The Dissertation Committee for Hao Wu
certifies that this is the approved version of the following dissertation:

# Practical Irregular Prefetching

Committee:

Calvin Lin, Supervisor

Don Fussell

Akanksha Jain

Christopher J. Rossbach

Dam Sunwoo

# Practical Irregular Prefetching

**by**

**Hao Wu**

**DISSERTATION**

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

**DOCTOR OF PHILOSOPHY**

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2020

Dedicated to my grandmother Liang, Yanling.

# Acknowledgments

I wish to thank the multitudes of people who helped me. This dissertation could not finish without the help of these people.

I would like to thank my advisor Calvin Lin. He is always supportive to all of his students. He always gives useful and insightful feedback to me during our discussion. He is creative and makes right direction suggestions when necessary. His feedback on writing is always to the point, which helps me a lot with my writing skills. He always support all his students financially and leaves a friendly and productive environment for research.

Akanksha Jain helps me a lot on my research. She gave me a lot of advice on research directions. Many of the work were initially based on her work, and she helped me a lot on how to interpret and improve those work. She also helped me a lot and gave me advice with writing. The dissertation cannot be finished without her help.

My intern supervisor Krishnendra Nathella and Dam Sunwoo helped me a lot on getting used to Arm's infrastructure. They also provided valuable advice on some research topics and executed a lot of the experiments that are present in our papers.

My committee members are: Calvin Lin, Akanksha Jain, Don Fussell, Chris Rossbach and Dam Sunwoo. They spent their valuable time to read my

dissertation and attend my proposal and final defense. They asked valuable questions on my work and provided great feedback.

I would like to thank other people from Arm Inc. Joseph Pusdesris comes with the idea of partitioning the LLC in the Triage paper. Jaekyu Lee developed the script that creates many of the graphs in our papers. I appreciate the unnamed developers at Arm who developed the simulation infrastructure that was used in our papers.

I am thankful to our group members and other students from UT Austin, including Matthew, Zhan, Jia, Oswaldo, Curtis, Chirag, and Molly. Their support and feedback were very helpful.

Last but not least, I appreciate the support from my parents and grandparents. They have always been supportive and encouraging during my difficult times. They initiated me for pursing my PhD degree.

# Practical Irregular Prefetching

by

Hao Wu, Ph.D.
The University of Texas at Austin, 2020

Supervisor: Calvin Lin

Memory accesses continue to be a performance bottleneck for many programs, and prefetching is an effective and widely used method for alleviating the memory bottleneck. However, prefetching can be difficult for irregular workloads, which the hardware has no clear patterns like sequential or strided patterns.

For irregular workloads, one promising approach is to perform temporal prefetching, which memorizes temporal correlations that happen in the past and use them to predict future memory accesses. To store these correlations, it requires megabytes of metadata which cannot be feasibly stored on-chip. As a result, previous temporal prefetchers store metadata off-chip in DRAM, which introduces hardware implementation difficulties, increases DRAM latencies and increases DRAM traffic overhead. For example, the STMS prefetcher proposed by Wenisch et al. has 3.42× DRAM traffic overhead for irregular SPEC2006 workloads. These problems make previous temporal prefetchers impractical to implement in commercial hardware.

In this thesis, we propose three methods to alleviate the metadata storage problems in temporal prefetching and make it practical in hardware.

First, we propose MISB, a new scheme that uses a metadata prefetcher to manage on-chip metadata. With only 1/5 traffic overhead compared to STMS, MISB achieves 22.7% performance speedup over a baseline with no prefetching compared to 10.6% for an idealized STMS and 4.5% for a realistic ISB.

Second, we present Triage, the first temporal prefetcher that stores its entire metadata on chip, which reduces hardware complexity and DRAM traffic by re-purposing part of last level cache to store metadata. Triage reduces 60% traffic compared to MISB and achieves 13.9% performance speedup over a baseline with no prefetching. In a bandwidth constrained 8-core environment, Triage has 11.4% speedup compared to 8.0% for MISB.

Third, we present a new resource management scheme for Triage's on-chip metadata. This scheme integrates ISP's compressed metadata representation and makes several improvements. For irregular benchmarks, this scheme reduces on-chip metadata storage requirement by 38% and achieves 29.6% speedup compared to Triage's 25.3%.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

## 1.1 The Problem

As Hennessy and Patterson point out [12], memory is slower than processors, and it often takes 150 to 300 CPU cycles to process a DRAM access. Figure 1.1 shows the performance gap between processors and memory in the past 40 years. This gap between processor and DRAM speed affects CPU performance significantly. For example, as Ayers et al. point out [1], more than 40% of total performance potential in a Google's web search binary is wasted on waiting for memory accesses.

An effective method to bridge this gap is data prefetching. By predicting future memory accesses, hardware prefetchers bring in data before they are actually used and hide the latency for accessing these memory locations.

The majority of prefetchers exploit regular access patterns such as sequential or strided patterns. These prefetchers rely on relations among accesses in their address space. As a result they are not capable of prefetching data structures with irregular access patterns such as linked lists or trees. Figure 1.2 shows the difference between regular and irregular access patterns. In general, irregular access patterns are hard to prefetch because there are no

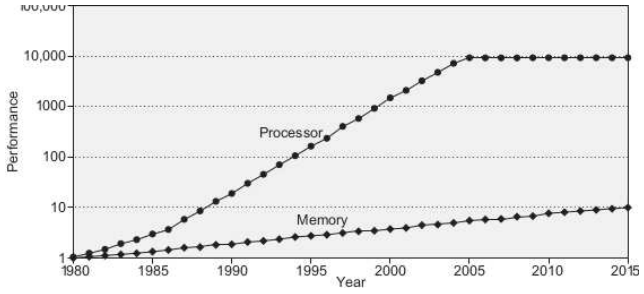



Figure 1.1: The gap between processor and memory performance, measured as time spent. Borrowed from *Computer Architecture: A Quantitative Approach 6th edition* by Hennessy and Patterson [12].

clear patterns between consecutive accesses.

A powerful method of prefetching such irregular access patterns is temporal prefetching. Temporal prefetchers memorize temporal correlations that happen in the past and use them to predict future memory accesses. For example, in the bottom graph of Figure 1.2, we observe the access pattern $A, D, C, B, E$. Next time when $A$ comes in, a temporal prefetcher can predict that $D, C, B, E$ are likely to be accessed in the near future.

Despite their ability to prefetch patterns that other prefetchers cannot, temporal prefetchers suffer from the problem of excessive amount of storage to record past access correlations. The storage size is usually too large to fit on-chip. As a result, previous prefetchers often store them in the DRAM and only fetch them on-chip when necessary. These mechanisms result in excessive amount of latency and traffic overhead and introduce hardware implementation difficulties. For example, the STMS prefetcher proposed by Wenisch [40] has 3.42× DRAM traffic overhead for irregular benchmarks.

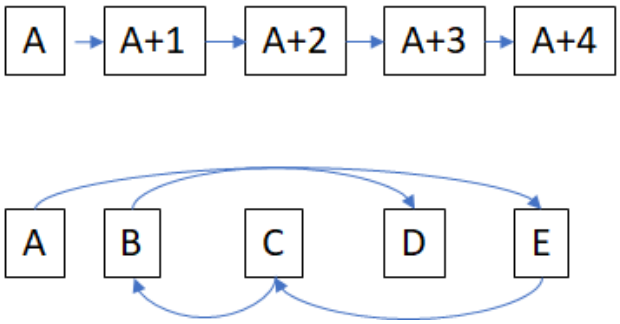



Figure 1.2: This graph shows regular and irregular patterns. Regular access patterns are fixed in their address space (e.g. a fixed delta between consecutive accesses) while irregular access patterns don't observe it.

To make irregular prefetching practical in hardware, it is critical to tackle the problem of having excessive amount of storage. The Irregular Stream Buffer [16] (ISB) introduces a partial solution to this problem by adding an on-chip metadata cache for irregular prefetching. ISB manages this metadata cache by synchronizing with TLB.

Unfortunately, ISB is still inefficient and costs too much hardware resource. ISB suffers from three problems. First, ISB has an excessive amount of traffic going off-chip to DRAM. Experiments show that ISB has 1.5 to 5 times traffic overhead on average. This traffic overhead adverse impacts both performance and energy consumption. Second, ISB, similar to any off-chip based temporal prefetchers, requires complicated management of metadata in DRAM which is not always viable. In certain hardware designs, simpler design without off-chip metadata is desired even with a slight loss of performance. Finally, ISB only addresses benchmarks with irregular access patterns. For benchmarks with both regular and irregular patterns it does not provide a

nice solution.

In this thesis, we address these three problems and propose three solutions for them. Our solution is the first commercially viable temporal prefetcher.

## 1.2 Our Solution

First, we present the Managed ISB(MISB) [43], a temporal prefetcher that utilizes a new metadata management and improves both performance and DRAM traffic overhead. MISB achieves 22.7% performance speedup over a baseline that has no prefetchers, compared to 10.6% of STMS [40] and 4.5% of ISB. MISB has 70% traffic overhead one fifth of STMS (342%) and one sixth of ISB (411%).

Second, we present Triage [42], the first on-chip only irregular temporal prefetcher for this purpose. It reduces complexity of hardware design and amount of DRAM traffic by removing off-chip metadata storage. It reduces performance speedup from MISB by 4.4% on single-core with reduction of traffic of 60%. This traffic reduction helps multi-core since DRAM bandwidth is more constraint in multi-core environment. In an 8-core system Triage achieves 11.4% speedup while MISB only achieves 8.0%

Third, we present Reeses, a new mechanism to manage on-chip metadata for Triage. This new mechanism reduces on-chip metadata storage requirement by 38% and has 4.3% more performance over Triage.

The rest of the thesis is organized as follows. Chapter 2 talks about

background of prefetching. Chapter 3 describes our metadata management scheme of MISB. Chapter 4 describes our on-chip only prefetcher of Triage. Chapter 5 describes our on-chip metadata scheme. Chapter 6 concludes the thesis and presents future work.

# Chapter 2

# Background

## 2.1 Temporal Prefetcher

Temporal prefetchers predict memory accesses by memorizing pairs of accesses that are correlated with each other in temporal order. For example, if the access stream $A, B, C, X, Y$ is memorized, when $A$ arrives, the prefetcher can prefetch $B$. Charney et al proposes correlation-based prefetcher [6] [7] and the term "temporal prefetcher" is introduced by Wenisch et al [41].

Markov prefetcher by Joseph et al. [18] is a simple and effective temporal prefetcher. However, the storage size of Markov prefetcher limits its performance. Nesbit et al propose Global History Buffer (GHB) prefetcher [27] that utilizes global address correlation, which is widely used in most followup works. Figure 2.1 shows an example of how GHB organizes its metadata.

Since the metadata of GHB is merely a large FIFO buffer whose values have a huge reuse distance, it is very difficult to cache the metadata and no previous prefetchers have successfully cached them on-chip.

Jain et al propose Irregular Stream Buffer (ISB) prefetcher [16] provides a solution that caches part of the metadata on-chip. It correlates physical addresses to consecutive structural addresses. Figure 2.1 shows an example of

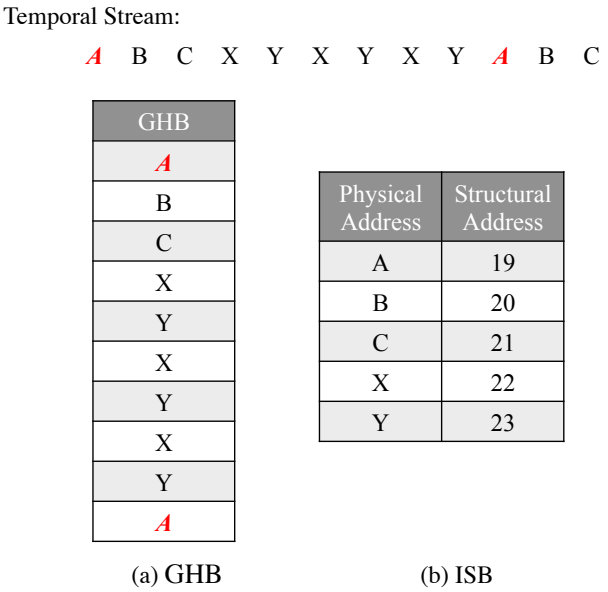



Figure 2.1: Metadata for GHB and ISB [43]

how ISB organizes its metadata and uses it to do prefetch.

The metadata of ISB is stored in two mappings: physical-to-structural(PS) mapping and structural-to-physical (SP) mapping. During training, when access stream $A, B, C, X, Y$ is observed, ISB assigns consecutive structural addresses $19, 20, 21, 22, 23$ to it. The next time $A$ is accessed, ISB searches PS mapping for the structural address for $A$ (19), predicts the next structural address (20), and searches SP mapping for the physical address to prefetch ($B$).

There are two benefits for ISB's metadata storage. First, this approach allows ISB to do PC-localization, which improves accuracy and coverage of prefetching. PC-localization is a technique that separates streams by their

program counter (PC). Second, this approach allows ISB to cache part of its metadata on-chip. Unlike GHB, ISB does not have redundancy in its physical address space, making its metadata size much smaller (about 6× reduction) than GHB.

ISB synchronizes on-chip metadata storage with only addresses that are also located in TLB. The reasoning for this approach is that the latency for fetching the entries not in the TLB is covered by page walk and does not lead to extra latency on the critical path. Since traditional systems often possess a small TLB, ISB can keep the entire TLB resident metadata on chip. For example, for a system with 128 entries of 4KB pages, there are only maximum of 512KB of active memory footprint at one time and requires only 32KB of on chip storage.

As a result, the amount of metadata required on-chip is proportional to $pagesize$ and $TLBsize$. As a result, TLB synchronization has three deficiencies.

First, TLB synchronization manages metadata at page line granularity and creates a lot of useless traffic. Since there is little spatial locality among temporal streams, 90% of the metadata brought on-chip is not used. Extra traffic due to useless metadata fetches hurt performance.

Second, TLB synchronization does not scale to multiple level TLBs, which are used in modern processors.

Third, TLB synchronization does not scale to huge pages, which are

often used for optimizing workloads with large footprint.

To solve these two issues, we need a method to get the metadata without TLB synchronization. This leads to our first piece of work in this thesis - Managed ISB (MISB). We will discuss it in the next chapter.

## 2.2 Other Prefetchers

Many non-temporal prefetchers have been proposed. In this section we discuss some of them that will be used as comparison in following chapters.

### 2.2.1 Regular Prefetcher

Next line prefetcher [36] is one of the earliest prefetchers which always prefetch the next cache line of the current access. Stream buffer prefetcher [19] can detect streams before performing prefetching. Sandbox prefetcher [32] uses sandboxes to determine the degree for prefetching. Best Offset prefetcher [26] evaluate different offsets to determine the best offset for prefetching. Signature Path prefetcher [22] uses previous access deltas (signatures) to determine the next prefetch.

All these prefetchers try to find strides that have the best performance. They work well for strided or streaming access patterns. For irregular access patterns they have little to none performance.

### 2.2.2 Spatial Prefetcher

Spatial Memory Streaming (SMS) [38] records access patterns within a spatial region and uses this information for future prefetching. This scheme works well when access patterns within a spatial region is similar across different regions. Typical workloads that benefits from this prefetcher include database programs and web workloads.

### 2.2.3 Pointer Prefetcher

Pointer prefetchers are prefetchers that exploit relationship between pointers and their destinations. There are two types of pointer prefetchers: compiler-based and hardware based. Compiler-based pointer prefetchers [25, 34] insert prefetch instructions based on programming properties. Hardware-based pointer prefetchers like CDP [8] guess pointer relationship during execution and performs prefetching.

As Jain and Lin [16] point out, there are several deficiencies for pointer prefetchers compared to temporal prefetchers. First, pointer prefetcher only exploit pointer structure while temporal prefetcher exploit other sources of irregular access patterns as well. Second, software-based pointer prefetchers have poor timeliness. Third, hardware-based pointer prefetchers have low prediction accuracy due to its inability to precisely identify pointer instances.

# Chapter 3

# MISB[1]

As described in chapter 2, Irregular Stream Buffer (ISB) prefetcher [16] provides a solution that caches part of the metadata on-chip for temporal prefetchers by synchronizing on-chip metadata with TLB. This approach leads to three deficiencies:

- It fetches large amounts of useless metadata (90% of its loaded metadata is never used), which leads to poor metadata cache efficiency (30% hit rate) and high metadata traffic overhead (411% traffic overhead for irregular SPEC 2006 workloads).
- It does not scale to large pages since the size of the on-chip cache is proportional to the page size. Large pages are important for many programs that have large memory footprints.
- It does not work for modern two-level TLBs. If the metadata cache is synchronized with the L1 TLB, the latency of L2 TLB hits is too short to hide the latency of off-chip metadata requests. If the metadata

[1]Portions of this chapter have been published in ISCA 2019 [43]

cache is synchronized with the L2 TLB, the metadata cache would be impractically large—on the order of 200-400KB.

We propose Managed ISB (MISB) to solve these three issues introduced due to its TLB synchronization mechanism. Instead of synchronzing with TLB, MISB uses a metadata prefetcher to load useful metadata on-chip and uses LRU replacement policy to evict lines from on-chip metadta cache.

There are three key insights behind our metadata management scheme. First, TLB-based cache management is wasteful because metadata for a physical page, which includes 64 consecutive physical-to-structural mappings, typically exhibits poor spatial locality, yielding metadata cache utilization of about 10%. Thus, metadata should be cached at a finer granularity. Second, because structural addresses are temporally ordered, the structural address space has precisely the information that is needed to fill the metadata cache with useful entries ahead of time. Thus, we prefetch metadata entries by using next-line prefetching for structural-to-physical mappings. Third, many off-chip metadata requests are to addresses that have not been seen before, so no metadata exists for these addresses. Thus, we use a Bloom filter [3] to record the physical addresses that have associated metadata; by checking the Bloom filter before issuing metadata prefetches, we dramatically reduce the number of metadata requests for unmapped physical addresses.

Our results are magnificent. We reduced off-chip traffic overhead for metadata to 70% from ISB's 411%, a reduction of 5.9×. For irregular SPEC2006

benchmarks, MISB achieves 22.7% speedup compared to 4.5% for ISB and 10.6% for idealized STMS. MISB also scales to large pages and 2-level TLBs. For example, MISB achieves 25.5% speedup on a system with 2MB pages while ISB has no speedup for them.

## 3.1 Our Solution

MISB is composed of three components: (1) a metadata cache, (2) a metadata prefetcher, and (3) a metadata filter that avoids issuing spurious metadata requests. These three components together allow MISB to judiciously manage metadata. We will describe each component in more detail.

**Metadata Caching** The on-chip metadata cache helps reducing latency and traffic from accessing off-chip. As mentioned in chapter 2, we require translations for both structural to physical and physical to structural to work, so we have separate structure for PS cache and SP cache.

Upon receiving a memory access, the training unit records consecutive memory accesses from the same PC, and assigns correct structural address for a given physical address. During prediction, MISB accesses both PS cache and SP cache for prediction.

We cache both PS and SP cache off-chip. Unlike ISB, MISB manages its metadata at a fine granularity in cache line level. A logical metadata cache line contains 8 entries of mapping. Each time MISB does not find a metadata cache line on-chip, it goes to off-chip metadata for that cache line, and pops

them to on-chip metadata. MISB uses LRU replacement policy to decide which cache line to be evicted from on-chip metadata.

**Metadata Prefetching** If we only have simple metadata caching scheme as mentioned above, MISB cannot efficiently issues prefetches because the latency to fetch metadata off-chip adversely affect prefetching performance. To solve this, MISB introduces a metadata prefetcher.

Since structural address space has temporal locality. We can simply utilize a next line prefetcher for prefetching metadata. That is to say, if we are accessing structural address 71, we can bring in 72 and it will be used in the near future.

**Metadata Filtering** MISB accesses off-chip metadata for each missing on-chip metadata entry. If an entry does not exist off-chip, this access is futile and results in useless traffic to DRAM. To reduce this traffic, we introduce a Bloom filter for off-chip accesses. Each off-chip entry is recorded in the Bloom filter. Before MISB accesses off-chip, it first checks the Bloom filter and squashes all accesses off-chip if they do not exist in the Bloom filter.

### 3.1.1 Overall Operation

Figure 3.1 shows the overall design of MISB. The Training Unit finds correlated addresses within a PC-localized stream and assigns them consecutive structural addresses. On-chip mappings are stored in the PS and the

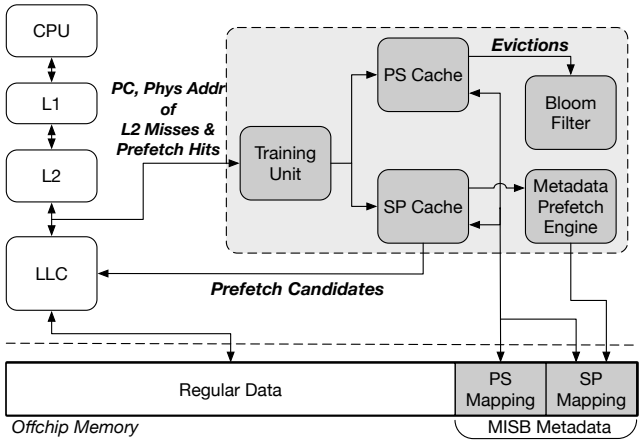



Figure 3.1: Overview of MISB

SP caches, and on eviction, they are written to the corresponding off-chip PS and SP tables. The Bloom filter tracks valid PS mappings and filters invalid off-chip requests for PS mappings.

Conceptually, MISB's overall training and prediction algorithms are similar to ISB's, but they differ significantly in the way that MISB manages the movement of metadata between the on-chip metadata caches and off-chip metadata storage. We now describe MISB's metadata management scheme and its interactions with ISB's training and prediction algorithms.

**Prediction** On prediction, MISB first queries the on-chip PS cache with the trigger physical address. If the PS request hits, MISB predicts prefetch requests for the next few structural addresses. If the PS request misses, MISB issues an off-chip *PS load* request, delaying the prediction until the request completes. For example, in Figure 3.2-❶, the trigger address $M$ misses in the

PS cache and initiates a *PS load* for $M$. When the request completes, the new mappings are inserted into both the PS and SP caches, as indicated by the shaded entries in Figure 3.2-❷.

Regardless of whether the *PS load* hits or misses in the cache, when we find its structural address $s$, we issue a data prefetch request for structural address $s + 1$, which causes MISB to query the on-chip SP mapping for $s + 1$ (structural address 1024 in Figure 3.2-❸). If found, a data prefetch for the corresponding physical address is issued. If not found, the predicted structural address $s + 1$ is placed in a small (32-entry) buffer and an *SP1 load* request is issued to off-chip memory. At the same time, future requests to structural addresses $s + 2$, $s + 3$ and so on are anticipated with the issuance of *SP2 prefetch* requests. For example, in Figure 3.2-❸, an *SP2 prefetch* request is issued for structural address 1025, assuming a metadata prefetch degree of 1. The degree of metadata prefetching can be tuned, and like PS requests, each SP request carries mappings for 8 consecutive structural addresses. SP requests also fill both the PS and SP caches. In Section 3.2.4, we show that our metadata prefetching scheme improves hit rates for both PS and SP caches.

**Training** MISB's training is similar to ISB's training algorithm. The Training Unit keeps track of consecutive memory references for a given PC and assigns PC-localized correlated addresses consecutive structural addresses. The on-chip PS and SP caches are updated with newly assigned mappings, and if mappings change, they are marked dirty so that they can be written to off-chip

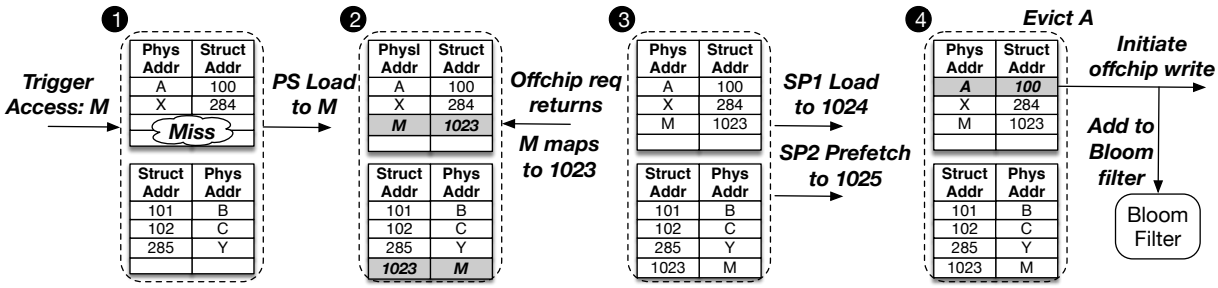



Figure 3.2: MISB: PS and SP Transactions.

memory.

**Metadata Organization** As shown in Figure 3.2, MISB's on-chip metadata caches are organized at a fine granularity, where each cache entry holds one mapping (8 bytes). A fine-grained organization ensures better metadata cache efficiency because individual mappings can be retained and evicted based on their usefulness. For example, if one portion of the stream is more likely to be reused than another, then our metadata cache can selectively retain mappings for the first portion and discard mappings for the second portion. To maximize off-chip bandwidth utilization, off-chip requests are issued at the granularity of 64 bytes (or 8 mappings). Unless specified otherwise, our metadata caches are 8-way set-associative.

Unlike ISB, both the PS caches and SP caches are managed using an LRU policy, which allows the PS and SP caches to retain the mappings that see the most utility in the respective caches. For example, in Figure 3.2, for the stream A, B, C, the PS Cache has the physical to structural mapping for

A, and the SP cache has structural to physical mappings for B and C, but not A. Both our PS and SP caches are writeback caches, so dirty evictions in these caches result in an off-chip store request.

Finally, our off-chip storage includes two tables: The PS Table and the SP Table. *PS loads* are served by the PS Table, while *SP1 loads* and *SP2 prefetches* are served by the SP Table. By contrast, since ISB does not need the SP Table for prefetching, ISB maintains just the off-chip PS Table and uses the PS entries to construct the on-chip SP table.

**Bloom Filter** As shown in Figure 3.2-❹, on an off-chip store request from the PS cache, the corresponding store address is added to the Bloom filter to indicate that an off-chip mapping exists for this physical address. The Bloom filter is then probed on future *PS loads*, and the load is issued only if the Bloom filter confirms that a mapping will exist in off-chip memory.

An ideal Bloom filter with infinite resources can eliminate all false positives, but with limited resources, the Bloom filter produces false positives. To reduce false positives, we provision 17KB for the Bloom filter and use the h3 hash [33], which provides a good tradeoff between space efficiency and traffic. For more bandwidth-constrained environments, the Bloom filter budget can be increased to further reduce traffic.

## 3.2 Evaluation

### 3.2.1 Methodology

For single-core configurations, we evaluate MISB using a proprietary cycle-level simulator that is correlated against the RTL of commercially-available CPU designs. This highly accurate and flexible simulator is developed and maintained by a team of engineers. Our generic CPU model implements the ARMv8 AArch64 ISA and uses the configuration shown in Table 3.1. The simulator employs a simple fixed-latency memory model, but it models bandwidth constraints accurately and stalls the execution accordingly. Our small page configuration uses pages sizes of 4KB, and our large page configuration uses a page size of 2MB.

For multi-core configurations, we use ChampSim [11, 23], a trace-based simulator that includes an out-of-order core model with a detailed memory system. ChampSim's cache subsystem includes FIFO read and prefetch queues, with demand requests having higher priority than prefetch and metadata requests. The main memory model simulates data bus contention, bank contention, and bus turnaround delays; bus contention increases memory latency. The main memory read queue is processed out of order and uses a modified Open Row FR-FCFS policy. Our ChampSim simulations use the configuration in Table 3.1 and replicate single-core performance trends from our proprietary simulator[2].

[2]Absolute quantities such as IPC, MPKI, and traffic in GB/s do not match exactly between the two simulators, but the relative differences in these quantities are similar.

| | |
|---|---|
| Core | Out-of-order, 2GHz,<br>4-wide fetch, decode, and dispatch<br>128 ROB entries |
| TLB | 48-entry fully-associative L1 I/D-TLB<br>1024-entry 4-way L2 TLB |
| L1I | 64KB private, 4-way, 3-cycle latency |
| L1D | 64KB private, 4-way, 3-cycle latency<br>Stride prefetcher |
| L2 | 512KB private, 8-way, 7-cycle latency |
| L3 | 2MB per core, shared, 16-way<br>12-cycle latency |
| DRAM | Single-Core:<br>85ns latency, 32GB/s bandwidth<br>Multi-Core:<br>8B channel width, 800MHz,<br>tCAS=20, tRP=20, tRCD=20<br>2 channels, 8 ranks, 8 banks, 32K rows<br>32GB/s bandwidth total |

Table 3.1: Machine Configuration

**Benchmarks** We present single-core results for all memory-bound workloads from SPEC2006 [13]. For detailed analyses, we choose a subset of benchmarks that are known to have irregular access patterns [16]. For SPEC benchmarks we use the reference input set. For all single-core benchmarks, we use SimPoints [35] to find representative regions. Each SimPoint has 30 million instructions, and we generate at most 30 SimPoints for each SPEC benchmark.

We present multi-core results for CloudSuite [9] 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 benchmarks chosen uniformly randomly from all memory-bound benchmarks, and for 8-core results, we choose 8 benchmarks chosen uniformly randomly. Overall, we simulate 80 4-core mixes and 35 8-core mixes. For each mix, we simulate the simultaneous execution of SimPoints of the constituent benchmarks until each benchmark has executed at least 500 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 100 million instructions.

**Prefetchers** We compare MISB against four irregular prefetchers, namely, Spatial Memory Streaming (SMS) [38], Sampled Temporal Memory Streaming (STMS) [40], Irregular Stream Buffer (ISB) [16], and Domino [2]. SMS captures irregular patterns by applying irregular spatial footprints across memory regions. STMS, ISB, and Domino 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. Thus, our performance results for STMS and Domino

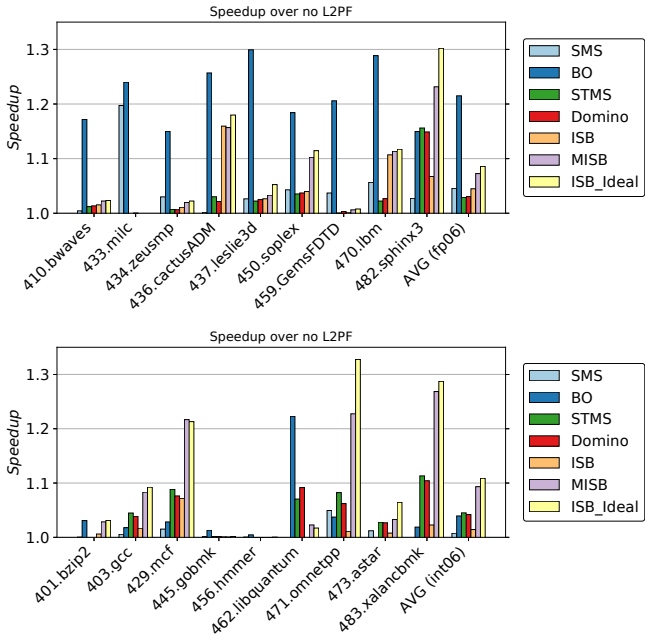



Figure 3.3: Comparison of Prefetchers on SPECfp 2006 (left) and SPECint 2006 (right).

represent the upper bound of performance for these prefetchers. To estimate their traffic overhead, we count the number of metadata requests, but the requests are never issued to the memory system. **Throughout our evaluation, references to STMS and Domino refer to these idealized implementations.** We also try variants of STMS and Domino that cache the index table in a 32 KB on-chip cache and probabilistically update the off-chip index table [40]. These implementations also do not incur any latency and traffic penalty and are meant to evaluate the impact of probabilistic metadata update on traffic and performance.

For ISB and MISB, we faithfully model the latency and traffic of all metadata requests. For MISB, we use 49KB of on-chip storage, which contains 32KB for the on-chip metadata cache and 17KB for the Bloom filter. We also compare against an idealized version of ISB which has access to all the metadata instantaneously, thereby representing an upper bound of performance for ISB and MISB.

Unless otherwise specified, all prefetchers train on the L2 access stream, and prefetches are inserted into the L2 cache. Unless otherwise specified, all prefetchers use a prefetch degree of 1, which means that they issue at most one prefetch on every trigger access.

We also evaluate MISB as the irregular component of a hybrid prefetcher that uses the Best Offset Prefetcher (BO) [26] as the regular prefetcher. We choose BO because it won the Second Data Prefetching Championship [31].

### 3.2.2 Single-Core Results

Figure 3.3 compares all the prefetchers on memory-intensive benchmarks from SPECfp (left) and SPECint (right). On SPECint, which mostly consists of challenging irregular benchmarks, MISB outperforms all prefetchers with a speedup of 9.3% vs. 4.5% for STMS, the second best prefetcher. On SPECfp, which mostly consists of regular benchmarks, BO outperforms all temporal prefetchers with an overall speedup of 21.5%. Temporal prefetchers do not perform well on regular benchmarks because they cannot prefetch compulsory misses, but we show later in this section that temporal prefetchers combine well with regular prefetchers. Because the benefit of temporal prefetching is most pronounced for irregular benchmarks, the rest of this section focuses on a subset of 7 irregular benchmarks (5 from SPECint and 2 from SPECfp).

**Irregular SPEC2006** The top graph of Figure 3.4 shows that for the irregular SPEC2006 benchmarks, MISB outperforms all other prefetchers. Its 22.7% speedup comes close to the 26.9% speedup of an idealized ISB that incurs no metadata overhead. By contrast, realistic ISB achieves a 4.5% speedup, which illustrates the severe limitations of ISB's metadata management scheme on a modern system with a 2-level TLB. MISB also outperforms idealized versions of STMS (10.6% speedup) and Domino (9.5% speedup), which illustrates its benefits over unrealistically optimistic versions of GHB-based temporal prefetchers. Regular prefetchers, such as BO and SMS, do not perform well

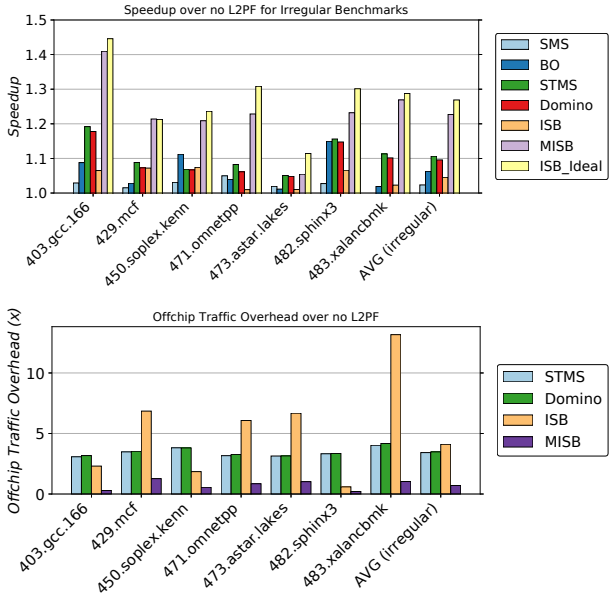



Figure 3.4: Irregular SPEC2006 Results

on irregular benchmarks, achieving only 6.3% and 2.3% speedup, respectively.

The bottom graph of Figure 3.4 shows that MISB's traffic overhead is significantly lower than that of the other prefetchers. In particular, MISB's traffic overhead over a baseline with no prefetching is 70%, while STMS, Domino, and ISB incur five to six times more traffic (342% for STMS, 348% for Domino, and 411% for ISB). The overhead includes traffic due to metadata requests and useless prefetches, but as we will see, ISB and MISB issue very few useless prefetches, so the vast majority of their traffic overhead can be attributed to metadata requests. We expect these traffic savings to translate directly to both energy and power savings.

MISB's traffic overhead can be reduced from 70% to 45% by using it at the L3 cache (train on L3 accesses and prefetch into the L3), but this reduction in traffic comes at the cost of performance, as speedup is reduced from 22.7% to 19.0%.

Figure 3.5 shows that probabilistic update [40] reduces STMS' traffic at the cost of performance. In particular, STMS with probabilistic update reduces STMS' speedup from 10.6% to 5.4%, and it reduces traffic overhead from 342% to 209%, which is still much higher than MISB's traffic overhead of 70%.

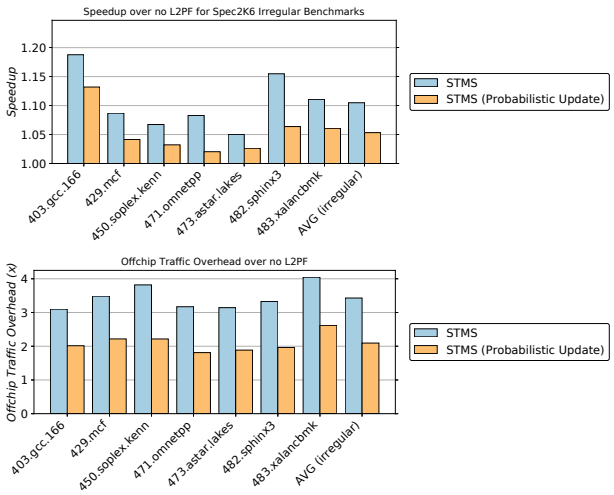



Figure 3.5: Impact of Probabilistic Update on STMS

To summarize MISB's benefits, Figure 3.6 shows that MISB outperforms other temporal prefetchers in nearly all dimensions, including accuracy and timeliness. Like ISB, MISB has high accuracy (87.3% for MISB vs. 64.1% for STMS and 60.9% for Domino) and good timeliness (83.1% for MISB vs.

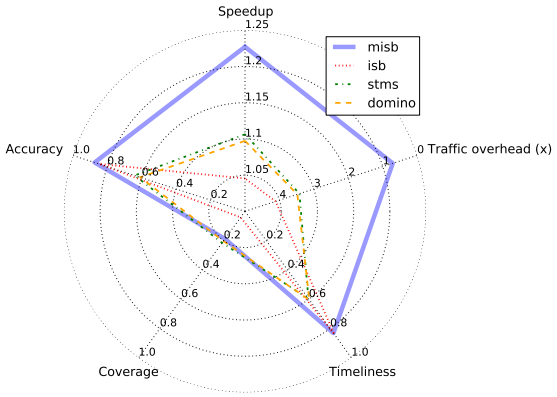



Figure 3.6: MISB Improves Along Multiple Dimensions.

59.6% for STMS and 60.4% for Domino). MISB's 18.8% coverage is slightly lower than that of Domino (20.3%) and STMS (21.5%) because our idealized implementations of Domino and STMS do not incur any latency for accessing off-chip metadata, whereas for MISB, the metadata latency causes a 5.0% loss in coverage. Nevertheless, MISB achieves higher speedup than idealized STMS and Domino because benefits in other dimensions easily compensate for the small loss in coverage.

**Large Page Workloads** Figure 3.7 shows that MISB retains its benefits in the presence of 2MB pages. In particular, MISB achieves 25.5% speedup over no L2 prefetching (vs. 9.1% for BO, 2.7% for SMS, 12.8% for STMS, and 12.0% for Domino). As we would expect, with huge pages, ISB sees only a 0.2% speedup because at 8MB, the metadata for TLB-resident pages is too large for ISB to maintain in its on-chip caches. MISB retains its traffic benefits

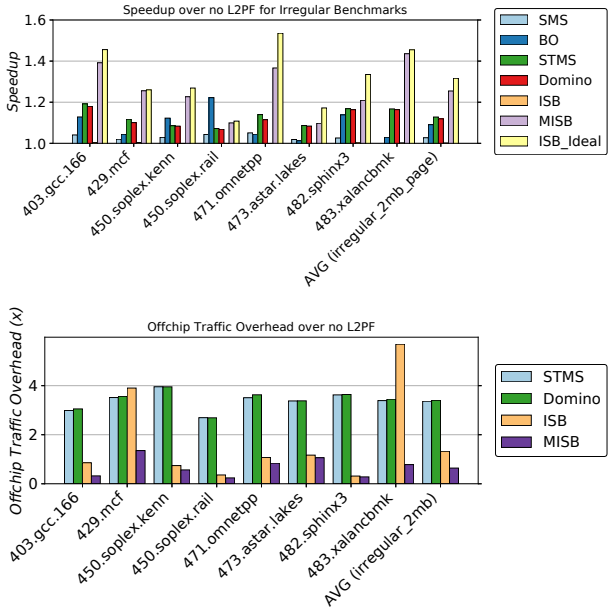



Figure 3.7: Large Page Results

with large pages: Its traffic overhead is 64%, which is much lower than ISB's 132%, Domino's 340%, and STMS' 337%.

**Hybrid Prefetchers** It would be difficult to imagine a chip vendor providing an irregular prefetcher without also providing a regular prefetcher, so we combine each of our temporal prefetchers with BO and SMS. Figure 3.8 shows that for the irregular subset of SPEC2006, the BO-MISB hybrid outperforms other hybrids with a 25.6% speedup (vs. 14.1% for BO-STMS). Since BO alone sees only a 6.3% speedup, we conclude that the remaining performance benefit comes from MISB's ability to prefetch irregular memory access patterns. If we further add SMS to the hybrid prefetcher, the BO-SMS-MISB hybrid achieves

a 26.2% speedup. For SPECfp benchmarks, the BO-MISB hybrid improves performance by 23.9%, a slight improvement over BO alone (21.5% speedup).

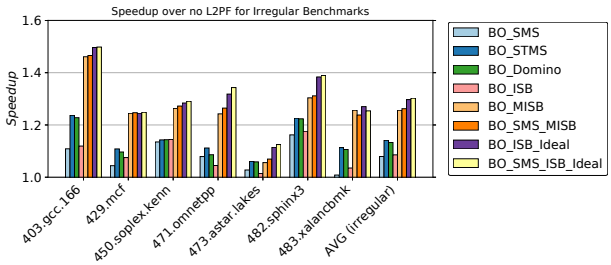



Figure 3.8: Hybrid Results

### 3.2.3 Multi-Core Results

We now evaluate MISB on multi-core configurations.

**CloudSuite Benchmarks** Figure 3.9 shows that a realistic MISB outperforms idealized STMS and idealized Domino on CloudSuite benchmarks, even though the idealized prefetchers incur no latency or traffic penalty for metadata accesses[3]. In particular, MISB improves performance by 7.2%, while idealized STMS and Domino improve performance by 4.0% and 3.9%, respectively. These performance improvements can be explained by MISB's superior coverage (31.0% for MISB vs. 13.6% for STMS and 13.4% for Domino) and accuracy (89.8% for MISB vs. 79.0% for STMS and 77.7% for Domino).

Figure 3.10 shows that MISB's metadata traffic overhead is significantly

[3]For CloudSuite workloads, we train all prefetchers on L2 misses instead of L2 accesses, which results in better IPC and lower traffic for all prefetchers.

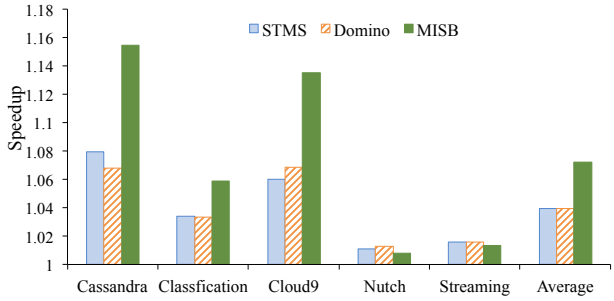



Figure 3.9: Speedup Comparison on CloudSuite

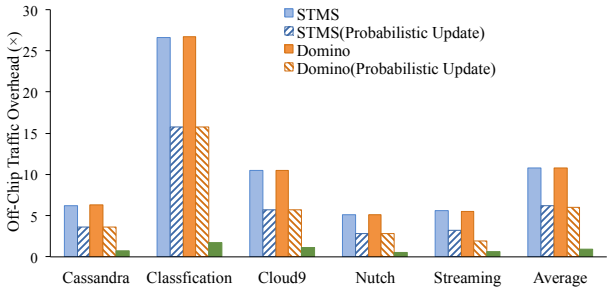



Figure 3.10: Traffic Comparison on CloudSuite

lower than that of STMS and Domino: MISB's traffic overhead is 96.2%, while idealized STMS' and Domino's are 1082.7% and 1081.5%, respectively. The traffic overhead of STMS and Domino can be reduced to 621.6% and 596.9%, respectively, by employing probabilistic updates to the off-chip structures [40], but this optimization degrades performance. For STMS, the performance drops from 4.0% to 2.0%, whereas for Domino, performance drops from 3.9% to 1.8%.

Our results show that contrary to prior claims [2], PC-localization is quite beneficial for server benchmarks. Figure 3.11 compares the compressibility of PC-localized cache access streams to global access streams, showing that

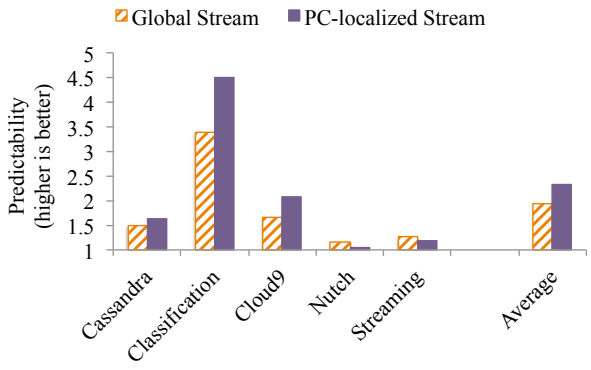



Figure 3.11: Benefits of PC-Localization For CloudSuite

PC-localized streams are more compressible and therefore more predictable than the global stream.[4] These results also explain MISB's higher coverage and accuracy on server workloads. A second concern [2] is that PC-localized predictions are untimely for server workloads because instructions repeat much less frequently than in scientific workloads. Our results show that timeliness is not an issue when prefetching into the L2 or L3 (prior work prefetches into a prefetch buffer that is probed in parallel to the L1 [2]). In fact, at the LLC, MISB is more timely than even idealized STMS and Domino.

**Multi-Programmed SPEC Benchmarks** To avoid aggravating memory pressure, low metadata overhead is critical for scaling the benefits of temporal prefetchers to multi-core systems. MISB works well for 4-core and 8-core systems. On 4-core multi-programmed workloads, a realistic MISB improves

[4]We use the Sequitur algorithm [28] to compute compressibility of global and per-PC streams. Given a sequence of symbols, Sequitur constructs a compressed representation of the sequence by substituting repeating phrases with concise *rules*.

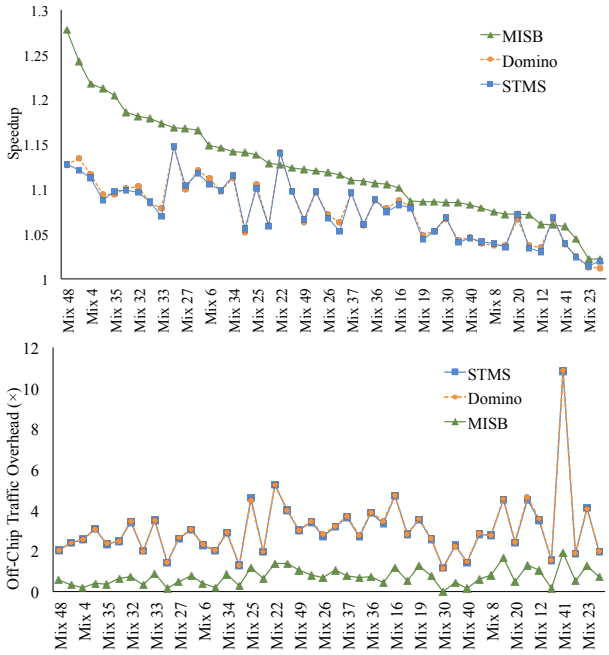



Figure 3.12: MISB Scales to 8-Core Systems.

performance by 19.9%, compared to 8.8% for STMS and 9.6% for Domino. On 8-cores, MISB's benefit reduces to 12.1% because the metadata traffic overhead starts to stress available bandwidth, but the top graph in Figure 3.12 shows that MISB still outperforms idealized versions of STMS (7.5% speedup) and Domino (7.6% speedup) that do not incur performance penalty for metadata traffic. The bottom graph in Figure 3.12 shows that the key to MISB's scalability is its low traffic overhead, which is 72.5%, while idealized STMS and Domino incur 304.7% and 306.8% traffic overhead respectively.

### 3.2.4 Understanding MISB's Benefits

The left graph in Figure 3.13 shows that MISB's benefits depend on its metadata cache and prefetcher working in concert. We make two observations. First, without metadata prefetching, MISB's speedup is reduced from 22.7% to 6.5%; without an adequate metadata cache budget [5], its speedup is reduced from 22.7% to 8.9%. Second, MISB's caching and prefetching scheme can be applied even in the absence of PC-localization, but the loss of PC-localization severely hurts performance, reducing speedup from 22.7% to 7.3%.

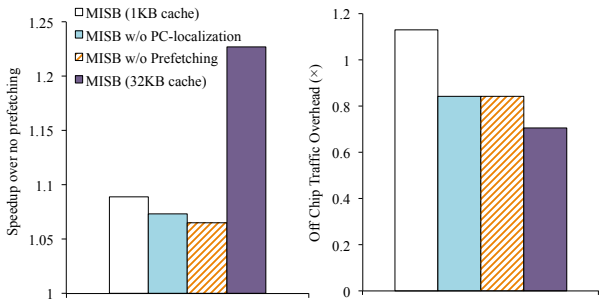



Figure 3.13: MISB Benefits from Both Metadata Caching and Prefetching.

The right graph in Figure 3.13 shows that metadata caching significantly reduces MISB's traffic overhead. If we were to reduce the metadata cache budget from 32KB to 1KB, traffic overhead increases from 70% to 113%.

**Metadata Cache Hit Rates** Figure 3.14 shows that for both the PS and SP caches, MISB's metadata management yields significantly better hit rates

[5]We reduce the metadata cache size from 32KB to 1KB to evaluate the benefit of metadata caching. The specifics of the MISB design require a little bit of on-chip metadata cache to properly train off-chip metadata.

than ISB's (43.0% vs. 27.1% for the PS cache, and 66.5% vs. 32.5% for the SP cache). MISB's improved hit rates are primarily caused by its accurate metadata prefetching. We find that more than 90% of metadata retrieved by ISB's TLB-sync scheme is never used, which both hurts metadata cache efficiency and incurs high traffic overhead.

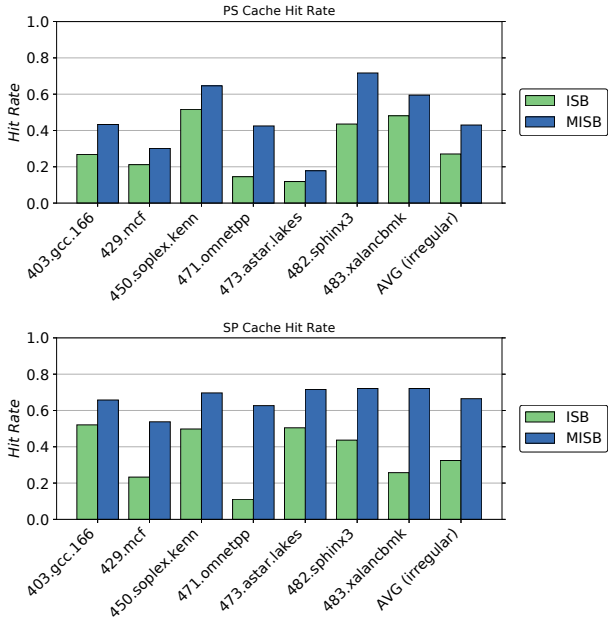



Figure 3.14: On-Chip Metadata Cache Hit Rate.

**Metadata Traffic** Finally, Figure 3.15 shows a breakdown of MISB's off-chip prefetcher traffic. We see that our Bloom filter reduces spurious PS loads by not issuing traffic requests marked in striped blue, resulting in traffic savings of 8.5% (78.5% traffic overhead without the bloom filter vs. 70.0% traffic overhead with the bloom filter). We also see that by reducing the

Bloom filter's false positive rate (unfiltered PS loads), we can further reduce traffic.

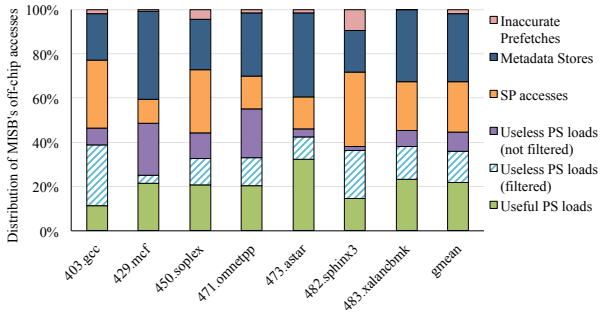



Figure 3.15: Traffic Breakdown for MISB.

## 3.3 Conclusion

MISB improves on ISB by managing metadata efficiently. By utilizing fine-grained caching and accurate data prefetching, MISB improves performance by 22.7% (vs. 4.5% for ISB and 10.6% for idealized STMS), while reducing off-chip traffic to 70% (vs. 411% for ISB and 342% for STMS).

Although MISB significantly reduces traffic overhead and improves performance, it still has an off-chip storage. This off-chip storage can lead to extra resource consumption which could be infeasible in certain chip design. To solve this problem, we propose our next solution, the Triage prefetcher, in the next chapter.

# Chapter 4

# Triage[1]

Metadata is necessary for temporal prefetching. Unfortunately, all previous temporal prefetchers have to store them off-chip, leading to three issues. First, off-chip metadata consumes significant extra energy because DRAM operations consume much more energy than SRAM. Second, extra traffic caused by off-chip metadata accesses can adversely impact performance in bandwidth constraint environments. Third, storing metadata off-chip require extra hardware complexity for chip vendors, including changing memory interface and operating systems.

To solve this issue, we propose a new temporal prefetcher that does not require off-chip metadata. We have two basic insights for this. First, as Figure 4.1 shows, most of the benefits of prefetching come from a small portion of metadata, so it is possible to achieve significant amount of coverage with only a small fraction of the entire metadata. Second, last level cache (LLC) is not efficient enough for irregular workloads, and the benefit of having a larger LLC is often outweighed by using them for prefetcher metadata.

Based on these two insights, our Triage prefetcher reuse a fraction of

[1]Portions of this chapter have been published in MICRO 2019 [42]

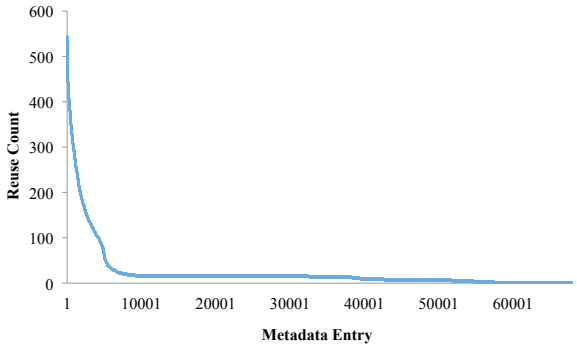



Figure 4.1: Metadata Reuse Distribution for the *mcf* benchmark: For an execution with 60K metadata entries, only 15% of metadata entries are reused more than 15 times.

LLC as metadata storage, and throws away extra metadata that do not fit in LLC. We also introduce a dynamic partitioning mechanism and replacement policy so that the LLC space is used effectively.

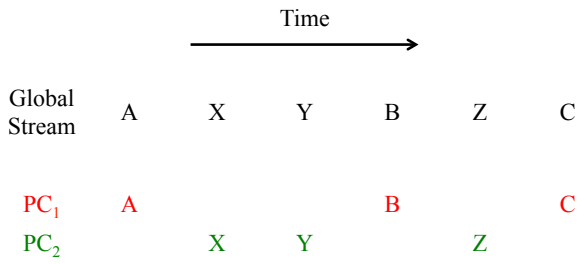
## 4.1 Our Solution

To utilize valuable on-chip cache space effectively and efficiently, the Triage design 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 discuss our solution for each design question in turn.

**Metadata Representation** Triage learns PC-localized correlated address pairs and records them in a tabular format. For example, the top side of Figure 4.2 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 |
| | |

Triage's Metadata Organization

Figure 4.2: Triage's metadata organization.

While tables are a poor choice for organizing off-chip metadata, they are the ideal choice for organizing on-chip metadata because of their space efficiency. In particular, compared to other metadata organizations [16, 40, 43], our table-based organization avoids metadata redundancy by representing each correlated address pair only once. One drawback of our table-based organi-

zation is that higher degree prefetching requires multiple metadata lookups, but this latency 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 up to eight off-chip metadata entries.)

Section 4.1.2 provides more details about how Triage's entries are organized in the LLC and how we use compact address representations to reduce the metadata footprint.

**Metadata Replacement** We build Triage's metadata replacement policy on three observations. First, most metadata reuse can be attributed to a few metadata entries (see Figure 4.1). 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.

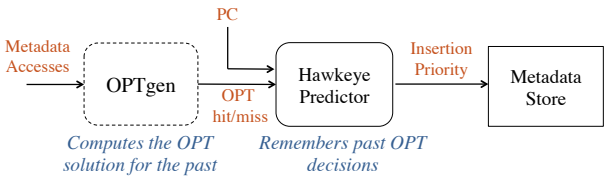



Figure 4.3: Triage's metadata replacement is based on the Hawkeye [17] cache replacement policy.

To accomplish these goals, we modify a state-of-the-art cache replacement policy called Hawkeye [17], which learns from the optimal solution for

past memory references. To emulate the optimal solution 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 4.3 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.

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 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 4.1.2, we provide more details on how we manage the metadata replacement at a fine granularity even though many metadata entries share the same logical last-level cache line.

**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 have different

metadata cache requirements, our solution dynamically determines the number of ways that should be allocated to data and 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 cache sizes, so OPTgen can be used to estimate the profitability of adjusting the amount of cache space devoted 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 needs an additional 1KB space), and we use these copies as sandboxes to evaluate the optimal hit rate at different metadata cache sizes. If we find that increasing the metadata cache size will increase optimal metadata hit rate by more than 5%, we increase the number of ways that are allocated to metadata entries. Similarly, if we find that reducing the metadata cache size decreases the metadata hit rate by less than 5%, we reduce the number of ways allocated to metadata entries. For simplicity, Triage chooses between three possible allocations for metadata cache (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 cache sizes.

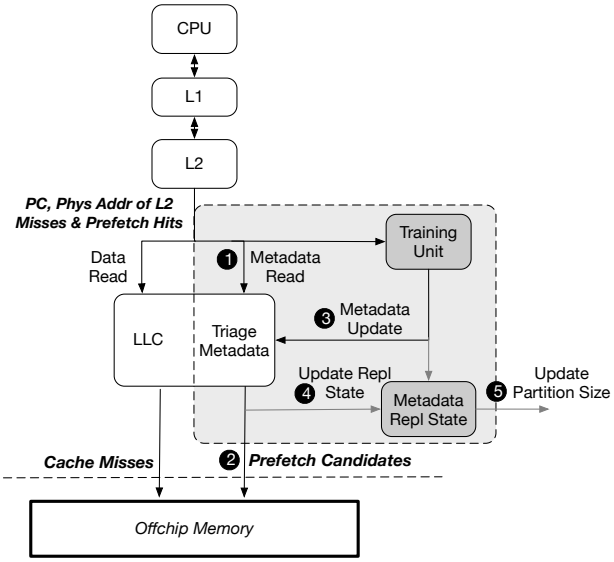



Figure 4.4: Overview of Triage.

### 4.1.1 Overall Operation

Figure 4.4 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. Irrespective of whether the load resulted in a metadata hit or miss, the Training Unit is updated, 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.

**Training** The training unit keeps the most recently accessed address for each PC. When a new access $B$ arrives for a certain 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 is different from what the metadata store currently holds, then the confidence counter is decremented. If the training unit determines that $A$'s neighbor is the same as what the metadata store currently holds, 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 state, including replacement predictors and per-line replacement state, are updated on

(1) metadata misses, and (2) metadata hits that result in a successful prefetch request (a prefetch request that does not hit in the cache and is actually issued to memory. The replacement state is not updated on metadata hits that result in redundant prefetches (prefetch requests that hit in the cache) because such metadata entries are not useful.

Our metadata replacement is based on the Hawkeye policy which looks at a long history of past metadata requests to filter undesirable metadata entries. Much like the Hawkeye policy, the replacement policy is trained on the behavior of a few sampled sets only.

**Metadata Partition Updates** Triage partitions the cache between data and metadata by using way partitioning. In particular, it uses OPTgen hit rates as discussed above to periodically (every 1000 metadata accesses) re-compute the portion of the cache allocated to the metadata store. If Triage decides to increase or decrease the amount of metadata store, dirty lines are flushed and the newly allocated/deallocated portion of the cache is marked invalid immediately.

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, Triage allocates 1.5MB of the shared LLC for metadata and partitions the metadata space in a 2:1 ratio among the two cores.

### 4.1.2 Hardware Design

Triage's metadata entries must be organized at a fine granularity since metadata entries for irregular prefetchers do not exhibit spatial locality. Because the LLC is organized at a much coarser granularity, each metadata entry stores multiple tagged entries within a cache line. In particular, each metadata entry is 4 bytes (described shortly), and we store 16 of them 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 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 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, in each metadata entry, we record the compressed tag of the trigger address and the compressed tag and set_id of the next address, which sums to 31 bits[2]. The remaining 1 bit is used as confidence counter.

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

## 4.2 Evaluation

### 4.2.1 Methodology

We evaluate Triage using similar mechanisms as MISB, which is described in section 3.2

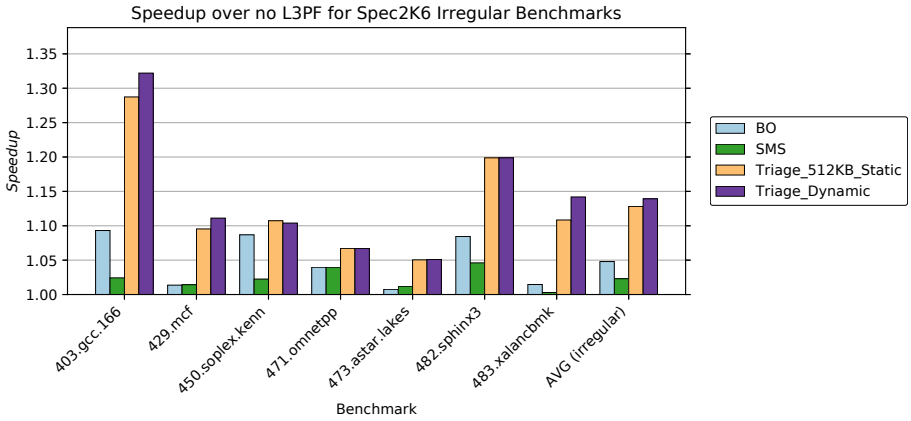



Figure 4.5: Triage outperforms BO and SMS

We evaluate two versions of Triage, static and dynamic. The static version picks a fixed metadata store size that gives the best performance on average and statically partitions the LLC using this size. For our industrial strength simulator, the best static metadata store size for a 2MB LLC is 512KB, and for the ChampSim simulator, the best static metadata store size is 1MB. The dynamic version of Triage modulates the size of the metadata store dynamically as described in Section 4.1.

All prefetchers train on the L2 miss stream, and prefetches are inserted in the last-level cache (L3). Unless specified, all prefetchers use a prefetch

degree of 1, which means that they issue at most one prefetch on every trigger access.

## 4.3 Comparison With Prefetchers That Store Metadata On Chip

Figure 4.5 shows that Triage outperforms state-of-the-art prefetchers that have on-chip metadata only. In particular, Triage achieves a speedup of 12.8% and 13.9% for the static and dynamic configurations, respectively, whereas BO and SMS see a speedup of 4.8% and 2.3%, respectively. Triage's superior performance can be explained by its higher coverage (23% for Triage vs. 16% for BO and 4% for SMS) and higher accuracy (82% for Triage vs. 44% for BO and 41% for SMS) as shown in Figure 4.8.

Triage-Dynamic is slightly better than Triage-Static as it modulates the metadata store size for gcc and xalancbmk. 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 4.7 sheds more insight on Triage's performance. We see that a version of Triage that does not reduce LLC capacity achieves a 20.7% speedup. Reducing the cache by 1 MB results in a 7.4% loss in performance, but we find that this loss is compensated by Triage's high coverage as Triage sees an overall speedup of 12.8% with a 1MB metadata store.

For completeness, Figure 4.6 compares all prefetchers on the remaining memory-intensive SPEC 2006 benchmarks. Because these benchmarks are

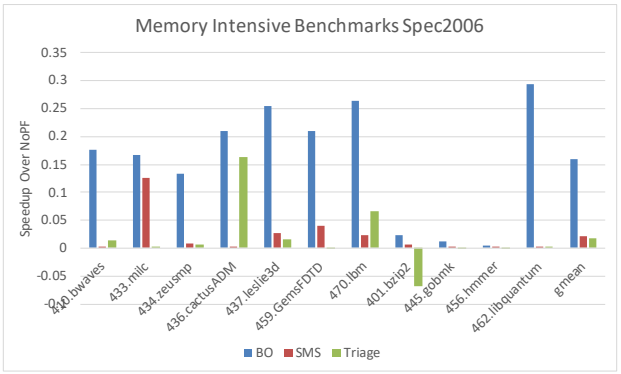



Figure 4.6: Results on regular SPEC 2006 benchmarks.

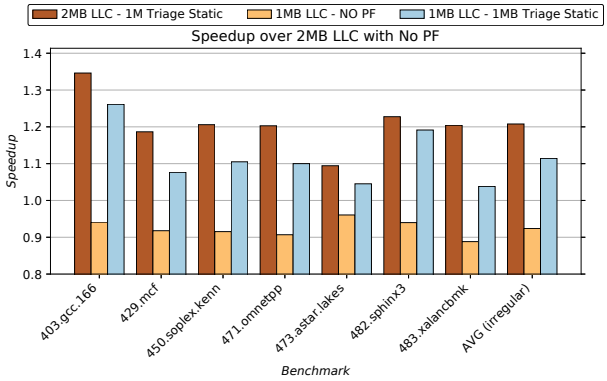



Figure 4.7: Breakdown of Triage's Performance Improvements

regular, Triage does not outperform BO, but we see that Triage's dynamic partitioning scheme avoids hurting performance on most benchmarks. 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. More sophisticated partitioning schemes that account for cache utility more accurately will help improve Triage in these scenarios.

**Sensitivity to Replacement Policy** Figure 4.9 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 4.10 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.

### 4.3.1 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 4.11 compares Triage against overly optimistic idealized versions of STMS and Domino and against a realistic version of MISB [43]. 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 4.11). 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.

To put these results in context, Figure 4.12 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.

**Energy Evaluation** Triage is more energy-efficient than other temporal prefetchers. Figure 4.13 shows that Triage's metadata accesses are $4 - 22\times$ 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 $10\times$ to $50\times$ more energy than an LLC access [4, 15], 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.

### 4.3.2 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 4.14). 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 4.14 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).

### 4.3.3 Evaluation on Multi-Programmed SPEC Mixes

Figure 4.15 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%.

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 4.16 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 re-iterate that Triage can prefetch irregular memory accesses that BO cannot.

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 4.17 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.

**Comparison On Mixes With Regular Programs** For completeness, Figure 4.18 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.

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 4.19 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 4.19 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.

## 4.4 Conclusion

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 and incurs significant traffic. To solve this problem, we presented Triage, a temporal prefetcher that removes the off-chip metadata requirement and stores metadata only on chip, making it practical to implement. Our experiments show that Triage performs better than other spatial prefetchers that only use on-chip metadata. We find that in a multi-core setting with

workload mixes consisting of both regular and irregular workloads, a hybrid prefetcher of Best Offset and Triage works well.

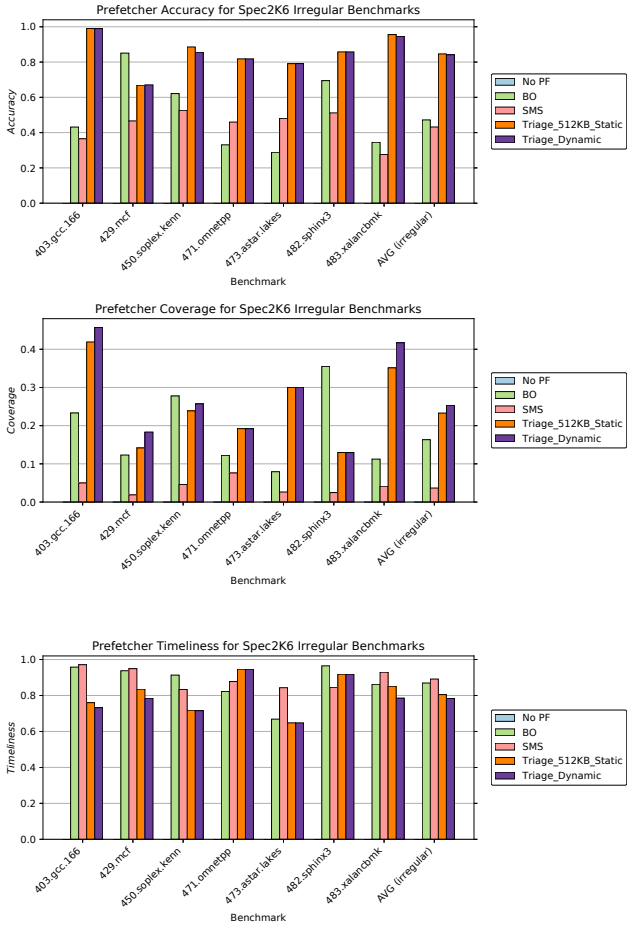



Figure 4.8: Triage improves coverage and accuracy.

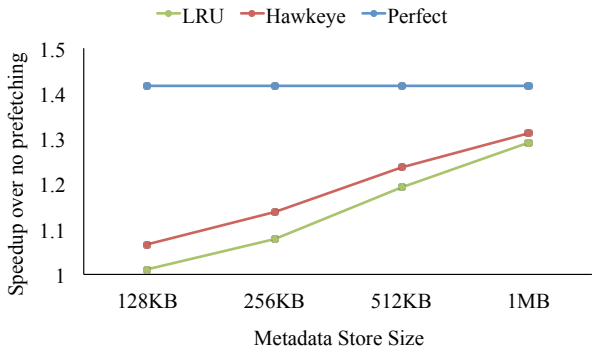



Figure 4.9: Sensitivity to Metadata Store Size (assuming no loss in LLC capacity).

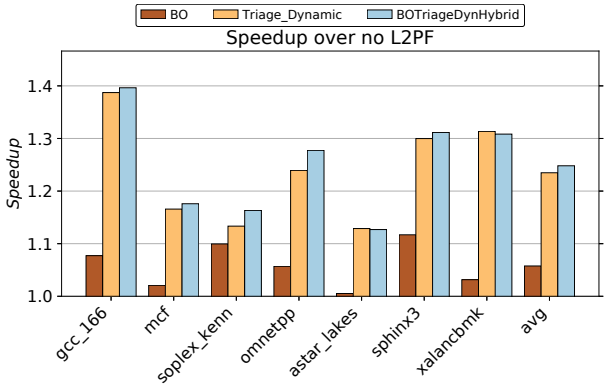



Figure 4.10: Triage performs well as part of a hybrid prefetcher.

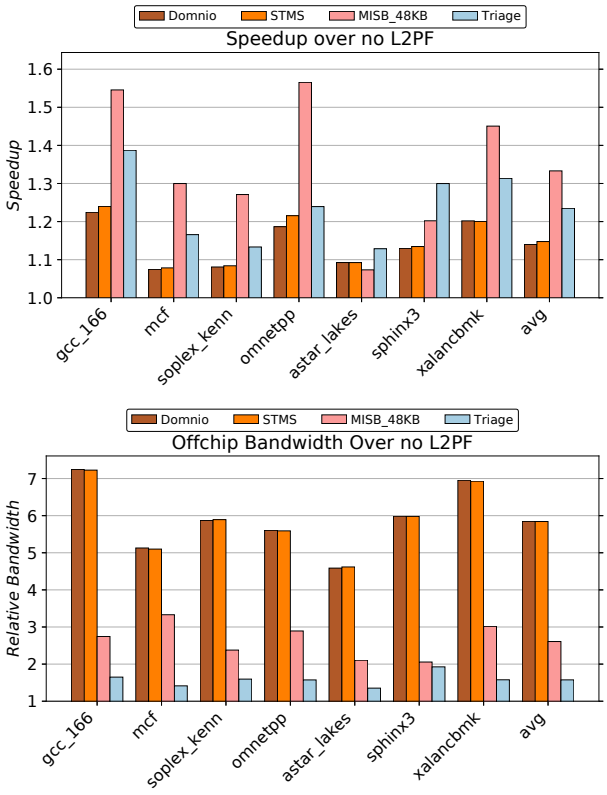



Figure 4.11: Triage reduces traffic compared to off-chip temporal prefetchers while offering good performance improvements.

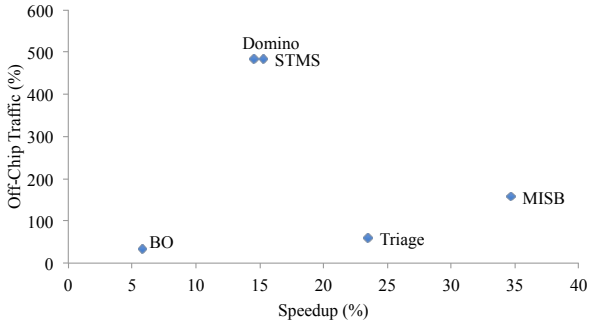



Figure 4.12: Design Space of Temporal Prefetchers.

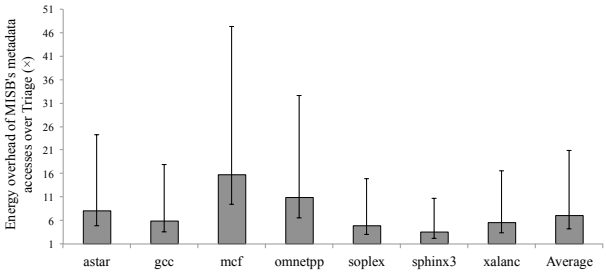



Figure 4.13: Triage is more energy efficient than MISB.

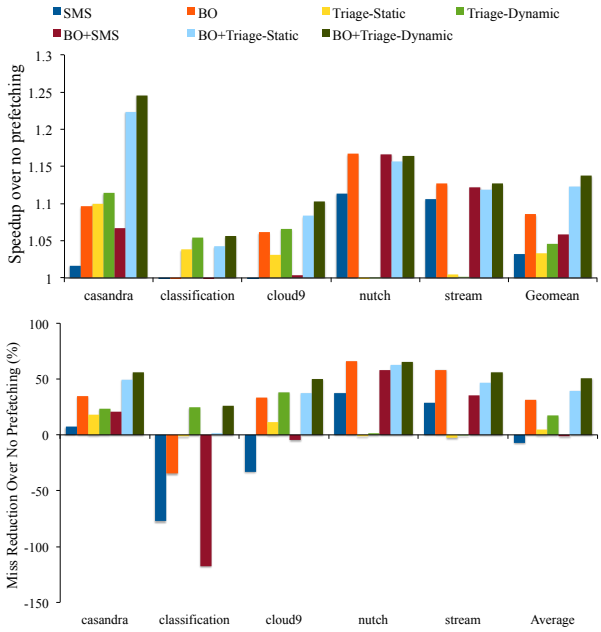



Figure 4.14: Triage works well for server workloads.

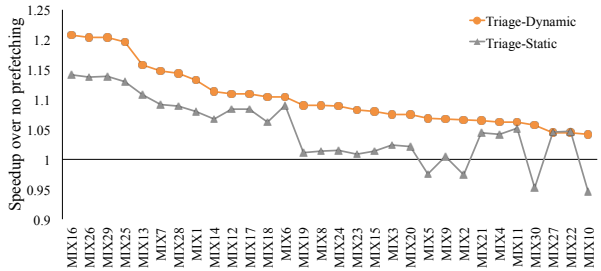



Figure 4.15: Triage-Dynamic improves over Triage-Static for shared caches.

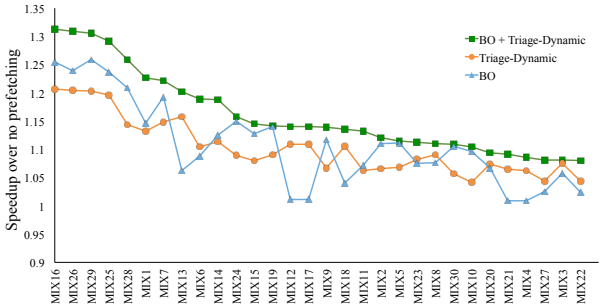



Figure 4.16: Triage works well on multi-programmed mixes of irregular programs running on a 4-core system.

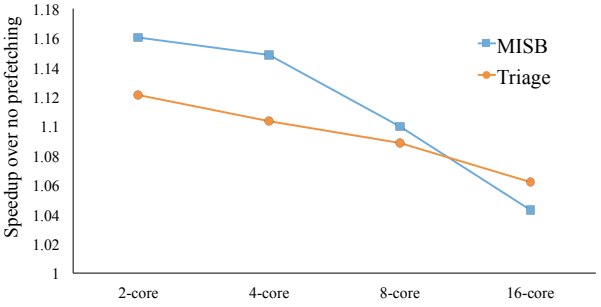



Figure 4.17: Triage outperforms MISB in bandwidth-constrained environments.

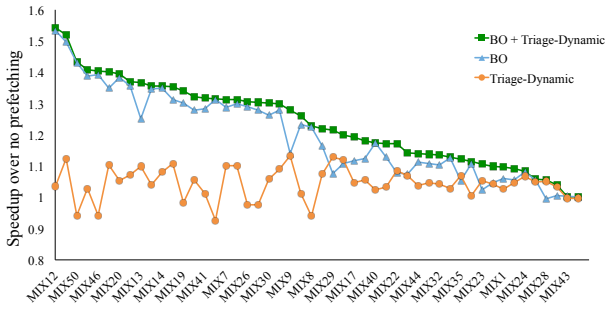



Figure 4.18: Triage works well on multi-programmed mixes of regular and irregular programs running on a 4-core system.

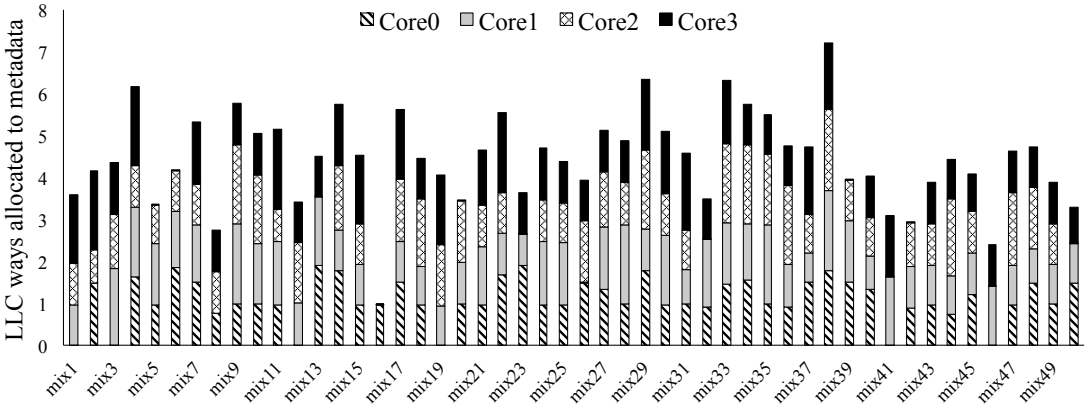



Figure 4.19: Dynamic Triage allocates different metadata store sizes to different cores.

# Chapter 5

# Reeses

Triage introduces a new design point for irregular prefetching by removing the need for off-chip metadata storage. By re-purposing part of last level cache (LLC) into metadata storage, Triage stores the most useful metadata on-chip and prefetches accordingly.

However, Triage causes a conflict between cache space and metadata space. Since the overall space of LLC is fixed, increasing metadata size inevitably decreases cache size used to store actual data and causes loss in cache hit rate. If we can reduce metadata size of Triage, more cache space can be used to store data.

To optimize metadata size for Triage, we observe that Triage as a temporal prefetcher does poorly for workloads with regular patterns. In particular, Triage records redundant information for regular patterns. For example, in Figure 5.1, Triage needs to record 5 mappings $(A, B), (B, B+1), (B+1, B+$

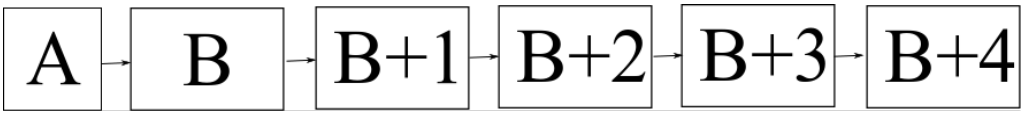



Figure 5.1: This graph shows an access stream containing both regular and irregular patterns.

$2), (B+2, B+3), (B+3, B+4)$ to prefetch the entire stream. Unlike Triage, a regular prefetcher like IP-stride prefetcher [19, 30, 36] only needs to record a single stride for prefetching $B+1, \cdots, B+4$.

Integrated Stream Prefetcher (ISP) proposed by M. Pabst [29] reduces space requirement for workloads with regular patterns by provides a method to represent both temporal and spatial access patterns. In ISP, each entry represents a spatial pattern instead of a single address. For example, ISP represents Figure 5.1 by one metadata entry: $(A, B, delta = 1, len = 5)$. Compared to ISB [16], ISP reduces metadata storage for regular patterns and do not impact irregular patterns. On average, ISP reduces metadata requirement for SPEC2006 [13] by $2\times$.

However, simply applying ISP's metadata representation to Triage does not work. We observe the following two problems in naively applying ISP to Triage.

First, directly applying ISP to Triage results in more conflict misses compared to Triage. Triage represents its metadata in set associative cache. Like other set associative cache, the set ID is determined by lower bits of addresses. However, since stream length has to be represented as a limited size value in hardware, trigger addresses in a long stream are not uniformly distributed and tend to have similar lower bits. For example, Figure 5.2 shows a long stream of length 1024 starting from $fabcde0001000000$. Assume ISP's maximum allowed stream length is 64. ISP breaks this stream into chunks like $(A, \cdots, A+63)$, $(A+64, \cdots, A+127)$, $\cdots$, $(A+960, \cdots, A+1023)$. All the

trigger addresses are different by 64 in this stream and have the same lower 6 bits, which are all conflicted in the same set in the metadata. The conflict misses problem is more severe in Triage since Triage does not have off-chip metadata like ISP to back up these conflicted entries.

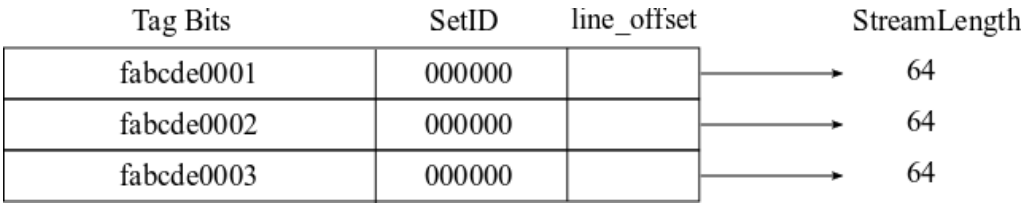



Figure 5.2: Set conflict in Reeses using Triage's organization.

Second, Triage's dynamic scheme uses OPTgen samplers to estimate metadata hit rate for different metadata sizes. Since the amount of OPTgen sampler is limited, Triage cannot experiment every possible value of metadata sizes. So it can overestimate the amount of necessary metadata. This limits the benefits of metadata reduction from ISP.

To solve these two problems, we propose a new resource management schemes, Reeses, for Triage's on-chip metadata. Reeses uses ISP's compressed metadata representation and makes the following two improvements to address these two problems.

First, we propose a new method managing on-chip metadata in set-associative cache by using hashed tags and set ID. This new scheme reduces metadata entry size and amount of conflict misses in metadata and improves performance from Triage.

Second, we propose a new Size Based Allocation (SBA) scheme that allocates metadata sizes based on metadata storage requirement dynamically. Combined with ISP, it better utilizes last level cache and yields more performance compared to old dynamic scheme of Triage.

Overall, by applying the compressed metadata representation and these two mechanism, Reeses achieves 4.3% more speedup than Triage for irregular benchmarks, and 2.7% for all SPEC2006 benchmarks. In a 4-core system, Reeses achieves 0.6% more speedup for CloudSuite and 2.2% more speedup for SPEC benchmark mixes.

## 5.1 Our Solution

The overall flow of Reeses is similar to Triage. For each LLC access, the metadata store is queried to see if a metadata entry is found. Reeses generates prefetches accordingly when such an entry is found. Regardless of whether an entry is found, Reeses updates the training unit with the access address, and the training unit decides whether a new metadata entry needs to be updated to the metadata store.

The major differences between Reeses and Triage are the following three parts. First, instead of a simple mapping from source address to target address, each metadata entry contains a stream of addresses like in ISP. Second, we use hashed tags and set ID to make metadata entries more compact and reduce conflict misses in searching for a metadata entry. Third, we have a new method SBA to determine the size of metadata store dynamically in the run. The rest

of the section describes each of these three parts.

### 5.1.1 Metadata Entry Composition

Figure 5.3 shows the structure of metadata representation in Reeses. Each Reeses entry is composed of four parts: tag, entry, stride and stream length. Tag represents a compressed version of the previous address. The compression scheme will be further described in section 5.1.2. The entry represents the first address in a strided stream following the previous address. Stride and stream length depicts the stride and length of the strided stream.

Stream A, B, B+1, B+2, B+3, C, D

| Tag | Target | StreamLength | Stride |
|---|---|---|---|
| A | B | 3 | 1 |
| B+3 | C | 1 | 1 |
| C | D | 1 | 1 |

Figure 5.3: Metadata Representation for Reeses. When there is a strided stream, Reeses will consolidate them into one entry.

**Training** Table 5.1 shows the transition table on how the training unit creates metadata entries. When a new address comes in, Reeses first checks if it belongs to the existing stream. If it does, Reeses simply updates the current stream. If it doesn't, Reeses writes out the current stream to metadata store and prepares for the next stream.

**Prediction** Upon seeing an address $A$, Reeses checks whether it has an entry in the metadata store. If such an entry is found, all addresses from the following address will be put in a prefetch buffer. Future access will prefetch addresses from this prefetch buffer until it is exhausted.

| Existing entry for PC | New Address | Action |
|---|---|---|
| (h(A), B, stride, len) | Same address as last address | Do nothing |
| (h(A), B, stride, len) | C in the same stream | Update entry to (h(A), B, len+1) |
| (h(A), B, stride, len) | C in different stream | Move entry to on-chip metadata, Update TU to (h(B+stride*(len-1)), C, 1, 1) |

Table 5.1: Reeses Training Transition Table

### 5.1.2 Metadata Organization

In Triage, the lower bits in cache line address is used as set ID in metadata. This approach works well in Triage because the lower bits of line addresses is uniformly distributed in the address space. However, in Reeses it is not the case and Figure 5.2 shows why.

Since stream length has limited bits in metadata, long streams need to be broken into small streams. When we look at all streams in Figure 5.2, the lower bits of adjacent streams tend to be similar, which are different by a multiple of stream length. As a result, this value is not uniformly distributed. Using it as set ID results in more conflict misses than desired.

Simply moving set ID to higher bits doesn't work either. When a

workload has all irregular patterns, different addresses in the same region can point to different addresses, causing a lot of conflict misses.

To solve this problem we would like to have a method mapping address to set ID such that each set ID has roughly the same amount of address assigned to it. Hong et al. [14] propose Touche, a compressed cache tag method by applying XOR on different sections of address. We use a similar approach here for the set ID. However, instead using the property of compressing, we use them as a hash function to hash address into different buckets labeled by their set IDs. Figure 5.4 shows how our scheme works.

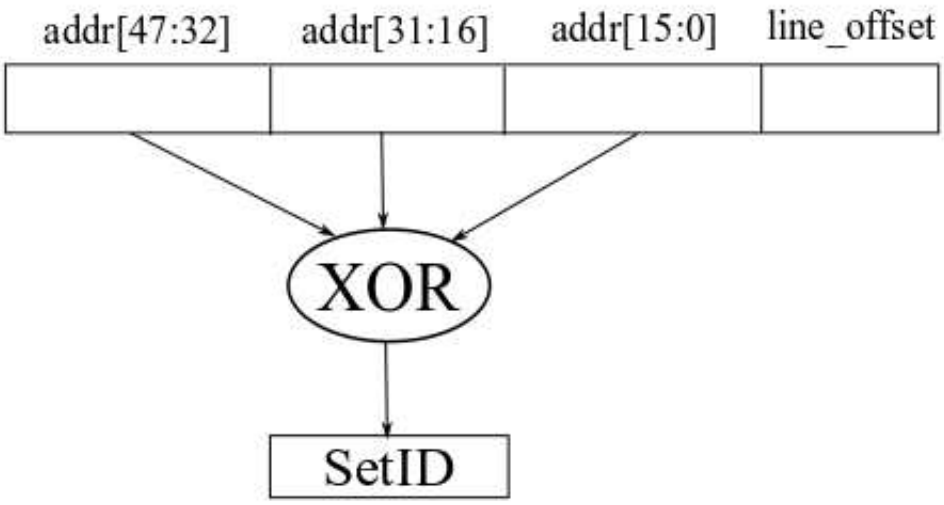



Figure 5.4: The computation method of SetID.

Similar technique can be used to compress tags for metadata. Figure 5.5 shows how the compression of tag works.

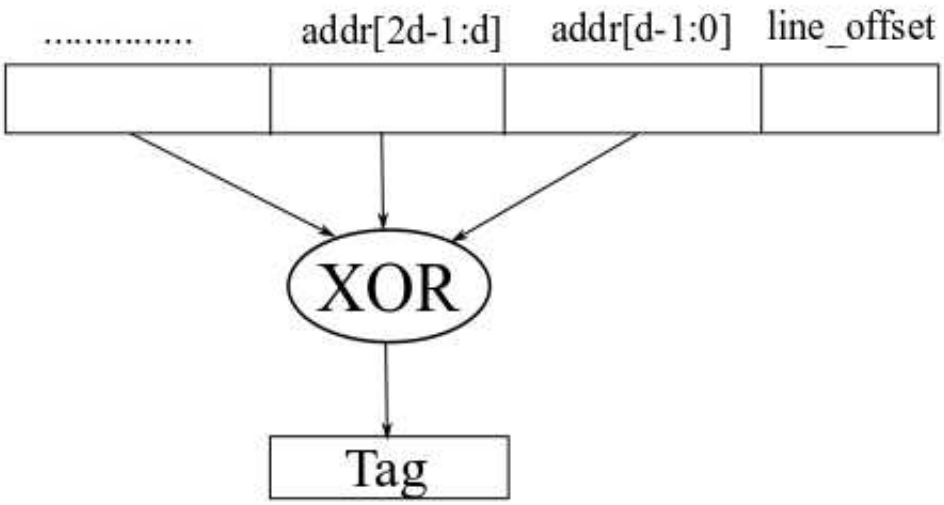



Figure 5.5: The computation method of Tag.

### 5.1.3 Dynamic Metadata Store Size

Reeses uses size based allocation (SBA) for metadata storage allocation. Similar to Triage, Reeses assigns separate ways to data and metadata. Unlike Triage, the number of ways is determined by the amount of required metadata size.

We have a Bloom filter [3] to record whether an address has been visited before and a global counter to record the amount of addresses we have seen. When we add a new entry to the metadata store, we also add it to this bloom filter. If an address is not present in the Bloom filter, it increments the global counter. The global counter indicates the amount of metadata Reeses needs.

Since Bloom Filters can yield false positives on a query, this scheme

can under-estimate the number of unique metadata entries. To accommodate false positives, we randomly increase the global counter on Bloom Filter hits. In particular, 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 [3]. 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.

## 5.2 Evaluation

### 5.2.1 Methodology

We use ChampSim [11, 23], a trace-based simulator that includes an out-of-order core model with a detailed memory system, to evaluate our results for Reeses. The detail description of ChampSim can be find in Section 3.2 Our configuration in Table 5.2. Both prefetchers are evaluated with degree of 16, which is the best configuration for both (See Figure 5.13).

**Benchmarks** We present single-core results for all memory-bound workloads from SPEC2006 [13], especially its irregular subset. For multi-core results we use CloudSuite [9] and multi-programmed SPEC benchmarks. More details of the benchmarks have been described in Section 3.2

| | |
|---|---|
| Core | Out-of-order, 2GHz,<br>4-wide fetch, decode, and dispatch<br>128 ROB entries |
| TLB | 48-entry fully-associative L1 I/D-TLB<br>1024-entry 4-way L2 TLB |
| L1I | 64KB private, 4-way, 3-cycle latency |
| L1D | 64KB private, 4-way, 3-cycle latency<br>Stride prefetcher |
| L2 | 512KB private, 8-way, 7-cycle latency |
| L3 | 2MB per core, shared, 16-way<br>12-cycle latency |
| DRAM | 8B channel width, 800MHz,<br>tCAS=20, tRP=20, tRCD=20<br>2 channels, 8 ranks, 8 banks, 32K rows<br>32GB/s bandwidth total |

Table 5.2: Machine Configuration

**Cache Replacement Policy** We experiment with two cache replacement policies: LRU and Hawkeye [17] replacement policies.

### 5.2.2 Single Core Results

Figure 5.6 shows that Reeses outperforms Triage. Compared to Triage's 25.3% speedup, applying the SBA dynamic allocation achieves 28.2% speedup on irregular SPEC2006 benchmarks, applying Reeses's metadata representation but using old allocation scheme achieves 27.9% speedup, while applying both of the optimizations achieve 29.6% speedup.

For all SPEC2006 benchmarks, Triage achieves 8.1% speedup, while Triage+SBA achieves 9.8%, Reeses with old dynamic scheme achieves 9.3% and Reeses with SBA achieves 10.8%.

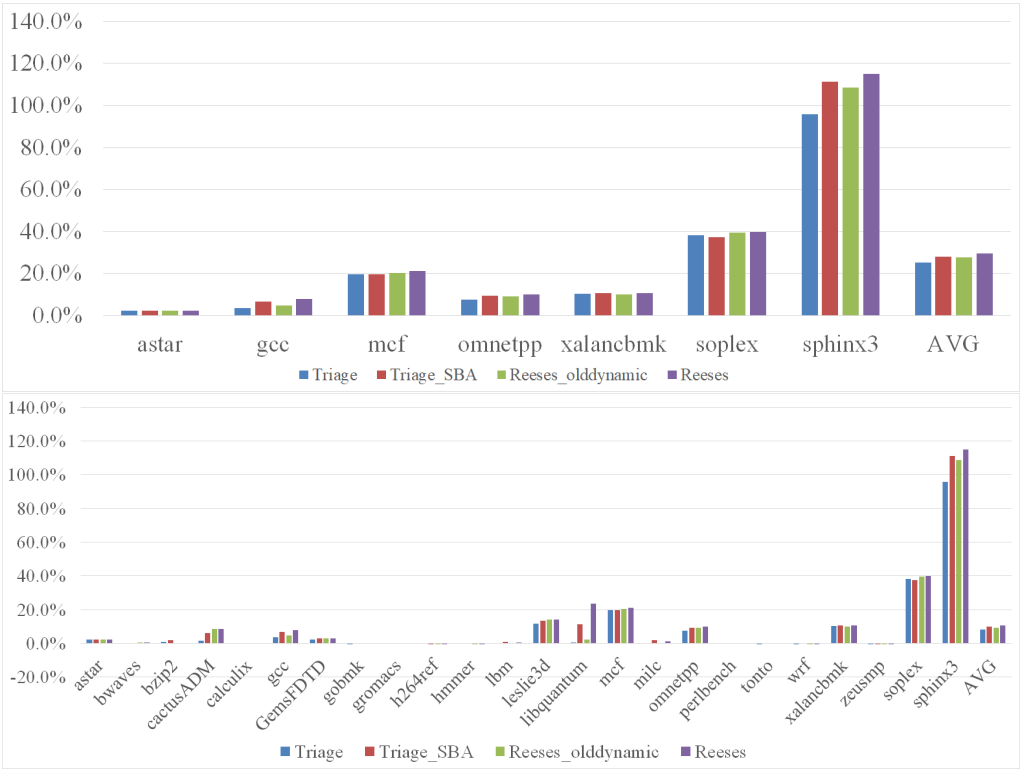



Figure 5.6: Speedup over no Prefetch on LRU Replacement Policy.

### 5.2.3 Multi Core Results

**CloudSuite Benchmarks** Figure 5.7 shows the performance of Reeses compared to Triage. Reeses has 4.6% speedup over no prefetch while Triage has 4.0%.

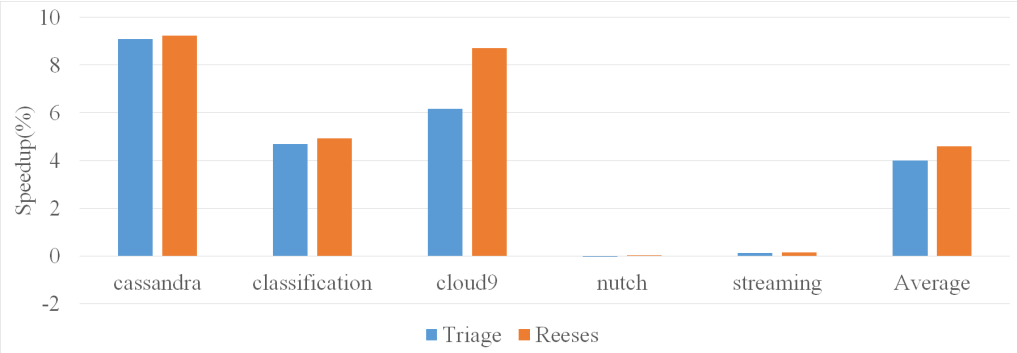



Figure 5.7: Speedup Comparison on CloudSuite

**Multi-Programmed SPEC Benchmarks** Reeses performs well on multi-programmed SPEC benchmarks. Figure 5.8 shows the performance for Reeses in 4, 8, 16 core multi-core SPEC benchmarks. On 4-core multi-programmed workloads, Reeses has 8.4% speedup compared to 6.2% for Triage. On 8-cores, Reeses has 7.1% speedup compared to 5.6% for Triage. On 16-cores, Reeses has 6.7% compared to 6.2% for Triage. On irregular-only benchmarks, Reeses achieves 14.2%, 13.5%, 14.0% for 4, 8, 16 cores respective, compared to 13.5%, 12.9%, 13.6% for Triage, as shown in Figure 5.9.

### 5.2.4 Understanding Reeses's Performance

**Metadata Reduction** Figure 5.11 shows the amount of metadata required between Triage and Reeses. On average, Reeses reduces the total amount of metadata from $448KB$ to $278KB$, a reduction of 38%. The amount of reduction is larger for benchmarks with regular patterns. For example, Reeses reduces the amount of metadata for libquantum by 94%.

**Reduction of Conflict Misses** Figure 5.11 shows the amount of non-compulsory misses in metadata between old and new scheme of generating set ID for Reeses. The amount of averages is reduced from 2546 to 1881, a reduction of 26%.

### 5.2.5 Sensitivity Study

**Compressed Tag Bits** Figure 5.12 shows the impact of compressed tag bits to average performance. The average difference for different bits is within 0.5%.

**Degree Analysis** Figure 5.13 shows the average performance for difference degrees between Triage and Reeses. For all degrees Reeses is better than original Triage. Degree of 16 has the best performance for both prefetchers, so we will compare between them.

## 5.3 Conclusion

On-chip only prefetcher like Triage reduces hardware complexity and traffic overhead for temporal prefetching. However, handling allocation of on-chip metadata is a challenge. Reeses improves on-chip metadata by applying a more compact metadata representation and more efficient metadata allocation method. It achieves better performance compared to original Triage.

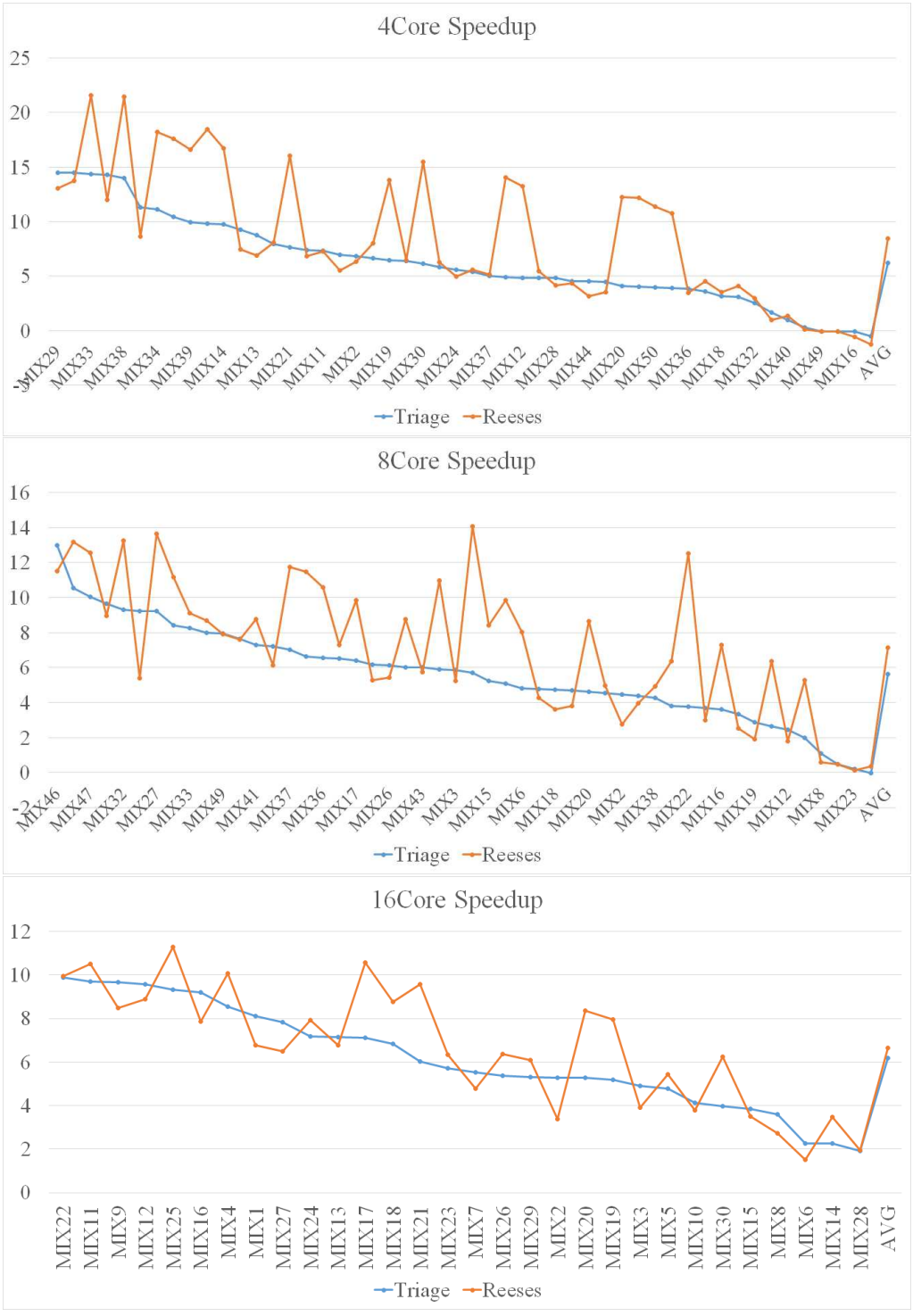



Figure 5.8: Reeses's performance on multi-core SPEC2006 benchmarks.

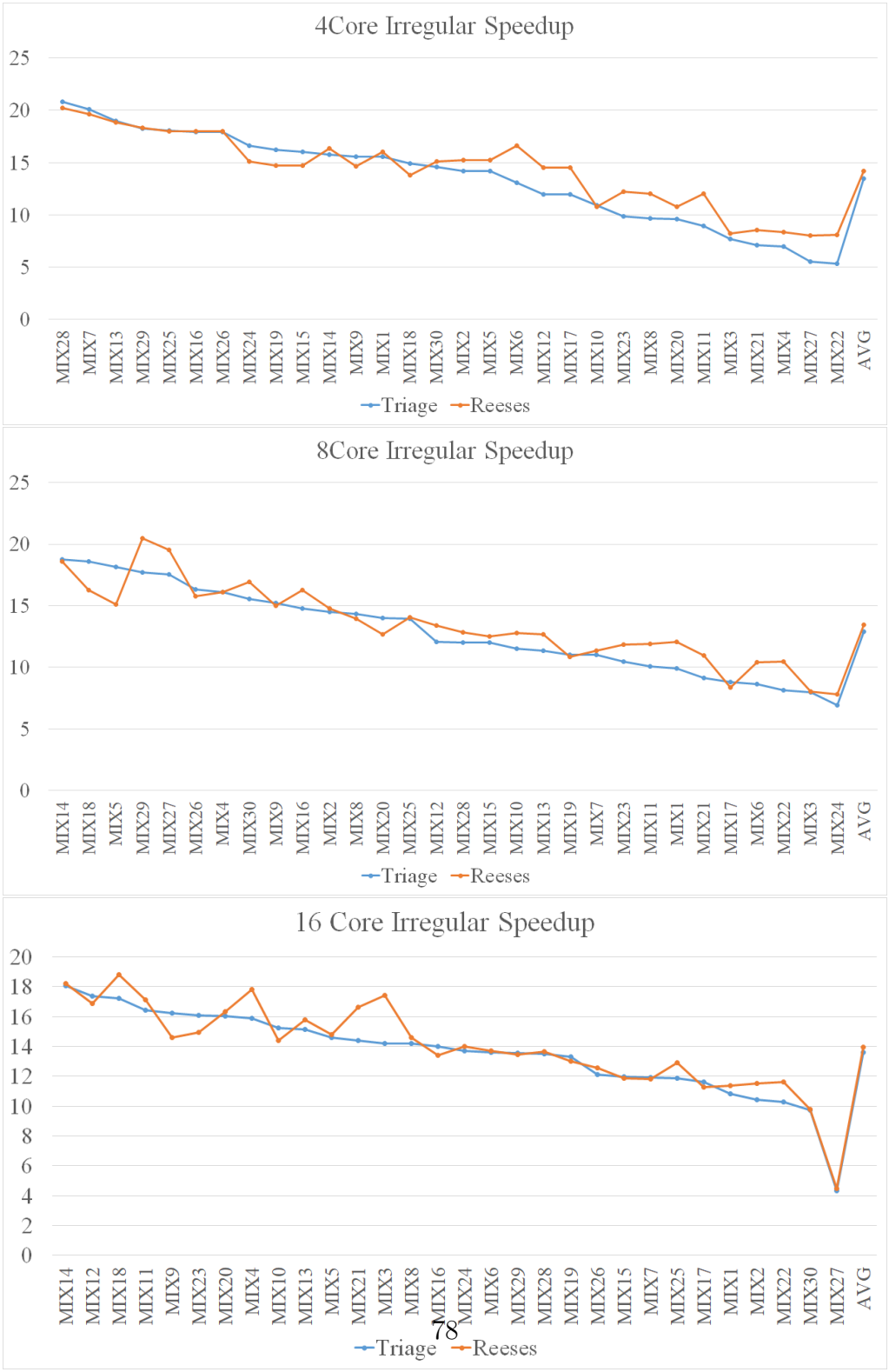





Figure 5.9: Reeses's performance on multi-core irregular SPEC2006 benchmarks.

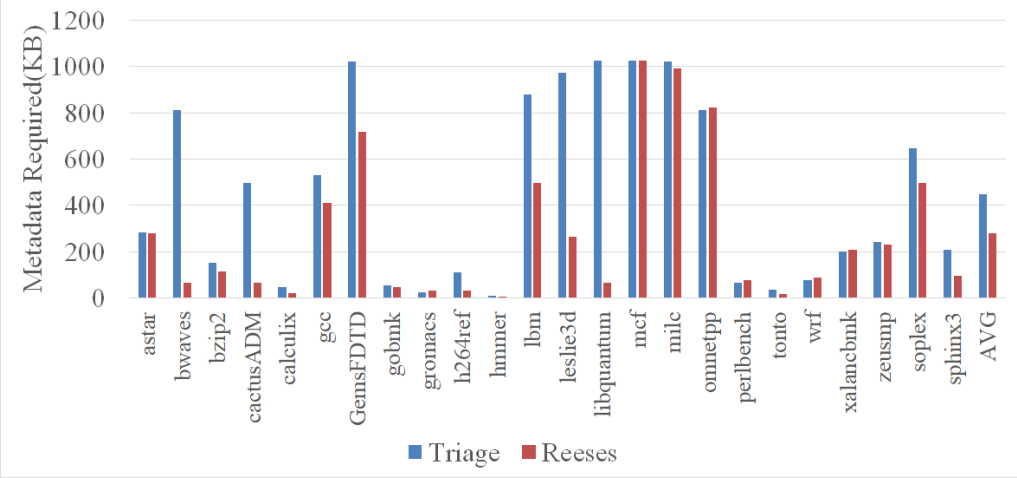



Figure 5.10: Amount of metadata required between Triage and Reeses.

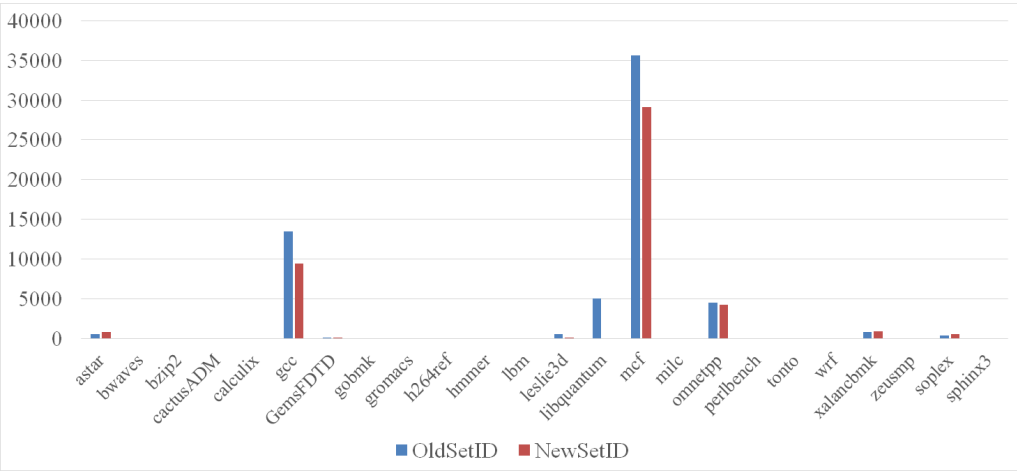



Figure 5.11: Number of capacity and conflict misses between old and new method of generating set ID.

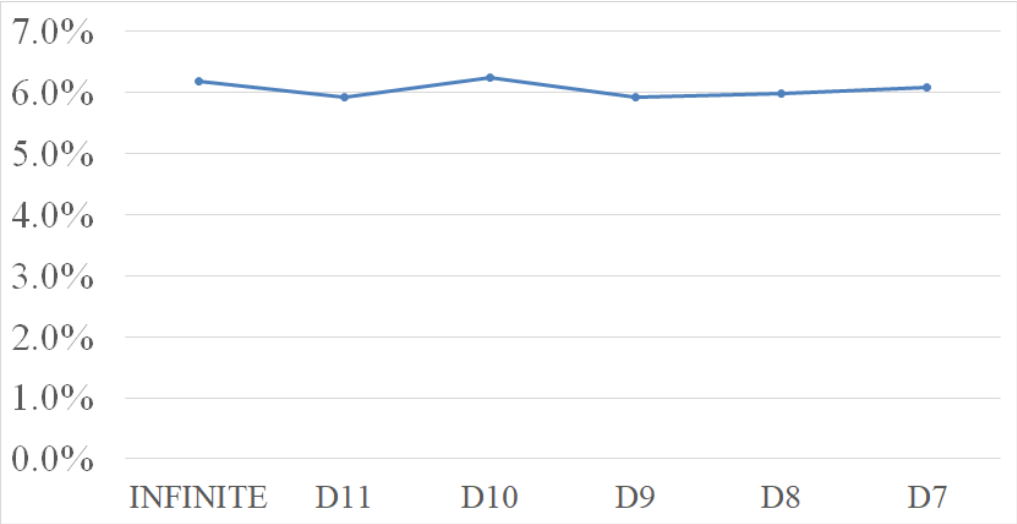



Figure 5.12: Impact of performance for different compressed tag bit count.

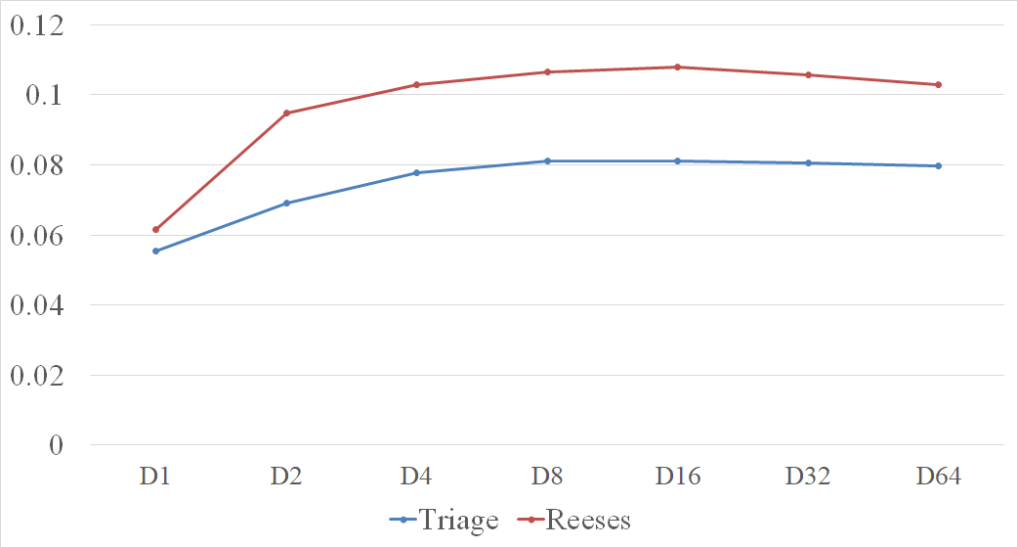



Figure 5.13: Average Performance for different degree.

# Chapter 6

# Conclusion and Future Work

In this thesis, we present several solutions for improving irregular prefetching and for making it practical.

First, we present MISB, an efficient method for managing off-chip metadata for temporal prefetching. Second, we present Triage, the first on-chip only temporal prefetcher. Finally, we present Reeses, a metadata management scheme that improves metadata efficiency for Triage. All these work towards building a practical irregular prefetcher.

Looking into future, this thesis leaves the following two future research questions:

- How can we further improve irregular prefetching?
- Can we apply temporal prefetching to areas other than data prefetching?

## 6.1 Further Improvement

Despite all the benefits, temporal prefetchers have limitations that restricts amount of benefits. First, temporal prefetchers cannot get compulsory

misses. Second, temporal prefetchers' metadata does not scale to workload size. Third, temporal prefetchers suffer from aliasing problem.

**Compulsory Misses** Temporal prefetchers exploit past temporal correlations to do prefetch. As a result, they cannot prefetch compulsory misses since these misses are addresses that have not occurred. Previous work [16, 37, 42, 43] solve this problem by running a spatial prefetcher for these compulsory misses. These spatial prefetchers do not work well for irregular workloads. To cover compulsory misses on irregular workloads, further work need to be researched to apply temporal correlations to addresses that have not been visited.

**Workload Scalability** Temporal prefetchers record past temporal correlations for accesses. When workload size grows, the amount of required metadata also grows. The problem is more severe for on-chip only prefetchers like Triage because they do not have an off-chip metadata to back up growing metadata.

One idea to solve this problem is to classify different types of metadata. Metadata can be classified into three types: those won't be reused, those will be reused in short-term and those will be reused in long-term. We do not need to store the ones that won't be reused. We can store the short-term ones in on-chip metadata and long-term ones in off-chip metadata. Classifying metadata in the run is the challenge here.

**Aliasing** Temporal correlation is not guaranteed to be one-on-one. Multiple addresses can correlate to the same address, which causes aliasing. Temporal prefetchers could prefetch wrong addresses from the candidates, which reduces prefetching coverage and increases cache pollution.

Contextual information can be used to distinguish between aliasing. For example, different successor of an address can come from different instruction, different branch history or different previous accesses. Using this information can potentially distinguish between aliased accesses.

There are two challenges for using contextual information. First, it is difficult to identify which information is useful. Using incorrect information can hurt performance. Second, storing contextual information requires more metadata storage, increasing the already existing massive hardware resource usage problem.

## 6.2 Other Applications

As Ayers et al. points out, instruction cache misses contribute to significantly cost in warehouse scale workloads. For example, 13.8% of the performance potential of a Google's web search load test is wasted in front end latencies which are dominated by instruction cache misses. [1]

Previous work [10, 20] have already applied temporal prefetching to instruction prefetching and achieved reasonable performance benefits. For example, SHIFT proposed by Kaynak et al. [20] eliminates 81% of instruction

cache misses in 16-core server workload.

Instruction prefetching faces different challenges than data prefetching. First, time different between consecutive instruction fetches is much lower than consecutive data accesses. As a result, timely prefetching is more important in instruction prefetching. Second, instruction streams are more regular since only jump and call instructions can change the direction of instruction flow. Therefore, there are more potential in applying compression scheme like Reeses in instruction prefetching. Third, instruction prefetching can utilize branch information [21, 24].

# Index

# Bibliography

[1] Grant Ayers, Nayana Prasad Nagendra, David I. August, Hyoun Kyu Cho, Svilen Kanev, Christos Kozyrakis, Trivikram Krishnamurthy, Heiner Litz, Tipp Moseley, and Parthasarathy Ranganathan. Asmdb: Understanding and mitigating front-end stalls in warehouse-scale computers. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 462–473, New York, NY, USA, 2019. ACM.

[2] Mohammad Bakhshalipour, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. Domino temporal data prefetcher. In *High Performance Computer Architecture (HPCA), 2018 IEEE 24th International Symposium on*, pages 131–142, 2018.

[3] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

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

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

[6] Mark J. Charney and Anthony P. Reeves. Generalized correlation based hardware prefetching. Technical Report EE-CEG-95-1, School of Electrical Engineering, Cornell University, 1995.

[7] Mary J. Charney. *Correlation-based hardware prefetching.* PhD thesis, Cornell University, 1996.

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

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

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

[11] Paul Gratz, Jinchun Kim, and Gino Chacon. 2nd cache replacement championship, 2017.

[12] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, Sixth Edition*. Morgan Kaufmann Publishers, 2017.

[13] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.

[14] Seokin Hong, Bulent Abali, Alper Buyuktosungolu, Michael Healy, and Prashant 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*, pages 453–465, 2019.

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

[16] Akanksha Jain and Calvin Lin. Linearizing irregular memory accesses for improved correlated prefetching. In *46rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, December 2013.

[17] Akanksha Jain and Calvin Lin. Back to the future: Leveraging Belady's algorithm for improved cache replacement. In *Proceedings of the 43$^{th}$ International Symposium on Computer Architecture (ISCA)*, June 2016.

[18] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 252–263, 1997.

[19] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, pages 364–373, 1990.

[20] Cansu Kaynak, Boris Grot, and Babak Falsafi. Shift: Shared history instruction fetch for lean-core server processors. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 272–283, New York, NY, USA, 2013. Association for Computing Machinery.

[21] Cansu Kaynak, Boris Grot, and Babak Falsafi. Confluence: Unified instruction supply for scale-out servers. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 166–177, New York, NY, USA, 2015. Association for Computing Machinery.

[22] J. Kim, S. H. Pugsley, P. V. Gratz, A. L. N. Reddy, C. Wilkerson, and Z. Chishti. Path confidence based lookahead prefetching. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016.

[23] Jinchun Kim, Elvira Teran, Paul V Gratz, Daniel A Jiménez, Seth H Pugsley, and Chris Wilkerson. 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)*, pages 737–749, 2017.

[24] R. Kumar, C. Huang, B. Grot, and V. Nagarajan. Boomerang: A metadata-free architecture for control flow delivery. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, HPCA'17, pages 493–504, 2017.

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

[26] Pierre Michaud. Best-offset hardware prefetching. In *High Performance Computer Architecture (HPCA), 2016 IEEE 22th International Symposium on*, 2016.

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

[28] Craig G Nevill-Manning and Ian H Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 7:67–82, 1997.

[29] Matthew Pabst. Analyzing spatial and temporal locality for integrated stream prefetching. Technical report, The University of Texas at Austin, 2018.

[30] Subbarao Palacharla and Richard E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the 21st Annual*

*International Symposium on Computer Architecture*, ISCA '94, pages 24–33, 1994.

[31] Seth Pugsley, Alaa Alameldeen, and Chris Wilkerson. 2nd data prefetching championship, 2015.

[32] Seth H Pugsley, Zeshan Chishti, Chris Wilkerson, Peng-fei Chuang, Robert L Scott, Aamer Jaleel, Shih-Lien Lu, Kingsum Chow, and Rajeev Balasubramonian. Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, 2014.

[33] M. V. Ramakrishna, E. Fu, and E. Bahcekapili. Efficient hardware hashing functions for high performance computers. *IEEE Trans. Comput.*, 46(12):1378–1381, December 1997.

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

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

[36] Alan Jay Smith. Sequential program prefetching in memory hierarchies. *IEEE Transactions on Computers*, 11(12):7–21, 1978.

[37] Stephen Somogyi, Thomas F. Wenisch, Anastasia Ailamaki, and Babak Falsafi. Spatio-temporal memory streaming. In *ISCA*, pages 69–80, 2009.

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

[39] Thomas F. Wenisch. *Temporal Memory Streaming*. PhD thesis, Carnegie Mellon University, Department of Computer Science, 2007.

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

[41] Thomas F. Wenisch, Stephen Somogyi, Nikolaos Hardavellas, Jangwoo Kim, Anastassia Ailamaki, and Babak Falsafi. Temporal streaming of shared memory. *SIGARCH Computer Architecture News*, 33(2):222–233, May 2005.

[42] Hao Wu, Akanksha Jain, Calvin Lin, Krishnendra Nathella, and Dam Sunwoo. Temporal prefetching without the off-chip metadata. In *Proceedings of the 52th International Symposium on Microarchitecture*, MICRO'19, 2019.

[43] Hao Wu, Krishnendra Nathella, Dam Sunwoo, Akanksha Jain, and Calvin Lin. Efficient metadata management for irregular data prefetching. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA'19, pages 449–461, New York, NY, USA, 2019. ACM.

[44] Xiangyao Yu, Christopher J Hughes, Nadathur Satish, and Srinivas Devadas. Imp: Indirect memory prefetcher. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 178–190. ACM, 2015.

# Vita

Hao Wu was born in Guangzhou, China on 20 March 1990, the son of Sanmao Wu and Wendao Huang. He received the Bachelor of Science degree in Computer Science from Tsinghua University in 2010. He joined The University of Texas at Austin in 2010.

Permanent Email: wuhaotsh@gmail.com

Permanent address: 2317 Speedway
Austin, Texas 78712

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.