SPECULATIVE TECHNIQUES FOR MEMORY HIERARCHY MANAGEMENT

A Dissertation

by

LAITH MOHAMMAD ALBARAKAT

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Paul V. Gratz
Committee Members,	Daniel A. Jiménez
	Samuel Palermo
	I-Hong Hou
Head of Department,	Miroslav M. Begovic

May 2021

Major Subject: Computer Engineering

ABSTRACT

The "Memory Wall" [1], is the gap in performance between the processor and the main memory. Over the last 30 years computer architects have added multiple levels of cache to fill this gap, cache levels that are closer to the processors are smaller and faster. On the other hand, the levels that are far from the processors are bigger and slower. However the processors are still exposed to the latency of DRAM on misses. Therefore, speculative memory management techniques such as prefetching are used in modern microprocessors to bridge this gap in performance.

First, we propose Synchronization-aware Hardware Prefetching for Chip Multiprocessors, a novel hardware data prefetching scheme designed for prefetching shared-memory, multi-threaded workloads. This is the first work we are aware of to characterize the causes of poor prefetching performance in shared- memory multi-threaded applications. These are the inability to prefetch beyond synchronization points and tendency to prefetch shared data before it has been written. SB-Fetch, a low-complexity, low-overhead prefetcher design that addresses both issues.

Second, we propose a new prefetching algorithm, Set-Level Adaptive Prefetching for Compressed Caches (SLAP-CC), which seeks to address this problem by varying the prefetching aggressiveness based on how much effective capacity is available in each set. The ontributions of this work is characterize the increase and per-set variability of cache efficiency which typical cache compression schemes create, and propose a new prefetching scheme, SLAP-CC, designed to leverage this cache efficiency variability.

Third, we propose a new scheduling mechanism that predicts hard-to-prefetch loads at issue time and preemptively schedule them for execution as soon as they are ready, to allow the cache hierarchy to start the mishandling mechanism sooner. Such scheduling mechanism reduces the miss penalty on the dependent instructions after a hard-to-prefetch loads.

ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Paul V. Gratz for the opportunity to participate in research with the Computer Architecture, Memory Systems and Interconnection Networks(CAMSIN) research group. His advice and feedback that made this work successful.

I would also like to express my gratitude to my advisory committee members, Dr. Daniel A. Jiménez, Dr. I-Hong Hou and Dr. Samuel Palermo for their time and effort reviewing this work.

Finally, I would like to thank all my family for their love, encouragement, and support.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a dissertation committee consisting of Dr. Paul V. Gratz, Dr. I-Hong Hou, Dr. Samuel Palermo of the Department of Computer and Electrical Engineering and Dr. Daniel A. Jimeǹez of the Department of Computer Science and Engineering.

All other work conducted for the dissertation was completed by the student independently.

Funding Sources

There are no outside funding contributions to acknowledge related to the research and compilation of this document.

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

Big Data has revolutionized many areas of scientific research, science and technology. For the last decade, computer scientists developed tools to mine, structure, and visualize data in ways never before possible. As we move to the future to welcome the "Insight Era.", where machine learning and artificial intelligence are refined to facilitate data-driven decision-making. Our ability to perform computation on massive data sets is expected to increase.

Data-intensive computation forms significant challenges for computer architects. In particular, the volume and movement of large data sets generate significant stress on the memory hierarchy, leading to performance bottleneck.

1.1 Speculative Techniques for Memory Hierarchy

The memory hierarchy has been a determining factor of overall system performance. Access to off-chip memory costs many cycles and consumes more energy. Therefore, designing efficient memory hierarchy requires low access latencies and maximizes the hit rates.

Figure 1.1 outlines a typical cache hierarchy in a modern system. The "Memory Wall" [1], is the gap in performance between the processor and the main memory. Over the last 30 years computer architects have added multiple levels of cache to fill this gap, cache levels that are closer to the processors are smaller and faster. On the other hand, the levels that are far from the processors are bigger and slower. The goal of the this hierarchy is to improve average memory access time by frequently handling the demand requests at the cache, and avoiding the long access latency of DRAM. Despite this hierarchy the processors are still exposed to the latency of DRAM on misses. Cache size is a key design parameter that impacts performance, die area, and power consumption. With the end of Dennard scaling [2], growing cache size comes at an increasingly high cost in terms of power and energy consumption.

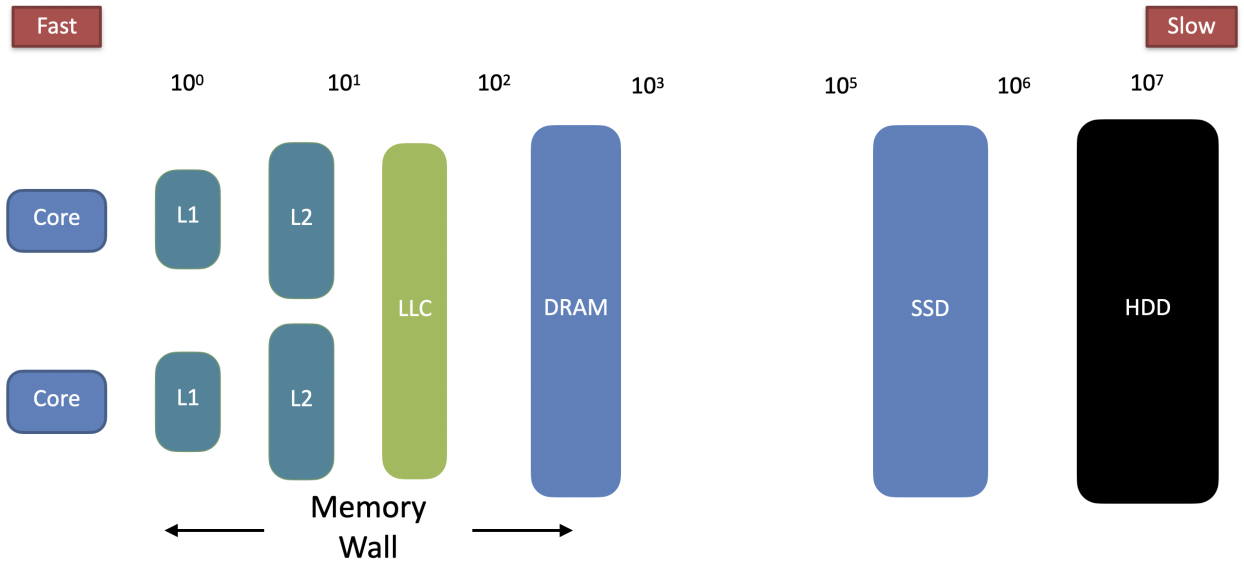


Figure 1.1: Memory access latency.

To hide memory access latency, computer architects have focused on speculative memory management techniques such as data prefetching; cache replacement policies; and reuse prediction.

Data prefetching in particular gained a lot of interest in the computer architect community and has been deeply studied. In addition to that, hardware prefetchers have been deployed in modern microprocessor design. Prefetching is predicting a subsequent memory access and fetching the required values ahead of the memory access to hide any potential long memory latency. To be effective, prefetching must be timely, accurate and introduce low hardware overhead. Ideally, a perfect prefetcher could make all memory accesses hit in the level one cache. However, in practice prefetchers may not be timely or accurate. Despite the fact that hardware prefetchers have an impressive impact on instructions per cycle by reducing the number of cache misses, the processor is still exposed to the memory wall on a cache miss due to Hard-to-Prefetch loads.

Another cache optimization technique that interacts positively with data prefetching is cache compression. Cache compression has the potential to increase the effective capacity

of the cache with extra area overhead. Compressed caches can achieve the benefits of larger caches without making the cache physically larger by fitting more cache blocks in the same cache space. Building a compressed cache centered around two main components: a compression algorithm that maps the data to a compressed format with fewer bits, and cache layout to fit compressed blocks in the cache [3]. To be effective, cache compression must be fast, simple, and effective in saving storage space. Therefore, the compression ratio should be large and the hardware complexity of implementing the scheme should be low in terms of area and power. The biggest challenge to build a compressed cache in modern microprocessors is the decompression latency. Unlike compression latency, which can take place off the critical path, decompression is on the critical path and increases the cache access time.

1.2 Dissertation Statement

In this dissertation, we aim to develop a new set of speculative techniques for modern memory hierarchies, resulting in performance improvements and attractive to be implemented in future systems.

To this end, we first characterize the causes of poor prefetching performance in shared-memory multi-threaded applications. These are the inability to prefetch beyond synchronization points and tendency to prefetch shared data before it has been written. Building upon this characterization, we propose SB-Fetch, a low-complexity, low-overhead prefetcher design that addresses both issues.

Second, we propose a new prefetching algorithm, Set-Level Adaptive Prefetching for Compressed Caches (SLAP-CC), which seeks to address this problem by varying the prefetching aggressiveness based on how much effective capacity is available in each set. The contributions of this work is characterize the increase and per-set variability of cache efficiency which typical cache compression schemes create, and propose a new prefetching scheme, SLAP-CC, designed to leverage this cache efficiency variability.

Third, we propose a new scheduling mechanism that predicts hard-to-prefetch loads at issue time and preemptively schedule them for execution as soon as they are ready, to allow

the cache hierarchy to start the mishandling mechanism sooner. Such scheduling mechanism reduces the miss penalty on the dependent instructions after a hard-to-prefetch loads.

Each speculative technique is designed to have low hardware complexity so that it can be easily implemented in future systems.

1.3 Dissertation Organization

In the following chapters, each speculative technique is first introduced with the main research motivations that lead up to this work, followed by detailed design implementation, then followed by evaluation methodology and results obtained. To put our work into perspective, Chapter 2 introduces a background and some of the most relevant related work. Chapter 3 discusses the synchronization-aware hardware prefetching for chip multiprocessors. Chapter 4 introduces set-level adaptive prefetching for compressed caches. In Chapter 5 we present our new scheduling mechanism for hard to prefetch loads. Finally, Chapter 6 concludes this dissertation.

2. BACKGROUND AND RELATED WORK

Our work builds on the contributions of many other researchers to the field. To place our work in the context of other research, we now review some of the most recent related work.

2.1 Data Prefetching

Data prefetching is a well known technique in which the cache is pre-filled with useful data ahead of an actual demand load request coming from the processor. Several hardware prefetchers with diverse prefetching strategies have been proposed in the literature.

History-based prefetching is the most commonly used among hardware prefetching strategies, where a prefetch engine uses the history of memory references to predict future references and generate prefetch requests. Spatial Memory Streaming (SMS) prefetcher is one of the current top-performing, light-weight, history based prefetchers [4, 5]. SMS predicts the future access pattern within a spatial region around a miss, based on a history of access patterns initiated by that missing instruction [4]. SMS uses the concept of a spatial region which begins with the first demand miss to a region and ends with the eviction or invalidation of any block from that region [4]. Spatial prefetchers are ineffective for pointer-based data structures with arbitrary memory layouts and have shown limited effectiveness for some workloads with many pointer-chasing access patterns [4, 6, 7, 5].

Runahead-based prefetching uses the execution core to pre-execute a set of instructions speculatively instead of stalling the resources on a long latency cache miss that goes all the way to memory [8, 9, 10]. Run-ahead was originally proposed in the context of in-order cores by Dundas and Mudge [11]. Mutlu et al. proposed an implementation to support runahead execution in out-of-order processors [8], in the implementation, when a memory operation misses in the second-level cache, the processor enters runahead mode and speculatively pre-executed future instructions to initiate prefetching. The drawback with using runahead prefetchers, there is a large overhead to recover from the runahead mode and restart normal

execution. Also, because of this overhead, the effectiveness of this mechanism to handle shorter latencies like the first-level cache miss latencies is reduced [12, 13, 6, 5, 8, 9, 10].

Branch-predictor-directed prefetchers reuse existing branch predictors to explore future control flow. These techniques use the branch predictor to recursively make future predictions to find instruction-block addresses for prefetch. Because branch predictors are decoupled from the rest of the pipeline, predictors can theoretically advance ahead of execution to an arbitrary extent to predict future control flow. Liu *at el.* [14, 12, 13, 6, 5] proposed branch-based data prefetching, which associated the history of data references to the previous branch instructions in the Branch Target Buffer (BTB). The BTB is then used to issue prefetches for load instructions following the branch instruction in the program flow. Pinter *at el.* proposed Tango prefetcher for superscalar implementations, to further improve the quality of stride-based reference prediction table approach proposed by Chen *at el.* [15]. Some prefetchers, such as B-Fetch [12, 13, 6, 5], are triggered by the fetch of a branch instruction by the processor, making them more suitable for prefetching beyond synchronization points as we will discuss. B-Fetch is a data cache prefetcher that employs two speculative components. It speculates on the expected path through future basic blocks, using a lookahead mechanism that relies on branch prediction to predict that execution path, and a scheme to predict the effective addresses of load instructions along that path based on the register file transformations per-basic block [12, 13, 6, 5]. B-Fetch records the variation of register contents at earlier branch instructions and uses this knowledge to predict the effective address. B-Fetch is a light-weight and very accurate prefetcher, but it requires a very complex hardware and a lot of hooks to the microarchitecture [12, 13, 6, 5].

Sequential Pattern Prefetchers are limited to waiting until a cache miss occurs, and then reading either a set of lines sequentially following the current miss [16], a set of lines following a strided pattern with respect to the current miss [17], or a set of blocks spatially around the miss [4, 5]. More recent prefetchers attempt to predict complex, irregular access patterns [4, 18, 19, 20, 21, 22, 6]. While these methods show significant benefit, they are

inherently reactive, waiting until a cache miss occurs before they initiate prefetches down the speculated path.

Some recent prior work has examined the case of prefetching in specialized multi-threaded environments. In particular Lee *et al* examine prefetching mechanisms for GPGPUs [23] and Izraelevitz *et al* discuss how a policy of "always-abort" can improve performance for hardware transactional memory [24]. While these a works have a similar intent to the work presented here the specialized domains of GPUs and HTM respectively make their solutions hard to generalize to traditional shared memory CPUs.

2.2 Cache Compression

Cache compression has the potential to increase the effective capacity of the cache with extra area overhead [25]. Compressed caches can achieve the benefits of larger caches without making the cache physically larger by keeping more cache lines in the same cache space. Designing a compressed cache includes a compression algorithm and compressed cache layout. The compression algorithm represents a cache block with fewer bits, while a compaction mechanism determines how to read/write a compressed cache block to the cache [26, 27, 28, 29, 30].

Zero-value algorithms are limited to zero cache lines with low latency. Zero-Content Augmented cache design uses auxiliary cache to represent zero cache lines [31]. *Frequent Value Compression (FVC)* encodes frequent values present in cache lines with fewer bits [29, 28]. The downside of FVC, it only compress frequent values and cannot exploit other commonly found patterns [29, 28, 26]. *Pattern-Based Compression* [32, 33, 3] exploits the fact that many values are narrow (e.g., small integers) and can be represented using a small number (e.g., 4-8) of bits, but are normally stored in full 32-bit or 64-bit words. Base-Delta-Immediate (BDI) compression exploits spatial value locality, i.e., the observation that values that are spatially close in memory also tend to have small differences in their values [26]. BDI represents a block using one or more base values and an array of differences from the base values [26].

Typical cache stores one tag address to track a fixed-size cache block. The corresponding tag is used to perform hit detect for the cache block. In order to track more blocks, compressed caches are required to store more tags, which result in more area, power. Simple cache design double the number of tags (i.e., 2x Block Tags), allowing them to track up to twice as many cache blocks in the cache [32]. Decoupled Compressed Cache (DCC) [21, 22] and Skewed Compressed Cache (SCC) [25, 34] use more complex designs to reduce the number of extra tags, by exploiting spatial locality and use super-block tags to track more blocks and reduce the extra tag overhead.

2.3 Load Criticality

Cache optimization techniques such as data prefetching and cache compression cache have an impressive impact on performance and effectively minimize the impact of the performance gap between the execution core and memory subsystem. Even though the processor is still exposed to the memory wall on a cache miss due to hard to prefetch loads. Modern Out-of-Order microprocessors employ large instruction queue to exploit instruction-level parallelism. Program criticality can be described using a data-dependency graph; there exists one or more paths through the graph with a maximum length; such a path is called a critical path [35]. Instruction along that path called criticality instructions. A critical load is a load instruction that is on the critical path. Due to variable latency when executing load instruction based on the cache level that is responsible to source the target data, non critical load instructions have a big impact on the execution time when cache misses take place. In a system with data prefetcher, these loads expose the processor to long latency because they hard-to-prefetch.

3. SYNCHRONIZATION-AWARE HARDWARE PREFETCHING FOR CHIP MULTIPROCESSORS *

Shared-memory, multi-threaded applications often require programmers to insert thread synchronization primitives (*i.e.* locks, barriers, and condition variables) in critical sections to synchronize data access between processes. Scaling performance requires balanced per-thread workloads with little time spent in critical sections. In practice, however, threads often waste time waiting to acquire locks/barriers, leading to thread imbalance and poor performance scaling. Moreover, critical sections often stall data prefetchers that mitigate the effects of waiting by ensuring data is preloaded in core caches when the critical section is done.

This chapter introduces a pure hardware technique to enable safe data prefetching beyond synchronization points in chip multiprocessors (CMPs). We show that successful prefetching beyond synchronization points requires overcoming two significant challenges in existing techniques. First, typical prefetchers are designed to trigger prefetches based on current misses. Unlike cores in single-threaded applications, a multi-threaded core stall on a synchronization point does not produce new references to trigger a prefetcher. Second, even if a prefetch were correctly directed to read beyond a synchronization point, it will likely prefetch shared data from another core before this data has been written. This prefetch would be considered "accurate" but highly undesirable because it would lead to three extra "ping-pong" movements due to coherence, costing more latency and energy than without prefetching. We develop a new data prefetcher, Synchronization-aware B-Fetch (SB-Fetch), built as an extension to a previous single-threaded data prefetcher. SB-Fetch addresses both issues for shared memory multi-threaded workloads. The novelty in SB-Fetch is that it explicitly issues prefetches for data beyond synchronization points and it distinguishes between data likely

*Reprinted with permission from "SB-Fetch: Synchronization aware hardware prefetching for chip multiprocessors" by L. M. AlBarakat, P. V. Gratz, and D. A. Jimenez 2020. Proceedings of the 34th ACM International Conference on Supercomputing, Copyright 2020 by Association for Computing Machinery.

and unlikely to incur cache coherence overhead. These two features are directly synergistic since blindly prefetching beyond synchronization is likely to incur coherence penalties. No prior work includes both features.

SB-Fetch is evaluated using a representative set of benchmarks from Parsec [36], Rodinia [37], and Parboil [38]. SB-Fetch improves execution time by 12.3% over baseline and 4% over best of class prefetching.

3.1 Introduction

The scaling of computer systems through the final CMOS process technology generations poses a grand challenge for the computing industry. Despite increasing transistor density, performance and power gains that traditionally accompanied process scaling have largely ceased. This trend has manifested in the current proliferation of chip-multiprocessors (CMPs) replacing single core processors as the dominant processor design, due to their lower power consumption for similar performance, however, blithely scaling core counts with future process technologies will quickly lead to diminishing returns, particularly for shared-memory, multi-threaded applications. In these applications, core and thread-count scaling often leads to performance destroying workload imbalances [39, 40]. One of the major causes of these thread-level workload imbalances, as well as degrading performance in general, is memory latency.

Prefetching is a well-studied technique to reduce the impact of memory latency. Prior work has shown that prefetching produces substantial performance gains on typical single-threaded and multi-application workloads [4, 41, 13, 22]. Unfortunately, multi-threaded applications typically see little to no performance benefit from existing prefetching schemes. Figure 3.1 shows the speedup of multi-threaded applications under a previously proposed prefetching scheme [12, 13]. The figure shows that, at best, the performance improvement of the previous scheme is marginally positive, and at worst performance is significantly degraded despite evidence that several are memory bound [40, 36][2]. There are two main

[2]On single-threaded, multi-programmed workloads, B-Fetch sees an average gain of 31%[13].

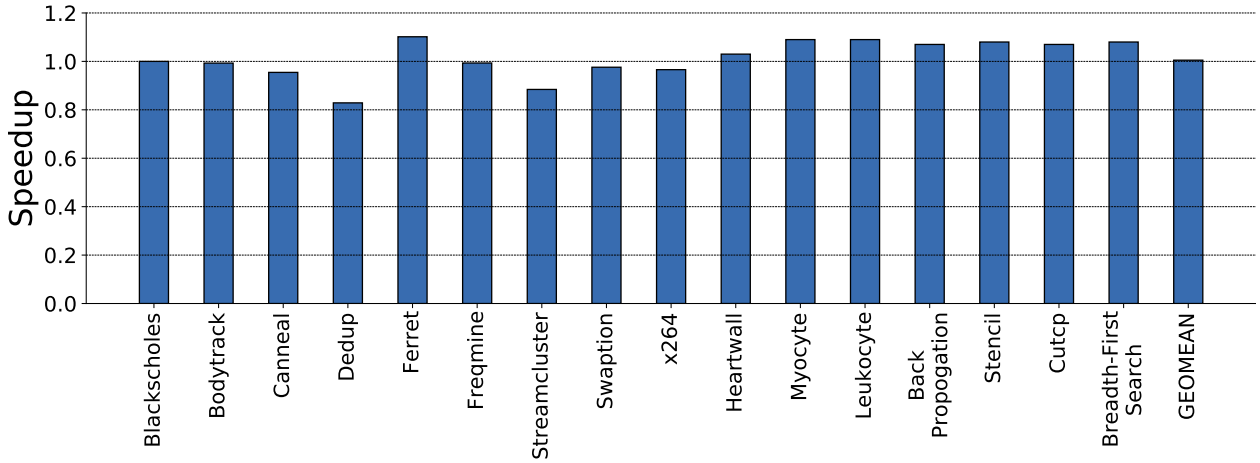


Figure 3.1: Speedup of Parsec [36], Rodinia [37], and Parboil [38] workloads with *B-Fetch* [12, 13], normalized against a no-prefetching baseline.

reasons for the poor performance of traditional prefetching techniques on these workloads: First, most prefetchers only issue a prefetch when a cache miss occurs in that core. In multi-threaded applications, the ideal time to pre-load the cache is while a given thread is waiting on thread synchronization. This represents a significant wasted opportunity because thread synchronization primitives contain no (relevant) cache misses.

Second, for those few prefetchers that issue prefetches without a triggering miss (*e.g.* B-Fetch [12, 13]), prefetching shared data, even with perfect accuracy, might incur excess invalidations in the event that the prefetched data is read before it is written in the producing core. This is the primary cause of B-Fetch's performance loss in the figure. No prior work we are aware of has identified and addressed these two issues in prefetching for multi-threaded applications.

Here, we present Synchronization-aware B-Fetch (SB-Fetch), a data prefetching scheme designed for prefetching shared-memory, multi-threaded workloads. This work makes the following contributions:

- This is the first work we are aware of to characterize the causes of poor prefetching performance in shared-memory multi-threaded applications. These are the inability to prefetch beyond synchronization points and tendency to prefetch shared data before it

has been written.

- Building upon this characterization, we propose SB-Fetch, a low-complexity, low-overhead prefetcher design that addresses both issues.

- We show that SB-Fetch provides a speedup of 12.3% over baseline and 4% over best of class prefetching [4, 21].

3.2 Motivation and Background

3.2.1 Data Prefetching

Data prefetching is a well known technique in which the cache is pre-filled with useful data ahead of an actual demand load request coming from the processor. Typically, the prefetching opportunity is limited to waiting until a cache miss occurs, and then reading either a set of lines sequentially following the current miss [16], a set of lines following a strided pattern with respect to the current miss [17], or a set of blocks spatially around the miss [4]. More recent prefetchers attempt to predict complex, irregular access patterns [4, 18, 19, 20, 21, 22]. While these methods show significant benefit, they are inherently reactive, waiting until a cache miss occurs before they initiate prefetches down the speculated path.

Some prefetchers, such as B-Fetch [12, 13], are triggered by the fetch of a branch instruction by the processor, making them more suitable for prefetching beyond synchronization points as we will discuss. B-Fetch is a data cache prefetcher that employs two speculative components. It speculates on the expected path through future basic blocks, using a lookahead mechanism that relies on branch prediction to predict that execution path, and a scheme to predict the effective addresses of load instructions along that path based on the register file transformations per-basic block. B-Fetch records the variation of register contents at earlier branch instructions and uses this knowledge to predict the effective address.

Some recent prior work has examined the case of prefetching in specialized multi-threaded environments. In particular Lee *et al* examine prefetching mechanisms for GPGPUs [23] and Izraelevitz *et al* discuss how a policy of "always-abort" can improve performance for hardware

transactional memory [24]. While these a works have a similar intent to the work presented here the specialized domains of GPUs and HTM respectively make their solutions hard to generalize to traditional shared memory CPUs.

3.2.1.1 B-Fetch Microarchitecture

Since B-Fetch is one of the few prefetchers that explicitly speculates down a future path of execution and that does not wait for a memory reference to miss before it starts prefetching, we will use it as a basis for the work in this chapter. Thus we here present a recap of the B-Fetch microarchitecture as originally published [13, 12].

Program construction can be mapped into a control flow graph as shown in Figure 3.2. As shown in the figure, the outcome of each branch determines which basic blocks will ultimately be executed. In the figure, there are three possible such paths, highlighted as Ⓐ, Ⓑ and Ⓒ. In each case, which loads are issued is directly dependent upon the path taken through the code as shown. Further, the particular effective addresses themselves are dependent upon the path taken through the code, as each basic block causes transformations to the data in the register file as execution continues.

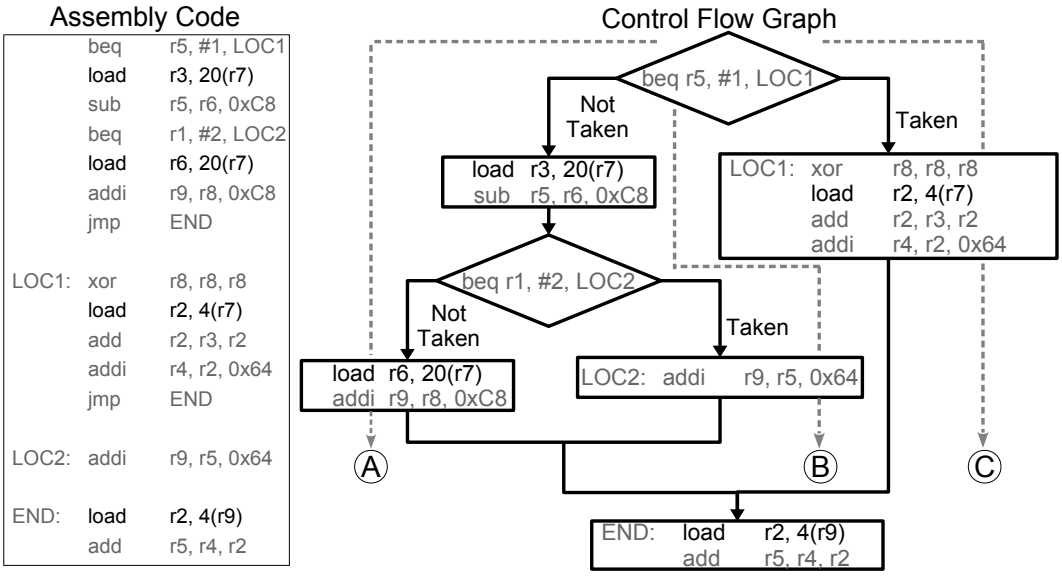


Figure 3.2: Data Access and Control Flow.

B-Fetch uses a lookahead mechanism that predicts the likely path of execution starting from the current non-speculative branch and issues prefetches for the memory references down that path. B-Fetch relies on the idea that register values at the time of effective address generation are correlated in a predictable way from their corresponding values at a time when their preceding branch instructions were executed and the transformations that occur to them over the course of the blocks to that point. Figure 3.3 shows the overall system architecture of a B-Fetch together with an out-of-order processor core. It shows the main core execution pipeline and the auxiliary hardware for B-Fetch preftecher.

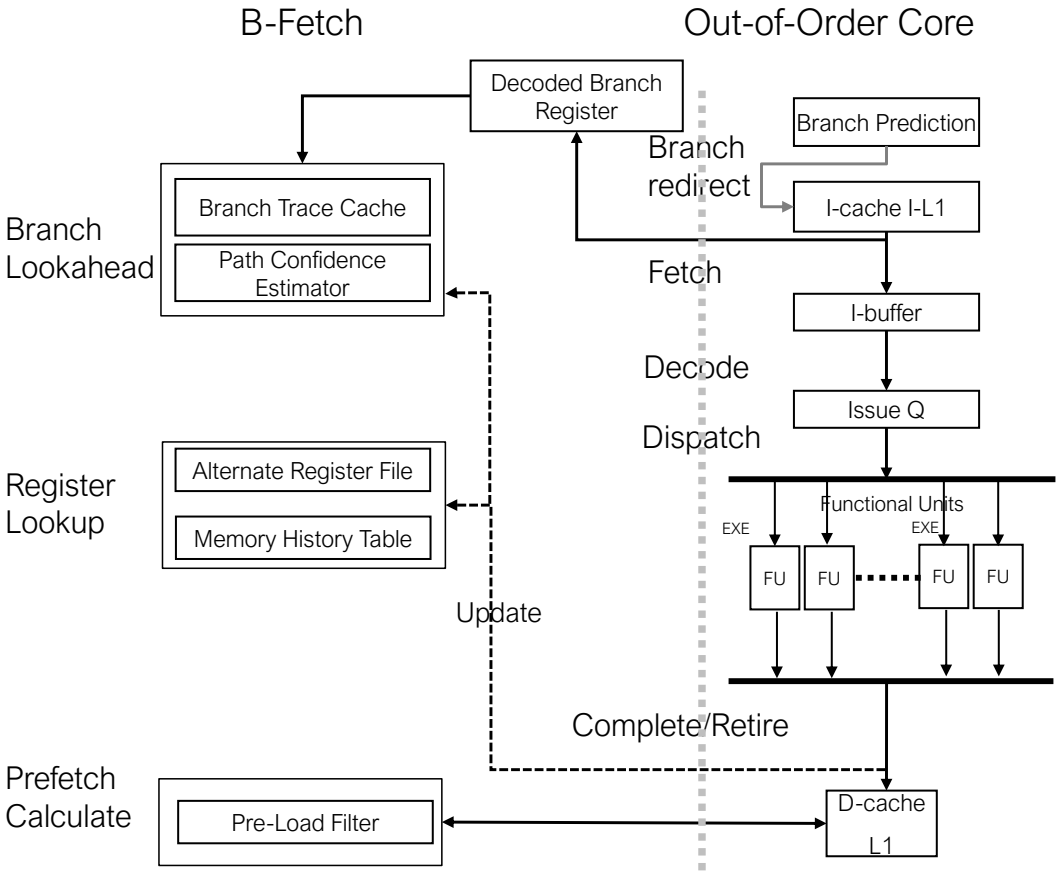


Figure 3.3: B-Fetch microarchitecture.

B-Fetch is composed of a 3-stage pipeline that runs parallel to the core pipeline. The

Decoded Branch Register (DBR) connects B-Fetch to the cores's Fetch stage. When a branch instruction is decoded in the main execution pipeline, the PC of the branch instruction is added to the DBR. This branch PC and target address starts the prediction of the future execution path, memory instructions, and their effective addresses.

Here we describe each of the major microarchitectural components of B-Fetch and their purpose.

Branch Lookahead Stage: This stage is similar to the fetch stage in the main pipeline. The duty of this stage is to generate the speculative exception path from the currently decoded branch. This stage includes two main components. First, the *Branch Trace Cache* that traces the branch instructions in the dynamic instruction stream. This is used to create set of pointers in the program control flow marked by branch instructions, allowing the prefetcher to skip the branch instructions in between. By doing so, the branch trace cache help guide the lookahead stage forward and the branch predictor and target buffer to help maneuver it in the right direction. The second component is *Path Confidence Estimator* that is used to throttle speculation in the event that the cumulative branch predictions to this point are not confident. This component prevents the issuing of useless prefetches and cache pollution.

Register Lookup Stage: This stage retains information about the registers which produce loads in each basic block to generate effective addresses for the given block. This stage includes two main components. First, the *Alternate Register File (ARF)* maintains a copy of the register file contents for use in generating predicted prefetch effective addresses. To ensure timely updates to the ARF, a copy of execution stage generated register values is used to perform updates. The second component is the *Memory History Table (MHT)* that maintains source register indices, current register values, and offset values to calculate effective addresses for prefetch candidates.

Prefetch Calculate Stage: This stage is responsible for generating the prefetch addresses that are issued to the prefetch queue. It synthesizes the data from the MHT and ARF to

produce a stream of predicted future memory references. This stream is then passed through the *pre-load filter* that keeps track of the issued but useless prefetches on a per-load basis. Loads found to typically produce useless prefetches are prohibited from producing a prefetch.

We note that it is beyond the scope of the current work to discuss the full details of the previously published B-Fetch microarchitecture, for that we point the reader towards the prior work [13, 12]. That said, we would like to point out that B-Fetch requires relatively little state (∼12KB) and relatively low hardware complexity (a handful of tables and some adders) to achieve accurate and high coverage on traditional workloads. Importantly, unlike other prefetching techniques, B-Fetch provides a direct mechanism for speculating upon the future path of the program and leveraging that speculation to issue prefetches, without the overhead of running the full core ahead, as in runahead execution [9]. Thus, we feel that B-Fetch makes an ideal starting point for attempting to efficiently issue prefetches beyond synchronization points.

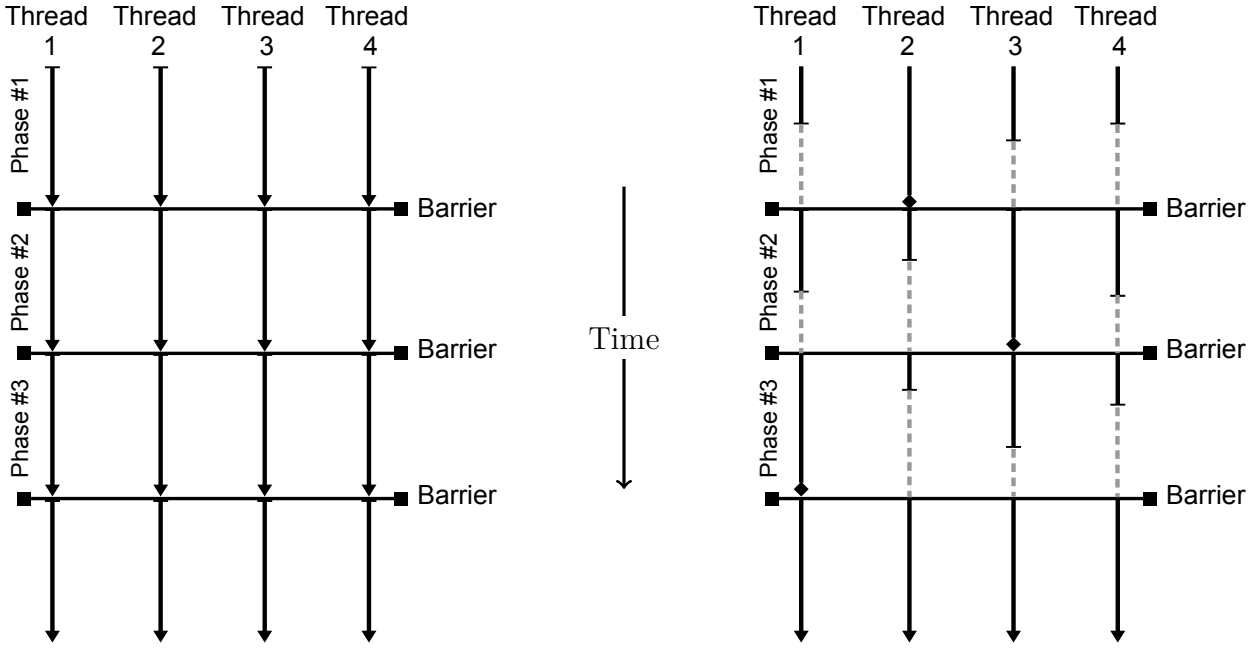
3.2.2 The Shared Memory Model

With growing core counts, fully exploiting the underlying microarchitecture and achieving scaling performance of single applications requires dividing that application into independent threads that can run simultaneously across the cores within a system and take advantage of thread-level parallelism (TLP). The dominant programming model for this form of TLP is shared memory multi-threading. In this programming model, an application is broken into independent threads that share a single, coherent view of memory. Typically, these independent threads share some data to complete the task. In this model, programmers insert explicit thread synchronization primitives (*i.e.* locks, barriers, and condition variables) to coordinate data sharing between threads, ensuring that data produced by one thread is not read by a consuming thread before it is written and so forth.

3.2.2.1 Shared Memory Synchronization

Synchronization is a central operation in parallel applications. The two major forms of explicit synchronization operations in shared memory multiprocessors are barriers and locks. A barrier used to ensure no process within a group cooperating processes can move beyond a certain point in the execution before all processes have reached the barrier. Barriers are commonly used to enforce such waiting.

Figure 3.4a illustrates how a barrier works. A task executes its code until it reaches a barrier. Then it waits until all other tasks have reached that barrier before proceeding. Ideally, all tasks start at the same time and reach the barrier at the same time, then start new phase of execution.



(a) Ideal barrier.

(b) Critical threads.

Figure 3.4: Ideal barrier synchronizes and critical threads in the execution phases.

A thread is critical if its progress determines the progress of the whole application and

forces other threads to wait for it. Due to load imbalance between threads, different threads can be critical during execution. As Figure 3.4b shows other threads need to wait untill the critical thread get to the barrier before resume execution. A synchronization barrier can lead to performance degradation. The slowest thread prevents forward progress of other threads and forces other threads to wait on the barrier. The performance of synchronization barriers in shared memory is often unpredictable and a performance bottleneck.

3.2.2.2 Architectural Support for Shared Memory Synchronization

To facilitate the construction of synchronization primitives, most architectures provide some form of read-modify-write instructions that are capable of updating (*i.e.*, reading and writing) a memory location as an atomic operation. For example, RISC style ISAs, such as Arm and ALPHA, support Load Linked (LL) and Store Conditional (SC) instructions to implement synchronization primitives [42, 43]. In this scheme, the LL instruction loads a block of data into the cache and marks this cache line for tracking. The following SC instruction attempts to write a new value to the same block. This write succeeds only if the block has not been referenced since the preceding LL. Any memory reference to the block from another processor between the LL and SC pair causes the SC to fail. Upon failure, the locking thread will typically retry the full LL/SC pair until atomic read/modify/write success is achieved.

For a thread to acquire the lock, it needs to load the lock and check if no other thread is holding the lock. After that it needs to own the lock. If the thread fails to acquire the lock, it will stay in a spin loop until it successfully acquires it. Once a thread acquires the lock it is safe to execute the critical section. Upon entering the critical section, only then is it "safe" to write or update data shared between threads because only one thread may enter the critical section to modify that data at any given time. Once this shared data is written, the thread then releases the lock to allow other threads to execute the critical section, and modify the shared data as well. Acquiring and releasing a lock involves executing primitive instructions.

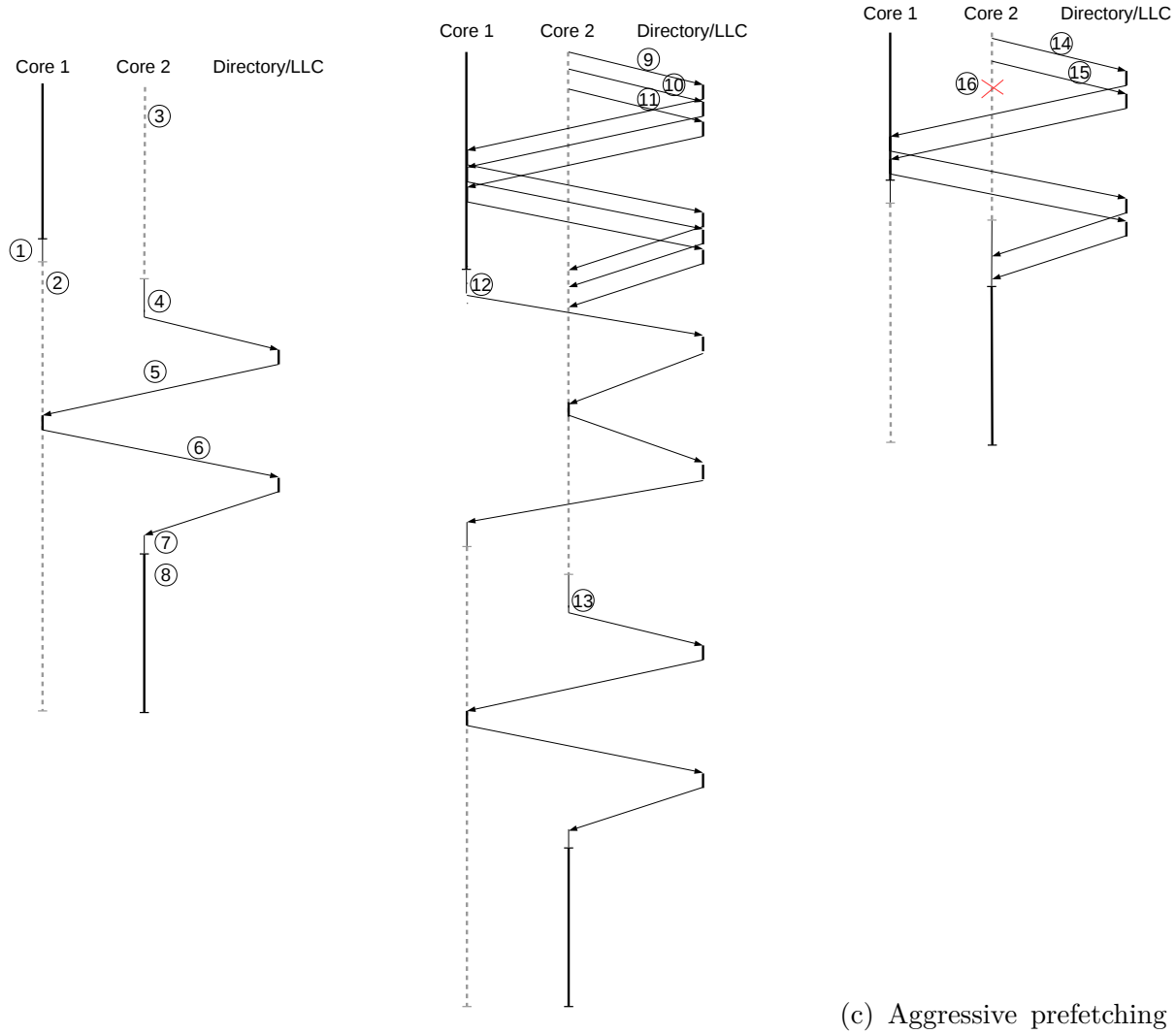
We note that CISC ISAs typically employ single read/modify/write atomic instructions that produce similar behavior in implementing shared memory synchronization semantics. For the purpose of discussion we focus on the LL/SC but our approach can be easily applied to CISC ISAs. In Section 3.3.2 we briefly discus the changes required to support CISC ISAs.

3.2.3 Cache Coherence

Cache coherence is the hardware mechanism by which shared data in different cores' private caches are kept coherent. Many coherence schemes have been proposed [44, 45, 46, 47, 48, 49]. One commonly used approach is a directory based cache coherence scheme. In this scheme, a directory, typically co-located with the shared, last-level cache, maintains the sharing state of all the cache lines in the individual cores' private caches. In such a scheme, when a core issues a request to acquire or change the state of a cache line in its private cache it must send a message to the directory. The directory then may need to send messages to the other cores' private caches, waiting for their acknowledgment before finally replying back to the requesting core. This transaction incurs a significant latency [50, 51, 52, 53].

Figure 3.5a illustrates the typical case for the sharing of data in a cache coherent shared memory system. The figure shows two threads communicating through shared data, synchronized by a lock. In the figure, core 1 first enters its critical section (①). In this case the cache line containing the shared data with the lock variable happens to be in the local private cache and the critical section finishes quickly. Note that, at ② and beyond, the cache line containing the lock variable will remain in the private cache of core 1, while core 2 executes, prior to its critical section. Here, at ③, core 2 is spinning, waiting for the write of this shared data. Once core 1 has completed its critical section, core 2 is free to enter the critical section and access the data in question. Since this data is on a cache line in core 1's private cache, a request is made to the directory for a shared copy, ④. At ⑤, this request leads to a writeback request to core 1. At ⑥, core 1's private cache issues a writeback to the LLC of the line containing the shared data. Finally, at ⑦, this line is then forwarded to core 2 and core 2's critical section can proceed (⑧). All of this back and forth between

the caches, directory and LLC, can incur hundreds of cycles of overhead at exactly *the most critical time in the execution of a multi-threaded application.*



(a) No-prefetching case.

(b) Premature prefetch of shared data.

(c) Aggressive prefetching of data while avoiding premature prefetches.

Figure 3.5: Cache coherence and prefetching.

3.2.3.1 Prefetching in Multithreaded Workloads

Multithreaded applications are just as likely to experience lost performance due to long-latency memory accesses as traditional, single threaded applications. Thus, prefetching should be a good way to improve performance. As discussed in the previous section, prefetching for multi-threaded applications produces unique challenges, in that threads waiting on synchronization typically do not induce prefetches for data beyond those synchronization points. Moreover, reckless prefetching of data beyond synchronization points could hurt performance due to premature prefetching of shared data before it has been written.

In the first case, where prefetching does not occur past synchronization points, there is a great lost opportunity for performance gain. As we see in Figure 3.5a, at point ③, core 2 is effectively idle waiting for core 1 to finish its critical section. If prefetching of data that core 2 will need after this synchronization point could be performed, it would be a great opportunity to leverage available, unused memory bandwidth in core 2. However, if core 2 is overly aggressive and prefetches shared data before it is written, the performance impact could be greater than the benefit of correct, on-time prefetches.

Figure 3.5b illustrates the danger of overly aggressive prefetching in this case. Here again, at the beginning of execution, the cache line containing shared data to be written by core 1 is currently residing in core 1's private cache, along with several other cache blocks that core 2 will need, but will not be written in core 1's critical section. At the beginning of the trace, while core 2 is idling, the prefetcher in core 2's private caches issues three prefetches, ⑨-⑪. While two of these prefetches, ⑨ and ⑩, are to data that ultimately will not be re-written in core 1, one prefetch, ⑪, has not yet seen its final write in core 1. Later, core 1 enters its critical section to write shared data to one of the cache line that was already prefetched (at ⑪), inducing another coherence transaction at ⑫ to retrieve ownership of the shared cache line core 2 just prefetched. Once the cache line has been retrieved from core 2, core 1 can safely write the new data and release its lock, at which point core 2 can enter the critical section, ⑬, and attempt to read the shared data core 1 just wrote. This incurs yet another

coherence transaction pulling this data back to core 2 again. As the figure illustrates, here prefetching actually incurs a large negative performance impact versus a baseline case where prefetching does not occur.

This resultant cache block transitioning back and forth between the cores due to overly aggressive prefetching has a much worse impact on multi-threaded applications than traditional single-threaded applications. In particular, while the impact of a bad prefetch on a single threaded application implies some wasted bandwidth, energy and some cache pollution; in multi-threaded applications, this extra latency often occurs exactly when the threads in question are literally "critical" to performance, in that they are the only threads executing during a mutex in their critical sections. Slowing down these critical sections has a significantly outsized influence on application performance. We empirically determined that these extra writebacks and invalidations account for the performance loss shown for several benchmarks using B-Fetch in Figure 3.1.

As shown in Figure 3.5c, ideally, one would like the idle cores to aggressively prefetch data while spinning on a lock, leveraging the available time and bandwidth, and yet avoid prefetching specifically only that data that will eventually be written by other cores. In the figure we see that the two useful prefetches, (14) and (15) are allowed to proceed while prefetch (16) is squashed before issuing because (16) is predicted to be invalidated by a future write in core 1. As we see, accurate prefetching beyond synchronization primatives can lead to significant performance increases while preventing performance regression due to premature prefetching. This is the goal of SB-Fetch.

Prior works address prefetching for multi-threaded workloads. Jerger *et al.,* [54] presents a taxonomy that classifies the effects of multiprocessor prefetches. While this work suggests invalidation filtering schemes could improve performance, it provides no practical mechanism for such a scheme, nor does it discuss explicitly prefetching beyond synchronization points as SB-Fetch does. They show that, without explicitly prefetching beyond synchronization points, the benefit of invalidation filtering is marginal. Liu *et al.,* [55] and Panda *et al.,* [56]

both present schemes that attempt to tune prefetch aggressiveness depending upon the criticality of the thread (among other things). This interesting approach is somewhat orthogonal to SB-Fetch and likely could be used in combination with SB-Fetch. Preliminary work by Panda et.al, [57] proposes a hardware prefetching framework that studies and classifies L1 misses across all threads to generate L2 cache prefetches. In our preliminary work [58], we initially examined the feasibility of prefetching for multithreaded workloads. Here we expand upon this prior work.

3.3 Proposed Design

SB-Fetch addresses the two issues with prefetching for multi-threaded workloads. It must continue prefetching beyond the synchronization semantic while the thread itself is busy waiting. It must also avoid issuing prefetches for shared data before it has been written.

The insight behind SB-Fetch is to use the decode stage in the actual processor pipeline to dynamically track the synchronization primitives and identify when a thread is spinning on a lock. For a thread to acquire a lock, it must load the lock and check that no other thread is currently holding the lock. Then it must own the lock. If the thread fails in acquiring the lock, it will stay in a spin loop until it successfully acquires it. Once a thread acquires the lock it is safe to execute the critical section. The thread needs to release the lock to allow other threads to execute the critical section as well. Acquiring and releasing a lock involves executing the synchronization primitive instructions LL and SC described in Section 4.1.

3.3.1 Overview

SB-Fetch is an extension to the prior work *B-Fetch* prefetcher described in Section 3.2. To address prefetching beyond synchronization points, we must detect when a thread is trying to acquire and release a lock in the instruction stream, then feed the first branch instruction after releasing the lock to the B-Fetch engine to start prefetching. To this end, *SB-Fetch* monitors the synchronization primitive instructions, LL/SC, in the dynamic instruction stream. The prefetcher identifies when a thread is spin waiting by the decoding of LL instructions. It

then learns the backward branches following the LL instruction that are part of the spin once the LL/SC pair are successful and records these. Later, when this synchronization point is encountered again, the prefetcher will ignore the "correct" backward branch prediction to skip ahead of the synchronization point, allowing prefetch to continue in the region beyond the critical section.

To solve the second issue, prefetch invalidation due to premature prefetching, *SB-Fetch* keeps track of prefetches that are invalidated via the cache coherence mechanism prior to their use. This information is used to filter these "unsafe" prefetches, prohibiting them from being prefetched in the future.

Figure 3.6 illustrates the overall system architecture of a system incorporating *B-Fetch* together with the additional components needed to implement SB-Fetch. The figure shows the main CPU execution pipeline and the additional hardware for the *B-Fetch* prefetcher. We note that, in code that does not have synchronization, SB-Fetch will perform identically to B-Fetch, already one of the top performing prefetchers for single-threaded code [13, 12].

3.3.2 System Components

As previously described, Figure 3.6 shows the *SB-Fetch* microarchitecture. In particular two components, the Synchronization Primitives Trace Cache (SPTC) and Invalidation Filter are added to the original B-Fetch microarchitecture. Here we describe each.

Synchronization Primitives Trace Cache (SPTC): The SPTC dynamically captures the atomic primitives that were used to implement synchronization semantics. Each entry acts as a state machine to indicate the beginning and ending of each critical section encountered. Here, an LL instruction followed by a SC to the same effective address, indicates the beginning of a critical section. Once a second SC is detected, it indicates the end of a critical section. In SB-Fetch, the SPTC receives information from the decode stage in the Out-of-Order pipeline. Figure 3.7 shows an entry in SPTC. Each entry in the SPTC includes the lower 32 bits of the effective address and 2 state bits. An entry is installed in SPTC on the beginning of a critical section, then the entry becomes valid once a second SC is

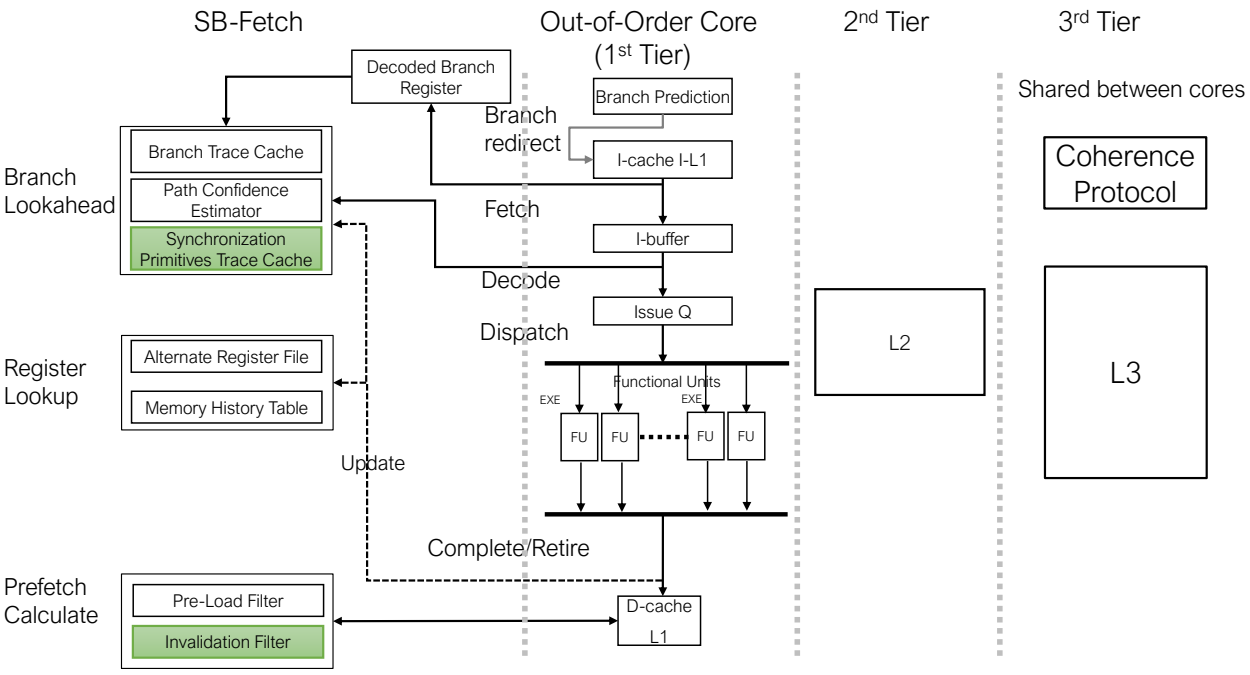


Figure 3.6: *SB-Fetch* microarchitecture. Additional components beyond *B-Fetch* highlighted in green.

detected, which indicates the end of the critical section. Then the first branch address after the critical section will be passed to branch lookahead component of the B-Fetch pipeline, so B-Fetch can predict the execution path starting from the current branch in order to prefetch data in the next basic block after the end of the synchronization semantic. The structures in B-Fetch/SB-Fetch pipeline are squashed and updated on commits.

Figure 3.7: Single Synchronization Primitives Trace Cache (SPTC) entry.

We note, as the SPTC is a multi-entry cache, it is possible to track many synchronization primitives at once, thus complex, multi-lock synchronization structures can be easily handled by this structure. We also note that while the above discussion revolves around the semantics

of LL/SC based synchronization, it would be actually somewhat easier to adapt the proposed mechanism to CISC ISAs that contain atomic read/modify/write mechanisms. In particular, instead of requiring monitoring for the sequence of an LL instruction followed by an SC, SB-Fetch would only need to monitor for the single atomic read/modify/write instruction itself.

Invalidation Filter: To prevent useless prefetches wasting time, bandwidth and energy, it is crucial to reduce the number of invalidations of data prefetched but never used in the local core. The Invalidation Filter tracks data recently prefetched from another core's private caches. In the event that a cache line prefetched from another core is invalidated prior to its use by the local core, the filter notes the associated load that caused the prefetch. Future prefetches associated with that load in that basic block will be dropped before issuing under the assumption that this load is likely to lead to a premature prefetch.

The invalidation filter consists of a table that keeps track of the prefetched cache block that was invalidated by the coherence protocol. The invalidation filter is indexed by a 10-bit hash of the load PC for the prefetch address. The invalidation filter has precedence over the branch confidence and per-load filter. That is, regardless of current branch confidence and per-load confidence, if a prefetch results in invalidation, we stop prefetching for the load PC that prefetch is predicted for.

The Invalidation filter by default will never un-learn that a given location is unsafe for prefetching. To allow for more adaptive behavior, we employ a simple, counter-based, random clear mechanism. The counter counts cycles up to a definable maximum, C_m. When this maximum is reached a single entry in the table is chosen to be cleared. Thus, the entire table is cleared every $C_m * k$ cycles where k is the size of the table.

3.3.3 Hardware Cost

The additional hardware storage requirements for SB-Fetch, B-Fetch, BOP and SMS are summarized in Table 3.1. Two additional components have been added to B-Fetch. In term of hardware budget Synchronization Primitives Trace Cache (SPTC) is 0.53125KB and the Invalidation Filter is 4.0KB. To optimize the performance of SMS, we used the configuration

Table 3.1: Hardware storage overhead in KB

Prefetcher	Component	# Entries	Size (KB)
B-Fetch	Branch Trace Cache	256	2.06
	Memory History Table	128	4.5
	Alternate Register File	32	0.156
	Per-Load Prefetch Filter	2048	2.25
	Additional Cache bits	-	1.37
	Prefetch Queue	100	0.51
	Path Confidence Estimator	2048	2
	TOTAL SIZE :		**12.84**
SB-Fetch	B-Fetch	–	12.84
	Primitives Trace Cache	128	0.53
	Invalidation Filter	1024	4
	TOTAL SIZE :		**17.37**
BOP	Recent Requests Table	256	6
	Additional Cache bits	-	4
	BO prefetcher state	-	0.8
	TOTAL SIZE :		**10.8**
SMS	Active Generation Table	64	0.57
	Pattern History Table	16k	36
	TOTAL SIZE :		**36.57**

used by Somogyi, *et al.* [41] and 2KB spatial regions, a 64-entry accumulation table, and a 16K-entry pattern history table. Thus, SB-Fetch incurs a small, 4.53125KB overhead over B-Fetch, which is still significantly less hardware state than SMS requires.

3.4 Evaluation

3.4.1 Methodology

We used gem5 [59], a cycle accurate simulator, to evaluate SB-Fetch. The baseline configuration is summarized in Table 3.2. We used a set of nine multi-threaded programs from PARSEC benchmark suite [60], four applications from the Rodinia benchmark suite [37], and three benchmarks from the Parboil benchmark suite [38]. These benchmark applications represent widely used shared memory applications that use the P-threads library to handle synchronization. The benchmark applications are cross-compiled for the ALPHA ISA and run on gem5 configured with the O3CPU CPU model (Out-of-Order) and the detailed

(classic) memory model. The benchmarks were run in Full System (FS) mode.

The baseline hardware is a 4-core CMP machine with three level cache hierarchy as specified in Table 3.2. Each core's private cache is split into I-cache (32KB) and D-cache(32KB), 256KB second level cache and 1024KB per core third level shared cache.

Table 3.2: Target Microarchitecture Parameters

Simulator	Gem5 Simulator, ALPHA ISA, Full System Simulation
Architecture	O3 processor, 4-wide, 192-entry ROB
ICache / DCache	32KB, 8-way set-associative
L2Cache	256KB, 8-way set-associative
Shared L3Cache	1024KB per core, 16-way set-associative
Memory	DDR3-1600 x64 channel, Micron MT41J512M8

SB-Fetch results are compared against four light-weight prefetcher designs: Stride, SMS, BOP and the original B-Fetch. In the cases of SMS and BOP, the code for these prefetchers as well as their configuration was directly adapted from their respective submissions to the First [61], and Second [62] Data Prefetching Competitions. We note that BOP was the winner of the Second Data Prefetching competition. The Stride prefetcher was configured as in prior work [13].

3.4.2 Results

3.4.2.1 Performance

Figure 3.8 shows the performance of each of the five prefetcher designs as the speedup compared to the baseline no-prefetching configuration. For all results, the execution time is the time spent in the region of interest (ROI). In the figure we see that SB-Fetch provides a significant performance increase across all benchmarks of 12.3% versus the baseline, beating the performance of the closest competitor, BOP 8.1%. Moreover, where the original B-Fetch showed performance regressions versus a non-prefetching baseline, SB-Fetch improves performance for every benchmark. Interestingly, SB-Fetch sees some of its biggest performance

gains for applications where the original B-Fetch saw significant performance losses. Given the speedup and the cost of storage overhead together, SB-Fetch presents a better solution for data prefetching in multi-threaded workloads. We also see that for each suite individually, SB-Fetch outperforms each of the competing techniques, largely by similar margins. This highlights the robustness of SB-Fetch's gains.

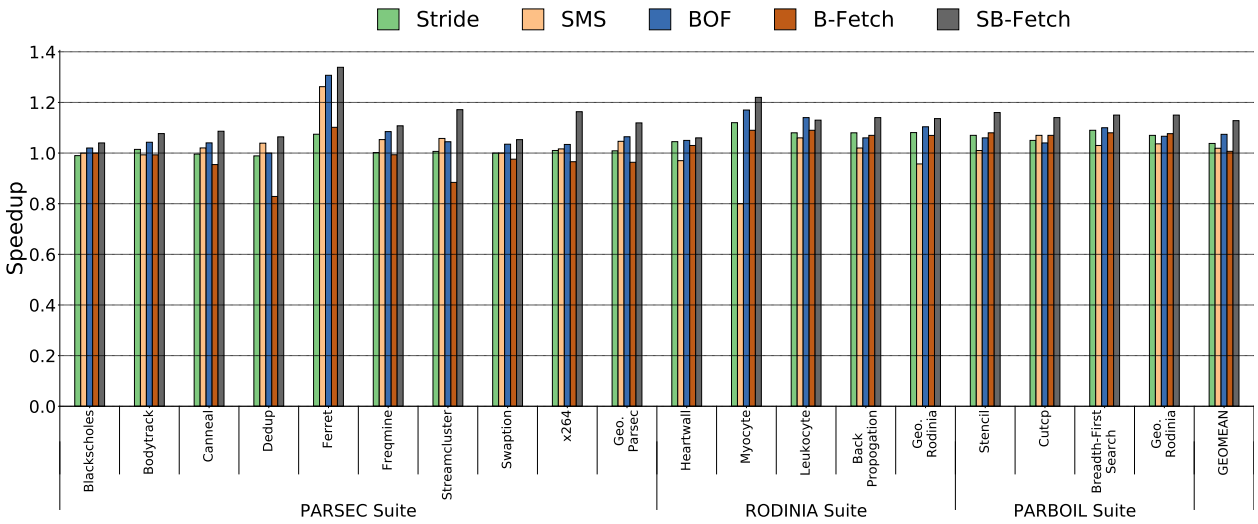


Figure 3.8: Multi-threaded workload speedups.

Figure 3.9 decomposes the benefits provided by different components of the proposed SB-Fetch. We observe that almost all benchmarks benefit to some degree from both techniques lock bypassing and invalidate filtering. For example, in ferret $\sim$ 50% of the benefit comes from lock bypassing and $\sim$ 50% from invalidate filtering. Meanwhile, x264 benchmark $\sim$ 21% comes from lock bypassing and $\sim$ 79% from invalidate filtering. We also note that each of the benchmarks that see significant performance losses for B-Fetch: dedup, streamcluster and x264, are also the benchmarks for which the invalidation filter provides a significant benefit. The matches the intuition that the cause of performance losses in these benchmarks is due to the impact of invalidations on critical thread execution.

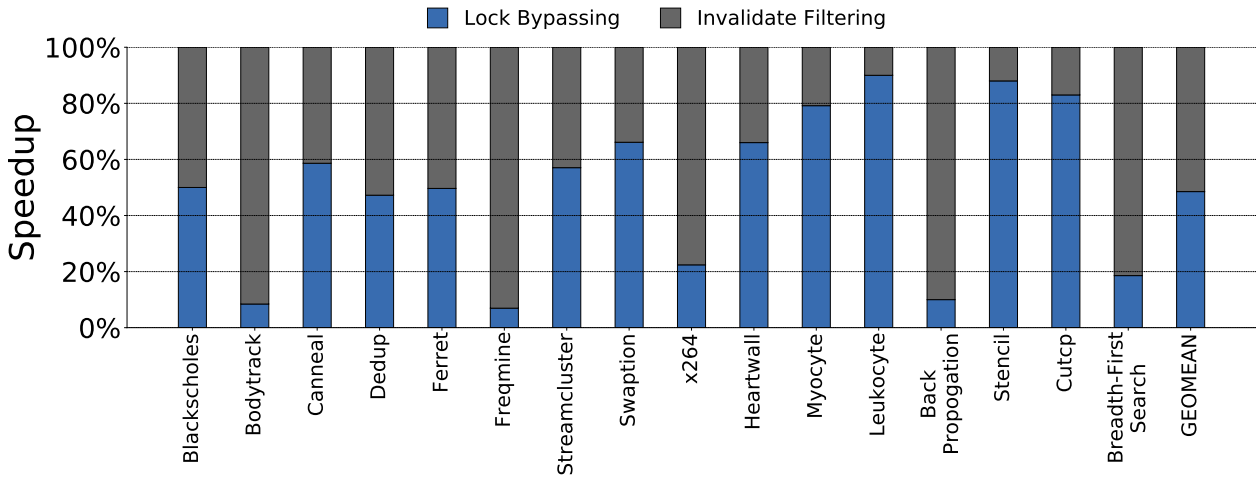


Figure 3.9: SB-Fetch speedup breakdown by mechanism.

3.4.2.2 Coverage and Accuracy Analysis

Figure 3.10 shows the number of useful versus useless prefetches for each prefetcher. Each bar is the arithmetic average across all benchmarks. The figure illustrates several points about the behavior seen in the performance results (Figure 3.8). First we see that, for the Stride prefetcher where very small performance gains are seen, generally few prefetches are issued, thus the performance gains are minimal. Interestingly for SMS, which sees some gains, there are actually fewer useful prefetches and more useless than even Stride. In this case, the useless prefetches were not enough to pollute the caches significantly, while there were more useful prefetches issued on the critical thread, thus more performance gains. BOP, which slightly outperforms SMS, appears to have nearly identical useful vs. useless prefetches. The original B-Fetch, while issuing slightly more useful prefetches than BOP, also issues more than twice as many useless prefetches. The original B-Fetch, often will get stuck in spin-lock loops, prefetching the lock cache line itself, which is not only useless but can cause worse performance because the lock cache line will have to be invalidated back to the core currently holding the lock. Further, for the occasions when B-Fetch does prefetch beyond the critical section (when it predicts the lock will not spin), B-Fetch often prefetches cache lines from other core's private caches before the writing core has written the data,

causing performance loss as the cache line ping pongs back and forth between the private caches. In the figure, we see that SB-Fetch, by contrast, successfully converts the majority of B-Fetch's useless prefetches into useful prefetches, this is the dominant reason why SB-Fetch outperforms the competition on these workloads.

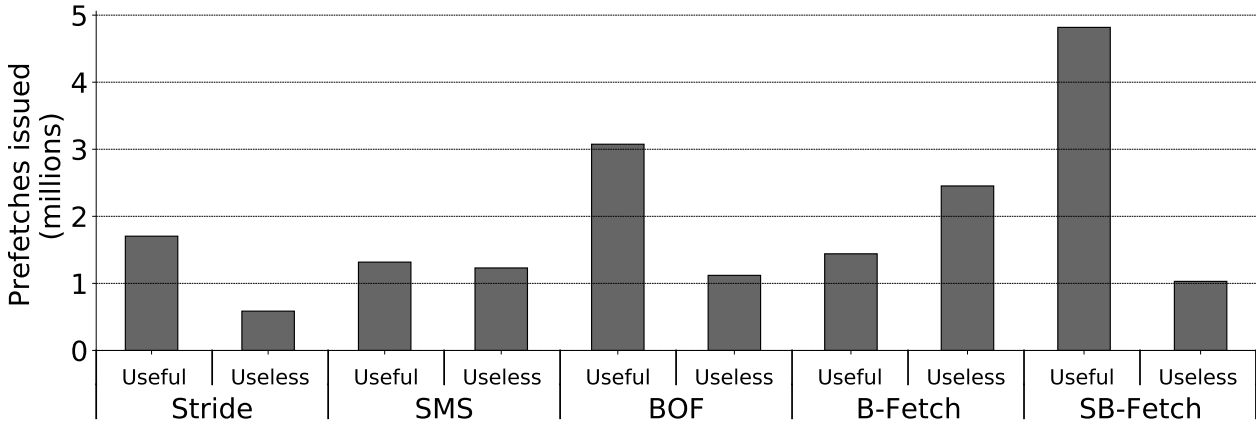


Figure 3.10: Useful and useless prefetches issued, averaged across all benchmarks for each prefetcher.

Figure 3.11 shows the percentage of invalidated prefetches for each prefetcher. Generally, we see that SB-Fetch has the lowest invalidation fraction, relative to the other prefetchers. Interestingly, we see that the rates of invalidation are actually quite low. We found that in part this is because the total number of prefetches issued can vary widely, with very few prefetches issued for blackscholes for instance.

Finally, Figure 4.10 examines the coverage for all prefetchers across all benchmarks. Here prefetching coverage is measured as the number of useful prefetches normalized to the number of data misses in the baseline configuration without a prefetcher. As shown in the figure, SB-Fetch achieves a geometric mean coverage of 35% which is the highest coverage across all benchmarks compared to the other prefetchers such as BOP that achieved a geometric mean of 23%.

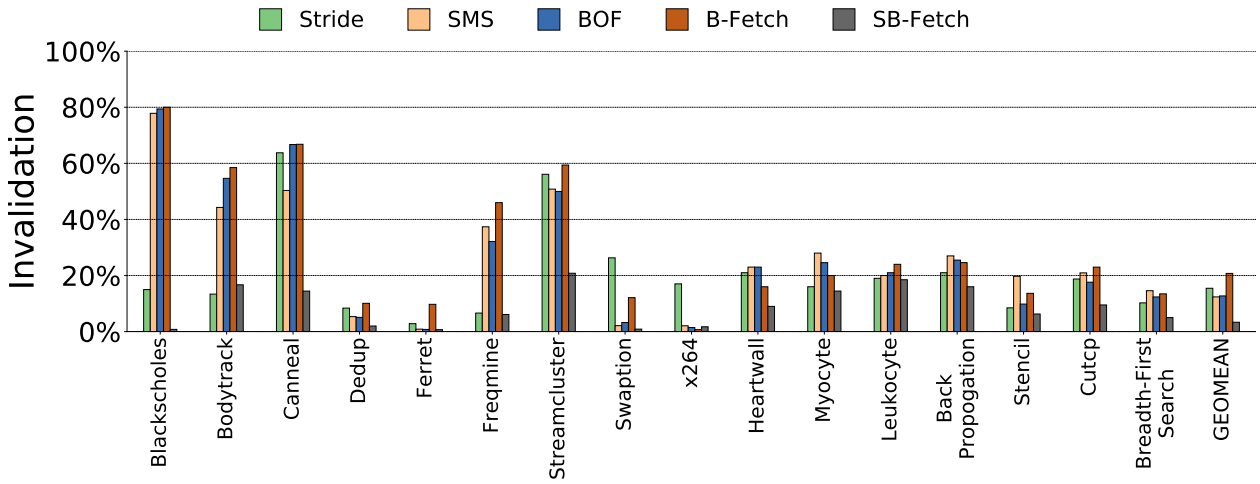


Figure 3.11: Invalidated prefetches for each prefetcher across all benchmarks.

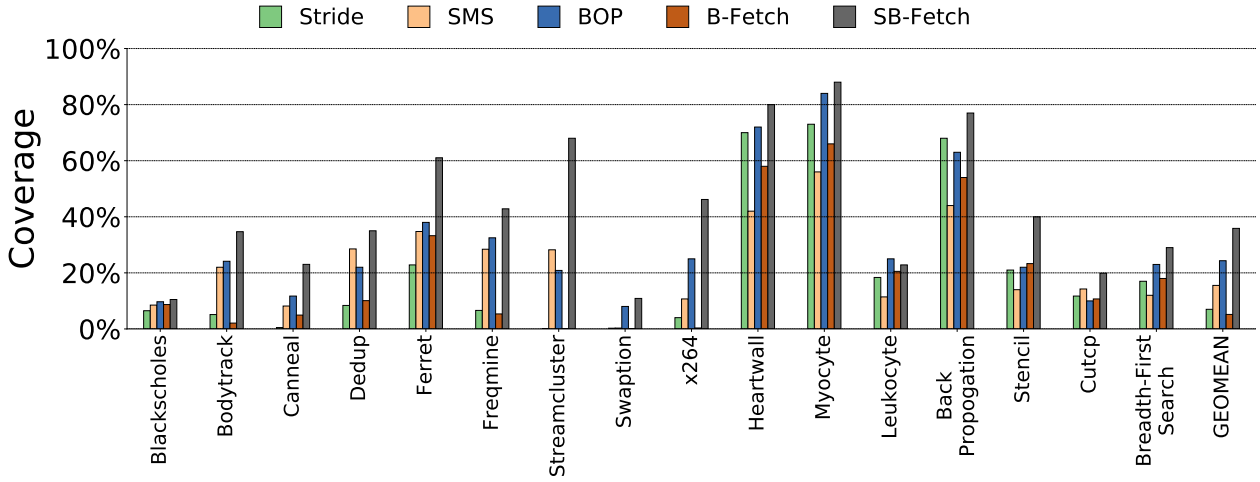


Figure 3.12: Coverage for each prefetcher across all benchmarks.

3.4.2.3 Sensitivity Analysis

This section provides a sensitivity analysis of SB-Fetch. We study the impact of different parameters and structures on the performance.

3.4.2.3.1 Invalidation Filter Size Figure 3.13 examines the impact of invalidation filter size on performance. This is the table that tracks cache lines which are invalidated due to coherence traffic. Here we see that generally SB-Fetch is highly insensitive to invalidation

filter size with only small gains seen as the filter grows.

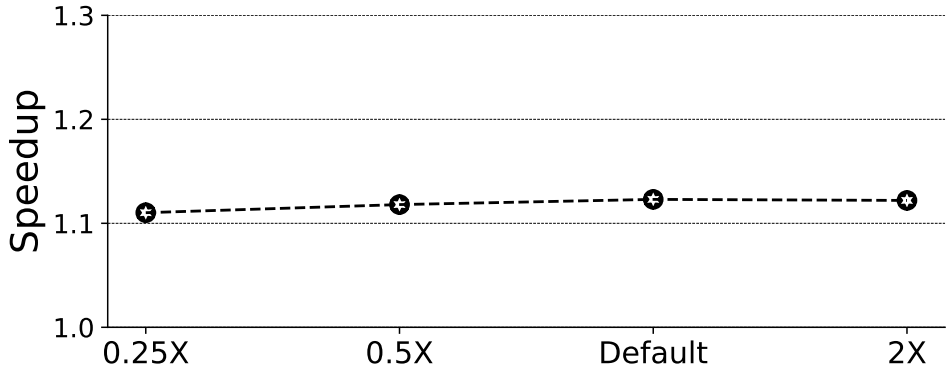


Figure 3.13: Invalidation Filter size sensitivity.

3.4.2.3.2 Synchronization Primitives Trace Cache Figure 3.14 examines the impact of invalidation filter size on performance. This cache tracks the beginning and ending of synchronization primatives so SB-Fetch can skip their branches. Similar to the Invalidation filter, SB-Fetch is highly insensitive to SPTC size.

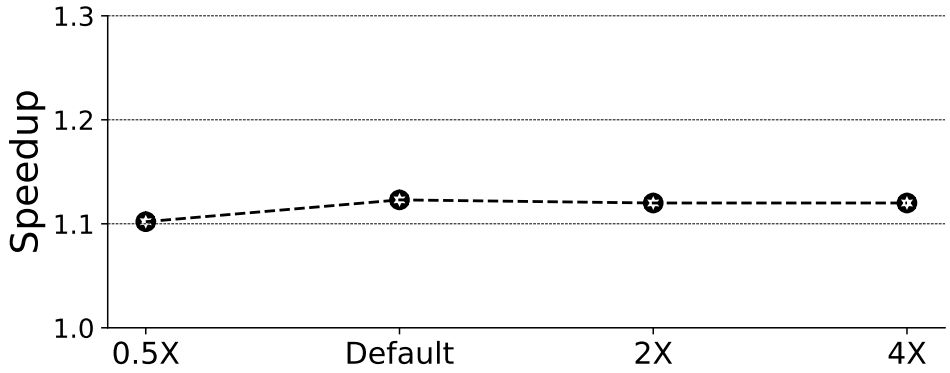


Figure 3.14: Synchronization Primitives Trace Cache sensitivity.

3.4.2.3.3 Branch Confidence Figure 3.15 examines the impact of the B-Fetch branch confidence threshold on performance. This confidence threshold throttles the aggressiveness of

the underlying B-Fetch prefetcher. Here we see that the best performance is achieved at the default .75 confidence. This value is the same as was default in the original B-Fetch.

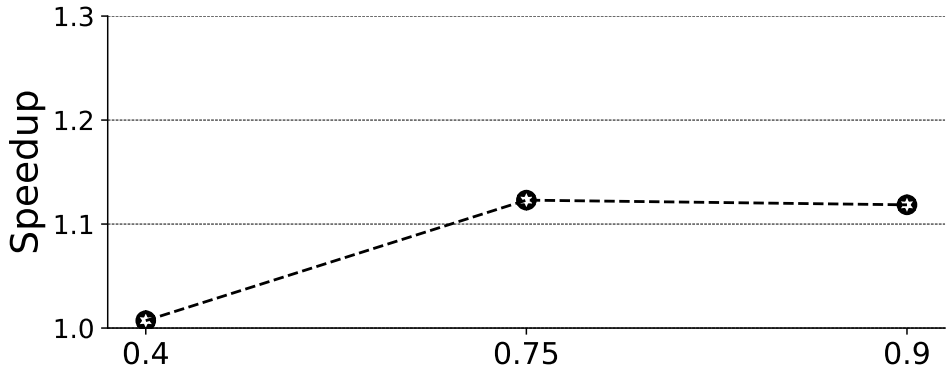


Figure 3.15: Branch confidence sensitivity.

3.4.2.3.4 Scalability Finally, figure 3.16 shows the performance scalbility for SB-Fetch going from 4 cores to 8 cores. For this number of cores SB-Fetch scales well, with an average performance increase from 12% to $\sim$ 16%.

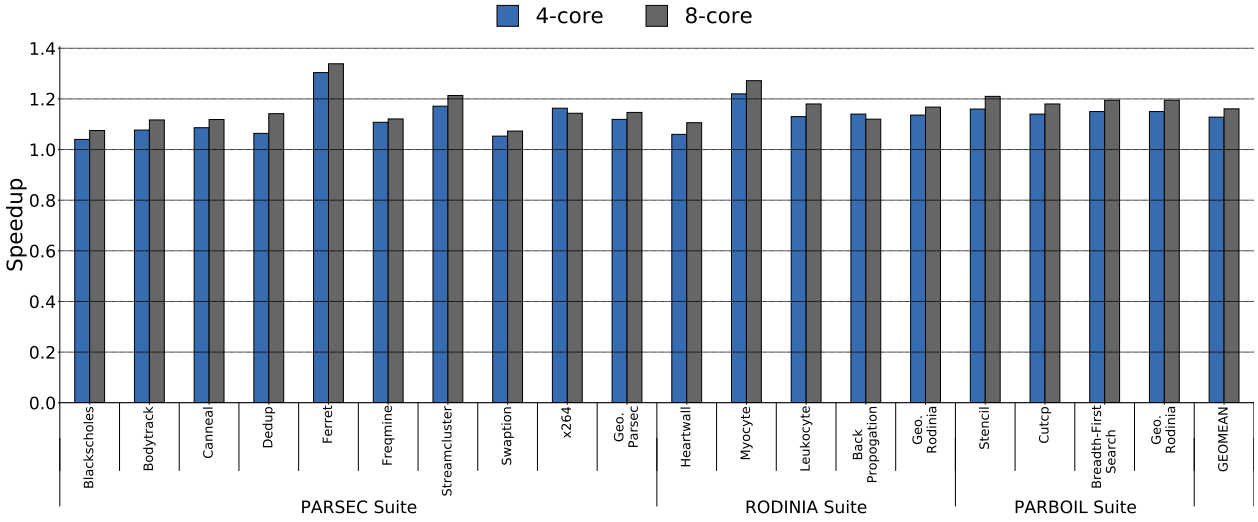


Figure 3.16: SB-Fetch speedup comparison for 4- and 8-core CMPs normalized to the baseline.

With increasing core-counts, shared memory multi-threading is becoming an ever more critical programming paradigm. Shared memory multi-threaded applications are similarly impacted by latency in the memory system as single-threaded applications, however, current memory prefetchers are unable to produce much performance benefit in these workloads. In this chapter we identify two primary causes for poor performance in existing prefetchers for multi-threaded workloads: the inability to prefetch beyond synchronization semantics and the premature prefetching of data before it has been written in the producing core when the prefetcher is able to prefetch beyond those semantics. We then show a low overhead technique which allows prefetching beyond synchronization semantics while avoiding prefetching of data which has not yet been written by its producing thread. This scheme, SB-Fetch, achieves a geometric mean speedup of 12.3% over baseline, more than twice the gains of the nearest competitor light-weight prefetcher on these workloads. As a final note, none of the proposed additions negatively impact the single thread performance gains seen in the proposed prefetcher.

4. SET-LEVEL ADAPTIVE PREFETCHING FOR COMPRESSED CACHES

Data prefetching and cache compression are well-studied techniques to reduce the impact of memory latency. Data prefetching predicts future memory accesses and prefills the cache with the corresponding memory blocks in advance of explicit demands. Cache compression tries to increase the effective capacity of the cache with minimum area overhead [63, 64, 26, 34, 65, 66, 67]. As we show, however, naïvely integrating the two techniques does not yield additive gains. In particular, we show that the extra ways per set that cache compression provides generally produce fewer hits than the ways in the baseline uncompressed cache. Hence, prefetching more aggressively to those sets with added ways due to compression and more conservatively to those sets with comparatively fewer ways can provide substantial benefit. In this chapter we present set-level adaptive prefetching for compressed caches (SLAP-CC), a compressibility aware prefetching technique that adapts prefetch aggressiveness to the workload compressibility to maximize prefetch coverage. SLAP-CC dynamically adjusts the prefetch confidence threshold on a per-set basis, based on the number of effective ways in a given cache set due to that set's compression. SLAP-CC achieves a geometric mean speedup of 18.0% over a baseline system with compressed cache and no prefetching and outperforms the state of the art prefetchers, SPP [68] and Best Offset [69] when combined with cache compression.

4.1 Introduction

High-performance processors include multiple levels of cache hierarchy within a single chip. In current processors from AMD, IBM, Intel, ARM and other vendors, the cache hierarchy makes up more than half of the die area of the processor [70, 71] and consumes a significant fraction of the on-chip power [72, 73, 74, 75, 76, 77]. This is because, with the large and growing latency differential between processor cores and DRAM, caches are critical for overall system performance [1]. Thus, performance gains in the cache hierarchy strongly

translate into system performance gains. Misses reduce caching benefits and have a strong impact on the performance of modern workloads, thus reducing cache misses is important. Increasing cache size can reduce cache misses by allowing data sets with greater footprints be cached, but at the cost of latency, area and power overheads, thus bounding the size of lower-level caches.

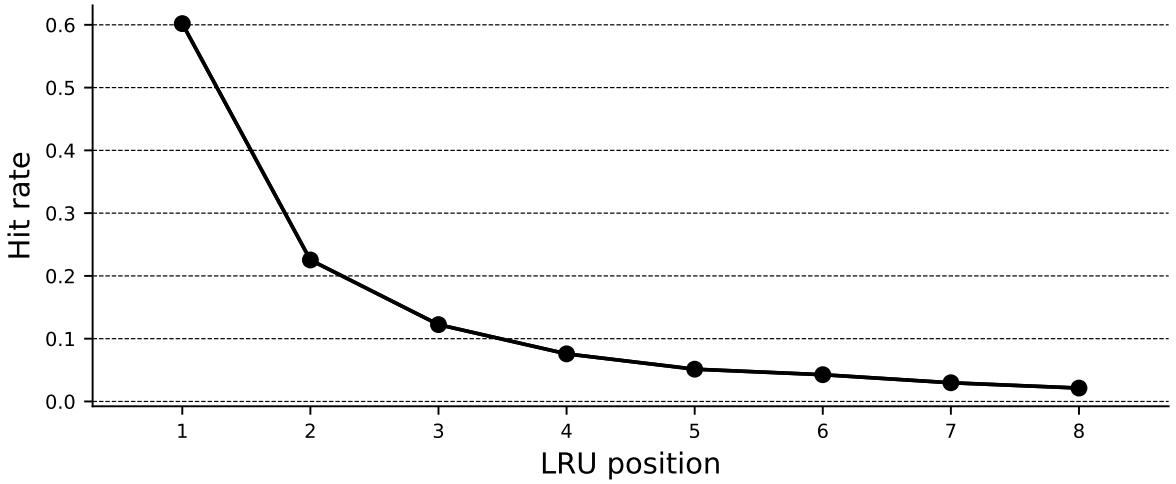
Two well-known techniques to reduce misses are cache compression and prefetching. Cache compression reduces misses by increasing the effective capacity of the cache without increasing the actual area/power of the cache [63, 64, 26, 34, 65, 66, 67]. We show, however, that cache compression can decrease cache efficiency (*i.e.* the amount of time a block remains live in the cache [78, 79, 80]), as the added ways in a given set are less likely to see hits over time than the ways that were there in the baseline cache. Further, cache compression incurs some extra latency on both miss fills for compression and hits for decompression. On the other hand, prefetching reduces misses by preemptively fetching data into the cache ahead of its use [17, 18, 20, 69, 68, 7, 81, 82]. However, inaccurate prefetches can push out useful data with prefetched data that will not be used. While prior work has shown that cache compression and prefetching can work well together [32, 33], we are aware of no work that has attempted to co-design the prefetching algorithm to better leverage the extra capacity cache compression provides. In this chapter, we propose Set-level Adaptive Prefetching for Cache Compression (SLAP-CC) with that goal in mind.

Cache compression is an optimization technique that has the potential to increase the effective capacity of the cache while reducing the area, power and timing overhead [26, 83, 84, 31, 34, 25]. The goal of cache compression is to reduce capacity misses by achieving a higher effective capacity. Thus, cache compression reduces costly off-chip accesses, translating into performance improvements [85]. The design of a compressed cache has two main components: a compression algorithm that leverages redundancies in the data values stored in each cache line to represent the data in compact form, and a compaction mechanism to store compressed blocks in the cache. While the extra capacity is beneficial, the benefit

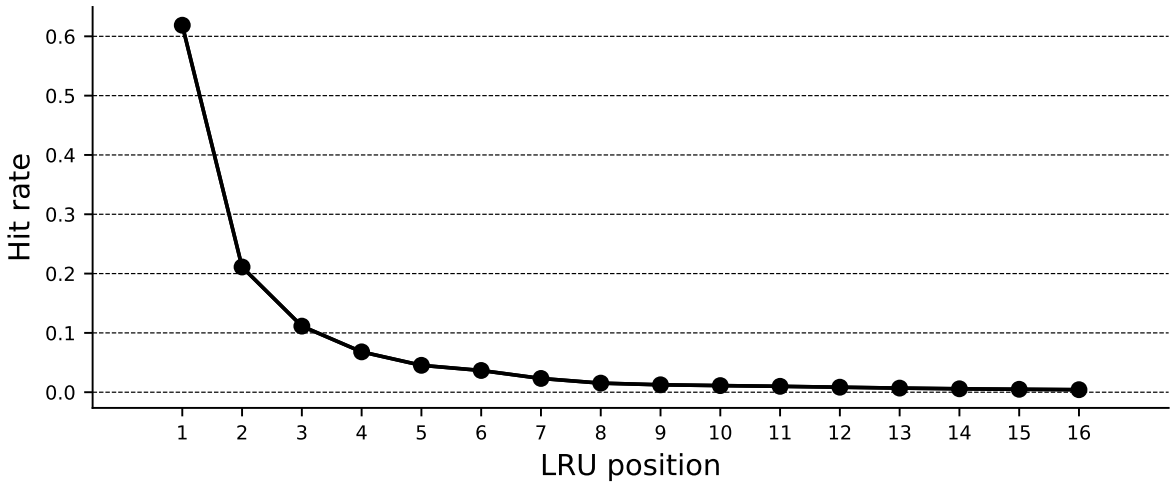
does not scale linearly with capacity. Moreover, the goal of reducing the latency of compression/decompression limits the scope of compression possible, with many practical cache compression proposals limiting capacity increases to at most 2x [26, 32]. As compression increases the effective capacity of the cache, the efficiency of cache decreases with capacity. Due to the phenomenon of compression locality, neighboring blocks often have similar compressibility [83]. Thus, cache compression varies greatly from one set to the next. Some sets can be heavily compressed while others cannot.

Data prefetching is a well-known technique in which the cache is pre-filled with useful data ahead of an actual demand load request from the processor. Many different prefetching techniques have been proposed over the years, leveraging various mechanisms to speculate on future patterns, ranging from a set of lines sequentially following the current miss [16], a set of lines following a strided pattern with respect to the current miss [17], or a set of blocks spatially around the miss [86]. While these methods show significant benefit, by their nature they can have variable aggressiveness in their speculation. A highly aggressive prefetcher is one which prefetches many cache lines in the hopes that some of them will be used, while a more conservative prefetcher might only prefetch those lines which it is certain will be useful in the future. While a more aggressive prefetcher has the potential to cover more of the misses of the machine, if the prefetching is too aggressive, it can lead to pushing out more useful data in the cache. More recent prefetchers [68, 7, 87] attempt to directly estimate how good their prefetches are explicitly via confidence and use that confidence to explicitly choose the right aggressiveness. However that confidence threshold is typically statically set at design time and is not adapted at runtime.

To examine the impact of compression on cache efficiency and per-way hit rates, Figure 4.1 shows the average L2 hitrate per way, sorted into an LRU stack for an uncompressed and an ideally compressed cache, where ideal compression provides double the capacity and number of ways per set. As the figure shows, the extra ways added by compression (*i.e.* ways 9-16) in the ideal case have a very low individual probability of receiving hits, relative



(a) Hitrate per way/LRU position in baseline, 256KB L2 8-way cache.



(b) Hitrate per way/LRU position in "perfectly compressed", 512KB L2 16-way cache.

Figure 4.1: Mean L2 hitrate per way under SPEC CPU2017 benchmarks, sorted into an LRU stack for a 256KB L2 cache of 8-ways - representing baseline; and a 512KB L2 cache of 16-ways - representing perfect compression.

the existening ways 1-8 in the LRU stack. Thus, we argue, given the low utility of the extra ways, more aggressive prefetching is warranted than in the baseline, uncompressed case. This is because, to achieve performance improvement, the utility (likelihood of use) of a given prefetch must be higher than the utility of the data it replaces in the cache. As the

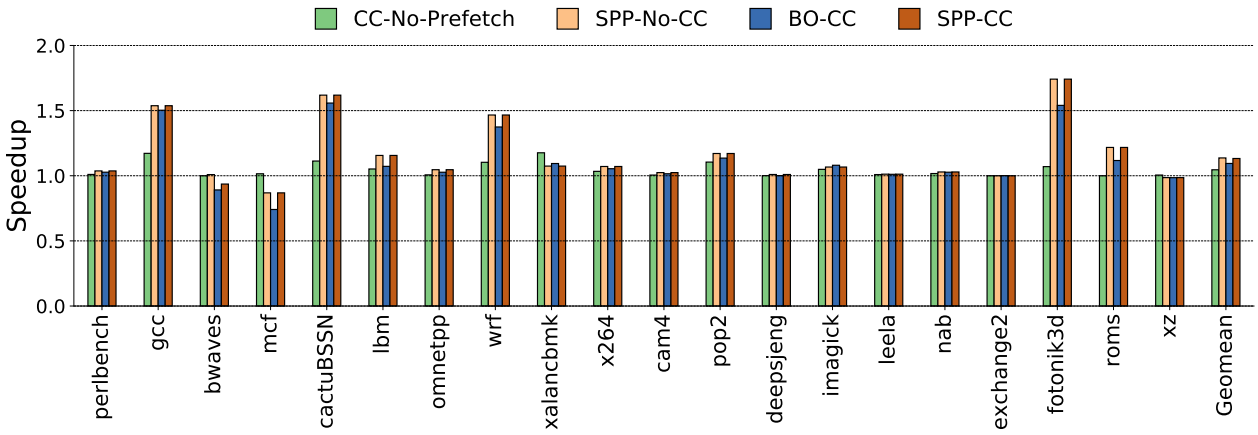


Figure 4.2: SPEC CPU 2017 Single-Core IPC Speedup with and without BΔI [26] cache compression (CC), the SPP [68] prefetcher and Best Offeset (BO) [69] prefetcher all normalized against no prefetching/no CC in the L2 cache.

compressed cache has data with less utility in the added ways, it will see higher benefit with aggressive prefetching than the baseline case. Our key observation, however, is that *because real (non-ideal) cache compression tends to vary greatly from set to set, the exact aggressiveness needed to achieve performance gains must also be varied on a per-set basis.* Although some prior work has studied how cache compression and prefetching interact [33, 88], we are aware of no prior work that attempts to vary prefetch aggressiveness on a per-set basis under cache compression.

Figure 4.2 illustrates the issue. In the figure we show the speedup of BΔI [26] cache compression alone, SPP [68] prefetching alone, Best Offset [69] prefetching + BΔI cache compression and SPP prefetching + BΔI cache compression (all in the L2 cache[1]), all normalized against a non-compressed, no-prefetching baseline system. The figure shows first that while cache compression provides some benefit versus baseline, the extra access latency keeps the gains from cache compression relatively modest at 5%. By contrast, standalone SPP prefetching improves performance significantly more than cache compression at 13%. Interestingly, combining both prefetching and cache compression actually significantly degrades performance on average versus prefetching alone. In most cases the prefetcher is not

[1]See Section 4.5.1 for full details of the methodology

able to effectively leverage the extra capacity enough to overcome the added latency of cache compression. Here we attempt to address problem by codesiging the prefetching algorithm to fit the cache compression scheme.

In this chapter we propose a new prefetching algorithm, Set-Level Adaptive Prefetching for Compressed Caches (SLAP-CC), which seeks to address this problem by individually confidence filtering prefetches based on how much effective capacity is available in the set the prefetch is destined towards. The individual contributions of this chapter are as follows:

- We characterize the increase and per-set variability of cache efficiency which typical cache compression schemes create.
- We propose a new prefetching scheme, SLAP-CC, designed to leverage this cache efficiency variability.
- SLAP-CC leverages different prefetching confidence thresholds based on the compression level of the set that the prefetch is destined towards, to determine if a prefetch should be placed in that set or not.

In simulation on SPEC CPU 2017 workloads we show that SLAP-CC with cache compression improves performance over a compressed cache baseline of 18%, outperforming non-codesigned prefetching schemes on the same compressed cache by more than 5%.

The remaining sections are organized as follows. Section 4.2 discusses the motivation and background for set-level adaptive prefetching. The design of SLAP-CC prefetching scheme is presented in Section 4.3. Section 4.4 describes the hardware implementation cost of SLAP-CC. A detailed performance evaluation is presented in Section 4.5. Finally, we conclude the chapter in Section 4.6.

4.2 Background and Motivation

In this section, we discuss related work to our proposed technique. We use this prior work to motivate the case for prefetching-compression co-design.

4.2.1 Cache Compression

Cache compression presents the potential to increase the effective capacity of the cache while minimizing the area, power and timing overheads. Typical compressed cache designs include a compression algorithm and compressed cache layout. The goal of the compression algorithm [26, 27, 28, 29, 30] is to represent a cache block with fewer bits, while cache layout determines how to read/write a compressed cache block to the cache. Cache efficiency for a given cache is defined as the fraction of time that the cache holds a live cache line, *i.e.*, the number of clock cycles that a cache line contains data that will be referenced again divided by the total number of clock cycles the cache line contains valid data [79]. Cache efficiency determines what fraction of the cache lines holds useful data, while the remaining fraction holds data that will not be referenced before eviction, therefore it can be replaced with useful data [80].

BΔI is a state-of-the-art low-overhead technique for compressing data in caches [26]. It takes advantage of the fact that data values stored within a cache block have a low dynamic range, *i.e.*, the fact that values that are in a cache line tend to have small differences in their values [26]. BΔI represents a cache block using one or more base values and an array of differences from the base values [26]. BΔI, exploits the low dynamic range of integers that do not require the full space allocated for them, *i.e.*, small integer values are sign-extended into 32-bit or 64-bit blocks, while all the information is contained in the least-significant bits [26]. Figure 4.3 shows an example of BΔI compression using a single base value. Having multiple base values can improve the effectiveness of BΔI compression. A typical implementation uses two base values: zero and the first non-zero value in the input cache line.

To perform compression using BΔI [26], each cache line is viewed as a set of fixed-size values *i.e.*, 8 8-byte, 16 4-byte, or 32 2-byte values for a 64-byte cache line. Then, it decides if the set of values can be represented in a more compact form as a base value with a set of differences from the base value [26]. On the other hand, to perform decompression operation BΔI takes the base value and an array of differences generate the corresponding set of values

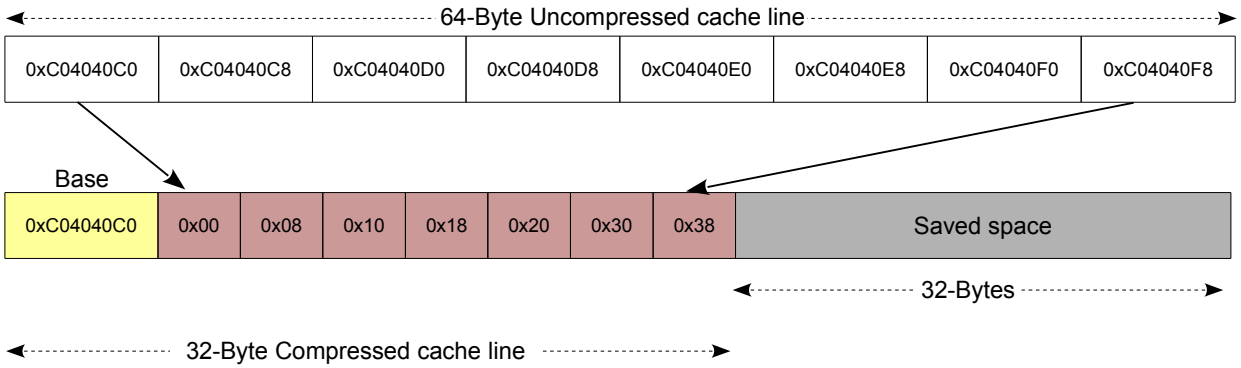


Figure 4.3: BΔI compression using one base value.

using a SIMD-style vector adder [26]. Therefore, the decompressed operation costs the same amount of time as an integer vector addition, using a set of simple adders. In practice, this implies 1-2 cycles added for compression and decompression.

4.2.2 Compressed Cache Layout

Compressed cache layout determines how to store and track more compressed cache blocks in the cache. To track more blocks in the cache, a compressed cache needs extra tags and metadata. A typical cache stores one tag for each fixed-size data block to detect hits. A compressed cache, where additional data blocks are stored in the data array, requires increasing the tag array. Alameldeen *et al.* [32], propose a simple cache design that doubles the number of tags, *i.e.*, 2X block tags, allowing to track up to twice as many cache blocks in the cache. Decoupled Compressed Cache (DCC) [63, 64] and Skewed Compressed Cache (SCC) exploit spatial locality and compression locality to pack neighboring blocks with similar compressibility in the same data entry, tracking them with one sparse super-block tag [34].

DCC [63, 64] has three main data structures: a Tag Array, a Sub-Blocked Back Pointer Array, and a Sub-Blocked Data Array. All three structures are indexed using the super-block address bits, so all blocks of the same super-block are mapped to the same data set [63, 64]. Decoupling sub-blocks from the address tag to eliminate expensive re-compaction when a

block's size changes [63, 64]. SCC [34] compacts blocks into a variable number of sub-blocks to reduce internal fragmentation, but retains direct tag-data mapping to find blocks quickly and eliminate extra metadata.

4.2.3 Dead Block Prediction and Replacement

Another way to improve the miss rate is to increase the number of live blocks in the cache through an improved replacement policy based on dead block prediction [89]. A cache block is live if it will be referenced again before its eviction. From the last reference until the block is evicted the block is dead [78]. Dead blocks contribute to poor cache efficiency when a cache block stays in the cache for long time after the last access and before it is picked by the replacemt policy for eviction [80, 79]. In case of least-recently-used (LRU) replacement policy and highly associative cache, a cache block has to move from the most-recently-used (MRU) position to the LRU position and then it is evicted [89]. Cache compression has the potential to increase the number of effective ways in a cache set [63]. Thus, the time for a cache line to move to LRU from MRU will increase, resulting in more dead blocks in the cache. In this case aggressive prefetching is a technique that increases the number of live blocks by prefetching less accurate prefetches to the cache. Recent work has significantly improved prediction for cache replacement [90, 91, 92], and taken into account prefetching along with prediction [93, 94]. Dead block prediction is orgthogonal to our proposal and are beyond the scope of this work.

4.2.4 Data Prefetching

Data prefetching is a speculative technique that prefills caches with useful data in advance of expected demand. Typically, prefetching techniques are activated on a cache miss. The prefetcher issues a prefetch request to a set of lines sequentially following the current miss [95], a set of lines following a strided pattern with respect to the current miss [15], or a set of blocks spatially around the miss [82]. Offset prefetchers such as Best-Offset (BO) prefetcher [69], the winner of the second Data Prefetching Competition [62], find the offset from the current

access or miss that maximizes useful prefetches. Best-Offset continues to perform prefetching with a selected offset till a new offset performs better than the current offset [69].

Some recently proposed prefetching techniques such as B-Fetch [7], SB-Fetch [81], SPP [68] and PPF [87] have confidence based self-throttling mechanisms. These self-throttling techniques avoid prefetching useless data by preventing lookahead down a wrong speculative path [7, 81, 68, 87]. These prefetch control mechanism keeps track of the confidence value of the path as a whole and when this confidence falls below a predetermined threshold value, the prefetcher is stopped. Usually, the magnitude of the threshold value is statistically predetermined and applies to all prefetches regardless of the workload, target set and oblivious to the current cache efficiency.

Here we aim to leverage the confidence value produced by the prefetcher in a fine-tuned, per-set, placement based filter. For the sake of simplicity we will use SPP [68] as the basis for our proposed technique as it already generates a path confidence that we can modify and use to achieve our goal. Nevertheless, any prefetcher that generates some form of prefetch using probability or locality measure could be modified to be used here. As we will focus on SPP, we present SPP in more detail next.

4.2.4.1 Signature Pattern Prefetcher (SPP)

SPP [68] is a state-of-the-art confidence-based multi-target prefetcher. It creates a signature associated with a page address by compressing the history of accesses [68]. The signature consists of up to four recent consecutive address deltas observed in a 4KB page [68]. Delta in this context refers to the arithmetic difference in effective address between two subsequent accesses to the same page. Each signature stores up to four memory access patterns in a compressed format, and a prefetch confidence value corresponds to each prefetch candidate [68, 87, 96]. To generate a new prefetch candidate, each prefetch candidate delta is XOed with a shifted version of the candidate delta's signature [68, 87, 96]. The new signature is used to re-index the signature table to generate a new prefetch candidate. When the candidate of the prefetch candidate above a predefined threshold value, a prefetch request

is issued to that cache hierarchy. While the basic idea of set-level adaptive prefetching is applicable to any confidence-based prefetcher, we develop a practical implementation of our proposed prefetcher using SPP as our underlying mechanism. Here we describe the basic architecture of SPP.

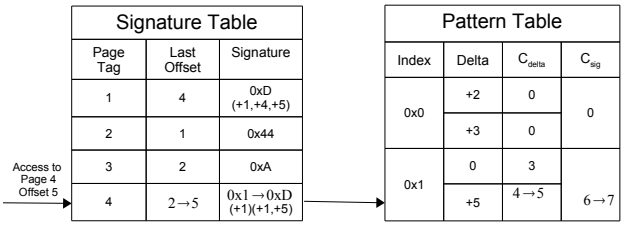


Figure 4.4: SPP Data-path Flow.

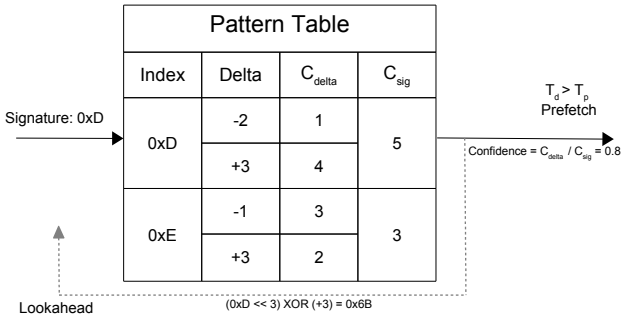


Figure 4.5: SPP lookahead prefetching.

Signature Table: Figure 4.4 shows the Signature Table that keeps track of 256 most recently accessed pages [68]. Upon a cache miss the Signature Table is indexed using the page number. Each entry in the Signature Table stores a "last block offset" and an "old signature" [68, 87]. The Last block offset corresponds to the block offset of the last memory access of that given page [68, 87]. The block offset is calculated with respect to the page

boundary. The signature is a 12-bit compressed representation of the past few memory accesses for that page [68, 87]. The signature is calculated as:

$$NewSignature = OldSignature << 3bits\ XOR\ Delta$$

Delta is the numerical difference between the block offset of the current and the previous memory access [68, 87]. In case a matching page entry is found, the stored signature is retrieved and used to index into the Pattern Table [68, 87]. This process is illustrated in Figure 4.4.

Pattern Table: As shown in Figure 4.4 the Pattern Table is indexed by the signature generated from the Signature Table [68, 87]. Pattern Table holds predicted delta patterns and their confidence estimates [68, 87]. Each entry indexed by the signature holds up to 4 unique delta predictions [68, 87].

Lookahead Prefetching: SPP performs recursive lookahead to generate prefetch candidates on the speculative path. On a cache miss SPP use the current prefetch as starting point to re-index the Pattern Table to generate prefetch candidates [68, 87]. As illustrated in Figure 4.5, SPP re-index the Pattern Table and updates the signature based on highest confidence prefetch from the last iteration. The prefetch 'depth' is the number of iterations on which SPP manages to predict prefetch candidates in the lookahead manner [68, 87]. While doing so, SPP also keeps compounding the confidence ineach depth [68, 87]. Thus, as depth increases, overall confidence keeps decreasing [68, 87].

Confidence Tracking: As shown in Figure 4.5, the Pattern Table keeps track of two counters for each delta value, C_{sig} which is the number of occurrences of each signature and C_{delta} which is the number of occurrences for a given delta per signature [68, 87]. To calculate the confidence for a given delta is approximated through $C_d = C_{delta}/C_{sig}$.

$$P_d = \alpha.C_d.P_{d-1}$$

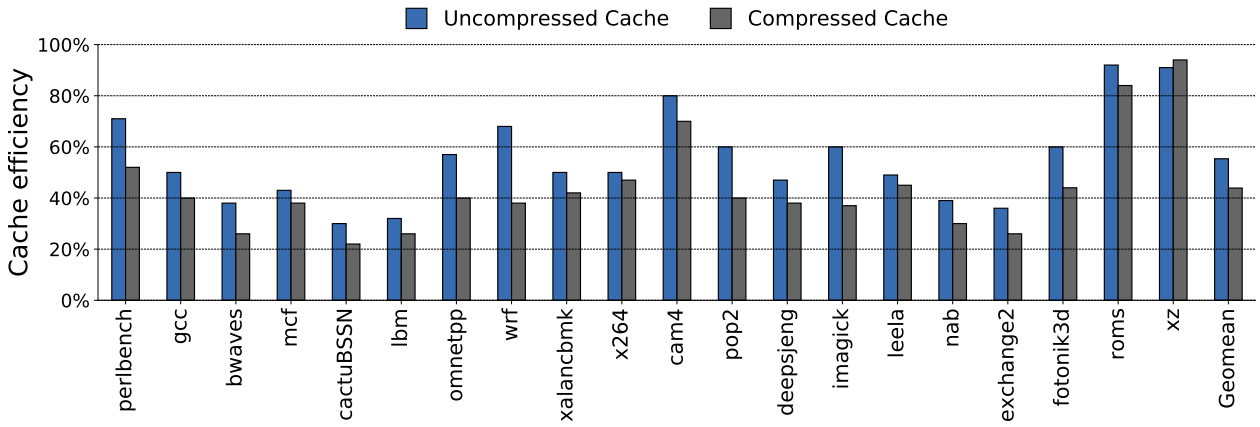


Figure 4.6: Cache efficiency.

SPP uses a global accuracy scaling factor α, which is the ratio of the number of prefetches which led to a demand hit to the number of prefetches recommended in total [68]. The scaling factor is used to throttle down or increase the aggressiveness of the lookahead process [68, 87]. The range of α is [0,1] [68, 87]. The lookahead depth is represented by d [68, 87]. The path confidence P_d is thresholded against prefetch threshold (T_p) to reject the low confidence suggestions and then against a numerically bigger fill threshold (T_f) to decide whether to send the prefetch to $L2$ Cache (high confidence prefetch) or Last Level Cache (low confidence prefetch) [68, 87].

4.2.5 Compression and Prefetching Interactions

Some prior work has studied the how cache compression and prefetching interacts. Alameldeen and Wood [33] showed that compression and prefetching can interact in strongly positive ways. They propose an adaptive prefetching mechanism that enables prefetching for the whole cache whenever beneficial. They use extra tags already provided for fitting more compressed blocks to also detect useless and harmful prefetches. In their compressed cache, they double the number of tags to potentially track twice the number of compressed blocks. However, in many cases, not all the blocks are compressible, so there are extra tags not being used. They leverage these tags to track recently evicted blocks and to find whether

prefetched blocks were evicting useful ones. They use a saturating counter that they incremented on useful prefetches, and decremented on useless or harmful prefetches. Using this counter, they disable prefetching when it does not help. Overall, they show that by leveraging the interaction between compression and prefetching, they can significantly improve performance.

Patel *et. al* [88], propose a synergistic cache compression and prefetching technique. The goal of this technique is to use the cache space saved by cache compression to implement the storage arrays required by data prefetching engine [88]. Charmchi, et al. [97] proposed Compressed cache Layout Aware Prefetching to builds on top sector-based compressed cache layouts to create a synergy between compressed cache and prefetching, by prefetching contiguous cache lines that can be compressed and co-allocated together with the missing cache line [97]. Building on a key observation that most workloads manifest spatial locality where neighboring blocks tend to simultaneously reside in the cache, and compression locality where neighboring blocks often have similar compressibility [83]. Thus different cache sets exhibit different compressibility. Highly compressed sets allocate a large number of compressed blocks.

In LRU managed compressed caches a highly compressed set exhibits less efficiency due to the increase in the time it takes a block to move from the MRU position to the LRU before being evicted. A confidence based prefetcher can take advantage of this observation by dynamically adjusting the magnitude of the threshold to issue less accurate prefetch candidates. Such a prefetcher will improve performance by increasing the coverage since part of the prefetch candidates turn to be useful prefetches and satisfy demand requests. Moreover, it improves the set efficiency by reducing the overall dead time of a cache line as dead blocks are more quickly moved out of the cache.

Figure 4.6 shows that cache blocks in an uncompressed cache are dead on average 41% of the time, for the benchmarks used for this study. When using compressed cache the ratio worsens, cache blocks are dead on average 52% of the time. Cache compression hurts cache

efficiency by increasing the fraction of time a block is dead. Replacing dead blocks with live blocks soon after a block becomes dead improves cache efficiency. Aggressive prefetching to a cache set that includes dead blocks or a highly compressed set can reduce the fraction of time a block is dead by speeding up the time to evict it. Having more live blocks in a set improves the hit rate which translates to performance improvement.

To the best of our knowledge, no previous work has tried to explicitly co-design a prefetcher for a compressed cache. Cache compression provides more capacity, so it is logical that this extra capacity could be used to hold more speculative prefetches. Traditional cache compression techniques, however, leave each set with a variable effective capacity increase per set. Thus, the correct degree of prefetching aggressiveness that can be sustained without performance loss in compressed caches actually depends on the degree of compression in that set.

4.3 Design

To improve on prior approaches to prefetching and with compressed caches outlined in Section 4.2, we propose the Set-level Adaptive Prefetching for Compressed Caches (SLAP-CC), a novel, low-overhead and accurate lookahead prefetcher designed specifically for the variability in ways available per set in compressed caches. Here we first give a basic design overview, followed by a detailed breakdown of the cache compression and prefetching technique.

4.3.1 Design Overview

Figure 4.7 shows the general structure and overview of the proposed SLAP-CC system. The major components of SLAP-CC are a cache compression system in the L2 cache and the SLAP-CC cache compression aware prefetcher. Here we focus on compression in the L2 cache, though SLAP-CC could be easily extended for L1 or LLC compression. We would argue that the L2 cache is the sweet spot for compression. Unlike the L1 which is highly timing sensitive, the L2 hit time is slow enough that added latency for decompression has

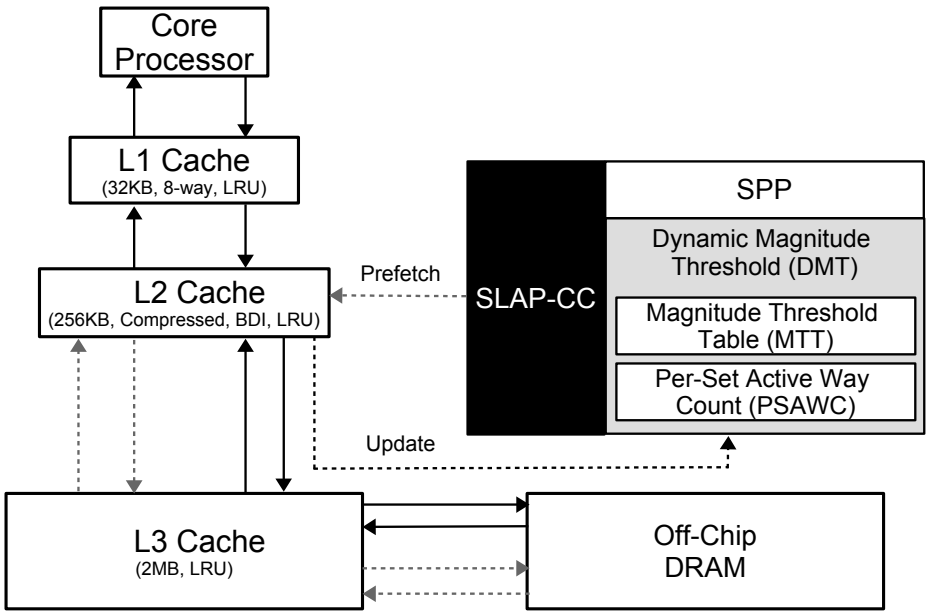


Figure 4.7: SLAP-CC Design Overview.

little impact. Unlike the LLC where capacities of tens of MB are common and working sets plus speculative prefetches often fit well, the L2 is still relatively capacity constrained so the added capacity has some impact. Thus, as shown in the figure, we implement a baseline cache compression policy, based off of the prior work BΔI [26], in the L2.

As shown in the figure, SLAP-CC implements prefetching in the L2 as well. As discussed previously the goal of the SLAP-CC design is to adapt prefetch aggressiveness on a per destination set basis. To this end, unlike prior prefetching techniques, SLAP-CC introduces an adaptive confidence threshold mechanism to filter individual prefetches depending upon which set those prefetches are destined towards, allowing it to be more aggressive where the cache is less efficient due to greater compressiblity and less aggressive where less space is available and the cache is more efficient due to low compressiblity.

4.3.2 SLAP-CC Cache Compression

The goal of SLAP-CC is not to create a new, aggressive cache compression algorithm. Rather, we seek to show that our per-compressed set prefetch filtering mechanism can work

with even a light-weight, low complexity and conservative cache compression scheme. To that end, our baseline L2 cache is compressed cache using the prior work BΔI algorithm [26] for cache line compression. The details of this compression algorithm were laid out previously in Section 4.2.1.

In keeping with our strategy to show our technique works with even a conservative cache compression algorithm, here we allow only two compression options for each cache line, either 64-bytes for uncompressed lines or 32-bytes for compressed lines. Thus, if a cache line can be compressed to 32-bytes or less it will only consume one-half of a normal line's space. Cache lines which cannot be compressed to less than 32-bytes are not compressed.

Here we assume that the baseline L2 cache is an 8-way set associative cache. To track the added blocks in the compressed cache, we doubled the size of the tag array and metadata. Given that each individual line may be compressed up to 2X, the maximum overall cache compression ratio that can be achieved by this design is 2X. Thus, the number of ways in each cache set can be anywhere from 8-way to 16-way depending on the compressibility of cache lines that map to that set.

As with most typical cache compression schemes, upon a miss's insertion into the cache, the SLAP-CC compression algorithm compresses the cache line. If the line cannot be compressed to less than 32-bytes it is inserted as a full, 64-byte cache line. If the cache line can be compressed to less than 32-bytes it is inserted, occupying only one half of line's space allowing two lines to be stored in that location. As mentioned above the SLAP-CC prefetcher must know the number of ways in a set in order to determine how aggressively to prefetch into that set. Accessing the tag array to determine the set's way count for every prefetch would be overly costly, thus SLAP-CC maintains a per-set active way count (PSAWC) in the SLAP-CC prefetch filter, as shown in Figure 4.8. Whenever a cache line is inserted, the new way count for that set is updated in the PSAWC at the same time.

4.3.3 SLAP-CC Prefetcher

The goal of SLAP-CC is to implement a per-set prefetch aggressiveness filtering based on the compressibilty of that set. To that end we will need to leverage prefetch confidence (*i.e.* estimated use probability). To date, existing prior work prefetching techniques which implement some form of confidence [7, 81, 68, 87] all use this confidence mechanism to throttle further prefetches down a given path. This is because they operate under the assumption that if a prefetch generated along a given speculative path has a confidence indicating that fetching it would be too aggressive, any further prefetches down that path will be even more speculative and aggressive and thus non-useful. Here this assumption is broken, as we wish to modulate our prefetching aggressiveness threshold depending on the number of ways available in the given set. Thus, it will likely be the case that while generating prefetches in sequence along a path with decreasing confidence some earlier, higher probability prefetches might be dropped (*e.g.* if the set they are destined for has fewer ways) while later, lower confidence prefetches down that path might be allowed to fetch (*e.g.* if the sets they are destined for has more effective ways.

The high level design of the SLAP-CC is illustrated in Figure 4.8. As we need a prefetcher which can generate a per-prefetch confidence, SLAP-CC will leverage and modify the path confidence of the prior work SPP [68] prefetcher, implemented as described in Section 4.2.4.1. The largest change being that SLAP-CC implements an adaptive, per-set compression level threshold. This allows SLAP-CC to modulate its aggressiveness on a per set basis. To this end, SPP is modified to remove its normal threshold based throttling function.

Instead for each prefetch that SPP would normally generate, the prefetch's set is looked up in the PSAWC to determine how many effective ways there are in that set, as shown in Figure 4.8. As described previously, this may be anywhere from 8 ways (*e.g.* if no ways are compressed) to 16 ways (*e.g.* if all the ways are compressed). As shown in Figure 4.8, this way counter is used to index into the Magnitude Threshold Table (MTT). Then the corresponding magnitude threshold is read. The confidence of the prefetch candidate is

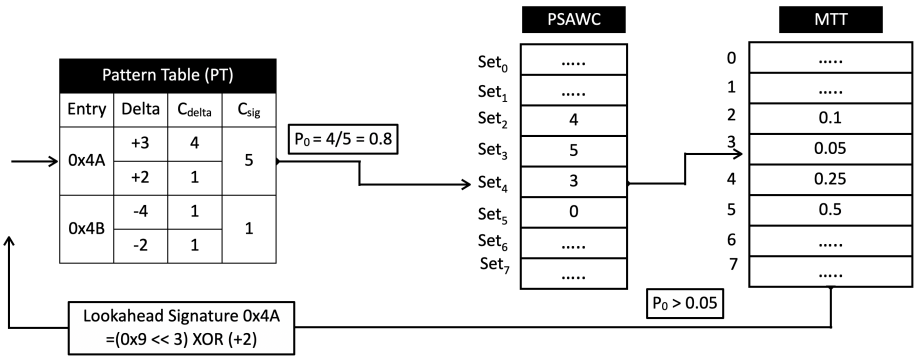


Figure 4.8: Dynamic Magnitude Threshold.

compared against the magnitude threshold to adaptively determine if the prefetch should issue or be dropped.

Unlike SPP that completely stops prefetching when it hits a predefined threshold magnitude, SLAP-CC stops when it hits the lowest threshold magnitude of any of the possible cache sets (*i.e.* the threshold set for a maximally compressed set, 16-ways in this case). Depending upon the target set of each individual prefetch candidate that determines whether the prefetch issue or not.

4.4 SLAP-CC Implementation

The additional hardware storage requirements for SLAP-CC are summarized in Table 4.1. As described in Section 4.3, SLAP-CC's prefetcher is based on the prior work SPP design with an overhead of 5.37KB in the L2. Our baseline underlying cache compression algorithm, BΔI, requires that we double the number of tags in the L2 to track the maximum blocks that might be in a given set, thus it requires an additional 8.5KB. To account for the extra latency of the increased tag set, as well as compression/decompression latency, 2 cycles are added to the L2 for hit access and miss insertion.

SLAP-CC requires two main new components beyond the typical prefetch engine and cache compression hardware, the Per-Set Active Way Count (PSAWC) table, and the Magnitude Threshold Table (MTT). In terms of hardware budget Per-Set Active Way Count is

Table 4.1: Hardware storage overhead in KB

	Component	# Entries	Size (KB)
SLAP-CC	Prefetcher	–	5.37
	Compressed Cache	–	8.5
	PSAWC	512	0.31
	MTT	16	0.020
	TOTAL SIZE :		**14.20**

0.31KB, and the Magnitude Threshold Table is 0.020KB. Thus, while the total combined size of SLAP-CC, including the prefetcher and cache compression component is 14.2KB, the overhead above a standalone prefetcher and cache compression scheme is only 0.33KB.

Note that, while our compression scheme doubles the number of tags maintained in the L2 cache, as we will show, typically only 1/2 of those ways are in use at any given time. These extra unused tag ways could be power gated to reduce power or used to store prefetcher or replacement policy meta data. In our future work we will explore mechanisms to leverage this space more effectively.

4.5 Evaluation

In this section, we evaluate the SLAP-CC system. First, we present the evaluation methodology, followed by performance. Finally, we present in-depth analysis on prefetching coverage and the impact of SLAP-CC on cache efficiency.

4.5.1 Methodology

We built our simulation infrastructure from ChampSim [98], a trace driven simulator used in the Second and Third Data Prefetching Championship (DPC2, DPC3) [62, 99], the Second Cache Replacement Competition (CRC2) [100] and the First Instruction Prefetching Championship(IPC1) [101]. We note that the standard version of ChampSim does not retain memory data values nor do its traces, and thus can not be used in cache compression studies. To address this issue, we modified the ChampSim tracer to retain data values along with addresses of memory accesses and regenerated all benchmark traces. We modeled a single-

Table 4.2: Simulation Parameters.

CPU Core	1 Cores, 4 GHz 256 entry ROB, 4-wide
Private L1 DCache	32 KB, 8-way, 4 cycles 8 MSHRs, LRU
Private L2 Cache	256 KB, 8-way 8/10 cycles (uncompressed/compressed) 16 MSHRs, LRU, Non-inclusive
Shared LLC	2MB/core, 16-way, 12 cycles 32 MSHRs, LRU, Non-inclusive
DRAM	4 GB 1-Channel (single-core) 64-bit channel, 1600MT/s

core, out-of-order machine [2]. The details of the configuration parameters are summarized in Table 4.2.

The uncompressed cache block size is fixed at 64 bytes. The compression granularity size is fixed at 32 bytes. In the event that a cache line is compressible but cannot be compressed to 32-bytes the uncompressed line is stored instead. To provide the additional tags and metadata we doubled the tag array. Prefetching is only activated upon L2 cache demand misses and could be directed to the L2 or last-level cache based on the prefetch confidence . There is no data prefetcher on L1 cache. The LRU replacement policy is used on all levels of cache hierarchies. Branch prediction is done using the hashed perceptron branch predictor [102]. The page size is 4KB. ChampSim operates all the prefetchers strictly in the physical address space.

We use all 20 workloads available in the SPEC CPU 2017 suite [103]. We use the SimPoint [104] methodology to reduce simulation times for these workloads. For performance evaluation, we warm up each core for 200M instructions and collect results over an additional 1B instructions. We compare SLAP-CC against SPP, SPP+BΔI, and BOP+BΔI. The

[2]Note: because SLAP-CC's focus is on the **private** L2 caches we expect that the results should carry over to multi-processor systems (if not provide even larger benefits as the L2 capacity increase can lead to less reliance on the shared LLC). In the future we plan to examine LLC cache compression and prefetching where multi-program workloads would have a bigger impact.

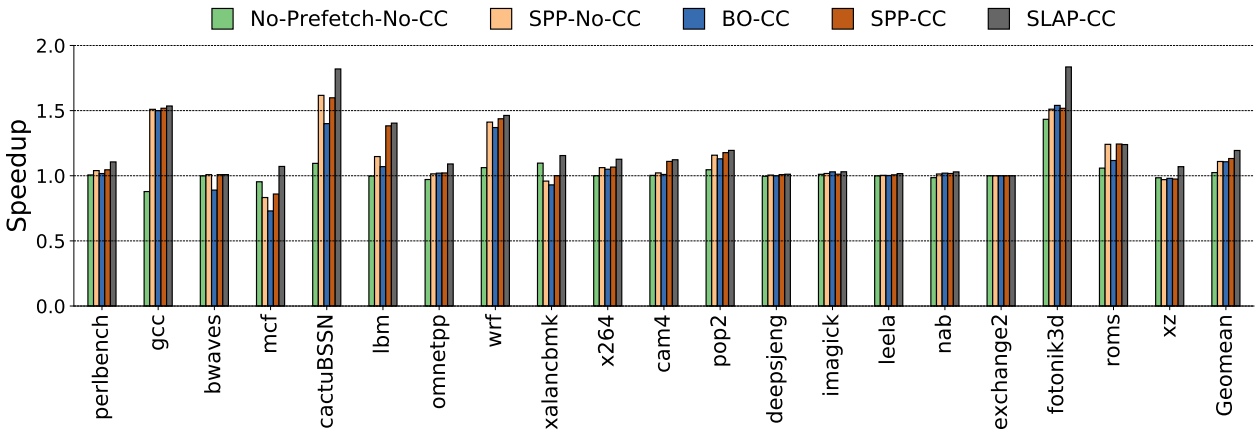


Figure 4.9: SPEC CPU 2017 Single-Core IPC Speedup, normalized against a system with a BΔI compressed L2 and no prefetching.

baseline uses a compressed cache with BΔI compression [26].

4.5.2 Performance Analysis

In this section we discuss results of our experiments. Here we first examine the performance of SLAP-CC against a BΔI baseline without prefetching and against prior work prefetching techniques together with BΔI compression. Next we analyze the impact of SLAP-CC on prefetching coverage and cache efficiency. Finally we examine the compression ratio achieved by BΔI on our workloads to understand the impact of compression in the L2 generally.

Speedup: The ultimate goal of this work is to reduce cache misses and thereby induce a performance gain in our workloads. Figure 4.9 shows the IPC speedup of SLAP-CC and two prior work prefetching techniques (SPP [68] and BOP [21]) together with BΔI, normalized against a baseline with L2 cache compression (BΔI) and no prefetching. We also show results for a non-compressed cache normalized against the baseline compressed cache to illustrate how much performance is gained by compressed caches without prefetching.

Overall, SLAP-CC outperforms or matches the prior work prefetching techniques on every benchmark. The geometric mean speedup for SLAP-CC is 18.0%, compared with 11.0% for SPP. Overall, we note that SLAP-CC's improvement over SPP+CC is more than

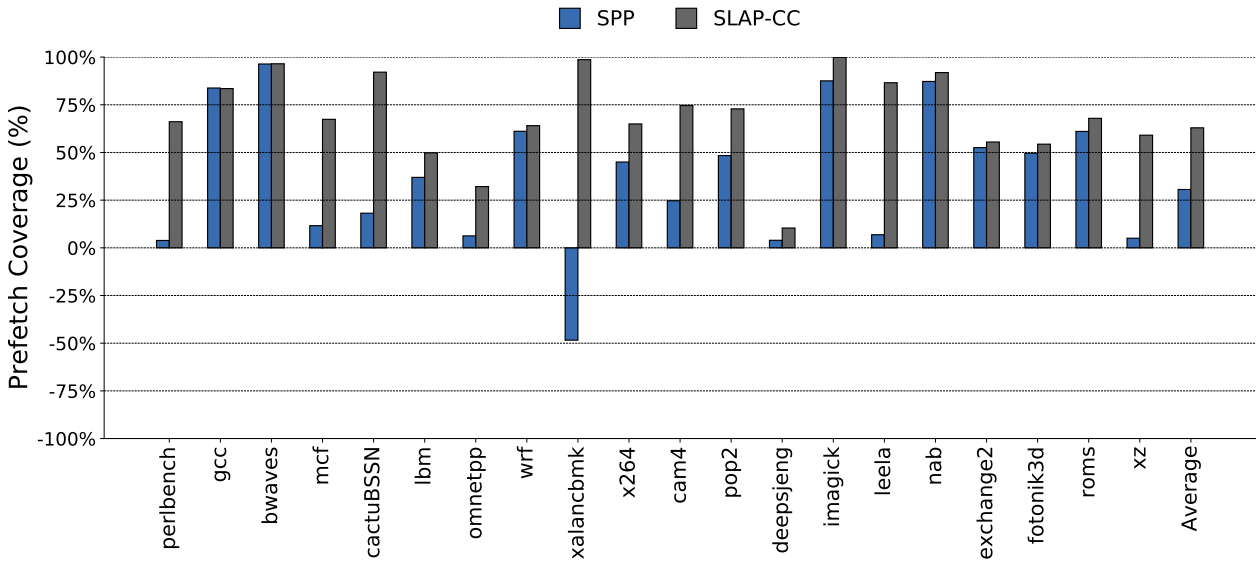


Figure 4.10: Prefetching coverage.

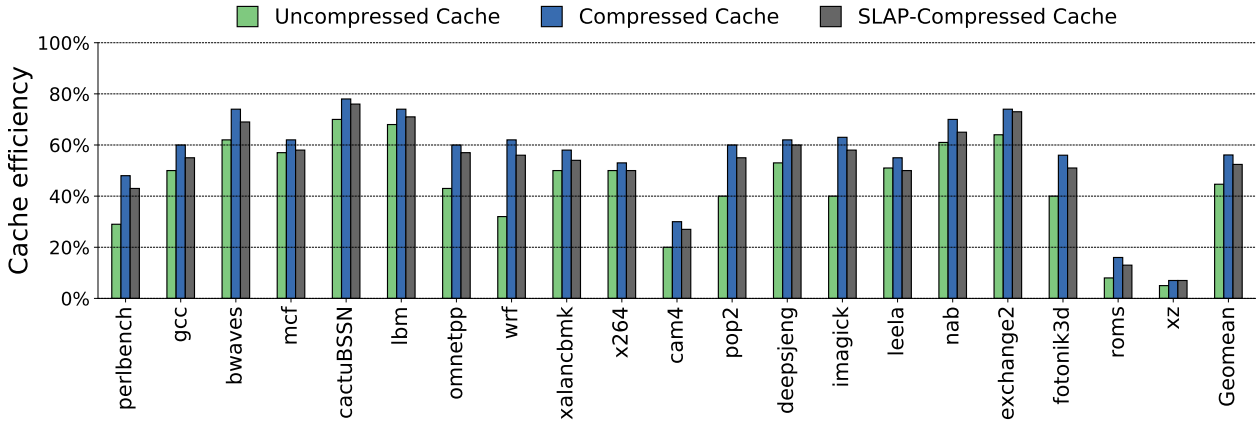


Figure 4.11: Cache efficiency.

the performance differential between no prefetching no cache compression and stand alone cache compression shown in Figure 4.2. Thus, we have accomplished our goal of making prefetching and cache compression performance additive.

Interestingly, we see that for several benchmarks (*e.g. mcf*), the prior work prefetching techniques plus compressed cache actually significantly reduce performance versus a compressed cache baseline. In those workloads, cache compression allows the retention of

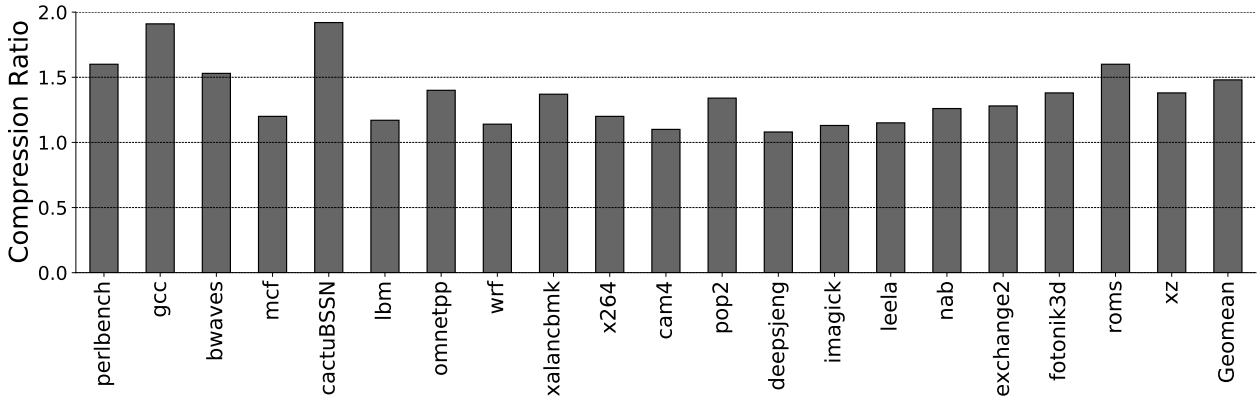


Figure 4.12: Compression Ratio.

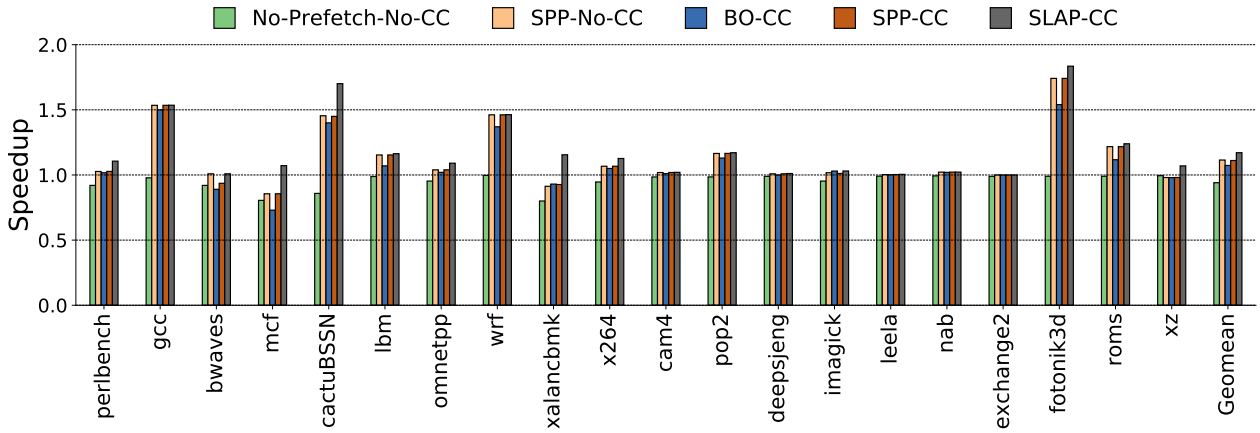


Figure 4.13: Speedups for a baseline system with a faster LLC access time.

important parts of the working set of the application, whereas overly aggressive prefetching knocks those useful components back out. This highlights how essential it is to co-design the prefetcher for the cache compression scheme so that they do not fight against each other. SLAP-CC in all cases improves performance to be greater than cache compression alone.

For some workloads, SLAP-CC works particularly well. For example, *cactuBSSN* in particular, yields remarkable speedups of 75% over the baseline and 25% more than SPP. Similarly *mcf* and *xalancbmk* show substantial gains versus the prior work prefetchers. We will now examine these results with more detailed analysis.

Coverage: The performance gain from SLAP-CC compared to other prefetchers is a

direct result of SLAP-CC's increase in coverage. Here we will focus on a comparison with SPP, since both SLAP-CC and SPP use similar mechanisms to generate prefetches. Figure 4.10 shows the prefetching coverage for each benchmark. As the figure shows, SLAP-CC prefetcher increases the coverage across all benchmarks compared to SPP. We see that coverage increases are among the largest for mcf, $cactusBSSN$ and $xalancbmk$, matching the highest performance increases on the speedup graphs above. $xalancbmk$ in particular shows a dramatic improvement in coverage; this benchmark's primary working set is somewhat larger than the L2 cache capacity and has a highly variable per-set compression ratio. Thus, cache compression benefits the benchmark well. Blindly, aggressively prefetching, however, can upset the locality of the less compressed sets, making it almost a perfect case for SLAP-CC. Interestingly, there are several benchmarks (*e.g. perlbench* and *leela*) where there are large improvements in coverage that do not translate into significant performance gains. In these workloads there are not sufficient L2 misses for SLAP-CC or other prefetching techniques to have an impact.

Cache Efficiency: As discussed in section 4.2, cache compression leads to worse cache efficiency compared to uncompressed caches. Figure 4.11 shows the improvement of cache efficiency for every benchmark with baseline uncompressed, baseline compressed and SLAP-CC. In the figure, we see that SLAP-CC achieves an average efficiency improvement of $\approx$ 4.0% over baseline compressed. Thus SLAP-CC covers close to half the difference between the uncompressed and baseline compressed cache. This shows that SLAP-CC is able make better use of the extra capacity that cache compression creates than without a codesigned prefetching scheme.

Cache Compression Ratio: Figure 4.12 shows the results of effective compression ratio using BΔI as described in Section 4.2.1. In the figure we see that on average BΔI achieves a compression ratio of $\approx$1.50X across all benchmarks. We see that the compression ratio varies significantly from benchmark to benchmark, with some benchmarks seeing little compression (*e.g. deepsjeng*) while others achieving a nearly perfect 2X ratio (*e.g. gcc*).

Sensitivity Analysis: Here we examine the impact of changing the last level cache latency. In particular, Figure 4.13 shows the speedup of all benchmarks when LLC access time is 35 cycles, instead of the baseline 50 cycles. In general, the figure shows that with faster LLC access all prefetchers lose 1-2%. Nevertheless, SLAP-CC maintains a significant performance differential with the best performing alternative SPP w/o cache compression. These results reinforce the intuition that, since the system compresses and prefetches into the L2 cache, the LLC configuration has little impact on the results we obtain.

4.6 Conclusion

Data compression and prefetching are promising techniques to bridge the gap in performance between the processor and memory subsystem. That said, blithely putting both techniques together without coordination of co-design can actually reduce the benefit of either one. In this chapter propose a new prefetching algorithm designed for compressed caches. In particular, SLAP-CC leverages the variability in compressibilty that typical cache compression algorithms see. SLAP-CC seeks to address this variability by individually confidence filtering prefetches based on how much effective capacity is available in the set the prefetch is destined towards. SLAP-CC is low overhead versus baseline cache compression and prefetching techniques and achieves a geometric mean speedup of 18.0% over baseline, outperforming prior work prefetching techniques when implemented with cache compression.

5. SCHEDULING MECHANISM FOR HARD TO PREFETCH LOADS

Modern data prefetchers prefetch the majority of cache lines ahead of explicit demand, minimizing the impact of the memory wall on the out-of-order execution core and thus improving performance. Despite the fact that hardware prefetchers have an impressive impact on instructions per cycle (IPC) by reducing the number of cache misses, the processor is still exposed to the memory wall on a cache miss due to hard to prefetch loads. Hard-Load is defined as the load instruction that misses in the cache despite the using a prefetcher to prefetch the data into that cache. In this work we propose a speculative technique to minimize the impact of hard-loads by scheduling hard-load instruction early to allow the cache hierarchy to start the miss handling mechanism earlier and reducing the impact of the cache miss on performance.

5.1 Introduction

Out-of-Order superscalar microprocessors exploit Instruction Level Parallelism (ILP) to maximize performance by executing more than one instruction at same time. Instructions may be executed in an order that is different from the program order. To sustain high performance from the Out-of-Order execution mode, the instruction scheduler must keep the instructions moving through the pipeline at the highest rate possible, while maintaining data dependencies and hardware resource constraints [105].

As the pipeline width grows, instruction window size increases, and the high impact of the distance between execution core and DRAM on the overall performance. It is crucial to preemptively schedule hard-load instructions as soon as possible rather than wait for the actual cycle at which they become the oldest in the instruction window.

Data prefetching is critical to the performance of modern superscalar processors and is implemented in many commercial products. Typical hardware prefetchers are activated and trained by cache misses sequence, then using models to predict future memory accesses and

prefetch them ahead of demands. Modern data prefetchers prefetch the majority of cache lines ahead of explicit demand, minimizing the impact of the memory wall on the out-of-order execution core and thus improving performance. Despite the fact that hardware prefetchers have an impressive impact on instructions per cycle (IPC) by minimizing the number of cache misses, the processor is still exposed to the memory wall on a cache miss due to hard-to-prefetch loads that go all the way to DRAM to source the requested data. Cache misses that go all the way to DRAM delay subsequent instructions and stall instruction commit leading to zero IPC.

Designing hardware prefetcher centers around prefetch coverage and accuracy. Prefetcher coverage is defined as the number of cache misses avoided due prefetching divided by the number of misses with no prefetching [87]. Prefetcher accuracy is defined as the fraction of prefetched cache lines that end up being useful for the application [87].

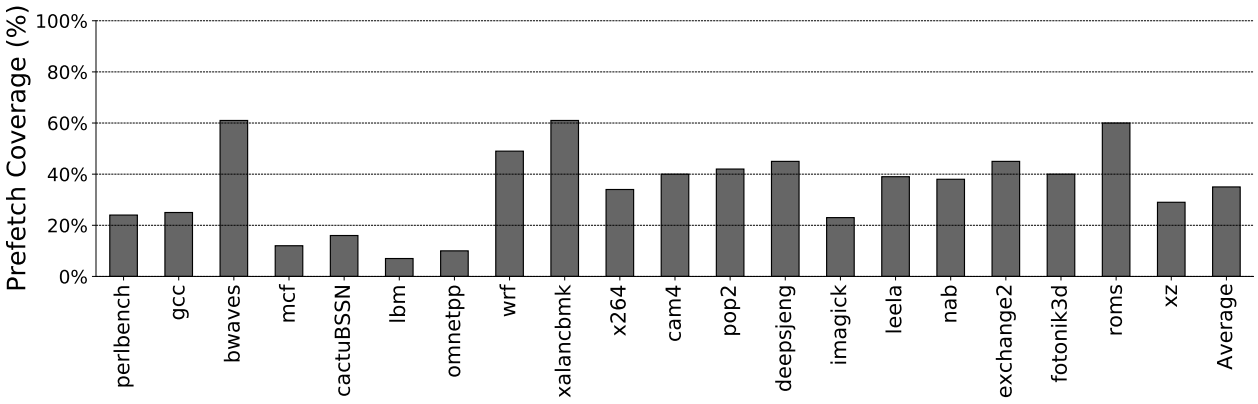


Figure 5.1: Prefetch Coverage of SPP.

For example, Figure 5.1 shows the prefetching coverage for the SPEC 2017 benchmarks on execution core modeled in ChampSim simulator. Prefetching coverage is measured by the number of useful prefetches divided by the number of cache misses without prefetching [68]. Figure 5.2 shows the speedup that might be achieved when using a perfect L2D-cache prefetcher normalized against the same system without prefetching. We can see that

there is substantial performance gain due to uncovered misses.

Due to uncertainty that the prefetcher has successfully prefetch the load data to L1/L2 cache ahead of the explicit demand, exposing the processor to long latency to source the data from LLC or DRAM due to hard-to-prefetch (H2P) loads. To reduce the impact of hard-to-prefetch loads, we we propose a scheduling mechanism that predicts the hard-to-prefetch loads at issue time and preemptively schedule them for execution as soon as they are ready, to allow the cache hierarchy to start the mishandling mechanism sooner. Such scheduling mechanism reduces the miss penalty on the dependent instructions after a hard-to-prefetch load.

In this chapter, we study the causes of hard-to-prefetch loads, and propose a speculative scheduling mechanism based on predicting hard-to-prefetch loads at issue time to speedup the miss handling mechanism and reduce the impact of hard-to-prefetch loads on dependent instructions. We identify three primary issues: (1) systematically hard-to-prefetch loads; (2) rare loads with low dynamic execution counts; and (3) multi-target load instructions. We propose a new scheduling mechanism that learns and predicts hard-to-prefetch loads and issue them to the execution units as soon as they are ready.

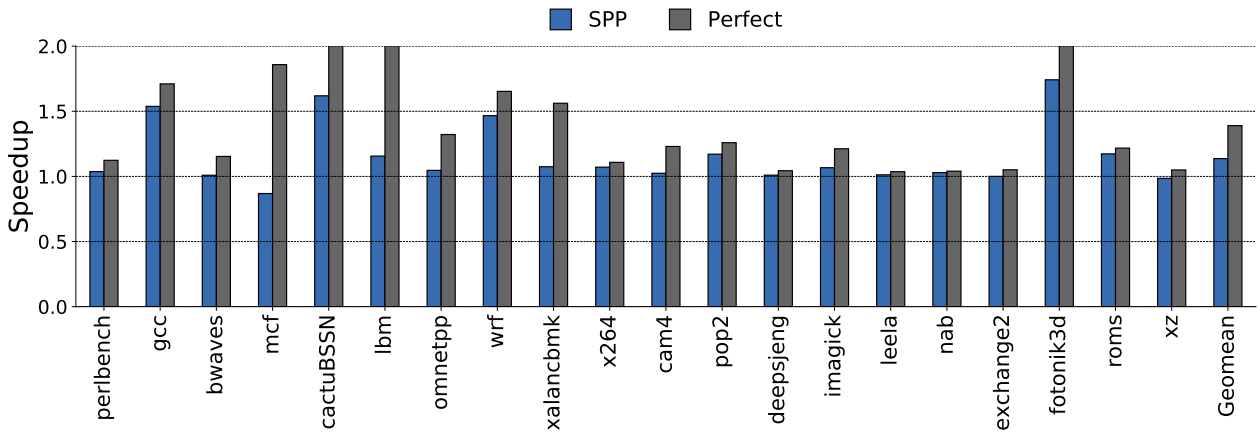


Figure 5.2: Speedup with SPP and Perfect Prefetcher.

The remainder of the chapter is organized as follows: Section 5.2 presents motivation and summarizes related work while Section 5.3 describes the Hard-to-Prefetch Loads Predictor and the Hard-to-Prefetch Loads Scheduling in depth. Section 5.4 gives an overview of the evaluation methodology and performance analysis. Finally, Section 5.5 provides some concluding remarks.

5.2 Background and Motivation

Below, we review instruction criticality, latency prediction, and finally review the state of the art in data prefetching.

5.2.1 Instruction Criticality

Data-dependency graph can be used to describe program criticality. one or more paths with a maximum length is called a critical path [35]. Instruction along that path called criticality instructions. A critical load is a load instruction that is on the critical path. Figure 5.3 shows the dataflow for a set of instructions. If every instruction takes the same number of cycles to execute, thus the longest path in the graph is the critical path and has a length of four, starting from I1 to I4. Therefore, instructions I1 to I4 are critical instructions.

Due to the variable latency to execute load instruction based on the cache level that is responsible to source the target data. A load instruction that is not on the critical path can have a big impact on the execution time when cache misses take place. In a system with data prefetcher, these loads expose the processor to long latency because they hard-to-prefetch.

Fields et al. [106, 107] proposed a method to identify the critical path of an application using directed graphs. Typical scheduling mechanisms tend to favor the selection of instruction along the critical path. While this can be important to achieve forward progress, it does not differentiate amongst memory accesses. Subramaniam et al. [35] proposed a criticality predictor for load instructions based on the number of the consumers of the load's data. They add counters to the ROB to track direct dependencies only, which can be determined when consumers enter the rename stage [35].

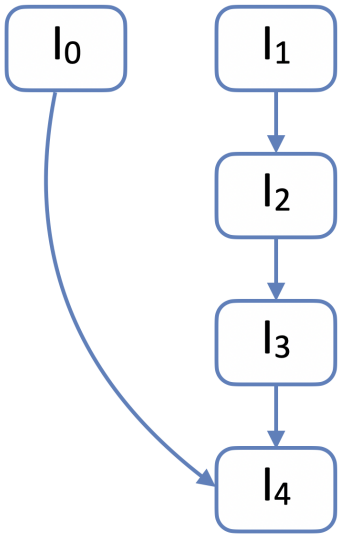


Figure 5.3: Dataflow graph.

5.2.2 Latency Prediction

To achieve scaling performance with increasing transistor density Out-of-Order processors exploit Instruction Level Parallelism by fetching and executing multiple instructions per cycle from a single instruction stream [40]. In order to exploit the ILP, Out-of-Order processors include a large instruction window and scheduler that searches for ready-to-execute instructions to issue them to the execution units. The performance of an Out-of-Order processor is bound by the critical path and the latency of hard-to-preftch load instructions. Hard-to-prefetch loads stall the ROB leading to zero IPC, which translates to performance degradation. Hard-to-prefetch loads are LLC misses that go all the way to DRAM to source the data. Identifying hard-to-prefetch loads and targeting them for early scheduling as soon as they are ready allows the cache hierarchy to get ahead start on the miss handling mechanism. Which results in reducing the impact of hard loads on the overall performance.

Liu et al. [55] propose a dynamic instruction sorting mechanism based on predicting the "waiting time" of all instructions. They used Latency History Table (LHT) to estimates the waiting time of instructions, sorting structure, and a Pre-issue Buffer (PB), where in-

structions are buffered before entering the issue queue [55]. The LHT is indexed using the predicted address of the load instruction and takes into account current requests to the data cache [55]. Also it can detect if a load aliases with an older load whose L1 miss is pending service.

Memik et al. [108] proposes a precise instruction scheduler which predicts when an instruction can start its execution and schedules the instructions such that they will arrive at the corresponding position in the pipeline at the exact cycle when the data will be available to them.

Yoaz et al. [109] proposes a mechanism to to improve the scheduling of load dependent instructions by employing a data cache Hit-Miss Predictor to predict the dynamic load latencies, then delay dependent instructions until the data is fetched [109].

Finally, Ghose et al. [110] propose a processor-side load criticality predictor to improve the memory controller scheduling by ranking load requests to the memory, using a commit block predictor which tracks loads that previously blocked the ROB [110].

We propose a scheduling mechanism that predicts hard-loads at issue time and schedules them early to allow the cache hierarchy to start the miss handling mechanism earlier, so it reduces the impact of the cache miss on performance.

5.2.3 Data Prefetching

Data prefetching is a well known technique in which the cache is pre-filled with useful data ahead of an actual demand load request coming from the processor. Typically, the prefetching opportunity is limited to waiting until a cache miss occurs, and then reading either a set of lines sequentially following the current miss [16], a set of lines following a strided pattern with respect to the current miss [17], or a set of blocks spatially around the miss [86]. More recent prefetchers attempt to predict complex, irregular access patterns [86, 18, 19, 20, 69, 68]. While these methods show significant benefit, they are inherently reactive, waiting until a cache miss occurs to activate the prtefetcher to initiate prefetches down the speculated path.

Prefetcher coverage and accuracy are two important metrics used to evaluate the effec-

tiveness of a prefetcher. Prefetcher coverage measures fraction of demand misses eliminated by the prefetcher. Prefetcher accuracy measures to the fraction of prefetched cache lines that end up being useful to the application.

Ideal prefetcher would accomplish both high accuracy and coverage. Since accuracy is at variance with coverage, typical Prefetchers include a throttling mechanisms to trade-off coverage for accuracy and vice versa. Improving prefetch coverage at the expense of accuracy leads to cache pollution.

5.3 Hard-to-Prefetch Loads Scheduling

Hard-to-Prefetch loads are the loads that stall the head of the reorder buffer (ROB) and account for a large percent of the execution time. These loads go all the way to the DRAM exposing the Out-of-Order core to the memory wall. We propose a scheduling mechanism that targets these loads for early scheduling by giving them higher priority over other ready-to-execute instructions, allowing the cache subsystem to start the miss handling mechanism earlier. Such a scheduling mechanism minimizes the impact of hard loads on the critical path execution. Next, we will discuss the sequence of steps to implement this in the OOO pipeline.

5.3.1 Identifying Hard Loads

Out-of-Order cores use a large instruction window to hide the latency of dependent instruction by utilizing instruction level parallelism. To predict hard to prefetch loads at scheduling time, first we need to identify the hard to prefetch loads in in a given instruction window and then select the hardest and oldest to issue to the execution units. We propose Hard to Prefetch Loads Predictor (HLP), which tracks loads which have previously stalled the ROB. When a load instruction stalls the ROB head, we calculate the commit distance, which is the number of instructions that are ready to commit but waiting on the load to be ready to commit but waiting on the load instruction to commit. When a load instruction stalls the ROB head, the predictor is accessed and the load instruction installed in the HLP

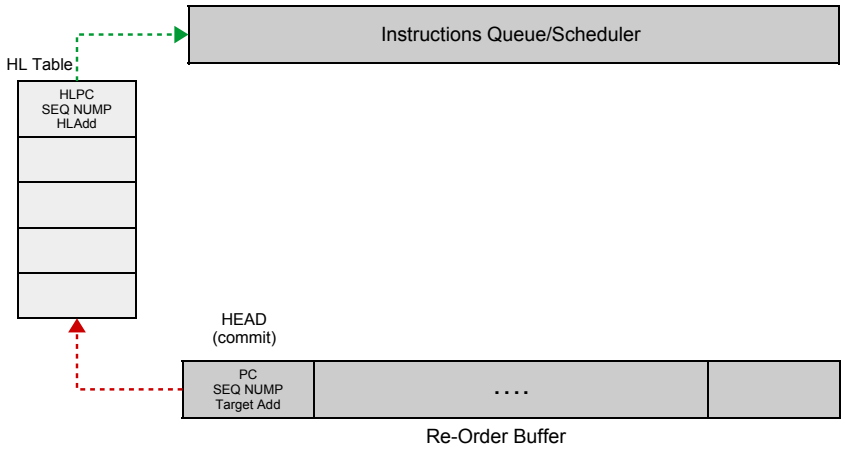


Figure 5.4: HLS Design Overview.

table. When a load instruction in the instruction window with the same PC hits the HLP table, the instruction scheduler will mark this load as a hard load and issue it to the execution unit.

In order for a load instruction to be classified as Hard-to-Prefetch Load. First, The control logic needs to identify that an instruction is a load instruction by recognizing the opcode of the instruction. Second, the instruction must be waiting at the head of ROB. Finally, the instructions that follow the load instruction are ready to commit but waiting on the load. The instruction pointer and the number of stall cycles also recorder for each entry in the HPL table.

To allow the HLT to un-learn that a given load is no longer a Hard-to-Prefetch Load, we employ a simple, counter-based, random clear mechanism. The counter counts cycles up to a definable maximum value. When this maximum is reached the content of the table is cleared.

5.3.2 Scheduler Design

Figure 5.4 shows the high-level microarchitecture of Hard-Prefetch-Loads Predictor and how it interacts with the ROB and the instruction scheduler. Hard to prefetch loads are identified by their ROB stall time and commit distance. When a load stalls the head of ROB and followed by ready to commit instruction, we install a new entry in the HLT that includes the Load-PC, Load-IP, effective address, number of stall cycles.

At the scheduling time the scheduler control logic looks up the HLT to predict if any of the ready loads are hard to prefetch loads. If a load is predicted to be hard to prefetch then that load is issued to the execution units immediately. In our evaluation we assume that the prediction can be performed every cycle or every other cycle.

By scheduling hard to prefetch loads ahead of other load instructions, we prioritize the mishandling for these loads. In the case of an L2 miss, the fetch address request goes to the last level cache, then to DRAM in case LLC miss. No changes are required to the memory controller or the arbiter, therefore we avoid starvation of non-hard load instruction

5.4 Evaluation

In this section, we evaluate the Hard-to-Prefetch load scheduling. First, we present the evaluation methodology, followed by single core performance.

5.4.1 Methodology

We used the ChampSim [98], a trace driven simulator, to evaluate HLS against prior work techniques. The ChampSim simulator used in ChampSim used in the Third Data Prefetching Championship (DPC3) [99], the Second Cache Replacement Competition (CRC2) [100] and the First Instruction Prefetching Championship(IPC1) [101]. We extended ChamSim capabilities to model a Hard-Prefetch-Loads Predictor and process benchmark traces that include data values. We modeled a single-core out-of-order machine. The details of the configuration parameters are summarized in Table 5.1.

We use all the 20 workloads available in the SPEC CPU 2017 suite [103]. Using the Sim-

CPU Core	1 Cores, 4 GHz 256 entry ROB, 4-wide
Private L1 DCache	32 KB, 8-way, 4 cycles 8 MSHRs, LRU
Private L2 Cache	256 KB, 8-way, 8 cycles 16 MSHRs, LRU, Non-inclusive
Shared LLC	2MB/core, 16-way, 12 cycles 32 MSHRs, LRU, Non-inclusive
DRAM	4 GB 1-Channel (single-core) 64-bit channel, 1600MT/s

Table 5.1: Simulation Parameters.

Point [104] methodology. For performance evaluation, we warm up each core for 200M instructions and collect results over an additional 1B instructions. We compare hard-prefetch-loads scheduler against a baseline system with no hard-prefetch-loads scheduler.

5.4.2 Performance Analysis

In this section we discuss results of our experiments. Here we first examine the performance of HLS against a baseline without prefetching. Next we analyze the impact of instruction window size on HLS.

The ultimate goal of this work is to reduce the impact of cache misses by preemptively scheduling hard-load instructions such that the dependent instructions do not see the whole penalty. Figure 5.5 shows the IPC speedup of HLS normalized against a baseline with no prefetching. Overall, HLS outperforms the prior work prefetching techniques on every benchmark. The geometric mean speedup for HLS is 16.0%, compared with 13.0% for SPP.

5.5 Conclusion

In this chapter we present a scheduling mechanism to reduce the impact of Hard-to-Prefetch loads by predicting Hard-to-Prefetch at issue time and preemptively schedule them as soon as possible.

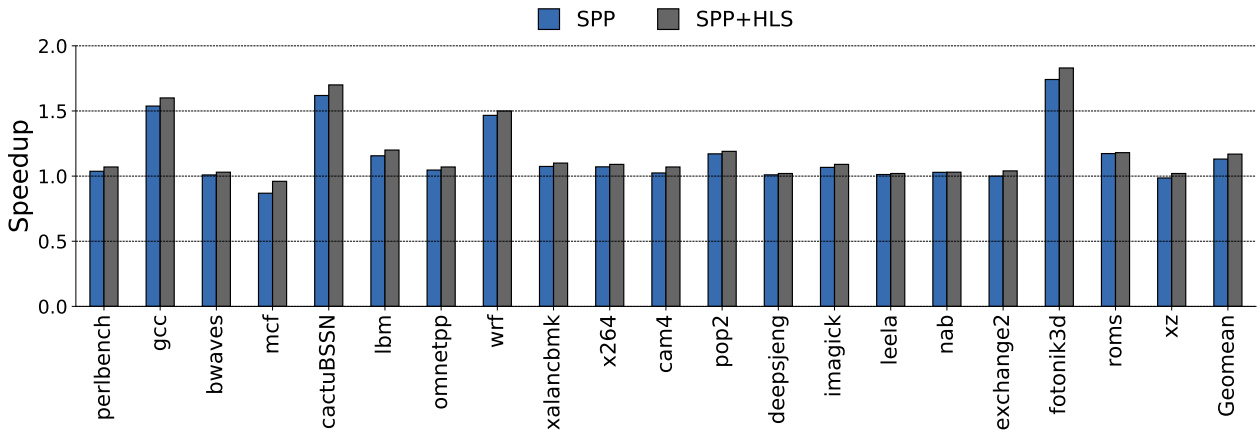


Figure 5.5: SPEC CPU 2017 Single-Core IPC Speedup.

6. CONCLUSION

The power and area constraints on microprocessor design make it impractical to naively increase the size of cache structures to close the gap of performance between the processor and the memory subsystem. Therefore, in modern microprocessors, speculative techniques for memory hierarchy are an essential part of the design to achieve better performance.

In this dissertation, we discussed three speculative techniques that efficiently manage the memory hierarchy. First, we introduce SB-Fetch: synchronization-aware hardware prefetching for chip multiprocessors to address the problems of prefetching for shared memory applications. Prefetching in shared memory workloads is different and harder than in single threaded workloads due to locks and sharing conflicts. SB-Fetch explicitly issues prefetches beyond synchronization points and reduces early-prefetch invalidations to achieve significant performance impact.

Second, we present SLAP-CC: set-level adaptive prefetching for compressed caches. Where we explicitly co-designed a prefetcher for a compressed cache. We also showed that putting compression and prefetching together without coordination reduce the benefit of either one. SLAP-CC adapts prefetch aggressiveness to the workload compressibility and adjusts prefetching aggressiveness on a per-set basis. SLAP-CC is low overhead technique with significant performance impact.

Third, we propose a scheduling mechanism for hard-to-prefetch loads which reduces the impact of cache misses by preemptively scheduling hard-load instructions such that the dependent instructions do not see the whole penalty.

The aforementioned speculative techniques for modern memory hierarchies, resulting in performance improvements and attractive to be implemented in future systems.

REFERENCES

[1] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, pp. 20–24, Mar. 1995.

[2] R. H. Dennard, F. H. Gaensslen, H. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[3] S. Sardashti and D. A. Wood, "Could compression be of general use? evaluating memory compression across domains," *ACM Trans. Archit. Code Optim.*, vol. 14, Dec. 2017.

[4] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial memory streaming," in *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ISCA '06, (Washington, DC, USA), pp. 252–263, 2006.

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

[6] L. AlBarakat, "Multithreading aware hardware prefetching for chip multiprocessors," tech. rep., Texas A&M University, July 2017.

[7] D. Kadjo, J. Kim, P. Sharma, R. Panda, P. Gratz, and D. Jimenez, "B-fetch: Branch prediction directed prefetching for chip-multiprocessors," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 623–634, 2014.

[8] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: an alternative to very large instruction windows for out-of-order processors," in *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*, pp. 129–140, Feb 2003.

[9] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: an alternative to very large instruction windows for out-of-order processors," in *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*, pp. 129–140, Feb 2003.

[10] O. Mutlu, H. Kim, and Y. N. Patt, "Efficient runahead execution: Power-efficient memory latency tolerance," *IEEE Micro*, vol. 26, p. 10âĂŞ20, Jan. 2006.

[11] J. Dundas and T. Mudge, "Improving data cache performance by pre-executing instructions under a cache miss," in *Proceedings of the 11th International Conference on Supercomputing*, ICS '97, (New York, NY, USA), pp. 68–75, ACM, 1997.

[12] R. Panda, P. Gratz, and D. Jimenez, "B-fetch: Branch prediction directed prefetching for in-order processors," *IEEE Comput. Archit. Lett.*, vol. 11, pp. 41–44, July 2012.

[13] D. Kadjo, J. Kim, P. Sharma, R. Panda, P. Gratz, and D. Jimenez, "B-fetch: Branch prediction directed prefetching for chip-multiprocessors," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pp. 623–634, 2014.

[14] Y. Liu and D. R. Kaeli, "Branch-directed and stride-based data cache prefetching," in *Proceedings International Conference on Computer Design. VLSI in Computers and Processors*, pp. 225–230, Oct 1996.

[15] Tien-Fu Chen and Jean-Loup Baer, "Effective hardware-based data prefetching for high-performance processors," *IEEE Transactions on Computers*, vol. 44, no. 5, pp. 609–623, 1995.

[16] A. J. Smith, "Sequential program prefetching in memory hierarchies," *Computer*, vol. 11, pp. 7–21, December 1978.

[17] T. Chen and J. Baer, "Effective hardware-based data prefetching for high-performance processors," *IEEE Transactions on Computers*, vol. 44, pp. 609–623, 1995.

[18] A. Jain and C. Lin, "Linearizing irregular memory accesses for improved correlated prefetching.," in *MICRO*, pp. 247–259, 2013.

[19] Y. Ishii, M. Inaba, and K. Hiraki, "Unified memory optimizing architecture: memory subsystem control with a unified predictor," in *Proceedings of the 26th ACM International Conference on Supercomputing*, pp. 267–278, ACM, 2012.

[20] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, "Efficiently prefetching complex address patterns," in *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture*, 2015.

[21] P. Michaud, "Best-offset hardware prefetching," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 469–480, March 2016.

[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)*, pp. 1–12, Oct 2016.

[23] J. Lee, N. B. Lakshminarayana, H. Kim, and R. Vuduc, "Many-thread aware prefetching mechanisms for gpgpu applications," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 213–224, Dec 2010.

[24] J. Izraelevitz, L. Xiang, and M. L. Scott, "Performance improvement via always-abort htm," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 79–90, Sep. 2017.

[25] S. Sardashti, A. Seznec, and D. A. Wood, "Yet another compressed cache: A low-cost yet effective compressed cache," *ACM Trans. Archit. Code Optim.*, vol. 13, Sept. 2016.

[26] G. Pekhimenko, V. Seshadri, O. Mutlu, M. A. Kozuch, P. B. Gibbons, and T. C. Mowry, "Base-delta-immediate compression: Practical data compression for on-chip caches," in *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 377–388, 2012.

[27] X. Chen, L. Yang, R. P. Dick, L. Shang, and H. Lekatsas, "C-pack: A high-performance microprocessor cache compression algorithm," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 18, no. 8, pp. 1196–1208, 2010.

[28] J. Yang and R. Gupta, "Frequent value locality and its applications," *ACM Trans. Embed. Comput. Syst.*, vol. 1, pp. 79–105, Nov. 2002.

[29] J. Yang, Y. Zhang, and R. Gupta, "Frequent value compression in data caches," in *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 33, (New York, NY, USA), pp. 258–265, Association for Computing Machinery, 2000.

[30] M. Ekman and P. Stenstrom, "A robust main-memory compression scheme," in *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, ISCA '05, (USA), pp. 74–85, IEEE Computer Society, 2005.

[31] J. Dusser, T. Piquet, and A. Seznec, "Zero-content augmented caches," in *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, (New York, NY, USA), pp. 46–55, Association for Computing Machinery, 2009.

[32] A. R. Alameldeen and D. A. Wood, "Adaptive cache compression for high-performance processors," in *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.*, pp. 212–223, 2004.

[33] A. R. Alameldeen and D. A. Wood, "Interactions between compression and prefetching in chip multiprocessors," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pp. 228–239, 2007.

[34] S. Sardashti, A. Seznec, and D. A. Wood, "Skewed compressed caches," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 331–342, 2014.

[35] S. Subramaniam, A. Bracy, H. Wang, and G. H. Loh, "Criticality-based optimizations for efficient load processing," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, pp. 419–430, 2009.

[36] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[37] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, (Washington, DC, USA), pp. 44–54, IEEE Computer Society, 2009.

[38] J. A. Stratton, C. I. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. Liu, and W. mei W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," 2012.

[39] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, 2008.

[40] E. Fatehi and P. Gratz, "Ilp and tlp in shared memory applications: A limit study," in *the 23rd International Conference on Parallel Architectures and Compilation (PACT)*, pp. 113–126, 2014.

[41] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi, "Spatio-temporal memory streaming," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pp. 69–80, 2009.

[42] Digital, *Alpha Architecture Handbook*. Digital Press, 1992.

[43] E. H. Jensen, G. W. Hagensen, and J. M. Broughton, "A new approach to exclusive data access in shared memory multiprocessors," tech. rep., Technical Report UCRL-97663, Lawrence Livermore National Laboratory, 1987.

[44] J. Archibald and J.-L. Baer, "Cache coherence protocols: Evaluation using a multiprocessor simulation model," *ACM Trans. Comput. Syst.*, vol. 4, pp. 273–298, Sept. 1986.

[45] J. Laudon and D. Lenoski, "The sgi origin: A ccnuma highly scalable server," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, (New York, NY, USA), pp. 241–251, ACM, 1997.

[46] M. M. K. Martin, D. J. Sorin, M. D. Hill, and D. A. Wood, "Bandwidth adaptive snooping," in *Proceedings Eighth International Symposium on High Performance Computer Architecture*, pp. 251–262, Feb 2002.

[47] M. R. Marty and M. D. Hill, "Coherence ordering for ring-based chip multiprocessors," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pp. 309–320, Dec 2006.

[48] A. L. Cox and R. J. Fowler, "Adaptive cache coherency for detecting migratory shared data," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, (New York, NY, USA), pp. 98–108, ACM, 1993.

[49] J. Huh, J. Chang, D. Burger, and G. S. Sohi, "Coherence decoupling: Making use of incoherence," in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, (New York, NY, USA), pp. 97–106, ACM, 2004.

[50] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi, "Cuckoo directory: A scalable directory for many-core systems," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pp. 169–180, Feb 2011.

[51] S. Kaxiras and G. Keramidas, "Sarc coherence: Scaling directory cache coherence in performance and power," *IEEE Micro*, vol. 30, pp. 54–65, Sept 2010.

[52] S. S. Mukherjee and M. D. Hill, "Using prediction to accelerate coherence protocols," in *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No.98CB36235)*, pp. 179–190, July 1998.

[53] X. Zhou, H. Chen, S. Luo, Y. Gao, S. Yan, W. Liu, B. Lewis, and B. Saha, "A case for software managed coherence in many-core processors," *Proceedings of the 2nd USENIX Conference on Hot Topics in Parallelism (HOTPAR'10)*, 06 2010.

[54] N. D. E. Jerger, E. L. Hill, and M. H. Lipasti, "Friendly fire: understanding the effects of multiprocessor prefetches," in *2006 IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 177–188, March 2006.

[55] P. Liu, J. Yu, and M. C. Huang, "Thread-aware adaptive prefetcher on multicore systems: Improving the performance for multithreaded workloads," *ACM Trans. Archit. Code Optim.*, vol. 13, pp. 13:1–13:25, Mar. 2016.

[56] B. Panda and S. Balachandran, "Introducing thread criticality awareness in prefetcher aggressiveness control," in *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1–6, March 2014.

[57] B. Panda and S. Balachandran, "Hardware prefetchers for emerging parallel applications," in *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 485–485, 2012.

[58] L. M. AlBarakat, P. V. Gratz, and D. A. Jimeñez, "Mtb-fetch: Multithreading aware hardware prefetching for chip multiprocessors," *IEEE Computer Architecture Letters*, vol. 17, no. 2, pp. 175–178, 2018.

[59] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, pp. 1–7, Aug. 2011.

[60] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," Tech. Rep. TR-811-08, Princeton University, January 2008.

[61] S. H. Pugsley, A. Alameldeen, and C. Wilkerson, "The first data prefetching championship (dpc-1)," 2009.

[62] A. R. Alameldeen, S. H. Pugsley, and M. Ferdman, "The 3rd data prefetching championship," *https://dpc3.compas.cs.stonybrook.edu*, 2019.

[63] S. Sardashti and D. A. Wood, "Decoupled compressed cache: Exploiting spatial locality for energy-optimized compressed caching," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, (New York, NY, USA), pp. 62–73, Association for Computing Machinery, 2013.

[64] S. Sardashti and D. A. Wood, "Decoupled compressed cache: Exploiting spatial locality for energy optimization," *IEEE Micro*, vol. 34, no. 3, pp. 91–99, 2014.

[65] A. Ghasemazar, P. Nair, and M. Lis, "Thesaurus: Efficient cache compression via dynamic clustering," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, (New York, NY, USA), pp. 527–540, Association for Computing Machinery, 2020.

[66] S. Hong, B. Abali, A. Buyuktosunoglu, M. B. Healy, and P. J. Nair, "Touché: Towards ideal and efficient cache compression by mitigating tag area overheads," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, (New York, NY, USA), pp. 453–465, Association for Computing Machinery, 2019.

[67] S. Hong, P. J. Nair, B. Abali, A. Buyuktosunoglu, K.-H. Kim, and M. B. Healy, "Attaché: Towards ideal memory compression by mitigating metadata bandwidth overheads," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, pp. 326–338, IEEE Press, 2018.

[68] J. Kim, S. H. Pugsley, P. V. Gratz, A. N. Reddy, C. Wilkerson, and Z. Chishti, "Path confidence based lookahead prefetching," in *Microarchitecture (MICRO), 2016 49rd Annual IEEE/ACM International Symposium on*, IEEE, 2016.

[69] P. Michaud, "A best-offset prefetcher," in *High Performance Computer Architecture (HPCA), 2016 IEEE 20th International Symposium on*, IEEE, 2016.

[70] A. Saporito, M. Recktenwald, C. Jacobi, G. Koch, D. P. D. Berger, R. J. Sonnelitter, C. R. Walters, J. . S. Lee, C. Lichtenau, U. Mayer, E. Herkel, S. Payer, S. M. Mueller, V. K. Papazova, E. M. Ambroladze, and T. C. Bronson, "Design of the ibm z15 microprocessor," *IBM Journal of Research and Development*, vol. 64, no. 5/6, pp. 7:1–7:18, 2020.

[71] T. Singh, S. Rangarajan, D. John, C. Henrion, S. Southard, H. McIntyre, A. Novak, S. Kosonocky, R. Jotwani, A. Schaefer, E. Chang, J. Bell, and M. Co, "3.2 zen: A next-generation high-performance x86 core," in *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, pp. 52–53, 2017.

[72] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores: Reducing the energy of mature computations," in *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, (New York, NY, USA), pp. 205–218, Association for Computing Machinery, 2010.

[73] L. Villa, M. Zhang, and K. Asanović, "Dynamic zero compression for cache energy reduction," in *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 33, (New York, NY, USA), pp. 214–220, Association for Computing Machinery, 2000.

[74] Jun Yang and R. Gupta, "Energy efficient frequent value data cache design," in *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002. (MICRO-35). Proceedings.*, pp. 197–207, 2002.

[75] D. H. Albonesi, "Selective cache ways: on-demand cache resource allocation," in *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 248–259, 1999.

[76] N. Bellas, I. Hajj, and C. Polychronopoulos, "Using dynamic cache management techniques to reduce energy in a high-performance processor," in *Proceedings. 1999 International Symposium on Low Power Electronics and Design (Cat. No.99TH8477)*, pp. 64–69, 1999.

[77] K. Flautner, Nam Sung Kim, S. Martin, D. Blaauw, and T. Mudge, "Drowsy caches: simple techniques for reducing leakage power," in *Proceedings 29th Annual International Symposium on Computer Architecture*, pp. 148–157, 2002.

[78] An-Chow Lai, C. Fide, and B. Falsafi, "Dead-block prediction dead-block correlating prefetchers," in *Proceedings 28th Annual International Symposium on Computer Architecture*, pp. 144–154, 2001.

[79] D. C. Burger, J. R. Goodman, and A. Kagi, "The declining effectiveness of dynamic caching for general-purpose microprocessors," tech. rep., 1995.

[80] H. Liu, M. Ferdman, J. Huh, and D. Burger, "Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency," in *2008 41st IEEE/ACM International Symposium on Microarchitecture*, pp. 222–233, 2008.

[81] L. M. AlBarakat, P. V. Gratz, and D. A. Jiménez, "Sb-fetch: Synchronization aware hardware prefetching for chip multiprocessors," in *Proceedings of the 34th ACM International Conference on Supercomputing*, ICS '20, (New York, NY, USA), Association for Computing Machinery, 2020.

[82] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial memory streaming," in *33rd International Symposium on Computer Architecture (ISCA'06)*, pp. 252–263, 2006.

[83] G. Pekhimnko, V. Seshadri, Y. Kim, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Linearly compressed pages: A low-complexity, low-latency main memory compression framework," in *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 172–184, 2013.

[84] A. Arelakis and P. Stenstrom, "Sc2: A statistical compression cache scheme," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pp. 145–156, IEEE Press, 2014.

[85] A. Hartstein, V. Srinivasan, T. R. Puzak, and P. G. Emma, "Cache miss behavior: Is it *sqrt*(2)?," in *Proceedings of the 3rd Conference on Computing Frontiers*, CF '06, (New York, NY, USA), pp. 313–320, Association for Computing Machinery, 2006.

[86] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial memory streaming," in *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ISCA '06, (Washington, DC, USA), pp. 252–263, 2006.

[87] E. Bhatia, G. Chacon, S. Pugsley, E. Teran, P. V. Gratz, and D. A. Jiménez, "Perceptron-based prefetch filtering," in *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, (New York, NY, USA), p. 1âĂŞ13, Association for Computing Machinery, 2019.

[88] B. Patel, N. Hardavellas, and G. Memik, "Scp: Synergistic cache compression and prefetching," in *2015 33rd IEEE International Conference on Computer Design (ICCD)*, pp. 164–171, 2015.

[89] S. Khan, D. Burger, D. A. JimÃĬnez, and B. Falsafi, "Using dead blocks as a virtual victim cache," in *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 489–500, 2010.

[90] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely, and J. Emer, "SHiP: signature-based hit predictor for high performance caching," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture - MICRO-44 '11*, p. 430, ACM Press.

[91] D. A. Jiménez and E. Teran, "Multiperspective reuse prediction," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture - MICRO-50 '17*, MICRO '17, pp. 436–448, IEEE.

[92] Z. Shi, X. Huang, A. Jain, and C. Lin, "Applying deep learning to the cache replacement problem," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, pp. 413–425, Association for Computing Machinery.

[93] J. Kim, E. Teran, P. V. Gratz, D. A. Jiménez, S. H. Pugsley, and C. Wilkerson, "Kill the program counter: Reconstructing program behavior in the processor cache hierarchy," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, (New York, NY, USA), pp. 737–749, ACM, 2017.

[94] A. Jain and C. Lin, "Rethinking belady's algorithm to accommodate prefetching," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 110–123, June 2018.

[95] A. J. Smith, "Sequential program prefetching in memory hierarchies," *Computer*, vol. 11, no. 12, pp. 7–21, 1978.

[96] R. Bera, A. V. Nori, O. Mutlu, and S. Subramoney, "Dspatch: Dual spatial pattern prefetcher," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, (New York, NY, USA), p. 531âĂŞ544, Association for Computing Machinery, 2019.

[97] N. Charmchi, C. Collange, and A. Seznec, "Compressed cache layout aware prefetching," in *2019 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 25–28, 2019.

[98] "The champsim simulator," *https://github.com/ChampSim/ChampSim*.

[99] H. Kim, "The 2nd data prefetching championship," *http://comparch-conf.gatech.edu/dpc2/*, 2015.

[100] A. R. Alameldeen, C. Wilkerson, S. H. Pugsley, A. Jaleel, B. Falsafi, M. Sutherland, and J. Picorel, "The 2nd cache replacement championship," *https://crc2.ece.tamu.edu*, 2017.

[101] S. H. Pugsley, A. R. Alameldeen, and M. Al-otoom, "The 1st instruction prefetching championship," *https://research.ece.ncsu.edu/ipc/welcome/*, 2020.

[102] D. A. Jimenez and C. Lin, "Dynamic branch prediction with perceptrons," in *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, pp. 197–206, 2001.

[103] "Standard performance evaluation corporation cpu2017 benchmark suite," *https://www.spec.org/cpu2017/*.

[104] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using simpoint for accurate and efficient simulation," in *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '03, (New York, NY, USA), pp. 318–319, Association for Computing Machinery, 2003.

[105] A. Perais, A. Seznec, P. Michaud, A. Sembrant, and E. Hagersten, "Cost-effective speculative scheduling in high performance processors," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pp. 247–259, 2015.

[106] B. Fields, S. Rubin, and R. Bodik, "Focusing processor policies via critical-path prediction," in *Proceedings 28th Annual International Symposium on Computer Architecture*, pp. 74–85, 2001.

[107] B. Fields, R. Bodik, and M. D. Hill, "Slack: maximizing performance under technological constraints," in *Proceedings 29th Annual International Symposium on Computer Architecture*, pp. 47–58, 2002.

[108] G. Memik, G. Reinman, and W. Mangione-Smith, "Precise instruction scheduling," *Journal of Instruction-Level Parallelism*, vol. 7, pp. 1–29, 04 2005.

[109] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan, "Speculation techniques for improving load related instruction scheduling," in *Proceedings of the 26th International Symposium on Computer Architecture (Cat. No.99CB36367)*, pp. 42–53, 1999.

[110] S. Ghose, H. Lee, and J. F. Martínez, "Improving memory scheduling via processor-side load criticality information," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, (New York, NY, USA), pp. 84–95, Association for Computing Machinery, 2013.