Evaluation of Hardware Data Prefetchers on Server Processors

MOHAMMAD BAKHSHALIPOUR, Sharif University of Technology
SEYEDALI TABAEIAGHDAEI, Sharif University of Technology
PEJMAN LOTFI-KAMRAN, Institute for Research in Fundamental Sciences (IPM)
HAMID SARBAZI-AZAD, Sharif University of Technology & Institute for Research in Fundamental Sciences (IPM)

Data prefetching, i.e., the act of predicting application's future memory accesses and fetching those that are not in the on-chip caches, is a well-known and widely-used approach to hide the long latency of memory accesses. The fruitfulness of data prefetching is evident to both industry and academy: nowadays, almost every high-performance processor incorporates a few data prefetchers for capturing various access patterns of applications; besides, there is a myriad of proposals for data prefetching in the research literature, where each proposal enhances the efficiency of prefetching in a specific way.

In this survey, we evaluate the effectiveness of data prefetching in the context of server applications and shed light on its design trade-offs. To do so, we choose a target architecture based on a contemporary server processor and stack various state-of-the-art data prefetchers on top of it. We analyze the prefetchers in terms of the ability to predict memory accesses and enhance overall system performance, as well as their imposed overheads. Finally, by comparing the state-of-the-art prefetchers with impractical ideal prefetchers, we motivate further work on improving data prefetching techniques.

CCS Concepts: •**General and reference** →**Surveys and overviews;** *Evaluation;* •**Computer systems organization** →*Architectures;*

Additional Key Words and Phrases: Data Prefetching, Scale-Out Workloads, Server Processors, and Spatio-Temporal Correlation.

ACM Reference format:
Mohammad Bakhshalipour, Seyedali Tabaeiaghdaei, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2019. Evaluation of Hardware Data Prefetchers on Server Processors. *ACM Comput. Surv.* 52, 3, Article 52 (June 2019), 29 pages.
DOI: https://doi.org/10.1145/3312740

New Paper, Not an Extension of a Conference Paper.
Author's addresses: Mohammad Bakhshalipour and Seyedali Tabaeiaghdaei, Department of Computer Engineering, Sharif University of Technology; Pejman Lotfi-Kamran, School of Computer Science, Institute for Research in Fundamental Sciences (IPM); Hamid Sarbazi-Azad, Department of Computer Engineering, Sharif University of Technology and School of Computer Science, Institute for Research in Fundamental Sciences (IPM).

DOI: https://doi.org/10.1145/3312740

1 INTRODUCTION

Server workloads like *Media Streaming* and *Web Search* serve millions of users and are considered an important class of applications. Such workloads run on large-scale data-center infrastructures that are backed by processors which are essentially tuned for low latency and quality-of-service guarantees. These processors typically include a handful of high-clock frequency, aggressively-speculative, and deeply-pipelined cores so as to run server applications as fast as possible, satisfying end-users' latency requirements [34, 36, 39, 71, 74, 80, 81].

Much to processor designer's chagrin, bottlenecks in the memory system prevent server processors from getting high performance on server applications. As server workloads operate on a large volume of data, they produce active memory working sets that dwarf the capacity-limited on-chip caches of server processors and reside in the off-chip memory; hence, these applications frequently miss the data in the on-chip caches and access the long-latency memory to retrieve it. Such frequent data misses preclude server processors from reaching their peak performance because cores are idle waiting for the data to arrive [36, 37, 40, 61, 62, 74, 109].

System architects have proposed various strategies to overcome the performance penalty of frequent memory accesses. *Data Prefetching* is one of these strategies that has demonstrated significant performance potentials. Data prefetching is the art of predicting future memory accesses and fetching those that are not in the cache before a core explicitly asks for them in order to *hide* the long latency of memory accesses. Nowadays, virtually every high-performance computing chip uses a few data prefetchers (e.g., *Intel Xeon Phi* [102], *IBM Blue Gene/Q* [43], *AMD Opteron* [24], and *UltraSPARC III* [45]) to capture regular and/or irregular memory access patterns of various applications. In the research literature, likewise, there is a myriad of proposals for data prefetching, where every proposal makes the prefetching more efficient in a specific way.

In this study, we assess the effectiveness of hardware data prefetchers in the context of server applications. We stack recent, as well as classic, hardware data prefetchers on a system which is modeled based on a modern high-end server processor. Then we perform a detailed analysis of every prefetcher and quantify its ability at predicting cache misses and improving overall system performance, as well as its imposed overheads. In a nutshell, we make the following contributions in this paper:

- We characterize memory access patterns of server applications and show how these patterns lead to different classes of correlations, from which data prefetchers can predict future memory accesses.
- We implement state-of-the-art hardware data prefetchers in the research literature for evaluating their usefulness at covering data misses and boosting the performance of server workloads.
- We perform a sensitivity analysis for every data prefetcher to shed light on how much silicon real estate it needs to be able to predict a reasonable number of cache misses.
- We highlight the overheads of every data prefetching technique and discuss the feasibility of implementing it in modern server processors.
- We compare the state-of-the-art data prefetchers with ideal data prefetchers and show that, notwithstanding four decades of research on hardware data prefetching, there is still a significant room for improvement.

The rest of this paper is organized as follows. Section 2 briefly surveys non-hardware-data prefetching strategies that target eliminating or reducing the negative effect of data misses and highlights their limitations. Section 3 presents a short background on various classes of hardware data prefetching techniques and discusses their opportunities and limitations. Section 4 describes the mechanism of several state-of-the-art data prefetchers that we evaluate in this survey. Section 5 details the evaluation methodology, and Section 6 discusses the outcomes of the evaluation. Section 7 touches on techniques that can be used with hardware data prefetchers to maximize their efficiency. Finally, Section 8 concludes the paper and draws a roadmap for future research in hardware data prefetching.

2 NON-HARDWARE DATA PREFETCHING

Progress in technology fabrication accompanied by circuit-level and microarchitectural advancements have brought about significant enhancements in the processors' performance over the past decades. Meanwhile, the performance of memory systems has not held speed with that of the processors, forming a large gap between the performance of processors and memory systems [4, 41, 42, 108, 114]. As a consequence, numerous approaches have been proposed to enhance the execution performance of applications by bridging the processor-memory performance gap. Hardware data prefetching is just one of these approaches. Hardware data prefetching bridges the gap by proactively fetching the data ahead of the cores' requests in order to eliminate the idle cycles in which the processor is waiting for the response of the memory system. In this section, we briefly review the other approaches that target the same goal (i.e., bridging the processor-memory performance gap) but in other ways.

Multithreading [86] enables the processor to better utilize its computational resources, as stalls in one thread can be overlapped with the execution of other thread(s) [5, 25, 33, 63, 64, 96]. Multithreading, however, only improves *throughput* and does nothing for (or even worsens) the *response time* [42, 72, 74], which is crucial for satisfying the strict latency requirements of server applications.

Thread-Based Prefetching techniques [22, 23, 38, 59, 70] exploit idle thread contexts or distinct pre-execution hardware to drive helper threads that try to overlap the cache misses with speculative execution. Such helper threads, formed either by the hardware or by the compiler, execute a piece of code that prefetches for the main thread. Nonetheless, the *additional* threads and fetch/execution bandwidth may not be available when the processor is fully utilized. The abundant request-level parallelism of server applications [36, 74] makes such schemes ineffective in that the helper threads need to compete with the main threads for the hardware context.

Runahead Execution [84, 85] makes the execution resources of a core that would otherwise be stalled on an off-chip cache miss to go ahead of the stalled execution in an attempt to discover additional load misses. Similarly, **Branch Prediction Directed Prefetching** [58] utilizes the branch predictor to run in advance of the executing program, thereby prefetching load instructions along the expected future path. Such approaches, nevertheless, are *constrained* by the accuracy of the branch predictor and can cover simply a portion of the miss latency, since the runahead thread/branch predictor may not be capable of executing far ahead in advance to *completely* hide a cache miss. Moreover, these approaches can only prefetch *independent* cache misses [44] and may not be effective for many of the server workloads, e.g., *OLTP* and *Web* applications, that are characterized by long chains of dependent memory accesses [10, 93].

On the software side, there are efforts to re-structure programs to boost chip-level **Data Sharing** and **Data Reuse** [55, 68] in order to decrease off-chip accesses. While these techniques are useful for workloads with modest datasets, they fall short of efficiency for big-data server workloads, where the multi-gigabyte working sets of workloads dwarf the few megabytes of on-chip cache capacity. The ever-growing datasets of server workloads make such approaches *unscalable*. **Software Prefetching** techniques [18, 20, 77, 95, 97, 117] profile the program code and insert prefetch instructions to eliminate cache misses. While these techniques are shown to be beneficial for small benchmarks, they usually require significant *programmer effort* to produce optimized code to generate timely prefetch requests.

Memory-Side Prefetching techniques [46, 103, 115] place the hardware for data prefetching near DRAM, for the sake of saving precious SRAM budget. In such approaches (e.g., [103]), prefetching is performed by a user thread running near the DRAM, and prefetched pieces of data are sent to the on-chip caches. Unfortunately, such techniques lose the *predictability* of core requests [82] and are incapable of performing *cache-level optimizations* (e.g., avoiding cache pollution [106]).

3 BACKGROUND

In this section, we briefly overview a background on hardware data prefetching and refer the reader to prior work [35, 82] for more details. For simplicity, in the rest of the paper, we use the term *prefetcher* to refer to the *core-side hardware data prefetcher.*

3.1 Predicting Memory References

The first step in data prefetching is *predicting* future memory accesses. Fortunately, data accesses demonstrate several types of correlations and localities, that lead to the formation of *patterns* among memory accesses, from which data prefetchers can predict future memory references. These patterns emanate from the layout of programs' data structures in the memory, and the algorithm and the high-level programming constructs that operate on these data structures.

In this work, we classify the memory access patterns of applications into three distinct categories: (1) *strided*, (2) *temporal*, and (3) *spatial* access patterns.

3.1.1 Strided Accesses. Strided access pattern refers to a sequence of memory accesses in which the *distance* of consecutive accesses is constant (e.g., $\{A, A+k, A+2k, \dots\}$). Such patterns are abundant in programs with dense matrices and frequently come into sight when programs operate on multi-dimensional arrays. Strided accesses also appear in pointer-based data structures when memory allocators arrange the objects sequentially and in a constant-size manner in the memory [26].

3.1.2 Temporal Address Correlation. Temporal address correlation [10] refers to a sequence of addresses that favor being accessed together and in the same *order* (e.g., if we observe $\{A, B, C, D\}$, then it is likely for $\{B, C, D\}$ to follow $\{A\}$ in the future). Temporal address correlation stems fundamentally from the fact that programs consist of loops, and is observed when data structures such as lists, arrays, and linked lists are traversed. When data structures are stable [19], access patterns recur, and the temporal address correlation is manifested [10].

3.1.3 Spatial Address Correlation. Spatial address correlation [11] refers to the phenomenon that similar access patterns occur in different *regions* of memory (e.g., if a program visits locations $\{A, B, C, D\}$ of Page X, it is probable that it visits locations $\{A, B, C, D\}$ of other pages as well). Spatial correlation transpires because applications use various objects with a regular and fixed layout, and accesses reappear while traversing data structures [11].

3.2 Prefetching Lookahead

Prefetchers need to issue *timely* prefetch requests for the predicted addresses. Preferably, a prefetcher sends prefetch requests well in advance and supply enough storage for the prefetched blocks in order to hide the entire latency of memory accesses. An early prefetch request may cause evicting a useful block from the cache, and a late prefetch may decrease the effectiveness of prefetching in that a portion of the long latency of a memory access is exposed to the processor.

Prefetching lookahead refers to *how far ahead of the demand miss stream the prefetcher can send requests.* An aggressive prefetcher may offer a high prefetching lookahead (say, eight) and issue many prefetch requests ahead of the processor to hide the entire latency of memory accesses; on the other hand, a conservative prefetcher may offer a low prefetching lookahead and send a single prefetch request in advance of the processor's demand to avoid wasting resources (e.g., cache storage and memory bandwidth). Typically, there is a trade-off between the aggressiveness of a prefetching technique and its accuracy: making a prefetcher more aggressive usually leads to covering more data-miss–induced stall cycles but at the cost of fetching more useless data.

Some pieces of prior work propose to dynamically *adjust* the prefetching lookahead [58, 65, 106]. Based on the observation that the optimal prefetching degree is different for various applications and various execution phases of a particular application, as well, these approaches employ heuristics to increase or decrease the prefetching lookahead. For example, SPP [65] monitors the accuracy of issued prefetch requests and reduce the prefetching lookahead if the accuracy becomes smaller than a predefined threshold.

3.3 Location of Data Prefetcher

Prefetching can be employed to move the data from lower levels of the memory hierarchy to any higher level[1]. Prior work used data prefetchers at all cache levels, from the primary data cache to the shared last-level cache.

The location of a data prefetcher has a profound impact on its overall behavior [78]. A prefetcher in the first-level cache can observe all memory accesses, and hence, is able to issue highly-accurate prefetch requests, but at the cost of imposing large storage overhead for recording the metadata information. In contrast, a prefetcher in the last-level cache observes the access sequences that have been filtered at higher levels of the memory hierarchy, resulting in lower prediction accuracy, but higher storage efficiency.

3.4 Prefetching Hazards

A naive deployment of a data prefetcher not only may not improve the system performance but also may significantly harm the performance and energy-efficiency [8]. The two well-known major drawbacks of data prefetching are (1) cache pollution and (2) off-chip bandwidth overhead.

3.4.1 Cache Pollution. Data prefetching may increase the demand misses by replacing useful cache blocks with useless prefetched data, harming the performance. Cache pollution usually occurs when an aggressive prefetcher exhibits low accuracy and/or when prefetch requests of a core in a many-core processor compete for shared resources with demand accesses of other cores [31].

3.4.2 Bandwidth Overhead. In a many-core processor, prefetch requests of a core can delay demand requests of another core because of contending for memory bandwidth [31]. This interference is the major obstacle of using data prefetchers in many-core processors, and the problem gets thornier as the number of cores increases [29, 66].

3.5 Placing Prefetched Data

Data prefetchers usually place the prefetched data into one of the following two structures: (1) the cache itself, and (2) an auxiliary buffer next to the cache. In case an auxiliary buffer is used for the

[1]We use the term higher (lower) levels of the memory hierarchy to refer to the levels closer to (further away from) the core, respectively.

prefetched data, demand requests first look for the data in the cache; if the data is not found, the auxiliary buffer is searched before sending a request to the lower levels of the memory hierarchy.

Storing the prefetched data into the cache lowers the latency of accessing data when the prediction is correct. However, when the prediction is incorrect or when the prefetch request is not timely (i.e., too early), having the prefetched data in the cache may result in evicting useful cache blocks.

4 STATE-OF-THE-ART DATA PREFETCHERS

In this section, we introduce state-of-the-art data prefetchers and describe their mechanism. Based on the type of access patterns they capture, we classify the design space of hardware data prefetchers into three categories.

4.1 Stride Prefetching

Stride prefetchers are widely used in commercial processors (e.g., *IBM Power4* [107], *Intel Core* [28], *AMD Opteron* [24], *Sun UltraSPARC III* [45]) and have been shown quite effective for desktop and engineering applications. Stride prefetchers [6, 49, 50, 57, 90, 98, 100, 116] detect streams (i.e., the sequence of consecutive addresses) that exhibit strided access patterns (Section 3.1.1) and generate prefetch requests *by adding the detected stride to the last observed address*. While early stride prefetchers [6, 100] capture only access patterns that are separated by a *constant* stride, latter proposals target access patterns that exhibit *multiple* [50] or *variable* [98] strides.

Offset prefetching [79, 92] is an evolution of stride prefetching, in which, the prefetcher *does not try to detect strided streams*. Instead, whenever a core requests for a cache block (e.g., A), the offset prefetcher prefetches the cache block that is distanced by k cache lines (e.g., $A + k$), where k is the *prefetch offset*. In other words, offset prefetchers do not correlate the accessed address to any specific stream; rather, they treat the addresses *individually*, and based on the prefetch offset, they issue a prefetch request for every accessed address. It is noteworthy that the offset prefetcher may adjust the prefetch offset dynamically based on the application's behavior.

Stride prefetchers impose minimal area overhead and are highly effective when a program exhibits strided access patterns (e.g., programs with dense matrices), but fall short of efficiency for pointer-chasing applications, in which strided accesses are scarce [84, 111]. We include two prefetchers from this class: (1) Instruction-Based Stride Prefetcher [6], and (2) Best-Offset Prefetcher [79].

4.1.1 Instruction-Based Stride Prefetcher (IBSP). We incorporate IBSP as it is prevalent in today's commercial processors. The prefetcher tracks the strided streams on a per load instruction basis: the prefetcher observes accesses issued by individual load instructions and sends prefetch requests if the accesses manifest a strided pattern. Figure 1 shows the organization of IBSP's metadata table, named *Reference Prediction Table (RPT)*. *RPT* is a structure tagged and indexed with the *Program Counter (PC)* of load instructions. Each entry in the *RPT* corresponds to a specific load instruction; it keeps the *Last Block* referenced by the instruction and the *Last Stride* observed in the stream (i.e., the distance of two last addresses accessed by the instruction).

Upon each trigger access (i.e., a cache miss or a prefetch hit), the *RPT* is searched with the PC of the instruction. If the search results in a miss, it means that no history does exist for the instruction, and hence, no prefetch request can be issued. Under two circumstances, a search may result in a miss: (1) whenever a load instruction is a new one in the execution flow of the program, and ergo, no history has been recorded for it so far, and (2) whenever a load instruction is re-executed after a long time, and the corresponding recorded metadata information has been evicted from the *RPT* due to conflicts. In such cases when no matching entry does exist in the *RPT*, a new entry is allocated for the instruction, and possibly a victim entry is evicted. The new entry is

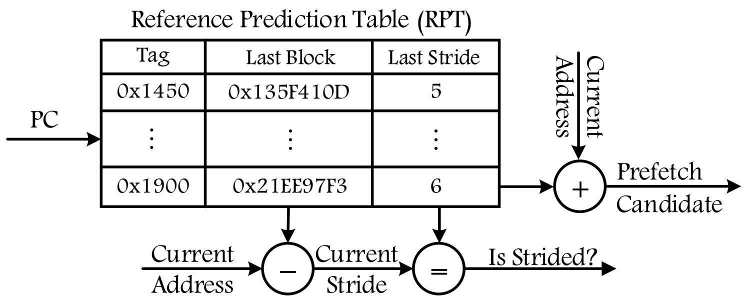

Fig. 1. The organization of Instruction-Based Stride Prefetcher (IBSP). The 'RPT' keeps track of various streams.

tagged with the PC, and the *Last Block* field of the entry is filled with the referenced address. The *Last Stride* is also set to zero (an invalid value) as no stride has yet been observed for this stream. However, if searching the *RPT* results in a hit, it means that there is a recorded history for the instruction. In this case, the recorded history information is checked with the current access to find out whether or not the stream is a strided one. To do so, the difference of the current address and the *Last Block* is calculated to get the *current stride*. Then, the current stride is checked against the recorded *Last Stride*. If they do not match, it is implied that the stream does not exhibit a strided access pattern. However, if they match, it is construed that the stream is a strided one as three consecutive accesses have produced two *identical* strides. In this case, based on the lookahead of the prefetcher (Section 3.2), several prefetch requests are issued by *consecutively* adding the observed stride to the requested address. For example, if the current address and the current stride are A and k, respectively, and the lookahead of prefetching is three, prefetch candidates will be $\{A + k, A + k + k, A + k + k + k\}$. Finally, regardless of the fact that the stream is strided or not, the corresponding *RPT* entry is updated: the *Last Block* is updated with the current address, and the *Last Stride* takes the value of the current stride.

4.1.2 Best-Offset Prefetcher (BOP). BOP is a recent proposal for offset prefetching, as well as the winner of the Second Data Prefetching Championship (DPC-2) [91]. BOP extends Sandbox Prefetcher (SP) [92], which is the primary proposal for offset prefetching, and enhances its *timeliness*. We first describe the operations of SP and then elucidate how BOP extends it.

SP is an offset prefetcher and attempts to dynamically find the offsets that yield *accurate* prefetch requests. To find such offsets, SP defines an *evaluation period* in which it assesses the prefetching accuracy of multiple predefined offsets, ranging from $-n$ to $+n$, where n is a constant, say, eight. For every prefetch offset, a *score* value is associated, and when the evaluation period is over, only offsets whose score values are beyond a certain threshold are considered accurate offsets; and actual prefetch requests are issued using such offsets.

In the evaluation period, for determining the score values, SP issues *virtual prefetch* requests using various offsets. Virtual prefetching refers to the act of adding the information of candidate prefetch addresses to specific storage rather than actually prefetching them. That is, in the evaluation period, instead of issuing numerous costly prefetch requests (Section 3.4) using all offsets, prefetch candidates are simply inserted into specific storage. Later, when the application generates actual memory references, the stored prefetch candidates are checked against them: if an actual memory reference matches with a prefetch candidate in the specific storage, it is implied that the candidate was an accurate prefetch request, and accordingly, the score value of the offset that has generated this prefetch candidate is incremented.

For the sake of storage efficiency, SP uses a *Bloom Filter* [13] as the specific storage for keeping the record of prefetch candidates of each offset. Generally, *Bloom Filter* is a probabilistic data structure that is used for examining whether an element is *not* a member of a set. The filter has an array of counters and several hash functions, where each hash function maps the input to a counter in the array. Upon inserting an element into the filter, all counters identified by all of the hash functions are incremented, signifying the membership of the element. Upon checking the membership of an element, all counters identified by all of the hash functions are searched. If at least one of the counters be zero, it is construed that the element is *not* a member of the set as none of the corresponding counters has been incremented. In the context of SP, prefetch candidates generated by offsets are added to the *Bloom Filter.* Then, upon triggering an actual memory reference, *Bloom Filter* is checked to find out if the current memory reference has been inserted into the *Bloom Filter* as a prefetch candidate; and accordingly, the score value of the offsets are manipulated.

Figure 2 shows the hardware realization of SP that mainly consists of a *Sandbox Prefetch Unit (SPU)* and a *Bloom Filter. SPU* maintains the status of several specific offsets and evaluates them in a round-robin fashion. Upon each triggering access (i.e., cache miss or prefetch hit), the cache line address is checked against the *Bloom Filter* to determine if the cache line would have been prefetched by the *under-evaluation* offset. If the *Bloom Filter* contains the address, the score value of the offset is incremented; then the procedure repeats for the other offsets. When the evaluation period is over, only offsets whose score values are beyond a certain threshold are allowed to issue prefetch requests.

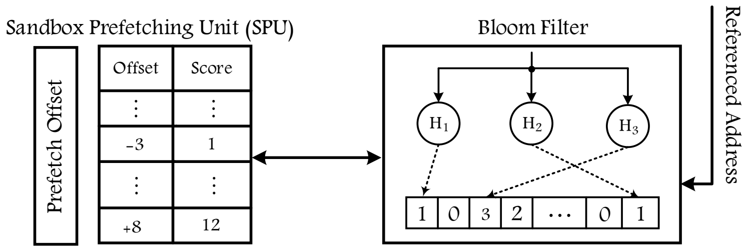

Fig. 2. The organization of Sandbox Prefetcher (SP). The 'SPU' keeps the score values of various evaluated offsets. The 'Bloom Filter' serves as temporary storage for keeping the prefetch candidates during the evaluation period of each offset. Each circle in the 'Bloom Filter' represents a different hash function. The 'Prefetch Offset' in the figure represents the under-evaluation offset.

SP chooses prefetch offsets merely based on their accuracy and ignores the timeliness. Nonetheless, accurate but *late* prefetches do not accelerate the execution of applications as much as timely prefetch requests do. Therefore, BOP tweaks SP and attempts to select offsets that result in timely prefetch requests, having the prefetched blocks ready before the processor actually asks for them.

Figure 3 shows the hardware structure of BOP. Similar to SP, BOP evaluates the efficiency of various offsets and chooses the best offset for prefetching. However, unlike SP, BOP promotes offsets that generate timely prefetch candidates rather than merely accurate ones. The main idea behind BOP is: "For k to be a timely prefetch offset for line A, line $A - k$ should have been accessed *recently*." That is, offsets whose prefetch candidates are used by the application *not much longer than they generated* are considered as the suitable offsets, and accordingly, their score values are incremented. In order to evaluate the timeliness of prefetch requests issued using various offsets, BOP replaces SP's *Bloom Filter* with a set-associative *Recent Requests Table (RRT).* The size of the

RRT is purposely chosen to be small in order to keep *only* recent requests. For every triggering event A, under the evaluation period of offset k, the score of the offset is incremented if line $A - k$ hits in the *RRT*. In other words, under the evaluation period of offset k, if line A is requested by the processor and line $A - k$ hits in the *RRT*, it is construed that "if offset k had issued a prefetch request upon line $A - k$, its prefetch requests (i.e., $A - k + k = A$) would have been timely." Therefore, k is identified as an offset whose prefetch requests match with the demand of the processor, and thus, its score is incremented.

Unlike SP that evaluates offsets in the range of $-n$ to $+n$ for some constant n, BOP evaluates 46 (almost random) constant offsets that were picked empirically. The other components and functions of BOP are similar to those of SP.

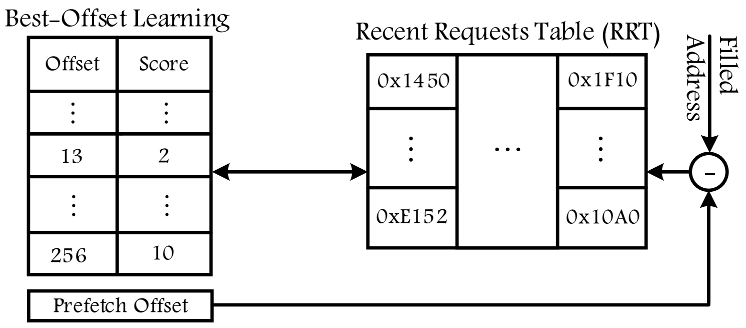

Fig. 3. The organization of Best-Offset Prefetcher (BOP). The 'Best-Offset Learning' keeps track of score values associated with various offsets, serving the function of 'SPU' in SP. The 'Recent Requests Table' holds the recent prefetch candidates. The 'Prefetch Offset' in the figure represents the under-evaluation offset.

4.2 Temporal Prefetching

Temporal prefetching refers to replaying the sequence of past cache misses in order to avert future misses. Temporal data prefetchers [9, 10, 21, 51, 56, 88, 103, 110, 111] record the sequence of data misses in the order they appear and use the recorded history for predicting future data misses. Upon a new data miss, they search the history and find a matching entry and replay the sequence of data misses after the match in an attempt to eliminate potential future data misses. A tuned version of temporal prefetching has been implemented in *IBM Blue Gene/Q*, where it is called List Prefetching [43].

Temporal prefetching is an ideal choice to eliminate long chains of *dependent* cache misses, that are common in pointer-chasing applications (e.g., *OLTP* and *Web*) [10]. A dependent cache miss refers to a memory operation that results in a cache miss and is dependent on data from a prior cache miss. Such misses have a marked effect on the execution performance of applications and impede the processor from making forward progress since both misses are fetched *serially* [10, 44]. Because of the lack of strided/spatial correlation among dependent misses, stride and spatial prefetchers are usually unable to prefetch such misses [104]; however, temporal prefetchers, by recording and replaying the sequences of data misses, can prefetch dependent cache misses and result in a significant performance improvement.

Temporal prefetchers, on the other face of the coin, also have shortcomings. Temporal prefetching techniques exhibit low accuracy as they do not know where streams end. That is, in the foundation of temporal prefetching, there is no wealth of information about *when prefetching should be stopped*; hence, temporal prefetchers continue issuing many prefetch requests until another triggering event occurs, resulting in a large overprediction. Moreover, as temporal prefetchers rely on address repetition, they are unable to prevent compulsory misses (unobserved misses) from happening. In other words, they can only prefetch cache misses that at least once have been observed in the past; however, there are many important applications (e.g., *DSS*) in which the majority of cache misses occurs only once during the execution of the application [11], for which temporal prefetching can do nothing. Furthermore, as temporal prefetchers require to store the correlation between addresses, they usually impose large storage overhead (tens of megabytes) that cannot be accommodated on-the-chip next to the cores. Consequently, temporal prefetchers usually place their metadata tables off-the-chip in the main memory. Unfortunately, placing the history information off-the-chip increases the latency of accessing metadata, and more importantly, results in a drastic increase in the off-chip bandwidth consumption for fetching and updating the metadata.

We include two state-of-the-art temporal prefetching techniques: (1) Sampled Temporal Memory Streaming [110], and (2) Irregular Stream Buffer [51].

4.2.1 *Sampled Temporal Memory Streaming (STMS).*

STMS is a state-of-the-art temporal data prefetcher that was proposed and evaluated in the context of server and scientific applications. The main observation behind STMS is that *the length of temporal streams widely differs* across programs and across different streams in a particular program, as well; ranging from a couple to hundreds of thousands of cache misses. In order to efficiently store the information of various streams, STMS uses a circular FIFO buffer, named *History Table*, and appends every observed cache miss to its end. This way, the prefetcher is not required to fix a specific predefined length for temporal streams in the metadata organization, that would be resulted in wasting storage for streams shorten than the predefined length or discarding streams longer than it; instead, all streams are stored next to each other in a storage-efficient manner. For locating every address in the *History Table*, STMS uses an auxiliary set-associative structure, named *Index Table*. The *Index Table* stores a *pointer* for every observed miss address to its last occurrence in the *History Table*. Therefore, whenever a cache miss occurs, the prefetcher first looks up the *Index Table* with the missed address and gets the corresponding pointer. Using the pointer, the prefetcher proceeds to the *History Table* and issues prefetch requests for addresses that have followed the missed address in the history.

Figure 4 shows the metadata organization of STMS, which mainly consists of a *History Table* and an *Index Table*. As both tables require multi-megabyte storage for STMS to have reasonable miss coverage, both tables are placed off-the-chip in the main memory. Consequently, every access to these tables (read or update) should be sent to the main memory and brings/updates a cache block worth of data. That is, for every stream, STMS needs to wait for two long (serial) memory requests to be sent (one to read the *Index Table* and one to read the correct location of the *History Table*) and their responses to come back to the prefetcher before issuing prefetch requests for the stream. The delay of the two off-chip memory accesses, however, is compensated over several prefetch requests of a stream if the stream is long enough.

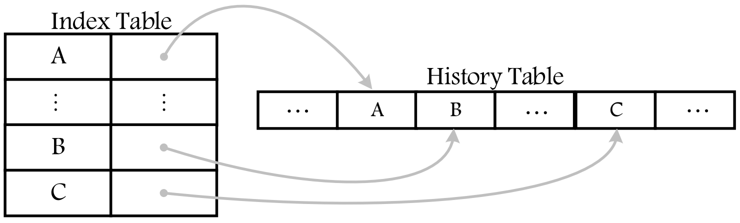

Fig. 4. The organization of Sampled Temporal Memory Streaming (STMS).

4.2.2 Irregular Stream Buffer (ISB). ISB is another state-of-the-art proposal for temporal data prefetching that targets irregular streams of temporally-correlated memory accesses. Unlike STMS that operates on the global miss sequences, ISB attempts to extract temporal correlation among memory references on a per load instruction basis (Section 4.1.1). The key innovation of ISB is the introduction of an extra *indirection* level for storing metadata information. ISB defines a new conceptual address space, named *Structural Address Space (SAS)*, and *maps* the temporally-correlated physical address to this address space in a way that they appear *sequentially*. That is, with this indirection mechanism, physical addresses that are temporally-correlated and used one after another, regardless of their distribution in the *Physical Address Space (PAS)* of memory, become spatially-located and appear one after another in *SAS*. Figure 5 shows a high-level example of this linearization.

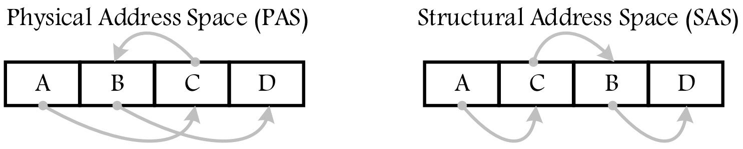

Fig. 5. An example of linearizing scattered temporally-correlated memory references.

ISB utilizes two tables to record a *bidirectional* mapping between address in *PAS* and *SAS*: one table, named *Physical-to-Structural Address Mapping (PSAM)*, records temporally-correlated physical addresses and their mapping information (i.e., to which location in *SAS* they are mapped); the other table, named *Structural-to-Physical Address Mapping (SPAM)*, keeps the *linearized* form of physical addresses in *SAS* and the corresponding mapping information (i.e., which physical addresses are mapped to every structural address). The main purpose of such a linearization is to represent the metadata in a spatially-located manner, paving the way to putting the metadata off-the-chip and *caching* its content in on-chip structures [14]. Like STMS, ISB puts its metadata information off-the-chip to save the precious SRAM storage; however, unlike STMS, ISB caches the content of its off-chip metadata tables in on-chip structures. Caching the metadata works for ISB as a result of the provided spatial locality, which is *not* the case for STMS. By caching the metadata information, ISB (1) provides faster access to metadata since the caches offer a high hit ratio, and it is not required to proceed to the off-chip memory for every metadata access, and (2) reduces the metadata-induced off-chip bandwidth overhead as many of the metadata manipulations coalesce in the on-chip caches. Figure 6 shows an overview of the metadata structures of ISB.

Another important contribution of ISB is the synchronization of off-chip metadata manipulations with Translation Lookaside Buffer (TLB) misses. That is, whenever a TLB miss occurs, concurrent with resolving the miss, ISB fetches the corresponding metadata information from the off-chip metadata tables; moreover, whenever a TLB entry is evicted, ISB evicts its corresponding entry from the on-chip metadata structures and updates the off-chip metadata tables. Doing so, ISB ensures that the required metadata is always present in the on-chip structures, significantly hiding the latency of off-chip memory accesses that would otherwise be exposed.

4.3 Spatial Prefetching

Spatial data prefetchers predict future memory accesses by relying on spatial address correlation, i.e., the similarity of access patterns among multiple *regions* of memory. Access patterns demonstrate spatial correlation because applications use data objects with a regular and fixed layout, and accesses reoccur when data structures are traversed [11]. Spatial data prefetchers [11, 15, 16, 65, 67, 87, 88,

Physical-to-Structural Address Mapping
(PSAM)

Physical Address	Structural Address
A	m
⋮	⋮
B	m+1
X	n

Structural-to-Physical Address Mapping
(SPAM)

Structural Address	Physical Address
m, m+1, …	A, B, …
⋮	⋮
n, n+1, …	X, Y, …
⋮	⋮

Fig. 6. The organization of Irregular Stream Buffer (ISB).

101, 105] divide the memory address space into fixed-size sections, named *Spatial Regions*, and learn the memory access patterns over these sections. The learned access patterns are then used for prefetching future memory references when the application touches the *same* or *similar Spatial Regions*.

Spatial data prefetchers impose low area overhead because they store *offsets* (i.e., the distance of a block address from the beginning of a *Spatial Region*) or *deltas* (i.e., the distance of two consecutive accesses that fall into a *Spatial Region*) as their metadata information, and not complete addresses. Another equally remarkable strength of spatial data prefetchers is their ability to eliminate compulsory cache misses. Compulsory cache misses are a major source of performance degradation in important classes of applications, e.g., scan-dominated workloads, where scanning large volumes of data produces a bulk of unseen memory accesses that cannot be captured by caches [11]. By utilizing the pattern that was observed in a past *Spatial Region* to a new unobserved *Spatial Region*, spatial prefetchers can alleviate the compulsory cache misses, significantly enhancing system performance.

The critical limitation of spatial data prefetching is its ineptitude in predicting pointer-chasing–caused cache misses. As dynamic objects can potentially be allocated everywhere in the memory, pointer-chasing accesses do not necessarily exhibit spatial correlation, producing bulks of dependent cache misses for which spatial prefetchers can do very little (cf. Section 4.2).

We include two state-of-the-art spatial prefetching techniques: (1) Spatial Memory Streaming [105], and (2) Variable Length Delta Prefetcher [101].

4.3.1 *Spatial Memory Streaming (SMS).*

SMS is a state-of-the-art spatial prefetcher that was proposed and evaluated in the context of server and scientific applications. Whenever a *Spatial Region* is requested for the first time, SMS starts to observe and record accesses to that *Spatial Region* as long as the *Spatial Region* is actively used by the application. Whenever the *Spatial Region* is no longer utilized (i.e., the corresponding blocks of the *Spatial Region* start to be evicted from the cache), SMS stores the information of the observed accesses in its metadata table, named *Pattern History Table (PHT)*.

The information in *PHT* is stored in the form of $\langle event, pattern \rangle$. The *event* is a piece of information to which the observed access pattern is correlated. That is, it is expected for the stored access pattern to be used whenever *event* reoccurs in the future. SMS empirically chooses *PC+Offset* of the trigger access (i.e., the PC of the instruction that first accesses the *Spatial Region* combined with the distance of the first requested cache block from the beginning of the *Spatial Region*) as the *event* to which the access patterns are correlated. Doing so, whenever a *PC+Offset* is reoccurred, the correlated access pattern history is used for issuing prefetch requests. The *pattern* is the history of accesses that happen in every *Spatial Region*. SMS encodes the patterns of the accesses as a *bit vector*. In this manner, for every cache block in a *Spatial Region*, a bit is stored, indicating whether the block has been used during the latest usage of the *Spatial Region* (`'1'`) or not (`'0'`).

Therefore, whenever a *pattern* is going to be used, prefetch requests are issued only for blocks whose corresponding bit in the stored *pattern* is '1.' Figure 7 shows the hardware realization of SMS.

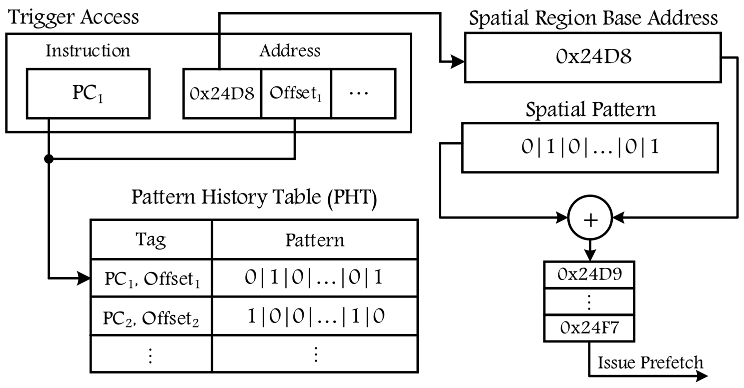

Fig. 7. The organization of Spatial Memory Streaming (SMS).

4.3.2 Variable Length Delta Prefetcher (VLDP). VLDP is a recent state-of-the-art spatial data prefetcher that relies on the similarity of *delta* patterns among *Spatial Regions* of memory. VLDP records the distance between consecutive accesses that fall into *Spatial Regions* and uses them to predict future misses. The key innovation of VLDP is the deployment of *multiple* prediction tables for predicting delta patterns. VLDP employs several history tables where each table keeps the metadata based on a specific length of the input history.

Figure 8 shows the metadata organization of VLDP. The three major components are *Delta History Buffer (DHB)*, *Delta Prediction Table (DPT)*, and *Offset Prediction Table (OPT)*. *DHB* is a small table that records the delta history of *currently-active Spatial Regions*. Each entry in *DHB* is associated with an active *Spatial Region* and contains details like the *Last Referenced Block*. These details are used to index *OPT* and *DPTs* for issuing prefetch requests.

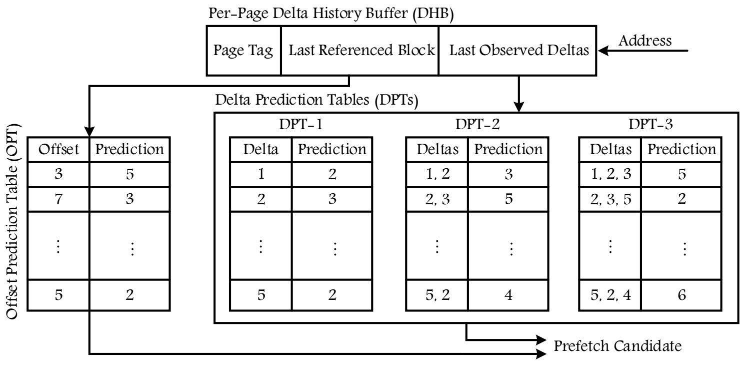

Fig. 8. The organization of Variable Length Delta Prefetcher (VLDP).

DPT is a set of key-value pairs that correlates a delta sequence to the next expected delta. VLDP benefits from multiple *DPTs* where each *DPT* records the history with a different length of the input. *DPT*$-i$ associates a sequence of i deltas to the next expected delta. For example, if the last three deltas in a *Spatial Region* are d_3, d_2, and d_1 (d_1 is the most recent delta), *DPT*-2 stores $[\langle d_3, d_2\rangle \longrightarrow d_1]$, while *DPT*-1 records $[\langle d_2\rangle \longrightarrow d_1]$. While looking up the *DPTs*, if several of them offer a prediction, the prediction of the table with the *longest* sequence of deltas is used, because predictions that are made based on longer inputs are expected to be more accurate [11]. This way, VLDP eliminates wrong predictions that are made by short inputs, enhancing both accuracy and miss coverage of the prefetcher.

OPT is another metadata table of VLDP, that is indexed using the *offset* (and not delta) of the first access to a *Spatial Region*. Merely relying on deltas for prefetching causes the prefetcher to need to observe at least first two accesses to a *Spatial Region* before issuing prefetch requests; however, there are many *sparse Spatial Regions* in which a few, say, two, of the blocks are used by the application. Therefore, waiting for two accesses before starting the prefetching may divest the prefetcher of issuing enough prefetch requests when the application operates on a significant number of sparse *Spatial Regions*. Employing *OPT* enables VLDP to start prefetching immediately after the first access to *Spatial Regions*. *OPT* associates the offset of the first access of a *Spatial Region* to the next expected delta. After the first access to a *Spatial Region*, *OPT* is looked up using the offset of the access, and the output of the table is used for issuing a prefetch request. For the rest of the accesses to the *Spatial Region* (i.e., second access onward), VLDP uses only *DPTs*.

Even though VLDP relies on prediction tables with a single next expected delta, it is still able to offer a prefetching lookahead larger than one (Section 3.2), using the proposed *multi-degree* prefetching mechanism. In the *multi-degree* mode, upon predicting the next delta in a *Spatial Region*, VLDP *uses the prediction as an input* for *DPTs* to make more predictions.

4.4 Spatio-Temporal Prefetching

Temporal and spatial prefetching techniques capture separate subsets of cache misses, and hence, each omits a considerable portion of cache misses unpredicted. As a considerable fraction of data misses is predictable only by one of the two prefetching techniques, spatio-temporal prefetching tries to combine them in order to reap the benefits of both methods. Another motivation for spatio-temporal prefetching is the fact that the effectiveness of temporal and spatial prefetching techniques varies across applications. As discussed, pointer-chasing application (e.g., *OLTP*) produce long chains of dependent cache misses which cannot be effectively captured by spatial prefetching but temporal prefetching. On the contrary, scan-dominated applications (e.g., *DSS*) produce a large number of compulsory cache misses that are predictable by spatial prefetchers and not temporal prefetchers.

We include Spatio-Temporal Memory Streaming (STeMS) [104], as it is the only proposal in this class of prefetching techniques.

STeMS synergistically integrates spatial and temporal prefetching techniques in a unified prefetcher; STeMS uses a temporal prefetcher to capture the stream of *trigger accesses* (i.e., the first access to each spatial region) and a spatial prefetcher to predict the expected misses *within* the spatial regions. The metadata organization of STeMS mainly consists of the metadata tables of STMS [110] and SMS [105]. STeMS, however, seeks to stream the sequence of cache misses *in the order they have been generated by the processor*, regardless of how the corresponding metadata information has been stored in the history tables of STMS and SMS. To do so, STeMS employs a *Reconstruction Buffer* which is responsible for reordering the prefetch requests generated by the temporal and the spatial prefetchers of STeMS so as to send prefetch requests (and deliver their responses) in the order the processor is supposed to consume them.

Table 1. Evaluation parameters.

Parameter	**Value**
Chip	14 nm, 4 GHz, 4 cores
Core	Out-of-order execution, 4-wide dispatch/retirement, 64-entry LSQ, 256-entry ROB
I-Fetch Unit	32 KB, 2-way, 2-cycle load-to-use, 24-entry pre-dispatch queue, Perceptron branch predictor [53]
L1-D Cache	32 KB, 2-way, 2-cycle load-to-use
L2 Cache	1 MB per core, 16-way, 11 cycle lookup delay, 128 MSHRs
Memory	240-cycle delay, 37.5 GB/s peak bandwidth, two memory controllers

For enabling the *reconstruction* process, the metadata tables of SMS and STMS are slightly modified. SMS is modified to record the order of the accessed cache blocks within a spatial region by encoding spatial patterns as ordered lists of offsets, stored in *Patterns Sequence Table (PST)*. Although *PST* is less compact than *PHT* (in the original SMS), the offset lists maintain the order required for accurately interleaving temporal and spatial streams. STMS is also modified and records only spatial triggers (and not all events as in STMS) in a *Region Miss Order Buffer (RMOB)*. Moreover, entries in both spatial and temporal streams are augmented with a *delta* field. The delta field in a spatial (temporal) stream represents the number of events from the temporal (spatial) stream that is interleaved between the current and next events of the same type. Figure 9 gives an example of how STeMS reconstructs the total miss order.

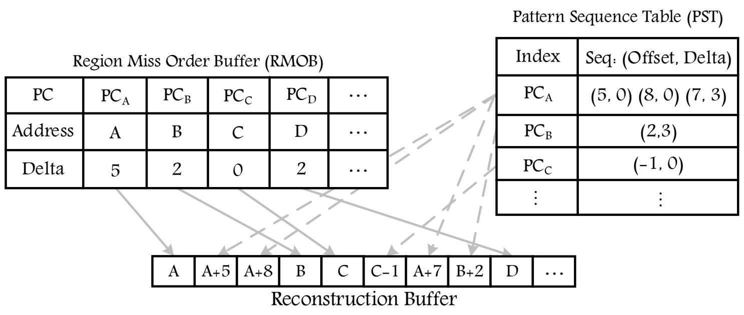
Region Miss Order Buffer (RMOB)
PC | PC_A | PC_B | PC_C | PC_D | ...
Address | A | B | C | D | ...
Delta | 5 | 2 | 0 | 2 | ...
Pattern Sequence Table (PST)
Index | Seq: (Offset, Delta)
PC_A | (5, 0) (8, 0) (7, 3)
PC_B | (2,3)
PC_C | (-1, 0)
A | A+5 | A+8 | B | C | C-1 | A+7 | B+2 | D | ...
Reconstruction Buffer

Fig. 9. The organization of Spatio-Temporal Memory Streaming (STeMS) and the reconstruction process.

5 METHODOLOGY

Table 1 summarizes the key elements of our methodology, with the following sections detailing the modeled platform, workloads, simulation infrastructure, and the configuration of the evaluated prefetchers.

5.1 CMP Parameters

Our platform, which is modeled after *Intel Xeon Processor™* [3], is a quad-core processor with 4 MB of last-level cache and two memory channels. Core microarchitecture includes 32 KB L1-D and L1-I caches. The cache block size is 64 bytes in the entire memory hierarchy. Two memory channels are located at the corner of the chip and provide up to 37.5 GB/s of off-chip bandwidth. We use *CACTI* [83] to estimate the delay of on-chip caches.

5.2 Workloads

We simulate systems running *Solaris* and executing the workloads listed in Table 2. We include a variety of server workloads from competing vendors, including *Online Transaction Processing*, *Decision Support System*, *Web Server*, and *CloudSuite* [1]. Prior work [36] has shown that these workloads have characteristics that are representative of the broad class of server workloads.

Table 2. Application parameters.

OLTP - Online Transaction Processing (TPC-C)	
DB2	IBM DB2 v8 ESE, 100 warehouses (10 GB), 2 GB buffer pool
DSS - Decision Support Systems (TPC-H)	
Qry 2 and 17	IBM DB2 v8 ESE, 480 MB buffer pool, 1 GB database
Web Server (SPECweb99)	
Apache	Apache HTTP server v2.0, 16 K connections, fastCGI, worker threading
Zeus	Zeus web server v4.3, 16 K connections, fastCGI
CloudSuite	
Data Serving	Cassandra 0.7.3 Database, 15 GB Yahoo! Cloud Serving Benchmark
MapReduce	Hadoop 0.20.2, Bayesian classification algorithm
Media Streaming	Darwin Streaming Server 6.0.3, 7500 Clients, 60 GB dataset, high bitrates
Web Search	Nutch 1.2/Lucene 3.0.1, 230 Clients, 1.4 GB Index, 15 GB Data Segment

5.3 Simulation Infrastructure

We use trace-driven simulations for evaluating the ability of each prefetcher at predicting future cache misses and timing simulations for performance studies. The trace-driven experiments use traces that are obtained from the in-order execution of applications with a fixed instruction-per-cycle (IPC) of 1.0. We run trace-driven simulations for 12 billion instructions and use the first half as the warmup and the rest for the actual measurements. We use *ChampSim* full-system simulator [2] for timing experiments. For every application, we create five checkpoints with warmed architectural components (e.g., branch predictors, caches, and prediction tables). Each checkpoint is drawn over an interval of 10 seconds of simulated time as observed by the Operating System (OS). We execute 200 K instructions from each checkpoint and use the first 40 K instructions for warm-up and the rest for measurements.

5.4 Prefetchers' Configurations

For every prefetcher, we do a sensitivity analysis to find the storage requirement for the prefetcher to have a reasonable miss coverage. For most of the prefetchers, we begin with an infinite storage and reduce the area until the miss coverage drops more than 5% as compared to its peak value; doing so, we ensure that every prefetcher is able to provide its maximum possible performance improvement, negating the limiting effect of dissimilar storage requirements of different prefetchers, enabling fair comparison among various approaches. The miss coverage of stride prefetching techniques (e.g., BOP), nonetheless, is not in its highest value when the prefetcher has infinite storage. The storage overhead of these approaches is directly tied to their other parameters (e.g., evaluation period): providing a larger storage budget for them does not necessarily result in a higher performance improvement. Therefore, for these prefetching techniques, we start with the configuration parameters suggested in the original proposal and modify (increase or decrease) them to get the highest possible miss coverage.

To have a fair comparison, we consider the following items in the implementation of the competing approaches:

- All prefetchers are trained on L1-D misses (LLC accesses). This way, each core has its own prefetcher and issues prefetch requests for itself, independent of others.
- Except for STMS and STeMS, other prefetching techniques directly prefetch into the primary data cache (L1-D). STMS and STeMS rely on temporal correlation of global misses and hence, are inaccurate (cf. Section 4.2). Consequently, streaming the prefetched data into the L1-D would significantly pollute the cache. For STMS and STeMS, we place the prefetched data in a small buffer next to L1-D cache. It is worth mentioning that the other prefetching techniques do *not* benefit from such an approach (i.e., prefetching into a small prefetch buffer). This is because of the fact that the other prefetching techniques do not prefetch the *next miss in time* and usually prefetch cache blocks that a program may need far from the current miss. Therefore, prefetching into a small prefetch buffer would result in the early eviction of the prefetched blocks, diminishing the performance.
- The prefetching lookahead of all methods, except SMS, is set to four, providing a sensible trade-off between the performance improvement and the off-chip bandwidth overhead. The prefetching lookahead of SMS depends on the recorded patterns and varies across regions. As SMS does *not* keep the order of prefetch candidates, it is not possible to enforce a predefined prefetching lookahead to it.

We simulate the competing data prefetchers with the following configurations. As a point of reference, we also include the *opportunity* results for temporal, spatial, and spatio-temporal data prefetching techniques.

5.4.1 Instruction-Based Stride Prefetcher (IBSP). We simulate IBSP with a 32-entry fully-associative *RPT*.

5.4.2 Best-Offset Prefetcher (BOP). A 64-entry fully-associative *RR Table* is used, and 46 offsets are evaluated based on the original proposal [79].

5.4.3 Sampled Temporal Memory Streaming (STMS). STMS uses a 6-million-entry *Index Table* and a 6-million-entry *History Table*. The *Index Table* is indexed by the lower bits of the trigger address. At any time, up to four concurrent streams are tracked for issuing prefetch requests. STMS requires minimal on-chip storage for tracking active streams and stores bulk of metadata in the main memory off the chip. *History Table* entries coalesce into a 64 B buffer and are flushed into the main memory when the buffer is filled. Upon reading/updating the *Index Table*, the corresponding entry is fetched into a small 64 B buffer and is written to the memory after the modification. STMS also uses four 64 B buffers, each dedicated to an active stream, for keeping the addresses of active streams for prefetching. Prefetched cache blocks are placed into a 2 KB prefetch buffer near the cache.

5.4.4 Irregular Stream Buffer (ISB). Two 4 K-entry on-chip structures are used to cache the content of two 3 M-entry off-chip metadata tables. Parts of the metadata that may be used by the processor are always in the on-chip structures.

5.4.5 Spatial Memory Streaming (SMS). Our sensitivity analysis demonstrates that 16 K-entry *PHT* is sufficient for SMS to reach the peak miss coverage. The memory space is divided into 2 KB *Spatial Regions*.

5.4.6 Variable Length Delta Prefetcher (VLDP). VLDP is equipped with a 16-entry *DHB*, 64-entry *OPT*, and three 128-entry fully-associative *DPTs*. The size of spatial regions is set to 2 KB.

5.4.7 Spatio-Temporal Memory Streaming (STeMS). STeMS uses a 2 M-entry *RMOB*, a 2 M-entry *Index Table*, and a 16 K-entry *PST*. The indexing scheme of metadata tables is exactly the same as that of STMS and SMS. STeMS uses the on-chip structure of STMS, with the same configuration, for reading and updating the *RMOB* and the *Index Table* entries. Additionally, STeMS leverages the *Filter Table* and the *Accumulation Table* of SMS, again with the same configuration, for dealing with the off-chip *PST*. Up to four streams are tracked concurrently. The *Reconstruction Buffer* of each active stream has 256 entries. The size of the prefetch buffer is set to 2 KB.

5.4.8 Temporal Opportunity (T-Opp). Like prior studies of measuring repetitiveness of access sequences [10, 20], we adopt the Sequitur hierarchical data compression algorithm [89] to recognize the opportunity for temporal prefetching using data miss sequences. Sequitur creates a grammar whose production rules resemble the repetitions in its input. Every production rule associates a tag to a string of tokens and other rule tags. Sequitur extends the grammar's root production rule by one symbol at a time, incrementally; when a symbol is added, the grammar is adjusted to form the new production rules. Therefore, Sequitur catches new repetitions that the added symbol creates. We compare temporal prefetchers against Sequitur to see what fraction of the opportunity they are able to cover.

5.4.9 Spatial Opportunity (S-Opp). Upon each cache miss to a new spatial region, the ideal spatial prefetcher issues prefetches for all the blocks that will be requested from that spatial region in the future. Hence, with an ideal spatial prefetcher, the only access that may miss in the cache is the first access to a spatial region. Regarding the data supply, an ideal spatial prefetcher performs like a cache whose block size is equal to the size of a spatial region (i.e., 2 KB), but the number of blocks in the cache remains unchanged.

5.4.10 Spatio-Temporal Opportunity (ST-Opp). Ideal spatio-temporal data prefetcher eliminates all misses that can be captured by either the ideal spatial data prefetcher or the ideal temporal data prefetcher. To measure the opportunity, we first eliminate all misses that can be captured by an ideal spatial data prefetcher. Then, we run the Sequitur algorithm on the remaining misses to eliminate those that can be captured by an ideal temporal prefetcher.

6 EVALUATION RESULTS

We run trace-based simulations for profiling and miss coverage studies and detailed cycle-accurate timing simulations for performance experiments.

6.1 Coverage and Accuracy

To compare the effectiveness of prefetching techniques, Figure 10 shows the coverage and overprediction of the competing prefetching techniques. Covered misses are the ones that are successfully eliminated by a prefetcher. Overpredictions are wrongly prefetched cache blocks, which cause bandwidth overhead and potentially pollute the cache or prefetch buffer. The wrong prefetches are normalized against the number of cache misses in the baseline system with no data prefetcher.

Corroborating prior work [8, 10, 36], stride prefetchers are not effective at capturing a significant fraction of data misses of server workloads. This is because of the fact that server applications use complex data structures and pointer-chasing code, and hence, do not exhibit significant strided access patterns. IBSP rarely issues prefetch requests, resulting in low miss coverage and low overprediction rate. BOP is more aggressive and consequently, covers more misses at the cost of a higher overprediction rate.

Our results show that STMS outperforms ISB regarding both miss coverage and the overprediction. We find that *PC localization* is the main source of the inefficiency of ISB. ISB correlates temporal streams with load instructions and aggressively issues prefetch requests for the future references of the load instructions. We find that this mechanism suffers from two main obstacles: (1) PC-localization breaks the strong temporal correlation among global addresses that is dominant in server workloads (see the results of T-Opp), and (2) PC-localization prefetches the next misses of a load instruction, which may not be the next misses of the program. As server workloads have large instruction working sets (in the range of few megabytes [36]), the re-execution of a specific load instruction in the execution order may take a long time; hence, the prefetched cache blocks of a PC-localized temporal prefetcher may be evicted from the cache due to conflicts, prior to the re-execution of the load instruction. While STMS captures more cache misses than ISB, it falls short of covering the entire temporal opportunity. The gap between STMS and T-Opp ranges from 14% in Web Search to 38% in DSS Qry2 with an average of 21%.

SMS covers more cache misses than VLDP with lower overprediction rate. Our investigations show that the *multi-degree* prefetching mechanism (i.e., increasing the prefetching lookahead beyond one) of VLDP is the most contributor to its large overprediction rate. Once VLDP has predicted the next access of a spatial region, it makes more prefetches using the prediction as an input to the metadata tables. We find that this strategy is inaccurate for server workloads, and gets worse as VLDP further repeats this process. With all this, both SMS and VLDP are far from covering the opportunity of spatial prefetching: on average, the best-performing spatial prefetcher covers less than 43% of the spatial opportunity, leaving a significant fraction of spatially-predictable misses unpredicted.

Except for MapReduce, which exhibits labyrinthine access patterns, an ideal spatio-temporal prefetcher is able to eliminate more than three-fourths of cache misses. While there is a high spatio-temporal opportunity, STeMS covers only a small fraction of it, with a considerable overprediction rate. Despite high overhead (i.e., large tables and bandwidth overhead), STeMS covers only 28% of the opportunity, making it quite ineffective.

6.2 Cycle-Accurate Evaluation

Figure 11 shows the performance improvement of the competing prefetchers over a baseline with no data prefetcher. As a result of the moderate coverage, stride prefetchers are unable to significantly boost the performance. Among stride prefetchers, BOP offers a higher performance improvement, primarily because of being more aggressive.

STMS outperforms ISB thanks to its higher coverage and accuracy. However, STMS suffers from a high start-up cost, as the first prefetch request is sent after two memory round-trip latency. This is why notwithstanding offering high miss coverage, STMS is unable to significantly boost the performance of applications that are dominated with short temporal streams (e.g., Web Search).

SMS outperforms VLDP due to its higher coverage and lower overprediction rate. While SMS and VLDP are effective at boosting the performance in most of the evaluated workloads, they offer little benefits for the MapReduce workload. We find that most of the misses that SMS and VLDP cover in this workload are independent misses, and hence, they are already fetched in parallel with the out-of-order execution.

The performance improvement of STeMS ranges from 1% in DSS Qry2 to 18% in Data Serving. Like STMS, the metadata of STeMS is located off-the-chip, and hence, STeMS suffers from a high start-up latency. STeMS can start issuing prefetch requests only after three serial memory round-trip access latencies. This makes STeMS ineffective, especially for applications that are dominated by *sparse* spatial regions (i.e., regions that have few predicted blocks). In such regions, the start-up latency of prefetching is not compensated because only few cache blocks are prefetched after waiting a long time for the metadata to arrive.

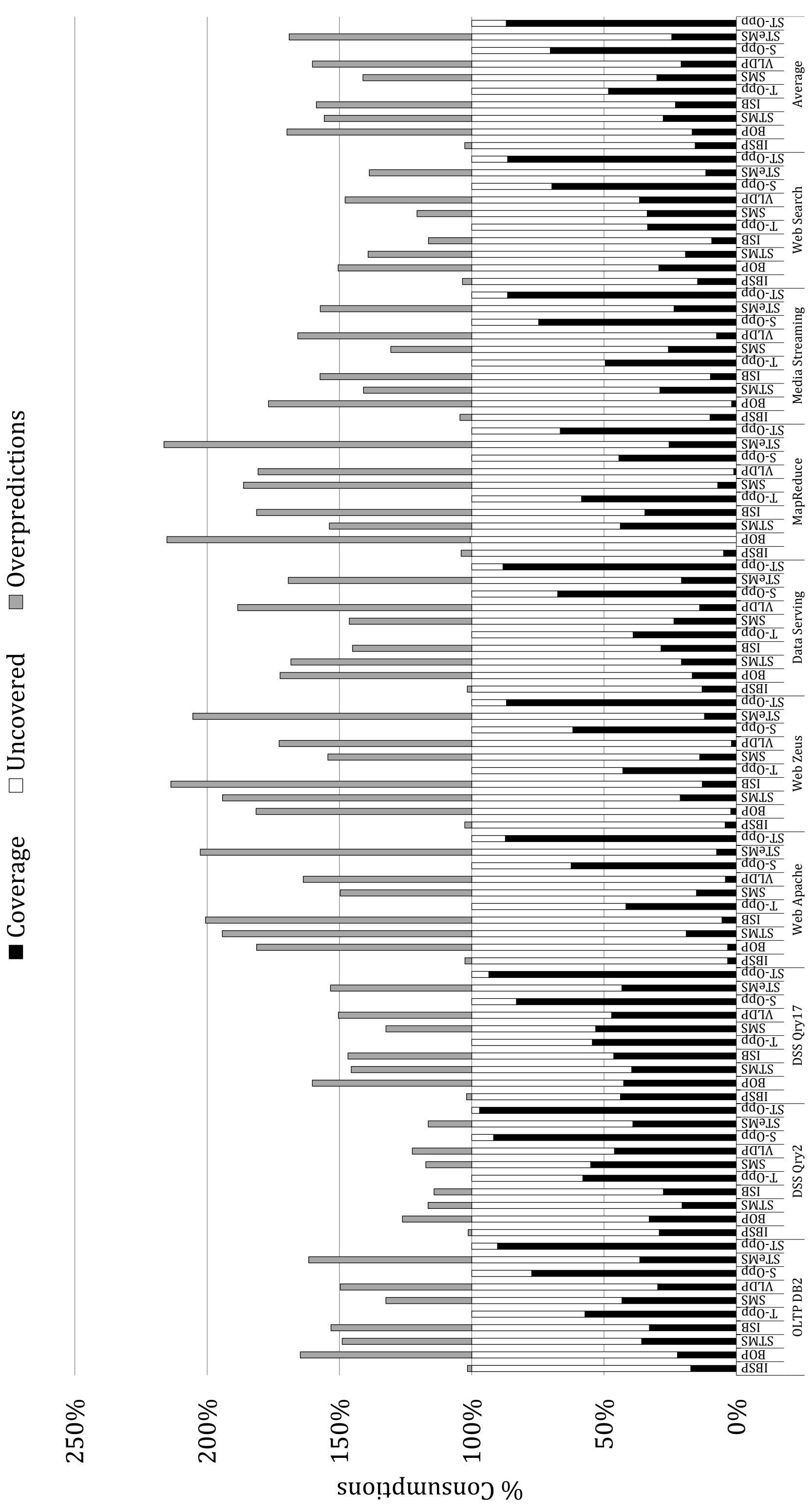

Fig. 10. Coverage and overprediction of the evaluated prefetchers.

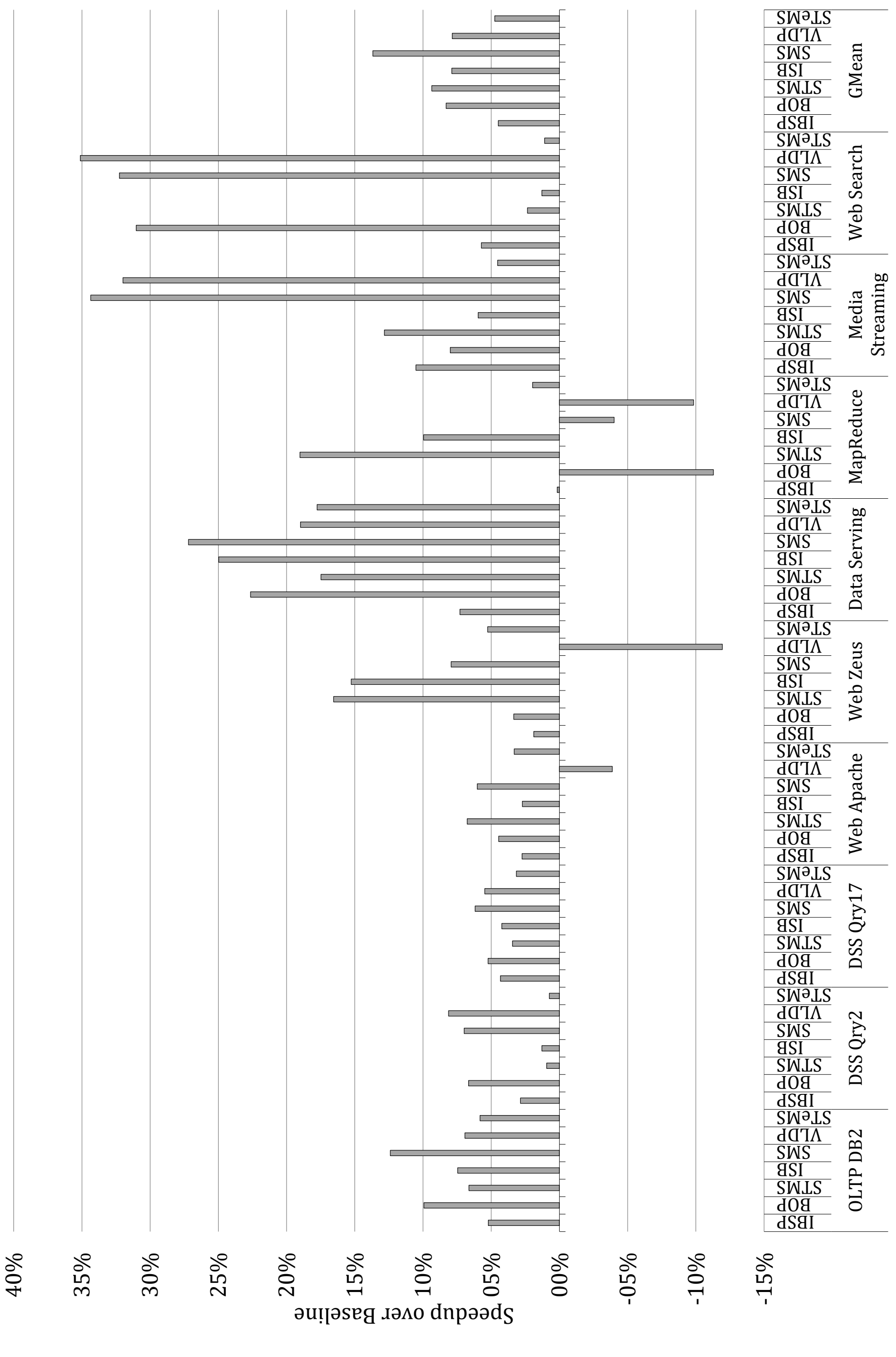

Fig. 11. The performance improvement of various prefetching approaches.

6.3 Storage Requirement

The effectiveness of prefetchers depends on the size of the history on which the predictions are made. Figure 12 shows the on-chip and off-chip storage requirements of the evaluated prefetchers. Obviously, the storage requirement of temporal prefetching techniques is an order of magnitude higher than the other prefetching techniques. The high storage requirement necessitates moving the metadata tables to the off-chip memory. The metadata storage of temporal prefetchers is determined by the application's active data working set and is pretty large. Transferring the metadata tables to the main memory solves the problem of area limitation because the multi-megabyte storage requirement of the temporal prefetchers accounts for only a small portion of the available capacity in today's multi-gigabyte main memories.

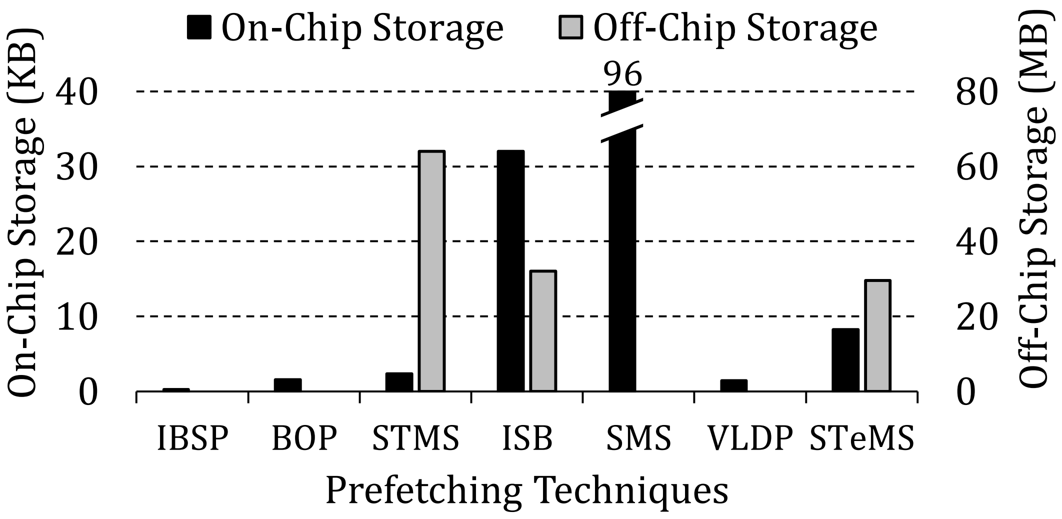

Fig. 12. Storage overhead of evaluated data prefetchers.

Prefetchers that do not rely on temporal correlations impose less area overhead, as they need less metadata to achieve high coverage. Stride prefetchers need minimal area for metadata, as they do not store any long-term history and just track several active streams/offsets. Spatial prefetchers need to store the memory access patterns of different spatial regions. VLDP stores the required metadata in a compressed manner by recording the *delta* of two consecutive accesses to a spatial region instead of full addresses. SMS, on the other hand, correlates the observed patterns to the program code, which leads to growing the metadata storage requirement with the code size.

6.4 Off-Chip Bandwidth Overhead

The increase in core count has been driving the designs into memory bandwidth wall mainly because of poor pin-count scalability [7, 12, 47, 52, 94]. As a result, prefetching techniques, for being effective in the context of multi-core and many-core systems, are required to use minimal off-chip bandwidth.

The bandwidth usage of the baseline system varies from one workload to another, ranging from 1.6 GB/s in MapReduce to 7.7 GB/s in Apache. The average bandwidth utilization is 12% (4.5 GB/s) across all workloads. Corroborating prior work [10, 36], our studies show that due to low instruction- and memory-level parallelisms, server workloads consume only a small fraction of off-chip bandwidth available in today's commercial processors; hence, there is an abundant underutilized off-chip bandwidth, which can be used by data prefetchers.

Figure 13 shows the bandwidth usage of various prefetching techniques averaged across all workloads. The bandwidth requirement of prefetching techniques mainly depends on the accuracy of the prefetcher and the metadata fetch/update rate if the metadata is located off-the-chip. Among the evaluated data prefetchers, STMS and STeMS require an order of magnitude higher off-chip bandwidth, making them unscalable for many-core processors. The majority of the traffic overhead of these two techniques comes from their costly metadata fetches and updates. ISB, on the other hand, linearizes the correlation tables and synchronizes the fetch-and-update requests with TLB replacements, resulting in a drastic reduction in off-chip bandwidth usage. For prefetchers with on-chip metadata, the off-chip bandwidth overhead is due to (1) wrong prefetch requests that proceed to the off-chip memory, and (2) extra cache misses caused by wrong prefetches that are served from the off-chip main memory. IBSP is extremely conservative and rarely issues prefetch requests and only imposes a negligible memory traffic overhead. SMS and VLDP rely on spatial correlation, and hence, are expected to be more accurate and foist less traffic overhead. However, VLDP produces a higher traffic overhead (as compared to SMS), mainly because of issuing many wrong prefetches, which itself comes from inefficiencies in its multi-degree prefetching mechanism (cf. Section 6.1).

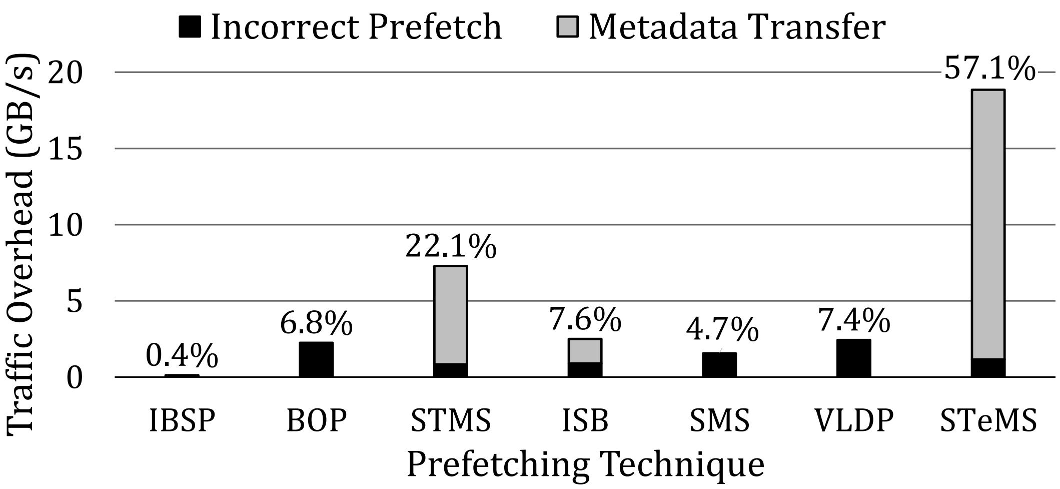

Fig. 13. Bandwidth overhead of evaluated data prefetchers.

7 COMPLEMENTARY WORK

There has also been a large body of work on increasing the efficiency of hardware data prefetching. Stream Chaining [27] is a general mechanism to connect the mutually-localized sequences of cache misses to form a long sequence of successful prefetch requests, thereby increasing the timeliness and the accuracy of prefetch requests. Predictor Virtualization [14] is a mechanism to store the metadata of data prefetchers in existing on-chip caches, eliminating the necessity of having large dedicated storage. Controlling the aggressiveness of prefetchers [32, 48, 54, 106] or disabling them at run-time [58, 60, 113] as strategies to cope with prefetching hazards have also been widely studied in the literature. Moreover, many pieces of prior work target reducing the delay of the interconnection network that connects cores to cache banks in server processors [17, 73, 75, 76]. Such techniques significantly improve the timeliness of prefetcher requests.

Many pieces of prior work also proposed adjustments to the other system components in order to make the prefetching more efficient. PACMan [112] and ICP [99] manage the last-level cache in the presence of a hardware data prefetcher, trying to reduce prefetch-induce cache interferences. Reforming memory scheduling policies so as to provide fairness [30] and/or prevent bandwidth pollution [69] have also been researched by prior work.

8 CONCLUSION AND FUTURE OUTLOOK

Data prefetching has been an attractive area of research in the past four decades; as a result of a large volume of research emphasizing on its importance, data prefetcher has become an inextricable component of modern high-performance processors.

In this paper, we presented a survey along with an evaluation of state-of-the-art data prefetching techniques. Specifically, we identified the limitations of the state-of-the-art prefetchers, encouraging further research in this area. We showed that there is a large gap between what state-of-the-art prefetchers offer and the opportunity; one direction for future work is bridging the gap between the best-performing prefetchers' performance and the ideal opportunity. Moreover, we showed that some of the prefetchers impose large area and/or off-chip bandwidth overheads; future work may target mitigating these overheads, paving the way for using such prefetchers in area- and/or bandwidth-constrained systems.

ACKNOWLEDGMENT

We thank Mehran Shakerinava of HPCAN-Sharif for his help in conducting some of the experiments of this work. We thank ACM CSUR's reviewers for their valuable comments and suggestions. This work is supported in part by a grant from Iran National Science Foundation (INSF).

REFERENCES

[1] CloudSuite. Available at http://cloudsuite.ch, 2012.
[2] ChampSim. https://github.com/ChampSim/, 2017.
[3] Intel® Xeon® Processor E3-1245 v6. Available at https://www.intel.com/content/www/us/en/products/processors/xeon/e3-processors/e3-1245-v6.html, 2017.
[4] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a Modern Processor: Where Does Time Go? In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 266–277, 1999.
[5] H. Akkary and M. A. Driscoll. A Dynamic Multithreading Processor. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 226–236. IEEE, 1998.
[6] J.-L. Baer and T.-F. Chen. An Effective On-chip Preloading Scheme to Reduce Data Access Penalty. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 176–186, 1991.
[7] M. Bakhshalipour, A. Faraji, S. A. V. Ghahani, F. Samandi, P. Lotfi-Kamran, and H. Sarbazi-Azad. Reducing Writebacks Through In-Cache Displacement. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 24(2):16, 2019.
[8] M. Bakhshalipour, P. Lotfi-Kamran, A. Mazloumi, F. Samandi, M. Naderan-Tahan, M. Modarressi, and H. Sarbazi-Azad. Fast Data Delivery for Many-Core Processors. *IEEE Transactions on Computers (TC)*, 67(10):1416–1429, 2018.
[9] M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad. An Efficient Temporal Data Prefetcher for L1 Caches. *IEEE Computer Architecture Letters (CAL)*, 16(2):99–102, 2017.
[10] M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad. Domino Temporal Data Prefetcher. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 131–142. IEEE, 2018.
[11] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad. Bingo Spatial Data Prefetcher. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2019.
[12] M. Bakhshalipour, H. Zare, P. Lotfi-Kamran, and H. Sarbazi-Azad. Die-Stacked DRAM: Memory, Cache, or MemCache? *arXiv preprint arXiv:1809.08828*, 2018.
[13] B. H. Bloom. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Commun. ACM*, 13(7):422–426, July 1970.

[14] I. Burcea, S. Somogyi, A. Moshovos, and B. Falsafi. Predictor Virtualization. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 157–167, 2008.

[15] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Stealth Prefetching. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 274–282, 2006.

[16] C. F. Chen, S.-H. Yang, B. Falsafi, and A. Moshovos. Accurate and Complexity-Effective Spatial Pattern Prediction. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pages 276–287, 2004.

[17] C.-H. O. Chen, S. Park, T. Krishna, S. Subramanian, A. P. Chandrakasan, and L.-S. Peh. SMART: A Single-cycle Reconfigurable NoC for SoC Applications. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 338–343, Mar. 2013.

[18] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving Hash Join Performance Through Prefetching. *ACM Transactions on Database Systems (TODS)*, 32(3), Aug. 2007.

[19] T. M. Chilimbi. On the Stability of Temporal Data Reference Profiles. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 151–160, 2001.

[20] T. M. Chilimbi and M. Hirzel. Dynamic Hot Data Stream Prefetching for General-Purpose Programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 199–209, 2002.

[21] Y. Chou. Low-Cost Epoch-Based Correlation Prefetching for Commercial Applications. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 301–313, 2007.

[22] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen. Dynamic Speculative Precomputation. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 306–317, 2001.

[23] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen. Speculative Precomputation: Long-Range Prefetching of Delinquent Loads. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 14–25, 2001.

[24] P. Conway and B. Hughes. The AMD Opteron Northbridge Architecture. *IEEE Micro*, 27(2):10–21, Mar. 2007.

[25] H. Cui, J. Wu, C.-C. Tsai, and J. Yang. Stable Deterministic Multithreading Through Schedule Memoization. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 207–221. USENIX Association, 2010.

[26] F. Dahlgren and P. Stenstrom. Effectiveness of Hardware-Based Stride and Sequential Prefetching in Shared-memory Multiprocessors. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pages 68–, 1995.

[27] P. Diaz and M. Cintra. Stream Chaining: Exploiting Multiple Levels of Correlation in Data Prefetching. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 81–92, 2009.

[28] J. Doweck. Inside Intel® Core Microarchitecture. In *IEEE Hot Chips Symposium (HCS)*, pages 1–35, 2006.

[29] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt. Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 335–346, 2010.

[30] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt. Prefetch-Aware Shared Resource Management for Multi-Core Systems. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 141–152, 2011.

[31] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt. Coordinated Control of Multiple Prefetchers in Multi-Core Systems. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 316–326, 2009.

[32] E. Ebrahimi, O. Mutlu, and Y. N. Patt. Techniques for Bandwidth-Efficient Prefetching of Linked Data Structures in Hybrid Prefetching Systems. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pages 7–17, 2009.

[33] H. A. Esfeden, F. Khorasani, H. Jeon, D. Wong, and N. Abu-Ghazaleh. CORF: Coalescing Operand Register File for GPUs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2019.

[34] P. Esmaili-Dokht, M. Bakhshalipour, B. Khodabandeloo, P. Lotfi-Kamran, and H. Sarbazi-Azad. Scale-Out Processors & Energy Efficiency. *arXiv preprint arXiv:1808.04864*, 2018.

[35] B. Falsafi and T. F. Wenisch. *A Primer on Hardware Prefetching*. Morgan & Claypool Publishers, 2014.

[36] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the Clouds: A Study of Emerging Scale-Out Workloads on Modern Hardware. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 37–48, 2012.

[37] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Quantifying the Mismatch Between Emerging Scale-Out Applications and Modern Processors. *ACM Transactions on Computer Systems (TOCS)*, 30(4):15:1–15:24, Nov. 2012.

[38] I. Ganusov and M. Burtscher. Future Execution: A Prefetching Mechanism That Uses Multiple Cores to Speed Up Single Threads. *ACM Transactions on Architecture and Code Optimization (TACO)*, 3(4):424–449, Dec. 2006.

[39] B. Grot, D. Hardy, P. Lotfi-Kamran, C. Nicopoulos, Y. Sazeides, and B. Falsafi. Optimizing Data-Center TCO with Scale-Out Processors. *IEEE Micro*, 32(5):1–63, Sept. 2012.

[40] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding Sources of Inefficiency in General-Purpose Chips. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 37–47. ACM, 2010.

[41] R. A. Hankins, T. Diep, M. Annavaram, B. Hirano, H. Eri, H. Nueckel, and J. P. Shen. Scaling and Characterizing Database Workloads: Bridging the Gap Between Research and Practice. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 116–120, 2003.

[42] N. Hardavellas, I. Pandis, R. Johnson, N. G. Mancheril, A. Ailamaki, and B. Falsafi. Database Servers on Chip Multiprocessors: Limitations and Opportunities. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 79–87, 2007.

[43] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, A. Gara, G. Chiu, P. Boyle, N. Chist, and C. Kim. The IBM Blue Gene/Q Compute Chip. *IEEE Micro*, 32(2):48–60, 2012.

[44] M. Hashemi, Khubaib, E. Ebrahimi, O. Mutlu, and Y. N. Patt. Accelerating Dependent Cache Misses with an Enhanced Memory Controller. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 444–455, 2016.

[45] T. Horel and G. Lauterbach. UltraSPARC-III: Designing Third-Generation 64-bit Performance. *IEEE Micro*, 19(3):73–85, 1999.

[46] C. J. Hughes and S. V. Adve. Memory-Side Prefetching for Linked Data Structures for Processor-in-Memory Systems. *Journal of Parallel and Distributed Computing*, 65(4):448–463, Apr. 2005.

[47] J. Huh, D. Burger, and S. W. Keckler. Exploring the Design Space of Future CMPs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 199–210, 2001.

[48] I. Hur and C. Lin. Memory Prefetching Using Adaptive Stream Detection. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 397–408, 2006.

[49] S. Iacobovici, L. Spracklen, S. Kadambi, Y. Chou, and S. G. Abraham. Effective Stream-Based and Execution-Based Data Prefetching. In *Proceedings of the International Conference on Supercomputing (ICS)*, pages 1–11, 2004.

[50] Y. Ishii, M. Inaba, and K. Hiraki. Access Map Pattern Matching for Data Cache Prefetch. In *Proceedings of the International Conference on Supercomputing (ICS)*, pages 499–500, 2009.

[51] A. Jain and C. Lin. Linearizing Irregular Memory Accesses for Improved Correlated Prefetching. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 247–259, 2013.

[52] D. Jevdjic, S. Volos, and B. Falsafi. Die-Stacked DRAM Caches for Servers: Hit Ratio, Latency, or Bandwidth? Have It All with Footprint Cache. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 404–415, 2013.

[53] D. A. Jiménez and C. Lin. Dynamic Branch Prediction with Perceptrons. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 197–206, 2001.

[54] V. Jiménez, R. Gioiosa, F. J. Cazorla, A. Buyuktosunoglu, P. Bose, and F. P. O'Connell. Making Data Prefetch Smarter: Adaptive Prefetching on POWER7. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 137–146, 2012.

[55] R. Johnson, S. Harizopoulos, N. Hardavellas, K. Sabirli, I. Pandis, A. Ailamaki, N. G. Mancheril, and B. Falsafi. To Share or Not to Share? In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 351–362, 2007.

[56] D. Joseph and D. Grunwald. Prefetching Using Markov Predictors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 252–263, 1997.

[57] N. P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 364–373, 1990.

[58] D. Kadjo, J. Kim, P. Sharma, R. Panda, P. Gratz, and D. Jimenez. B-Fetch: Branch Prediction Directed Prefetching for Chip-Multiprocessors. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 623–634, 2014.

[59] M. Kamruzzaman, S. Swanson, and D. M. Tullsen. Inter-Core Prefetching for Multicore Processors Using Migrating Helper Threads. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 393–404, 2011.

[60] M. Kandemir, Y. Zhang, and O. Ozturk. Adaptive Prefetching for Shared Cache Based Chip Multiprocessors. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 773–778, 2009.

[61] T. S. Karkhanis and J. E. Smith. A First-Order Superscalar Processor Model. pages 338–349, 2004.

[62] M. Kayaalp, K. N. Khasawneh, H. A. Esfeden, J. Elwell, N. Abu-Ghazaleh, D. Ponomarev, and A. Jaleel. RIC: Relaxed Inclusion Caches for Mitigating LLC Side-Channel Attacks. In *Proceedings of the Design Automation Conference (DAC)*, pages 7:1–7:6. ACM, 2017.

[63] F. Khorasani, H. A. Esfeden, N. Abu-Ghazaleh, and V. Sarkar. In-Register Parameter Caching for Dynamic Neural Nets with Virtual Persistent Processor Specialization. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 377–389. IEEE, 2018.

[64] F. Khorasani, H. A. Esfeden, A. Farmahini-Farahani, N. Jayasena, and V. Sarkar. RegMutex: Inter-Warp GPU Register Time-Sharing. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 816–828. IEEE Press, 2018.

[65] J. Kim, S. H. Pugsley, P. V. Gratz, A. L. N. Reddy, C. Wilkerson, and Z. Chishti. Path Confidence Based Lookahead Prefetching. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 60:1–60:12, 2016.

[66] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter. ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, 2010.

[67] S. Kumar and C. Wilkerson. Exploiting Spatial Locality in Data Caches Using Spatial Footprints. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 357–368, 1998.

[68] J. R. Larus and M. Parkes. Using Cohort-Scheduling to Enhance Server Performance. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference (ATEC)*, pages 103–114, 2002.

[69] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt. Prefetch-Aware DRAM Controllers. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 200–209, 2008.

[70] J. Lee, C. Jung, D. Lim, and Y. Solihin. Prefetching with Helper Threads for Loosely Coupled Multiprocessor Systems. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 20(9):1309–1324, Sept. 2009.

[71] K. Lim, P. Ranganathan, J. Chang, C. Patel, T. Mudge, and S. Reinhardt. Understanding and Designing New Server Architectures for Emerging Warehouse-Computing Environments. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 315–326, 2008.

[72] J. L. Lo, L. A. Barroso, S. J. Eggers, K. Gharachorloo, H. M. Levy, and S. S. Parekh. An Analysis of Database Workload Performance on Simultaneous Multithreaded Processors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 39–50, 1998.

[73] P. Lotfi-Kamran, B. Grot, and B. Falsafi. NOC-Out: Microarchitecting a Scale-Out Processor. In *Proceedings of the 45th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pages 177–187, Dec. 2012.

[74] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer, and B. Falsafi. Scale-Out Processors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 500–511, 2012.

[75] P. Lotfi-Kamran, M. Modarressi, and H. Sarbazi-Azad. An Efficient Hybrid-Switched Network-on-Chip for Chip Multiprocessors. *IEEE Transactions on Computers*, 65(5):1656–1662, May 2016.

[76] P. Lotfi-Kamran, M. Modarressi, and H. Sarbazi-Azad. Near-Ideal Networks-on-Chip for Servers. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 277–288, 2017.

[77] C.-K. Luk and T. C. Mowry. Compiler-Based Prefetching for Recursive Data Structures. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 222–233, 1996.

[78] S. Mehta, Z. Fang, A. Zhai, and P.-C. Yew. Multi-Stage Coordinated Prefetching for Present-Day Processors. In *Proceedings of the International Conference on Supercomputing (ICS)*, pages 73–82, 2014.

[79] P. Michaud. Best-Offset Hardware Prefetching. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pages 469–480, 2016.

[80] A. Mirhosseini and T. F. Wenisch. The Queuing-First Approach for Tail Management of Interactive Services. *IEEE Micro*, 2019.

[81] A. Mirhosseini, A. Sriraman, and T. F. Wenisch. Enhancing Server Efficiency in the Face of Killer Microseconds. *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2019.

[82] S. Mittal. A Survey of Recent Prefetching Techniques for Processor Caches. *ACM Computing Surveys (CSUR)*, 49(2):35:1–35:35, Aug. 2016.

[83] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 3–14, 2007.

[84] O. Mutlu, H. Kim, and Y. N. Patt. Techniques for Efficient Processing in Runahead Execution Engines. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 370–381, 2005.

[85] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors. In *Proceedings of the International Symposium on High Performance Computer*

Architecture (HPCA), pages 129–, 2003.

[86] M. Nemirovsky and D. M. Tullsen. *Multithreading Architecture.* Morgan & Claypool Publishers, 1st edition, 2013.

[87] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith. AC/DC: An Adaptive Data Cache Prefetcher. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 135–145, 2004.

[88] K. J. Nesbit and J. E. Smith. Data Cache Prefetching Using a Global History Buffer. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pages 96–, 2004.

[89] C. G. Nevill-Manning and I. H. Witten. Identifying Hierarchical Structure in Sequences: A Linear-time Algorithm. *Journal of Artificial Intelligence Research*, 7(1):67–82, Sept. 1997.

[90] 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, 1994.

[91] S. Pugsley, A. Alameldeen, C. Wilkerson, and H. Kim. The 2nd Data Prefetching Championship (DPC-2), 2015.

[92] 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 *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pages 626–637, 2014.

[93] P. Ranganathan, K. Gharachorloo, S. V. Adve, and L. A. Barroso. Performance of Database Workloads on Shared-Memory Systems with Out-of-Order Processors. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 307–318, 1998.

[94] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin. Scaling the Bandwidth Wall: Challenges in and Avenues for CMP Scaling. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 371–382, 2009.

[95] A. Roth and G. S. Sohi. Effective Jump-Pointer Prefetching for Linked Data Structures. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 111–121, 1999.

[96] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 73–82. ACM, 2008.

[97] M. Sadrosadati, A. Mirhosseini, S. B. Ehsani, H. Sarbazi-Azad, M. Drumond, B. Falsafi, R. Ausavarungnirun, and O. Mutlu. LTRF: Enabling High-Capacity Register Files for GPUs via Hardware/Software Cooperative Register Prefetching. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 489–502. ACM, 2018.

[98] S. Sair, T. Sherwood, and B. Calder. A Decoupled Predictor-Directed Stream Prefetching Architecture. *IEEE Transactions on Computers (TC)*, 52(3):260–276, Mar. 2003.

[99] V. Seshadri, S. Yedkar, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. Mitigating Prefetcher-Caused Pollution Using Informed Caching Policies for Prefetched Blocks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(4):51:1–51:22, Jan. 2015.

[100] T. Sherwood, S. Sair, and B. Calder. Predictor-Directed Stream Buffers. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 42–53, 2000.

[101] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti. Efficiently Prefetching Complex Address Patterns. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 141–152, 2015.

[102] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu. Knights Landing: Second-Generation Intel Xeon Phi Product. *IEEE Micro*, 36(2):34–46, Mar. 2016.

[103] Y. Solihin, J. Lee, and J. Torrellas. Using a User-Level Memory Thread for Correlation Prefetching. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 171–182, 2002.

[104] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi. Spatio-Temporal Memory Streaming. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 69–80, 2009.

[105] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Spatial Memory Streaming. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 252–263, 2006.

[106] 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 International Symposium on High Performance Computer Architecture (HPCA)*, pages 63–74, 2007.

[107] J. M. Tendler, J. S. Dodson, J. Fields, H. Le, and B. Sinharoy. POWER4 System Microarchitecture. *IBM Journal of Research and Development*, 46(1):5–25, 2002.

[108] P. Trancoso, J.-L. Larriba-Pey, Z. Zhang, and J. Torrellas. The Memory Performance of DSS Commercial Workloads in Shared-Memory Multiprocessors. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pages 250–260, 1997.

[109] A. Vakil-Ghahani, S. Mahdizadeh-Shahri, M.-R. Lotfi-Namin, M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad. Cache Replacement Policy Based on Expected Hit Count. *IEEE Computer Architecture Letters (CAL)*, 17(1):64–67, 2018.

[110] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos. Practical Off-Chip Meta-Data for Temporal Memory Streaming. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pages 79–90, 2009.

[111] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi. Temporal Streaming of Shared Memory. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 222–233, 2005.

[112] C.-J. Wu, A. Jaleel, M. Martonosi, S. C. Steely, Jr., and J. Emer. PACMan: Prefetch-Aware Cache Management for High Performance Caching. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 442–453, 2011.

[113] C.-J. Wu and M. Martonosi. Characterization and Dynamic Mitigation of Intra-Application Cache Interference. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 2–11, 2011.

[114] W. A. Wulf and S. A. McKee. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, Mar. 1995.

[115] P. Yedlapalli, J. Kotra, E. Kultursay, M. Kandemir, C. R. Das, and A. Sivasubramaniam. Meeting Midway: Improving CMP Performance with Memory-Side Prefetching. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 289–298, 2013.

[116] C. Zhang and S. A. McKee. Hardware-Only Stream Prefetching and Dynamic Access Ordering. In *Proceedings of the International Conference on Supercomputing (ICS)*, pages 167–175, 2000.

[117] W. Zhang, B. Calder, and D. M. Tullsen. A Self-Repairing Prefetcher in an Event-Driven Dynamic Optimization Framework. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 50–64, 2006.

Received August 2017; revised December 2018; accepted February 2019