Practical Temporal Prefetching With Compressed On-Chip Metadata

Hao Wu, Krishnendra Nathella, Matthew Pabst, Dam Sunwoo, Akanksha Jain, and Calvin Lin

Abstract—Temporal prefetchers are powerful because they can prefetch irregular sequences of memory accesses, but temporal prefetchers are commercially infeasible because they store large amounts of metadata in DRAM. This paper presents Triage, the first temporal data prefetcher that does not require off-chip metadata. Triage builds on two insights: (1) Metadata are not equally useful, so the less useful metadata need not be saved, and (2) for irregular workloads, it is more profitable to use portions of the LLC to store metadata than data. We also introduce novel schemes to identify useful metadata, to compress metadata, and to determine the fraction of the LLC to dedicate for metadata.

Using an industrial-strength simulator running irregular workloads on a single-core system, we show that at a prefetch degree of 4, Triage improves performance by 41.1% compared to a baseline with no prefetching, whereas BO, a state-of-the-art prefetcher that uses only on-chip metadata, sees only 10.9% improvement. Compared with MISB, a temporal prefetcher that uses off-chip metadata, Triage provides a design alternative that reduces memory traffic by an order of magnitude (260.8% extra traffic for MISB at degree 1 vs. 56.9% for Triage), while reducing coverage by 20%.

Index Terms—Memory Systems, Prefetching, Temporal Prefetching

1 Introduction

Data prefetchers are important mechanisms for hiding the long latency of DRAM accesses. Most commercial prefetchers perform some form of strided or spatial prefetching, because such prefetchers provide significant benefits while incurring low implementation costs. By contrast, *temporal* prefetchers are alluring because they can learn arbitrary sequences of repeated memory references, so they are effective for workloads with irregular memory accesses, including those that arise from pointer-based data structures. But because these arbitrary sequences have to be memorized, temporal prefetchers require megabytes of metadata that must be stored in DRAM. Of course, going to DRAM to retrieve metadata incurs high latency, consumes DRAM bandwidth, and increases energy consumption, so the primary challenge has been to effectively manage this off-chip metadata.

Early solutions amortized the cost of off-chip metadata accesses across multiple prefetches [1], [2]. In 2013, the Irregular Stream Buffer (ISB) represented the metadata as an address mapping, which allowed portions of the metadata to be cached on chip, with the contents of this metadata cache synchronized with the contents of the TLB. More recently, the Managed ISB (MISB) extended the ISB by introducing a more efficient fine-grained metadata management scheme that includes a metadata prefetcher.

Unfortunately, all of these solutions still have important limitations: (1) Even MISB has metadata traffic of 260.8%, so its performance suffers in bandwidth-constrained environments, and it incurs high energy costs in any environment; (2) they add hardware complexity because they require changes to the memory interface and communication with the operating system.

In this paper, we present Triage,[1] the first temporal prefetcher that requires no off-chip metadata. Our solution builds on three observations.

1) For most workloads, the vast majority of prefetches use just a small fraction of the metadata, so only the most frequently used metadata needs to be stored.
2) For many workloads, the last-level cache (LLC) typically has lower utility than an effective prefetcher, so for irregular workloads, portions of the LLC can be more profitably used to store temporal prefetcher metadata instead of storing data. For example, for the irregular subset of SPEC2006, reducing the LLC by 1 MB reduces performance by 7.4%, but a state-of-the-art irregular prefetcher with unlimited resources can improve performance by 34.0%. Therefore, if we can distinguish the important metadata from the unimportant metadata, we can profitably use portions of the LLC to store important prefetcher metadata.
3) For regular accesses, temporal prefetchers store metadata highly inefficiently, so further compression is possible through the use of a novel metadata representation. For example, a temporal prefetcher would represent a regular sequence of n addresses as n pairs of correlated addresses, which could be represented more compactly as a 3-tuple, (address, stride, length).

Thus, our Triage prefetcher re-purposes a portion of the LLC as a *metadata store*, and any metadata that cannot be kept in the metadata store is simply discarded. To identify important metadata, Triage uses the Hawkeye replacement policy [4], which provides significant performance benefits for small metadata stores, as it identifies frequently accessed metadata over a long history of time. Of course, the ideal

1. This paper is an extension of our MICRO 2019 paper [3] titled "Temporal prefetching without the off-chip metadata"

size of this metadata store varies by workload, so we also introduce a dynamic cache partitioning scheme that determines the amount of the LLC cache that should be provisioned for metadata entries. Finally, we introduce a new metadata representation that compactly represents strided access patterns to significantly compress the metadata for regular workloads.

By forsaking off-chip metadata and by intelligently managing on-chip metadata, Triage offers different tradeoffs than state-of-the-art temporal prefetchers. For example, Triage reduces off-chip traffic overhead to 59.3% (compared with 260.8% for MISB), it reduces energy consumption for metadata accesses by 4-22×, and it offers a much simpler hardware design. In Section 4, we show that in bandwidth-rich environments these benefits come at the cost of lower performance (due to limited metadata), but in bandwidth-constrained environments, they translate to significantly better performance.

This paper makes the following contributions:[2]

- We introduce Triage, the first PC-localized[3] temporal data prefetcher that does not use off-chip metadata. Triage reuses a portion of the LLC for storing prefetcher metadata, and it includes an adaptive policy for dynamically provisioning the size of the metadata store at a fine granularity.
- We explore metadata organizations that are best suited for storing metadata in the LLC, and we find that in this new setting, tables [5] are the most compact data structure for tracking correlated addresses. The use of tables represents a return to simplicity for temporal prefetchers as recent solutions propose increasingly complex metadata representations to manage off-chip metadata. We also introduce a compressed representation for regular accesses to reduce the metadata tables footprint.
- We evaluate Triage using a highly accurate proprietary simulator for single-core simulations and the ChampSim simulator for multi-core simulations.
  - On single-core systems running SPEC 2006 workloads, Triage significantly outperforms state-of-the-art prefetchers that use only on-chip metadata (27.3% speedup for Triage vs. 4.3% for the Best Offset Prefetcher, 2.2% for SMS).
  - Triage's integrated stream representation results in a 2.7% performance improvement over Triage at a prefetch degree of 1 and 4.4% performance improvement at a prefetch degree of 16 (41.1% speedup with the integrated representation vs. 36.7% without).
  - On single-core systems, Triage achieves 80% of the performance of a state-of-the-art temporal prefetcher that uses off-chip metadata (27.3% for Triage vs. 34.0% for MISB [6]). On a 16-core system running multi-programmed irregular SPEC workloads, where bandwidth is more precious, Triage's speedup is 6.9%, compared to 4.9% for MISB.
  - Triage also works well as part of a hybrid prefetcher that combines a regular prefetcher with a temporal prefetchers (27.8% for Triage+BO vs. 4.3% for BO alone on single-core systems, and 10.9% for BO+Triage vs. 4.7% for BO alone on 4-core systems).
  - On a 4-core system running CloudSuite server benchmarks, BO+Triage improves performance by 9.7%, compared to 3.8% for BO alone.
- We outline and analyze the design space for temporal prefetchers along three dimensions, namely, on-chip storage, off-chip traffic, and overall performance, and we show that Triage provides an attractive design point with previously unexplored tradeoffs.

This paper is organized as follows. Section 2 places our work in the context of prior work. Section 3 then describes our solution, and Section 4 then presents our empirical evaluation. Finally, we conclude in Section 5.

2 Related Work

We now discuss related work in data prefetching. We start by contrasting our work with other temporal prefetchers before briefly discussing other classes of prefetchers.

2.1 Temporal Prefetching

Temporal prefetchers are quite general because they learn address correlations, that is, correlations between consecutive memory accesses. Chilimbi et al., report that temporal streams, which are sequences of correlated address pairs, are found extensively in both scientific and commercial workloads [7], [8]. Unfortunately, considerable state is required to memorize correlations among addresses.

One class of irregular prefetchers reduce this metadata requirement by forgoing address correlation to learn weaker forms of correlation, such as delta correlation [9], tag correlation [10], or context-address pair correlation [11]. However, these simplifications limit the scope of memory access patterns that can be learned.

A second class of prefetchers exploit address correlation by storing metadata in off-chip memory [1], [6], [12]. These prefetchers use novel ways to reduce the overhead of off-chip memory accesses, but the use of off-chip metadata has limited the commercial viability of such prefetchers.

Triage is the first data prefetcher that reaps the benefit of PC-localized address correlation—the most powerful form of temporal prefetching [12]—without using any off-chip metadata. While elements of Triage borrow from temporal prefetchers that use off-chip metadata, Triage is designed with a completely different design goal, which is to prioritize the efficiency of the on-chip metadata store. Therefore, Triage rethinks many design decisions for the on-chip setting, removing complexity where possible and adding new design components where necessary.

We now provide more details about how Triage's design differs from temporal prefetchers that use off-chip metadata. We classify existing temporal prefetchers into three categories based on their off-chip metadata organization.

2. This paper improves upon the original Triage design [3] in two ways. First, it employs a new dynamic partitioning scheme that is simpler and that allows the LLC to be partitioned between data and at metadata at a finer granularity. Second, it reduces the metadata table's footprint by employing a novel compressed representation for regular accesses.

3. PC localization is a method of creating more predictable reference streams by separating streams according to the address of the instruction that issued the load.

2.1.1 Table-Based Temporal Prefetchers

Joseph and Grunwald introduced the idea of prefetching correlated addresses in 1997 with their Markov Prefetcher [5]. The Markov Prefetcher uses a table to record multiple possible successors for each address, along with a probability for each successor, but unfortunately, it was too large to be stored on-chip despite optimizations that reduced the size of the table [10].

Therefore, early temporal prefetchers explore designs that reduce the traffic and latency costs of accessing an off-chip Markov table [13], [14]. For example, Solihin et al., redundantly store a chain of successors in each off-chip table entry, which increases table size but amortizes the cost of fetching metadata for temporal streams by grouping them in a single off-chip access. Triage also uses a table-based organization, but there are two main differences. First, Triage uses PC-localization, which improves coverage and accuracy and eliminates the need to track multiple successors in each table entry, reducing table sizes by $2\times$ to $4\times$. Second, Triage uses a customized replacement policy that identifies the most useful table entries, removing the need for off-chip metadata.

2.1.2 GHB-Based Temporal Prefetchers

Wenisch et al. find that tables are not ideal for organizing off-chip metadata because temporal streams can have highly variable lengths [1], [7]. Their STMS prefetcher [1], [2] instead uses a *global history buffer* to record a history of past memory accesses in an off-chip circular buffer. The GHB reduces the latency of off-chip metadata accesses by amortizing the cost of off-chip metadata lookup over long temporal streams, and it reduces metadata traffic by probabilistically updating the off-chip structures. Somogyi et al., build on STMS to combine spatial and temporal streams with their STeMs prefetcher [15].

While the GHB improves significantly over table-based solutions, it suffers from three drawbacks that are addressed by Triage: (1) The GHB makes it infeasible to combine address correlation with *PC-localization*, which is a technique to improve predictability by correlating addresses that belong to the same PC, (2) its metadata cannot be cached because it is organized as a FIFO buffer, and (3) it incurs metadata traffic overhead of 200-400%. Furthermore, Triage's integrated stream representation differs from STeMs as it is more flexible—spatial streams in Triage's integrated representation can be of any length, whereas STeMs aligns them at page boundaries—and it is simpler because it does not need to maintain timing metadata to orchestrate prefetches from off-chip metadata.

2.1.3 Irregular Stream Buffer

The Irregular Stream Buffer (ISB) combines address correlation with PC-localization by proposing a new off-chip metadata organization [6], [12]. In particular, ISB maps PC-localized correlated address pairs to consecutive addresses in a new address space, called the *structural address space*. Furthermore, ISB caches a portion of the physical-to-structural address mappings on chip by synchronizing the contents of the on-chip metadata cache with the TLB and by hiding the latency of off-chip metadata accesses during TLB misses. While ISB significantly improves coverage and accuracy of temporal prefetchers, it still incurs metadata traffic overheads of 200-400%, and its metadata cache utilization is quite poor due to the absence of spatial locality in the metadata cache.

MISB [6] addresses these issues by divorcing ISB's metadata cache from the TLB. In particular, MISB manages the metadata cache at a fine granularity, and it hides the latency of off-chip metadata accesses by employing highly accurate metadata prefetching. As a result, MISB reduces the traffic overhead of temporal prefetchers to 260.8%. Like MISB, Triage uses fine-grained metadata caching, but there are several differences between MISB and Triage: (1) Triage uses a space-efficient table-based organization that is more suited for on-chip metadata (MISB's metadata footprint is $2\times$ larger than Triage's metadata footprint because it tracks each correlation in two entries, a physical to structural address mapping, and a structural to physical address mapping), (2) Triage uses a smart metadata management policy for its on-chip metadata, and (3) Triage has no metadata traffic overhead.

Finally, some prefetchers do store their metadata in on-chip caches [16], [17], [18], but the metadata storage requirements for these prefetchers is relatively small (hundreds of KB), so there was no question that they would be stored somewhere on chip. By contrast, Triage shows how prefetchers whose metadata are too large to fit on chip can avoid storing off-chip metadata.

2.2 Non-Temporal Prefetching

Many prefetchers predict sequential [19], [20], [21] and strided [22], [23], [24], [25], [26], [27], [28] accesses, and while this class of prefetchers has enjoyed commercial success due to their extremely compact metadata, their benefits are limited to regular memory accesses.

Some irregular memory accesses can be prefetched by exploiting spatial locality [17], [29], [30], [31], [32], such that recurring spatial patterns can be prefetched across different regions in memory. For example, the SMS prefetcher [17] uses on-chip tables to correlate spatial footprints with the program counter that first accessed a memory region. These spatial locality-based prefetchers tend to be highly aggressive, issuing prefetches for many lines in a region at once. More importantly, they are limited to a very special class of irregular accesses that does not include access to pointer-based data structures, such as trees and graphs.

Other prefetchers directly target pointers by either using compiler hints or hardware structures to detect pointers [33], [34], [35], [36]. For example, Content Directed Prefetching [35] searches the content of cache lines for pointer addresses and eagerly issues prefetches for all pointers. Such prefetchers waste bandwidth as they prefetch many pointers that will not be used.

3 Our Solution

Triage repurposes on-chip cache space to store prefetcher metadata, and each metadata entry records PC-localized correlated address pairs. To effectively utilize valuable on-chip cache space, Triage considers the following design questions:

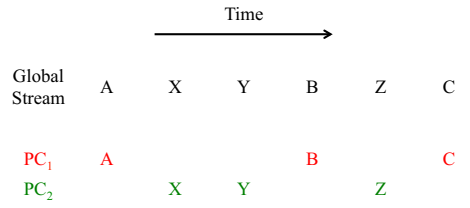
- How should metadata be represented to maximize space efficiency?
- Which metadata entries are likely to be the most useful?
- How much of the last-level cache should be dedicated to the metadata store?

We now explain how we address these questions.

3.1 Metadata Representation

Triage's metadata representation has two key features. First, unlike state-of-the-art temporal prefetchers, it uses a table to record address pairs. Second, for spatio-temporal streams—streams which have a mix of spatial and temporal accesses—we employ an *Integrated Stream Representation* to compactly represent the spatial components.

Figure 1 explains Triage's table-based organization for a purely temporal stream. In particular, the top side of Figure 1 shows a stream of memory references that is segregated into two *PC-localized* streams, and the bottom side shows the conceptual organization of metadata where each entry maps an address to its PC-localized neighbor.



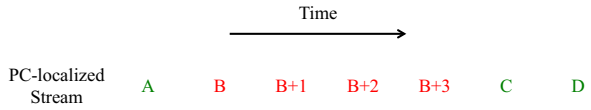
Addr	Neighbor
A	B
B	C
X	Y
Y	Z

Fig. 1. Triage's metadata organization.

While tables are a poor choice for organizing off-chip metadata (see Section 2), their space efficiency makes them an ideal choice for organizing on-chip metadata. In particular, compared to other metadata organizations [1], [6], [12], our table-based organization avoids metadata redundancy by representing each correlated address pair only once.

For spatio-temporal streams, address pairs are an inefficient way to represent the spatial components. For example, the text at the top of Figure 2 shows a spatio-temporal stream for which a naïve temporal-only representation results in 6 table entries (one entry for each address pair). This representation is wasteful for the spatial sub-stream $B, B+1, B+2, B+3$, which can be simply represented by tracking a stride of 1 and a stream length of 3.

To represent such spatio-temporal streams compactly, we modify Triage to additionally track a stride and stream length in each entry. The table at the bottom of Figure 2 shows this new integrated representation: Each table entry tracks a PC-localized neighbor, a stride, and a stream length. The spatial sub-stream $B, B+1, B+2, B+3$ is thus recorded in the first table entry with a stride of 1 and a stream length of 3; temporal sub-streams are marked with a stride of 0



Address	Neighbor	Stride	Stream Length
A	B	1	3
B+3	C	0	1
C	D	0	1

Fig. 2. Triage-ISR uses an integrated representation for spatio-temporal streams (spatial components are marked in red and temporal components are marked in green).

and stream length of 1. This results in a solution with just 3 table entries to represent the entire stream. We refer to this version of Triage as Triage-ISR, where ISR stands for *Integrated Stream Representation*.

Of course, Triage-ISR's compression benefits vary with the amount of spatial locality that the stream exhibits. For purely spatial streams, Triage-ISR can reduce metadata requirements by an order of magnitude, whereas for purely temporal streams, Triage-ISR offers no benefits. Section 3.4 provides more details about how this integrated representation is trained and used for prediction.

3.2 Metadata Replacement

Triage's metadata replacement policy manages the contents of its on-chip metadata store. We build Triage's metadata replacement policy on three observations. First, most metadata reuse can be attributed to a few metadata entries (see Figure 3). Second, even among the metadata entries that are frequently reused, fewer still account for prefetches that are not redundant, that is, prefetch requests that do not hit in the cache. Finally, metadata should be managed and evicted at a fine granularity because Triage targets irregular memory accesses, which exhibit poor spatial locality.

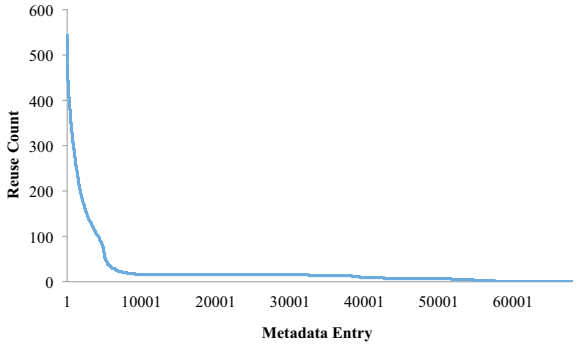


Fig. 3. Metadata reuse distribution for the *mcf* benchmark: For an execution with 60K metadata entries, only 15% of metadata entries are reused more than 15 times.

To accomplish these goals, we modify Hawkeye [4], a state-of-the-art cache replacement policy, which learns from the optimal solution for past memory references. To emulate the optimal policy for past memory references, Hawkeye examines a long history of past cache accesses (8× the size

of the cache), and it uses a highly efficient algorithm to reproduce the optimal solution. Figure 4 shows a high-level overview of Hawkeye, where OPTgen is used to train a PC-based predictor; the predictor learns whether loads by a given load instruction (PC) are likely to hit or miss with the optimal solution. On new cache accesses, the predictor informs the cache whether the line should be inserted with high priority or low priority.

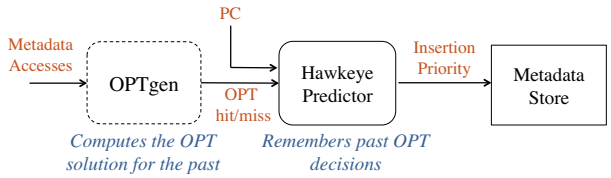


Fig. 4. Triage's metadata replacement is based on the Hawkeye [4] cache replacement policy.

Because Hawkeye can capture long-term reuse, it is a good fit for Triage, where the replacement policy must not be overwhelmed by the many useless metadata entries. We modify Hawkeye so that the policy is trained positively only when the metadata yields a prefetch that misses in the cache. We accomplish this by delaying Hawkeye's training when the prefetch request associated with a metadata entry is actually issued to memory. If the prefetch request hits in the cache, then the metadata reuse is ignored and is not seen by any component of the Hawkeye policy.

In Section 3.5, we explain how Triage is able to manage metadata at a finer granularity than the line size of the last-level cache.

3.3 Adjusting the Size of the Metadata Store

To avoid interference between application data and metadata, we partition the last-level cache by assigning separate ways to data and metadata. Since different applications require different metadata store sizes, our solution dynamically determines the number of ways that should be allocated to metadata. The partitioning is done at a way granularity, so if the cache has 16 ways, the metadata store can use 0, 1, 2, 3, 4, 5, 6, 7 or 8 ways.

To determine the number of ways to assign for metadata storage we employ a Bloom Filter [37]. The Bloom Filter records all the unique metadata entries encountered. In particular, when we add a new entry to the metadata store, we add its trigger address to the Bloom Filter. If the address is not already present in the Bloom Filter—a Bloom Filter Miss—we increment a global counter. We then calculate the number of ways to allocate for metadata storage using the following simple heuristic:

$$No.\ of\ metadata\ ways = ceil\left(\frac{Global\ Counter}{Number\ of\ sets\ in\ LLC}\right)$$

For example, if there are $32K$ sets in LLC and the global counter value is $100K$, we assign 4 ways for metadata storage.

Since Bloom Filters can yield false positives on a query, this scheme can underestimate the number of unique metadata entries. To accommodate false positives, we randomly increase the global counter on Bloom Filter hits with a probablity matching the false positive rate of the bloom filter. The false positive rate can be computed as

$$fp\_rate = (1 - e^{kn/m})^k$$

where k is the number of hash functions in the Bloom filter, n is the number of unique entries and m is the number of bits in the Bloom filter. On each hit in the Bloom filter, we generate a random value r. If $r < fp\_rate$, we also increment the global counter. This modification simulates the false positives in the Bloom filter.

The bloom filter is reset periodically. We find that the reset period does not have a major impact on the effectiveness of this scheme. For our evulation we reset the bloom filter every 30M instructions.

3.4 Overall Operation

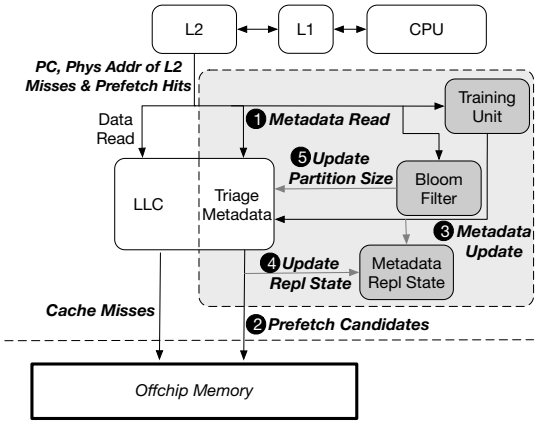


Fig. 5. Overview of Triage.

Figure 5 shows the overall design of Triage, where we see that a portion of the LLC is re-purposed for Triage's metadata store. On every LLC access, the metadata portion of the LLC is probed with the incoming address to check for a possible metadata cache hit ❶. If the metadata entry is found, it is read to generate a prefetch request ❷. Regardless of whether the load resulted in a metadata hit or miss, the Training Unit is updated (as explained below), and the newly trained metadata entry is added (or updated) in the metadata store ❸. The metadata replacement state is updated on metadata misses and metadata hits that generate a successful prefetch ❹, and a bloom filter periodically recomputes the amount of LLC that should be used as a metadata cache ❺. We now explain these operations in more detail.

3.4.1 Training

The Training Unit (TU) keeps the most recently accessed address for each PC. When a new access B arrives for a given PC, the Training Unit is queried for the last accessed address A by the same PC. Addresses A and B are then considered to be correlated. If subsequent addresses by the same PC exhibit spatial locality—for example, $B+1, B+2, B+3$—then the stride and stream lengths are updated appropriately in the training unit. The Training Unit entry is transferred to

Triage's metadata store when the spatial stream ends; the metadata store is indexed by the first address in the pair (A in this example).

More concretely, Table 1 shows the actions that are performed to train Triage, given an entry $A, B, stride, len$ in the Training Unit. For new entries, $stride$ is initialized to 0, and len is initialized to 1. If the new address is same as the last address—B in this case—then no action is taken. If the new address begins a spatial stream or continues a previously detected spatial stream, then the len parameter is incremented by 1. The beginning of a spatial stream is detected if $stride$ is currently set to 0 and if the stride between the new address and the last address is less than a constant (64 in our case). In all other cases, the Training Unit entry is written out to the metadata store, and it is re-initialized.

To avoid changing entries due to noisy data, each mapping in Triage's metadata store has an additional 1-bit confidence counter. If the Training Unit determines that A's neighbor differs from the value in the metadata store, then the confidence counter is decremented. If the Training Unit determines that A's neighbor matches the value in the metadata store, then the confidence counter is incremented. The neighbor is changed only when the confidence counter drops to 0. When the neighbor is changed, the corresponding stride and length fields are also updated corresponding to the new neighbor.

3.4.2 Prediction

Upon arrival of a new address A, Triage indexes the metadata by address A to find any available metadata entry. If an entry (say $(A, B, stride, len)$) is found, Triage puts all addresses that belong to the stream $(B, stride, len)$ into a *prefetch queue*. Future accesses from the same PC trigger the issue of a prefetch request from the head of this prefetch queue. If an entry is not found in the metadata store, nothing is added to the prefetch queue, but Triage still prefetches addresses from the prefetch queue until it is empty.

For higher degree prefetching—say a degree of d—Triage issues d prefetch requests from the prefetch queue, and it ensures that at least d candidates are appended onto the prefetch queue. For example, if Triage retrieves the entry $A, B, stride = 1, length = 2$ from the metadata store, it appends, B and $B + 1$ on the prefetch queue irrespective of the prefetch degree. However, this entry results in only 2 additions to the prefetch queue, for degrees greater than 2, Triage further retrieves the entry corresponding o $B+1$ from the metadata store to generate additional prefetch requests.

One drawback of our table-based organization is that higher degree prefetching requires multiple metadata lookups, but this penalty is significantly lower when the metadata resides completely on chip (˜20 cycles for accessing each LLC-resident metadata entry vs. 150-400 cycles for accessing each off-chip metadata entry.

3.4.3 Metadata Replacement Updates

Our metadata replacement is based on the Hawkeye policy [4], and like the Hawkeye policy, our metadata replacement policy is trained on the behavior of a few sampled sets. The metadata replacement predictors are trained on all metadata accesses, except those, that result in redundant prefetches. The replacement predictors are probed on all metadata accesses, including hits and misses, to update the per-metadata-entry replacement state. For more details on how the Hawkeye policy works, we refer the reader to the original paper [4].

3.4.4 Metadata Partition Updates

Triage partitions the cache between data and metadata by using way partitioning. The partitions are recomputed every 50,000 metadata accesses. If Triage decides to increase the amount of metadata store, dirty lines are flushed and the newly allocated/deallocated portion of the cache is marked invalid immediately. If Triage decides to decrease the amount of metadata store, lines with metadata entries are marked invalid.

For shared caches, Triage computes the metadata allocation for each core individually and allots the corresponding portion for each core's metadata. For example, if two cores are sharing a 4MB cache, and if cores 0 and 1 want 768KB and 256KB of metadata, respectively, then Triage allocates 1MB of the shared LLC for metadata, and it partitions the metadata store in a 3:1 ratio among the two cores.

3.5 Hardware Design

We now describe the detailed hardware changes required for Triage. We first discuss the indexing scheme and the tag lookup logic for the metadata store, and then describe the minimal changes to the baseline data cache replacement logic.

3.5.1 Metadata Store Indexing

Triage's metadata store is addressed by using the first address of each correlated pair. To index into Triage's metadata store, we could use the middle-order bits in the metadata address[4] to determine the setID because like data addresses, the middle bits of metadata addresses are likely to be uniformly distributed in the address space. However, Triage's metadata store does not see such a uniform distribution in the middle-order bits. Figure 6 explains why.

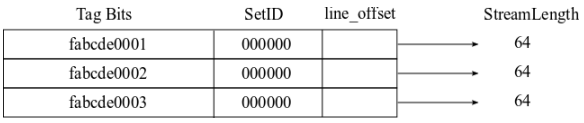


Fig. 6. Set conflicts in Triage-ISR using middle-order bits as setID.

Each row in Figure 6 represents a metadata entry's address, and different columns represent different portions of the entry's address. We see that since the stream length field is fixed to 6-bits (maximum stream length of 64), long spatial streams get broken down into small streams, and adjacent spatial streams, which differ by a multiple of the stream length, all have identical middle-order bits that map to set 0. Because the middle-order bits of the metadata addresses are not uniformly distributed, a naive set indexing scheme results in an unacceptable number of cache conflicts.

4. Middle bits are the bits that precede the bits representing the cache line offset.

Existing entry for PC	New Address	Action
$A, B, stride, len$	B (same as last address)	Do nothing
$A, B, 0, 1$	$B + stride$ (new spatial stream)	Update TU entry to $A, B, stride, len + 1$
$A, B, stride, len$	$B + (stride * k)$ (within a constant stride from last address)	Update TU entry to $A, B, stride, len + 1$
$A, B, stride, len$	C (no fixed stride)	Move TU entry to on-chip metadata Re-initialize TU to $B + stride * (len - 1)), C, 0, 1)$

TABLE 1
Triage's Training Algorithm

One solution to avoid conflict misses is to determine setID by using the higher-order bits of the metadata address. However, this solution is undesirable for workloads with predominantly irregular accesses because they tend to have more diversity in middle-order bits than the higher-order bits of their addresses, resulting in many conflict misses.

To solve this problem, we take inspiration from the Touche compressed cache design [38]. In particular, we determine the setID of the metadata store by applying an XOR between different portions of the address. Figure 7 shows how this indexing scheme works, and we find that this approach distributes metadata entries uniformly across sets for a diverse set of workloads.

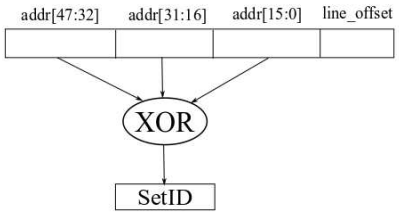


Fig. 7. Triage's set indexing scheme avoids conflict misses in the metadata store.

3.5.2 Metadata Store Tag Lookup

Each metadata entry in Triage is 42 bits long, but LLC line sizes are typically 64 to 128 bytes. Therefore, Triage's metadata entries within an LLC line must be organized at a fine granularity because metadata entries for irregular prefetchers do not exhibit spatial locality. We store multiple tagged metadata entries within each LLC cache line. For example, for a 64 byte LLC line, we store 12 metadata entries within a cache line. The metadata entries within a cache line are stored in the following format: tag-entry-tag-entry- $\cdots$ -tag-entry. On a metadata lookup, we first choose a physical LLC cache line from the metadata store, and we then find the relevant metadata entry by comparing the sub-tags within each cache line.

To store the metadata within 4 bytes, we use a compressed tag. To understand our compressed tag, realize that each physical address has a cache line offset of 6 bits and set_id of 16 bits, and the remaining bits are tags. We use a compression method similar Touche compression cache [38] to compress the tag to 7 bits. Thus, each metadata entry records the compressed tag of the trigger address and the compressed tag and set_id of the next address, which require a total of 30 bits.[5] We restrict the maximum stride and stream length to be both 64, and we need another 12 bits to store these information, making it total of 42 bits.

To identify the finer-grain metadata entries within a cache line, we require additional logic in the form of comparators and multiplexors. The extra logic is similar to that used in the Amoeba-Cache [39] and may incur additional latency or pipeline stages, but only for metadata accesses. We find that penalizing the LLC access latencies for both data and metadata by up to 6 cycles results in minimal performance impact (around 1% lower speedup on average for the irregular SPEC workloads).

3.5.3 Data Cache Replacement Logic

Since the cache is way-partitioned between data and metadata, we only need minor modifications to the data cache replacement logic to support Triage. Every time the partitions are recomputed by Triage, the outcome is stored in a status register. The status register holds the number of ways currently assigned for metadata. During replacement, the replacement logic simultaneously consults the metadata in the tag array and the status register to identify a candidate way for eviction. For example, for a 4-way LLC with SRRIP [40] replacement policy (2-3 bits per entry storing RRPV values), if Triage recently assigned 2 ways for metadata, on replacement, the status register is consulted to identify that only ways 0 and 1 store data and only the RRPV values for these ways are considered to identify the way with the largest RRPV for eviction.

4 Evaluation

4.1 Methodology

We evaluate Triage on single-core configurations using a cycle-level industrial simulator that models ARMv8 AArch64 CPUs and that has been correlated to be highly accurate against commercial CPU designs. The parameters of the CPU and memory system model used in the simulation are shown in Table 2. This model uses a simple memory model with fixed latency, but it models memory bandwidth constraints accurately.

For multi-core evaluation of Triage, we use ChampSim [41], a trace-based simulator that includes an out-of-order core model and a detailed memory system. ChampSim's cache subsystem includes FIFO read and prefetch queues, with demand requests having higher priority than prefetch requests. The main memory model simulates data bus contention, bank contention, and bus turnaround delays; bus contention increases memory latency. Our modeled processor for ChampSim also uses the configuration shown in Table 2. We confirm that the performance trends on the two simulators are the same.

5. The set_id of the trigger address is implicit in a set-associative cache, so it does not need to be stored.

Core	Out-of-order, 2GHz, 4-wide fetch, decode, and dispatch 128 ROB entries
TLB	48-entry fully-assoc L1 I/D-TLB 1024-entry 4-way assoc L2 TLB
L1I	64KB, 4-way assoc, 3-cycle latency
L1D	64KB, 4-way assoc, 3-cycle latency Stride prefetcher
L2	512KB, private, 8-way assoc 11-cycle load to use latency
L3	2MB/core, shared, 16-way assoc 20-cycle load-to-use latency Line size 64 bytes
DRAM	Single-Core: 85ns latency, 32GB/s bandwidth Multi-Core: 8B channel width, 800MHz, tCAS=20, tRP=20, tRCD=20 2 channels, 8 ranks, 8 banks, 32K rows 32GB/s bandwidth

TABLE 2
Machine Configuration

4.1.1 Benchmarks

We present single-core results for a subset of SPEC2006 benchmarks that are memory bound and are known to have irregular access patterns [12]. For SPEC benchmarks we use the reference input set. For all single-core benchmarks, we use SimPoints [42] to find representative regions. Each SimPoint is warmed up for 200 million instruction and run for 50 million instructions, and we generate at most 10 SimPoints for each SPEC benchmark.

We present multi-core results for CloudSuite [43] and multi-programmed SPEC benchmarks. For CloudSuite, we use the traces provided with the 2^{nd} Cache Replacement Championship. The traces were generated by running CloudSuite in a full-system simulator to intercept both application and OS instructions. Each CloudSuite benchmark includes 6 samples, where each sample has 100 million instructions. We warm up for 50 million instructions and measure performance for the next 50 million instructions.

For multi-programmed SPEC simulations, we simulate 4, 8, and 16 cores, such that each core runs a benchmark chosen uniformly randomly from all memory-bound benchmarks, including both regular and irregular programs. Overall, we simulate 80 4-core mixes, 80 8-core mixes, and 80 16-core mixes. Of the 80 mixes, 30 mixes include random mixes of irregular programs only, and the remaining 50 mixes include both regular and irregular programs. For each mix, we simulate the simultaneous execution of SimPoints of the constituent benchmarks until each benchmark has executed at least 30 million instructions. To ensure that slow-running applications always observe contention, we restart benchmarks that finish early so that all benchmarks in the mix run simultaneously throughout the execution. We warm the cache for 30 million instructions and measure the behavior of the next 30 million instructions.

4.1.2 Prefetchers

We evaluate Triage against two state-of-the-art on-chip prefetchers, namely, Spatial Memory Streaming (SMS) [17] and the Best Offset Prefetcher (BO) [28]. SMS captures irregular patterns by applying irregular spatial footprints across memory regions. BO is a regular prefetcher that won the Second Data Prefetching Championship.

We also evaluate Triage against existing off-chip temporal prefetchers, namely, Sampled Temporal Memory Streaming (STMS) [1], Domino [44], and MISB [6]. STMS, Domino, and MISB represent the state-of-the-art in temporal prefetching. For simplicity, we model idealized versions of STMS and Domino, such that their off-chip metadata transactions incur no latency or traffic penalty. Thus, our performance results for these prefetchers represent the upper bound performance of these prefetchers. For MISB, we faithfully model the latency and traffic of all metadata requests.

We evaluate two versions of Triage, denoted as Triage and Triage-ISR. The former reproduces the design from our MICRO paper [3] and does not include the integrated stream representation. It also uses a coarse-grained partitioning scheme that only permits metadata allocations of 0 ways, 4 ways, or 8 ways. Triage-ISR represents the design discussed in Section 3, including the integrated stream representation and the fine-grained partitioning using the Bloom Filter. For static versions of these designs, we pick a fixed metadata store size that gives the best average performance, and we then use this size to statically partition the LLC; we find that the best static metadata store size for a 2MB LLC is 1MB on both simulators.

All prefetchers train on the L2 access stream—including prefetch requests from L1—and prefetches are inserted into the L2. The metadata is stored in L3. Unless specified, all prefetchers use a prefetch degree of 1, which means that they issue at most one prefetch on every trigger access.

4.2 Comparison With Prefetchers That Store Metadata On Chip

Figure 8 shows that Triage outperforms state-of-the-art prefetchers that use only on-chip metadata. In particular, Triage-ISR—the best Triage configuration—achieves a speedup of 27.3%, whereas BO and SMS see a speedup of 4.3% and 2.2%, respectively.

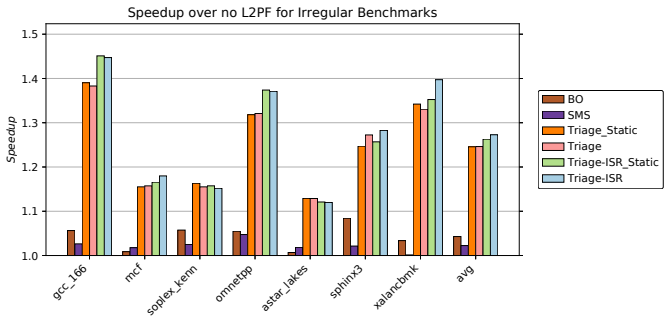


Fig. 8. Triage outperforms BO and SMS

The comparison among the different configurations of Triage leads to three observations. First, the integrated stream representation improves Triage's performance by 2.7% compared to the baseline version of Triage [3] that does not distinguish between regular and irregular accesses (27.3% speedup for Triage-ISR vs. 24.6% speedup for Triage). Second, Triage-ISR-Dynamic outperforms Triage-ISR-Static,

which shows that it is beneficial to modulate the metadata store size based on a precise estimation of metadata requirements. Finally, the integrated stream representation accentuates the benefit of the dynamic scheme because the benefits of compression vary across benchmarks and the dynamic scheme is able to adapt to these variations. As we will see later, the benefit of our dynamic scheme is even more pronounced in a shared cache setting where there is significantly more cache contention.

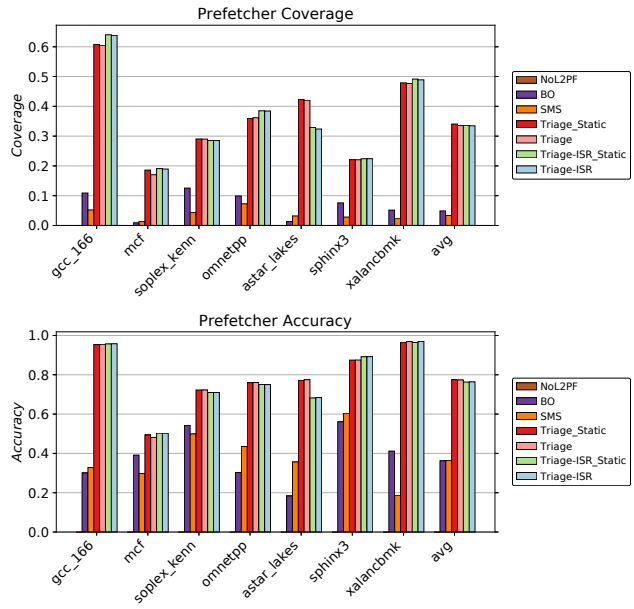


Fig. 9. Triage improves coverage and accuracy.

Triage's superior performance over BO and SMS can be explained by its higher coverage (33.6% for Triage vs. 4.9% for BO and 3.4% for SMS) and higher accuracy (77.4% for Triage vs. 36.2% for BO and 36.5% for SMS) as shown in Figure 9. Triage-ISR also improves coverage over Triage across all benchmarks except astar.[6] This improvement in coverage can be attributed to Triage-ISR's compressed metadata representation, which enables storing more metadata given a smaller metadata store. Figure 10 shows that the integrated stream representation reduces the metadata requirements by 3% on average and up to 12% for benchmarks with more regular accesses (sphinx3 and soplex).

4.2.1 Higher Degree Prefetching

Figure 11 shows the performance of our prefetchers at different prefetch degrees. As we increase degree from 1 to 4, Triage-ISR's performance grows from 27.3% to 41.1%. Increasing the degree beyond 4 does not improve Triage-ISR's performance. By comparison, BO and SMS achieve their best performance at degree 8 and 16, respectively, where they improve performance by 10.9% and 7.9% (over a baseline with no prefetcher), respectively. We also see that the gap between Triage and Triage-ISR increases as we increase degree, and with a degree of 4, Triage-ISR improves over Triage by 4.4%.

6. For astar, Triage-ISR has lower accuracy, which results in lower prefetch coverage.

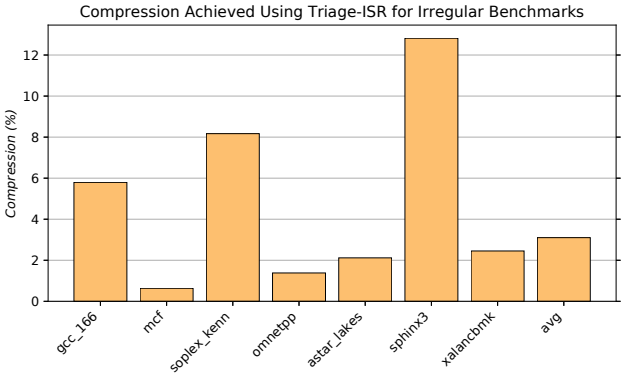


Fig. 10. Metadata compression achieved by Triage-ISR.

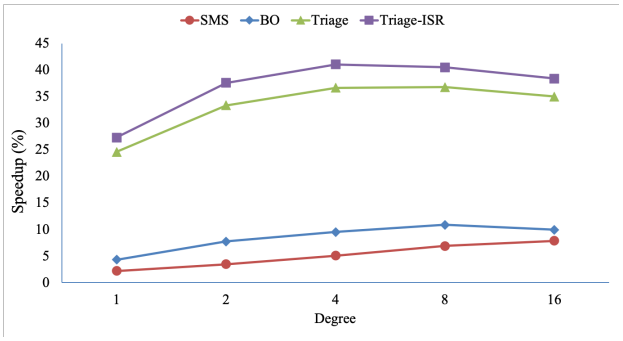


Fig. 11. Sensitivity to Prefetch degree.

4.2.2 Hybrid Prefetchers

Since Triage targets irregular memory accesses, it makes sense to evaluate it as a hybrid with regular memory prefetchers, such as BO. Figure 12 shows that a BO+Triage-ISR hybrid outperforms BO (27.8% speedup for BO+Triage-ISR vs. 4.3% for BO), which confirms that Triage successfully prefetches lines that BO cannot. We also see that Triage-ISR continues to outperform Triage even in this hybrid setting, which suggests that Triage-ISR's improvement over Triage can be attributed to its better coverage for irregular accesses.

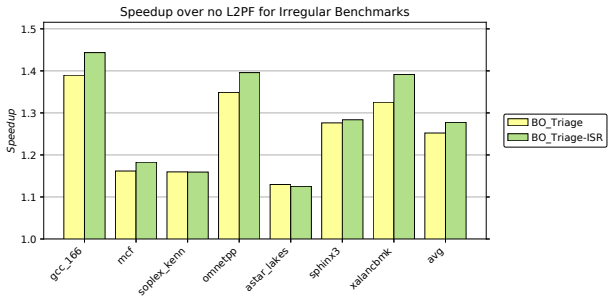


Fig. 12. Triage performs well as part of a hybrid prefetcher.

For completeness, Figure 13 compares all prefetchers on the remaining memory-intensive SPEC 2006 benchmarks.[7] Because these benchmarks are regular, BO+Triage-ISR does not significantly improve over BO, but Triage's dynamic

7. For astar, gcc, and soplex, we show results for the reference inputs which are more regular.

partitioning ensures that it causes no harm. As we would expect, Triage-ISR achieves high metadata compression for these benchmarks (9.6% on average and 86% for cactus), but because these benchmarks are highly regular and predictable, Triage does not benefit from either the extra metadata space or the extra cache space.

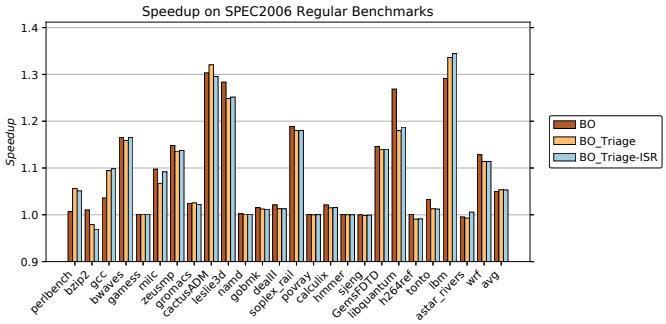


Fig. 13. Results on regular SPEC 2006 benchmarks.

4.2.3 *Understanding Triage's Benefits*

We now take a deeper look into the sources of Triage's performance benefits. Figure 14 shows that the performance benefit of irregular prefetching significantly outweighs the performance loss of reduced LLC capacity. In particular, we see that an optimistic version of Triage-ISR that is given a 1 MB on-chip metadata store in addition to its usual LLC capacity achieves a 34.0% speedup. On the other hand, a system with no Triage and a reduced LLC capacity of 1 MB lowers performance by only 7.4%. This loss in performance is easily compensated by Triage's benefits, as Triage-ISR sees an overall speedup of 26.3% with a fixed 1 MB metadata store and a 1 MB LLC.

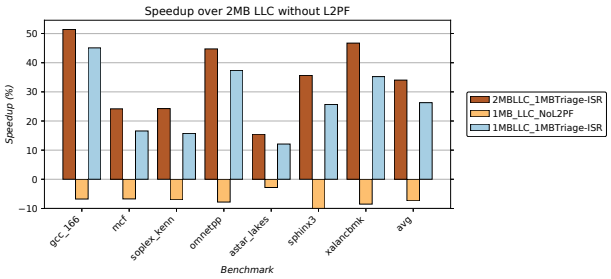


Fig. 14. Breakdown of Triage's Performance Improvements

Figure 15 compares the performance of Triage at different metadata store sizes and with different replacement policies (assuming no loss in LLC capacity). We make two observations. First, with just 1MB of metadata store, Triage achieves 75% of the performance of an idealized PC-localized temporal prefetcher, which is significant because typical temporal prefetchers consume tens of megabytes of off-chip storage. This result confirms the main insight of Triage that most prefetches can be attributed to a small percentage of metadata entries. Our second observation is that a smart replacement policy can improve the effectiveness of Triage at smaller metadata cache sizes, but when the metadata cache is sufficiently large (1 MB), the gap between LRU and Hawkeye shrinks. In particular, with a 256 KB metadata cache, Triage with an LRU policy achieves 11.3% speedup whereas Triage with the Hawkeye policy sees a 15.2% speedup.

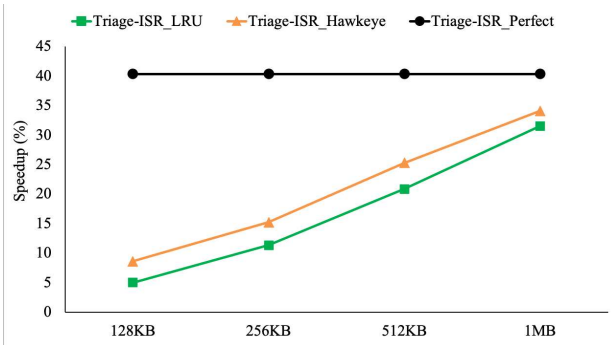


Fig. 15. Sensitivity to metadata store size (assuming no loss in LLC capacity).

4.3 Comparison With Prefetchers That Use Off-Chip Metadata

We now show that compared to temporal prefetchers that use tens of megabytes of off-chip metadata, Triage and Triage-ISR provide a more desirable tradeoff between performance, energy, and off-chip metadata traffic.

Figure 16 compares Triage and Triage-ISR against overly optimistic idealized versions of STMS and Domino and against a realistic version of MISB [6]. We see that Triage-ISR outperforms idealized STMS and Domino (27.3% for Triage-ISR vs 13.4% for Domino and 14.0% for STMS). Triage doesn't match MISB's 33.3% performance, but we see that it incurs much less traffic overhead (bottom graph in Figure 16). In particular, compared to a baseline with a 2 MB cache and no prefetching, Triage and Triage-ISR increase traffic by 53.9% and 56.9%, respectively, whereas STMS, Domino and MISB increase traffic by 441.8%, 441.1%, and 260.8%, respectively. While Triage and Triage-ISR incur some additional traffic due to inaccurate prefetches, we find that most of their traffic increase can be attributed to an effectively smaller LLC, which results in more demand misses. However, this increase in traffic is a good tradeoff as it facilitates high prefetcher coverage.

To put these results in context, Figure 17 compares all temporal prefetchers and the Best Offset (BO) prefetcher along two axes, namely, performance and traffic overhead. STMS, Domino, and MISB all use off-chip metadata, so they incur high off-chip traffic overheads and are in general more complex due to the complications of storing metadata off chip. Triage outperforms STMS and Domino while eliminating metadata overheads. Triage has lower performance than MISB, but it reduces traffic by more than half, offering an attractive design point for temporal prefetching. In fact, Triage's traffic overhead of 59.3% is not far from BO's 21.4% traffic overhead. BO's traffic overhead can be attributed to its large volume of inaccurate prefetches on irregular programs. By contrast, Triage is more accurate, but it incurs more traffic due to an effectively smaller LLC.

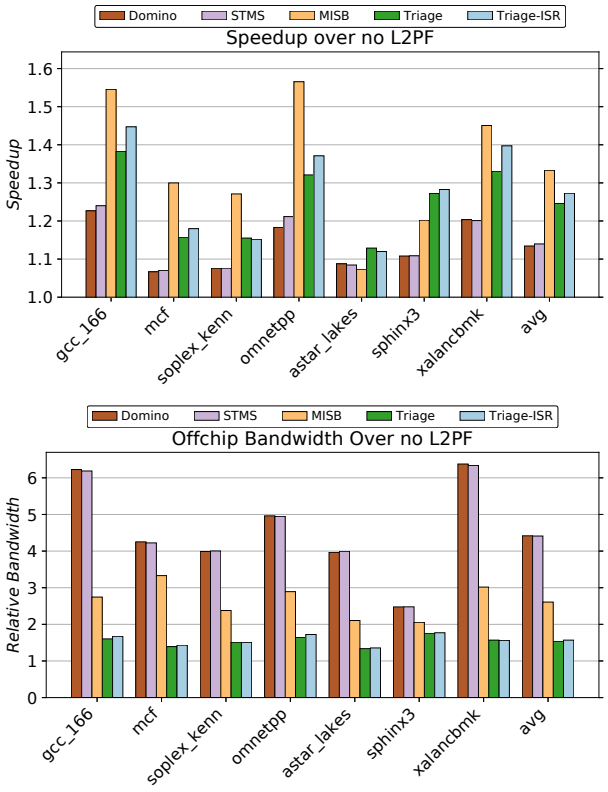


Fig. 16. Triage reduces traffic compared to off-chip temporal prefetchers while offering good performance improvements.

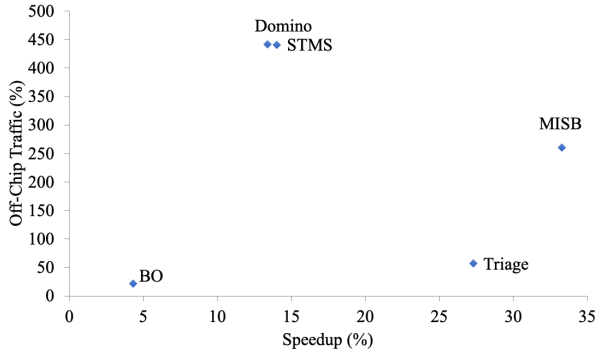


Fig. 17. Design Space of Temporal Prefetchers.

4.3.1 *Energy Evaluation*

Triage is more energy-efficient than other temporal prefetchers. Figure 18 shows that Triage's metadata accesses are 4–22× more energy efficient than MISB's. To estimate the energy consumption of Triage's metadata accesses, we count the number of LLC accesses for metadata, assuming 1 unit of energy for each LLC access. To estimate the energy consumption of MISB's memory accesses, we count the number of off-chip metadata accesses and multiply it by the average energy of a DRAM access. Since a DRAM access can consume anywhere from 10× to 50× more energy than an LLC access [45], [46], we assume that each DRAM access consumes 25 units of energy, and we add error bars to account for the lower bound (10 units of energy per DRAM access) and upper bound (50 units of energy DRAM access) of MISB's overall energy consumption.

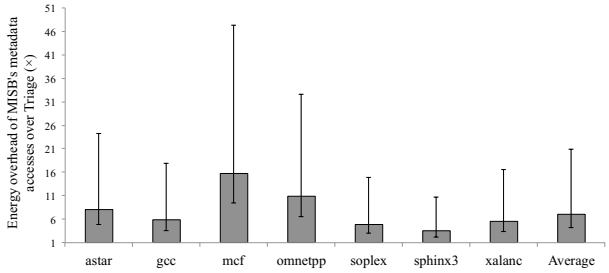


Fig. 18. Triage is more energy efficient than MISB.

At higher degrees, Triage's table-based design requires multiple LLC lookups, which will increase its overall energy requirements. In particular, we find that Triage's energy consumption doubles at degree 8, which is still much more energy efficient than MISB.

4.4 Evaluation on Multi-Programmed SPEC Mixes

We now evaluate Triage and Triage-ISR on multi-core systems and show that (1) the benefits of the dynamic scheme are greater for shared caches, and that (2) in bandwidth-constrained 8-core systems, Triage outperforms MISB due its lower traffic overhead.

Figure 19 shows that for multi-programmed mixes of irregular SPEC programs sharing an 8 MB last-level cache on a 4-core system, Triage-ISR-Dynamic is a significant improvement over Triage-ISR-Static. In fact, in this multi-core setting, a static version of Triage-ISR that allocates 4 MB for metadata performs worse than the Best Offset Prefetcher (3.2% for Triage-ISR-Static vs. 4.8% for BO). By contrast, Triage-ISR-Dynamic improves performance by 7.3%. At a prefetch degree of 16, Triage-ISR-Dynamic improves performance by 11.1%.

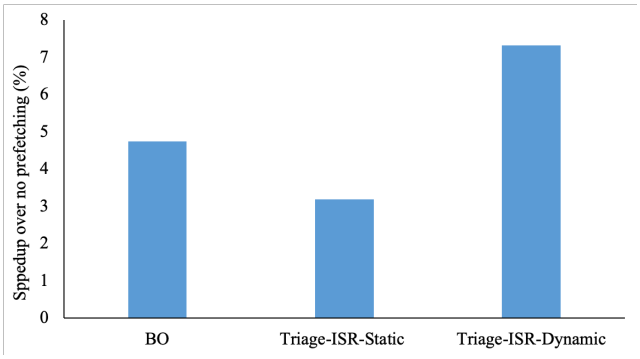


Fig. 19. Triage works well on multi-programmed mixes of irregular programs running on a 4-core system.

These results can be explained by noting that the LLC is a more valuable resource in shared systems. Triage-ISR-Dynamic works well in this setting because (1) it can modulate the portion of the LLC dedicated to metadata depending on the expected benefit of irregular prefetching, and (2) it can distribute the available metadata store among individual applications such that the application that benefits the most from irregular prefetching gets a larger portion of the metadata store.

Figure 20 compares the average speedup of Triage-ISR with MISB on 2-core, 4-core, 8-core, and 16-core systems where the cache is shared among different irregular programs. We see that while MISB outperforms Triage-ISR on a 2-core system (7.1% for Triage vs. 8.9% for MISB), Triage-ISR performs better on 8-core and 16-core systems. On a 16-core system, Triage-ISR outperforms MISB significantly (6.9% for Triage vs. 4.9% for MISB). These trends suggest that MISB's performance does not scale well to bandwidth-constrained environments because of its large metadata traffic overheads. By contrast, Triage's performance scales well with higher core counts.

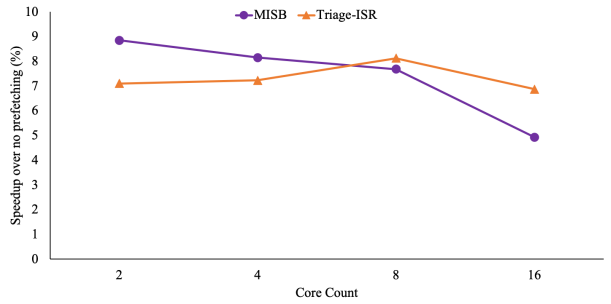


Fig. 20. Triage outperforms MISB when bandwidth is constrained.

4.4.1 *Comparison on Mixes With Regular Benchmarks*

For completeness, Figure 21 shows that Triage composes well with BO when the multi-programmed mixes include both regular and irregular programs. For a 4-core system, BO+Triage-ISR improves performance by 14.7%, whereas BO alone improves performance by 11.7%. Triage-ISR alone does not work well in this setting (3.5% speedup) because it cannot prefetch compulsory misses for regular programs.

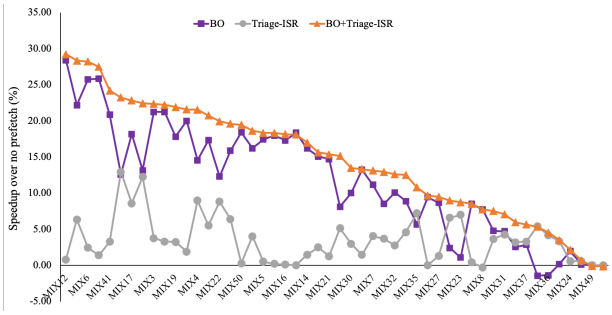


Fig. 21. Triage-ISR works well on multi-programmed mixes of regular and irregular programs running on a 4-core system.

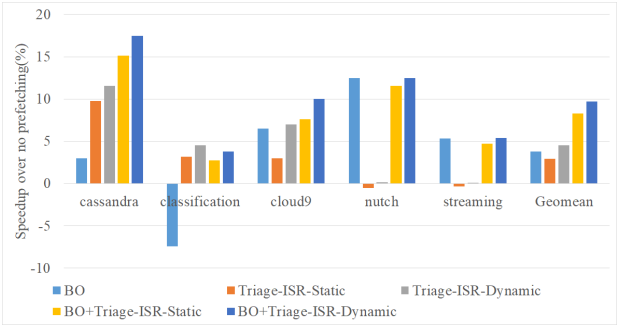
The dynamic version of Triage is essential in these scenarios because the cache is shared among irregular programs—which benefit from Triage—and regular programs—which do not benefit from Triage. For regular programs, a static version of Triage would reduce effective LLC capacity without providing much prefetching benefit. Figure 22 shows the number of ways allocated to each core on this 4-core system, and we see that (1) the total number of ways allocated to the metadata store varies across mixes, and (2) each application receives varying amounts of metadata space depending on a dynamic estimate of the usefulness of the metadata.

4.5 Evaluation on Server Workloads

We now use the CloudSuite benchmark suite running on a 4-core system to show that Triage is effective for server workloads (See Figure 4.5). On the highly irregular Cassandra, Classification, and Cloud9 benchmarks, Triage-ISR improves performance by 5.3%, whereas BO improves performance by 0.5% and SMS sees no performance gains. On the more regular Nutch and Streaming benchmarks, BO does well with 8.9% performance, whereas Triage sees no performance improvement because temporal prefetchers cannot prefetch compulsory misses.

In a hybrid setting, BO and Triage compose well, as Triage works well for the irregular benchmarks and BO works well for the regular ones. In particular, a BO+Triage-ISR hybrid outperforms all other prefetchers as it improves performance by 9.7%, whereas BO alone improves performance by only 3.8%.

Figure 4.5 also shows that Triage-ISR-Dynamic provides benefit over a static version of Triage-ISR in this setting, so we conclude that our dynamic scheme makes good decisions in trading off cache space for metadata storage. This benefit is most pronounced for the irregular benchmarks (Cassandra, Classification, and Cloud9) where the dynamic version outperforms the static scheme by 2.4% (7.6% for Triage-ISR-Dynamic vs. 5.3% for Triage-ISR-Static).



5 Conclusions

In this paper, we have introduced and evaluated the Triage data prefetcher, which represents a new design point for temporal prefetchers, one that dramatically reduces memory traffic at the expense of some coverage. Our empirical results show that this is a good tradeoff: When compared with state-of-the-art temporal prefetchers that use off-chip metadata, Triage significantly reduces traffic overhead (56.9% traffic overhead for Triage vs. 260.8% for MISB) while modestly reducing performance in bandwidth-rich environments and improving performance in bandwidth-constrained environments. When compared with other (non-temporal) prefetchers that only use on-chip metadata, Triage provides a significant performance advantage (41.1% speedup for Triage vs. 10.9% for BO).

We have also shown that by removing the use of off-chip metadata, temporal prefetchers can use vastly simpler metadata organizations. In particular, tables are a simpler and more compact representation in this setting, signifying a return to simplicity for temporal prefetchers. We have also introduced a new metadata representation that compactly

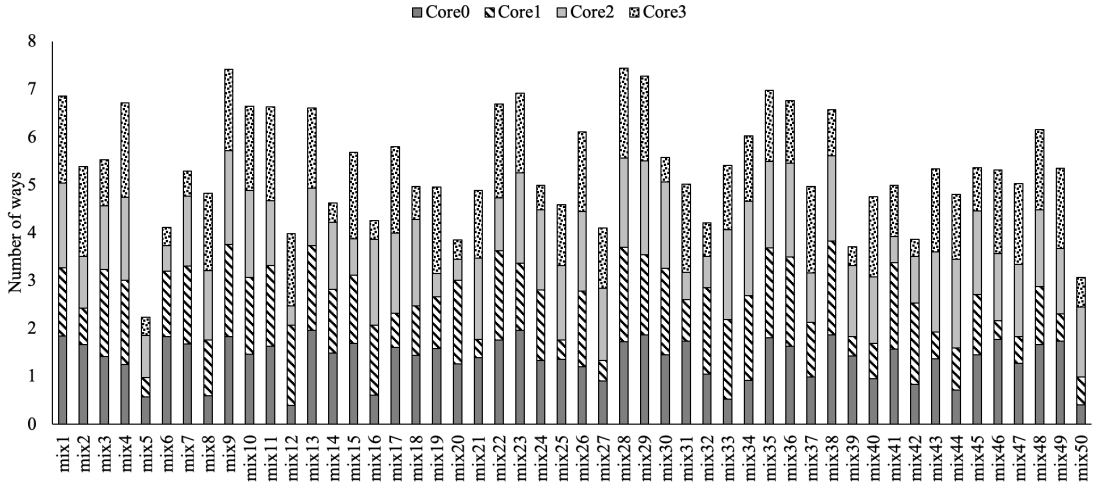


Fig. 22. Dynamic Triage allocates different metadata store sizes to different cores.

stores regular memory accesses, which is beneficial for irregular workloads, because it allows Triage to store a large amount of metadata in a limited amount of space.

Finally, this paper has made three larger points. First, we have made temporal prefetching practical for commercial deployment by removing the need for maintaining off-chip metadata, which adds complexity and energy overheads. Second, we have made temporal prefetching profitable in bandwidth-constrained environments by significantly reducing its traffic requirements. Indeed, Triage outperforms MISB in bandwidth-constrained 8-core and 16-core systems despite maintaining a metadata store that is orders of magnitude smaller. Finally, and more broadly, we show that on-chip caches can at times be used more profitably for caching metadata instead of data, which suggests future work that explores other kinds of metadata that could be usefully stored in on-chip data caches.

ACKNOWLEDGMENTS

This work was funded in part by a gift from Arm Research, NSF Grant CCF-1823546, and a gift from Intel Corporation through the NSF/Intel Partnership on Foundational Microarchitecture Research.

REFERENCES

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

[2] ——, "Making address-correlated prefetching practical," *IEEE Micro*, vol. 30, no. 1, pp. 50–59, 2010.

[3] H. Wu, K. Nathella, J. Pusdesris, D. Sunwoo, A. Jain, and C. Lin, "Temporal prefetching without the off-chip metadata," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 996–1008.

[4] A. Jain and C. Lin, "Back to the future: Leveraging belady's algorithm for improved cache replacement," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2016.

[5] D. Joseph and D. Grunwald, "Prefetching using markov predictors," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997, pp. 252–263.

[6] H. Wu, K. Nathella, A. Jain, D. Sunwoo, and C. Lin, "Efficient metadata management for irregular data prefetching," in *the 46th International Symposium on Computer Architecture (ISCA)*, 2019.

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

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

[9] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer," *IEEE Micro*, vol. 25, no. 1, pp. 90–97, 2005.

[10] Z. Hu, M. Martonosi, and S. Kaxiras, "TCP: tag correlating prefetchers," in *HPCA*, 2003, pp. 317–326.

[11] L. Peled, S. Mannor, U. Weiser, and Y. Etsion, "Semantic locality and context-based prefetching using reinforcement learning," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2015, pp. 285–297.

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

[13] Y. Chou, "Low-cost epoch-based correlation prefetching for commercial applications," in *MICRO*, 2007, pp. 301–313.

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

[15] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi, "Spatio-temporal memory streaming," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2009, pp. 69–80.

[16] I. Burcea, S. Somogyi, A. Moshovos, and B. Falsafi, "Predictor virtualization," in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS XIII. ACM, 2008, pp. 157–167.

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

[18] M. Ferdman, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Temporal instruction fetch streaming," in *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2008, pp. 1–10.

[19] A. Smith, "Sequential program prefetching in memory hierarchies," *IEEE Transactions on Computers*, vol. 11, no. 12, pp. 7–12, December 1978.

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

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

[22] S. Palacharla and R. E. Kessler, "Evaluating stream buffers as a secondary cache replacement," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, April 1994, pp. 24–33.

[23] J.-L. Baer and T.-F. Chen, "Effective hardware-based data prefetching for high-performance processors," *IEEE Transactions on Computers*, vol. 44, no. 5, pp. 609–623, May 1995.
[24] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic, "Memory-system design considerations for dynamically-scheduled processors," in *ISCA '97: Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997, pp. 133–143.
[25] Y. Ishii, M. Inaba, and K. Hiraki, "Access map pattern matching for high performance data cache prefetch," in *Journal of Instruction-Level Parallelism*, vol. 13, 2011, pp. 1–24.
[26] S. Sair, T. Sherwood, and B. Calder, "A decoupled predictor-directed stream prefetching architecture," *IEEE Transactions on Computers*, vol. 52, no. 3, pp. 260–276, March 2003.
[27] S. H. Pugsley, Z. Chishti, C. Wilkerson, P.-f. Chuang, R. L. Scott, A. Jaleel, S.-L. Lu, K. Chow, and R. Balasubramonian, "Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*. IEEE, 2014.
[28] P. Michaud, "Best-offset hardware prefetching," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 469–480.
[29] T. L. Johnson, M. C. Merten, and W.-M. W. Hwu, "Run-time spatial locality detection and optimization," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, 1997, pp. 57–64.
[30] S. Kumar and C. Wilkerson, "Exploiting spatial locality in data caches using spatial footprints," *SIGARCH Computer Architecture News*, vol. 26, no. 3, pp. 357–368, April 1998.
[31] D. Burger, T. R. Puzak, W.-F. Lin, and S. K. Reinhardt, "Filtering superfluous prefetches using density vectors," in *ICCD '01: Proceedings of the International Conference on Computer Design: VLSI in Computers & Processors*, 2001, pp. 124–133.
[32] C. F. Chen, S.-H. Yang, B. Falsafi, and A. Moshovos, "Accurate and complexity-effective spatial pattern prediction," in *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, ser. HPCA '04, 2004, pp. 276–288.
[33] J. Collins, S. Sair, B. Calder, and D. M. Tullsen, "Pointer cache assisted prefetching," in *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 35, 2002, pp. 62–73.
[34] A. Roth and G. S. Sohi, "Effective jump-pointer prefetching for linked data structures," in *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA)*, 1999, pp. 111–121.
[35] R. Cooksey, S. Jourdan, and D. Grunwald, "A stateless, content-directed data prefetching mechanism," *SIGARCH Computer Architecture News*, vol. 30, no. 5, pp. 279–290, October 2002.
[36] E. Ebrahimi, O. Mutlu, and Y. N. Patt, "Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems," in *HPCA*, 2009, pp. 7–17.
[37] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.
[38] S. Hong, B. Abali, A. Buyuktosungolu, M. Healy, and P. Nair, "Touche: Towards ideal and efficient cache compression by mitigating tag area overhead," in *MICRO'19: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 453–465.
[39] S. Kumar, H. Zhao, A. Shriraman, E. Matthews, S. Dwarkadas, and L. Shannon, "Amoeba-cache: Adaptive blocks for eliminating waste in the memory hierarchy," in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012, pp. 376–388.
[40] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2010, pp. 60–71.
[41] *2nd Cache Replacement Championship*, 2017. [Online]. Available: http://crc2.ece.tamu.edu/
[42] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," *ACM SIGOPS Operating Systems Review*, vol. 36, no. 5, pp. 45–57, 2002.
[43] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, 2012, pp. 37–48.
[44] M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Domino temporal data prefetcher," in *High Performance Computer Architecture (HPCA), 2018 IEEE 24th International Symposium on*, 2018, pp. 131–142.
[45] S. Borkar, "The Exascale Challenge," https://parasol.tamu.edu/pact11/ShekarBorkar-PACT2011-keynote.pdf, October 2011.
[46] B. Jacob, S. Ng, and D. Wang, *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.

Hao Wu is a Software Engineer at Google Platform. He received his Ph.D. in Computer Science from The University of Texas at Austin in July 2020. He received B.S. in Computer Science and Technology from Tsinghua University in July 2010. His works focuses on computer architecture, especially modeling and improving server performance through software simulation.

Krishnendra Nathella is a Staff Research Engineer at Arm Research. He graduated with a Masters in Computer Engineering from the University of Wisconsin at Madison in 2012. In 2010 he received his B.Tech degree in Electronics and Communication Engineering from SASTRA University, India. Krishnendra joined Arm Research in 2012 and has since worked on a variety of topics in computer architecture and currently focuses on resolving the front-end bottlenecks and memory bottlenecks in modern high-performance processors through developing better branch prediction, instruction prefetching and data prefetching techniques.

Matthew Pabst is an undergraduate at The University of Texas at Austin in the Turing Scholars Honors Program. He has research interests in architecture, systems, and security.

Dam Sunwoo is a Senior Principal Research Engineer at Arm Research. He received his M.S. and Ph.D. in Electrical and Computer Engineering at The University of Texas at Austin in 2005 and 2010, respectively. He received his B.S. degree in Electrical Engineering from Seoul National University in 2003. He has broad interests in computer architecture with a focus on microarchitecture for high-performance processors.

Akanksha Jain is a Research Engineer at Arm Research. She received her Ph.D. in Computer Science from The University of Texas in August 2016. In 2009, she received the B.Tech and M. Tech degrees in Computer Science and Engineering from the Indian Institute of Technology Madras. Her research interests are in computer architecture, with a particular focus on the memory system and on using machine learning techniques to improve the design of memory system optimizations.

Calvin Lin is a University Distinguished Teaching Professor at The University of Texas at Austin, where he is also the Director of the Turing Scholars Honors Program. He received the B.S.E. in Computer Science from Princeton University in 1984 and the Ph.D. in Computer Science from the University of Washington in 1992. He has broad interests in systems research, including compilers and computer architecture.