A Survey of Recent Prefetching Techniques for Processor Caches

SPARSH MITTAL, Oak Ridge National Laboratory

As the trends of process scaling make memory systems an even more crucial bottleneck, the importance of latency hiding techniques such as prefetching grows further. However, naively using prefetching can harm performance and energy efficiency and, hence, several factors and parameters need to be taken into account to fully realize its potential. In this article, we survey several recent techniques that aim to improve the implementation and effectiveness of prefetching. We characterize the techniques on several parameters to highlight their similarities and differences. The aim of this survey is to provide insights to researchers into working of prefetching techniques and spark interesting future work for improving the performance advantages of prefetching even further.

Categories and Subject Descriptors: A.1 [**General Literature**]: Introductory and Survey; H.3.4 [**Systems and Software**]: Performance Evaluation (Efficiency and Effectiveness); C.0 [**Computer Systems Organization**]: System Architectures

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Review, classification, data prefetching, instruction prefetching, hardware (HW) prefetching, software (SW) prefetching, speculative pre-execution, helper thread prefetching, cache pollution

ACM Reference Format:
Sparsh Mittal. 2016. A survey of recent prefetching techniques for processor caches. ACM Comput. Surv. 49, 2, Article 35 (August 2016), 35 pages.
DOI: http://dx.doi.org/10.1145/2907071

1. INTRODUCTION

As the on-chip core count increases at a much faster rate than the memory bandwidth,[1] the memory system becomes an increasingly crucial bottleneck in modern processor design. This has forced researchers to pursue aggressive approaches to hide memory latency, for example, use of large-size caches, multithreading, and prefetching. Of these, prefetching offers unique advantages. Large caches incur an energy penalty and consume precious chip area that may be better used for additional cores [Mittal 2014]. MT can improve the performance of parallel applications only. By comparison, prefetching does not incur a large area/energy penalty and can boost even serial

[1]We use the following acronyms frequently in this article: bandwidth (BW), Bloom filter (BF), chip multiprocessor (CMP), control flow graph (CFG), correlation (CoR), cycle per instruction (CPI), data cache (D-cache), hardware (HW), instruction per cycle (IPC), instruction cache (I-cache), last level cache (LLC), linked data structure (LDS), middle level cache (MLC), multithreading (MT), simultaneous multithreaded (SMT), software (SW), virtual CPU (vCPU), and virtual machine (VM).

This material is based on work supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research.
Author's address: S. Mittal, 1 Bethel Valley Road, Oak Ridge National Laboratory, Tennessee, 37830; email: mittals@ornl.gov.

DOI: http://dx.doi.org/10.1145/2907071

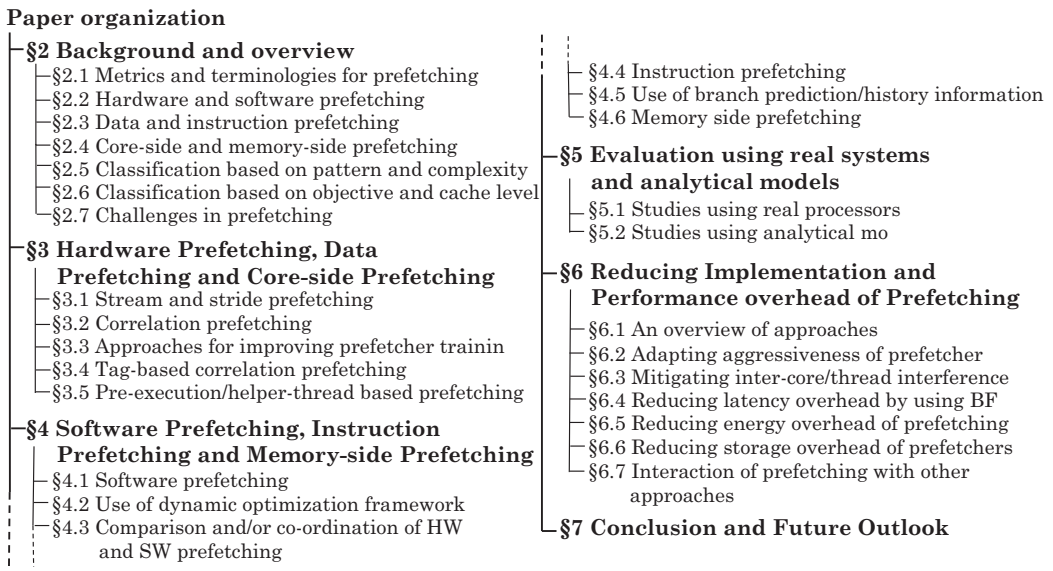

Fig. 1. Overall organization of the article.

applications. In the optimal case, prefetching can bring the performance close to that of a perfect cache by removing nearly all the cache misses [Ferdman et al. 2011; Annavaram et al. 2001b]. Thus, due to its advantages, prefetching is now used in nearly all high-performance commercial processors, such as AMD Opteron, IBM Power8, Intel Xeon, and Oracle Sparc M7.

However, realizing the full potential of prefetching requires careful management and addressing several key challenges. Unlike MT, prefetching requires prediction of future access patterns that is non-trivial in most cases. Complex access patterns demand sophisticated prefetchers that have huge metadata and latency overheads. Naive prefetchers may bring useless lines that consume cache space and may degrade performance by displacing the useful lines [Srinath et al. 2007]. Further, prefetching may interfere with other processor management policies (e.g., cache replacement policy) and cause BW contention. Clearly, although promising, prefetching by itself is not a panacea for improving performance. To address these challenges, several techniques have been recently proposed.

Contributions: In this article, we present a survey of prefetching techniques for processor caches. Figure 1 shows the organization of this article. Section 2 provides a background on and classification of prefetching techniques and then discusses the key challenges related to implementation and effectiveness of prefetching. Section 3 discusses techniques for hardware, data, and core-side prefetching, and Section 4 discusses techniques for software, instruction, and memory-side prefetching. Section 5 discusses techniques evaluated using real systems and analytical models. Section 6 presents several techniques for reducing overhead of prefetchers and improving their effectiveness. Section 7 concludes this article with an outlook towards future work.

Scope of the article: To strike a balance between breadth and brevity, we focus on recent research works that present innovations or insights focused on prefetching in caches and not on other techniques or processor components. We only discuss prefetching in central processing unit (CPU) and not in graphics processing unit (GPU). We present key ideas of research works and do not include their quantitative results, since they use different evaluation platforms and methodologies. To bring out the similarities

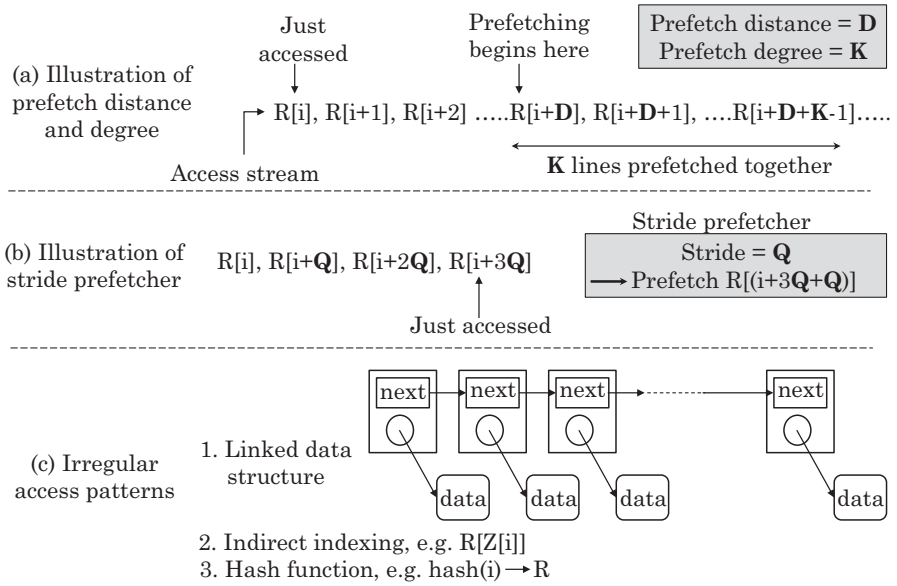

Fig. 2. Illustration of (a) prefetching metrics, (b) stride prefetcher, and (c) irregular access patterns.

and differences of the techniques, we classify them based on key features. We hope that this article will provide a bird's eye view of the state of- the art in prefetching techniques and will be useful for researchers, system designers, application developers, and others.

2. BACKGROUND AND OVERVIEW

We now discuss prefetching briefly (Section 2.1) and refer the reader to previous work for taxonomy and a detailed background [Srinivasan et al. 2004; Vanderwiel and Lilja 2000; Sherwood et al. 2000; Falsafi and Wenisch 2014]. Sections 2.2 through 2.6 characterize prefetching algorithms and research works based on several key parameters. Section 2.7 discusses tradeoffs involved in leveraging full potential of prefetching.

2.1. Metrics and Terminologies for Prefetching

A few parameters and metrics are useful for characterizing prefetchers [Emma et al. 2005]. The prefetch degree and distance are illustrated in Figure 2(a). *Coverage* shows the fraction of original misses eliminated by prefetched lines. *Accurate or useful* prefetches are those that eliminate original misses, while *harmful* prefetches are those that induce misses by replacing useful data. A *timely* prefetch is one where the prefetch request is completed and the data are placed in cache before they are accessed and opposite is true for a *late* prefetch. *Lookahead* shows how well in advance a prefetch is issued, such that prefetched data arrive in cache on time and are not evicted by other data. A prefetch is *redundant* if the data-block brought by it is already present in cache.

2.2. Hardware and Software Prefetching

Prefetching is performed in either HW or SW. HW prefetchers may use additional storage (e.g., a table) to detect specific memory access patterns such as strided accesses and use this to prefetch data that are expected to be referenced soon. SW prefetching inserts prefetch instructions in source-code based on compiler or post-execution analysis, for example, in an LDS, a compiler can insert prefetch instructions for the children of a visited node. An example of SW prefetching is shown in Figure 3.

```
int x[N]; int y[N];int z[N];
for(int j=0; j<N; j++)
{


  z[j] = 3*x[j]+y[j];



}
```

Original code

```
int x[N]; int y[N];int z[N];
for(int j=0; j<N/K; j++)
{
  PREFETCH(&x[j*K], K);
  PREFETCH(&y[j*K], K);
  for(int i=0; i<K; i++)
  {
   z[j*K+i] = 3*x[j*K+i]+y[j*K+i];
  }
}
```

Code with software prefetching instructions

Fig. 3. An illustration of prefetch instructions inserted for software prefetching.

2.3. Data and Instruction Prefetching

Compared to instruction access patterns, data access patterns show higher sensitivity to input dataset and less regularity, which makes data prefetching more challenging. The large instruction working set size of commercial workloads can lead to misses at the L1 and L2 caches, which underscores the need of instruction prefetching for them. However, for applications with a negligible I-cache miss rate (e.g., scientific), instruction prefetching is not required.

2.4. Core-Side and Memory-Side Prefetching

In core- (or processor) side prefetching, the prefetch requests are issued by an engine in cache hierarchy, while in memory-side prefetching, such an engine resides in the main memory subsystem (after any memory bus). Memory side prefetching can save precious chip space by storing metadata off-chip and can also perform optimizations at main memory side [Yedlapalli et al. 2013]. By comparison, core-side prefetching can avail more accurate knowledge of memory reference patterns and can perform cache level optimizations, such as avoiding cache pollution [Srinath et al. 2007].

2.5. A Classification Based on Pattern and Complexity

Prefetchers can also be classified based on the (ir)regularity or complexity of the miss/access pattern they target. The *"Next-K" line* prefetcher brings next K lines after the current miss. The *stride prefetcher* brings lines showing a strided pattern relative to the current miss [Chen and Baer 1995], refer to Figure 2(b). For example, if a past sequence of addresses accessed by loads have been A, $A+Q$, $A+2Q$, and $A+3Q$, then the data at address $A+3Q+Q$ can be prefetched, since this sequence has a stride of Q. For $Q=1$, this is referred to as *stream prefetching*.

For many applications, however, the access patterns are not perfectly strided, and these are termed *irregular* patterns (refer to Figure 2(c)). Correlation prefetching tracks past reference sequence or miss addresses to detect some correlation and use it to guess future miss addresses that are used for prefetching. Spatial prefetchers assume spatial locality and, thus, bring lines into the vicinity of a current miss [Somogyi et al. 2006]. Temporal prefetchers assume that recently seen address streams are expected to recur and, hence, they prefetch based on temporal streams from recent miss history [Wenisch et al. 2005].

Other sophisticated prefetchers target irregular access patterns in particular [Jain and Lin 2013; Somogyi et al. 2009]. Several other prefetchers and variants of the above-mentioned prefetchers have been proposed, which are discussed in this survey.

Table I. A Classification Based on Objective and Cache Level of Prefetching

Classification	References
Study/optimization objective	
Performance	Almost all
Energy	[Guo et al. 2011; Kolli et al. 2013; Dang et al. 2012, 2013; Yu and Liu 2014; Hur and Lin 2006, 2009; Guttman et al. 2015]
Fairness	[Ebrahimi et al. 2011; Chaudhuri et al. 2012; Albericio et al. 2012]
Level in cache hierarchy	
In first-level caches	[Marathe and Mueller 2008; Guo et al. 2011; Srinivasan et al. 2001; Ferdman et al. 2011; Kolli et al. 2013; Reinman et al. 1999; Ferdman et al. 2008; Ferdman and Falsafi 2007; Lim and Byrd 2008; Somogyi et al. 2006; Burcea et al. 2008; Yu and Liu 2014; Hur and Lin 2006; Annavaram et al. 2001b; Lai et al. 2001; Falcón et al. 2005; Yan and Zhang 2008; Hu et al. 2003; Wenisch et al. 2005; Peir et al. 2002; Chilimbi and Hirzel 2002; Alameldeen and Wood 2007; Lee et al. 2012; Guttman et al. 2015; Sherwood et al. 2000; Iacobovici et al. 2004; Joseph and Grunwald 1997; Kumar and Wilkerson 1998; Chen et al. 2004; Mehta et al. 2014]
In mid/last level caches	[Solihin et al. 2003; Kandemir et al. 2009; Ebrahimi et al. 2009; Dang et al. 2013; Hur and Lin 2006; Wu et al. 2011; Wang et al. 2003; Kim et al. 2014; Chaudhuri et al. 2012; Panda and Balachandran 2014; Chen and Aamodt 2008; Srinath et al. 2007; Lee et al. 2008; Pugsley et al. 2014; Nesbit et al. 2004; Sharma et al. 2005; Cantin et al. 2006; Hu et al. 2003; Ganusov and Burtscher 2005; Nesbit and Smith 2004; Chilimbi and Hirzel 2002; Alameldeen and Wood 2007; Lee et al. 2012; Guttman et al. 2015; Lin et al. 2001a; Mehta et al. 2014]

2.6. A Classification Based on Objective and Cache Level

Before moving to detailed discussion and classification of prefetching techniques in Sections 3 through 6, we first classify the prefetching techniques based on their optimization goal. Table I shows this classification and, from this, it is clear that prefetching can provide versatile optimizations. The first and last level caches have different properties (e.g., locality of access stream, characteristic such as private/shared, acceptable latency/storage overhead, etc.) which dictate choice of prefetching technique/parameters for them (e.g., Mehta et al. [2014]). For this reason, Table I also classifies the works based on the cache level where prefetching is used.

2.7. Challenges in Prefetching

Several challenges need to be addressed to realize the full potential of prefetching.

2.7.1. Implementation Overheads. While simple prefetchers (e.g., next-line prefetching) have limited coverage and accuracy, sophisticated prefetchers require a large amount of metadata (e.g., tens of MBs [Lai et al. 2001; Ferdman and Falsafi 2007]). Storing the metadata off-chip requires frequent and costly communication of data on and off chip, while storing it on-chip is only possible for small structures, which still consumes precious chip resources. Also, passing information about the miss address, program counter (PC), and so on, to prefetchers (of especially lower-level caches) introduces nontrivial changes (such as wire-routing) to chip design. To avoid redundant prefetches, the cache needs to be probed, which requires an extra port or sequential checking [Reinman et al. 1999].

2.7.2. Performance Tradeoffs. Due to the features of modern processors, such as out-of-order execution, reduction in misses brought by prefetching may not directly translate into performance improvement. Issuing prefetches in a timely manner requires estimating cache miss latencies and other timing information [Srinath et al. 2007; Marathe and Mueller 2008; Zhu et al. 2010], and this is especially challenging for SW-based prefetchers.

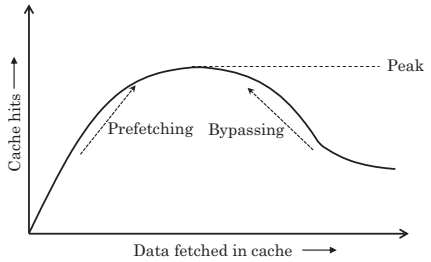

Fig. 4. An illustration of change in cache hits with amount of data fetched in cache.

2.7.3. Cache Pollution and Resource Contention. Prefetching can be seen as a complementary approach to bypassing and this is illustrated in Figure 4. As the volume of data fetched in cache increases, hit rate increases due to better storage utilization; however, thrashing begins as soon as the working set exceeds cache capacity. Prefetched blocks start evicting useful demand-fetched blocks or those brought into shared cache by prefetchers of other cores. Resultant cache pollution generates further misses, which may trigger more prefetches. With increasing core-count, inter-core interference escalates and, due to reduced per-core BW, contention from prefetch requests also increases.

Some techniques place prefetched blocks in an additional buffer (Section 6.1). However, accessing them sequentially or in parallel with cache causes energy/latency overhead. Also, they require reorganization of chip architecture and preclude the possibility of cache space sharing between demand-fetched and prefetched data. It is clear that achieving the optimum balance (peak point in Figure 4) requires a careful moderation of prefetching parameters and aggressiveness.

2.7.4. Reliability Challenges. Prefetching techniques can increase the soft error rate by increasing the residency time of data in the cache [Mittal and Vetter 2015]. Also, by inducing extra writes, they can cause hard errors and reduced device-lifetime in nonvolatile memories that have limited write endurance.

The techniques discussed in subsequent sections seek to address these challenges.

3. HARDWARE PREFETCHING, DATA PREFETCHING, AND CORE-SIDE PREFETCHING

In this section, we discuss HW, data, and core-side prefetching techniques that form the most prominent prefetching approaches. In this and the next sections, we discuss prefetching techniques by roughly organizing them into several groups. Although some of the techniques belong to multiple groups, we discuss them under a single group only.

3.1. Stream and Stride Prefetching

Jouppi [1990] presents a prefetching scheme using stream buffers. On a cache miss, sequential cache blocks are prefetched into a separate stream buffer until it is filled. Thus, the prefetched blocks are not placed into the L1 cache to avoid its pollution. The stream buffer is organized as a first in first out (FIFO) buffer. Next time, when the L1 cache sees a miss, the first entry of the stream buffer is checked and, on a hit, a block is brought into the L1 cache. As prefetched data are used, more prefetches are issued, which keeps the buffer sufficiently ahead of the instruction stream of the processor so the entire latency can be hidden. Jouppi also explored the use of multiple streaming buffers in parallel that can prefetch multiple intertwined reference streams. This is useful in several cases, for example, when applications access multiple arrays inside a loop.

Joseph and Grunwald [1997] present a prefetching approach based on the assumption that the miss address stream can be approximated by a Markov transition diagram. In this diagram, a weight is given to every transition from node P to node Q, which shows

the fraction of accesses to P where the next access happens to node Q. Assuming that the execution pattern is repetitive, this Markov diagram can be used to predict the miss address wthat follows the current miss address. Since programs show time-varying behavior and the degree of a node may become very large, constructing and storing a full Markov diagram is infeasible. To address this issue, they limit the number of nodes in the transition diagram along with their out-degrees. This allows the diagram to be stored in a table. When a miss address matches that in the table, the predicted next addresses can be prefetched. Higher priority is given to those "next addresses" that have a higher probability of transition to them from the current address. If the prefetch request queue is full, then low-priority requests are discarded.

Sherwood et al. [2000] propose a scheme where the stream buffer follows the stream predicted by an address predictor, instead of a fixed-stride predictor. This allows use of different predictors that can prefetch more effectively than the fixed-stride predictor. They demonstrate the use of a stride-filtered Markov predictor that uses a two-delta stride table before the Markov predictor table (a two-delta stride predictor is one where only a new stride, which is consecutively observed twice, can replace a predicted stride). In the write-back stage, the PC for a missed load is used to index the stride table. If the stride computed from (present miss address-last address) differs from two-delta stride or the last stride, then the address cannot be predicted by the stride, and, hence, it is stored in the Markov predictor. The Markov table is queried using the last miss address for finding the next prefetch address. In case of a hit, the Markov address is utilized for prefetching and, in case of a miss, the next stride address for prefetching is computed using the last address. They show that their predictor allows accurately prefetching for pointer-based and complex array-access-based applications.

Sair et al. [2002] analyze load miss streams to obtain insights that can improve prefetcher effectiveness. Based on the miss access patterns, they classify the load miss streams into four categories, viz. stride, next-line, same-object, and pointer-based miss stream. Same-object misses are further misses occurring on a heap object that has been accessed recently. These misses can be avoided by prefetching the entire object or, minimally, the blocks of the object to be accessed in the near future. Pointer-based misses happen when a pointer is dereferenced to access an object. Since these misses are most challenging to eliminate using prefetching, they discuss two metrics to analyze them and design a strategy for alleviating them. "Pointer variability" shows the number of pointer transitions that are frequently changing and that are stable (a load is called a pointer transition if it loads a pointer). Pointer variability quantifies the number of times a pointer transition for an address does not load the same pointer as the one loaded previously from that address. "Object fan out" shows how many pointers are transitioned and frequently miss in the cache. Thus, programs with high variability and fan out are more difficult to prefetch than those with small variability and fan out. They also show that the classification of misses can be accurately done in HW and, based on this classification, prefetching can be efficiently performed.

Iacobovici et al. [2004] note that unit stride and single non-unit stride prefetching techniques do not work for patterns that frequently appear in scientific applications. They present a multi-stride prefetcher that can detect and prefetch for a stream consisting of a maximum of two steady-state and two transitional strides, that is, four distinct strides. An example of multi-stride miss address stream is A, A+1, A+2, A+4, A+5, A+6, A+8, ... , which is composed of two stride components, viz. a unit stride that appears twice and a stride of two that occurs once. In the training phase, the prefetcher detects the stride patterns. If successive misses in a stream show identical stride, then the prefetcher takes them as belonging to a single stride stream, and the prefetcher remains in state 1. If a miss displays a different stride, then the prefetcher transitions to state 2. The stride from this miss sets the transitional stride (stride12).

A future miss that deviates from the second stride switches the state to state 1 and this stride sets the transitional stride (stride21). After training, if, for a missing load, the recorded stride is same as the one that is predicted, then a prefetch is triggered.

Zhu et al. [2010] note that, in a stream, the timing of data accesses happens in a predictable manner, for example, in a constant-stride prefetcher, adjacent accesses in a stream are expected to occur at nearly equal time intervals. Based on this, their prefetching technique stores both the addresses referenced and the timing information to predict *when* a miss happens, in contrast with the conventional stream-based prefetchers that only store the addresses. The timing information is measured in terms of the number of misses. Their technique classifies miss addresses into different streams based on whether they are from the same memory region or the same instruction. By virtue of avoiding untimely prefetches, their technique mitigates cache pollution and memory BW wastage.

Kim et al. [2014] present a technique that aims to identify all potential stride streams, including those detected by PC-based and delta-correlation-based prefetchers. Their technique sees whether the last miss and a previous miss form a stream where a fixed stride separates more than two miss addresses. Since the last miss may be part of different streams having dissimilar strides, tracking multiple streams is essential for choosing the best stream from them. In such a case, their technique chooses the longest stream, since it is likely to cover a larger number of misses and be more accurate than a short stream. This addresses the issue of overlapping streams. On detecting a stream, future accesses in that stream can be identified. The number of streams that can be tracked are limited by the available storage space. Still, since their technique tracks multiple streams, it can continue prefetching by skipping a few of them.

3.2. Correlation Prefetching

As shown in Table II, many works propose correlation prefetching (refer Section 2.5) techniques. We now discuss a few of them.

Lai et al. [2001] propose a dead-block prediction-based prefetching technique. Their technique records the memory reference trace to estimate when an L1-D cache block sees the last access. From this time, until a cache miss replaces the block, the block is dead [Mittal 2014]. They note that dead times are typically large and more than the time required for fetching data from the next level of cache. Their technique also uses address CoR to predict the block that will be referenced soon and prefetches it to eliminate the miss and improve performance. This block can be stored at the place of the dead block in the L1 cache itself. They show that their technique provides timely and effective data prefetching. However, their technique requires storage proportional to the application working set size. CoR data need to be stored across long recurring application phases and since the information about last reference is computed on each L1 access, it needs to be stored on-chip to achieve high BW. Hence, to provide reasonable coverage, their technique may require impractically large storage space [Ferdman and Falsafi 2007].

Nesbit and Smith [2004] note that conventional prefetch methods store miss address streams in a table, which provides fast lookup, but the table reserves a fixed space, and the entries in a table quickly get stale, which may trigger useless prefetches. They present an alternative organization for storing prefetch history. Their method decouples matching of prefetch key from storage of history required for prefetching. First, the prefetch key is used to access the "index table" to obtain a pointer into a global history buffer (GHB). In every GHB record, a global miss address and a link pointer are stored. Using link pointers, GHB entries are chained into address lists that are chronological list of addresses with an identical "index table" prefetch key. By using different keys, different history-based prefetching techniques are realized, for example,

Table II. Correlation and Pre-Execution-Based Prefetching Techniques

Classification	References
Correlation prefetching	[Ferdman et al. 2008; Solihin et al. 2003; Lai et al. 2001; Hu et al. 2003; Ferdman and Falsafi 2007; Chou 2007; Diaz and Cintra 2009; Manikantan et al. 2011; Dang et al. 2012; Liu et al. 2012; Jain and Lin 2013; Somogyi et al. 2006; Burcea et al. 2008; Wenisch et al. 2009; Somogyi et al. 2009; Srinath et al. 2007; Ferdman et al. 2011; Nesbit and Smith 2004; Huang et al. 2012; Roth et al. 1998]
Tag-based correlation prefetching	[Hu et al. 2003; Sharma et al. 2005]
Pre-execution based prefetching	[Zilles and Sohi 2001; Ganusov and Burtscher 2005, 2006; Zhang et al. 2007; Huang et al. 2012; Annavaram et al. 2001b; Collins et al. 2001b; Lu et al. 2005; Luk 2001; Balasubramonian et al. 2001; Collins et al. 2002; Rabbah et al. 2004; Aamodt et al. 2002]

stride prefetching can use PC of load instruction while Markov prefetching [Joseph and Grunwald 1997] can use PC-independent global miss addresses. The GHB can be sized depending on the length of the history required to be tracked, which leads to better storage efficiency than conventional tables. The GHB is organized as a FIFO buffer and, thus, stale entries are automatically removed from it. Thus, accurate reconstruction of access pattern allows them to implement sophisticated prefetch techniques that exploit complex access patterns.

Chou [2007] note that as off-chip latency increases, latency of on-chip computation that separates overlapping off-chip accesses becomes negligible, and very little useful work can be performed during this time. Thus, application execution can be logically divided into "epochs," such that each epoch has on-chip computation periods and off-chip accesses. Eliminating off-chip accesses removes the epoch, and decreasing the number of epochs translates into performance improvement. Conventional CoR prefetching techniques avoid single misses and epochs are eliminated as a secondary effect. However, avoiding single misses may not remove an entire epoch and, thus, may not improve performance. Instead of eliminating individual misses, their technique prefetches all the off-chip misses in the epochs to remove them entirely. Their technique does not attempt to eliminate cache misses that overlap with another miss that triggered the epoch, and this helps in reducing the size of the CoR table. Their correlating prefetcher is stored in main memory and its access latency is hidden by exploiting memory level parallelism, that is, the table is accessed during the time an off-chip access stalls the processing core. Thus, without wasting on-chip space, prefetches can still be issued in a timely manner.

Liu et al. [2012] present a miss CoR-based prefetching technique. In their technique, CoR between a miss and a previous miss is ascertained when they happen closely in space and time, where space CoR means that these misses lie within a specified address range. After dynamically capturing these miss correlations, their technique uses compression to save them along with the data block content. Thus, along with the demand data, prefetch metadata are brought with minimal overhead. This allows very large CoR history. Compression is used to save metadata of each block within the original block size in the Dynamic random access memory (DRAM). Based on the miss correlations, accurate prefetches can be issued for improving performance.

Roth et al. [1998] present a dependence-based prefetching technique that works by identifying a program kernel that computes addresses of LDS elements. Assuming that, in the near future, the program will follow similar steps to traverse the structure, a prefetching engine speculatively executes this kernel together with the main program. As an address is loaded, the loads consuming that address are predicted, and prefetches for those loads are immediately issued, and, in this way, dependence information is

utilized. By virtue of executing only those loads required for traversing the LDS, the prefetching engine can run far ahead of the main program and thus perform prefetch in a timely manner. Using this, their technique can cover long LDS access latencies.

Wenisch et al. [2005] present a temporal memory streaming (MS) technique to eliminate coherent read misses in shared memory multiprocessors. They note that shared addresses are likely to be accessed together and in the same sequence. Also, recently accessed address streams tend to recur. Such temporal correlation is found in accesses to general data structures such as arrays and LDSs (e.g., trees and lists). By comparison, spatial or stride locality, found only in array-based data structures, relies on memory layout of the data structure. Based on temporal correlation, they extract temporal streams from miss history of recent sharers and move data to a subsequent sharer before the data are requested.

Somogyi et al. [2006] note that in commercial workloads, memory accesses show repetitive layouts spanning over large (e.g., several kilobytes) memory regions. Also, repetitive pattern of these accesses can be predicted by code-based correlation. Since these patterns may be non-contiguous, the use of larger cache block size for capturing such spatial correlation wastes the bandwidth. They present a technique that detects code-correlated spatial access patterns and brings such blocks into the cache before demand misses. On first reference to a spatial region, their technique predicts the cache blocks that will be referenced in that region during a monitoring interval. The monitoring interval is the time from first access to the region until any block accessed in the interval is invalidated or evicted from the cache. By virtue of exploiting correlation between code and access patterns, their technique achieves much higher prediction coverage than the address-based predictors since there are so much fewer distinct code sequences than data addresses.

Somogyi et al. [2009] propose spatio-temporal memory streaming (STeMS) to synergistically integrate spatial and temporal streaming. Temporal MS tracks past miss sequences to predict subsequent chains of dependent misses, while spatial MS tracks recurring data layout patterns in memory regions of fixed size to predict future misses. While temporal MS fails to predict compulsory misses and achieves low accuracy due to the inability to detect where streams terminate, spatial MS cannot establish order between predictions and is also limited due to the use of fixed size regions. Noting that the spatial access sequence recurs in a single region and across regions, STeMS tracks temporal sequence of region accesses. Also, spatial relationships in every region are used to predict the complete miss sequence. Based on it, cache blocks are prefetched to the requesting processor. Unlike a naive combination of spatial and temporal MS, STeMS avoids interference between the predictors and, thus, achieves higher prefetch accuracy. They also show that STeMS achieves comparable or better prefetch coverage and performance than using either spatial or temporal MS alone.

Panda and Balachandran [2014] note that in parallel applications data and code are shared and communicated between cores. Also, demand misses seen in a core repeat in other cores at a large time interval (e.g., average of tens of thousands of cycles). These miss streams, referred to as *cross-core miss streams*, cannot be eliminated by core-local stream prefetchers. They propose a cross-core spatial streaming technique in which the cross-core spatial streams and the cores involved in it are detected. Then, the spatial streams from a private MLC prefetcher are transmitted to MLC prefetchers of associated cores well in advance to allow them to prefetch data and eliminate cross-core misses for improving performance.

Cantin et al. [2006] present a technique, called *stealth prefetching*, for broadcast-based shared-memory multiprocessor systems. In such systems, memory latency values are high and early prefetches may cause state downgrades or invalidations in remote nodes. They define a region to encompass power-of-two number of cache lines and

identify non-shared regions using a coarse-grained coherence tracking scheme [Cantin et al. 2006]. They note that the majority of memory accesses happen in memory regions that are not shared when the access happens, and the majority of lines in such regions are accessed. Based on this, when the lines accessed in a region exceed a threshold, their technique prefetches a certain number of lines in the region from DRAM and dispatches them to the requesting processor. To improve prefetching accuracy, lines in a region that were previously accessed are tracked. Prefetched lines are stored with a no-permission coherence state and not kept individually coherent. If another processor obtains exclusive access to the region or sends a memory request to make the prefetched lines stale, then the prefetched lines in the original processor are invalidated. The prefetch requests are not broadcast to other processors, and they can still get exclusive copies of lines, and, thus, the prefetching is stealthy. Also, since multiple lines can be prefetched in a single request, the prefetching is aggressive and efficient.

3.3. Approaches for Improving Prefetcher Training

The training stream presented to a prefetcher decides the "repeating pattern" observed by the prefetcher and, hence, it has a significant impact on the efficacy of the prefetcher. We now discuss a few techniques to improve prefetcher training (e.g., Ferdman et al. [2011], Manikantan et al. [2011], Jain and Lin [2013], and Guttman et al. [2015]).

Manikantan et al. [2011] study extending the training stream stored by CoR prefetchers to improve their performance. They denote a primary miss as one that initiates a request to the next level of cache/memory and a secondary miss as a request where the data requested by a primary miss have not arrived in the cache. They note that presenting only a primary miss address stream to train a prefetcher precludes the opportunity of exploiting information provided by secondary misses and cache hits. They propose including secondary misses and cache hits in the training stream to improve the regularity seen by the prefetcher. The improvement in regularity is confirmed by a reduction in entropy measurement. While other techniques trigger a prefetch only on a primary miss, they suggest triggering prefetches on secondary misses also to improve the performance of prefetchers. While requiring minimal HW modifications, their technique reduces cache misses.

Jain and Lin [2013] present a prefetcher, called an irregular stream buffer, for targeting irregular streams of memory accesses that are temporally correlated. Their technique translates groups of correlated physical addresses into contiguous addresses in a new address space by using an extra indirection level. Based on this, their technique organizes prefetching metadata such that it is simultaneously spatially and temporally ordered. This reduces the problem of irregular stream prefetching to sequential prefetching in the new address space. This remapping also improves accuracy and coverage since, based on PC of the loading instruction, prefetcher input can be segregated into several streams [Nesbit and Smith 2004]. Further, storing most of its metadata on-chip allows us to use the LLC access stream (and not the LLC miss stream) to train the prefetcher, which leads to significant improvement in reference stream predictability.

3.4. Tag-Based Correlation Prefetching

Since applications reference a large number of addresses, the CoR prefetchers that work by tracking addresses incur large metadata overhead. To address this, tag-based correlation prefetching (TCP) techniques have been proposed that utilize the observation that, due to address locality, the tags formed by high-order address bits also show locality. Since one tag sequence may correspond to multiple address sequences, a tag-based CoR table requires a much smaller number of entries. We now discuss a few TCP techniques.

Hu et al. [2003] note that while a memory address always maps to a fixed cache set, a tag can appear in different cache sets, which happens when multiple addresses have the same tag but different set indices. Hence, tag sequences are highly repetitive both in a single set and across the sets. Based on this, they propose a TCP technique that has the same accuracy as an address-based CoR prefetching scheme but requires magnitude order smaller storage. Their technique monitors per cache-set tag sequences and makes predictions based on recurring tag correlation sequences. They show that their technique provides better performance than a prefetching approach based on correlations of both PC traces and addresses.

Sharma et al. [2005] present a prefetching technique that works by partitioning the memory address space into tag concentration zones (TCzones). If addresses of two misses have same lower order bits, then they are in the same TCzone. The prefetcher tracks L2 cache miss stream. The number of miss events, after which a miss stream shows the same data item again, is termed *recurring distance*. On detecting multiple misses with identical recurring distance, a pattern is inferred. At this point, the prefetcher starts recording for making future prediction. The prefetcher can work in one of the two modes, viz. absolute and differential. In absolute mode, it looks for value locality by monitoring pattern of tags within every TCzone and in differential mode, it looks for stride value locality by monitoring stride (i.e., delta between subsequent tags) pattern in every TCzone. Depending on which of the two patterns is dominant in the miss stream, the prefetcher can switch to it to improve the effectiveness of prefetching.

3.5. Pre-Execution or Helper-Thread-Based Prefetching

In processors with multithreading, while the main thread executes the program, another thread can redundantly execute a full or reduced version of the program to speculatively generate data addresses for performing prefetching. Such a thread is known as speculative slice, pre-execution (or pre-computation) thread, helper thread, or future thread. These threads progress ahead of the main thread and run nearly the same code. Thus, they actually compute load addresses instead of predicting them and allow restricting prefetching to probable control-flow paths instead of all possible paths. Figure 5 show some examples of helper-thread prefetching (based on Luk [2001]). We now discuss a few of these techniques (refer Table II).

Annavaram et al. [2001b] present a precomputation-based approach to predict prefetch addresses. For an instruction fetched in instruction fetch queue (IFQ) from I-cache, their technique determines the dependencies and stores them in IFQ as pointers along with the instruction. Using profiling, their technique identifies addresses referenced by the load/store instructions that may lead to majority (90%) of D-cache misses. When a load/store instruction that may cause a cache miss enters the IFQ, their technique tracks the dependence pointers that are stored in IFQ for generating a dependence graph of instructions that await execution. The dependence graphs are executed by a separate precomputation engine (PE) to produce load/store instructions early for prefetching. PE executes in a speculative manner and, thus, it does not affect the processor state, and it makes progress faster than the main execution by avoiding delays in reorder buffer and fetch queue that may be seen by the main execution. Their technique achieves performance reasonably close to that of perfect D-cache.

Luk [2001] present a software-controlled pre-execution scheme to accelerate programs with irregular access patterns. Their technique runs the original program itself and, thus, does not require program shortening. When the main thread is stalled, more resources can be given to the pre-execution thread to enhance overall performance. They suggest pre-execution schemes for dealing with different irregular access patterns. For the pointer chasing problem, one helper thread is spawned for pre-executing each pointer chain. Similarly, helper threads can be used to execute different procedure

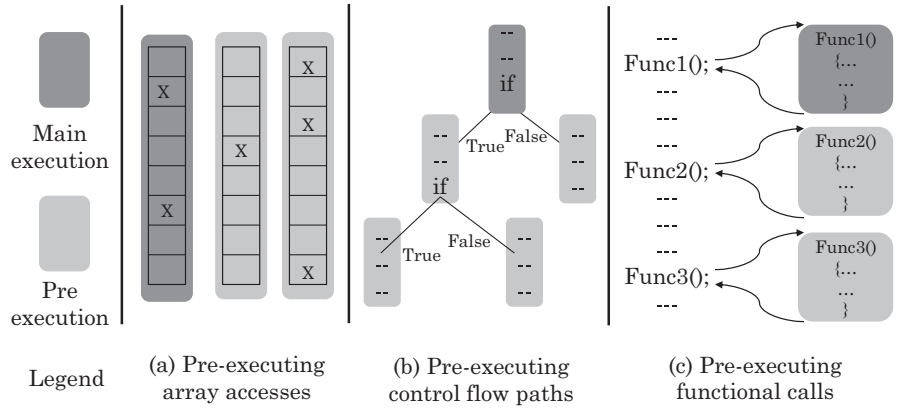

Fig. 5. Illustration of use of helper-thread prefetching.

calls or traverse different control-flow paths. On determining the correct execution path, their technique cancels the wrong-path pre-executions. Thus, their technique performs effective prefetching under complex data access patterns and control flows.

Collins et al. [2001b] present a technique where idle HW contexts are used to spawn speculative threads that aim to hide miss latency by triggering upcoming cache miss events well in advance of access by main thread. Since speculative threads cause contention for processor resources (e.g., fetch and memory bandwidth), such pre-execution is done only for those static loads, called delinquent loads, which lead to the majority of stalls in the main thread. For example, fewer than 10 static loads may lead to more than 80% of L1D cache misses. Speculative threads compute the address referenced by a future delinquent load to prefetch the corresponding data. They show that, compared to the case when only the main thread starts speculative threads, allowing the speculative thread to start additional speculative threads provides higher performance improvement due to aggressive speculation.

Collins et al. [2002] propose using a pointer cache (PtC), a dedicated cache that stores pointer transitions in the application. Given the effective address of a pointer and assuming the pointer points to an object, PtC provides this object's base address. When a load shows miss in L1 cache but hit in PtC, the first two cache blocks of the pointed-to object are prefetched. They also explore use of PtC with speculative pre-computation. With pointer-traversing codes, a speculative-thread in other techniques cannot progress faster than the main thread. In their technique, with help from PtC, a speculative thread can progress farther ahead by traversing data structures under pointer-transition induced cache misses. PtC helps in avoiding serial accesses to recurrent loads and, thus, by using PtC, speculative-threads' control instructions can go along the right object traversal path, which leads to accurate prefetching for the main thread. As pointer transitions change, PtC is also updated. For a fixed transistor budget, using a PtC with an L3 cache provides higher performance benefits than using a larger-sized L3 cache alone.

Aamodt et al. [2002] present a helper-thread based prefetching technique. Using profiling, they identify code regions with many I-cache misses. The instructions immediately preceding the basic blocks showing I-cache misses are identified and are called target points. Further, trigger points are identified where a helper thread can be initiated for prefetching instructions after the target point. Then, a helper thread is generated and is attached to the main thread. At runtime, on encountering a trigger point in the main thread, a helper thread is spawned in the spare context. The helper

Table III. A Classification of Research Works

Classification	References
SW prefetching (or compiler support)	[Beyler and Clauss 2007; Khan et al. 2014; Marathe and Mueller 2008; Solihin et al. 2003; Kandemir et al. 2009; Wang et al. 2003; Lu et al. 2003, 2005; Luk 2001; Chilimbi and Hirzel 2002; Zhang et al. 2006; Mehta et al. 2014; Lee et al. 2012; Rabbah et al. 2004; Collins et al. 2001b; Mowry et al. 1992; Fuchs et al. 2014; Guo et al. 2011; Chen et al. 2007; Zhang et al. 2007]
HW prefetching	Almost all others
Comparison/coordination of HW-SW prefetching	[Verma and Koppelman 2012; Guttman et al. 2015; Mehta et al. 2014; Lee et al. 2012]
Use of dynamic optimization framework	[Lu et al. 2003, 2005; Zhang et al. 2006, 2007]

thread speculatively executes instructions related to control flow that will be later encountered by a main thread and this achieves prefetching effect.

Ganusov and Burtscher [2006] present an event-driven helper threading approach for software emulation of simple or complex HW prefetchers. Based on the cache miss observed in the main thread, the helper thread predicts the data that will be accessed next and prefetches it. For making a prediction, the helper thread can use any simple or complex prefetching algorithm. After that, the helper thread again stalls, waiting for a miss in main thread. To enable efficient inter-thread communication without causing contention on shared cache ports, they use a FIFO-style event buffer. Using this, the helper thread can receive information about cache misses from the Re-order buffer (ROB) of the core running the main thread. When the main thread accesses a prefetched line, the corresponding load instruction is marked as consumer of the prefetched data. When this instruction is committed, a prefetch trigger is transmitted to the helper thread as if the cache miss was shown by this instruction. In this manner, their technique emulates the conventional HW prefetching and the helper thread can progress faster than the consumption of data by main thread. Their prefetching mechanism allows flexibly, turning on/off the prefetcher when desired without affecting the operation of the main thread.

4. SOFTWARE PREFETCHING, INSTRUCTION PREFETCHING, AND MEMORY-SIDE PREFETCHING

This section complements Section 3 by discussing techniques for SW, instruction, and memory-side prefetching. Table III classifies the research works based on HW or SW prefetching and use of dynamic optimization framework.

4.1. Software Prefetching

Chilimbi and Hirzel [2002] present a prefetching technique that is especially useful for pointer codes developed in weakly typed languages such as C/C++. First, a temporal data reference profile is collected from the application being executed. Then, profiling is disabled, and hot (e.g., frequently repeating) data streams are extracted from the reference profile using an analysis algorithm. Only reasonably long streams are extracted to amortize the prefetching overhead. Then, instructions are inserted into the application for detecting and prefetching the hot data streams. Then, analysis is disabled, and the application runs with the prefetch instructions. Afterwards, the inserted checks and prefetch instructions are removed. After this, the entire cycle of profiling step, analysis step, optimization step, and so on, begins again. Their technique improves performance while keeping the overhead low by focusing on few hot data streams.

Marathe and Mueller [2008] present a SW-only prefetching technique to reduce L1 cache misses. They first run the application with a training data set to extract annotated memory access trace. Using this, memory addresses generated by loads and

stores and their contents are monitored in an offline manner to see whether a load miss address may be predicted from previous instructions (called predictor instructions). Based on this, prefetch predictors are produced of which timely, accurate, and non-redundant predictors are selected. To see timeliness of a predictor, their technique checks whether a load miss is too close (in terms of processor cycles) to a predictor instruction, since, in such a case, prefetching will not be useful. When this distance is smaller than a threshold, the predictor is not used. Also, if a large fraction of prefetched data of a predictor is redundant, it is eliminated. Based on the selected predictors, SW prefetching instructions are placed in an application's assembly code directly. Working at instruction level enables their technique to have a broad view of memory access patterns spanning over boundaries of functions, modules, and libraries. Their technique integrates and generalizes multiple prefetching approaches such as self-stride, next-line and intra-iteration stride, same-object, and other approaches for pointer-intensive and function call-intensive programs.

Wang et al. [2003] present a HW-SW co-operative prefetching technique. Their technique uses compiler analysis to generate load hints, such as the spatial region and its size (number of lines), to prefetch the pointer in the load's cache line to follow for prefetching and the pointer data structure to recursively prefetch. Specifying size allows prefetching a variable-size region based on loop bounds. Based on these hints and triggered by L2 cache misses, prefetches are generated in HW at runtime. Unlike other SW techniques, in their technique, individual prefetch addresses are generated by HW and not by SW, which allows timely prefetching of data. Also, use of compiler hints allows reduced storage overhead and accurate prefetching even for complex access patterns, which is challenging for a HW-only approach. They show that while generating significantly less prefetch traffic, their technique still provides similar performance improvement as a HW-only prefetching technique.

Rabbah et al. [2004] present a compiler directed prefetching technique that uses speculative execution. The portion of program dependence graph relevant to the address computation of a long latency memory operation is termed as load dependence chain (LDC). The LDCs are identified by the compiler and precomputations are statically embedded in the program instruction stream with prefetch instructions. Using speculative execution of LDCs, future memory addresses are precomputed, which are utilized for performing prefetching. For pointer-based applications, LDCs contain instructions that may miss in D-caches, and, hence, generating prefetch addresses for such applications causes large instruction overhead. To address this, their technique provisions that if a load in the LDC sees a miss, successive precomputation instructions are bypassed.

Khan et al. [2014] present a SW prefetching technique based on runtime sampling and fast cache modeling. Their technique randomly samples memory instructions with low sampling ratio (e.g., 1 in 100,000). The blocks in D-cache that are accessed by sampled instructions are monitored for data reuse. Further, whenever the sampled instructions are re-executed, a stride sample is recorded. Based on data reuse samples recorded over the whole execution, a statistical cache model estimates per-instruction cache performance for given cache sizes. Based on it, delinquent loads are identified and stride samples for each delinquent load are analyzed to detect regular stride patterns. On detecting a dominant stride pattern for a delinquent load, their technique computes suitable prefetch distance and inserts a prefetch instruction for such a load. Based on cache modeling, their technique also identifies whether a data block will not be reused in MLC/LLC. For such data blocks, their technique uses a special prefetch instruction that prefetches data in L1 cache without polluting MLC/LLC. On eviction, this cache block is directly written back to main memory. Thus, by using cache bypassing, their technique reduces cache pollution. For single-thread applications, their

technique provides performance comparable to HW prefetching, but, with increasing core-count and resource-contention, the advantage of their technique improves.

4.2. Use of Dynamic Optimization Framework

Several input- and microarchitecture-dependent events cannot be predicted at compile time, which limits the effectiveness of SW prefetching. Dynamic optimization frameworks can address these limitations by allowing performance monitoring and addition/removal of optimizations at runtime. Several prefetching techniques that use these frameworks are discussed next.

Lu et al. [2003] use a user-mode dynamic optimization system, named ADaptive Object code REoptimization (ADORE) for implementing D-cache prefetching. Using ADORE, they monitor performance of binary execution and ascertain performance-critical traces/loops that show frequent D-cache misses. Prefetch instructions are inserted only in these loops/traces, and binary is patched to redirect further execution to optimized traces. Their approach provides higher performance than static prefetching, while incurring only small overhead.

Lu et al. [2005] note that SMT and CMP processors present different tradeoffs of helper-thread-based prefetching. In SMTs, several processor resources, for example, issue queues, and L1 caches, may be shared or partitioned. Both main and helper threads run on the same core, which enables fast synchronization. By comparison, in CMPs with private L1 cache and shared L2 cache, the main thread cannot easily start helper-thread execution for a specific L2 cache miss. Also, communication of register values between main and helper threads is not straightforward. To address this, they use the ADORE framework to implement helper-thread-based prefetching in CMPs. They bind the main thread to one core, while the helper code, runtime optimizer, and runtime performance monitoring codes are executed on another core. This minimizes the negative influence of helper threads on main threads and precludes the need of starting multiple thread slices. Performance monitoring code detects program regions with delinquent loads and the helper code for these regions prefetches for delinquent loads. The main thread initiates the helper thread and communicates with it using a mailbox in shared memory.

Zhang et al. [2006] present a prefetching technique that uses Trident [Zhang et al. 2006] dynamic optimization framework to collect frequently executed basic instruction blocks that form hot traces. By analysis of hot traces, delinquent loads and a suitable prefetch distance for them are identified, and then prefetch instructions are inserted into the hot trace. To adapt to dynamically changing workload behavior, they use HW monitoring to adjust prefetch distance for each load operation or even remove prefetch instructions. This allows their technique to achieve higher performance by utilizing runtime information.

Zhang et al. [2007] use the Trident framework to improve the effectiveness of helper-thread based prefetching. Trident monitors program's behavior and triggers compiler optimizations in a separate thread to adapt to that behavior. The hot execution traces of the main thread are stored in the code cache of the dynamic optimization system and are used to create p-slices. Generation of p-slices happens in the Trident framework, which allows adaptation based on program input, HW configuration (e.g., cache architecture, available HW contexts etc.), and runtime behavior (e.g., control-flow execution). To accelerate p-slice ahead of main thread's execution, they predict HW load stride to speculatively specialize the p-slice for reducing their overhead. Further, based on tracking the effectiveness of prefetching, they adapt the runahead distance of p-slices so memory access is fully covered. Control-flow hazards are mitigated by the streamlined nature of the hot traces. Also, prefetching addresses are monitored to detect and prevent a p-slice from diverging from the main thread.

4.3. Comparison and/or Coordination of HW and SW Prefetching

Lee et al. [2012] study the advantages and disadvantages of HW and SW prefetching and their interaction. As for advantages of SW prefetching, HW prefetchers fail for very short streams and require complex structures for detecting irregular access patterns. Also, when the number of streams present in the application exceeds the HW resources, HW prefetchers may not clearly distinguish them, whereas SW prefetchers can insert prefetch instructions for each stream individually. Further, the HW prefetchers in commercial processors place data in lower-level (L2 or L3) cache only, whereas SW prefetchers can place data directly into the right cache level. Also, SW prefetchers can easily ascertain loop bounds and avoid prefetching outside the bounds that (especially aggressive) HW prefetchers fail to do. Based on this, SW prefetching can be used for short streams, irregular access patterns, and L1 cache miss avoidance. As for advantages of HW prefetchers, they can account for runtime behavior and input variations and can adapt their aggressiveness. Also, SW prefetchers can greatly increase the instruction count (e.g., up to 100%), which wastes fetch and execution BW, although applications that use prefetching are generally memory bound, and, hence, additional instructions may not increase their completion time. By using HW and SW prefetchers together, prefetching can be performed for a larger variety of streams, and SW prefetch requests can be used to train the HW prefetcher. However, these prefetchers may also interact negatively, for example, when SW prefetches inhibit the ability of HW to detect streams properly or when the harmful prefetches brought by SW cause cache pollution and BW contention.

Mehta et al. [2014] note that different architectural features on different processors present unique tradeoffs and demand specific prefetching strategies for them. For example, in SandyBridge, the MLC streamer prefetcher and L1 SW prefetcher can coordinate to bring data to L1 cache, overcoming the limitations of the L1 HW prefetcher. By comparison, on Xeon Phi, no prefetching is done by the L2 streamer prefetcher in the presence of L1 SW prefetch commands. Yet, to bring data into the L1 cache, coordination between SW prefetch instructions at multiple cache levels can be utilized, such that LLCs prefetch first and use larger prefetch distance than L1 cache, and then the L1 cache prefetches data from the LLC. They present coordinated multi-stage prefetching techniques for each of the two processors. Their technique prefetches array references that are direct-indexed streaming, direct-indexed strided, and indirect indexed. For such references, prefetch instructions are inserted only for L1 cache for SandyBridge since other cache levels use HW prefetch. For Xeon Phi, prefetch instructions are inserted for all levels of cache. In a loop, prefetch instructions are introduced in a manner where computation is performed between them, which avoids the possibility of pipeline stall due to prefetches filling the miss status holding register (MSHR). Finally, their technique determines the prefetch distance, and this computation differs between Xeon Phi and SandyBridge since the former has in-order cores while the latter has out-of-order cores. They also discuss several processor-specific optimizations for improving the efficacy of prefetching.

4.4. Instruction Prefetching

Table IV shows several techniques for instruction prefetching (Section 2.3). We now discuss a few of them.

Zhang et al. [2002] present a prefetching technique that works by correlating execution history with cache miss history. For each cache miss, a correlated instruction is ascertained that was fetched a fixed number of instructions before the miss, and this correlation is stored in a table. For instance, for an I-cache with 12-cycle miss latency, an instruction fetched 12 cycles before the miss is used as the prefetch trigger. When

Table IV. A Classification Based on Data/Instruction and Memory/Core-Side Prefetching

Classification	References
Instruction prefetching	[Zhang et al. 2002; Srinivasan et al. 2001; Ferdman et al. 2011; Kolli et al. 2013; Reinman et al. 1999; Spracklen et al. 2005; Ferdman et al. 2008; Falcón et al. 2005; Yan and Zhang 2008; Kaynak et al. 2013; Annavaram et al. 2001a; Aamodt et al. 2002; Alameldeen and Wood 2007]
Data prefetching	Almost all others
Use of branch prediction information	[Srinivasan et al. 2001; Reinman et al. 1999; Kadjo et al. 2014; Zilles and Sohi 2001]
Memory-side prefetching	[Hur and Lin 2006; Yedlapalli et al. 2013; Hughes and Adve 2005; Solihin et al. 2003; Yang and Lebeck 2000]
Core-side prefetching	Almost all others

these correlated instructions are again encountered, a prefetch is performed. Since multiple execution paths may lead to one miss, multiple triggering instructions may be stored for that miss. This is useful for cache blocks that tend to show miss on only some of the execution paths. They associate neighboring cache misses with a single instruction to reduce CoR table size. To avoid redundant prefetches, they use a filtering mechanism that uses a confidence-counter scheme to retire ineffective correlations. This reduces unbeneficial prefetches and also obviates the need of probing the I-cache before prefetching.

Spracklen et al. [2005] analyze I-cache miss behavior of commercial workloads that exhibit high miss rates in both L1 and L2 I-caches. They show that eliminating misses due to discontinuous accesses is as important as removing those due to sequential accesses. They propose a discontinuity prefetching technique that can be used together with sequential prefetching for removing both types of misses. Control transfer instructions such as functional calls and branches lead to discontinuity in instruction fetch by transitioning to a non-sequential address. When such a transition leads to an I-cache miss, their technique inserts this discontinuity information in a table. The prefetcher moves ahead of the demand fetch stream, and when it finds a valid table entry, it prefetches the target of discontinuity.

Ferdman et al. [2008] note that as repeated traversals of program data structure lead to repetitive data-cache miss sequences, repeated traversals of CFG lead to repetitive sequences of I-cache misses. Also, almost all I-cache misses can be associated with these recurring sequences, termed *temporal instruction streams*. Their technique dynamically tracks temporal instruction streams themselves and records them in L2 cache. Afterwards, their technique predicts when these streams would repeat and, based on this, prefetches instructions before the demand requests are made. The techniques (e.g., Spracklen et al. [2005] and Reinman et al. [1999]) that use a branch predictor to traverse a program's CFG to predict discontinuous control flow have a limited lookahead distance since they use a branch predictor and work on basic-block sequence. By comparison, their technique works directly on I-cache misses and does not explore program's CFG, and, hence, it achieves higher lookahead distance, BW efficiency, and prefetching accuracy.

Ferdman et al. [2011] note that control-flow variations are amplified by microarchitectural components and these, along with random HW interrupts, lead to non-repetitive instruction history that degrades the effectiveness of conventional prefetching techniques. For example, control-flow variations may disturb the branch predictor state and L1 I-cache replacement sequence, leading to randomness of instruction stream, different miss sequences, and execution of wrong-path instructions. To address this, they propose using a correct-path, retire-order instruction sequence to track an accurate sequence of instruction-fetch. This provides a near-ideal recurring

instruction stream and eliminates the randomness introduced due to branch predictor, cache, and interrupts. On encountering a recorded address, a prefetch is triggered for subsequent requests based on replaying a recorded sequence beginning with the most recent position of the recurring address in the sequence. Further, instead of storing individual accesses, recording temporally and spatially correlated groups of accesses enables compact storage of history. Thus, only one address is stored per spatial region (e.g., a function), and storage of multiple iterations of tight loops is avoided. Their approach improves the prefetching coverage and accuracy and enables the L1 I-cache to achieve nearly 100% hit rate.

Kolli et al. [2013] note that the current call stack reflects the execution path of the program traversed to reach an execution point and the program context captured by it has strong correlation with L1 I-cache misses. Further, the return address stack (RAS) concisely summarizes the program context. On any call or return operation, their technique saves the RAS state into a signature consisting of current call stack along with direction/destination of call or return operation. These signatures are strongly correlated with L1 I-cache misses, and, hence, a sequence of misses seen for any signature is associated with that signature. Signatures constructed with fine-grained context information allow prefetching of the traversals deep into the call graph and within large functions. Further, they show that RAS signatures have high predictability and, thus, prediction of subsequent signatures can be accurately done based on current signatures, which provides higher prefetch lookahead. Their technique reduces storage and energy overhead of accurate HW instruction prefetching (incurred in Ferdman et al. [2011]), while still maintaining high prefetcher coverage and program performance.

4.5. Use of Branch Prediction or History Information

We now discuss some techniques that use information from branch prediction to speculate on program control flow for determining the data to be prefetched.

Srinivasan et al. [2001] present an instruction prefetching scheme that correlates branch instruction execution with misses in I-cache, based on the fact that control-flow alterations due to branches cause I-cache misses. Based on this, branch instructions are used to trigger prefetching of instructions that appear in the execution after a fixed number (say, K, for example, $K = 4$, etc.) of branches. For instance, a candidate basic block (BB_1) will be associated with a branch instruction (R_1) if an I-cache miss to BB_1 happens exactly K branches after the execution of R_1 occurs. On future execution of R_1, BB_1 will be prefetched. Thus, their technique avoids the need of a branch predictor to estimate the result of $K+1$ branch operations. Also, since another basic block starts at the branch instruction target address and may last several cache lines, their technique also stores the length of prefetch candidate blocks along with their addresses to prefetch the entire blocks in a timely manner.

Zilles and Sohi [2001] note that a few frequently executed static instructions (called problem instructions) cause a majority of branch mispredictions and cache misses. Their technique creates a code portion, called a *speculative slice*, that mimics the computation including the problem instruction and includes only those operations that are necessary to compute the outcome of problem instruction. By forking such slices well before the problem instruction, data prefetching can be done to avoid the penalty of misses.

4.6. Memory Side Prefetching

Solihin et al. [2003] present a technique where CoR prefetching is performed by a user thread running in main memory, and the prefetched data are sent to the L2 cache. L2 cache misses are tracked and recorded in a CoR table. Afterwards, for each miss, the CoR table is looked up and a prefetch of several lines is triggered for the L2 cache.

The CoR table is stored in main memory and, thus, changes to L2 cache are minimal. They show that by combining their technique with a core-side sequential prefetcher, the performance improvement can be increased further. Also, the prefetch algorithm used by the thread can be adapted on a per-application basis.

Yedlapalli et al. [2013] present a memory-side prefetcher (MSP) that fetches data on-chip from memory but, unlike in Solihin et al. [2003], does not push the data to the caches and, thus, avoids resource contention. They use a next-line prefetching scheme and prefetch when a row buffer hit occurs such that, first, the demand request is served and then the prefetch request is served. Successive requests to lines in that row then turn into prefetch hits. Data are maintained in a separate buffer at each memory controller and, thus, access to prefetched data does not use memory channel or bank, which lowers the queuing delays. The advantage of their MSP is that it can utilize knowledge of memory state to reduce row-buffer conflicts for reducing the miss latency itself, in addition to reducing the number of memory accesses. Also, unlike a core-side prefetcher, an MSP can easily adapt to the available memory BW. To achieve larger performance improvement, MSP can be integrated with a core-side prefetcher, whereby the former brings data on-chip and the latter leverages core request predictability to access the data brought by MSP without issuing an off-chip request.

Hughes and Adve [2005] present a prefetching technique where the prefetch engine resides on the processor-in-memory. A local or remote processor provides a summary of LDSs and likely traversals. Based on it, the prefetcher performs traversal independently and sends the data to the requesting processor. By virtue of its proximity to memory, their prefetcher provides faster service than a processor-side prefetcher, and this allows their prefetcher to run ahead of the processor, thus bringing data in advance of the processor access and pipelining data transfer over the network. They compare their prefetcher to a processor-side prefetcher by using programs with significant LDS memory stall time. They observe that both their prefetcher and the processor-side prefetcher provide better performance on different applications that highlight the need to combine them for optimal performance.

5. EVALUATION USING REAL SYSTEMS AND ANALYTICAL MODELS

Different experimentation approaches/platforms offer complementary insights, for example, real hardware allows quick and realistic evaluation, whereas simulators offer high flexibility to compare with different configuration/parameter choices that may even be unrealizable on real systems. Analytical models provide insights independent of a particular platform or program. Clearly, it is revealing to note the evaluation approach of a study, and, hence, Table V classifies the works based on this feature. We now discuss some works that use real processors or theoretical models for evaluation.

5.1. Studies Using Real Processors

Wu and Martonosi [2011] propose a technique that dynamically turns on/off the prefetcher, based on the LLC interference caused by it. They perform experiments on an Intel Core i7 processor, where, of the four different HW prefetchers, two prefetchers, viz. MLC spatial and MLC streamer prefetchers, can be externally controlled using SW-accessible state registers. The samples for application LLC miss counts are collected using Intel's Precise Event Based Sampling (PEBS) capability. Their technique works in two phases. In the profiling phase, the time taken to see a fixed number of LLC misses is observed with prefetchers turned on and off, respectively. If this time is higher when prefetchers are turned off, then it indicates that misses happen less frequently in time without the prefetchers, and, hence, prefetchers are turned off in the run phase; otherwise, the prefetchers are turned on. For a small implementation overhead, their technique effectively mitigates prefetching-induced LLC interference.

Table V. Experimentation Platform/Approach Used for Evaluating Prefetching Techniques

Classification	References
Real system	[Beyler and Clauss 2007; Khan et al. 2014; Marathe and Mueller 2008; Kang and Wong 2013; Huang et al. 2012; Jiménez et al. 2012; Lu et al. 2003, 2005; Chilimbi and Hirzel 2002; Wu and Martonosi 2011; Mehta et al. 2014; Lee et al. 2012; Guttman et al. 2015]
Analytical model	[Liu and Solihin 2011; Chen and Aamodt 2008]
Simulator	Almost all others

Jiménez et al. [2012] study prefetching on an IBM POWER7 processor. They first examine the performance and power consumption of a prefetcher for its different parameter settings using microbenchmarks and standard benchmarks. These settings include prefetch degree, stride-N (whether streams with a fixed stride greater than one cache line are prefetched), whether prefetching is done on store operations and whether the prefetcher is enabled/disabled. Since the optimal prefetcher setting varies between and within benchmarks, they propose an adaptive prefetch technique that dynamically configures the prefetcher parameters based on benchmark characteristics. Their technique works in two phases. In the exploration phase, different prefetcher parameter settings are tried, and the one that provides the largest IPC is used in the running phase. Since phase changes may also lead to a change of IPC, instead of comparing with individual measurements, they use a moving average buffer to record most recent Q IPC values for each setting and then compare different settings by using average values in the buffer. To avoid the impact of inefficient settings on performance, their technique does not try them for K exploration phases, where K is determined by the slowdown introduced by those settings. Thus, inefficient settings are penalized by being dropped from exploration.

Kang and Wong [2013] note that prefetching leads to contention for shared cache and bandwidth, and virtualization also complicates shared cache management by consolidating multiple VMs on a single CMP. Hence, a study of their interaction can allow effective LLC sharing among VMs and deciding whether prefetching should be enabled. They study HW prefetching in virtualized environments and account for various virtualization factors, such as number of vCPUs and VMs, interference between VMs and vCPU-core binding, and so on. They show that, for most configurations, the negative influence of prefetching is small, and prefetching degrades overall performance significantly only in few configurations. To avoid destructive interaction between the two approaches, they propose a prefetching-aware vCPU-core binding technique. Based on the cache access pattern (including both demand and prefetch requests) of every VM's workload, VMs are categorized into groups showing different cache sharing constraints. For example, a VM with a very low miss rate has the lowest constraint, while one with a high miss rate and showing negative self-prefetching influence (self-prefetching influence shows whether prefetching benefits or harms performance on consolidating it with the same application in a different VM) has the highest constraint. The higher the constraint, the higher the priority of allocating to a dedicated cache to avoid the negative impact of prefetching. VMs with lower constraint are given more shared caches, since they do not cause contention. Thus, based on the cache sharing constraints, scheduling of the vCPUs of every VM on suitable cores is done to improve performance.

Guttman et al. [2015] study HW and SW prefetching and their interaction on the Xeon Phi system. They show that when HW and SW prefetchers work together, SW prefetch misses train HW prefetchers directly, and the demand misses generated by SW prefetching train HW prefetchers indirectly. Further, when the SW prefetcher is effective in removing the majority of L2 demand misses, the HW prefetcher is not triggered frequently, and, thus, it is throttled. In this way, the HW prefetcher effectively

adapts itself to the SW prefetcher, and the coordinated HW+SW prefetching provides the best performance by virtue of prefetching for a wide variety of access patterns. For some applications, compiler-inserted prefetch instructions are ineffective, and, hence, those inserted manually by the programmer alone are effective for prefetching. For other applications, compiler-inserted instructions are sufficient, and, hence, programmer effort is not worthwhile. When prefetching improves performance, the resulting energy savings offset the energy overhead due to extra instruction/metadata and memory operations, and, thus, the overall effect of prefetching on energy is positive.

5.2. Studies Using Analytical Models

Liu and Solihin [2011] study the interaction of prefetching and bandwidth partitioning (BPT) and their impact on system performance. Based on CPI model and queuing theory, they develop an analytical model that takes as input several key system parameters (e.g., frequency, available BW, and cache block size) and application cache behavior indices (e.g., prefetching accuracy and coverage and prefetching frequency). Their model provides a composite prefetching metric that determines conditions in which prefetching improves performance. This metric is shown to be more effective than traditional metrics such as coverage and accuracy. Their BPT model accounts for prefetching and determines BW partitions for each core that lead to optimal performance. Based on their model, they derive several important insights and conclusions. Use of prefetching reduces the available BW, which enhances the role of BPT in improving performance. In BW-constrained systems, performance loss due to prefetching cannot be alleviated by BPT, and, hence, the decision of (de)activating prefetchers needs to be made before using BPT. Also, when the prefetcher of every core is activated, naively providing large BW to a core that performs accurate prefetching does not lead to highest performance (weighted speedup). Instead, BW partition of this core should be more constrained; this is because, by virtue of utilizing the BW efficiently, this core can donate some BW to other cores for improving overall performance.

Chen and Aamodt [2008] present an analytical model to evaluate the impact of HW prefetching, pending cache hits, and limited MSHR resources on superscalar processor performance with long latency memory systems. A CPI stack divides the application CPI into CPI from useful computation and that from miss events, for example, cache misses, branch mispredictions, and so on, and, of these, their technique models CPI from D-cache misses. Under no prefetching, the instruction trace produced by a cache simulator is analyzed to see misses in the cache. Under prefetching, several loads that might have seen misses become hits or pending hits (a memory reference where the cache block has been requested but the data have not arrived), depending on whether prefetching can fully hide the memory latency. A pending hit may be due to a demand miss or a prefetch miss. For each pending hit, they identify a previous instruction that brought the current instruction's required data into cache. The latency of the current instruction that can be hidden is computed as the number of instructions between the current and previous instructions divided by the processor's issue width. Then the actual latency of the current instruction is the delta between memory access latency and hidden latency; if memory latency is fully hidden, then the actual latency is zero. Using this approach, CPI due to D-cache misses under prefetching is estimated.

6. REDUCING IMPLEMENTATION AND PERFORMANCE OVERHEAD OF PREFETCHING

As discussed in Section 2.7, a careful choice of design parameters is required to reduce area/latency overheads of prefetching and its negative impact on performance. We first summarize some approaches proposed for this (Section 6.1) and then discuss several of these approaches (Sections 6.2 to 6.7).

6.1. An Overview of Approaches

(1) Controlling negative impact of prefetching:

—To avoid cache pollution, prefetched data can be stored in separate buffer(s) [Hur and Lin 2006; Yedlapalli et al. 2013; Jouppi 1990; Falcón et al. 2005; Zhang et al. 2002; Cantin et al. 2006; Somogyi et al. 2009; Joseph and Grunwald 1997; Roth et al. 1998] or in place of dead blocks [Lai et al. 2001].
—Some techniques temporarily disable the prefetcher (globally or for certain loads) [Kandemir et al. 2009; Yu and Liu 2014; Wu and Martonosi 2011; Jiménez et al. 2012; Alameldeen and Wood 2007; Kadjo et al. 2014; Nesbit et al. 2004], while others adapt its aggressiveness [Srinath et al. 2007; Jiménez et al. 2012; Nesbit et al. 2004; Zhang et al. 2007; Mehta et al. 2014; Ebrahimi et al. 2009; Hur and Lin 2006, 2009; Wang et al. 2003; Zhang et al. 2006; Albericio et al. 2012; Yu and Liu 2014]; for example, the prefetch degree/distance may be adapted. Some works use multiple prefetchers [Marathe and Mueller 2008; Guo et al. 2011] or switch between different prefetching modes [Sharma et al. 2005]. Some works start actual prefetching only after the prefetcher accuracy has been confirmed [Pugsley et al. 2014].
—Prefetch accuracy can be ascertained by seeing whether the prefetched or evicted block is accessed first [Alameldeen and Wood 2007; Dang et al. 2013; Kandemir et al. 2009].
—The position of prefetched blocks in the LRU stack can be controlled [Lin et al. 2001a; Srinath et al. 2007; Wu et al. 2011; Wang et al. 2003; Lin et al. 2001b]; for example, they can be placed near LRU to avoid pollution.
—Redundant prefetches can be avoided [Spracklen et al. 2005; Zhang et al. 2002; Marathe and Mueller 2008; Guo et al. 2011; Reinman et al. 1999; Kim et al. 2014]; for example, a prefetch request can be canceled if it has been recently demand fetched or if it matches an upcoming demand fetch request [Spracklen et al. 2005]. Similarly, in parallel applications where per-core prefetchers may redundantly prefetch shared data, only the longest stream that is beneficial for shared data should be prefetched [Kim et al. 2014].
—Prefetch filtering can be driven by the number of cache misses [Wu and Martonosi 2011; Albericio et al. 2012] or IPC [Jiménez et al. 2012; Nesbit et al. 2004].
—Data brought by certain cores can be pinned in cache if they are frequently the victim of harmful prefetches [Kandemir et al. 2009].
—Strategies for addressing inter-core interference can be used [Wu and Martonosi 2011; Ebrahimi et al. 2009; Yu and Liu 2014; Kandemir et al. 2009; Albericio et al. 2012] (refer to Section 6.3).

(2) Improving effectiveness of prefetching:

—Main memory policies can be made prefetch aware [Lee et al. 2008; Yedlapalli et al. 2013; Hur and Lin 2006; Lin et al. 2001a, 2001b; Ebrahimi et al. 2011]; for example, a DRAM controller can give different priorities to useful and useless prefetches and demand requests.
—Several works classify the miss stream into different categories to drive the prefetching algorithm or to gain insights [Sair et al. 2002; Iacobovici et al. 2004; Spracklen et al. 2005; Zhu et al. 2010].
—Since prefetching is only useful if a helper thread runs ahead of the main thread, helper threading may be applied only for those loops where it can prefetch delinquent loads on a loop's critical path [Lu et al. 2005]. The helper thread can be terminated once the main thread makes equal progress [Aamodt et al. 2002].
—For higher effectiveness, prefetching can be triggered by dead-block prediction (and not cache miss) [Lai et al. 2001] or branch instructions [Srinivasan et al. 2001].

—For achieving higher accuracy, Chilimbi and Hirzel [2002] prefetch a hot data stream only if it is sufficiently long (e.g., has more than 10 unique references).
—Some techniques primarily focus on those misses that have a bigger (or direct) impact on performance [Chou 2007; Manikantan and Govindarajan 2008]. Several other techniques focus on delinquent loads (e.g., Khan et al. [2014], Collins et al. [2001b], Lu et al. [2005], Zhang et al. [2006], Zilles and Sohi [2001], and Lu et al. [2003]).
—To improve accuracy of prefetching, access history can be stored in a GHB structure [Nesbit and Smith 2004; Nesbit et al. 2004; Diaz and Cintra 2009; Manikantan et al. 2011; Manikantan and Govindarajan 2008].
—Prefetching can be integrated with other approaches, for example, compression [Alameldeen and Wood 2007; Raghavendra et al. 2015], virtualization [Kang and Wong 2013], cache insertion/promotion [Wu et al. 2011], and BW partitioning [Liu and Solihin 2011].

(3) Reducing latency overhead:

—Prefetching can be done only when the processor/bus is idle [Chou 2007; Wang et al. 2003; Lin et al. 2001a], and a helper thread may be run only when idle HW is available [Luk 2001; Collins et al. 2001b; Aamodt et al. 2002].
—The helper thread can run a reduced version (e.g., Zilles and Sohi [2001], Annavaram et al. [2001b], Collins et al. [2001a, 2001b], and Balasubramonian et al. [2001]) of the program instead of the full version [Luk 2001], since its actual results are usually ignored.
—BF can be used for early hit/miss determination [Peir et al. 2002], storing candidate prefetched blocks [Pugsley et al. 2014], and estimating prefetching-caused pollution [Srinath et al. 2007] (refer Section 6.4).
—Using sampling, only a fraction of memory instructions may be tracked [Khan et al. 2014], which reduces profiling overhead.

(4) Reducing storage overhead:

—Prefetcher metadata may be stored off-chip [Wenisch et al. 2005; Solihin et al. 2003; Chou 2007; Liu et al. 2012; Ferdman and Falsafi 2007; Wenisch et al. 2009; Somogyi et al. 2009; Burcea et al. 2008] or in the cache hierarchy itself [Burcea et al. 2008].
—Metadata can be stored in compressed form [Liu et al. 2012].
—Correlated accesses or misses can be stored together and not separately [Zhang et al. 2002; Ferdman et al. 2011].
—Instead of address-based correlation, tag-based correlation can be used [Hu et al. 2003; Sharma et al. 2005].
—Timing information can stored in terms of miss-counter instead of CPU cycles [Zhu et al. 2010], which also provides a more accurate and stable measure of time.
—Access history can be shared between cores [Kaynak et al. 2013], and resource sharing can be used in helper-thread prefetching [Lu et al. 2005].

6.2. Adapting Aggressiveness of Prefetcher

Lin et al. [2001b] present a technique to filter useless prefetches by predicting spatial locality in a memory region. A bit vector, called a *density vector* (DV), is used to record which cache blocks in a memory region were fetched during an epoch. The epoch for a region ends and its next epoch begins when a miss to a block happens that is already in density vector. The number of bits in a DV that are "1" (i.e., set) shows available spatial locality, and the longest consecutive string of set bits shows whether the access pattern is dense. CoR between the two DVs is defined as a fraction of identical bits between them, and the local-CoR is defined as the CoR between the two most recent epochs in a region. They note that, for most programs, local-CoR is strong, which indicates that, over time, access patterns in a region do not change. Using this, they design a filter that tracks current DVs and also stores previous DVs. It exploits local-CoR between

DVs to filter out prefetch requests that it predicts will not be useful for a given region and epoch. They show that their technique can eliminate a large fraction of useless prefetches without harming performance.

Hur and Lin [2006] present an adaptive stream detection (ASD) technique that modulates the aggressiveness of prefetch policy based on the spatial locality present in the workload. The prefetcher runs in the memory controller and brings data in a prefetch buffer. On detecting access to k successive cache lines, a stream prefetcher begins prefetching from $(k + 1)$th line onwards, until it detects a useless prefetch. Generally, k is chosen at design time. However, for short streams, a stream prefetcher becomes ineffective. For example, for applications where every stream has a length of 2, using $k = 1$ leads to one useless prefetch after every useful prefetch. Their technique associates every memory access with a suitable stream length to generate a stream length histogram (SLH). As an example, if most memory requests occur in streams of length 2, then their technique only prefetches the second and not the third line of a stream. This adaptive stream detection approach avoids useless prefetches. By dynamically adapting the histogram, changes in application behavior can be accounted for. Their technique extends the scope of a stream to include even those having only two cache lines, which allows several commercial applications to be viewed as stream based and, thus, enable the use of a low-overhead stream-based prefetcher with them.

Hur and Lin [2009] present three approaches to further improve the effectiveness of the ASD stream buffer [Hur and Lin 2006]. To improve the stream detection mechanism of the stream buffer, they use a length-based stream detection approach, whereby, with increasing stream length, the time duration for which the stream filter waits for the next element of a stream is reduced, for example, for streams of lengths 1, 2 and 3, the wait time can be T, $T/2$, and $T/4$. If the next element does not arrive by this time, then the stream filter becomes available for allocation to a new stream, since the loss from not capturing the last element in the stream is higher for smaller streams than in long streams. This approach gives a greater chance for shorter streams to be fully prefetched, which increases the number of streams that can be gainfully prefetched in irregular applications. Since SLHs vary over time, their second approach uses an adaptive epoch length for the SLH feedback mechanism, based on whether SLHs observed in consecutive epochs are similar. The third approach uses information from SLH to prefetch a variable number of blocks at a time. When N consecutive memory requests are likely to appear in a burst, and the memory queue is not too busy, a maximum of N consecutive prefetch requests are generated, which helps in avoiding late prefetches.

Srinath et al. [2007] present a technique to dynamically adapt the aggressiveness of a prefetcher. In each interval, they monitor three metrics, viz. prefetcher accuracy, lateness, and the cache pollution due to prefetching. Prefetch accuracy is estimated by tracking the fraction of prefetched blocks that lead to hit of demand requests. A late prefetch is identified when the data at a prefetched address are requested by the core but the data have not arrived. Cache pollution is measured by using a BF–based predictor that estimates the demand-fetched L2 cache blocks evicted due to prefetched data. Each of the three metrics are classified in different ranges (e.g., high and low) by using individual thresholds for them. Based on them, prefetching parameters viz. prefetch degree and prefetch distance are adjusted. For example, if prefetches are accurate and do not cause pollution, then both the parameters are increased to amplify the aggressiveness of prefetching. Similarly, if prefetching causes large amounts of pollution, then, in the next interval, prefetched blocks are inserted near the LRU position, while in the case of low pollution, they are inserted into the midway position in the LRU chain.

Albericio et al. [2012] present ABS, an adaptive controller design that uses a hill-climbing approach for modulating the aggressiveness of prefetchers in a banked shared LLC. In their design, each LLC bank has a prefetch engine and an ABS controller that collects bank-local statistics. Prefetchers and ABS controllers of different banks are independent, which avoids the need of communication between them and requires looking up only a local bank for reducing useless prefetches. In every epoch, the aggressiveness of only one core's prefetcher is changed, and the miss ratio of the bank (computed as demand misses divided by demand requests of all the cores) is computed. The miss ratio in the current epoch is compared to that in a reference epoch. If the miss ratio in the current epoch is higher, then the change in the prefetcher aggressiveness is undone; otherwise, it is confirmed. Also, the accuracy of a core's prefetcher is computed by computing the ratio of the hits from the core being probed to prefetched blocks and the total number of prefetches issued by this core. If this accuracy value is lower than a threshold, then the aggressiveness of the prefetcher is decreased. Thus, each core's prefetcher can have dissimilar aggressiveness values in different LLC banks. They show that their technique improves performance and fairness.

Nesbit et al. [2004] present a prefetching technique based on monitoring delta correlations in memory zones. They divide the memory address space into concentration zones (CZones) of equal sizes. Within each CZone, patterns in miss address deltas (difference between successive addresses) are detected using a GHB. A prefetch is triggered when an access pattern is detected within a CZone. Their technique does not require a PC for load instructions that lead to misses. Since programs show different phases, the values of the prefetcher parameters (viz. CZone size and prefetch degree) that provide the best performance also change over the program execution. Hence, they also present an adaptive version of their technique. This technique begins in the UNSTABLE state with certain values of parameters. If, after an interval of fixed instructions, a phase change is detected, then the algorithm stays in the UNSTABLE state; otherwise, it switches to the TUNING state. In this state, the algorithm tries several parameter values (and "no prefetching"), and, at the end, the values providing the highest performance are selected, and the algorithm switches to a STABLE state. A phase change again switches the algorithm to the UNSTABLE state and resets the configuration. Adaptivity also allows us to turn off prefetching in case it harms the performance.

6.3. Mitigating Inter-Core/Thread Interference

Kandemir et al. [2009] propose two techniques for filtering harmful prefetches in a multi-core processor that causes inter-core interference and replaces useful data. They note that in any execution phase, a small set of cores brings or get affected by harmful prefetches, and such patterns change over different execution phases. Their first technique selectively suppresses prefetches from certain cores. The harmful prefetches from each core are recorded by tracking the block replaced by a prefetched block and checking whether the prefetched or discarded block is accessed first in later execution. At the end of each phase, if the ratio of harmful prefetches issued by a core and total harmful prefetches exceeds a threshold, then prefetches from a core are suppressed for a single (next) phase. The second technique records the cache misses seen by a core due to harmful prefetches and if, in a phase, the ratio of such misses seen by a core and the total such misses seen by all cores exceeds a threshold, then data blocks fetched by that core in the cache are marked as non-removable for a single (next) phase. Instead of the blocks from this core, those from other cores that are least recently used are selected. Thus, this technique aims to mitigate the negative effect of prefetching on cores that are harmed the most by it. By removing the effect of harmful prefetches, their techniques enable leveraging the benefit of SW prefetching, even for large core-counts.

Yu and Liu [2014] note that in a multicore system running multi-threaded application, different threads share data and, hence, if prefetched data blocks replace demand-fetched data blocks, all the sharers of that cache block need to invalidate their local copy. Thus, prefetching can lead to inter-thread invalidations and cause contention in shared resources such as LLC and main memory. A prefetching request that causes demand miss in an L1 cache due to invalidation of data block in its sharer's L1 caches is termed an *attacking prefetch request*. For each thread, they utilize a prefetcher that can prefetch both sequential and chained stream patterns and they use filtering mechanisms for each of the two patterns. For sequential streams, the data of all attacking prefetches are ignored, and only the address is stored in the pattern table for later use. For chained patterns, if L1 prefetching misses are found to be attacking prefetches, then they are not immediately ignored. Instead, they are issued and the linked pattern streams are maintained based on the return value. In other words, return value is used to compute the next node address in the linked stream, and then it is ignored (i.e., not moved to cache). Further, based on the runtime feedback about memory requirement and prefetching effectiveness of each thread, their technique adapts the mode and aggressiveness of its prefetcher. For example, for applications with high memory intensity but low prefetching intensity, the prefetcher can be temporarily shut down to reduce contention on shared resources. For applications showing high accuracy and intensity of prefetching, the aggressiveness of the prefetcher is increased to maximally improve performance. By contrast, the aggressiveness of a prefetcher showing low accuracy is reduced.

6.4. Reducing Latency Overhead by Using a Bloom Filter (BF)

A Bloom filter [Bloom 1970] is a probabilistic algorithm that uses multiple hash functions to quickly check whether an item is certainly a non-member or may be a member of a set. By slightly sacrificing the accuracy, BF significantly speeds up the classification process. We now discuss prefetching techniques that use BF.

Peir et al. [2002] present a BF-based predictor that predicts whether an access is a miss or may be a hit. By making this decision early in the pipeline, data can be prefetched in L1 in a timely and accurate manner. The first one, called *partitioned-address*, splits the line address into M partitions. If single or multiple address partitions in a requested line's address do not belong to a corresponding address partition of any cache line, then a cache miss is ascertained. In the second design, called *partial-address*, a bit array is indexed using the least-significant bits of the line address. Every bit shows whether a match is found between the partial address and any corresponding partial address of a cache line. If no match occurs, then a miss is identified. Once a miss is identified in L1 cache, a miss request can be issued to L2 cache to prefetch data in the L1 cache. They also use their technique for speculatively scheduling dependent instructions for boosting performance. They show that accuracy of their predictor is very close to 100%, and their technique achieves significant performance improvement.

Pugsley et al. [2014] present a technique to enable the use of aggressive prefetchers, while avoiding their limitations, such as BW wastage. Their technique tracks prefetch requests generated by a candidate prefetch pattern but does not actually prefetch requests. The addresses of all the cache blocks that would have been prefetched by a prefetch pattern, are stored in a "sandbox" that is implemented using BF. By comparing subsequent cache accesses against the addresses in the sandbox, both the accuracy of prefetcher and existence of prefetchable streams are ascertained. Only when accuracy of a prefetcher exceeds a threshold, actual prefetches are performed. By virtue of prefetching only after confirming, their technique avoids harmful prefetches and can confidently issue many prefetches along that pattern to improve performance by avoiding late prefetches.

6.5. Reducing Energy Overhead of Prefetching

Guo et al. [2011] present techniques to offset energy overheads of prefetching due to extra cache lookups (to avoid redundant prefetching) and prefetch-HW. Their first technique uses compiler to identify memory accesses such as scalar accesses, which are not advantageous for prefetching. Then, only accesses such as those to LDSs and arrays are passed to the prefetcher to reduce prefetch-HW lookups. The second technique annotates pointer and array accesses using a compiler. Based on them, at runtime, a suitable prefetching scheme (such as pointer prefetcher or stride prefetcher) is applied to optimize performance. The third technique reduces prefetch-HW lookups for access patterns with very small strides. For such patterns, a single lookup is performed for their multiple occurrences. A fourth technique reduces prefetching-induced cache tag lookups. It stores the most recently prefetched cache tags in a separate buffer. Each prefetching address is compared with this buffer, and, on a match, the prefetching operation is canceled. Otherwise, a cache tag lookup is performed.

Dang et al. [2013] present a filtering technique for improving the energy efficiency of prefetching. When insertion of prefetched data leads to eviction of an existing data block, their technique records the prefetch-victim address pair. As execution proceeds, either (1) the prefetch address or (2) the victim address may be accessed or (3) the prefetched block may be replaced before being accessed. The first case indicates a useful prefetch while the last two cases indicate a useless prefetch. Thus, based on which address in the pair is accessed first, their technique collects a utilization count of both issued and filtered prefetches. Access to a filtered address indicates incorrect filtering of a useful prefetch. In such a case, its address is recorded to develop a feedback mechanism for avoiding filtering of future useful prefetches. Based on these, the decision about issuing or filtering the prefetch request is taken.

6.6. Reducing Storage Overhead of Prefetchers

Burcea et al. [2008] present predictor virtualization (PV) as a technique to use memory hierarchy for emulating large predictor tables and show its application by virtualizing a spatial memory prefetcher [Somogyi et al. 2006]. Their technique reserves a part of physical memory space for storing the predictor table (PVTable). Another structure, called PVProxy, preserves the interface between the actual optimization technique and the predictor table. It stores a few predictor entries in a small on-chip structure, called PVCache. If an entry is not found in the PVCache, then a memory request is generated to bring the entry. This memory request does not differ from those issued by L1 caches. In this way, PVProxy is transparent to the memory hierarchy. The benefits of virtualizing the predictor are that it avoids wasting a large on-chip space for storing the predictor table, and if the optimization technique is turned off, then the entries of its predictor in L2 cache are soon replaced. Spatial memory streaming [Somogyi et al. 2006] uses two HW structures, viz. a pattern history table (PHT) and an active generation table (AGT). The spatial patterns found by AGT from active memory regions are stored in PHT. In their design, PHT requires 86KB storage, while AGT requires less than 1KB storage. Hence, Burcea et al. apply virtualization to PHT only and bring its on-chip storage requirement to less than 1KB. For this, PHT is itself stored in main memory (PVTable), and a few sets from it are stored in PVCache, which are properly delivered to the prefetching engine by PVProxy.

Ferdman and Falsafi [2007] note that, given the large-sized CoR tables, storing them on-chip forces limited size and reduced coverage, while storing them off-chip reduces prediction lookahead and increases prediction latency. To bring the best of the two together, CoR data are recorded off-chip in order that they will be used and are streamed to a small-sized on-chip table just before their use. In their technique,

long-recurring sequences of consecutively used last-touch signatures are stored off-chip, and, for each sequence, only one head signature is stored on-chip. When an access sequence repeats, its associated last-touch signature sequence is streamed from off-chip to on-chip table. Thus, their technique facilitates timely signature retrieval from off-chip and keeping the size of on-chip table small. They show that last-touch order of blocks can be approximated by a sequence of block evictions. They show that their technique reduces L1D cache misses with minimal on-chip storage overhead and off-chip traffic increase.

Kaynak et al. [2013] note that for processors with many simple (lean) cores, the storage overhead of stream-based prefetching techniques (e.g., Ferdman et al. [2008, 2011]) becomes very high. For homogeneous server workloads that execute similar requests on all cores, instruction access sequences generated on the cores have high (e.g., 90%) similarity. They propose that commonality and recurrence of instruction-level behavior across the cores can be leveraged to produce a single instruction history shared among all cores running the workload. This amortizes the area overhead of sophisticated instruction prefetchers. One randomly chosen core, termed the *history generator core*, generates the instruction fetch stream history. This is stored in the shared history buffer and read by the stream address buffer, which is private to each core and issues prefetches in coordination with I-cache misses. Since keeping a separate history buffer introduces several overheads (such as dedicated storage and logic), they propose embedding the history buffer in the LLC using the PV approach [Burcea et al. 2008]. The large capacity of LLC also allows us to easily support workload consolidation, where one history buffer can be allocated for each workload to maintain per-workload history.

6.7. Interaction of Prefetching with Other Approaches

Alameldeen and Wood [2007] study the interaction of prefetching with cache and BW compression and show that these approaches work synergistically to provide large performance improvement. This is because prefetching partially hides the decompression latency, and compression reduces the BW contention caused by prefetching. Also, compression increases the effective cache capacity that facilitates bringing more blocks into cache using prefetching. They also propose an adaptive prefetching mechanism to throttle prefetching when it hurts performance. For each of the private L1 caches and shared L2 cache, they use a saturating counter, which is decremented on a useless or harmful prefetch and incremented on a useful prefetch. Initially, the counters have their largest value. When they reach zero, prefetching is disabled for that cache. They use a "prefetch bit" for each cache block, which is set at the time a prefetched line is inserted in cache. On first access to this block, the bit is reset and counter is incremented. If an evicted line still has its prefetch bit set, then the prefetch is considered useless and the counter is decremented. For detecting harmful prefetches, they utilize additional tags employed for cache compression and store the addresses of replaced blocks in them. On a cache miss, each of the invalid tags in the cache set is examined in the LRU stack sequence. If a match is found, then the line is replaced by a currently cached line, but if any valid line has its prefetch bit set, then this harmful prefetch is assumed to have evicted the line, and the counter is decremented.

Wu et al. [2011] note that conventional cache replacement schemes give identical treatment to prefetch and demand requests, and, hence, in the presence of prefetching, they may not provide expected performance improvement. Their cache management approach dynamically predicts and mitigates the cache interference due to prefetching by altering the insertion and hit promotion schemes of the cache such that they handle prefetch and demand requests differently. They demonstrate their approach by utilizing prefetch/demand request information in improving re-reference

predictions generated by the Dynamic Re-Reference Interval Prediction (DRRIP) replacement policy. Thus, their technique synergistically integrates prefetching with intelligent cache replacement schemes. Also, for multicore multiprogrammed workloads, their technique eliminates inter-core and intra-core prefetch-induced interference in shared LLCs.

7. CONCLUSION AND FUTURE OUTLOOK

As key applications become even more data intensive and power and thermal budgets reach a plateau, effective latency hiding mechanisms such as prefetching are becoming more attractive than costly alternatives such as increases in cache size. However, several challenges remain to be addressed for fully realizing the potential of prefetching in next-generation computing systems. We now discuss a few of them.

In recent years, researchers have explored alternative memory technologies (e.g., non-volatile memories that are write-agnostic, gigabyte-size DRAM caches that may employ cache line sizes of few KBs [Mittal et al. 2015]) and fabrication approaches (e.g., 3D stacking which demands intelligent data placement and thermal management), and so on. These trends present several new constraints and optimization opportunities for prefetching. Also, while conventional prefetching techniques only optimize performance, higher-level objectives such as quality of service (QoS), relative applications priorities, and so on, also become important with the rising number of cores. These factors call for re-evaluation of traditional techniques and design of novel techniques for ensuring effective prefetching, and this will be an interesting direction for researchers in coming years.

Exascale systems seek to achieve 10^{18} computations per second within an energy budget of 20MW. It is clear that no single approach can bridge the performance and power efficiency gap between the current and the future systems. Hence, achieving synergistic integration of prefetching with other approaches, such as data compression, near-threshold voltage computing, dynamic voltage/frequency scaling (DVFS), and so on, will be vital and present a key challenge for computer architects.

In this article, we presented a survey of recent prefetching techniques for caches. We identified tradeoffs in the use of prefetching and the challenges that merit further investigation. We classified the works along several dimensions to highlight major research directions. It is hoped that this article will help the readers see the prefetching techniques in synthesis and understand their potential in improving performance of future processors.

REFERENCES

Tor Aamodt, Pedro Marcuello, Paul Chow, Per Hammarlund, and Hong Wang. 2002. Prescient instruction prefetch. In *Workshop on Multithreaded Execution, Architecture and Compilation*. 2–10.

Alaa R. Alameldeen and David A. Wood. 2007. Interactions between compression and prefetching in chip multiprocessors. In *HPCA*. 228–239.

Jorge Albericio, Rubén Gran, Pablo Ibánez, Víctor Viñals, and Jose María Llabería. 2012. ABS: A low-cost adaptive controller for prefetching in a banked shared last-level cache. *ACM Trans. Arch. Code Opt.* 8, 4 (2012), 19.

Murali Annavaram, Jignesh M. Patel, and Edward S. Davidson. 2001a. Call graph prefetching for database applications. In *International Symposium on High-Performance Computer Architecture*. 281–290.

Murali Annavaram, Jignesh M. Patel, and Edward S. Davidson. 2001b. Data prefetching by dependence graph precomputation. In *International Symposium on Computer Architecture*. 52–61.

Rajeev Balasubramonian, Sandhya Dwarkadas, and David H. Albonesi. 2001. Dynamically allocating processor resources between nearby and distant ILP. In *International Symposium on Computer Architecture*. 26–37.

Jean Christophe Beyler and Philippe Clauss. 2007. Performance driven data cache prefetching in a dynamic software optimization system. In *International Conference on Supercomputing*. 202–209.

Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

Ioana Burcea, Stephen Somogyi, Andreas Moshovos, and Babak Falsafi. 2008. Predictor virtualization. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. 157–167.

Jason F. Cantin, Mikko H. Lipasti, and James E. Smith. 2006. Stealth prefetching. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. 274–282.

Mainak Chaudhuri, Jayesh Gaur, Nithiyanandan Bashyam, Sreenivas Subramoney, and Joseph Nuzman. 2012. Introducing hierarchy-awareness in replacement and bypass algorithms for last-level caches. In *International Conference on Parallel Architectures and Compilation Techniques*. 293–304.

Chi F. Chen, Se-Hyun Yang, Babak Falsafi, and Andreas Moshovos. 2004. Accurate and complexity-effective spatial pattern prediction. In *International Symposium on High Performance Computer Architecture*. 276–287.

Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. 2007. Improving hash join performance through prefetching. *ACM Trans. Database Syst.* 32, 3 (2007), 17.

Tien-Fu Chen and Jean-Loup Baer. 1995. Effective hardware-based data prefetching for high-performance processors. *IEEE Trans. Comput.* 44, 5 (1995), 609–623.

Xi E. Chen and Tor M. Aamodt. 2008. Hybrid analytical modeling of pending cache hits, data prefetching, and MSHRs. In *International Symposium on Microarchitecture*. 59–70.

Trishul M. Chilimbi and Martin Hirzel. 2002. Dynamic hot data stream prefetching for general-purpose programs. *ACM SIGPLAN Notices* 37, 5 (2002), 199–209.

Yuan Chou. 2007. Low-cost epoch-based correlation prefetching for commercial applications. In *International Symposium on Microarchitecture*. 301–313.

Jamison Collins, Suleyman Sair, Brad Calder, and Dean M. Tullsen. 2002. Pointer cache assisted prefetching. In *International Symposium on Microarchitecture*. 62–73.

Jamison D. Collins, Dean M. Tullsen, Hong Wang, and John P. Shen. 2001a. Dynamic speculative precomputation. In *International Symposium on Microarchitecture*. 306–317.

Jamison D. Collins, Hong Wang, Dean M. Tullsen, Christopher Hughes, Yong-Fong Lee, Dan Lavery, and John P. Shen. 2001b. Speculative precomputation: Long-range prefetching of delinquent loads. In *International Symposium on Computer Architecture*. 14–25.

Xianglei Dang, Xiaoyin Wang, Dong Tong, Junlin Lu, Jiangfang Yi, and Keyi Wang. 2012. S/DC: A storage and energy efficient data prefetcher. In *Conference on Design, Automation and Test in Europe*. 461–466.

Xianglei Dang, Xiaoyin Wang, Dong Tong, Zichao Xie, Lingda Li, and Keyi Wang. 2013. An adaptive filtering mechanism for energy efficient data prefetching. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*. 332–337.

Pedro Diaz and Marcelo Cintra. 2009. Stream chaining: Exploiting multiple levels of correlation in data prefetching. In *International Symposium on Computer Architecture (ISCA)*. 81–92.

Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. 2011. Prefetch-aware shared resource management for multi-core systems. *International Symposium on Computer Architecture (ISCA)* (2011), 141–152.

Eiman Ebrahimi, Onur Mutlu, Chang Joo Lee, and Yale N. Patt. 2009. Coordinated control of multiple prefetchers in multi-core systems. In *International Symposium on Microarchitecture*. 316–326.

Philip G. Emma, Allan Hartstein, Thomas R. Puzak, and Viji Srinivasan. 2005. Exploring the limits of prefetching. *IBM J. Res. Dev.* 49, 1 (2005), 127–144.

Ayose Falcón, Alex Ramirez, and Mateo Valero. 2005. Effective instruction prefetching via fetch prestaging. In *International Parallel and Distributed Processing Symposium (IPDPS)*.

Babak Falsafi and Thomas F. Wenisch. 2014. A primer on hardware prefetching. *Synth. Lect. Comput. Arch.* 9, 1 (2014), 1–67.

Michael Ferdman and Babak Falsafi. 2007. Last-touch correlated data streaming. In *International Symposium on Performance Analysis of Systems & Software (ISPASS)*. 105–115.

Michael Ferdman, Cansu Kaynak, and Babak Falsafi. 2011. Proactive instruction fetch. In *International Symposium on Microarchitecture*. 152–162.

Michael Ferdman, Thomas F. Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2008. Temporal instruction fetch streaming. In *International Symposium on Microarchitecture*. 1–10.

Adi Fuchs, Shie Mannor, Uri Weiser, and Yoav Etsion. 2014. Loop-aware memory prefetching using code block working sets. In *International Symposium on Microarchitecture*. 533–544.

Ilya Ganusov and Martin Burtscher. 2005. Future execution: A hardware prefetching technique for chip multiprocessors. In *International Conference on Parallel Architectures and Compilation Techniques*. 350–360.

Ilya Ganusov and Martin Burtscher. 2006. Efficient emulation of hardware prefetchers via event-driven helper threading. In *International Conference on Parallel Architectures and Compilation Techniques*. 144–153.

Yao Guo, Pritish Narayanan, Mahmoud Abdullah Bennaser, Saurabh Chheda, and Csaba Andras Moritz. 2011. Energy-efficient hardware data prefetching. *IEEE Trans. Very Large Scale Integr. Syst.* 19, 2 (2011), 250–263.

D. Guttman, M. T. Kandemir, Meenakshi Arunachalam, and Vlad Calina. 2015. Performance and energy evaluation of data prefetching on intel Xeon Phi. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 288–297. DOI:http://dx.doi.org/10.1109/ISPASS.2015.7095814

Zhigang Hu, Margaret Martonosi, and Stefanos Kaxiras. 2003. TCP: Tag correlating prefetchers. In *International Symposium on High-Performance Computer Architecture*. 317–326.

Yan Huang, Zhi-min Gu, Jie Tang, Min Cai, Jianxun Zhang, and Ninghan Zheng. 2012. Reducing cache pollution of threaded prefetching by controlling prefetch distance. In *International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*. 1812–1819.

Christopher J. Hughes and Sarita V. Adve. 2005. Memory-side prefetching for linked data structures for processor-in-memory systems. *J. Parallel and Distrib. Comput.* 65, 4 (2005), 448–463.

Ibrahim Hur and Calvin Lin. 2006. Memory prefetching using adaptive stream detection. In *International Symposium on Microarchitecture*. 397–408.

Ibrahim Hur and Calvin Lin. 2009. Feedback mechanisms for improving probabilistic memory prefetching. In *International Symposium on High Performance Computer Architecture (HPCA)*. 443–454.

Sorin Iacobovici, Lawrence Spracklen, Sudarshan Kadambi, Yuan Chou, and Santosh G. Abraham. 2004. Effective stream-based and execution-based data prefetching. In *International Conference on Supercomputing*. 1–11.

Akanksha Jain and Calvin Lin. 2013. Linearizing irregular memory accesses for improved correlated prefetching. In *International Symposium on Microarchitecture*. 247–259.

Victor Jiménez, Roberto Gioiosa, Francisco J. Cazorla, Alper Buyuktosunoglu, Pradip Bose, and Francis P. O'Connell. 2012. Making data prefetch smarter: Adaptive prefetching on POWER7. In *International Conference on Parallel Architectures and Compilation Techniques*. 137–146.

Doug Joseph and Dirk Grunwald. 1997. Prefetching using Markov predictors. In *International Symposium on Computer Architecture*. 252–263.

Norman P. Jouppi. 1990. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *International Symposium on Computer Architecture*. 364–373.

David Kadjo, Jinchun Kim, Prabal Sharma, Reena Panda, Paul Gratz, and Daniel Jimenez. 2014. B-fetch: Branch prediction directed prefetching for chip-multiprocessors. In *International Symposium on Microarchitecture (MICRO)*. 623–634.

Mahmut Kandemir, Yuanrui Zhang, and Ozcan Ozturk. 2009. Adaptive prefetching for shared cache based chip multiprocessors. In *Conference on Design, Automation and Test in Europe*. 773–778.

Hui Kang and Jennifer L. Wong. 2013. To hardware prefetch or not to prefetch? A virtualized environment study and core binding approach. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 357–368.

Cansu Kaynak, Boris Grot, and Babak Falsafi. 2013. SHIFT: Shared history instruction fetch for lean-core server processors. In *International Symposium on Microarchitecture*. 272–283.

Muneeb Khan, Andreas Sandberg, and Erik Hagersten. 2014. A case for resource efficient prefetching in multicores. In *International Conference on Parallel Processing (ICPP)*. 101–110.

Taesu Kim, Dali Zhao, and Alexander V. Veidenbaum. 2014. Multiple stream tracker: A new hardware stride prefetcher. In *Computing Frontiers*. 34.

Aasheesh Kolli, Ali Saidi, and Thomas F. Wenisch. 2013. RDIP: Return-address-stack directed instruction prefetching. In *International Symposium on Microarchitecture*. 260–271.

Sanjeev Kumar and Christopher Wilkerson. 1998. Exploiting spatial locality in data caches using spatial footprints. In *International Symposium on Computer Architecture*. 357–368.

An-Chow Lai, Cem Fide, and Babak Falsafi. 2001. Dead-block prediction & dead-block correlating prefetchers. In *International Symposium on Computer Architecture*. 144–154.

Chang Joo Lee, Onur Mutlu, Veynu Narasiman, and Yale N. Patt. 2008. Prefetch-aware DRAM controllers. In *International Symposium on Microarchitecture*. 200–209.

Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. 2012. When prefetching works, when it doesn't, and why. *ACM Trans. Arch. Code Opt.* 9, 1 (2012), 2:1–2:29.

Chungsoo Lim and Gregory T. Byrd. 2008. Exploiting producer patterns and L2 cache for timely dependence-based prefetching. In *International Conference on Computer Design*. IEEE, 685–692.

Wei-Fen Lin, Steven K. Reinhardt, and Doug Burger. 2001a. Reducing DRAM latencies with an integrated memory hierarchy design. In *International Symposium on High-Performance Computer Architecture*. 301–312.

Wei-Fen Lin, Steven K. Reinhardt, Doug Burger, and Thomas R. Puzak. 2001b. Filtering superfluous prefetches using density vectors. In *International Conference on Computer Design*. 124–132.

Fang Liu and Yan Solihin. 2011. Studying the impact of hardware prefetching and bandwidth partitioning in chip-multiprocessors. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. 37–48.

Gang Liu, Jih-Kwon Peir, and Victor Lee. 2012. Miss-correlation folding: Encoding per-block miss correlations in compressed DRAM for data prefetching. In *International Parallel & Distributed Processing Symposium (IPDPS)*. 691–702.

Jiwei Lu, Howard Chen, Rao Fu, Wei-Chung Hsu, Bobbie Othmer, Pen-Chung Yew, and Dong-Yuan Chen. 2003. The performance of runtime data cache prefetching in a dynamic optimization system. In *International Symposium on Microarchitecture*. 180–190.

Jiwei Lu, Abhinav Das, Wei-Chung Hsu, Khoa Nguyen, and Santosh G. Abraham. 2005. Dynamic helper threaded prefetching on the sun UltraSPARC® CMP processor. In *International Symposium on Microarchitecture*.

Chi-Keung Luk. 2001. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *International Symposium on Computer Architecture*. 40–51.

R. Manikantan and R. Govindarajan. 2008. Focused prefetching: Performance oriented prefetching based on commit stalls. In *International Conference on Supercomputing*. 339–348.

R. Manikantan, R. Govindarajan, and Kaushik Rajan. 2011. Extended histories: Improving regularity and performance in correlation prefetchers. In *International Conference on High Performance and Embedded Architectures and Compilers*. 67–76.

Jaydeep Marathe and Frank Mueller. 2008. PFetch: Software prefetching exploiting temporal predictability of memory access streams. In *Workshop on MEmory Performance: DEaling with Applications, Systems and Architecture (MEDEA)*. 1–8.

Sanyam Mehta, Zhenman Fang, Antonia Zhai, and Pen-Chung Yew. 2014. Multi-stage coordinated prefetching for present-day processors. In *International Conference on Supercomputing*. 73–82.

Sparsh Mittal. 2014. A survey of architectural techniques for improving cache power efficiency. *Elsev. Sust. Comput.: Inform. Syst.* 4, 1 (2014), 33–43.

Sparsh Mittal and Jeffrey Vetter. 2015. A survey of techniques for modeling and improving reliability of computing systems. *IEEE Trans. Parallel Distrib. Syst.* (2015).

Sparsh Mittal, Jeffrey S. Vetter, and Dong Li. 2015. A survey of architectural approaches for managing embedded DRAM and non-volatile on-chip caches. *IEEE Trans. Parallel Distrib. Syst.* (2015).

Todd C. Mowry, Monica S. Lam, and Anoop Gupta. 1992. Design and evaluation of a compiler algorithm for prefetching. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. 62–73.

Kyle J. Nesbit, Ashutosh S. Dhodapkar, and James E. Smith. 2004. AC/DC: An adaptive data cache prefetcher. In *International Conference on Parallel Architectures and Compilation Techniques*. 135–145.

Kyle J. Nesbit and James E. Smith. 2004. Data cache prefetching using a global history buffer. *International Symposium on High Performance Computer Architecture* (2004).

Biswabandan Panda and Shankar Balachandran. 2014. XStream: Cross-core spatial streaming based MLC prefetchers for parallel applications in CMPs. In *International Conference on Parallel Architectures and Compilation*. 87–98.

Jih-Kwon Peir, Shih-Chang Lai, Shih-Lien Lu, Jared Stark, and Konrad Lai. 2002. Bloom filtering cache misses for accurate data speculation and prefetching. In *International Conference on Supercomputing*. 189–198.

Seth H. Pugsley, Zeshan Chishti, Chris Wilkerson, Peng-fei Chuang, Robert L. Scott, Aamer Jaleel, Shih-Lien Lu, Kingsum Chow, and Rajeev Balasubramonian. 2014. Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers. In *International Symposium on High Performance Computer Architecture (HPCA)*. 626–637.

Rodric M. Rabbah, Hariharan Sandanagobalane, Mongkol Ekpanyapong, and Weng-Fai Wong. 2004. Compiler orchestrated prefetching via speculation and predication. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 189–198.

K. Raghavendra, B. Panda, and M. Mutyam. 2015. PBC: Prefetched blocks compaction. *IEEE Trans. Comput.* (2015). DOI:http://dx.doi.org/10.1109/TC.2015.2493533

Glenn Reinman, Brad Calder, and Todd Austin. 1999. Fetch directed instruction prefetching. In *International Symposium on Microarchitecture*. 16–27.

Amir Roth, Andreas Moshovos, and Gurindar S. Sohi. 1998. Dependence based prefetching for linked data structures. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. 115–126.

Suleyman Sair, Timothy Sherwood, and Brad Calder. 2002. Quantifying load stream behavior. In *International Symposium on High-Performance Computer Architecture*. 197–208.

Saurabh Sharma, Jesse G. Beu, and Thomas M. Conte. 2005. Spectral prefetcher: An effective mechanism for L2 cache prefetching. *ACM Trans. Arch. Code Opt.* 2, 4 (2005), 423–450.

Timothy Sherwood, Suleyman Sair, and Brad Calder. 2000. Predictor-directed stream buffers. In *International Symposium on Microarchitecture*. 42–53.

D. Solihin, Jaejin Lee, and Josep Torrellas. 2003. Correlation prefetching with a user-level memory thread. *IEEE Trans. Parallel Distrib. Syst.* 14, 6 (2003), 563–580.

Stephen Somogyi, Thomas F. Wenisch, Anastasia Ailamaki, and Babak Falsafi. 2009. Spatio-temporal memory streaming. In *International Symposium on Computer Architecture (ISCA)*. 69–80.

Stephen Somogyi, Thomas F. Wenisch, Anastassia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2006. Spatial memory streaming. In *International Symposium on Computer Architecture (ISCA)*. 252–263.

Lawrence Spracklen, Yuan Chou, and Santosh G. Abraham. 2005. Effective instruction prefetching in chip multiprocessors for modern commercial applications. In *International Symposium on High-Performance Computer Architecture (HPCA)*. 225–236.

Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N. Patt. 2007. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *International Symposium on High Performance Computer Architecture*. 63–74.

Viji Srinivasan, Edward S. Davidson, and Gary S. Tyson. 2004. A prefetch taxonomy. *IEEE Trans. Comput.* 53, 2 (2004), 126–140.

Viji Srinivasan, Edward S. Davidson, Gary S. Tyson, Mark J. Charney, and Thomas R. Puzak. 2001. Branch history guided instruction prefetching. In *International Symposium on High-Performance Computer Architecture (HPCA)*. 291–300.

Steven P. Vanderwiel and David J. Lilja. 2000. Data prefetch mechanisms. *Comput. Surv.* 32, 2 (2000), 174–199.

Santhosh Verma and David M. Koppelman. 2012. The interaction and relative effectiveness of hardware and software data prefetch. *J. Circ., Syst. Comput.* 21, 02 (2012).

Zhenlin Wang, Doug Burger, Kathryn S. McKinley, Steven K. Reinhardt, and Charles C. Weems. 2003. Guided region prefetching: A cooperative hardware/software approach. In *International Symposium on Computer Architecture (ISCA)*. 388–398.

Thomas F. Wenisch, Michael Ferdman, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2009. Practical off-chip meta-data for temporal memory streaming. In *International Symposium on High Performance Computer Architecture (HPCA)*. 79–90.

Thomas F. Wenisch, Stephen Somogyi, Nikolaos Hardavellas, Jangwoo Kim, Anastassia Ailamaki, and Babak Falsafi. 2005. Temporal streaming of shared memory. In *International Symposium on Computer Architecture (ISCA)*. 222–233.

Carole-Jean Wu, Aamer Jaleel, Margaret Martonosi, Simon C. Steely Jr, and Joel Emer. 2011. PACMan: Prefetch-aware cache management for high performance caching. In *International Symposium on Microarchitecture*. 442–453.

C.-J. Wu and Margaret Martonosi. 2011. Characterization and dynamic mitigation of intra-application cache interference. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2–11.

Jun Yan and Wei Zhang. 2008. Analyzing the worst-case execution time for instruction caches with prefetching. *ACM Trans. Embedd. Comput. Syst.* 8, 1 (2008), 7.

Chia-Lin Yang and Alvin R. Lebeck. 2000. Push vs. pull: Data movement for linked data structures. In *International Conference on Supercomputing*. 176–186.

Praveen Yedlapalli, Jagadish Kotra, Emre Kultursay, Mahmut Kandemir, Chita R. Das, and Anand Sivasubramaniam. 2013. Meeting midway: Improving CMP performance with memory-side prefetching. In *International Conference on Parallel Architectures and Compilation Techniques*. 289–298.

Jiyang Yu and Peng Liu. 2014. A thread-aware adaptive data prefetcher. In *International Conference on Computer Design (ICCD)*. 278–285.

Weifeng Zhang, Brad Calder, and Dean M. Tullsen. 2006. A self-repairing prefetcher in an event-driven dynamic optimization framework. In *International Symposium on Code Generation and Optimization*. 50–64.

Weifeng Zhang, Dean M. Tullsen, and Brad Calder. 2007. Accelerating and adapting precomputation threads for effcient prefetching. In *International Symposium on High Performance Computer Architecture (HPCA)*. 85–95.

Yi Zhang, Steve Haga, and Rajeev Barua. 2002. Execution history guided instruction prefetching. In *International Conference on Supercomputing*. 199–208.

Huaiyu Zhu, Yong Chen, and Xian-He Sun. 2010. Timing local streams: Improving timeliness in data prefetching. In *International Conference on Supercomputing*. 169–178.

Craig Zilles and Gurindar Sohi. 2001. Execution-based prediction using speculative slices. In *International Symposium on Computer Architecture*. 2–13.

Received June 2016; revised November 2015; accepted March 2016