MULTITHREADING AWARE HARDWARE PREFETCHING FOR CHIP MULTIPROCESSORS

A Thesis

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

MASTER OF SCIENCE

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

August 2017

Major Subject: Computer Engineering

Copyright 2017 Laith Mohammad Albarakat

ABSTRACT

To take advantage of the processing power in the Chip Multiprocessors design, applications must be divided into semi-independent processes, that can run concurrently on multiple cores within a system. Therefore, programmers must insert thread synchronization semantics (i.e. locks, barriers, and condition variables) to synchronize data access between processes. Indeed, threads spend long time waiting to acquire the lock of a critical section. In addition, a processor have to stall execution to wait for load data accesses to complete. Furthermore, there are often independent instructions which include load instructions beyond synchronization semantics that could be executed in parallel while a thread waits on the synchronization semantics. The conveniences of the cache memories come with some extra cost in Chip Multiprocessors. Cache Coherence mechanisms address the Memory Consistency problem. However, Cache Coherence adds considerable overhead to memory accesses. Having aggressive prefetcher on different cores of a Chip Multiprocessor, can definitely lead to significant system performance degradation when running multi-threaded applications. This result of prefetch-demand interference when a prefetcher in one core ends up pulling shared data from a producing core before it has been written, the cache block will end up transitioning back and forth between the cores and result in useless prefetch, saturating the memory bandwidth and substantially increase the latency to critical shared data.

We present a hardware prefetcher that enables large performance improvements from prefetching in Chip Multiprocessors by significantly reducing prefetch-demand interference. Furthermore, it will utilize the time that a thread spends waiting on synchronization semantics to run ahead of the critical section to speculate and prefetch

independent load instruction data beyond the synchronization semantics.

DEDICATION

To My family

ACKNOWLEDGEMENTS

I would like to express my deep gratitude and appreciation to my research adviser, Dr. Paul V. Gratz for the opportunity to participate in research with the Computer Architecture, Memory Systems and Interconnection Networks(CAMSIN) research group. He has been a great support and a source of motivation and enthusiasm through my academic journey at Texas A&M University.

I would like to express my sincere gratitude to my advisory committee members Dr. Daniel Jimenez and Dr. I-Hong Hou for their valuable feedbacks.

I would like to thank my family and friends for their unconditional support and their encouragement in all my endeavors.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supervised by a thesis committee consisting of Professor Paul V. Gratz and Professor I-Hong Hou of the Department of Electrical and Computer Engineering and Professor Daniel A. Jiménez of the Department of Computer Science and Engineering.

All work for the thesis was completed by the student, under the advisement of Professor Paul V. Gratz of the Department of Electrical and Computer Engineering.

Funding Sources

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

NOMENCLATURE

CMP	Chip Multiprocessors
POSIX	Portable Operating System Interface
RISC	Reduced Instruction Set Computer
CISC	Complex Instruction Set Computer
ISA	Instruction Set Architecture

TABLE OF CONTENTS

Page

ABSTRACT ii

DEDICATION iv

ACKNOWLEDGEMENTS v

CONTRIBUTORS AND FUNDING SOURCES vi

NOMENCLATURE vii

TABLE OF CONTENTS viii

LIST OF FIGURES x

LIST OF TABLES xi

1. INTRODUCTION 1

1.1 Thesis Statement 5
1.2 Document Organization 6

2. BACKGROUND AND PRIOR WORK 7

2.1 Programming Model for Shared Memory 7
2.1.1 Shared Memory Synchronization 9
2.2 Architectural Support for Shared Memory Synchronization 12
2.2.1 Atomic Operations 12
2.2.2 Barriers 13
2.2.3 Spinlocks 14
2.3 Cache implementation for Shared Memory 15
2.3.1 Cache Coherence 16
2.4 Data Prefetching Techniques 17

3. MULTITHREADING B-FETCH FOR CHIP MULTIPROCESSORS . . . 21

3.1 B-Fetch for Chip Multiprocessors 21
3.2 MTB-Fetch for Chip Multiprocessors 26
3.2.1 System Architecture 27

3.2.2 System Components 28
3.2.3 Hardware Cost 30

4. EVALUATION 32

4.1 Methodology 32
4.2 Results and Analysis 33

5. CONCLUSION AND FUTURE WORK 36

REFERENCES 37

LIST OF FIGURES

FIGURE Page

2.1 Shaere memory 8

2.2 Ideal barrier synchronizes 10

2.3 Critical threads in the execution phases. 10

2.4 A barrier makes the fastest task wait for the slowest task before it can proceed 11

2.5 Barrier in shared memory 14

2.6 Lock in shared memory 15

3.1 Data Access and Control Flow. 21

3.2 B-Fetch microarchitecture 23

3.3 ALPHA code for lock implementation. 26

3.4 MTB-Fetch microarchitecture. 27

3.5 Single Synchronization Primitives Trace Cache (SPTC) entry. 28

3.6 Invalidation filter table. 29

4.1 Multi-threaded workload speedups. 33

4.2 Normalized Geomean if IPC. 34

4.3 Number of useful and useless prefetches issued. 35

LIST OF TABLES

TABLE Page

3.1 Hardware storage overhead in KB, adopt from B-Fetch[17]. 31

4.1 Target Microarchitecture Parameters 32

1. INTRODUCTION

The increasing difference between processor and memory speed is known as "Memory Wall" [35]. Due to this difference, memory latencies have become a critical bottleneck for processor performance. To mitigate this bottleneck, current architectures adopt multi-level cash hierarchy that trade off capacity for lower latency at each level. The goal of the hierarchy is to improve the apparent average memory access time by frequently handling a memory request at the cache, avoiding the comparatively long access latency of Main Memory. Even with multi-level cash hierarchies the latency of misses can still be quite high. Prefetching is an approach that tries to reduce cache misses by predicting future memory accesses and issue requests for the corresponding memory blocks in advance of explicit accesses. To be effective, prefetcher must be accurate when predicting the correct address of future memory access and timely so that the prefetched data arrives at its destination before cache miss occurs. Despite these challenges, hardware prefetching can improve overall program execution time by overlapping computation with memory accesses.

Since the end of the frequency scaling era [33], there is a clear trend towards Chip Multiprocessors architectures. Current mobile phone already have multiple cores, other aggressive architectures such as Intel's TeraScale and Tilera's TILE-Gx100 use 80 and 100 cores respectively [7]. the scalability of these aggressive architectures are limited by the cache coherent overhead[14]. To take advantage of the processing power in the CMPs design, applications must be divided into semi-independent processes that can run concurrently on multiple cores within a system.

Writing parallel programs that fully exploit parallel architecture is more difficult than writing sequential programs due to coordination complexity. Coordination

arranged access to shared-data, manage parallelism, and handel inter-process communication. To enable parallel execution computation needs to be partitioned. To do so, programers frequently insert thread synchronization semantics (i.e. locks, barriers, and condition variables) to synchronize data access between threads, to ensuring correctness and avoid races condtions. Usually theses semantics lead to perfomance degrdations. Many progarming languages were designed with parallelism in mind. For example, the model of threads explicitly share memory and allow programmers to employ mechanisms such as locks to ensure mutually exclusive access to shared resources. To ensure mutual exclusion to shared resources, modern architectures provide instructions capable of updating (i.e. reading and writing) a memory location as a single atomic operation *viz.* Load Linked /Store Conditional (LL/SC) in Alpha, IBM PowerPC and ARM; and test-and-set and SWP in x86 and SPARC to construct locks, condition variables, barriers and spinlocks.

Chip Multiprocessors architectures are increasingly hierarchical regarding memory access. Multi-level caches are used to reduce the frequency of accesses to the relatively slow main memory to increase memory bandwidth and to hide access latency [24]. By sharing the same cache, cores may communicate more efficiently [24]. On the other hand, the advantage of having cache memories comes with some extra cost when the system has multiple processors. Copies of data which have been retrieved and modified by a processor in its local cache become inconsistent with the original copy in main memory. When another processor accesses the same data item it should receive the latest up-to-date copy and not an older version of it. Cache coherence mechanisms address the memory consistency problem. It ensures that the value of an item retrieved by any processor in the system is the most up-to-date one. However, Cache coherence adds considerable overhead in accessing memory in Chip Multiprocessors. Cache coherence logic is in the critical path of accessing memory

and can easily become the main bottleneck, exacerbating the processor and memory speed gap, leading to significant performance degradation.

Thread synchronization semantics (i.e. locks, barriers, and condition variables) are used to synchronize access to shared data between threads. Fatehi at el. [15] observed that there is as much as a 6x slowdown due to these synchronization semantics. The overheads of thread synchronization semantics scales with core count, dramatically reducing the overall scalability of the application. Furthermore, there are often independent instructions which include load instructions beyond a synchronization semantic that could be executed in parallel while a thread waits on the synchronization semantic. In Chip Multiprocessors, cache coherence is achieved by means of an invalidation-based coherence protocol.

Many hardware prefetchers with diverse prefetching strategies have been proposed in literature [30, 10, 32, 31, 34, 27, 4, 19, 29, 22]. Most existing prefetchers prefetch future memory block based on current cache misses. For example, sequential prefetchers prefetch the lines sequentially following the current miss [30] and stride prefetchers monitor memory accesses generated by memory instructions in order to identify constant-stride references. Then, prefetch lines show the strided pattern with respect to the current miss [10]. Region based prefetchers like Spatial Memory Streaming (SMS) [31] makes use of code-based correlation to take advantage of spatial locality over larger regions of memory in the applications. SMS records access patterns over spatial regions over a period of time called the spatial region generation. Then use the information to preftech future memory block in spatial region around a miss. One potential issue with SMS is that it cannot predict the first misses into a region. Branch-predictor-directed prefetchers like B-Fetch [26] use the core 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.

Data prefetchers [30, 32] are ineffective for parallel programs, due to the fact that they use local knowledge to decide which memory block to prefetch. The problem with parallel programs, they regularly to use the same data at almost the same time. Another well-known problem in parallel programs is that a finite-size buffer or producer/consumer in which one thread will generate data that will be consumed by another thread. Coherence protocol maintains agreement between all replicated copies when the data are modified by threads running on diffrent cores on the system. Having a prefetcher that does not pay attention to this buffering strategy and the coherence protocol ultimately leads to significant system performance degradation. In particular, when a prefetcher ends up pulling data from a producing core before its been written back, such a prefetcher will increase the number of useless prefetches. In addtion, threads may have to wait for a long time to acquire a lock of a critical section. Furthermore, a processor may have to stall for a long time waiting for all of its load data accesses to complete before releasing the lock.

B-Fetch [17] presents a novel idea that combineds branch prediction with prefetching. It can achieve high accuracy with low hardware overhead. B-Fetch, uses lookahead mechanism that reuses branch prediction to predict the future execution path. Detailed simulation using gem5[6] shows a geometric mean speedup of 23.4% for single-threaded workloads and 28.6% for multi-application workloads over a baseline system without prefetching. The current implementation of B-Fetch [17] operates on spinloop while a thread is waiting on a lock.

The contribution of this project is to develop a hardware prefetcher that uses the information about synchronization semantics to identify when a thread is spinning,

waiting on the lock or barrier. The spin time can be utilized to run ahead of the execution to speculate and prefetch data beyond synchronization semantics. The prefetcher will take into account the effect of coherence protocol. It will include learning mechanisms to minimize the number of invalidated prefetch memory block. To implement the proposed prefetcher we will extentd the current implementation of B-Fetch [17]. To do so, first, we need to add a trace cache to the decode stage in the Out-of-Order core to help identify synchronization semantics in the dynamic instruction stream. When a thread is spinning, waiting at a barrier or waiting to acquire a lock, and the core is not being used for any other purpose, B-Fetch [17] can run ahead the excution and prefetch data that can be useful in the future to all threads. The second component is the learning queue, which includes the prefetch memory block that were invalidated because of coherence protocol. Once a memory block is inserted into the queue, the prefetcher will never prefetch it again and ensure that the prefetch memory block will not result in coherence invalidation.

Detailed simulation using gem5 [6] simulator for the implementation and the PARSEC Benchmark Suite[5] compiled for the Alpha ISA shows a geometric mean speedup of 9.3% for multi-threaded workloads over a baseline system without prefetching.

1.1 Thesis Statement

This thesis proposes a hardware data prefetching mechanism to target multi-threaded workloads. The prefetcher leverages the synchronization primitives in the dynamic instructions stream. The prefetcher identify when a thread is waiting to acquire the lock, and utilize the time run ahead of the critical section to speculate and prefetch independent load instruction data beyond the synchronization semantics.

The proposed prefetcher outperforms the best in class lightweight prefetcher,

and chieves a geometric average speedup of 9.3% speedup over baseline with a low hardware impact.

To the best of our knowledge, we are not aware of existing prefetcher design which uses the waiting time on synchronization semantics, to ahead of the critical section to speculate and prefetch independent load instruction data beyond the synchronization semantics.

1.2 Document Organization

The rest of the thesis is organized as follows. Chapter 2 gives the required background which includes the programming model for shared memory, architectural support for shared memory synchronization, cache implementation for shared memory then review of data prefetching techniques. Chapter 3 reviews the orignal B-fetch then it describes the proposed prefetcher architecture in detail. Chapter 4 discusses our methodology and evaluation of results. Finally, Chapter 5 concludes this thesis and discusses future work.

2. BACKGROUND AND PRIOR WORK

This section reviews some related concepts before embarking on the specifics of the thesis. It presents shared memory programming models, shared memory synchronization, then hardware support for memory synchronization. Finally, it discusses the different data prefetching techniques that have been proposed in literature and compares our proposed solution against a few prefetchers.

2.1 Programming Model for Shared Memory

The Chip Multiprocessors revolution has cause the issue of parallel programming to be in the forefront of application development. Chip Multiprocessors replaced the conventional computational model with a parallel programming model due to the fact that multiple processors on a CMP are aailable to programmers as separate processing units. To take advantage of the processing power in the CMPs design, applications need to be written using different programming paradigms, in which applications must be divided into independent threads that can run simultaneously across the cores within a system. Once multi-threading has been implemented, programs can take advantage of thread-level parallelism (TLP) by running the separate threads in at the same time.

Transforming the algorithmic solution into a correct and efficient parallel program to fully exploits the proceesing power in the parallel architecture is a difficult task. Due to the coordination needed to manages access to shared-data, parallelism, and inter-process communication[18]. The two primary shared memory programming models are Pthreads and OpenMP. Pthreads or Portable Operating System Interface (POSIX) Threads is a set of C programming language types and procedure calls and is typically accessed via run-time library and operating system calls [8].

OpenMP extends the programming language (C, C++ or Fortran) with compiler directives to make parallelism and data privacy explicit. OpenMP annotations change the semantics of loops and data persistence and it is possible for OpenMP annotations to assert incorrect program transformations which manifest themselves as race conditions or deadlocks [9].

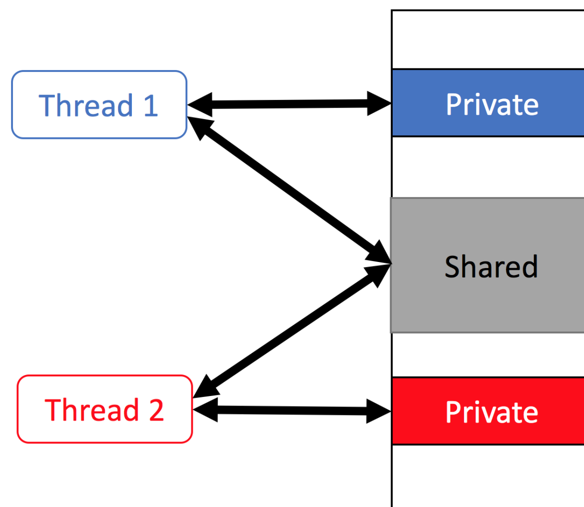

Figure 2.1: Shaere memory

In the POSIX model , the dynamically allocated heap memory and the global variables are shared among the threads. Moreover, sharing memory is the fastest interprocess communication mechanism between multiple threads. The operating system maps a memory segment in the address space of several threads. As figure 2.1 shows, by sharing the same memory, threads can communicate to coordinate their execution by using only the basic memory read and write operations without calling operating system functions. When multiple threads access the shared data, program-

mers have to be aware of race conditions and deadlocks. To protect critical section, i.e., the portion of code where only one thread must reach shared data, Pthreads provides mutex Mutexes and Condition Variables POSIX Mutexes, Condition Variables and Semaphores as primary means of implementing thread synchronization [2, 20].

OpenMP is a shared memory application programming interface that provides a portable, scalable model for programmers of shared memory parallel applications [9]. OpenMP is a set of compiler directives, pragmas, and a runtime that provide management of the thread pool and library routines [9]. The directive instruct the he compiler to create threads, perform synchronization operations, and manage shared memory. OpenMP provides a variety of Synchronization primitives that control how the execution of each thread proceeds relative to other team threads [9].

Regardless of the syntax of the parallel programming model, both of these models ultimately have shared memory and synchronization primitives that can be leveraged by our preftecher to identify when a thread is waiting to acquire the lock, and utilize the time to run ahead of the critical section to speculate and prefetch independent load instruction data beyond the synchronization semantics.

2.1.1 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.

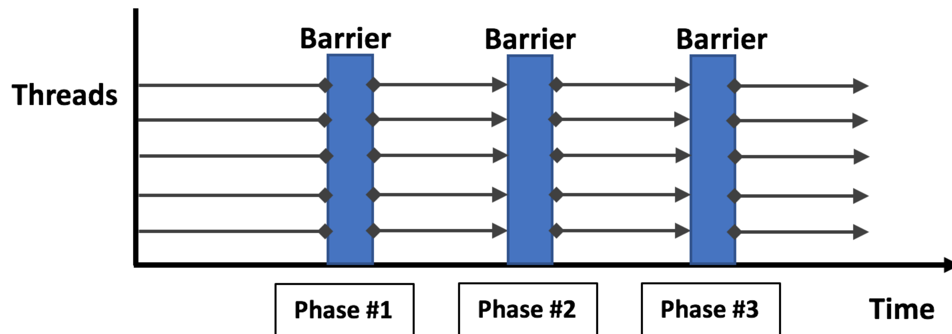

Figure 2.2: Ideal barrier synchronizes

Figure 2.2 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.

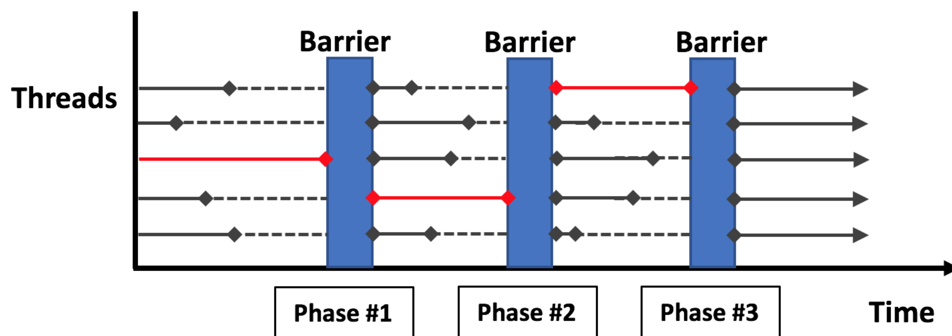

Figure 2.3: Critical threads in the execution phases.

Due to load imbalance between threads, different threads can be critical threads

during the execution phases. As figure 2.3 show other threads need to wait untill the critical thread get to the barrier before resume execution.

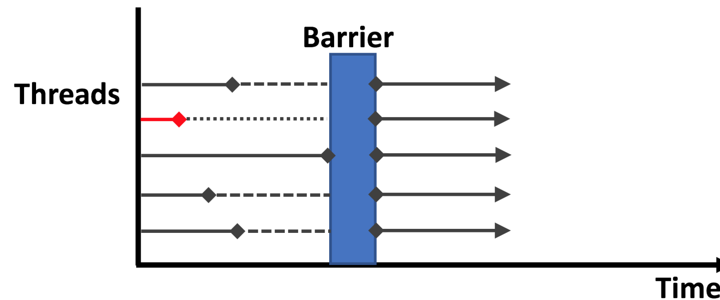

Figure 2.4: A barrier makes the fastest task wait for the slowest task before it can proceed

Implementing barrier synchronization is quite complex and often proves to be a performance bottleneck. A barrier is an expensive synchronization mechanism since the semantics of barriers require the computation to wait for the slowest task to arrive before the rest can proceed. Figure 2.4 shows how barrier synchronization can cause performance degradation by making all tasks wait for the slowest.

A lock is just a variable that holds the state of the lock at any instant in time. Locks can be used to provide individual threads with exclusive access to shared data and a critical section of code. The exclusive access applicable to fine-grained many shared memory parallel applications. To mininmize serialized processing and maximized parallelism fine-grained share littel data or code between threads.

POSIX also provides *Mutexes* which allow only one thread to lock a mutex variable at any given time. When multible threads try to lock a mutex, only one will

successed. Also, *Condition Variables* used to allows synchroniztion based on data value [25].

2.2 Architectural Support for Shared Memory Synchronization

There are different ways of supporting synchronization primitives in hardware. A widely known practice is to implement the lowest level of synchronization in the form of atomic instructions in hardware and to implement all other synchronization primitives on top of that in software. This section outline how synchronization semantics are implemented in shared memory Chip Multiprocessors.

2.2.1 Atomic Operations

Atomic operations are defined as operations whose execution is not interfered with by other concurrent activities. To facilitate the construction of synchronization primitives, most architectures provide read-modify-write instructions that are capable of updating (i.e., reading and writing) a memory location as a single atomic operation. Both RISC and CISC include atomic operations to support synchronization primitives. The *compare_and_swap* primitive was originally introduced in IBM 370 architecture [16]. It also exists in modern x86, IA-64, and SPARC machines. The *load_linked/store_conditional* primitive can be found in modern POWER, MIPS, and ARM machines. ALPHA ISA support *Load Linked (LL) and Store Conditional (SC)*, a Load Linked (LL) instruction loads a block of data into the cache. The following Store Conditional (SC) instruction attempts to write to the same block. It 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. To implement an atomic primitive, library routines typically retry the LL/SC pair repeatedly until the SC succeeds [12].

The problem with atomic read-modify-write instructions is that they result in in-

terprocessor communication in every atomic operation. When a core wants to modify a shared variable, it sends a message to the variable's home core to acquire exclusive ownership. In response, the home core typically sends invalidation messages to other cores sharing the data. The resulting latency severely impacts the performance of synchronization operations.

2.2.2 Barriers

Memory Barrier (MB) enforces a perceived ordering of memory operations, it ensures that all following loads or stores will not access memory until after all previous loads and stores have accessed memory, as observed by other processors. Memory barriers can be store or load barriers, store barriers ensure that all the store operations specified before the barrier will appear to happen before all the store operations specified after the barrier with respect to the other components of the system. On the other hand load barriers ensure that all the load operations specified before the barrier will appear to happen before all the load operations specified after the barrier with respect to the other components of the system [1].

[1]

Figure 2.5 shows how pairing barrier is used to ensure safe access to the critical section. Conceptually, it includes three steps 1) Acquire software lock; 2) Critical section - read/write shared data; 3) Clear software lock.

```
<acquire software lock>
MB(memory barrier #1)
<critical section - read/write shared data>
MB(memory barrier #2)
<clear software lock>
```

Figure 2.5: Barrier in shared memory

The first barrier in the code above prevents any reads from being prefetched before the lock is acquired. The second barrier prevents any writes and reads in the critical section being delayed past the clearing of the lock. Such delay may impcat the other users of shared data[12].

2.2.3 Spinlocks

A spinlock is a lock that make a thread wait in a spin loop while constantly checking if the lock is available. Because thread remains active but is not doing a useful task, the use of such technique is busy waiting. When the thread get hold of the spinlock, it will hold it, untill it finish executing the critical section then explicitly release it. Spinlocks are efficient if threads are going to be blocked for only short periods avoiding rescheduling or context switching from the operating system. Figure 2.6 shows an example using ALPHA assembly language to implement a spinlock.

Spinlocks are useful if threads will be blocked for only short periods avoiding rescheduling or context switching from the operating system. Figure 2.6 shows an example using ALPHA assembly language to implement a spinlock.

```
spin_loop:
        LDQ_L R1,lock_variable
        BLBS  R1,already_set
        OR    R1,#1,R2
        STQ_C R2,lock_variable
        BEQ   R2,stq_c_fail



   <critical section: updates
    various data structures>


        STQ_C R2,lock_variable



already_set:
        BR     spin_loop

stq_c_fail:
        BR    stq_c_loop
```

Figure 2.6: Lock in shared memory

Form the code above we can see, if the *lock_variable* is already set, the loack is already set and no stores will execute. If not the store will be executed. This loop use regulare load. This code increase the possibility that *LDQ_L* hits in the cache and the *LDQ_L*/*STQ_C* sequence is completed quickly and successfully. The *lock_variable* is actually being changed from 0 to 1, and the *STQ_C* fails due to an interrupt. Both conditional branches are forward branches, so they are most likly predicted not to be taken. Finally, an ordinary STQ instruction is used to clear the *lock_variable*.

2.3 Cache implementation for Shared Memory

This section presents the required hardware support to guarantee the correctness of exectuting shared memory program on Chip Multiprocessors. It highlights Cache Coherence and Synchronization Problem.

2.3.1 Cache Coherence

In Chip Multiprocessors, a coherent cache allows multiple copies of the same memory location to exist in multiple processors. It is critical to ensure that these copies are consistent by having processors broadcast the values of updates or invalidation, otherwise application errors will occur. Due to sharing data blocks, cache misses and memory traffic are a performance bottleneck for parallel computing in Chip Multiprocessors. Cache coherence becomes a significant problem when parallel applications make local replicas of shared data to improve scalability and performance.

The two main classes of coherence protocols are snooping and directory. Snoopy-based protocols used interconnection network to broadcast messages to all cores. A cache controller send a request for a block by broadcasting a request message to all other coherence controllers. The coherence controllers send back data in response to the request. The shared wire become a performance bottleneck as the number of processors increases. In directory-based, no broadcast is necessary. In directory-based, no broadcast is necessary. A directory will maintain the required information for coherence. A cache controller communicates with a common directory whenever the processor's action may cause an inconsistency between its cache and the other caches or memory.

The improvement of cache design becomes more and more complex due to the impact of super-pipelining, super-scaling, prefetching, speculation, etc. Cache Coherence logic is in the critical path of accessing memory and can easily become the main bottleneck, exacerbating the processor and memory speed gap leading to significant performance degradation. Having aggressive prefetching on different cores of a Chip Multiprocessors is very beneficial for memory latency tolerance on many

applications, but it can lead to significant system performance degradation and bandwidth waste. This result of prefetch-demand interference happens when a prefetcher in one core ends up pulling data from a producing core before its been written. The cache block will end up transitioning back and forth between the cores and result in useless prefetch and saturating the memory bandwidth.

2.4 Data Prefetching Techniques

Hardwar data prefetching has earned much attention as means to bridge the performance gap between processor and memory system. Several hardware prefetchers with diverse prefetching strategies have been proposed in the literature. In the following sections, several hardware prefetching strategies will be presented and examined by comparing their relative strengths and weaknesses.

1. *History-based prefetching* is the most commonly used among hardware prefetching strategies. In these strategies, a prefetch engine is used to predict future data references and to issue prefetching instructions. The prefetch engine captures the history access patterns or the history of cache misses to predict future accesses by a processor. Spatial Memory Streaming (SMS) prefetcher [32] is one of the current top-performing, light-weight, history based prefetchers. 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. SMS introduces the notion of a spatial region generation that begins with the first miss to access a block within a region and ends with the eviction or invalidation of any block from that region. 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[32].

2. *Address-correlating prefetching* is a class of prefetchers that exploits the fact

that algorithms tend to traverse data structures in the same way repeatedly, leading to recurring cache miss sequences. Correlation between accesses to pairs of memory locations was suggested as early as 1976 [3]. Cooksey et al. proposed Content Directed Prefetcher (CDP), a type of address-correlating prefetching, [11] to target pointer-intensive applications. CDP examines each address-sized word of the fetched or subsequently prefetched data in order to find likely pointer addresses. Then, it initiates prefetch requests for those data that are identified as potential addresses. Due to this aggressive approach, CDP has the potential to run many instances ahead of the current execution sequence and prefetch data, pointed by likely pointer addresses, into the cache. CDP does not require any state information and also does not require any training. However, it tends to generate a lot of useless prefetches.

3. *Runahead-based prefetching* uses the execution resources of a core that would otherwise be stalled on a long-latency event like off-chip cache miss to pre-execute a set of instructions speculatively, then using the results obtained to issue prefetching. Run-ahead was originally proposed in the context of in-order cores by Dundas and Mudge [13]. Mutlu et al. proposed an implementation to support runahead execution in out-of-order processors [23], 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. Although the prefetcher is effective in the event of second-level cache miss, it suffers from a few drawbacks. First, there is a large overhead in restarting normal execution after restoring the checkpoint, when the miss returns. Also, because of this overhead, the effectiveness of this mechanism to handle shorter latencies like the first-level cache miss latencies

is reduced. Additionally, since the same hardware is used for runahead mode execution, computation cannot be overlapped with a second-level cache miss.

4. *Machine learning based prefetchers*, several studies have employed machine learning technique to enhance microarchitectural components. Peled et al. proposed context-based memory prefetcher[27], which approximates semantic locality using reinforcement learning. The prefetcher identifies access patterns by applying reinforcement learning methods over machine and code attributes that provide hints on memory access semantics. The prefetcher consists of three units 1) collection unit to track context history and, using the current memory address, to create context-address pairs. 2) prediction unit looks up the current context in the context table to generate prefetches. 3) feedback unit closes the reinforcement learning loop by updating the scores of previously encountered contexts, if the current memory access hits any prefetches they issued. Context-based prefetcher makes it possible for naive, pointer-based implementations of irregular algorithms to achieve performance comparable to that of spatially optimized code. Rahman et al. [28] used machine-learning technique to predict the optimal combination of prefetchers for a given application, based on program characterization and utilized hardware performance events in conjunction with a pruning algorithm to obtain a concise and expressive feature set.

5. *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.* [21] 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.* [10]. B-Fetch [17, 26] 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 the future execution path, and the effective addresses of load instructions along that path. By recording the variation of register contents at earlier branch instructions and exploits, B-Fetch 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.

6. *Sequential Pattern Prefetchers* are limited in extent to prefetch only blocks at consecutive addresses. Best-Offset prefetcher select the prefetch offset automatically and dynamically, trying to adapt to the application behavior, which may vary over time, it tests several different offsets to find the best prefetch offset [22]. Without using the program counter or other core registers Signature Path Prefetcher can learn how to preftech complex data access patterns. SPP use compresses history signature, and it can achive a balance between aggressive prefetching and accuracy[19].

3. MULTITHREADING B-FETCH FOR CHIP MULTIPROCESSORS

This chapter reviews B-Fetch[17, 26], a branch directed, lightweight data prefetcher to improve performance in CMPs. Then, it presents Multithreading B-Fetch (MTB-Fetch) . The architecture of the prefetcher and details of the proposed design are also presented.

3.1 B-Fetch for Chip Multiprocessors

Progams construction can be mapped into a control flow graph as shown in figure 3.1. The outcome of the branch determines which basic block will be executed. In case of taken path on the right, it leads to one of the load instructions and the not taken path leads to the other load instructions. The prediction of the branch outcome can be used to determine which load instructions will be excuted in the basic block subsequently following a branch instruction.

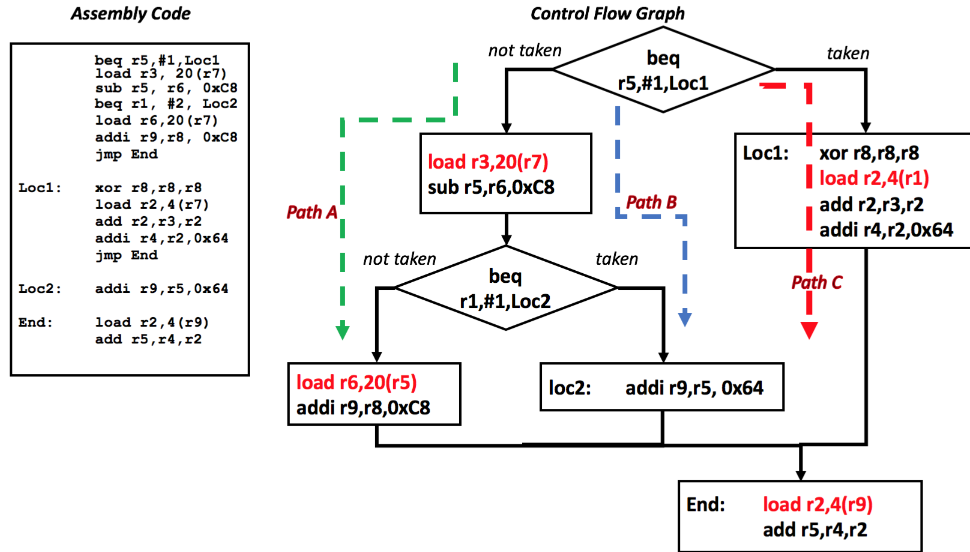

Figure 3.1: Data Access and Control Flow.

In a program future data execution depends on the branch instruction results along the execution path and the per-block register transformations along that path. 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 [17, 26] . 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 [17, 26]. B-Fetch uses two speculative microarchitecture components. Frist, it speculates on the expected excution path through a future basic blocks, this speculation is directed by a lookahead mechanism that depends on branch prediction to predict the future execution path. Second, it speculates on the effective addresses of load instructions along that path. To do so, B-Fetch records the variation of register contents at earlier branch instructions and exploits this knowledge to predict the effective address [17, 26]. Since B-Fetch uses the variation of register values rather than the effective address history, it can issue useful prefetches even for instructions that with irregular control flow and data access patterns.

Figure 3.2 shows the overall system architecture of a B-Fetch in an out-of-order core. It shows the main core execution pipeline and the auxiliary hardware for B-Fetch preftecher. The base line design include an out-of-order pipeline with a 4-wide issue width.

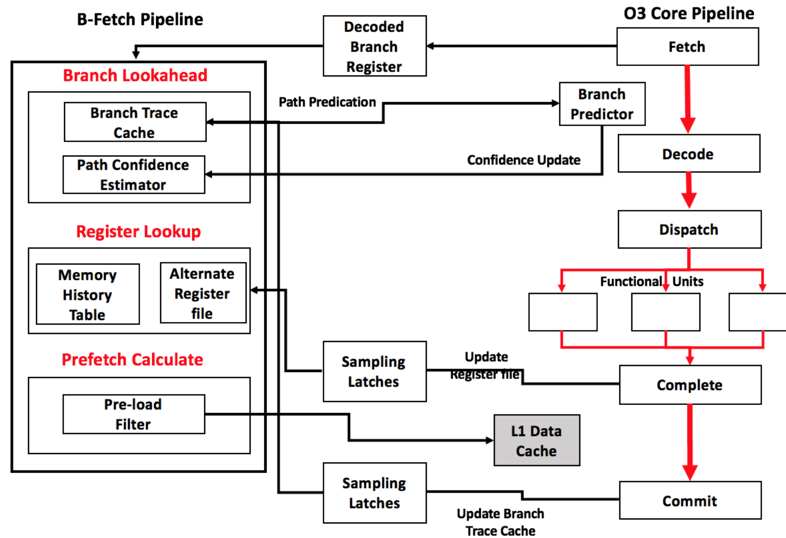

Figure 3.2: B-Fetch microarchitecture

The B-Fetch hardware compose of 3-stage pipline parallel to the core pipeline. The Decoded Branch Register (DBR) connect B-Fetch to the cors's Fetch stage. When branch instuctions are decoded in the the main execution pipeline, the PCs value of the branch instuctions are added to the DBR. For B-Fetch to sart prefetching it needs the branch branch PCs and target addresses, after that B-Fetch engine starts to predict future execution path, memory instructions, and their effective addresses [17].

The following stages form the B-Fetch pipeline:

- *Branch Lookahead Stage* is similar to the fetch stage in the main pipeline branch lookahead functions. The duty of this satage is to generate the speculative exception path from the currently decoded branch. Due to the accuracy of the branch predictor, branch lookahead stage uses confidence path estimator as required along with the lookahead mechanism that stops the lookahead from

going down too deep along a speculative path[17, 26]. This stage includes two main components. First, *Branch Trace Cache* trace the branch instructions in the dynamic instruction stream to create set of pointers in the progam control flow marked by branch instructions to allow jumps between basic blocks and skip the branch instuctions in between. An entry in the branch trace cache include the branch address and the following branch address and two state bits. The branch trace cache is indexed by a hash of the current branch PC, predicted branch direction, and the target address. When the branch trace cache is indexed using the hash, then the next branch passd to branch predictor to predict its target and direction, which then used to index the he trace cache again to check if a valid path forward exists for this branch [17]. By doing so, the branch trace cache help guide lookahead stage forward and the branch predictor and target buffer to help maneuver it in the right direction. To save space, the lower 32 bits of the 43 bit addresses are used. The second component is *Path Confidence Estimator* which used to prevent looking down the wrong execution path, which can lead to useless prefetches and cache pollution in L1 data cache. The second component is *Path Confidence Estimator* used to avoid looking ahead down the wrong excution path, which can lead to cache pollution in the L1 data cache. B-Fetch engine will stop prefeching when the confidence of the path falls below a certain preset threshold, lookahead mechanism stalls to wait for the confidence to improve.

- *Register Lookup Stage* looks up information about the registers form loads in a basic block to to generate effective addresses within a given block. This stage includes two main components. First, *Alternate Register File (ARF)* to maintain a copy of the register file contents for use in generating predicted

prefetch effective addresses. To ensure timly updates to Alternate Register File, a sampling-latch delayed copy of execution stage generated register values is used to perfom the updates. To keep track of the modifying instruction order each register in the Alternate Register File is augmented with an instruction sequence. This design provide accuracy when generating prefetch effective addresses and improve performance significantly [17, 26]. The second component is the *Memory History Table (MHT)* which maintains source register indices, current register values, and offset values to calculate effective addresses for prefetch candidates. Each entry is indexed by the hash of the current branch PC, predicted branch direction, and the target address generated in the Branch Lookahead Stage.

- *Prefetch Calculate Stage* is responsible for generating the prefetch addresses that are issued to the prefetch queue, after filtering by a per-load confidence estimator [17].

$$Offset = [\Delta RegisterValue] + StaticOffset \tag{3.1}$$

$$PrefetchAddress = [RegisterValue] + Offset \tag{3.2}$$

Using equation 3.2 B-Fetch generates the effective prefetching address based on the current value of the linked register (RegVal) added with the Offset value. The Offset is computed as the difference between the effective address and RegVal. when a memory instruction executes in the main pipeline, the MHT is indexed using the prior branch PC and the Offset is updated[17].

3.2 MTB-Fetch for Chip Multiprocessors

The insight behind Multithreading B-Fecth (MTB-Fetch) is to use the decode stage in the actual processor pipeline to trace the synchronization primitives and identify when a thread is spinning on a lock. Figure 3.3 shows the implementation of a lock using ALPHA ISA. 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. 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 primitive instructions. To implement MTB-Fetch on top of B-Fetch framework, we need to add two components : Synchronization Primitives Trace Cache and Invalidation Filter to keep track of all prefetches that were invalidated due to coherence protocol.

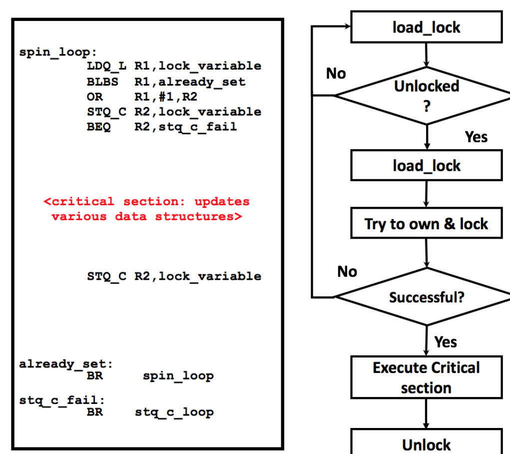

Figure 3.3: ALPHA code for lock implementation.

3.2.1 System Architecture

To implement MTB-Fetch on top of B-Fetch framework, we need to solve two problems. First, identify 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 B-Fetch engine to start prefetching. The second problem is to deal with invalidation due to cache coherence mechanisms. If a prefetch memory block ends up invalidated, the prefetcher needs to keep track of this information and to learn not to prefetch it again. Figure 3.4 shows the MTB-Fetch microarchitecture. We added Synchronization Primitives Trace Cache and Invalidation Filter to the original B-Fetch microarchitecture.

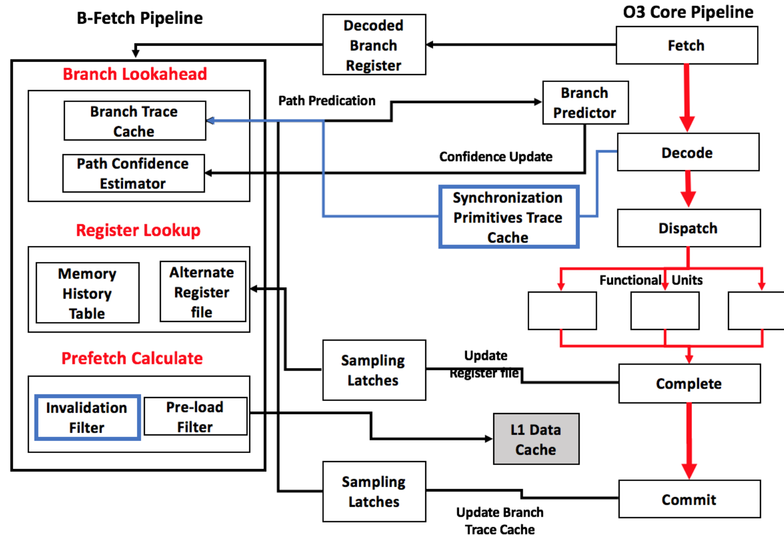

Figure 3.4: MTB-Fetch microarchitecture.

3.2.2 System Components

We shall now go through descriptions of each component used in MTB-Fetch. All the hardware structures are important in realizing an accurate and flexible branch directed prefetching scheme.

- **Synchronization Primitives Trace Cache (SPTC):** The SPTC captures the dynamic atomic primitives that were used to construct the synchronization semantics. Each entry acts as a state machine to indicate where the beginning and the end is of critical section. A *LDQ_L* is followed by a *STQ_C* to the same effective address, indicating the beginning of a critical section . Once a second *STQ_C* is detected, it indicates the end of a critical section. Then the first branch address after the critical section will be passed to branch lookahead in B-Fetch pipeline, so B-Fetch can predict execution path starting from the current branch in order to prefetch data in the next basic block.

Figure 3.5: Single Synchronization Primitives Trace Cache (SPTC) entry.

Figure 3.5 shows an entry in SPTC. Each entry in the SPTC includes 64 bits of the load effective address and 2 state bits.

- **Invalidation Filter** To avoid useless prefetches, wasted bandwidth and energy, it is crucial to reduce the impact of cache coherence mechanisms on prefetches data blocks. Filtering prefetch requests that were invalidated from outside the core by the cache coherence mechanisms is very important for systems that prefetch directly into the L1 cache, and those that run multiple threads on different cores. Basically, the branch confidence mechanism might be thought of as a prefetch filtering mechanism. However in multi-core system, one core may end up pulling data from a producing core before it has been written. To deal with invalidated prefetches, our invalidation filter measures the confidence of prefetches launched from a given load PC.

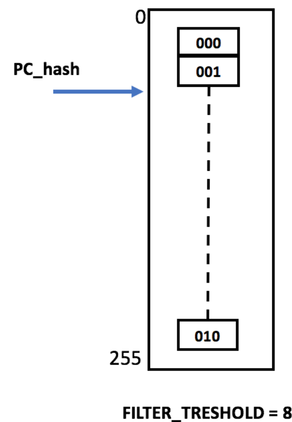

Figure 3.6: Invalidation filter table.

Similar to the pre-load filter, invalidation filter consists of three different tables which contain 3-bit up-down saturating counters for corresponding prefetch loads. Figure 3.6 shows an invalidation table. Each table is indexed, using

the PC of the load instruction by different hash function, and the counter is incremented when the prefetch address was not invalidated by coherence protocol. If the prefetch address was invalidated, the counter is decremented.

3.2.3 Hardware Cost

The additional hardware storage requirements for MB-Fetch, B-Fetch and SMS are summarized in Table 3.1. Two additonal components have been added to B-Fetch. In term of hardware budget Primitives Trace Cache (PTC) costs 2.06KB and the Invalidation Filter costs 2.25KB. To optimize the performance of SMS, we used the configuration used by Somogyi, et al. [32] and 2KB spatial regions, a 64-entry accumulation table, and a 16K-entry pattern history table.

Table 3.1: Hardware storage overhead in KB, adopt from B-Fetch[17].

Prefetcher	Component	# Entries	Size (KB)
MB-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	32	0.156
	Path Confidence Estimator	32	0.156
	Primitives Trace Cache	256	2.06
	Invalidation Filter	2048	2.25
	TOTAL SIZE : 17.15		
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	32	0.156
	Path Confidence Estimator	32	0.156
	TOTAL SIZE : 12.84		
SMS	Active Generation Table	64	0.57
	Pattern History Table	16k	36
	TOTAL SIZE : 36.57		

4. EVALUATION

In this chapter, we present the experimental methodology used to evaluate MTB-Fetch. We compare our results against three prefetchers.

4.1 Methodology

We use gem5 [6], a cycle accurate simulator, to evaluate MTB-Fetch. The baseline configuration is summarized in Table 4.1. We used a set of nine multithreaded programs from PARSEC benchmark suite [4]. The benchmark applications represent widely used parallel workload suits PARSEC, which use Pthreads libraries to handle synchronization. The benchmark applications are cross-compiled for ALPHA ISA with the O3CPU CPU model (Out-of-Order) and the detailed (classic) memory model with caches. We ran the benchmarks in Full System (FS) mode. The baseline hardware is a 4 cores CMP, ALPHA ISA machine with three level cache hierarchy as specified in table 4.1. Each core's private cache is split into Icache (32KB) and Dcache(62KB), 256KB second level cache and 1024KB per core third level shared cache.

Table 4.1: 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

First, we created checkpoints at the Region of Interest (ROI) and then stopped executing. Then we restore simulation at the checkpoints (i.e. the beginning of the ROI) using the out-of-order processor model O3CPU. We present the performance as the speedup compared to the baseline configuration (i.e. $\frac{ExecutionTime\ baseline}{ExecutionTime\ MTB-Fetch}$). The execution time is the time spent in the ROI.

MTB-Fetch results are compared against two light-weight prefetcher designs, the Stride prefetcher and the SMS prefetcher, configured as described in Section 4.1 and the orignal B-Fetch. For Stride prefetcher we used prefetching the next 8 strided addresses [17].

4.2 Results and Analysis

We first present results for MTB-Fetch versus the competing light-weight prefetchers on shared-memory, multithreaded application workloads.

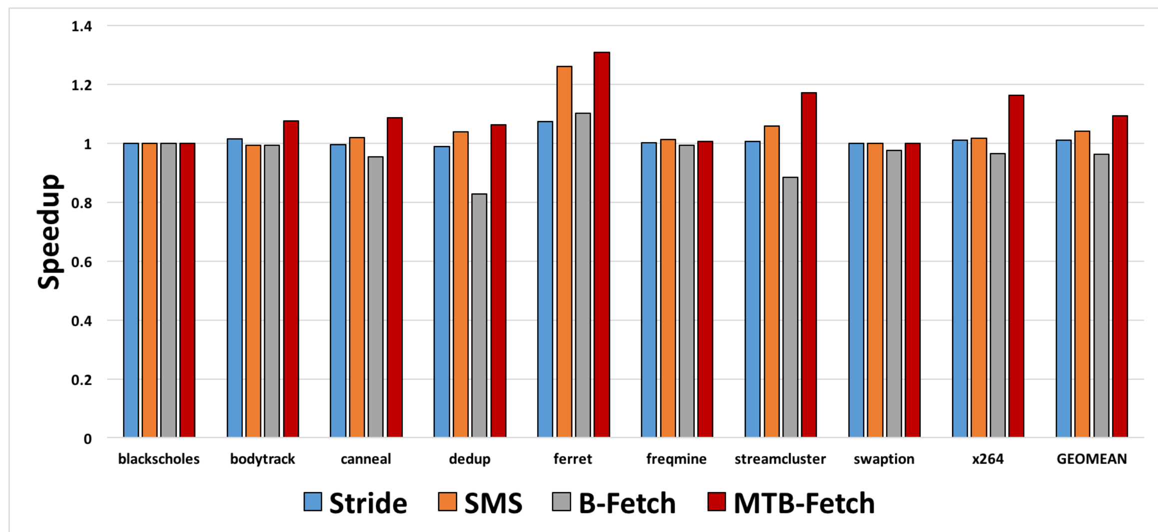

Figure 4.1: Multi-threaded workload speedups.

Figure 4.1 shows the speedup for multi-threaded workloads. We notice that Stride and B-Fetch perform poorly, compared to other prefetchers, across all the benchmarks. Thus, we focus on comparing MTB-Fetch and SMS. Each column refer to the geometric mean across four cores. The Geomean refers to the geometric mean across the entire set of benchmarks. MTB-Fetch achieves a geometric average speedup of 9.3% compared to 4.1% for SMS.

MTB-Fetch outperforms SMS in all but $freqmine$ and achieves the perfomance on $blackscholes$. MTB-Fetch were able to achieve maxmuim speeedup of 30% on $ferret$ and outperformed all prefetchers. Considering the speedup and hardware costs, MTB-Fetch offer the best solution for data prefetching in multi-threaded workloads.

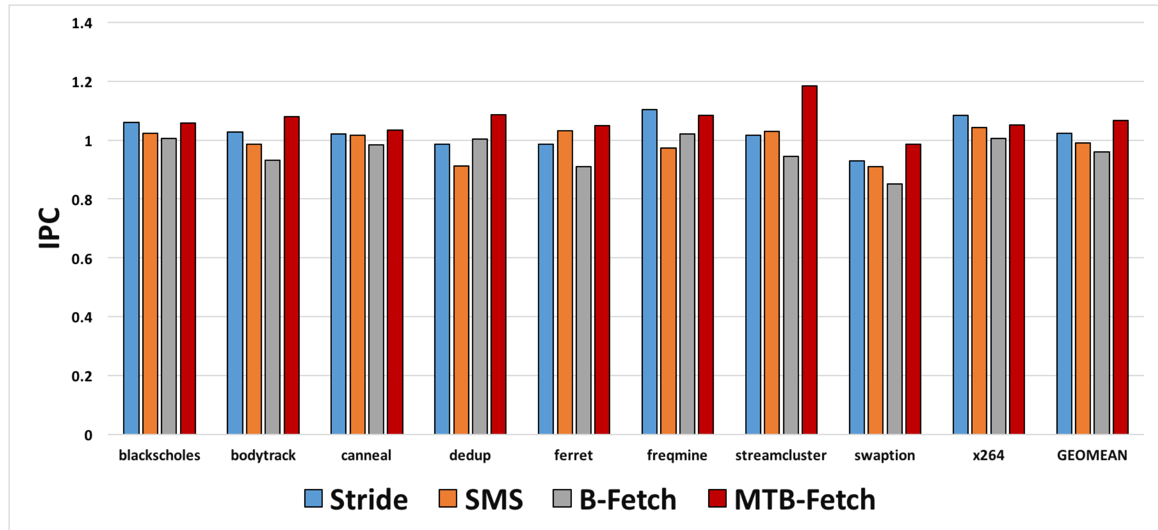

Figure 4.2: Normalized Geomean if IPC.

Figure 4.2 demonstrates the impact of four diffrent preftechers on the system performance (IPC) as compared to a baseline (no-prefetching) system. The figure shows for $blackscholes$ all preftecher achieved normalized IPC graterer than one,

but in fact, there is no improvement in the execution time. Therefore, we believe using the time spent in the ROI is more accurate to find the overall speedup.

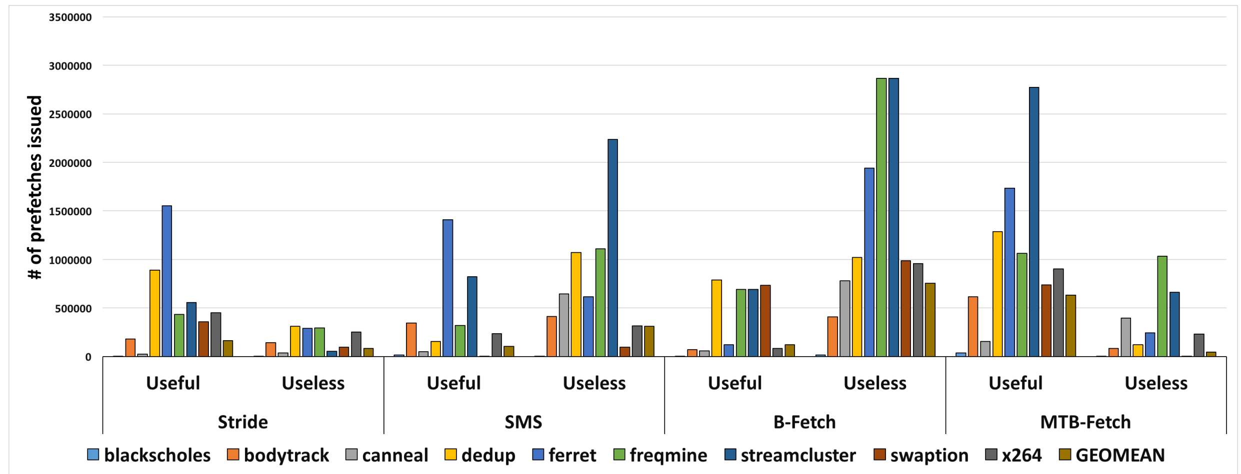

Figure 4.3: Number of useful and useless prefetches issued.

Figure 4.3 shows the number of useful and useless prefetch issues by all preftechers for all the workloads. Useful prefetch are used by demand miss, while useless prefetch are evicted before demand miss. The figure shows that MTB-Fetch has the least useless prefetch due to the use of invalidation filter. The use of invalidation filter reduced the useless prefetches by more than 50% across the entire set of workloads.

5. CONCLUSION AND FUTURE WORK

Thread synchronization overhead is a performance bottleneck for multi-threaded shared-memory applications. This overhead scales with core count and dramatically reducing the overall scalability of the application. Indeed, threads spend long time waiting to acquire the lock of a critical section. In addition, a processor have to stall execution to wait for load data accesses to complete. Furthermore, there are often independent instructions which include load instructions beyond synchronization semantics that could be executed in parallel while a thread waits on the synchronization semantics.

We proposed a prefetcher that leverages the synchronization primitives in the dynamic instructions streams. MTB-Fetch outperforms the best in class lightweight prefetcher. MTB-Fetch achieves a geometric average speedup of 9.3% speedup over baseline with a low hardware impact.

For this study we have considered a 4 core machine and used PARSEC benchmark suite [4]. Future work includes working with 8 and 16 cores to see how much more beneft we can achieve. We also wish to work with other shared-memory applications, such as SPLASH.

REFERENCES

[1] Kernal.org.

[2] Ieee standard for information technology- portable operating system interface (posix) base specifications, issue 7. *IEEE Std 1003.1-2008 (Revision of IEEE Std 1003.1-2004)*, pages c1–3826, Dec 2008.

[3] J. L. Baier and G. R. Sager. Dynamic improvement of locality in virtual memory systems. *IEEE Trans. Softw. Eng.*, 2(1):54–62, January 1976.

[4] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. Technical Report TR-811-08, Princeton University, January 2008.

[5] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[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. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[7] Andre R. Brodtkorb, Christopher Dyken, Trond R. Hagen, Jon M. Hjelmervik, and Olaf O. Storaasli. State-of-the-art in heterogeneous computing. *Sci. Program.*, 18(1):1–33, January 2010.

[8] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[9] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[10] Tien-Fu Chen and Jean-Loup Baer. Effective hardware-based data prefetching for high-performance processors. *Computers, IEEE Transactions on*, 44(5):609–623, May 1995.

[11] Robert Cooksey, Stephan Jourdan, and Dirk Grunwald. A stateless, content-directed data prefetching mechanism. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pages 279–290, New York, NY, USA, 2002. ACM.

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

[13] James Dundas and Trevor Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the 11th International Conference on Supercomputing*, ICS '97, pages 68–75, New York, NY, USA, 1997. ACM.

[14] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.

[15] Ehsan Fatehi and Paul Gratz. Ilp and tlp in shared memory applications: A limit study. In *the 23rd International Conference on Parallel Architectures and Compilation (PACT)*, pages 113–126, 2014.

[16] F. Ryan Johnson, Radu Stoica, Anastasia Ailamaki, and Todd C. Mowry. Decoupling contention management from scheduling. *SIGARCH Comput. Archit. News*, 38(1):117–128, March 2010.

[17] David Kadjo, Jinchun Kim, Prabal Sharma, Reena Panda, Paul Gratz, and Daniel Jimenez. B-fetch: Branch prediction directed prefetching for chip-multiprocessors. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 623–634, Washington, DC, USA, 2014. IEEE Computer Society.

[18] Henry Kasim, Verdi March, Rita Zhang, and Simon See. Survey on parallel programming model. In *Proceedings of the IFIP International Conference on Network and Parallel Computing*, NPC '08, pages 266–275, Berlin, Heidelberg, 2008. Springer-Verlag.

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

[20] Vipin Kumar. *Introduction to Parallel Computing.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[21] Yue Liu and D.R. Kaeli. Branch-directed and stride-based data cache prefetching. In *Computer Design: VLSI in Computers and Processors, 1996. ICCD '96. Proceedings., 1996 IEEE International Conference on*, pages 225–230, Oct 1996.

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

[23] 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*, pages 129–140, Feb 2003.

[24] Camelia Muoz-Caro, Javier Diaz, and Alfonso Nio. A survey of parallel programming models and tools in the multi and many-core era. *IEEE Transactions on Parallel Distributed Systems*, 23:1369–1386, 2011.

[25] Oline. https://computing.llnl.gov/tutorials/pthreads/.

[26] Reena Panda, Paul Gratz, and Daniel Jimenez. B-fetch: Branch prediction directed prefetching for in-order processors. *IEEE Comput. Archit. Lett.*, 11(2):41–44, July 2012.

[27] Leeor Peled, Shie Mannor, Uri Weiser, and Yoav Etsion. Semantic locality and context-based prefetching using reinforcement learning. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 285–297, New York, NY, USA, 2015. ACM.

[28] S. Rahman, M. Burtscher, Ziliang Zong, and A. Qasem. Maximizing hardware prefetch effectiveness with machine learning. In *High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on*, pages 383–389, Aug 2015.

[29] Manjunath Shevgoor, Sahil Koladiya, Rajeev Balasubramonian, Chris Wilkerson, Seth H. Pugsley, and Zeshan Chishti. Efficiently prefetching complex address patterns. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 141–152, New York, NY, USA, 2015. ACM.

[30] A.J. Smith. Sequential program prefetching in memory hierarchies. *Computer*, 11(12):7–21, Dec 1978.

[31] Stephen Somogyi, Thomas F. Wenisch, Anastasia Ailamaki, and Babak Falsafi. Spatio-temporal memory streaming. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 69–80, New York, NY, USA, 2009. ACM.

[32] Stephen Somogyi, Thomas F. Wenisch, Anastassia Ailamaki, Babak Falsafi, and Andreas Moshovos. Spatial memory streaming. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ISCA '06, pages 252–263, Washington, DC, USA, 2006. IEEE Computer Society.

[33] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs Journal*, 30(3):202–210, 2005.

[34] Carole-Jean Wu, Aamer Jaleel, Margaret Martonosi, Simon C. Steely, Jr., and Joel Emer. Pacman: Prefetch-aware cache management for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 442–453, New York, NY, USA, 2011. ACM.

[35] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.