 95% of latency for these two benchmarks, while the other two prefetchers hide less than 66% latency for both of these benchmarks.

Reordered Memory Accesses. Due to limitations of our simulation infrastructure, prefetchers can only observe memory references in program order (the timing simulation is out-of-order, and it is decoupled from the functional aspects), so these results do not account for effects of memory reordering. However, as we show in Section 5.4, we expect ASP to be robust to memory reordering and see larger relative wins when re-ordered memory references are simulated.

Harmful Effects. Table 5.3 quantifies the overhead of each prefetcher. VLDP and ASP both increase memory traffic by 3-4% for regular workloads and 26% for irregular workloads. AMPM is more inaccurate for regular workloads but it significantly more accurate on irregular workloads.

Finally, as explained in Section 5.1, improved timeliness can also delay the service of critical demand loads, so we find that ASP increases the service time for demand loads by 22.5%, which is almost twice the delay introduced by both AMPM

	AMPM	VLDP	ASP
Memory Traffic (Regular)	6.6%	3.2%	3.7%
Memory Traffic (Irregular)	14.5%	26.4%	26.2%
Increase in latency of demand loads	10.3%	9.4%	22.5%

Table 5.3: Overhead of AMPM, VLDP and ASP.

and VLDP. However, ASP's additional delay for demand loads is compensated by the effectiveness of its prefetch requests.

5.3.3 Higher Degree Prefetching

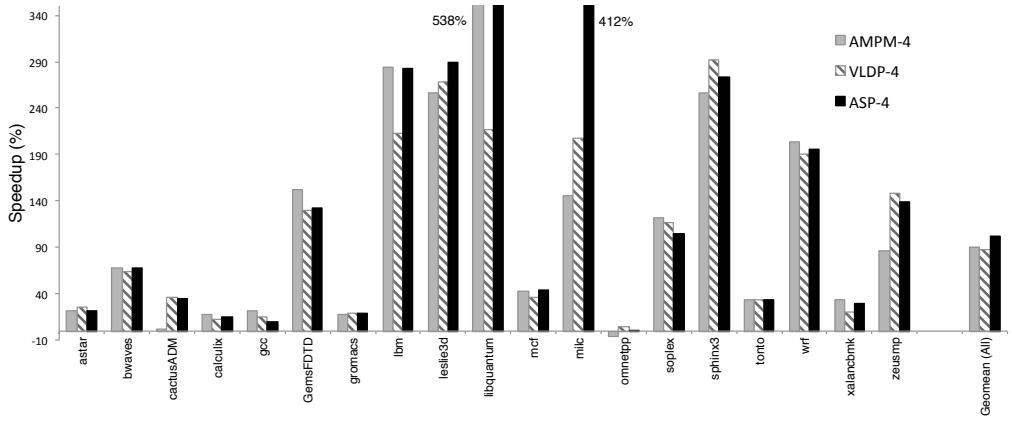

Figure 5.9: Comparison between AMPM, VLDP and ASP with degree 4 prefetching.

Increasing prefetch degree can also improve coverage and timeliness for most prefetchers, so we compare AMPM, VLDP and ASP at higher degrees. Figure 5.9 shows that ASP outperforms both AMPM and VLDP even at a prefetch degree of 4, but the gap between ASP and others is smaller in comparison with degree 1 prefetching. In particular, ASP achieves a speedup of 102.4%, while AMPM and

VLDP achieve speedups of 90.4% and 87.66%.

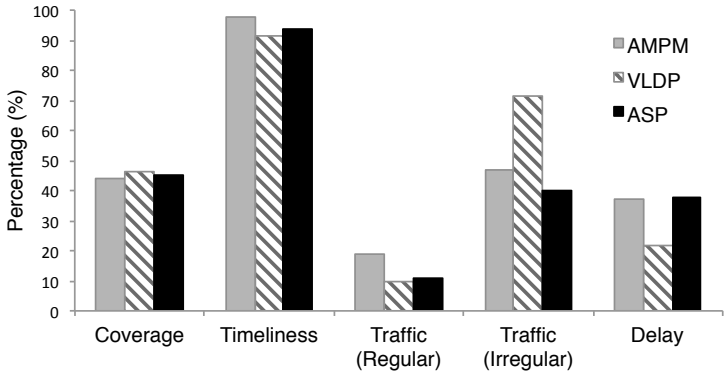

Figure 5.10: Coverage, Timeliness and Overhead with degree 4 prefetching.

Figure 5.10 shows that at higher degree, the coverage for all three prefetchers is roughly similar and all prefetchers hide more than 90% of the memory latency. VLDP continues to see relatively lower timeliness than both AMPM and ASP because even at higher degree, it predicts sequences of short strides.

The most interesting difference among the prefetchers at higher degree is their traffic overhead. For regular benchmarks, AMPM incurs a 19% traffic overhead while VLDP and ASP result in less than 10% extra traffic, because of their higher accuracy in predicting complex delta patterns. For irregular benchmarks, all three prefetchers issue many more useless prefetch requests, but VLDP's traffic overhead is significantly higher than both AMPM and ASP(71.6% vs. 46.8% and 39.9%). In summary, at higher degree, ASP wins due to its timeliness and accuracy benefits, which are critical in bandwidth-constrained environments.

5.3.4 Lookahead Prefetching

Another way of improving timeliness without increasing degree and traffic is to improve prefetch lookahead, that is, the gap between the trigger access and the

prefetched address. We find that AMPM performs best with a lookahead of 8, and VLDP performs best with a lookahead of 4. In fact, VLDP's inability to look farther ahead than 4 memory accesses is a fundamental limitation of its table-based design.

With a lookahead of 8, AMPM gets 69% speedup, and with a lookahead of 4, VLDP gets 63% speedup. Both of these numbers are still significantly less than ASP's 93% speedup. These results demonstrate that ASP's timeliness scheme, which accounts for stream interleaving, is a superior to simply increasing lookahead.

5.3.5 Sensitivity To History Length

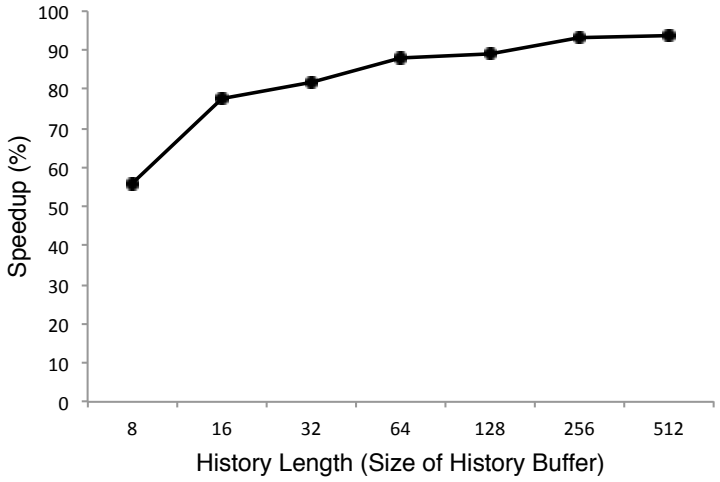

Figure 5.11: ASP's sensitivity to history length.

Figure 5.11 shows how ASP's performance varies with size of its history buffer. A longer history allows ASP to look for longer patterns and find strides that are beneficial for longer prefetch distances. We observe that ASP's performance benefit decreases slowly from 93% to 88% as we decrease the history length from 512 to 64 and declines sharply to 56% once the history length is below 16. These results indicate that prefetchers such as VLDP are limited due to their inability to

consider long histories.

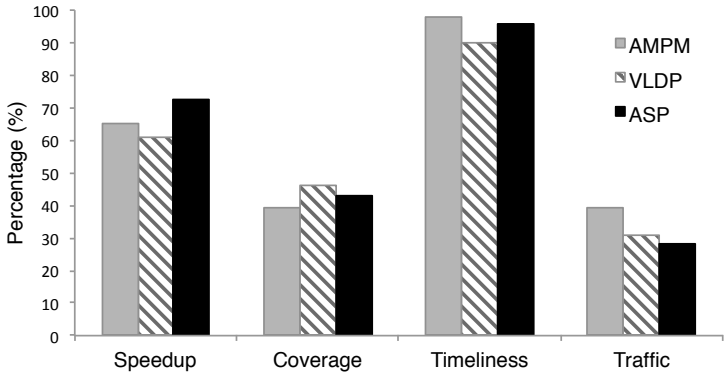

Figure 5.12: Comparison on 2 cores(degree 4).

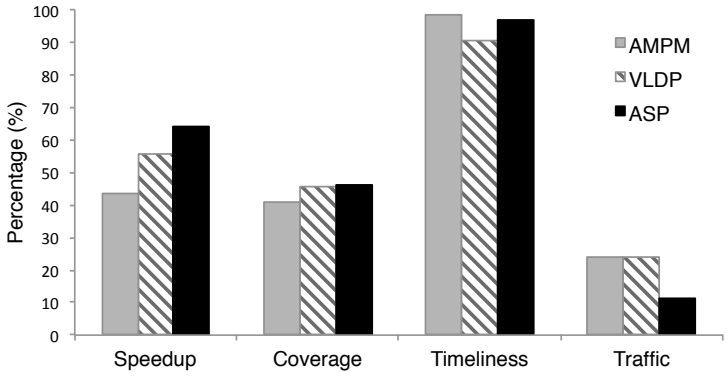

Figure 5.13: Comparison on 4 cores(degree 4).

5.3.6 Multi-Core Workloads

Figure 5.12 and Figure 5.13 compare the three prefetchers on multi-programmed workloads running on 2 and 4 cores respectively. ASP outperforms AMPM and VLDP on both 2-core and 4-core mixes. In particular, on a 2-core system ASP achieves speedup of 72.3% (vs. 65% for AMPM and 60.7% for VLDP), and on a

4-core system, ASP achieves a speedup of 64% (vs. 43.3% for AMPM and 56% for VLDP).

ASP sees superior performance because it combines high coverage and timeliness with low traffic. On both 2-core and 4-core systems, ASP is the only prefetcher to have both high coverage and high timeliness. By contrast, AMPM tends to have high timeliness but low coverage, while VLDP tends to have high coverage but low timeliness. ASP also has the lowest traffic overhead on both configurations, which is critical to its performance in a bandwidth-constrained environment.

5.4 Discussion

This section uses three examples to explain why aggregate strides can capture a wider class of delta patterns than AMPM and VLDP.

Example 1: Comparison with AMPM We first show that AMPM can, in fact, learn some recurring delta patterns. We then show a case that AMPM cannot learn.

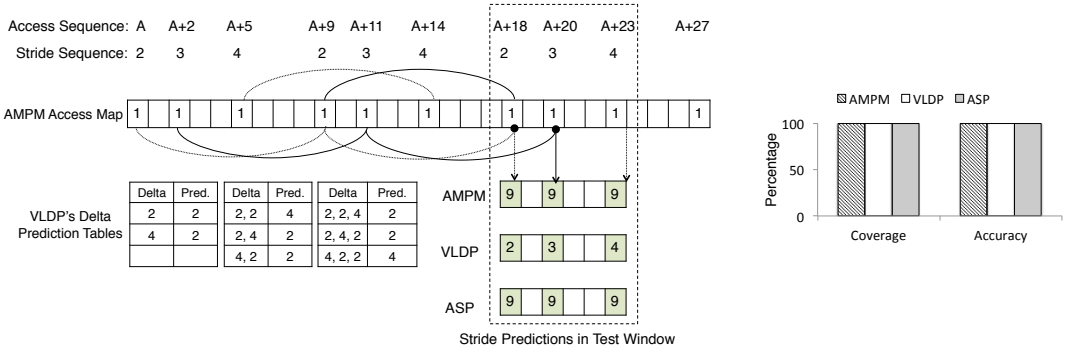

Figure 5.14: Example 1 with delta pattern 2, 3, 4: AMPM, VLDP and ASP achieve 100% coverage and 100% accuracy. The arrows represent the strides detected by AMPM.

Figure 5.14 shows an access sequence with a delta pattern of 2, 3, 4. The

top row in the figure shows the access stream, and the second row shows the corresponding stride sequence. To help understand AMPM and VLDP's decisions, we also show the access map and delta prediction tables learned by AMPM and VLDP, respectively. We show the operation of all prefetchers in a *test window* that includes one iteration of the delta prefetcher after the initial training phase; the behavior of the test window is guaranteed to repeat in the steady state. The stride predictions for all three prefetchers in the test window are shown in the bottom right, and the measured coverage and accuracy from an actual execution of a microbenchmark with the stride pattern is shown on the right side of the figure.

For the stride pattern in Figure 5.14, AMPM consistently predicts a stride of 9 because it cannot find two consecutive matches for strides 2, 3 or 4. By contrast, VLDP learns the precise sequence of strides, and it is able to generate correct predictions using the first delta prediction table. Finally, ASP chooses the most common aggregate stride, which in this case is 9. All prefetchers achieve 100% coverage and 100% accuracy.

This example illustrates two points: (1) the aggregate stride of 9 is as effective as the precise delta pattern of 2,3,4; (2) AMPM is not limited to constant stride patterns, and it can learn some complex delta patterns by finding the aggregate stride.

Example 2: Comparison with AMPM We use an example to illustrate the conditions under which AMPM is unable to learn delta patterns. Figure 5.15 shows the behavior of the three prefetchers with a delta pattern of 2, 2, 4. AMPM alternates between strides of 4, 8 and 2 because it looks for the smallest stride that repeats in its spatial bitmap (the arrows show how these strides are inferred from AMPM's access map). For the test window, these predictions result in prefetches of A+20, A+26 and A+22. Of these predictions, only A+20 and A+26 are correct, which yields 66% accuracy. Moreover, AMPM is unable to predict A+24 which

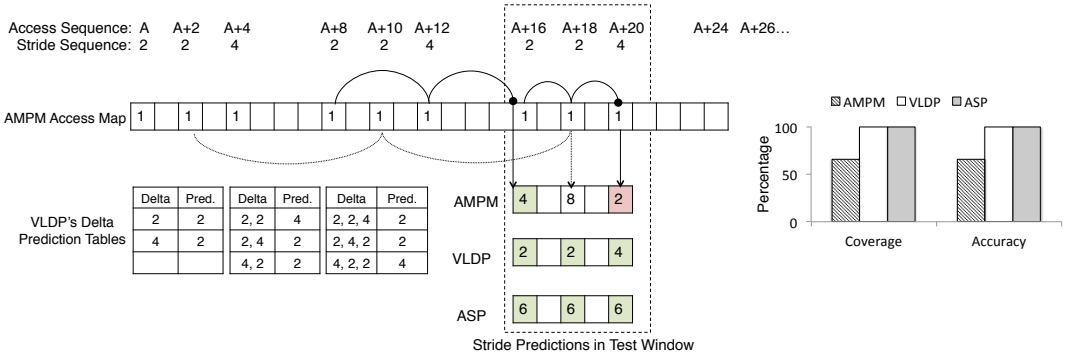

Figure 5.15: Example 2 with delta pattern 2, 2, 4: AMPM oscillates between strides of 4, 8 and 2.

yields 66% coverage (A+18 would have been predicted with a stride of 8 on seeing A+10, just like A+18 results in predicting A+26).

As expected, VLDP learns the precise delta pattern 2, 2, 4, and in the steady state, it is able to predict all accesses with 100% accuracy. ASP learns an aggregate stride of 8 (sum of 2,2 and 4) and is also able to achieve 100% coverage and accuracy with this aggregate stride.

This example shows that AMPM's ability to find the aggregate stride for delta patterns is limited. In particular, AMPM finds the aggregate stride only when the aggregate strides over all subsequences of the delta pattern are different. For example, in the stride pattern of (2,2,4), the sum of subsequence (2,2) is equal to the third stride 4, which causes AMPM to infer a stride of 4 (which produces a correct prediction but not the correct aggregate stride). Similarly, because the first two elements in the stride sequence (2,2,4) are the same, AMPM infers a stride of 2, which is incorrect.

Example 3: Comparison with VLDP Memory accesses can be re-arranged at various levels of the memory hierarchy. Unfortunately, memory re-ordering can hurt VLDP because it relies on precise sequences for accurate learning.

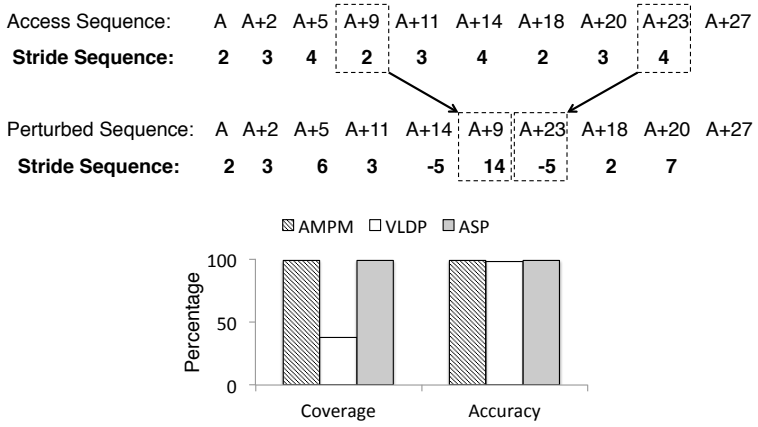

Figure 5.16: Example 3 with pattern 2, 3, 4 with perturbations: VLDP loses 60% coverage.

Figure 5.16 shows a perturbed sequence for the example in Figure 5.14, such that only one element in every iteration of the delta sequence arrives out-of-order. Below the perturbed access sequence, we also show the strides that VLDP is likely to train on when it observes this sequences of memory references.

The first thing to note in Figure 5.16 is that even small perturbations in the access stream can result in the stride sequence becoming highly unpredictable.

Second, the graph in Figure 5.16 shows that while AMPM and ASP continue to see high coverage and accuracy with the perturbation, VLDP sees a significant reduction in its coverage. This result clearly illustrates that Order-Based Prefetchers are fundamentally not robust to any reordering in the memory system.

5.5 Summary

In this chapter, we have presented the Aggregate Stride Prefetcher (ASP), which accommodates complex stride patterns, not by learning them precisely but by learning their aggregate stride. To provide insight into ASP's success, we have compared ASP with various aspects of two state-of-the-art prefetchers, AMPM and VLDP.

We find that ASP offers many benefits.

- Like AMPM, ASP is robust to the reordering of memory requests, which might occur with out-of-order execution or optimizing cache controllers. While AMPM's robustness comes from learning spatial patterns, ASP's robustness comes from the commutativity of addition. By contrast, VLDP is designed to learn precise patterns, so it is sensitive to re-ordering by design.

- Like VLDP, ASP can issue accurate prefetches for workloads that have complex stride patterns, which again comes from its simplified learning goal. AMPM, on the other hand, is less successful at learning complex stride patterns.

- Like VLDP, ASP can learn globally, which allows it to train quickly. By contrast, AMPM's spatial patterns are tied to specific regions of memory, so it cannot learn from the behavior of accesses to other regions of memory.

- Finally, ASP provides an approach to improving timeliness that is unique to its approach and that is superior to both AMPM and VLDP.

Our evaluation has shown the somewhat surprising result that long prefetch distances can be beneficial, even if they sometimes delay demand requests. The keys to success are to have an accurate prefetcher and to have sufficiently long prefetch streams that the small number of delayed demand misses are outweighed by the improved timeliness of a large number of prefetches.

	AMPM	VLDP	ASP
Complex Patterns		✓	✓
Robust to Memory Reordering	✓		✓
Global Learning		✓	✓
Timeliness	Medium	Low	High

Table 5.4: Summary of ASP's benefits.

Chapter 6

Conclusions

In this thesis, we have presented solutions that significantly advance the state-of-the-art for three memory system optimizations, namely, cache replacement, irregular prefetching and regular prefetching. A common theme in all our solutions is the pursuit of ambitious learning goals that are enabled by the effective use of long-term information.

For cache replacement, we have introduced the Hawkeye replacement policy that uses a long history of past memory references to learn and mimic Belady's optimal solution. For irregular prefetching, we have introduced the Irregular Stream Buffer, which is the first prefetcher to combine PC-localization and address correlation as it uses a novel meta-data organization to efficiently capture long-term repetition. Finally, we have introduced the Aggregate Stride Prefetcher, which uses long-term information to pursue the simple learning goal of finding aggregate strides to produce accurate and timely prefetches.

To fully exploit long histories, we have introduced novel history representations that facilitate efficient management of long histories and also reveal useful information about the program's execution. For example, we have introduced the notion of liveness intervals for cache replacement, which allows us to model over-

lapping demand for cache accesses and compute Belady's optimal solution for past references. The idea of representing demand contention as overlapping liveness intervals has implications for other resource management problems, such as, resource partitioning in multi-core environments. Second, we have introduced the structural address space, which linearizes irregular memory accesses to reveal repetitive sequences of memory accesses. Because structural addresses translate temporal correlation to spatial locality, they provide an easy abstraction to understand and exploit repetition in irregular programs.

Looking to the future, this thesis opens up the following broad research questions:

- How can we further exploit long-term behavior?
- Can long-term behavior benefit other problems in the memory system?

6.1 How can we further exploit long-term behavior?

While the solutions presented in this thesis have sigificantly advanced the state-of-the-art, there is still room for improvement. One common limitation of both Hawkeye and ISB is that they rely on load instructions to organize and remember their history information, and load instructions may not always be the best granularity for such problems. For example, an unrolled loop might benefit from aggregating information across related load instructions, while a load instruction that accesses a large, complex object may benefit from *de-aliasing* the load instruction with richer program context such as path history. Finding the right set of *features* for any problem is difficult because it depends on both the program and the characteristic of the optimization problem. To explore this design space systematically, we believe that offline machine learning techniques, such as clustering, may offer useful insights. On the other hand, online machine learning techniques can help us design practical

solutions with many rich features. For example, Sparse Distributed Memories [47] are a simple and space-efficient way to combine multiple contexts for prediction problems.

6.2 Can long-term behavior benefit other problems?

Given our observations about long-term behavior for caching and prefetcher, it is natural to ask if long-term behavior can help us improve other prediction problems in the memory system. One avenue of future research is to explore the utility of long-term information for other prefetching problems, such as TLB and instruction prefetching, and we believe that the ideas presented in this thesis will be useful in exploiting long-term behavior for these problems as well. For example, the notion of indirection for irregular accesses, and the notion of finding aggregate strides should be broadly applicable, even if exact design features vary. Similarly, our insights about cache replacement may generate new solutions for other caching systems, including software caches.

Another potentially interesting application for long-term behavior is in exploring better solutions for complex resource allocation problems at the microarchitectural level. Given the significant contention for shared resources among many cores, there are many examples of hardware resource allocation problems, including cache partitioning [77, 70], degree control [24], and memory scheduling in multi-core environments. Intuitively, resource allocation problems are likely to benefit from long-term behavior for the same reasons that cache replacement benefits from long-term behavior as they all require modeling overlapping demands of independent requests.

More broadly, long-term information will be critical in defining and identifying program phases, which has diverse applications for many microarchitectural problems.

In summary, in this thesis, we have significantly improved memory system performance by rethinking our learning goals and by identifying new representations that fit our learning goals. Our design philosophy is extensible to many other problems in the memory system, and it offers exciting opportunities for future research.

Bibliography

[1] The 2nd Data Prefetching Championship(DPC2). http://comparch-conf.gatech.edu/dpc2/final_program.html, 2015.

[2] J. Abella, A. González, X. Vera, and M. F. P. O'Boyle. Iatac: A smart predictor to turn-off l2 cache lines. *ACM Trans. Archit. Code Optim.*, 2(1):55–77, March 2005.

[3] A. R. Alameldeen, A. Jaleel, M. Qureshi, and J. Emer. 1st JILP workshop on computer architecture competitions (JWAC-1) cache replacement championship. 2010.

[4] J.-L. Baer and T.-F. Chen. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers*, 44(5):609–623, May 1995.

[5] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, pages 78–101, 1966.

[6] L. A. Belady and F. P. Palermo. On-line measurement of paging behavior by the multivalued MIN algorithm. *IBM Journal of Research and Development*, 18:2–19, 1974.

[7] I. Burcea, S. Somogyi, A. Moshovos, and B. Falsafi. Predictor virtualization. In *Proceedings of the 13th international conference on Architectural support for*

programming languages and operating systems, ASPLOS XIII, pages 157–167. ACM, 2008.

[8] D. Burger, T. R. Puzak, W.-F. Lin, and S. K. Reinhardt. Filtering superfluous prefetches using density vectors. In *ICCD '01: Proceedings of the International Conference on Computer Design: VLSI in Computers & Processors*, pages 124–133, 2001.

[9] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout. An evaluation of high-level mechanistic core models. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2014.

[10] J. B. Carter, W. C. Hsieh, L. Stoller, M. R. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. A. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a smarter memory controller. In *HPCA*, pages 70–79, 1999.

[11] M. Chaudhuri. Pseudo-LIFO: the foundation of a new family of replacement policies for last-level caches. In *42nd Annual IEEE/ACM International Symposium on Microarchitecture(MICRO)*, pages 401–412, 2009.

[12] C. F. Chen, S.-H. Yang, B. Falsafi, and A. Moshovos. Accurate and complexity-effective spatial pattern prediction. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, HPCA '04, pages 276–288, 2004.

[13] T. M. Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 191–202, 2001.

[14] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure lay-

out. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming Language Design and Implementation*, PLDI '99, pages 1–12, 1999.

[15] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *InACM SIGPLAN Conference on Programming Language Design and Implementation*, 2002.

[16] Y. Chou. Low-cost epoch-based correlation prefetching for commercial applications. In *MICRO*, pages 301–313, 2007.

[17] I.-H. Chung, C. Kim, H.-F. Wen, and G. Cong. Application data prefetching on the ibm blue gene/q supercomputer. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, 2012.

[18] J. Collins, S. Sair, B. Calder, and D. M. Tullsen. Pointer cache assisted prefetching. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 35, pages 62–73, 2002.

[19] R. Cooksey, S. Jourdan, and D. Grunwald. A stateless, content-directed data prefetching mechanism. *SIGARCH Computer Architecture News*, 30(5):279–290, October 2002.

[20] P. Diaz and M. Cintra. Stream chaining: exploiting multiple levels of correlation in data prefetching. In *ISCA*, pages 81–92, 2009.

[21] M. Dimitrov and H. Zhou. Combining local and global history for high performance data prefetching. In *Journal of Instruction-Level Parallelism Data Prefetching Championship*, volume 13, 2011.

[22] R. G. Dreslinski, A. G. Saidi, T. Mudge, and S. K. Reinhardt. Analysis of hardware prefetching across virtual page boundaries. In *Proceedings of the 4th international conference on Computing frontiers*, pages 13–22. ACM, 2007.

[23] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero, and A. V. Veidenbaum. Improving cache management policies using dynamic reuse distances. In *45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 389–400, 2012.

[24] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt. Coordinated control of multiple prefetchers in multi-core systems. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 316–326. ACM, 2009.

[25] E. Ebrahimi, O. Mutlu, and Y. N. Patt. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In *HPCA*, pages 7–17, 2009.

[26] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. Memory-system design considerations for dynamically-scheduled processors. In *ISCA '97: Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 133–143, 1997.

[27] H. Gao and C. Wilkerson. A dueling segmented LRU replacement algorithm with adaptive bypassing. In *JWAC 2010-1st JILP Workshop on Computer Architecture Competitions: Cache Replacement Championship*, 2010.

[28] A. González, C. Aliagas, and M. Valero. A data cache with multiple caching strategies tuned to different types of locality. In *International Conference on Supercomputing*, pages 338–347, 1995.

[29] E. G. Hallnor and S. K. Reinhardt. A fully associative software-managed cache design. In *International Symposium on Computer Architecture (ISCA)*, pages 107–116, 2000.

[30] G. Hamerly, E. Perelman, J. Lau, and B. Calder. SimPoint 3.0: Faster and more flexible program phase analysis. *Journal of Instruction Level Parallelism*, 7(4):1–28, 2005.

[31] Z. Hu, S. Kaxiras, and M. Martonosi. Timekeeping in the memory system: Predicting and optimizing memory behavior. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ISCA '02, pages 209–220, 2002.

[32] Z. Hu, M. Martonosi, and S. Kaxiras. TCP: tag correlating prefetchers. In *HPCA*, pages 317–326, 2003.

[33] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng. The garbage collection advantage: improving program locality. In *Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 69–80, 2004.

[34] I. Hur and C. Lin. Memory prefetching using adaptive stream detection. In *Proceedings of the 39th International Symposium on Microarchitecture*, pages 397–408, 2006.

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

[36] A. Jain and C. Lin. Linearizing irregular memory accesses for improved correlated prefetching. In *46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, December 2013.

[37] A. Jain and C. Lin. Back to the future: Leveraging beladys algorithm for

improved cache replacement. In *4rrd Annual IEEE/ACM International Symposium on Computer Architecture (ISCA)*, June 2016. [To Appear].

[38] A. Jaleel. Memory characterization of workloads using instrumentation-driven simulation– a Pin-based memory characterization of the SPEC CPU2000 and SPEC CPU2006 benchmark suites. Technical report, VSSAD Technical Report 2007, 2007.

[39] A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob. Cmp$im: A Pin-based on-the-fly multi-core cache simulator. In *Proceedings of the Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS), co-located with ISCA*, pages 28–36, 2008.

[40] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr., and J. Emer. Adaptive insertion policies for managing shared caches. In *17th International Conference on Parallel Architectures and Compilation Techniques*, pages 208–219, 2008.

[41] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer. High performance cache replacement using re-reference interval prediction (RRIP). In *International Symposium on Computer Architecture (ISCA)*, pages 60–71. ACM, 2010.

[42] D. A. Jiménez. Insertion and promotion for tree-based PseudoLRU last-level caches. In *46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 284–296, 2013.

[43] T. L. Johnson, M. C. Merten, and W.-M. W. Hwu. Run-time spatial locality detection and optimization. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 57–64, 1997.

[44] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *Proceed-*

ings of the 24th Annual International Symposium on Computer Architecture, pages 252–263, 1997.

[45] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *International Symposium on Computer Architecture (ISCA)*, pages 364–373, 1990.

[46] G. B. Kandiraju and A. Sivasubramaniam. Going the distance for tlb prefetching: an application-driven study. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, 2002.

[47] P. Kanerva. *Sparse distributed memory*. MIT press, 1988.

[48] G. Keramidas, P. Petoumenos, and S. Kaxiras. Cache replacement based on reuse-distance prediction. In *25th International Conference on Computer Design (ICCD)*, pages 245–250, 2007.

[49] S. Khan, Y. Tian, and D. A. Jimenez. Sampling dead block prediction for last-level caches. In *43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 175–186, 2010.

[50] M. Kharbutli and Y. Solihin. Counter-based cache replacement algorithms. In *Proceedings of the International Conference on Computer Design (ICCD)*, pages 61–68, 2005.

[51] C. S. Kim. LRFU: a spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Transactions on Computers*, pages 1352–1361, 2001.

[52] S. Kumar and C. Wilkerson. Exploiting spatial locality in data caches using spatial footprints. *SIGARCH Computer Architecture News*, 26(3):357–368, April 1998.

[53] A.-C. Lai and B. Falsafi. Selective, accurate, and timely self-invalidation using last-touch prediction. In *the 27th International Symposium on Computer Architecture (ISCA)*, pages 139–148, 2000.

[54] A.-C. Lai, C. Fide, and B. Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *Proceedings. 28th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2001.

[55] H. Liu, M. Ferdman, J. Huh, and D. Burger. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 222–233, 2008.

[56] C.-K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. *SIGOPS Operating Systems Review*, 30(5):222–233, September 1996.

[57] P. Michaud. Best-offset hardware prefetching. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 469–480. IEEE, 2016.

[58] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith. Ac/dc: An adaptive data cache prefetcher. In *IEEE PACT*, pages 135–145, 2004.

[59] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. *IEEE Micro*, 25(1):90–97, 2005.

[60] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In *ACM SIGMOD Record*, pages 297–306. ACM, 1993.

[61] S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary

cache replacement. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 24–33, April 1994.

[62] A. Patel, F. Afram, S. Chen, and K. Ghose. MARSSx86: A Full System Simulator for x86 CPUs. In *Design Automation Conference 2011 (DAC'11)*, 2011.

[63] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder. Using SimPoint for accurate and efficient simulation. In *the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 318–319, 2003.

[64] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh. Increasing tlb reach by exploiting clustering in page translations. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 558–567. IEEE, 2014.

[65] M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, pages 895–913, 1999.

[66] 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*. IEEE, 2014.

[67] T. R. Puzak. *Analysis of Cache Replacement-algorithms*. PhD thesis, University of Massachusetts Amherst, 1985.

[68] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive insertion policies for high performance caching. In *International Symposium on Computer Architecture (ISCA)*, pages 381–391. ACM, 2007.

[69] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A case for MLP-aware cache replacement. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 167–178, 2006.

[70] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 423–432, 2006.

[71] M. K. Qureshi, D. Thompson, and Y. N. Patt. The V-Way cache: demand-based associativity via global replacement. In *International Symposium on Computer Architecture (ISCA)*, pages 544–555, 2005.

[72] K. Rajan and R. Govindarajan. Emulating optimal replacement with a shepherd cache. In *the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 445–454, 2007.

[73] J. T. Robinson and M. V. Devarakonda. Data cache management using frequency-based replacement. In *the ACM Conference on Measurement and Modeling Computer Systems (SIGMETRICS)*, pages 134–142, 1990.

[74] A. Roth, A. Moshovos, and G. S. Sohi. Dependence based prefetching for linked data structures. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, ASPLOS-VIII, pages 115–126, 1998.

[75] A. Roth and G. S. Sohi. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, ISCA '99, pages 111–121, 1999.

[76] S. Sair, T. Sherwood, and B. Calder. A decoupled predictor-directed stream

prefetching architecture. *IEEE Transactions on Computers*, 52(3):260–276, March 2003.

[77] D. Sanchez and C. Kozyrakis. Vantage: scalable and efficient fine-grain cache partitioning. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 57–68. ACM, 2011.

[78] V. Seshadri, O. Mutlu, M. A. Kozuch, and T. C. Mowry. The evicted-address filter: A unified mechanism to address both cache pollution and thrashing. In *the 21st Int'l Conference on Parallel Architectures and Compilation Techniques*, pages 355–366, 2012.

[79] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chisthi. Efficiently prefetching complex address patterns. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 141–152. ACM, 2015.

[80] P. Shivakumar and N. Jouppi. Cacti 3.0: An integrated cache timing, power, and area model. Technical report, Technical Report 2001/2, Compaq Computer Corporation, 2001.

[81] Y. Smaragdakis, S. Kaplan, and P. Wilson. EELRU: simple and effective adaptive page replacement. In *ACM SIGMETRICS Performance Evaluation Review*, pages 122–133, 1999.

[82] A. Smith. Sequential program prefetching in memory hierarchies. *IEEE Transactions on Computers*, 11(12):7–12, December 1978.

[83] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 234–244, 2000.

[84] Y. Solihin, J. Lee, and J. Torrellas. Using a user-level memory thread for correlation prefetching. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 171–182, 2002.

[85] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi. Spatio-temporal memory streaming. In *ISCA*, pages 69–80, 2009.

[86] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Spatial memory streaming. In *ISCA '06: Proceedings of the 33th Annual International Symposium on Computer Architecture*, pages 252–263, 2006.

[87] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007.

[88] R. Subramanian, Y. Smaragdakis, and G. H. Loh. Adaptive caches: Effective shaping of cache behavior to workloads. In *the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 385–396. IEEE Computer Society, 2006.

[89] Z. Wang, D. Burger, K. S. McKinley, S. K. Reinhardt, and C. C. Weems. Guided region prefetching: a cooperative hardware/software approach. *SIGARCH Computer Architecture News*, 31(2):388–398, May 2003.

[90] Z. Wang, K. McKinley, A. L. Rosenberg, and C. C. Weems. Using the compiler to improve cache replacement decisions. In *Parallel Architectures and Compilation Techniques*, pages 199–208, 2002.

[91] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos. Temporal streams in commercial server applications. In *IISWC*, pages 99–108, 2008.

[92] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos. Practical off-chip meta-data for temporal memory streaming. In *HPCA*, pages 79–90, 2009.

[93] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos. Making address-correlated prefetching practical. *IEEE Micro*, 30(1):50–59, 2010.

[94] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi. Temporal streaming of shared memory. *SIGARCH Computer Architecture News*, 33(2):222–233, May 2005.

[95] W. A. Wong and J.-L. Baer. Modified LRU policies for improving second-level cache behavior. In *High-Performance Computer Architecture, 2000*, pages 49–60, 2000.

[96] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely, Jr., and J. Emer. SHiP: Signature-based hit predictor for high performance caching. In *44th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 430–441, 2011.

[97] C.-J. Wu, A. Jaleel, M. Martonosi, S. C. Steely, Jr., and J. Emer. PACMan: prefetch-aware cache management for high performance caching. In *44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 442–453, 2011.

[98] Y. Xie and G. H. Loh. Pipp: promotion/insertion pseudo-partitioning of multi-core shared caches. In *ACM SIGARCH Computer Architecture News*, pages 174–183, 2009.