Temporal Prefetching Without the Off-Chip Metadata

Hao Wu
haowu@cs.utexas.edu
The University of Texas at Austin
Austin, Texas

Krishnendra Nathella
Krishnendra.Nathella@arm.com
Arm Inc.
Austin, Texas

Joseph Pusdesris
Joseph.Pusdesris@arm.com
Arm Inc.
Austin, Texas

Dam Sunwoo
Dam.Sunwoo@arm.com
Arm Inc.
Austin, Texas

Akanksha Jain
akanksha@cs.utexas.edu
The University of Texas at Austin
Austin, Texas

Calvin Lin
lin@cs.utexas.edu
The University of Texas at Austin
Austin, Texas

ABSTRACT

Temporal prefetching offers great potential, but this potential is difficult to achieve because of the need to store large amounts of prefetcher metadata off chip. To reduce the latency and traffic of off-chip metadata accesses, recent advances in temporal prefetching have proposed increasingly complex mechanisms that cache and prefetch this off-chip metadata. This paper suggests a return to simplicity: We present a temporal prefetcher whose metadata resides entirely on chip. The key insights are (1) only a small portion of prefetcher metadata is important, and (2) for most workloads with irregular accesses, the benefits of an effective prefetcher outweigh the marginal benefits of a larger data cache. Thus, our solution, the Triage prefetcher, identifies important metadata and uses a portion of the LLC to store this metadata, and it dynamically partitions the LLC between data and metadata.

Our empirical results show that when compared against spatial prefetchers that use only on-chip metadata, Triage performs well, achieving speedups on irregular subset of SPEC2006 of 23.5% compared to 5.8% for the previous state-of-the-art. When compared against state-of-the-art temporal prefetchers that use off-chip metadata, Triage sacrifices performance on single-core systems (23.5% speedup vs. 34.7% speedup), but its 62% lower traffic overhead translates to better performance in bandwidth-constrained 16-core systems (6.2% speedup vs. 4.3% speedup).

CCS CONCEPTS

• **Computer systems organization → Processors and memory architectures**.

KEYWORDS

Data prefetching, irregular temporal prefetching, caches, CPUs

ACM Reference Format:
Hao Wu, Krishnendra Nathella, Joseph Pusdesris, Dam Sunwoo, Akanksha Jain, and Calvin Lin. 2019. Temporal Prefetching Without the Off-Chip Metadata. In *The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52), October 12–16, 2019, Columbus, OH, USA.* ACM, New York, NY, USA, 13 pages.

MICRO-52, October 12–16, 2019, Columbus, OH, USA

ACM ISBN 978-1-4503-6938-1/19/10…$15.00
DOI: 10.1145/3352460.3358300

1 INTRODUCTION

By hiding the long latencies of DRAM accesses, data prefetchers are critical components of modern memory systems. One particularly elusive form of prefetching, known as temporal prefetching, is promising because it identifies arbitrary streams of correlated memory addresses, so it is suitable for irregular workloads, including those that use pointer-based data structures. Unfortunately, temporal prefetching essentially memorizes pairs of correlated addresses, so it requires metadata that are too large to fit on chip.

Early forms of temporal prefetching explored metadata organizations that reduce the cost of accessing off-chip metadata by amortizing metadata accesses across multiple prefetches [45]. More recently, the Irregular Stream Buffer (ISB) [24] introduced an alternative metadata representation that enabled portions of the metadata to be cached on chip. This metadata caching scheme was subsequently revised with a new metadata management scheme that includes a metadata prefetcher, reducing metadata traffic from 482% to 156% [47].

Unfortunately, the presence of off-chip metadata in these solutions presents three issues. First, even 156% metadata traffic overhead can impact performance, particularly in bandwidth-constrained multi-core systems. Second, off-chip metadata traffic consumes significant energy, since DRAM accesses consume more power than on-chip operations. Third, off-chip metadata adds hardware complexity because it requires (1) changes to the memory interface, (2) communication with the OS, and (3) methods for managing the metadata, which can include both a metadata cache replacement policy and a metadata prefetcher [47].

In this paper, we present a new temporal data prefetcher that addresses these issues by not maintaining any off-chip metadata. Our work is motivated by two observations. First, most of the coverage for state-of-the-art temporal prefetchers [24, 47] comes from a small number of metadata entries, so it is possible to get substantial coverage without storing megabytes of metadata (see Figure 1). Second, the marginal utility of the last-level cache (LLC) [34, 40] is typically outweighed by the benefits of an effective prefetcher. For example, for the irregular subset of SPEC2006, reducing the cache by 1 MB reduces performance by 7.4%, but a state-of-the-art irregular prefetcher with unlimited resources can improve performance by 41.7%. Therefore, if we can distinguish the important metadata

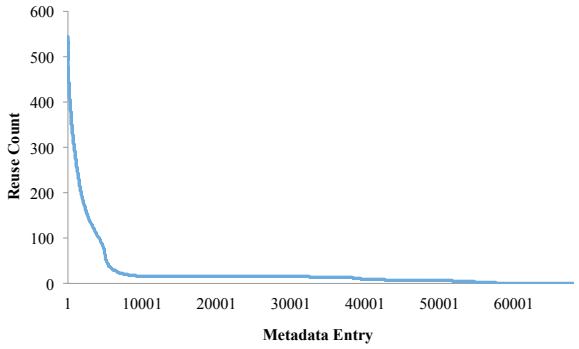


Figure 1: Metadata reuse distribution for the *mcf* benchmark: For an execution with 60K metadata entries, only 15% of metadata entries are reused more than 15 times.

from the unimportant metadata, it can be profitable to use large portions of the LLC to store important prefetcher metadata.

Thus, our prefetcher, which we call the Triage prefetcher, repurposes 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 [25], 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 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.

By forsaking off-chip metadata and by intelligently managing on-chip metadata, Triage offers vastly different tradeoffs than state-of-the-art temporal prefetchers. For example, Triage reduces off-chip traffic overhead from 156.4% to 59.3%, 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 prefetcher metadata), but they translate to better performance in bandwidth-constrained environments.

Triage's metadata organization has the added benefit of a simplified and compressed metadata representation. In particular, we find that without the need to store metadata off chip, tables [27] are the most compact data structure for tracking correlated addresses, because they have no redundancy. By contrast, previous solutions [24, 45] introduce varying degrees of metadata redundancy to facilitate off-chip metadata management. Since our metadata store competes for space in the LLC, this compactness has a direct performance benefit.

To summarize, this paper makes several contributions:

- We introduce Triage, the first PC-localized[1] temporal data prefetcher that does not use off-chip metadata. Triage reuses a portion of the LLC for storing prefetcher metadata, and it includes a simple adaptive policy for dynamically provisioning the size of the metadata store.
- We evaluate the Triage prefetcher 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 (23.5% speedup for Triage vs. 5.8% for the Best Offset Prefetcher, 2.8% for SMS). Triage achieves 70% of the performance of a state-of-the-art temporal prefetcher that uses off-chip metadata (23.5% for Triage vs. 34.7% for MISB [47]).
  - On a 4-core system running CloudSuite server benchmarks, BO+Triage improves performance by 13.7%, compared to 8.6% for BO alone.
  - On a 16-core system running multi-programmed irregular SPEC workloads, where bandwidth is more precious, Triage's speedup is 6.2%, compared to 4.3% for MISB.
  - Triage also works well as part of a hybrid prefetcher that combines a regular prefetcher with a temporal prefetchers (24.8% for Triage+BO vs. 5.8% for BO alone on single-core systems, and 10.0% for BO+Triage vs. 4.4% for BO alone on 16-core systems).
- 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. Unfortunately, considerable state is required to memorize correlations among addresses. To reduce metadata requirements, some prefetchers forgo address correlation to learn weaker forms of correlation, such as delta correlation [33], tag correlation [20], or context-address pair correlation [36], but these simplifications limit the scope of memory access patterns that can be learned. Other prefetchers exploit address correlation by storing metadata in off-chip memory [24, 45, 47], 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 [24]—without using any off-chip metadata.

Temporal prefetchers with off-chip metadata can be placed in three categories based on their metadata organization. The primary focus of all three categories is to reduce the overhead of accessing off-chip metadata. While elements of Triage borrow from this existing work, Triage is designed with a completely different design

[1]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.

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.

Table-Based Temporal Prefetchers. Joseph and Grunwald introduced the idea of prefetching correlated addresses in 1997 with their Markov Prefetcher [27]. 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 [20].

Therefore, early temporal prefetchers explore designs that reduce the traffic and latency costs of accessing an off-chip Markov table [10, 43]. 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× to 4×. Second, Triage uses a customized replacement policy that identifies the most useful table entries, removing the need for off-chip metadata.

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 [9, 45]. Their STMS prefetcher [45, 46] 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. 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%.

Irregular Stream Buffer. The Irregular Stream Buffer (ISB) combines address correlation with PC-localization by proposing a new off-chip metadata organization [24, 47]. 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 [47] 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 156%. 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× 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 [6, 17, 44], 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. Finally, metadata optimizations have been explored in the context of memory compression [13, 19, 48], but these techniques are orthogonal to our work.

2.2 Non-Temporal Prefetching

Many prefetchers predict sequential [21, 28, 42] and strided [3, 15, 22, 32, 35, 37, 39] 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 [7, 8, 26, 30, 44], such that recurring spatial patterns can be prefetched across different regions in memory. For example, the SMS prefetcher [44] 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 [11, 12, 14, 38]. For example, Content Directed Prefetching [12] 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

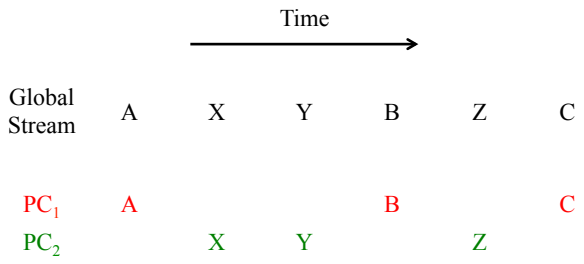
Triage repurposes on-chip cache space to store prefetcher metadata. To effectively utilize valuable on-chip cache space, Triage considers the following design questions:

- 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 our solution addresses these questions.

Metadata Representation. Triage learns PC-localized correlated address pairs and records them in a table. For example, the top side

of Figure 2 shows a stream of memory references that is segregated into two *PC-localized* streams, and the bottom side shows the conceptual organization of Triage's metadata. In particular, each entry in Triage maps an address to its PC-localized neighbor.



Addr	Neighbor
A	B
B	C
X	Y
Y	Z

Figure 2: 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 [24, 45, 47], our table-based organization avoids metadata redundancy by representing each correlated address pair only once. 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.)

Section 3.2 provides details about Triage's use of compact address representations to further reduce the metadata footprint.

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 1). 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.

To accomplish these goals, we modify Hawkeye [25], 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 3 shows a high-level overview of Hawkeye, where OPTgen is

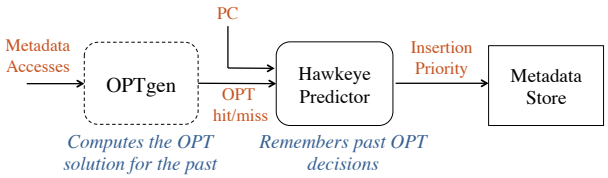


Figure 3: Triage's metadata replacement is based on the Hawkeye [25] cache replacement policy.

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.

Because the Hawkeye policy 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 the Hawkeye policy 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 3.2, we explain how Triage is able to manage metadata at a finer granularity than the line size of the last-level cache.

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. Our dynamic cache allocation scheme is based on two insights. First, the OPTgen component of Hawkeye can cheaply model the optimal hit rate at different metadata store sizes, so OPTgen can be used to estimate the profitability of devoting more cache space to metadata entries. Second, the optimal hit rate scales linearly with cache size, so we need not estimate optimal hit rate at every possible metadata store size—we can instead estimate hit rate at two points and interpolate.

More concretely, we maintain two copies of OPTgen (each copy consumes 1KB space), and we use these copies as sandboxes to evaluate the optimal hit rate at different metadata store sizes. If Triage finds that an increase in the metadata store size increases optimal metadata hit rate by more than 5%, it increases the number of ways that are allocated to metadata entries. Similarly, if Triage finds that a reduction of the metadata store size decreases the metadata hit rate by less than 5%, it reduces the number of ways allocated to metadata entries. For simplicity, Triage chooses between three possible allocations for metadata store(0 MB, 512 KB and 1 MB), but our scheme can be extended to any number of partitioning configurations by time-sharing the OPTgen copies to evaluate different metadata store sizes.

The partition sizes are re-evaluated periodically to adapt to changes in program phases.

3.1 Overall Operation

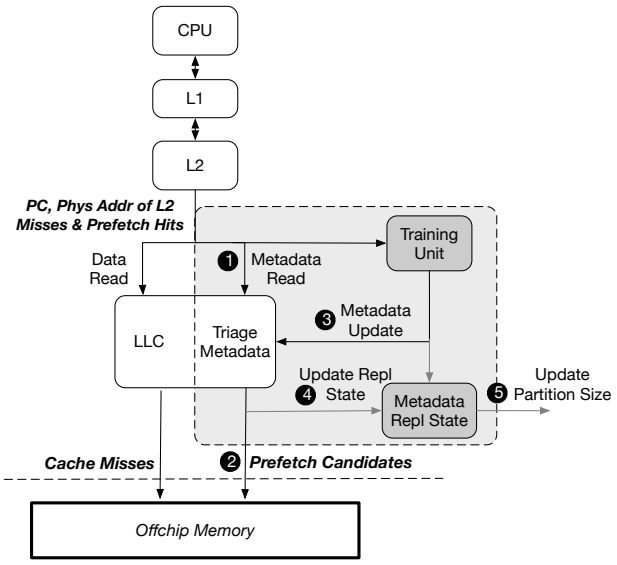


Figure 4: Overview of Triage.

Figure 4 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 the metadata replacement state periodically recomputes the amount of LLC that should be used as a metadata cache ❺. We now explain these operations in more detail.

Training. The Training Unit 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, and the entry (A, B) is stored in Triage's metadata store; the metadata store is indexed by the first address in the pair (A in this example).

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.

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)) is found, Triage issues a prefetch for B. If an entry is not found, no prefetch is issued.

Metadata Replacement Updates. Our metadata replacement is based on the Hawkeye policy [25], 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 [25].

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 (by using per-core OPTgens) and allots the corresponding portion of the LLC for each core's metadata. For example, if two cores are sharing a 4MB cache, and if core 0 wants 1MB of metadata and core 1 wants 512KB of metadata, then Triage allocates 1.5MB of the shared LLC for metadata, and it partitions the metadata store in a 2:1 ratio among the two cores.

3.2 Hardware Design

Each metadata entry in Triage is 4 bytes long, but LLC line sizes are typically 64 to 128 bytes. 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. Therefore, we store multiple tagged metadata entries within each LLC cache line. For example, for a 64 byte LLC line, we store 16 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 11 bits, and the remaining bits are tags. We construct a lookup table to compress the tag to 10 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 31 bits[2]. The remaining bit is used as a confidence counter.

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 AmoebaCache [31] and may incur additional latency or pipeline stages, but only for metadata accesses. In Section 4.6, we describe a sensitivity study that penalizes LLC access latencies for both data and metadata by up to 6 cycles, and we see that the performance impact is minimal (around 1% lower speedup on average for the irregular SPEC workloads).

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

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 1. This model uses a simple memory model with fixed latency, but it models memory bandwidth constraints accurately.

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
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 1: Machine Configuration

For multi-core evaluation of Triage, we use ChampSim [2, 29], 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 1. We confirm that the performance trends on the two simulators are the same.

Benchmarks. We present single-core results for a subset of SPEC2006 benchmarks that are memory bound [18] and are known to have irregular access patterns [24]. For SPEC benchmarks we use the reference input set. For all single-core benchmarks, we use SimPoints [41] 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 [16] 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.

Prefetchers. We evaluate Triage against two state-of-the-art on-chip prefetchers, namely, Spatial Memory Streaming (SMS) [44] and the Best Offset Prefetcher (BO) [32]. SMS captures irregular patterns by applying irregular spatial footprints across memory regions. BO is a regular prefetcher that won the Second Data Prefetching Championship [1].

We also evaluate Triage against existing off-chip temporal prefetchers, namely, Sampled Temporal Memory Streaming (STMS) [45], Domino [4], and MISB [47]. 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 complete instantly with no latency or traffic penalty. 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, static and dynamic. The static version picks a fixed metadata store size that gives the best average performance and statically partitions the LLC using this size; we find that the best static metadata store size for a 2MB LLC is 1MB on both simulators. The dynamic version of Triage modulates the size of the metadata store dynamically as described in Section 3.

All prefetchers train on the L2 access stream, 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 5 shows that Triage outperforms state-of-the-art prefetchers that use only on-chip metadata. In particular, Triage achieves a speedup of 23.4% and 23.5% for the static and dynamic configurations, respectively, whereas BO and SMS see a speedup of 5.8% and 2.2%, respectively. Triage's superior performance can be explained by its higher coverage (42.0% for Triage vs. 13.0% for BO and 4.6% for SMS) and higher accuracy (77.2% for Triage vs. 43.3% for BO and 39.6% for SMS) as shown in Figure 6.

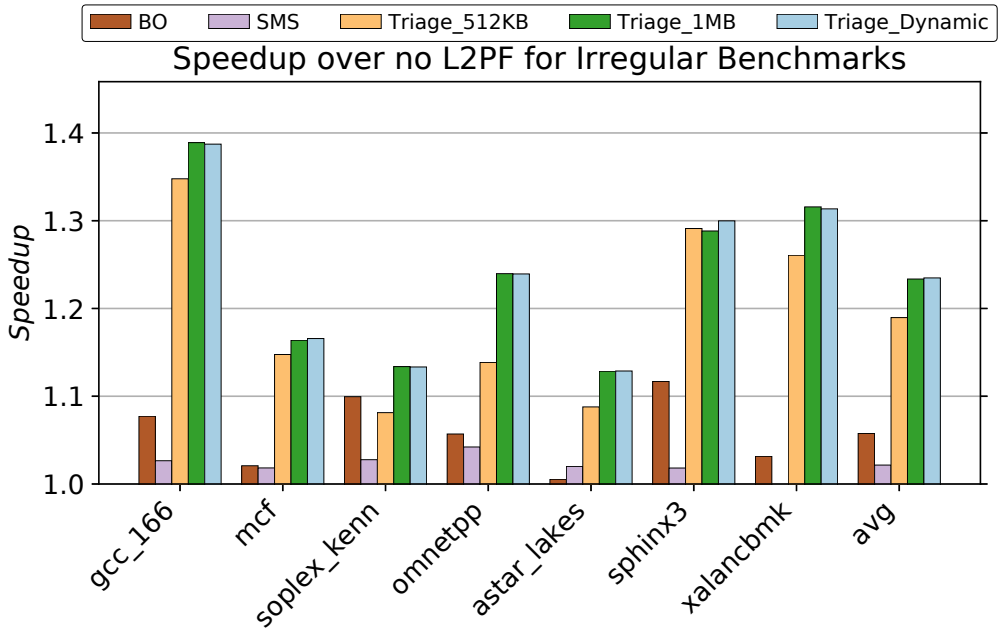


Figure 5: Triage outperforms BO and SMS

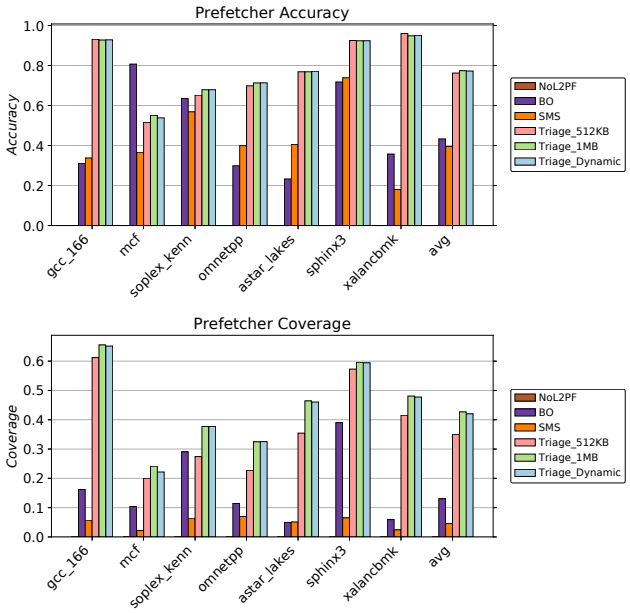


Figure 6: Triage improves coverage and accuracy.

Triage-Dynamic is slightly better than Triage-Static as it modulates the metadata store size for mcf and sphinx3. As we will see later, the benefit of our dynamic scheme is most pronounced in a shared cache setting where the cache is shared by both regular and irregular benchmarks.

Figure 7 sheds more insight on Triage's performance benefits, as we see 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 that is given a 1 MB on-chip metadata store in addition to its usual LLC capacity achieves a 31.2% speedup. On the other hand, a system with no Triage and a reduced LLC capacity of 1 MB achieves 7.4% lower performance than the baseline. This loss in performance is easily compensated by Triage's benefits as Triage sees an overall speedup of 23.4% with a 1 MB metadata store and a 1 MB LLC.

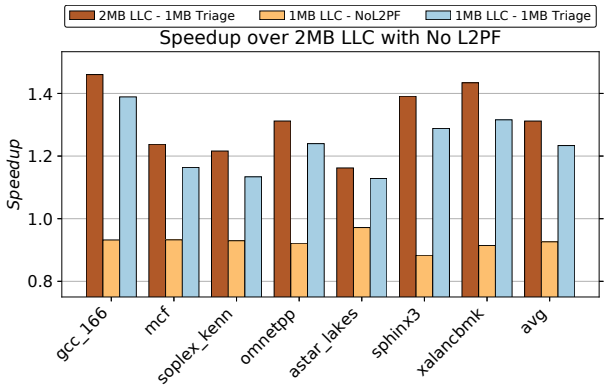


Figure 7: Breakdown of Triage's Performance Improvements

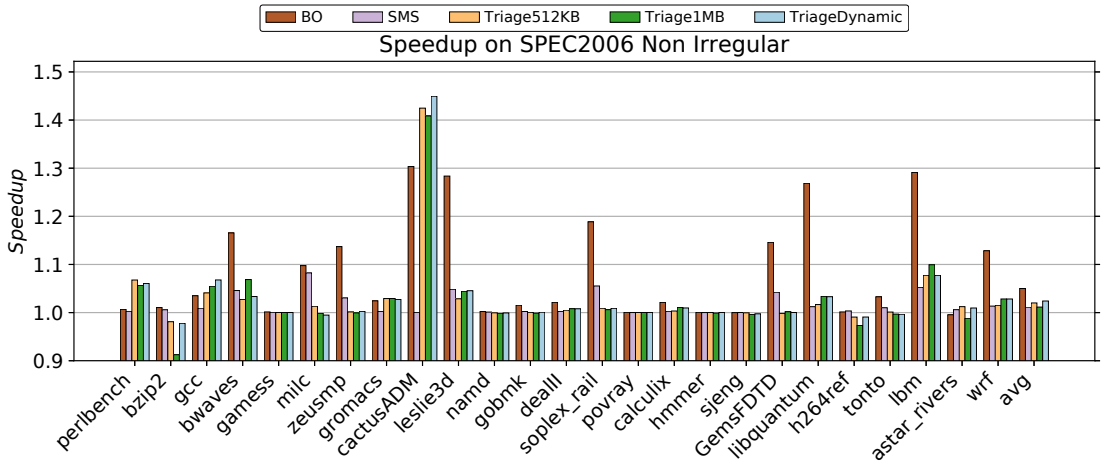


Figure 8: Results on regular SPEC 2006 benchmarks.

For completeness, Figure 8 compares all prefetchers on the remaining memory-intensive SPEC 2006 benchmarks[3]. Because these benchmarks are regular, Triage does not outperform BO, but we see that Triage's dynamic partitioning avoids hurting performance on most benchmarks as it picks a 512 KB metadata store instead of a 1 MB metadata store. On bzip2, Triage hurts performance because it detects metadata reuse, but the prefetches issued by these metadata entries are not enough to cover the loss in LLC space. As future work, more sophisticated partitioning schemes that account for cache utility more accurately could help improve Triage in these scenarios.

Sensitivity to Replacement Policy. Figure 9 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 7.7% speedup whereas Triage with the Hawkeye policy sees a 13.7% speedup.

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 10 shows that a BO+Triage hybrid outperforms BO (24.8% speedup for BO+Triage vs. 5.8% for BO), which shows that Triage successfully prefetches lines that BO cannot.

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

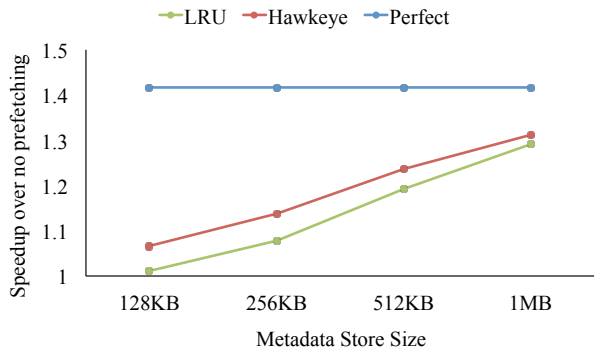


Figure 9: Sensitivity to Metadata Store Size (assuming no loss in LLC capacity).

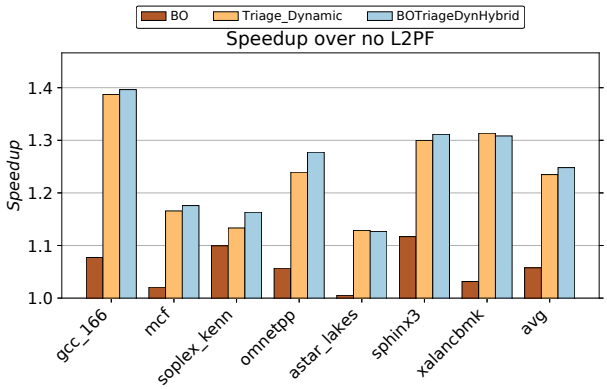


Figure 10: Triage performs well as part of a hybrid prefetcher.

4.3 Comparison With Prefetchers That Use Off-Chip Metadata

Existing temporal data prefetchers use tens of megabytes of off-chip metadata. Compared to these prefetchers, Triage provides a simpler design and a more desirable tradeoff between performance and off-chip metadata traffic. Figure 11 compares Triage against overly optimistic idealized versions of STMS and Domino and against a realistic version of MISB [47]. We see that Triage outperforms idealized STMS and Domino (23.5% for Triage vs 14.5% for Domino and 15.3% for STMS). Triage doesn't match MISB's 34.7% performance, but we see that it incurs much less traffic overhead (bottom graph in Figure 11). In particular, compared to a baseline with a 2 MB cache and no prefetching, Triage increases traffic by 59.3%, whereas STMS, Domino and MISB increase traffic by 482.9%, 482.7%, and 156.4% respectively.

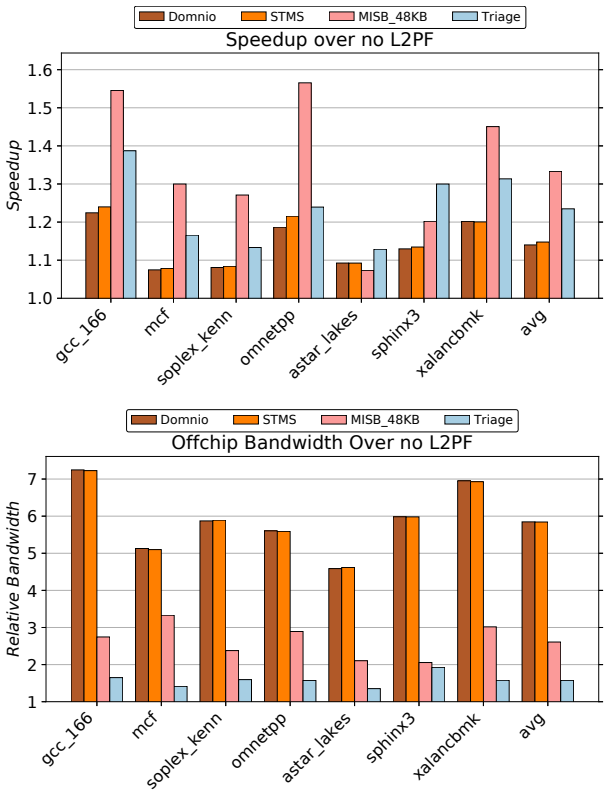


Figure 11: Triage reduces traffic compared to off-chip temporal prefetchers while offering good performance improvements.

To put these results in context, Figure 12 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 introduced by 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 comparable to BO's 33.8% 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 traffic due to an effectively smaller LLC.

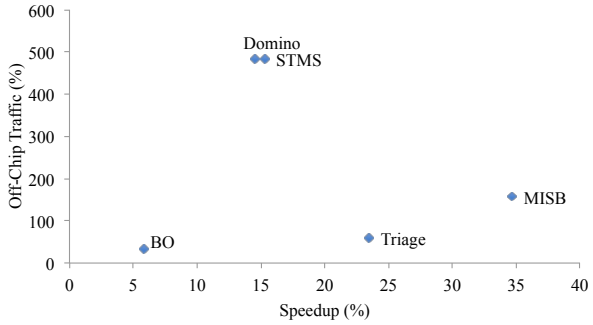


Figure 12: Design Space of Temporal Prefetchers.

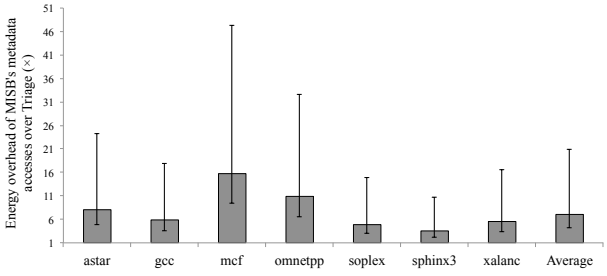


Figure 13: Triage is more energy efficient than MISB.

Energy Evaluation. Triage is more energy-efficient than other temporal prefetchers. Figure 13 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 [5, 23], 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.

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 Server Workloads

To evaluate its effectiveness for server workloads, we evaluate Triage on the CloudSuite benchmark suite running on a 4-core system (See Figure 14). On the highly irregular Cassandra, Classification, and Cloud9 benchmarks, Triage improves performance by 7.8%, whereas BO improves performance by 4.8% and SMS sees no performance gains. On the more regular Nutch and Streaming benchmarks, SMS and BO do well (10.9% and 14.7% performance improvement), 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 hybrid outperforms all other prefetchers as it improves performance by 13.7%, whereas BO alone improves performance by only 8.6% (50.6% miss reduction for BO+Triage vs. 31.4% miss reduction for BO). A BO+SMS hybrid (5.8% speedup) does not provide much improvement and, in fact, degrades performance compared to BO alone, because both BO and SMS target regular access patterns, so when they are combined, their collective inaccuracy creates more contention for bandwidth.

Figure 14 also shows that Triage-Dynamic provides benefit over a static version of Triage in this setting, so we conclude that our dynamic scheme makes good decisions about 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 2.3% (7.8% for Triage-Dynamic vs. 5.5% for Triage-Static).

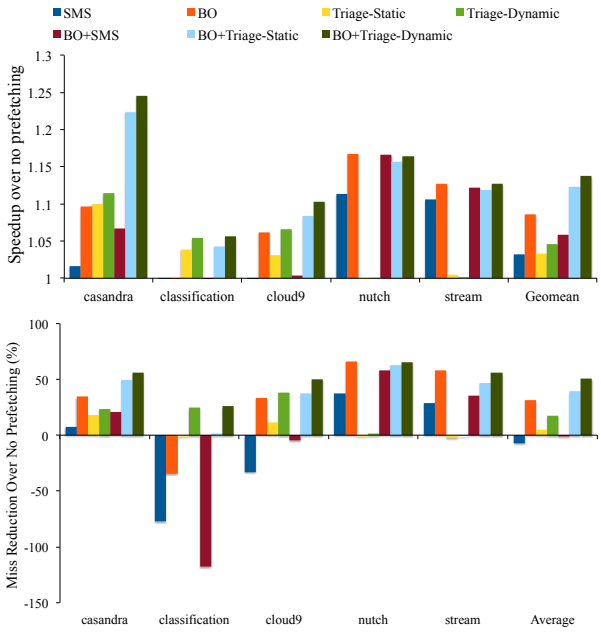


Figure 14: Triage works well for server workloads.

4.5 Evaluation on Multi-Programmed SPEC Mixes

Figure 15 shows that for multi-programmed mixes of SPEC programs sharing the last-level cache, Triage-Dynamic is a significant improvement over Triage-Static. In particular, for mixes of irregular workloads sharing an 8 MB LLC on a 4-core system, a static version of Triage with 4 MB of metadata and 4 MB of data improves performance by only 4.8%. By contrast, Triage-Dynamic improves performance by 10.2%.

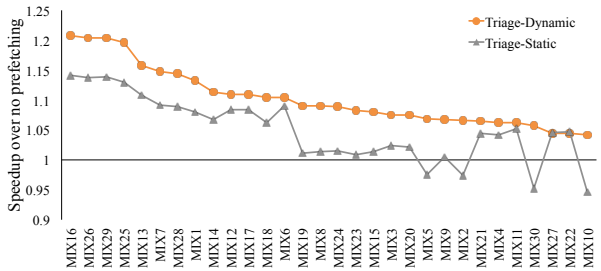


Figure 15: Triage-Dynamic improves over Triage-Static for shared caches.

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

Comparison With Prefetchers That Store Metadata On Chip. Figure 16 shows that Triage compares favorably to spatial prefetchers, such as BO, on 4-core systems. In particular, a combination of BO and Triage-Dynamic outperforms BO alone on a 4-core system, as we see that BO improves performance by 10.6%, Triage-Dynamic improves performance by 10.2%, and a combination of BO and Triage-Dynamic improves performance by 15.9%. These results reiterate that Triage can prefetch irregular memory accesses that BO cannot.

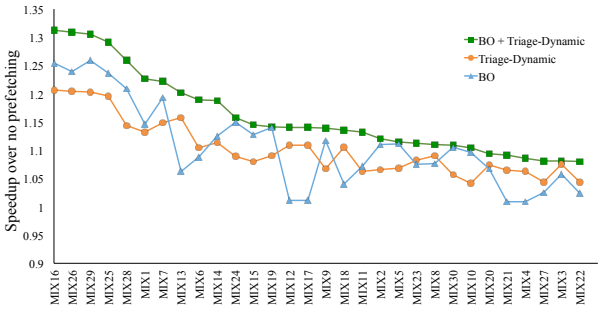


Figure 16: Triage works well on multi-programmed mixes of irregular programs running on a 4-core system.

We observe similar trends on 8-core and 16-core systems. On an 8-core system, BO+Triage improves performance by 12.6% (vs. 7.4% for BO alone), and on a 16-core system, BO+Triage improves performance by 10.0% (vs. 4.4% for BO alone).

Comparison With Prefetchers That Store Metadata Off Chip. Figure 17 compares the average speedup of Triage 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 on a 2-core system (12.1% for Triage vs. 16.0% for MISB), its benefit shrinks on an 8-core system (8.8% for Triage vs. 10.0% for MISB). On a 16-core system, Triage outperforms MISB (6.2% for Triage vs. 4.3% 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.

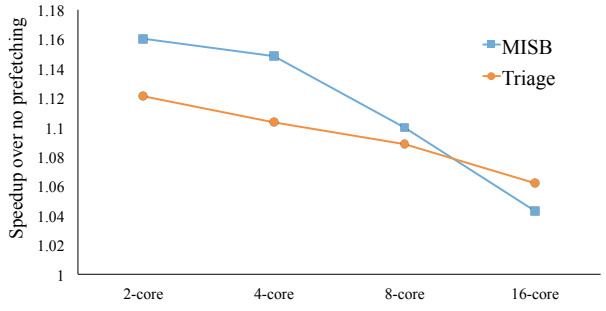


Figure 17: Triage outperforms MISB in bandwidth-constrained environments.

Comparison On Mixes With Regular Programs. For completeness, Figure 18 shows that Triage composes well with BO when the multi-programmed mixes include both regular and irregular programs. In particular, for a 4-core system, BO+Triage improves performance by 23%, whereas BO alone improves performance by 19.3%. Triage alone does not work well in this setting (4.3% speedup) because it cannot prefetch compulsory misses for regular programs.

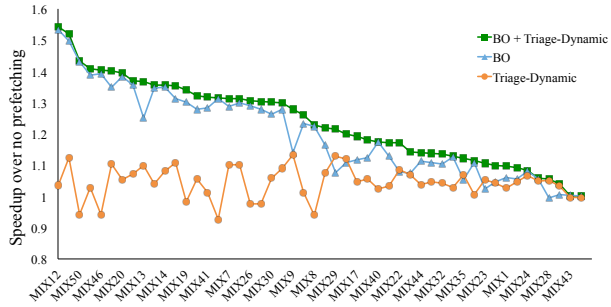


Figure 18: Triage works well on multi-programmed mixes of regular and irregular programs running on a 4-core system.

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 19 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.

For example, the leftmost bar in Figure 19 represents a mix with 1 regular program (milc on core 0), two irregular programs (xalancbmk on core 1 and omnetpp on core 3), and one regular/irregular program (bzip2 on core 2). For this mix, Triage-Dynamic allocates an average of 22% of the LLC capacity to metadata (the maximum metadata allocation can go up to 50% of the LLC). It distributes this metadata store appropriately among different workloads: Milc is not allocated any metadata space because it does not benefit from irregular prefetching, omnetpp is allocated the maximum metadata space (10% of total LLC capacity) because it benefits the most from irregular prefetching, and the other benchmarks are allocated 6% each.

4.6 Sensitivity Studies

Sensitivity to Degree. Figure 20 shows the performance of our prefetchers at different prefetch degrees. As we increase degree from 1 to 8, Triage's performance grows from 23.5% to 36.2%, and its performance saturates at a degree of 8. By comparison, BO and SMS at degree 8 improve performance by 11.1% and 7.0% (over a baseline with no prefetcher), respectively. Triage is consistently more accurate than BO: At higher degrees, BO's accuracy is only 21.5% (vs. 50.5% for Triage).

Sensitivity to Epoch Length. Triage-Dynamic periodically recomputes the fraction of the LLC that should be repurposed for metadata. We find that Triage's metadata partitions are stable over long periods of time and that resizing partitions more frequently than 50,000 LLC accesses does not affect performance. Thus, we conclude that while metadata partition sizes vary significantly across benchmarks, they tend to change infrequently for a given benchmark.

Sensitivity to Changes in Cache Latency. As discussed in Section 3.2, the metadata portion of the LLC is managed at a finer granularity than the data portion of the LLC. We expect these modifications to only affect the latency of metadata accesses, not the latency of data accesses. However, to study the worst-case scenario, we increase the latency of accessing both data and metadata up to 6 cycles, and we find that even with 6 cycles of additional latency, we only see a 1% drop in performance (normalized to a baseline with no prefetching and no extra latency).

5 CONCLUSIONS

Temporal prefetchers can be highly effective for irregular memory access patterns, but they have yet to be commercially adopted because they need to store large amounts of metadata in DRAM. This off-chip metadata adds complexity, adds energy overhead, and incurs significant traffic. In this paper, we have presented Triage,

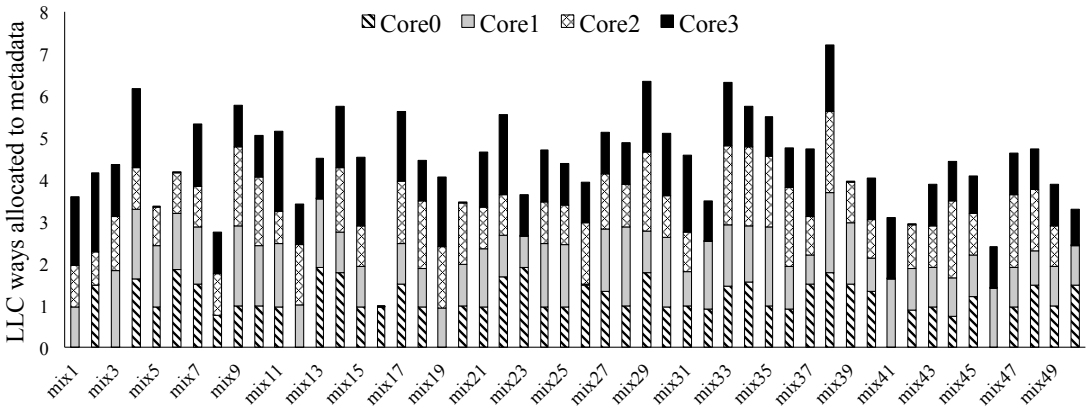


Figure 19: Dynamic Triage allocates different metadata store sizes to different cores.

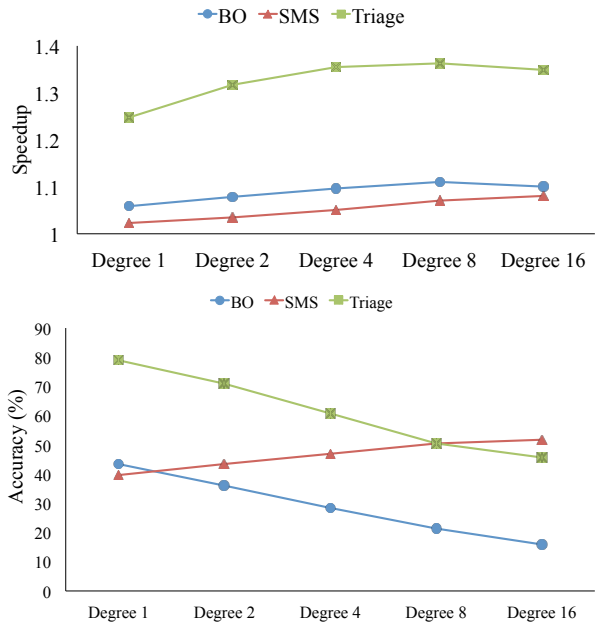


Figure 20: Sensitivity to Prefetch degree.

a temporal prefetcher that removes the off-chip metadata requirement and stores metadata only on chip, making it practical to implement.

To store its metadata on chip, Triage re-purposes the LLC as a metadata store because for irregular workloads, the marginal benefit of a larger cache is significantly outweighed by the benefit of an effective temporal prefetcher. To ensure that Triage does not degrade performance for workloads that do not benefit from irregular prefetching, Triage includes a dynamic mechanism for intelligently partitioning the cache between data and metadata based on the utility of the metadata.

We have shown that when compared with prefetchers that only use on-chip metadata, Triage provides a significant performance advantage (23.5% speedup for Triage vs. 5.8% for BO). When compared with state-of-the-art temporal prefetchers that use off-chip metadata, Triage significantly reduces traffic overhead (59.3% traffic overhead for Triage vs. 156.4% for MISB). This traffic reduction translates to better speedup in bandwidth-constrained 16-core systems, where Triage outperforms MISB despite having access to a metadata store that is orders of magnitude smaller. Triage's traffic overhead is comparable to state-of-the-art spatial prefetchers, such as BO, which speaks to its practicality. Overall, Triage provides a new and attractive design point for temporal prefetchers with vastly different tradeoffs than previous solutions.

ACKNOWLEDGMENTS

We thank Jaekyu Lee for his help in setting up the proprietary simulation infrastructure. This work was funded in part by NSF Grant CCF-1823546 and a gift from Intel Corporation through the NSF/Intel Partnership on Foundational Microarchitecture Research.

REFERENCES

[1] 2015. *2nd Data Prefetching Championship* (2015). http://comparch-conf.gatech.edu/dpc2

[2] 2017. *2nd Cache Replacement Championship* (2017). http://crc2.ece.tamu.edu/

[3] Jean-Loup Baer and Tien-Fu Chen. 1995. Effective Hardware-Based Data Prefetching for High-Performance Processors. *IEEE Trans. Comput.* 44, 5 (May 1995), 609–623.

[4] Mohammad Bakhshalipour, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2018. Domino Temporal Data Prefetcher. In *High Performance Computer Architecture (HPCA), 2018 IEEE 24th International Symposium on*. 131–142.

[5] Shekhar Borkar. 2011. The Exascale Challenge. https://parasol.tamu.edu/pact11/ShekarBorkar-PACT2011-keynote.pdf.

[6] Ioana Burcea, Stephen Somogyi, Andreas Moshovos, and Babak Falsafi. 2008. Predictor virtualization. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems (ASPLOS XIII)*. ACM, 157–167.

[7] Doug Burger, Thomas R. Puzak, Wei-Fen Lin, and Steven K. Reinhardt. 2001. Filtering Superfluous Prefetches Using Density Vectors. In *ICCD '01: Proceedings of the International Conference on Computer Design: VLSI in Computers & Processors*. 124–133.

[8] Chi F. Chen, Se-Hyun Yang, Babak Falsafi, and Andreas Moshovos. 2004. Accurate and Complexity-Effective Spatial Pattern Prediction. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture (HPCA '04)*. 276–288.

[9] Trishul M. Chilimbi. 2001. Efficient Representations and Abstractions for Quantifying and Exploiting Data Reference Locality. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 191–202.

[10] Yuan Chou. 2007. Low-Cost Epoch-Based Correlation Prefetching for Commercial Applications. In *MICRO*. 301–313.

[11] Jamison Collins, Suleyman Sair, Brad Calder, and Dean M. Tullsen. 2002. Pointer Cache Assisted Prefetching. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 35)*. 62–73.

[12] Robert Cooksey, Stephan Jourdan, and Dirk Grunwald. 2002. A stateless, content-directed data prefetching mechanism. *SIGARCH Computer Architecture News* 30, 5 (October 2002), 279–290.

[13] Arjun Deb, Paolo Faraboschi, Ali Shafiee, Naveen Muralimanohar, Rajeev Balasubramonian, and Robert Schreiber. 2016. Enabling technologies for memory compression: Metadata, mapping, and prediction. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*. IEEE, 17–24.

[14] Eiman Ebrahimi, Onur Mutlu, and Yale N. Patt. 2009. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In *HPCA*. 7–17.

[15] Keith I. Farkas, Paul Chow, Norman P. Jouppi, and Zvonko Vranesic. 1997. Memory-system Design Considerations for Dynamically-scheduled Processors. In *ISCA '97: Proceedings of the 24th Annual International Symposium on Computer Architecture*. 133–143.

[16] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. 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*. 37–48.

[17] Michael Ferdman, Thomas F Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2008. Temporal instruction fetch streaming. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 1–10.

[18] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News* 34, 4 (September 2006), 1–17. https://doi.org/10.1145/1186736.1186737

[19] Seokin Hong, Prashant Jayaprakash Nair, Bulent Abali, Alper Buyuktosunoglu, Kyu-Hyoun Kim, and Michael Healy. 2018. Attache: Towards ideal memory compression by mitigating metadata bandwidth overheads. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 326–338.

[20] Zhigang Hu, Margaret Martonosi, and Stefanos Kaxiras. 2003. TCP: Tag Correlating Prefetchers. In *HPCA*. 317–326.

[21] Ibrahim Hur and Calvin Lin. 2006. Memory Prefetching Using Adaptive Stream Detection. In *Proceedings of the 39th International Symposium on Microarchitecture*. 397–408.

[22] Yasuo Ishii, Mary Inaba, and Kei Hiraki. 2011. Access Map Pattern Matching for High Performance Data Cache Prefetch. In *Journal of Instruction-Level Parallelism*, Vol. 13. 1–24.

[23] Bruce Jacob, Spencer Ng, and David Wang. 2010. *Memory systems: cache, DRAM, disk*. Morgan Kaufmann.

[24] Akanksha Jain and Calvin Lin. 2013. Linearizing Irregular Memory Accesses for Improved Correlated Prefetching. In *46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[25] Akanksha Jain and Calvin Lin. 2016. Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

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

[27] Doug Joseph and Dirk Grunwald. 1997. Prefetching Using Markov Predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*. 252–263.

[28] 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 (ISCA)*. 364–373.

[29] Jinchun Kim, Elvira Teran, Paul V Gratz, Daniel A Jiménez, Seth H Pugsley, and Chris Wilkerson. 2017. Kill the Program Counter: Reconstructing Program Behavior in the Processor Cache Hierarchy. In *Proceedings of the Twenty-Second Int'Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 737–749.

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

[31] Snehasish Kumar, Hongzhou Zhao, Arrvindh Shriraman, Eric Matthews, Sandhya Dwarkadas, and Lesley Shannon. 2012. Amoeba-Cache: Adaptive Blocks for Eliminating Waste in the Memory Hierarchy. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 376–388.

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

[33] Kyle J. Nesbit and James E. Smith. 2005. Data Cache Prefetching Using a Global History Buffer. *IEEE Micro* 25, 1 (2005), 90–97.

[34] Anant Vithal Nori, Jayesh Gaur, Siddharth Rai, Sreenivas Subramoney, and Hong Wang. 2018. Criticality aware tiered cache hierarchy: a fundamental relook at multi-level cache hierarchies. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 96–109.

[35] Subbarao Palacharla and Richard E. Kessler. 1994. Evaluating Stream Buffers as a Secondary Cache Replacement. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 24–33.

[36] Leeor Peled, Shie Mannor, Uri Weiser, and Yoav Etsion. 2015. Semantic locality and context-based prefetching using reinforcement learning. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 285–297.

[37] 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 *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*. IEEE.

[38] Amir Roth and Gurindar S. Sohi. 1999. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA)*. 111–121.

[39] Suleyman Sair, Timothy Sherwood, and Brad Calder. 2003. A decoupled predictor-directed stream prefetching architecture. *IEEE Trans. Comput.* 52, 3 (March 2003), 260–276.

[40] Amna Shahab, Mingcan Zhu, Artemiy Margaritov, and Boris Grot. 2018. Farewell My Shared LLC! A Case for Private Die-Stacked DRAM Caches for Servers. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 559–572.

[41] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically characterizing large scale program behavior. *ACM SIGOPS Operating Systems Review* 36, 5 (2002), 45–57.

[42] A.J. Smith. 1978. Sequential Program Prefetching in Memory Hierarchies. *IEEE Trans. Comput.* 11, 12 (December 1978), 7–12.

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

[44] Stephen Somogyi, Thomas F. Wenisch, Anastassia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2006. Spatial Memory Streaming. In *ISCA '06: Proceedings of the 33th Annual International Symposium on Computer Architecture*. 252–263.

[45] Thomas F. Wenisch, Michael Ferdman, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2009. Practical off-chip meta-data for temporal memory streaming. In *HPCA*. 79–90.

[46] Thomas F. Wenisch, Michael Ferdman, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2010. Making Address-Correlated Prefetching Practical. *IEEE Micro* 30, 1 (2010), 50–59.

[47] Hao Wu, Krishnendra Nathella, Akanksha Jain, Dam Sunwoo, and Calvin Lin. 2019. Efficient Metadata Management for Irregular Data Prefetching. In *the 46th International Symposium on Computer Architecture (ISCA)*.

[48] Vinson Young, Sanjay Kariyappa, and Moinuddin Qureshi. 2019. Enabling Transparent Memory-Compression for Commodity Memory Systems. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 570–581.