SB-Fetch: Synchronization Aware Hardware Prefetching for Chip Multiprocessors

Laith M. AlBarakat
Texas A&M University
lalbarakat@tamu.edu

Paul V. Gratz
Texas A&M University
pgratz@tamu.edu

Daniel A. Jiménez
Texas A&M University
djimenez@tamu.edu

ABSTRACT

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 paper 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 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 [4], Rodinia [7], and Parboil [39]. SB-Fetch improves execution time by 12.3% over baseline and 4% over best of class prefetching.

ICS '20, June 29-July 2, 2020, Barcelona, Spain

ACM ISBN 978-1-4503-7983-0/20/06…$15.00
https://doi.org/10.1145/3392717.3392735

CCS CONCEPTS

• **Computer systems organization** → **Multicore architectures**.

KEYWORDS

Shared-memory, chip multiprocessors, hardware prefetching

ACM Reference Format:
Laith M. AlBarakat, Paul V. Gratz, and Daniel A. Jiménez. 2020. SB-Fetch: Synchronization Aware Hardware Prefetching for Chip Multiprocessors. In *2020 International Conference on Supercomputing (ICS '20), June 29-July 2, 2020, Barcelona, Spain.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3392717.3392735

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 [11, 13]. 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 [20, 22, 37, 38]. Unfortunately, multi-threaded applications typically see little to no performance benefit from existing prefetching schemes. Figure 1 shows the speedup of multi-threaded applications under a previously proposed prefetching scheme [20, 33]. 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

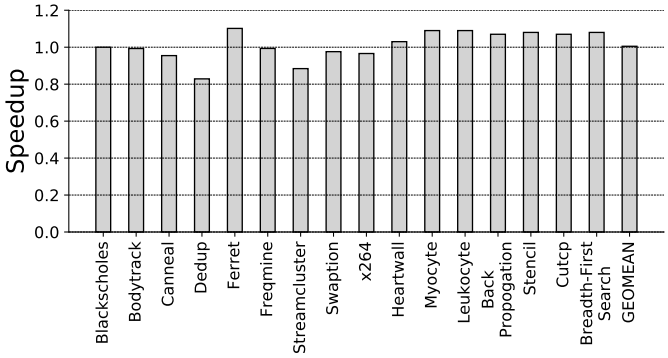


Figure 1: Speedup of Parsec [4], Rodinia [7], and Parboil [39] workloads with *B-Fetch* [20, 33], normalized against a no-prefetching baseline.

several are memory bound [4, 11][1]. There are two main 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 [20, 33]), 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 [28, 38].

[1] On single-threaded, multi-programmed workloads, B-Fetch sees an average gain of 31%[20].

2 MOTIVATION AND BACKGROUND

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 [36], a set of lines following a strided pattern with respect to the current miss [8], or a set of blocks spatially around the miss [38]. More recent prefetchers attempt to predict complex, irregular access patterns [15, 17, 22, 28, 35, 38]. 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 [20, 33], 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 [24] and Izraelevitz *et al* discuss how a policy of "always-abort" can improve performance for hardware transactional memory [16]. 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.

2.1.1 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 paper. Thus we here present a recap of the B-Fetch microarchitecture as originally published [20, 33].

Program construction can be mapped into a control flow graph as shown in Figure 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 (A), (B) and (C). 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.

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

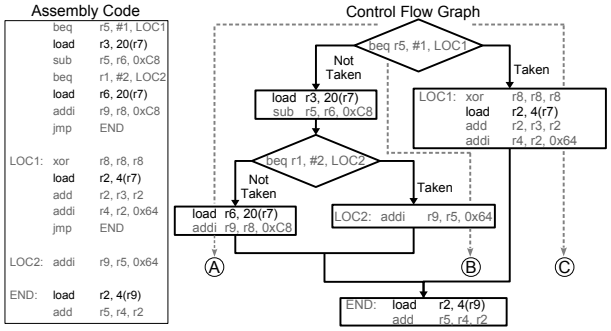


Figure 2: Data Access and Control Flow.

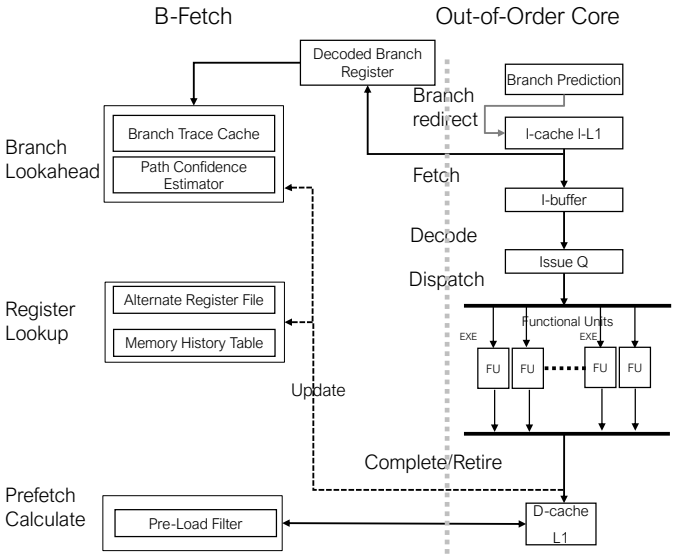


Figure 3: B-Fetch microarchitecture.

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 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.

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 [20, 33]. 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 [30]. Thus, we feel that B-Fetch makes an ideal starting point for attempting to efficiently issue prefetches beyond synchronization points.

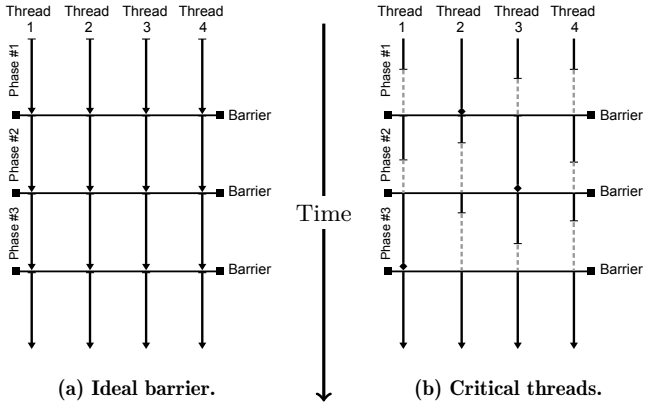
2.2 The Shared Memory Model

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.

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 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 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 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.

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 [10, 18]. 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.

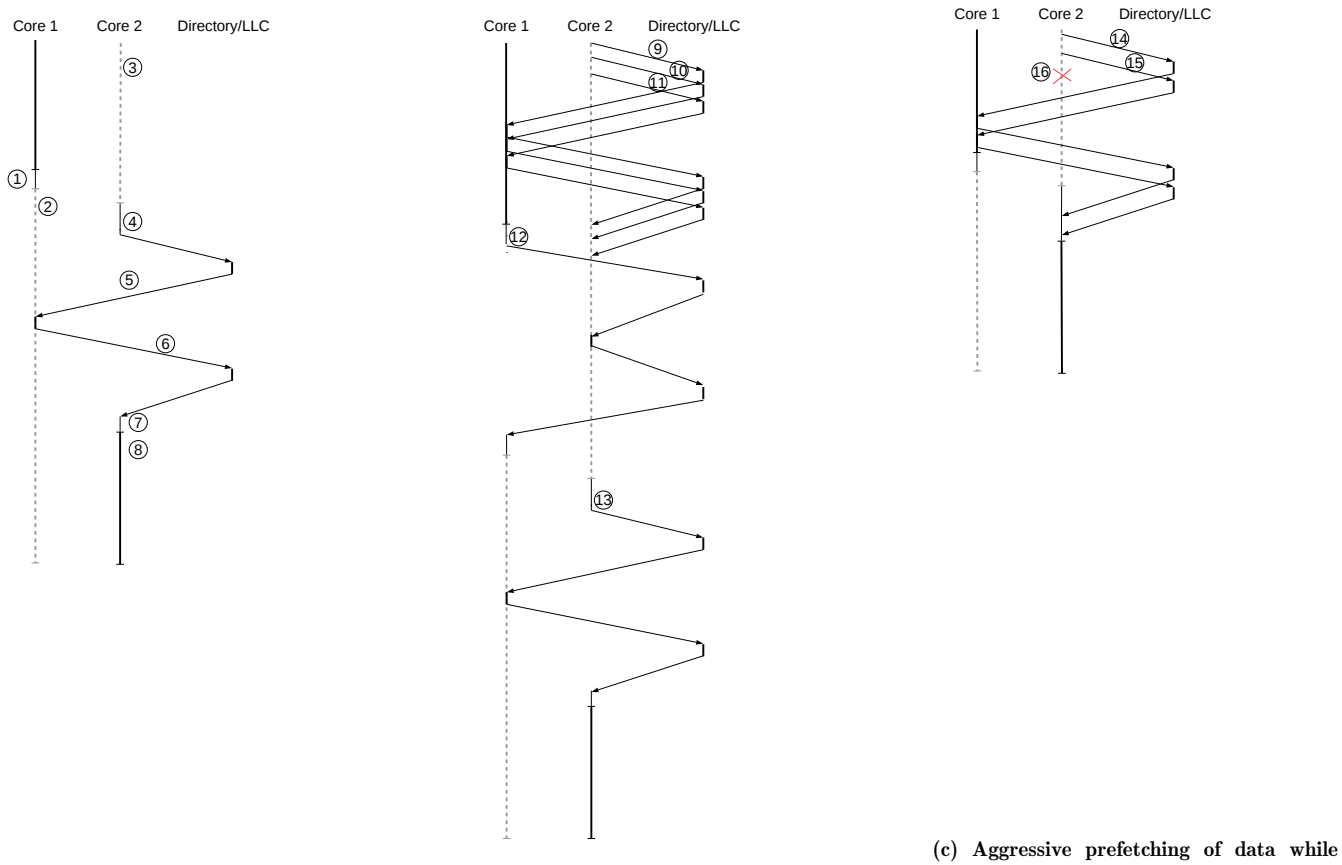
For a thread to acquire the lock, it needs to load the lock and check if no other thread is holding the lock. After that it needs to own the lock. If the thread fails to acquire the lock, it will stay in a spin loop until it successfully acquires it. Once a thread acquires the lock it is safe to execute 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 3.2 we briefly discus the changes required to support CISC ISAs.

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 [3, 9, 14, 23, 26, 27]. 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 [12, 21, 29, 40].

Figure 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,



(a) No-prefetching case. **(b) Premature prefetch of shared data.** **(c) Aggressive prefetching of data while avoiding premature prefetches.**

Figure 5: Cache Coherence and Prefetching.

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.*

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 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 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 1.

As shown in Figure 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.,* [19] 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.,* [25] and Panda *et al.,* [32] 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, [31] proposes a hardware prefetching framework that studies and classifies L1 misses across all threads to generate L2 cache prefetches. In our preliminary work [2], we initially examined the feasibility of prefetching for multithreaded workloads. Here we expand upon this prior work.

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 1.

3.1 Overview

SB-Fetch is an extension to the prior work *B-Fetch* prefetcher described in Section 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"

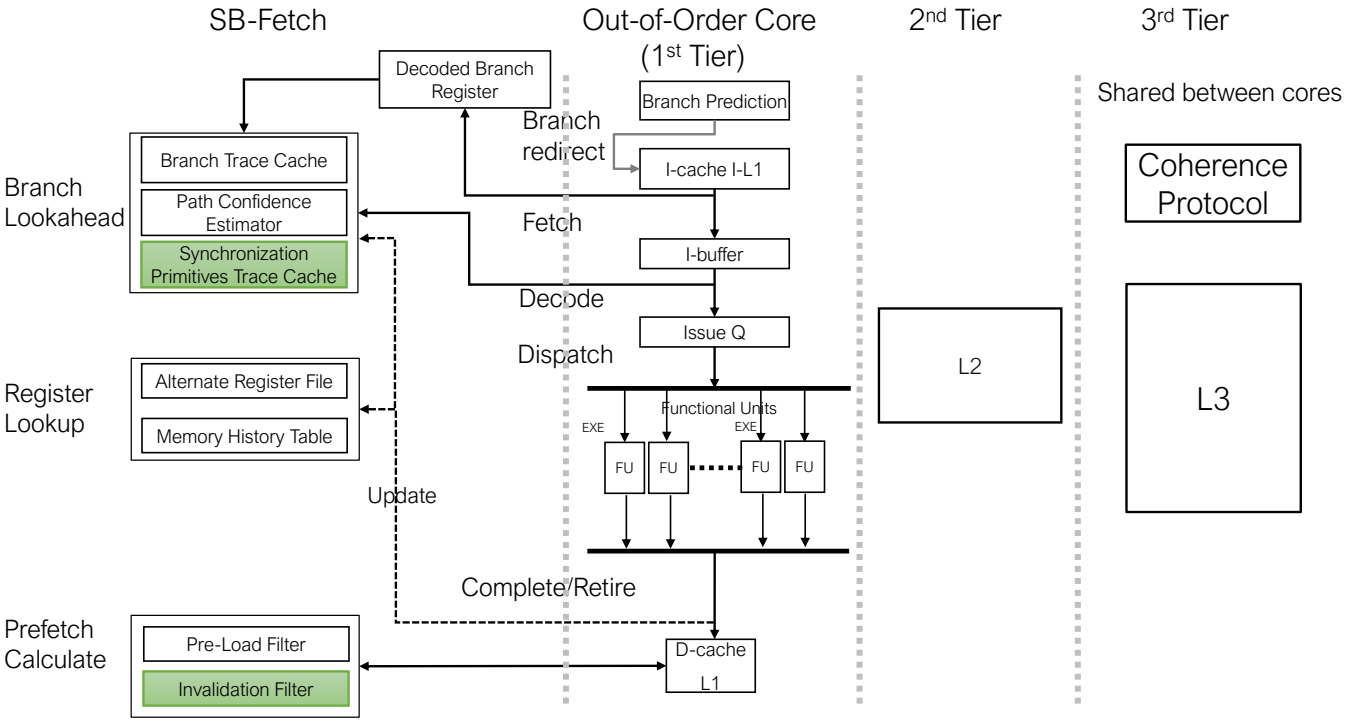


Figure 6: *SB-Fetch* microarchitecture. Additional components beyond *B-Fetch* highlighted in green.

Figure 7: Single Synchronization Primitives Trace Cache (SPTC) entry.

prefetches, prohibiting them from being prefetched in the future.

Figure 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 [20, 33].

3.2 System Components

As previously described, Figure 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 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 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.

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 Hardware Cost

The additional hardware storage requirements for SB-Fetch, B-Fetch, BOP and SMS are summarized in Table 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 used by Somogyi, *et al.* [37] 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.

4 EVALUATION

4.1 Methodology

We used gem5 [6], a cycle accurate simulator, to evaluate SB-Fetch. The baseline configuration is summarized in Table 2. We used a set of nine multi-threaded programs from PARSEC benchmark suite [5], four applications from the Rodinia benchmark suite [7], and three benchmarks from the Parboil benchmark suite [39]. 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.

Table 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**

The baseline hardware is a 4-core CMP machine with three level cache hierarchy as specified in Table 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 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 [34], and Second [1] 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 [20].

4.2 Results

4.2.1 Performance. Figure 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).

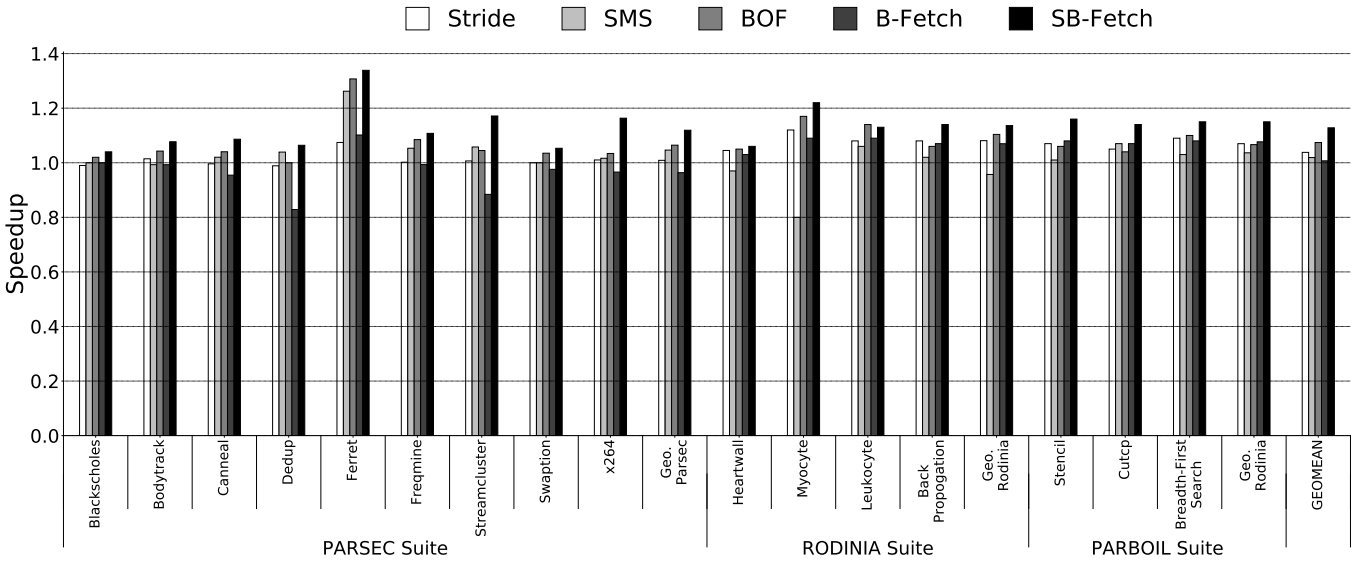


Figure 8: Multi-threaded workload speedups.

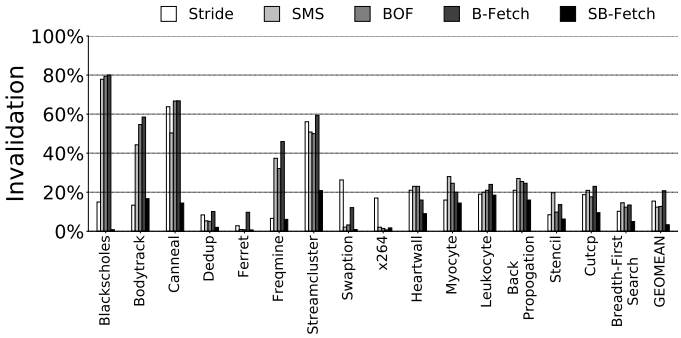


Figure 9: Invalidated prefetches for each prefetcher across all benchmarks.

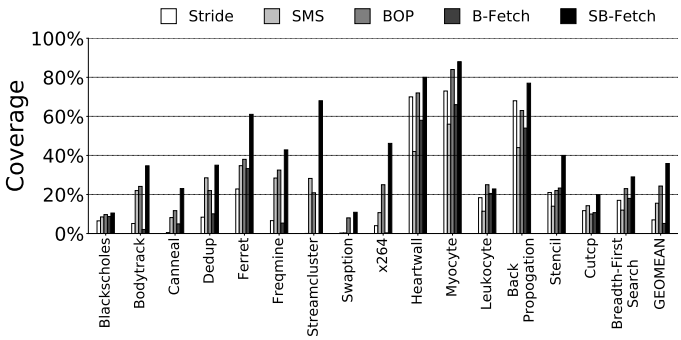


Figure 10: Coverage for each prefetcher across all benchmarks.

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.

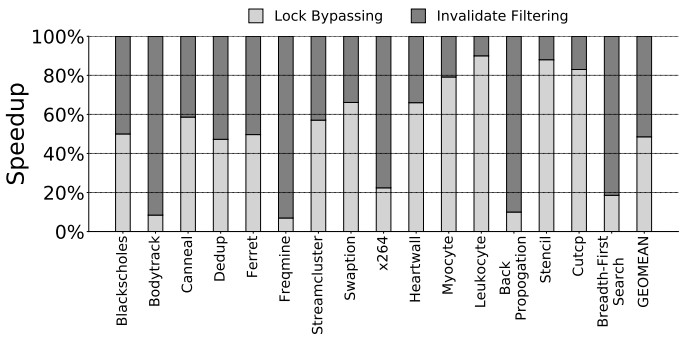


Figure 11: SB-Fetch speedup breakdown by mechanism.

Figure 11 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.

4.2.2 Coverage and Accuracy Analysis. Figure 12 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 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.

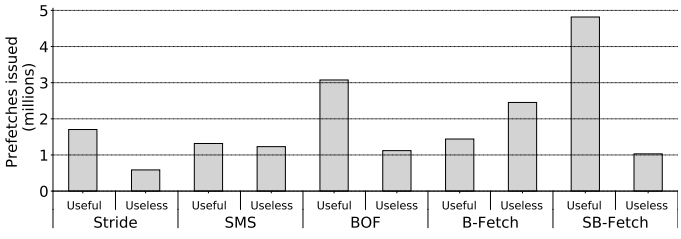


Figure 12: Useful and useless prefetches issued, averaged across all benchmarks for each prefetcher.

Figure 9 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 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%.

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.

Invalidation Filter Size: Figure 14 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.

Synchronization Primitives Trace Cache: Figure 15 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.

Branch Confidence: Figure 16 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.

Scalability: Finally, figure 13 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%.

5 CONCLUSION

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 paper 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

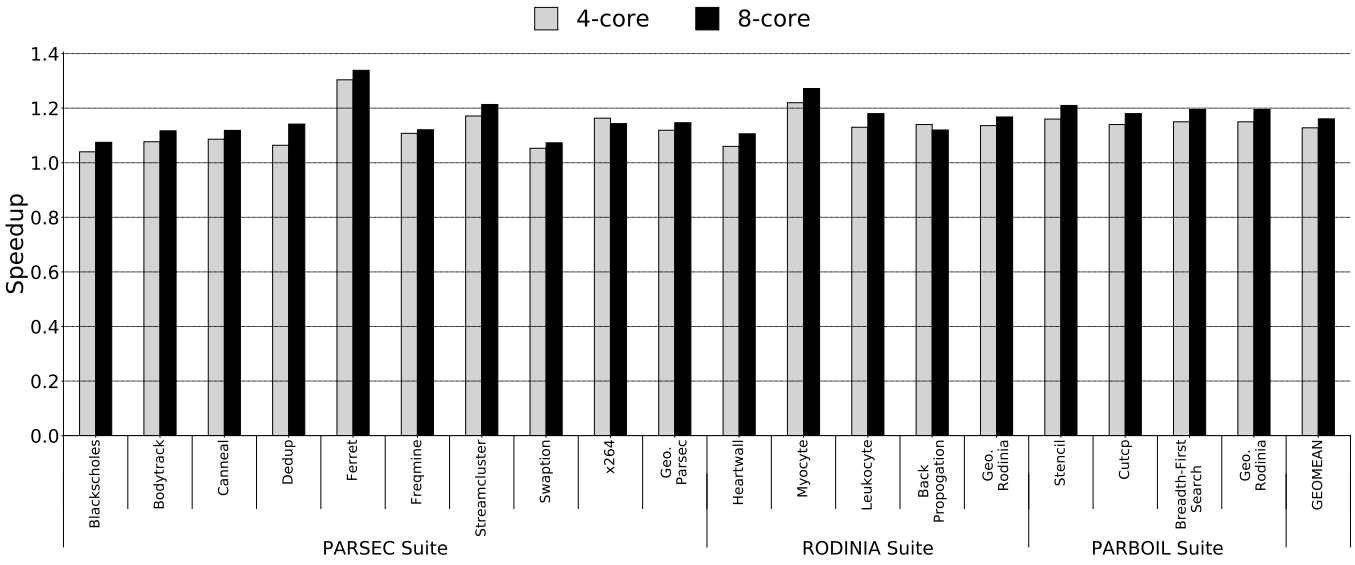


Figure 13: SB-Fetch speedup comparison for 4- and 8-core CMPs normalized to the baseline.22

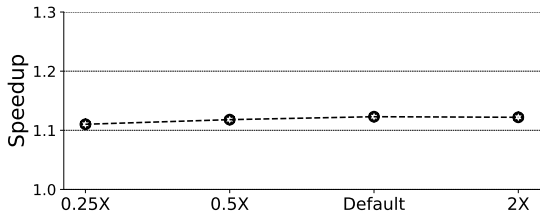


Figure 14: Invalidation Filter Size Sensitivity.

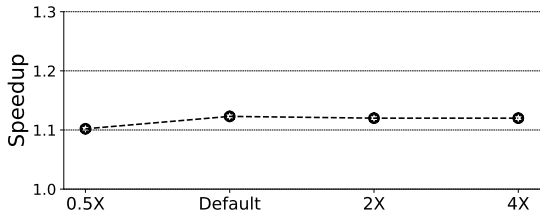


Figure 15: Synchronization Primitives Trace Cache Sensitivity.

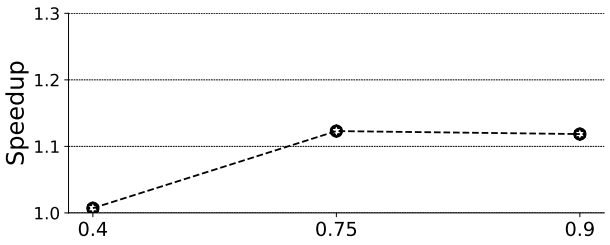


Figure 16: Branch confidence sensitivity.

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.

ACKNOWLEDGMENTS

We thank the National Science Foundation, which partially supported this work through grants I/UCRC-1439722, CCF-1823403 and CCF-1912617 and Intel Corp. for their generous support.

REFERENCES

[1] Alaa Alameldeen, Zeshan Chishti, Aamer Jaleel, Daniel Luchi, and Chris Wilkerson. 2015. The Second Data Prefetching Championship (DPC-2).

[2] L. M. AlBarakat, P. V. Gratz, and D. A. JimÃĪnez. 2018. MTB-Fetch: Multithreading Aware Hardware Prefetching for Chip Multiprocessors. *IEEE Computer Architecture Letters* 17, 2 (2018), 175–178.

[3] James Archibald and Jean-Loup Baer. 1986. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Trans. Comput. Syst.* 4, 4 (Sept. 1986), 273–298. https://doi.org/10.1145/6513.6514

[4] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*.

[5] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. *The PARSEC Benchmark Suite: Characterization and Architectural Implications.* Technical Report TR-811-08. Princeton University.

[6] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7.

[7] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the 2009 IEEE International Symposium on*

Workload Characterization (IISWC) (IISWC '09). IEEE Computer Society, Washington, DC, USA, 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[8] Tienfu Chen and Jeanloup Baer. 1995. Effective Hardware-based Data Prefetching for High-performance Processors. *IEEE Trans. Comput.* 44 (1995), 609–623.

[9] Alan L. Cox and Robert J. Fowler. 1993. Adaptive Cache Coherency for Detecting Migratory Shared Data. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA '93)*. ACM, New York, NY, USA, 98–108. https://doi.org/10.1145/165123.165146

[10] Digital. 1992. *Alpha Architecture Handbook*. Digital Press. https://books.google.com/books?id=JmoiAQAAMAAJ

[11] Ehsan Fatehi and Paul Gratz. 2014. ILP and TLP in Shared Memory Applications: A Limit Study. In *the 23rd International Conference on Parallel Architectures and Compilation (PACT)*. 113–126.

[12] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi. 2011. Cuckoo directory: A scalable directory for many-core systems. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. 169–180. https://doi.org/10.1109/HPCA.2011.5749726

[13] Mark D Hill and Michael R Marty. 2008. Amdahl's law in the multicore era. *Computer* 41, 7 (2008).

[14] Jaehyuk Huh, Jichuan Chang, Doug Burger, and Gurindar S. Sohi. 2004. Coherence Decoupling: Making Use of Incoherence. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*. ACM, New York, NY, USA, 97–106. https://doi.org/10.1145/1024393.1024406

[15] Yasuo Ishii, Mary Inaba, and Kei Hiraki. 2012. Unified memory optimizing architecture: memory subsystem control with a unified predictor. In *Proceedings of the 26th ACM International Conference on Supercomputing*. ACM, 267–278.

[16] J. Izraelevitz, L. Xiang, and M. L. Scott. 2017. Performance Improvement via Always-Abort HTM. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 79–90. https://doi.org/10.1109/PACT.2017.16

[17] Akanksha Jain and Calvin Lin. 2013. Linearizing irregular memory accesses for improved correlated prefetching.. In *MICRO*. 247–259.

[18] Eric H Jensen, Gary W Hagensen, and Jeffrey M Broughton. 1987. *A new approach to exclusive data access in shared memory multiprocessors*. Technical Report. Technical Report UCRL-97663, Lawrence Livermore National Laboratory.

[19] N. D. E. Jerger, E. L. Hill, and M. H. Lipasti. 2006. Friendly fire: understanding the effects of multiprocessor prefetches. In *2006 IEEE International Symposium on Performance Analysis of Systems and Software*. 177–188. https://doi.org/10.1109/ISPASS.2006.1620802

[20] David Kadjo, Jinchun Kim, Prabal Sharma, Reena Panda, Paul Gratz, and Daniel Jimenez. 2014. B-Fetch: Branch Prediction Directed Prefetching for Chip-Multiprocessors. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. 623–634.

[21] S. Kaxiras and G. Keramidas. 2010. SARC Coherence: Scaling Directory Cache Coherence in Performance and Power. *IEEE Micro* 30, 5 (Sept 2010), 54–65. https://doi.org/10.1109/MM.2010.82

[22] J. Kim, S. H. Pugsley, P. V. Gratz, A. L. N. Reddy, C. Wilkerson, and Z. Chishti. 2016. Path confidence based lookahead prefetching. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. https://doi.org/10.1109/MICRO.2016.7783763

[23] James Laudon and Daniel Lenoski. 1997. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA '97)*. ACM, New York, NY, USA, 241–251. https://doi.org/10.1145/264107.264206

[24] J. Lee, N. B. Lakshminarayana, H. Kim, and R. Vuduc. 2010. Many-Thread Aware Prefetching Mechanisms for GPGPU Applications. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. 213–224. https://doi.org/10.1109/MICRO.2010.44

[25] Peng Liu, Jiyang Yu, and Michael C. Huang. 2016. Thread-Aware Adaptive Prefetcher on Multicore Systems: Improving the Performance for Multithreaded Workloads. *ACM Trans. Archit. Code Optim.* 13, 1, Article 13 (March 2016), 25 pages. https://doi.org/10.1145/2890505

[26] M. M. K. Martin, D. J. Sorin, M. D. Hill, and D. A. Wood. 2002. Bandwidth adaptive snooping. In *Proceedings Eighth International Symposium on High Performance Computer Architecture*. 251–262. https://doi.org/10.1109/HPCA.2002.995715

[27] M. R. Marty and M. D. Hill. 2006. Coherence Ordering for Ring-based Chip Multiprocessors. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. 309–320. https://doi.org/10.1109/MICRO.2006.14

[28] P. Michaud. 2016. Best-offset hardware prefetching. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 469–480. https://doi.org/10.1109/HPCA.2016.7446087

[29] S. S. Mukherjee and M. D. Hill. 1998. Using prediction to accelerate coherence protocols. In *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No.98CB36235)*. 179–190. https://doi.org/10.1109/ISCA.1998.694773

[30] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. 2003. 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*. 129–140.

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

[32] B. Panda and S. Balachandran. 2014. Introducing Thread Criticality awareness in Prefetcher Aggressiveness Control. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1–6. https://doi.org/10.7873/DATE.2014.092

[33] Reena Panda, Paul Gratz, and Daniel Jimenez. 2012. B-Fetch: Branch Prediction Directed Prefetching for In-Order Processors. *IEEE Comput. Archit. Lett.* 11, 2 (July 2012), 41–44.

[34] Seth H Pugsley, Alaa Alameldeen, and Chris Wilkerson. 2009. The First Data Prefetching Championship (DPC-1).

[35] Manjunath Shevgoor, Sahil Koladiya, Rajeev Balasubramonian, Chris Wilkerson, Seth H Pugsley, and Zeshan Chishti. 2015. Efficiently Prefetching Complex Address Patterns. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture*.

[36] A. J. Smith. 1978. Sequential Program Prefetching in Memory Hierarchies. *Computer* 11 (December 1978), 7–21. Issue 12.

[37] Stephen Somogyi, Thomas F. Wenisch, Anastasia Ailamaki, and Babak Falsafi. 2009. Spatio-temporal Memory Streaming. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. 69–80.

[38] Stephen Somogyi, Thomas F. Wenisch, Anastassia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2006. Spatial Memory Streaming. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA '06)*. Washington, DC, USA, 252–263.

[39] John A. Stratton, Christopher I. Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Liu, and Wen mei W. Hwu. 2012. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing.

[40] Xiaocheng Zhou, Hu Chen, Sai Luo, Ying Gao, Shoumeng Yan, Wei Liu, Brian Lewis, and Bratin Saha. 2010. 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).