CRITICALITY-AWARE MEMORY SYSTEMS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Saugata Ghose

August 2014

CRITICALITY-AWARE MEMORY SYSTEMS

Saugata Ghose, Ph.D.

Cornell University 2014

Research on computer memory systems has been of increasing importance over the last decade, as they have become a significant bottleneck for application performance. While newer memory systems offer increased memory level parallelism, they cannot be used blindly due to contention for shared resources, making a beneficial and valid sequencing of memory requests requisite in order to exploit these improvements. Traditional approaches to improving this sequencing rely on highly sophisticated memory systems, where significant amounts of inference are often required to make these sophisticated decisions.

Unfortunately, this design philosophy may no longer be sustainable. For example, as memory clock frequencies continue to scale while processor frequencies remain stagnant, sophisticated memory controllers are already being squeezed out, forcing computer architects to revert to simpler designs. We use this as an opportunity to symbiotically involve the processor cores in the decision-making process, simultaneously offloading the complexity from these memory decision makers while extracting richer information on each memory operation.

This work studies the concept of load criticality, where the processors themselves identify the loads which they believe to be most important. Using loads that block at the end of the processor pipeline as an indicator of criticality, we annotate these load block predictions onto memory requests, for use by various components in memory. Our research finds that even using small, sim-

ple predictors for load criticality can offer comparable performance to complex state-of-the-art schedulers for both parallel applications and multiprogrammed workloads on a contemporary multicore system. This same predictor can be used to obtain significant performance improvements and energy savings when using hardware prefetchers.

Ultimately, our criticality-aware design approach achieves the performance of traditionally-complex memory systems, and does so with trivial overheads that are attractive for future commercial adoption.

BIOGRAPHICAL SKETCH

Saugata Ghose made his grand entrance into the world one Father's Day in the corn fields of Ames, IA. One could say that his destiny as a computer architect was sealed from birth: not only is he the son of one, but he also shares a birthday with IBM. There was no fighting fate, and in two years' time, the Ghose family would move to the birthplace of IBM, thus beginning Saugata's somewhat odd love of the Upstate New York tundra.

After working hard and playing harder during high school, Saugata traveled afar to the State University of New York at Binghamton, where he earned dual B.S. degrees in Computer Science and Computer Engineering. During his time there, Saugata won several awards for academic achievement and service. His time as an undergraduate researcher with Dr. Aneesh Aggarwal all but set in stone a graduate career in architecture.

Unwilling to leave his beloved upstate behind, Saugata moved an hour north to Ithaca, to begin what would be a seven-year residency (much to the delight of the landlords at Plaza East Apartments). At the Cornell University School of Electrical and Computer Engineering, under the tutelage of Dr. José Martínez, he happily whittled away his time performing research in the areas of adaptable multicore architectures and memory systems. Between delicious gastronomic endeavors with friends and educating many an undergraduate, Saugata honed his desires for a career in academia. Through all of these (mis)adventures, he earned the ASEE National Defense Science and Engineering Graduate Fellowship, as well as the 2013 ECE Director's PhD Teaching Assistant Award.

His successful dissertation defense in July 2014 brought a 25-year-long career as a student to a glorious end. Saugata now faces the daunting prospects of leaving New York behind and pursuing a career in the real world.

To Dadu, Baba, Mommom, Rimi, and everyone else
who believed in (and put up with) me along the way.

ACKNOWLEDGMENTS

This work would not have been possible had it not been for my advisor, Dr. José Martínez. Throughout my graduate career, he has offered invaluable advice and countless forms of support. He has encouraged me when I was dejected about a project, and given tough love when I needed a push. If I had to do it all again, I'd gladly choose to work with you, José. For you, I promise that I'll stop unnecessarily apologizing as much as I do (sorry that I had to bring it up here).

I owe an enormous debt of gratitude to my educators, past and present, all of whom have been a genuine source of inspiration. I have learned quite a lot from each of the Computer Systems Laboratory (CSL) faculty, both technically and professionally. In particular, I would like to thank my committee members, Dr. David Albonesi and Dr. Rajit Manohar, for their help and mentoring.

Hyodong Lee, Monica Lin, and Rebant Srivastava deserve much praise for their assistance in this work. Hyo is a tireless worker and all around a wonderful person. Monica mixes a drive for perfection with a knack for knowing just when to let the sarcasm and levity peek through. Rebant, with his unmatched enthusiasm, has been a loyal collaborator and friend. I know the three of you will go very far, and I'm honored to have played a small role in your lives.

My fellow CSL compatriots have been fantastic for support in the trenches throughout this arduous process. I am thankful to have had Meyrem and Nevin Kırman, Maia Kelner, and Xiaodong Wang in my research group. From round after round of Kelnerball, to my supposed love of one J. Bieber, the fun I had with Janani Mukundan was perfectly counterbalanced with many a fruitful discussion, and it's been a tough final year without her in the office. Special thanks belong to Raymond Huang, Shreesha Srinath, and Jonathan Tse, with whom I've constantly bounced off ideas and enjoyed countless evenings.

I have been fortunate to have friends such as Conway Niu, Frank Zurla, Houtan Fanisalek, Vanessa Crans, and Danielle Crans share in my escapades throughout my graduate years, as well as many more.

Matthew Manela, my most trusted partner in so many endeavors, has eternally championed me. He has offered many pearls of wisdom over the years, most notably that the mark of a good Italian restaurant is their Fettuccine Alfredo. Whenever Matt and I get lost in some crazy vision, Mallory Emerson has always been the voice of reason, until of course she starts joining us in crazy machinations (only to bring everyone back to reality again). Having the two of them and Harrison in my life brings me unbounded joy, and I can't wait for the next Saugata Goose caper (I, of course, want to retain animation rights).

Robert Karmazin is the first person I go to for my ridiculous endeavors. Late night soda runs, trips to Mt. Pleasant to stargaze, gunning that U-Haul down Game Farm Rd.—Ithaca just wouldn't be the same for me if we weren't such close friends. While I admire his efforts to perform the doctoral exams in alphabetical order, I look forward to the day he can escape to the real world.

My experiences at Cornell would not have been the same without Lillian Hsu. Not only was she crazy enough to remain friends, but she watched over me with her (grand)motherly tendencies. If my stomach could sing, it would dedicate a song to her daily home-cooked feasts, and for the best desserts it will ever have. Her vernacular is one-of-a-kind, and I don't think I could ever truly "flummox about" with anyone else. Lillian, if you ever finally procure that three year Muranda cheddar, I'll elevate you from second class friend status, deal?

I could simply say that Jessica Thompson and I share pretty much every musical interest, the same sense of humor, and a love of Buffalo chicken and the Yankees, and that would be enough for us to be great friends. Of course, it's

been much more than that. She's always there at my beckon and call, and has been the most faithful of friends from our days of off-color math and following police chases, all the while laughing at my dumb jokes. Oh, now's as good time as any to remind you that you don't really need hubcaps to drive, Jess.

This, of course, brings me to Jillian Thompson, the best friend one could ask for. If I could go back in time, the first thing I'd do is smack my sixth grade self for hating her (though she had stabbed me in a nightmare). From a friendship that started as little more than absurd silliness and a desire to reclaim a screen name that was rightfully mine, I've gained my most introspective conversations, and a friend who fights constantly for me. I've never met anyone as passionate or sensitive (I've seen her softer side, so we're even), or one who I can be so at ease with. Much as you try, Jill, you'll never be rid of my harassment.

As they say, you can never choose your family, but boy did I luck out in that lottery. Without the encouragement of my father, Kanad, I doubt I would have such an interest in computers. As much a friend as a dad, I owe much of my sensibilities (and snobbery) to him. Indrani, my mother, showed me how to be compassionate, and dedicated her life to caring for my sister and me. Unfortunately, I've been a lifelong victim of her sense of humor. I remember when my sister Ipsita was born, and how much I had wanted a dog instead. Was I ever wrong—having a little sister is one of the great joys of my life, even when she's not so fond of me (big brother duties require me to be a pain in your posterior). I had loving, caring grandparents, and my grandfather Ananda Mohan instilled in me a vivid curiosity for the sciences—I hope I would've made you all proud. My family taught me the virtues of education and hard work, and above all how to be a good person. Their endless belief in me is the reason I have achieved so many of my goals, and for that, nothing can say thank you enough.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

AHB	Adaptive History-Based Scheduler
ALU	Arithmetic Logic Unit
CAS	Column Access Strobe
CBP	Commit Block Predictor
CLEAR	Checkpointed Early Load Retirement
CLPT	Critical Load Prediction Table
CMAC	Cerebellar Model Articulation Controller
CMP	Chip Multiprocessor
CPU	Central Processing Unit (Processor)
DAG	Directed Acyclic Graph
DDR	Double Data Rate
DIMM	Dual In-Line Memory Module
DRAM	Dynamic Random Access Memory
FCFS	First-Come-First-Serve
FDP	Feedback Directed Prefetching
FR-FCFS	First-Ready First-Come-First-Serve
GPU	Graphics Processing Unit
ILP	Instruction Level Parallelism
ISA	Instruction Set Architecture
L1	Level 1 Cache
L2	Level 2 Cache
LDQ	Load Queue
LIMCOS	Loads Incurring Majority of Commit Stalls
LLC	Last Level Cache
LSQ	Load-Store Queue
MLP	Memory-Level Parallelism
MORSE	Multi-Objective Reconfigurable Self-Optimizing Scheduler
MSHR	Miss Status Holding Register
MT/s	Megatransfers per Second
PAR-BS	Parallelism-Aware Batch Scheduler
PC	Program Counter
RAS	Row Access Strobe
RL	Reinforcement Learning
ROB	Reorder Buffer
SIMD	Single Instruction, Multiple Data
SRAM	Static Random Access Memory
SSD	Solid State Drive
SSE	Streaming SIMD Extensions
TCM	Thread Cluster Memory Scheduler

CHAPTER 1

INTRODUCTION

1.1 The Limitations of Autonomous Memory Controllers

Modern computer processors have today found their performance constrained by the performance of the memory system, a roadblock commonly known as the *memory wall* [80]. Memory operation latencies have not scaled as quickly as processor performance, and despite improvements to the memory architecture to deliver greater memory-level parallelism (MLP), a miss to the dynamic random access memory (DRAM) devices that make up main memory can be very costly, taking hundreds of processor clock cycles. Even misses to a last level cache (LLC) can incur a penalty of dozens of cycles. To this end, much contemporary research in computer architecture has focused on improving various aspects of the memory subsystem.

In the memory hierarchy, decisions about how to handle memory requests have traditionally been made independent of the processor cores that issue these requests. While this eases system design complexity by increasing modularity, it forces designers to rely on potentially large amounts of inference inside the memory when making these decisions. These decision-making mechanisms, typically unaware of the state of the applications and processors, are often unable to make choices that best address the direct needs of the programs. Instead, we have relied on memory-centric objectives that can be directly measured within memory, such as improving the throughput of individual threads or assessing the accuracy of in-memory predictors.

An excellent example of this can be found in the memory controller. The controller receives requests for main memory from the processor, and must *schedule* (decide the most optimal order of) these requests for servicing by DRAM. Current research in memory scheduling attempts to perform "smarter" request interleavings to better take advantage of the MLP offered by DRAM [11, 23, 28, 36, 37, 38, 52, 53, 55, 65]. In a push to better identify these smarter interleavings, a number of sophisticated memory schedulers have arisen [28, 38, 52]. Due to the design abstraction of the interface between the processor and its memory system, only the address of the requested memory location is typically sent down through the memory system. This requires these more sophisticated memory schedulers to perform a significant amount of inference on the properties of a request within the memory scheduler itself, as the scheduler does not have access to these properties directly.

Typically, complex schedulers must factor in device timing constraints, the hierarchical design of DRAM, and contention for shared structures (such as the row buffer and memory channel buses) [29], reordering the requests to achieve the best expected memory throughput (or other such memory-centric metrics). This reordering can be problematic, though, as it ignores the needs of a particular application. Within the processor, we find that certain loads are more *critical* (important) to improving program performance than others. Unfortunately, as current schedulers are unable to identify this importance, they are unable to take advantage of this information. Moreover, while memory-centric metrics were previously thought to serve as an adequate proxy for program performance, we have found this to often not be the case, compounding the lost potential in current schedulers resulting from the inability to prioritize critical loads. We find this principle to be true in several locations within the memory.

Concurrently, a second issue with current memory designs has arisen. For a long time, DRAM design exploited process improvements to increase density, but raw performance did not scale as aggressively as it did for processors (the main premise behind the memory wall). Partly as a result, the ratio of the CPU clock cycle (which also regulates the on-chip memory controllers) to the DRAM clock cycle was typically large. This allowed researchers to propose increasingly more sophisticated memory scheduling mechanisms that still met the DRAM cycle time. Designs such as the reinforcement-learning-based memory scheduler [28, 52] require multiple CPU cycles to make a single scheduling decision.

Based on recent DRAM trends, however, we conjecture that this approach to memory scheduler design is not sustainable. Memory bus frequencies for double data rate (DDR) DRAM have grown steadily over the last decade, doubling every 3-4 years. This growth will continue with DDR4, where devices are already expected to reach a data transfer rate of 3200 MT (megatransfers) per second [31]. With CPU frequencies remaining more or less stagnant, their on-chip memory controllers will have fewer available cycles to make a scheduling decision. This means that we will no longer be able to conduct such high degrees of inference within the memory controller. As the available time continues to decrease, memory controllers will be forced to revert to simpler scheduler designs.

1.2 A Symbiotic Solution to Both Problems

Conventional wisdom leads us to assume that these simpler designs will result in poorer scheduling algorithms. We believe this is a flawed conclusion, based in part on an archaic scheduler design approach—scheduling decisions are taken wholly by the memory controller, yet virtually the only information supplied with each memory request is its effective address. In this scenario, the only way to attain high-performance scheduling from such limited observation of a program's execution is to resort to heavyweight controller designs that are highly reliant on inference, such as those discussed in Section 1.1.

In our work, we turn to assistance from the processor to *symbiotically* make decisions within the memory system, solving both of these problems simultaneously. The processor has significantly greater time available for processing and predicting information about a dynamic load instance, beginning from the decode stage. The processor can make a more sophisticated prediction about the *criticality* (importance) of a load, while also having access to a much greater source of data about the load instance, the application, and the current state of the processor. By the time a load is issued, the CPU sends this predigested information along with the request address to the memory hierarchy, where the different hierarchy components can use this information in the best possible way (which can differ by component).

This idea of sending an annotation about urgency is akin to the way a freight shipping company (such as FedEx or UPS) works. Without any such information, the company will try to maximize its efficiency by shipping as many packages as possible, such as by filling cargo planes in the order that packages are

received and dispatching the planes once they fill up. This maximized throughput target is similar to how current memory schedulers try to maximize bandwidth utilization. In reality, the shipping company is actually notified of the urgency of each package by the sender, who dictates that certain packages must arrive in one day, while others have no urgency and can be shipped using a slower method. As a result, the shipping company will sacrifice its maximum potential throughput in order to guarantee that these priority packages arrive at their destinations on time, even if it comes at the cost of delaying those packages which were already declared to not be urgent. Likewise, our processor, playing the role of the sender, achieves the same thing by informing the memory hierarchy of the loads that are most urgent (and even ranking *how* important they are, just as parcel senders mark how soon a package should arrive).

Our research explores two complementary aspects of load criticality. We must first identify *what* makes a load critical. This involves not only understanding the characteristics of loads and programs within the CPU, but also designing hardware that can take this information and properly mark these memory requests. Second, once we have this information, we must decide *how* to best use this information throughout the memory hierarchy. As we see, there are several locations where we can use this information in a variety of ways. At the end of the day, our goal is to *prioritize those loads which matter most to the currently-executing application, thereby increasing overall system performance through a targeted optimization approach*.

1.3 Key Contributions

This work makes the following contributions to the field of memory systems:[1]

- We are the first to recognize this issue of memory scheduling logic being "squeezed out" by continued increases in DDR DRAM bus frequencies;
- We break down the barrier between the processor and the memory controller, allowing the system to perform symbiotic decision making through the use of limited communication between these devices;
- We identify and analyze the usefulness of predicting loads that stall at the head of the processor reorder buffer for our criticality metric;
- We propose a simple mechanism that can use load criticality predictions from within the processor to improve memory scheduling decisions;
- We perform a quantitative comparison against several ad-hoc criteria previously proposed in literature for indicating load criticality;
- We identify memory access behavior within parallel applications that can be better exploited by hardware prefetchers; and
- We use a simple criticality-based filter to target the utilization of aggressive prefetchers, delivering both improved performance with lower DRAM energy consumption.

[1] Portions of this work (including but not limited to Chapters 3 and 4) were previously published in the Proceedings of the 40th Annual International Symposium on Computer Architecture [21].

CHAPTER 2

BACKGROUND

2.1 Contemporary Memory Systems

Computer processors rely on a memory system in order to provide it with both volatile and non-volatile storage of information (which can consist both of instructions for programs that need to be executed, and data that is used and manipulated by these programs). A vast number of modern processors implement a *modified Harvard architecture*, which possesses the following properties [25, 26]:

- At the processor interface, independent buses for data and instruction requests;
- Independent highest-level caches (closest to the processor) for data and instruction memory;
- A shared memory address space amongst instructions and data; and
- Lower level caches, DRAM, and non-volatile storage that do not partition data and instructions into discrete locations.

Most modern processors contain several processor cores on a single silicon die, and enable parallel programming by implementing a shared memory system, where all of these cores have access to the same address space in order to communicate with each other through memory. The most common implementations use a shared last level cache (LLC). Higher levels of cache are private to each processor core, allowing the system to reduce the effects of inter-thread cache interference (which is especially beneficial when executing independent

multiprogrammed workloads). To facilitate the shared memory space, these higher private caches use *cache coherence* protocols to coordinate the consistency of this data across all of the cores (e.g., managing duplicate copies of the same data, handling updates written by one of the cores). These caches are implemented using bistable static RAM (SRAM) transistor cells in order to provide faster access to stored memory [29]. Lower-level caches have a much greater capacity than higher levels, but this comes at the cost of a longer cache access latency. Multi-level caches in contemporary processors are hierarchically organized to provide low average-case latencies while minimizing the penalty incurred by a highest-level cache miss, through exploiting both *spatial* and *temporal* locality in the memory access patterns of programs.

Beyond the LLC, a series of off-chip DRAM devices (traditionally referred to as *main memory*) maintain large numbers of active virtual pages in a volatile space [29]. A DRAM device contains a capacitive element for every bit of memory stored. While the use of a capacitive element greatly increases the storage density in comparison to SRAM, the memory access latency grows significantly, and the capacitive elements cannot be directly modified. The most common modern instance of DRAM, double data rate (DDR) DRAM, is discussed further in Section 2.1.1.

Logically under the DRAM sits a much larger non-volatile storage element known as the *hard drive*. The hard drive stores all of the memory pages, both those actively mapped within DRAM and those that are not currently in use. Previously, hard drives were exclusively made of magnetic disks, but today, solid state drives (SSDs) have become common as well. Currently, both forms of hard drives are significantly slower than DRAM (trading off density and non-

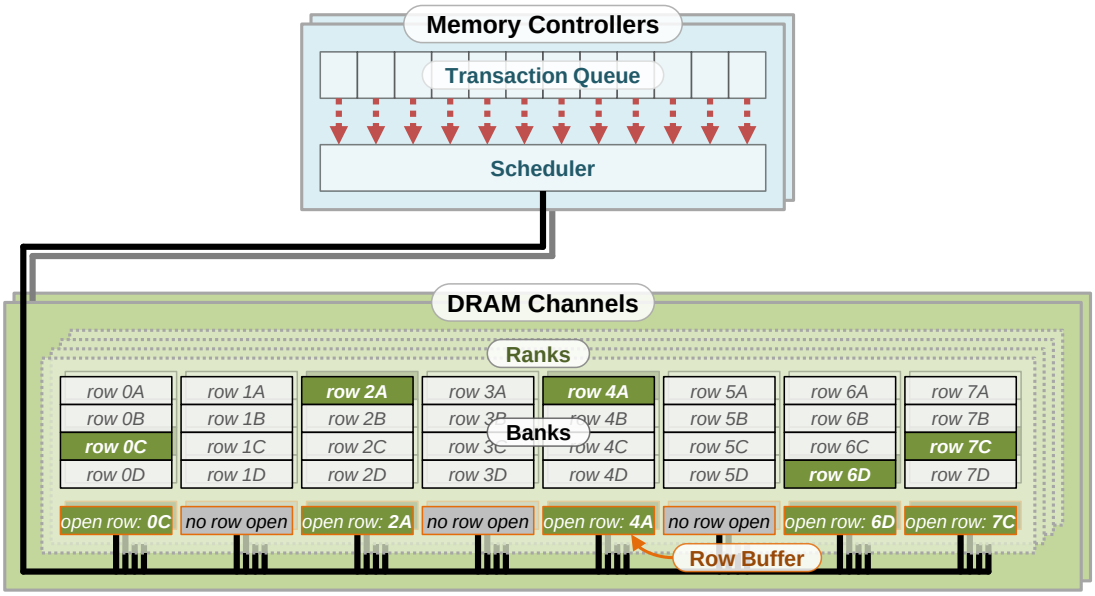

Figure 2.1: Logical organization of DDR DRAM.

volatility for performance), though this may change with the introduction of newer non-volatile technologies within the next decade [8, 41].

2.1.1 Double Data Rate DRAM

Modern DRAM devices, such as used in double data rate (DDR) DRAM, contain a large number of capacitive elements. These devices are organized into a hierarchy of smaller structures that allow the system to exploit greater memory level parallelism (MLP), in order to mitigate the shortcomings of long-latency DRAM operations. Figure 2.1 illustrates how this hierarchy is logically organized for DDR DRAM memory systems [29].

At the highest level, the DRAM devices are divided into memory channels. These channels behave independent of one another, and each has a dedicated

bus that connects them to the processor die. Each of these channels and its corresponding bus are managed by its own memory controller, whose job is to sequence pending memory requests and maintain the correct state of active DRAM pages. These operations are known as *memory scheduling*, and are one of the major memory system activities targeted by our work; as such, we will discuss scheduling in more depth in Section 2.1.2. Over the last decade, the memory controller has largely been integrated on die with the processor cores, and operates at the same clock frequency as the processor cores themselves. Physically, a DRAM channel can consist of one or more dual in-line memory modules (DIMMs), each of which contains several DRAM chips [29, 47, 49].

Within each channel exists a number of physical and logical subdivisions of these capacitive elements, as shown in Figure 2.1. At the finest granularity, each memory channel can return a 64-byte line of data. Multiple such lines of data combine to form a *row*, which is typically sized to match the size of a memory page within the system. While individual lines can be modified and retrieved, the majority of operations within DRAM take place at the row level. As aforementioned, these capacitive elements cannot be modified directly, in order to reduce the amount of charge required for storage. As a result, when the controller issues a command to a particular memory row, that row must be *activated*. Activation copies the values stored within the capacitive element into a series of SRAM cells known as a *row buffer*, from which all read and write operations can be performed. When the controller is finished with the row, the contents of the row buffer are written back to the corresponding row in memory, in an operation known as *precharging*. Physically, each line within a row is actually partitioned across multiple DRAM chips in order to reduce row access latencies [47, 49].

As only one row can be activated within a row buffer at a time, these rows are clustered into *banks*. Each bank contains a series of rows, and has its own row buffer. Within a channel, the banks can mostly operate independently, allowing for the concurrent processing of memory operations (known as *bank level parallelism*). The memory controller must, however, coordinate activities to some degree across multiple banks in order to manage shared resources, such as a single command bus and a single data bus.

In order to increase DIMM densities, the concept of a *rank* was introduced. Ranks are collections of DRAM chips that operate in lockstep. For example, when a row is requested in Bank 2, that row is opened in Bank 2 for each of the ranks. *Address interleaving* is used to ensure that no more than a single copy of a memory location exists within the entirety of DRAM (and to exploit spatial locality in the memory request stream). This interleaving is often chosen to maximize the probability that the rows open for that same bank in other ranks will also contain lines to be requested soon [29].

The values stored within DRAM must be periodically *refreshed*, as the capacitive elements leak charge and, if unchecked, will eventually lose the value stored within them. A refresh operation is typically as simple as activating and then precharging a row [29]. However, since activation and precharge require the use of the row buffer, a currently-refreshing bank is unable to service any DRAM requests. The memory controller typically handles refresh operations, and maintains countdown timers to ensure that these operations take place before any data loss is expected to occur. In order to minimize the impact of refreshing, such operations are staggered across banks, though further mitigation of refresh overheads is an active area of research [6, 7, 45, 51].

2.1.2 The Memory Controller

On die, the main job of the memory controller is to collect pending requests for its managed memory channel and determine how and when to service these requests [29]. The controller maintains a *transaction queue*, as seen in Figure 2.1. This queue maintains all of the requests received by the controller for that particular channel which are yet to be serviced.[1] A *memory scheduler* examines the state of the DRAM (which is tracked and saved within the memory controller) and the pending requests in the transaction queue, and determines which of these requests should be issued next to the DRAM device. Based on the state of the DRAM bank being accessed, a series of commands will be sent off-chip to activate the row if need be, perform reads and writes, and to precharge the row at some point back into the capacitive storage elements.

As mentioned in Section 2.1.1, all of the various components within a memory channel share a single bus, and must therefore coordinate their timings such that requests do not overlap. Unlike other buses, where queues are used to hold requests which lose arbitration, the data bus from DRAM to the die does not have any outbound queuing, hence the need to ensure the lack of request overlap. The controller must also typically keep track of which rows in each bank are activated in the row buffer, in order to determine if precharge and activate commands are necessary for the next requests to those banks. While managing this hardware contention, the controller needs to track how long each of the operations take, as a bank currently performing an operation will be unable to execute another operation concurrently.

[1] In many controller implementations, requests that are in progress within DRAM are expanded and stored within a command queue. For the purposes of our work, this design distinction is irrelevant.

In addition, due to the design of the DRAM devices, a number of timing constraints arise, all of which must be obeyed by the memory controller [29]. For example, when switching between one rank and another, the memory controller must allot a certain delay to ensure that the DRAM circuitry properly completes the switch. Another example of a timing constraint is the four activation window, which states that in a given sliding window of time, no more than four activate commands can be issued to a bank, in order to keep the DRAM power network stable. The controller must also keep track of which rows need to be refreshed, maintaining a countdown timer to the next refresh period. All of these timing constraints are statically defined as part of the device specification [47, 49].

As it is, managing these timings alone can lead to significant complexity in the controller design. The transaction queue is necessary since a number of requests will find themselves delayed due to these timings and to contention. In its simplest form, a memory scheduler will go through the queue in the order that the requests were received in (known as *first-come-first-serve*, or FCFS) [65]. The oldest request to an idle bank (for our purposes, a bank that is not in the process of servicing a prior request, and is not performing a refresh operation) will be scheduled.

Rixner et al. performed some of the first studies into modifying the order of operations within the memory controller [65]. Their work found that if the prioritization of these requests was altered (while of course still maintaining correct behavior with regards to the DRAM specification), significant performance benefits were attainable. In particular, two scheduling approaches were found to be beneficial. The first was an *open row policy* for controllers, where

upon completion of servicing a request, a row would not be precharged immediately, and would instead be left open in case other requests would hit in that open row (in effect exploiting spatial locality). The second change takes advantage of this open row behavior, and opportunistically prioritizes requests that are to the row currently open in the row buffer, ahead of older requests that are to closed rows. This form of scheduling, known as *first-ready first-come-first-serve* (FR-FCFS), improves on average memory access times by reducing the total number of activate and precharge commands that need to be issued.

Much research since then has studied improving further upon the FR-FCFS scheduler prioritization, resulting in more sophisticated scheduling mechanisms [11, 23, 28, 36, 37, 38, 52, 53, 55, 65]. Unfortunately, in present memory controllers, there is very little information gathered about each pending request. In fact, the only data that the controller receives from the processor cores is the address of the requested data location. This can often lead memory schedulers to perform significant inference on each of these requests in order to extract unobservable information, which can greatly increase the complexity of scheduler design. In addition, schedulers that adapt at runtime often require some metric to gauge their performance and detect if further adaptation is required. Again due to the lack of visibility to the controller, performance metrics from the processor perspective cannot be measured, and thus schedulers turn to measuring memory-centric metrics (such as bandwidth utilization or average memory access time), with the expectation that these serve as adequate proxies for overall system performance [23, 28, 36, 37, 38, 53, 55].

Chapter 4 will discuss our approach to memory scheduling, and how the use of criticality information directly annotated by the processor cores will avoid

some of the pitfalls of present memory scheduler designs. Prior research in memory scheduling is discussed in Section 7.3.

2.1.3 Prefetching

Due largely to the loop-recurrent nature of computer programs, memory access requests often exhibit some form of regularity (a *strided access pattern*) in their sequence of addresses. The order of instructions fetched from memory also exhibits regularity, as instructions are often executed in order until control flow divergence occurs. These patterns can be exploited by predicting ahead of time which data and instructions will be requested by the processor in the future, assuming that this regularity holds during subsequent execution. These mechanisms *prefetch* certain cache lines from lower levels of memory at some point before the instruction needing the data or the processor fetch mechanism issues the request. If this prefetch prediction is correct and performed early enough, the performance penalty that would normally occur due to a cache miss can often be avoided completely. However, if the prediction is incorrect, potentially useless lines will have been retrieved into the cache, causing harmful *pollution* by unnecessarily evicting other lines from the cache [70].

There is a large body of work in prefetching, and mechanisms have been proposed both in hardware and in software. Software prefetch support provides additional instructions in the instruction set architecture (ISA) that can be used to retrieve data from lower levels of memory into higher level caches. Often, software developers must explicitly insert these instructions manually into their programs, requiring them to anticipate the expected memory latency, the timeliness of a request (i.e., if it is too early or too late), and program con-

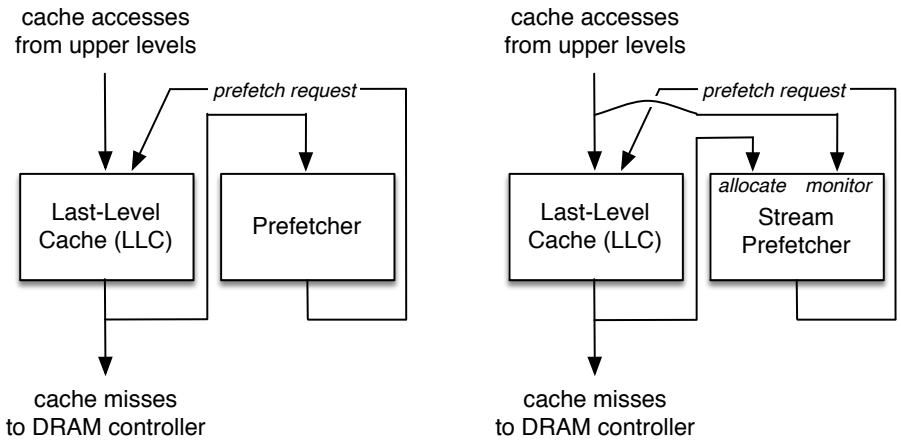

Figure 2.2: Traditional approaches for attaching prefetchers to the cache: (a) Prefetcher reads sequence of cache misses; (b) Stream prefetcher uses sequence of cache misses to allocate entries, and sequence of all accesses to monitor and prefetch (*split* attachment).

trol flow. However, as the program will explicitly be sending this information, the accuracy of these requests is often much higher than their hardware counterparts. Depending on the ISA, instructions are sometimes provided to let developers explicitly dictate how far up the cache hierarchy a prefetch request can be placed into [25]. While this does give greater control to the developer, it does undo some of the benefits of abstraction, forcing programmers to have some knowledge of their underlying hardware. Another drawback is that these ISA extensions can often be processor-specific, reducing program cross-compatibility. Some compilers also provide compile-time support to automatically insert prefetch instructions, though this can reduce the accuracy of the prefetch requests due to limitations in program analysis [50].

In hardware, circuits are built to intercept the stream of memory requests, and to extract potential patterns. These patterns are then extrapolated into ex-

pected future sequences. Naturally, different types of hardware prefetchers are differentiated by their precise mechanisms used to extract and extrapolate these patterns. Typically, these prefetchers, as designed, attach to a cache as shown in Figure 2.2(a). The prefetchers observe all misses being dispatched from a cache to lower memory levels, under the hypothesis that learning the stream of misses will allow the prefetcher to predict subsequent misses. When the prefetcher anticipates these misses, it will issue prefetch requests to these lower levels, using the same mechanisms that a non-predicted (or *demand*) request would use. In order to avoid redundant requests, a hardware prefetcher will first look up these predicted misses in the cache that it is attached to. If the predicted miss hits in the cache, the data is already present, and there is no need for the prefetch request to unnecessarily consume bandwidth. Such a mechanism also allows the prefetcher to take advantage of the miss status holding registers (MSHRs) within the cache, to coalesce multiple requests for the same cache line (to further reduce unnecessary bandwidth utilization).

Today, most modern processors implement some form of software and hardware prefetching. For example, the IBM POWER7 processor contains a prefetcher that is controlled via register, and can perform both next line prefetching and strided prefetching of varying aggressiveness [33]. Prior versions of the IBM POWER processor family have also contained stream prefetchers [74]. The POWER ISA also provides support for prefetch instructions [27]. In its latest IA-64 architectures, Intel provides both next line prefetchers and stream prefetcher along multiple points within the cache hierarchy [25]. The Intel SSE instruction set for x86 architectures contains instructions that can explicitly request cache lines and select into which cache levels these lines are placed [25].

While there are several proposed hardware prefetch mechanisms, we focus our attention now on a small subset of them, as they form the basis for most of the more sophisticated mechanisms: The sequential prefetcher, the stride prefetcher, and the stream prefetcher (which is considered to be state-of-the-art). Other prefetcher schemes, including derivative hardware mechanisms, are discussed in Section 7.4.

Sequential Prefetcher

The sequential prefetcher is the simplest of these three prefetch designs. In its most basic form, the *next line prefetcher*, for every missed line, requests the subsequent line in address order. In a more aggressive setup, the sequential prefetcher can fetch the next N lines. The simplicity of these prefetchers makes them quite attractive, as they provide significant speedups with very minimal logic: Unlike more sophisticated prefetchers, there is no training period, and their design easily exploits spatial locality. However, this can result in unnecessary cache pollution without much net benefit, especially for programs with non-unit access strides. While increasing N can improve the chances of finding a useful line, this aggressive behavior comes with a high performance cost. Przybylski showed that, for single-threaded workloads, when N is greater than one, the negative impact of pollution outweighed the performance benefits of useful prefetching [61]. Another pitfall, the result of having no training, is that sequential prefetchers generally do not contain logic to distinguish the prefetch direction, which limits their potential applicability.

Stride Prefetcher

Stride prefetchers try to overcome these problems by attempting to learn the distance between miss addresses [19]. When a request misses in the cache, an entry in the stride prefetcher is allocated. A subsequent miss generated by the same static instruction will be sent to that entry, which will try to determine the direction and difference in addresses. Further misses that confirm the success of the prediction will lead to more prefetches based on the determined stride, while an incorrect prediction forces the prefetcher to stop issuing requests until it can again learn the access pattern.

Stride prefetchers are quite successful for highly regular memory access patterns, especially if the data is accessed by the same instruction being executed in a high-iteration loop. Unfortunately, stride prefetchers often prove to be too dependent on this regularity, and are too rigid to identify complex memory behavior. Furthermore, each request needs to send the program counter of the instruction to the prefetcher, resulting in extra communication overhead from the processor and additional work/storage in the core. For a stride prefetcher, the number of available entries is a tunable parameter.

Stream Prefetcher

Stream prefetchers attempt to improve on stride prefetchers by relaxing some of the requirements [35, 58, 70, 74]. Instead of observing requests by PC, the stream prefetcher maintains *monitoring windows*. If a miss is observed that does not fall into any existing monitoring windows, a new entry (see Figure 2.3) is allocated, with a *start pointer* initialized to the request's effective address. For this new

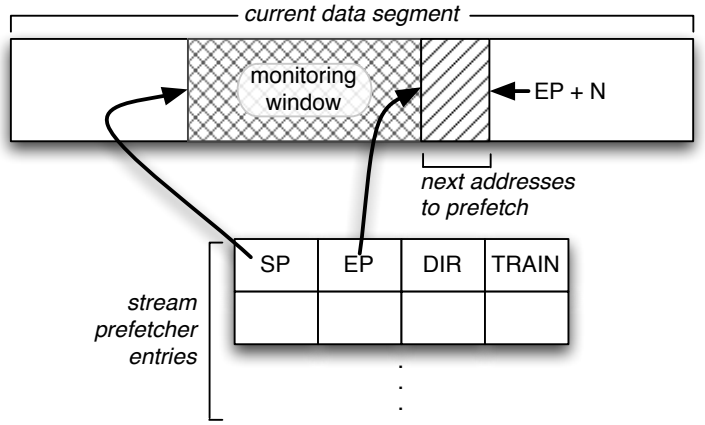

Figure 2.3: Stream prefetcher table entry. SP is the *start pointer*, referring to the beginning of the monitoring window. EP is the *end pointer*, and points to the next address to prefetch. The *prefetch degree*, shown as N, controls the number of blocks to prefetch each time. DIR is the direction of the stream, and TRAIN stores whether the entry is ready to prefetch.

entry, the prefetcher waits for the next two cache misses that fall within 16 blocks of the start pointer, and then uses them to train the direction of prefetching. After training, the *end pointer* keeps track of the last address that was requested. When a subsequent memory request's address falls between the current start and end pointers (the monitoring window), then the prefetcher will retrieve the next N blocks, where N is the *prefetch degree*, and advance the end pointer.[2] The size of the monitoring window can never grow beyond the *prefetch distance*, and so eventually the start pointer is advanced along with the end pointer.

As with the stride prefetcher, we can also tune the number of entries. The stream prefetcher, while still monitoring request patterns, is less reliant on regularity, as any request falling within the monitoring window allows the prefetcher to advance, thus allowing for flexibility not only in the size of the stride,

[2]As proposed for the POWER4, only useful prefetches that fell within the monitoring window were tracked [74]. However, we did not find any appreciable speedups from this approach. Without a very large prefetch degree, their mechanism greatly reduces the opportunities available for successful stream detection, as non-prefetched requests often contribute to the sequence.

but also in the order of accesses. As the entries are no longer tracked by PC, this extra information no longer needs to be transmitted to the caches, and requests to the same memory region by different static instructions can now be combined into a single request stream. However, the stream prefetcher requires a mechanism to detect that a prefetch has been used, and must send this information to the prefetcher separate from the addresses missed in the cache. The stream prefetcher also has a long training period, and therefore will not be useful for short streams of data.

Unlike other prefetchers, which are attached to the cache as shown in Figure 2.2(a), stream prefetchers perform significantly better when attached as shown in Figure 2.2(b) [70, 74]. By performing monitoring on all of the requests coming into a cache, and not just misses, the stream prefetcher is able to more quickly confirm that it has identified a correct stream of requests, since the cache would otherwise filter out the complete pattern if these requests were cache hits. However, the cache is still used to filter out which requests stream entries are allocated for, greatly reducing the pressure due to having a limited number of entries available. Throughout this work, all unmodified stream prefetchers are set up to train on all of the requests coming into a cache.

2.1.4 Challenges

In the memory hierarchy, decisions about memory requests have traditionally been decoupled from the processor cores that issue the requests. While this eases system design complexity by increasing modularity, it forces designers to rely on potentially large amounts of inference inside the memory when making these decisions. These decision-making mechanisms, typically unaware of

the state of the applications and processors, are often unable to make choices that best address the direct needs of the programs. Instead, we have relied on memory-centric objectives that improve the throughput of the subsystem or of threads. While this was previously thought of as an adequate proxy for program performance, we have found that this is in fact not the case, and such memory-centric decisions fail to properly address the urgency of individual loads within the processor.

An excellent example of this can be found in the memory controller. The controller receives requests for memory from the processor, and must decide what the optimal order of scheduling these requests is. As discussed in Section 2.1.2, complex schedulers typically factor in device timing constraints, the hierarchical design of DRAM, and contention for shared structures (such as the row buffer and the memory channel buses) when reordering the requests to achieve the best expected memory throughput. This reordering can be problematic, though, as it ignores the needs of the processor itself. Within each application, we find that certain loads are more *critical* (important) to improving program performance than others. Unfortunately, as current schedulers are unable to identify this importance, they are unable to take advantage of this information. We find this to be true in a number of locations within memory.

Concurrently, a second issue with current memory design has arisen. For a long time, DRAM design exploited process improvements to increase density, but performance did not scale as aggressively. Partly as a result, the ratio of CPU (including on-chip memory controllers) to DRAM clock cycle was typically large. This allowed researchers to propose increasingly more sophisticated memory scheduling mechanisms that still met the DRAM cycle time. Designs

such as the reinforcement-learning-based scheduler [28, 52] require multiple CPU cycles to make a single scheduling decision.

Based on recent DRAM trends, however, we conjecture that this approach to memory scheduler design is not sustainable. Figure 2.4 shows that DDR memory bus frequencies have grown steadily over the last decade, doubling every three to four years, whereas CPU frequencies have remained more or less stagnant [9, 14, 24, 26]. This growth will continue with DDR4, where devices are already expected to reach a data transfer rate of 3200 MT/s [31]. The result is that integrated memory controllers running at the CPU clock will have fewer available cycles to make a scheduling decision. Ultimately, we will no longer be able to conduct such high degrees of inference within the memory controller, as the complex circuitry (and associated critical timing path) will be "squeezed out." As the available time continues to decrease, memory controllers will be forced to revert to simpler scheduler designs.

Conventional wisdom leads us to assume that these simpler designs will result in poorer scheduling algorithms. We believe this is a flawed conclusion, based in part on an archaic scheduler design approach—scheduling decisions are taken wholly by the memory controller, yet virtually the only information supplied with each memory request is its effective address. In this scenario, the only way to attain high-performance scheduling from such limited observation of a program's execution is indeed to resort to heavyweight controller designs. Ideally, memory controllers need to retain the sophistication of complex controllers without somehow requiring the associated circuitry along the critical path.

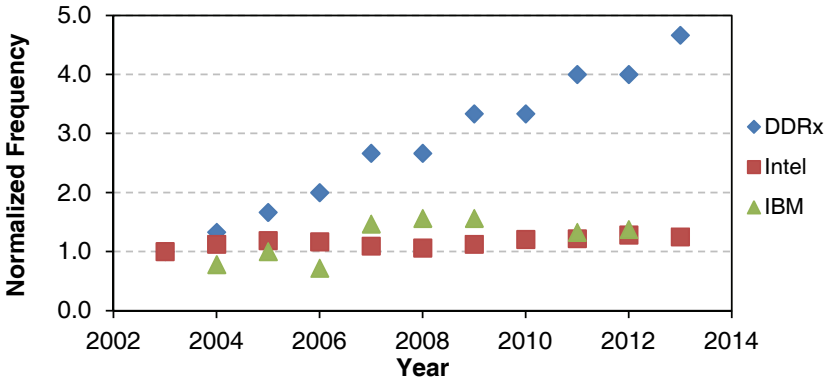

Figure 2.4: Peak frequency of DDR memories and processors released in each of the last ten years, normalized to the highest released frequency in 2003 [9, 14, 24, 26].

2.2 Criticality: Targeted System Optimizations

Numerous hardware optimizations in computer architecture target individual instructions. Oftentimes, these optimizations can be too costly to apply to every instruction [15]. The concept of *criticality* allows architects to identify a subset of these instructions where such optimizations will be most effective. The goal of such targeted optimization is to reap the majority of the benefits that one would achieve applying the optimization globally, while saving on expensive resources such as power or hardware complexity. Naturally, this does not come for free—criticality identification mechanisms are often placed in hardware and can involve sophisticated analysis, which in turn results in significant overheads.

There are two general subfields of criticality relevant to our work. *Instruction criticality* identifies a subset of all instructions within a program that should be targeted to improve overall program objectives (often performance). *Load criticality*, on the other hand, is a more ill-defined concept that typically uses

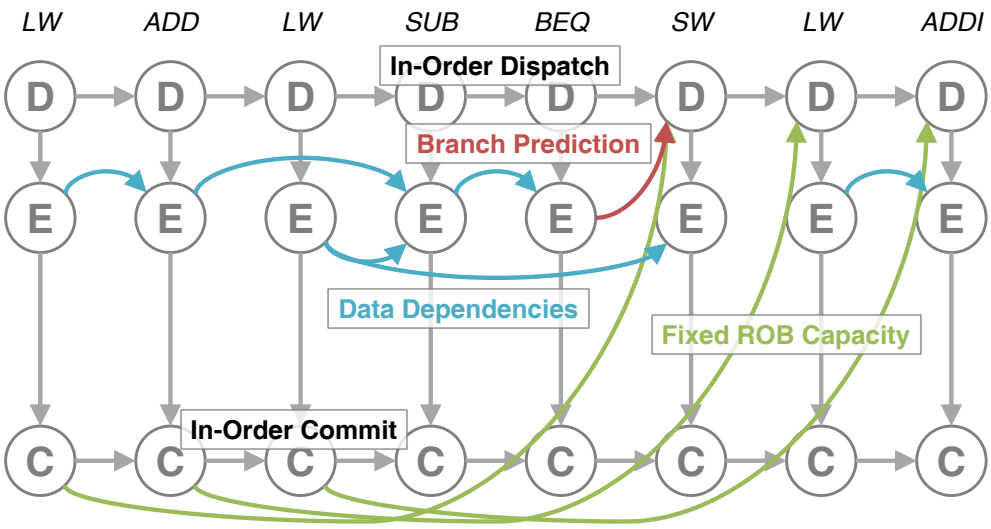

Figure 2.5: Example directed acyclic graph for Fields instruction criticality. In addition to in-order dispatch and commit, $E \rightarrow E$ edges capture data dependencies, $C \rightarrow D$ edges capture the fixed capacity of the reorder buffer, and $E \rightarrow D$ edges capture branch misprediction penalties.

one of a number of ad-hoc metrics to identify which load instructions should be prioritized by the memory subsystem.

2.2.1 Instruction Criticality

Instruction criticality research has mainly focused on identifying which instructions within the processor fall along the *critical path* of program execution. Due to the ability of superscalar out-of-order processors to exploit instruction level parallelism (ILP) and memory level parallelism (MLP), the latencies of several instructions will overlap, and many instructions with full latency overlap will have their delays masked completely by longer, more critical instructions whose longer delays contribute to further delaying the completion of the program. The premise of instruction criticality is that these critical instructions are chosen for

optimizations that are applied within the processor pipeline, which would otherwise have be applied to all in-flight instructions. A number of different instruction criticality prediction mechanisms have been proposed and used for a variety of purposes; these are discussed in more detail in Section 7.1. Here, we shall focus on a more formal approach to critical instruction identification.

Fields et al. proposed the use of a directed acyclic graph (DAG) to capture the critical path of program execution for a single thread within an out-of-order processor [15, 16]. Figure 2.5 shows an example criticality DAG. Each dynamic instance of an instruction is modeled as three timing nodes: dispatch time (D), execution time (E), and commit time (C). These nodes can be saved as timestamps when a particular event finishes taking place, and in the case of the E node represents when the instruction exits the ALU [16]. Ordering and dependencies, both within an instruction and across instructions, are captured in the DAG as directed edges. While most of these edges are statically annotated based on correct program behavior and true data dependencies, some edges attempt to capture structural hazards within the processor. The modeled edges include:

- $D \rightarrow D$, to represent in-order dispatch;
- $D \rightarrow E$, as instructions can only execute after being dispatched;
- $E \rightarrow D$, to account for branch misprediction penalties;
- $E \rightarrow E$, to capture data dependencies amongst instructions;
- $E \rightarrow C$, as instructions can only commit after being executed;
- $C \rightarrow D$, to account for a finite reorder buffer (ROB) capacity; and
- $C \rightarrow C$, to represent in-order commit.

The fully-constructed DAG can then identify the critical path through a simple back-propagation algorithm [15]. The commit time of the final instruction is equivalent to the time at which the program completed. From this node, the DAG can be traversed in reverse, by identifying which of the edges were *last-arriving* (i.e., which source nodes for the edges have the most recent timestamp). The intuition is that the event responsible for the last-arriving edge is the reason that the destination node could not complete earlier, and that the arrival times for other, earlier sources did not have any material effect on that time (i.e., they have *slack* [17]). By recursively applying this algorithm to the source nodes that resulted in the last-arriving edges, a path through the DAG can be found of nodes whose *lateness* directly prevented other instruction events from taking place, and thereby increased the program execution time. Note that it is perfectly acceptable for a node to have two last-arriving edges, and that in such cases, the critical path is considered to have diverged. All traversed critical paths will reach the dispatch node of the first instruction in the program, due to the way in which the graph has been constructed, thus ensuring that critical path identification through back propagation will identify the path through the entire program [16].

This approach represents the generally accepted method of performing static critical path identification. Unfortunately, it has a number of shortcomings for direct application to memory-level optimizations. The Fields model does not explicitly capture any interactions that take place within memory, such as memory access time and request coalescing. While this worked for the original study, which simulated a fixed-latency memory [15], a more realistic memory model exhibits much greater latency variability, and can have a large effect on the overall critical path. Another issue is that a static approach to criticality approach

can identify the most critical instructions, but will not account for other instructions that, while previously not critical due to their masked latency, now find themselves on the critical path. This can often happen when a long-latency instruction is optimized, and a second instruction with limited slack no longer has its complete latency masked by the now-optimized instruction. Such dynamic effects are important to capture at runtime in order to find further opportunities for improvement.

Despite these potential issues, a formal approach to critical path analysis can be beneficial for analyzing the performance of other predictors. As a potential direction for future work, we propose one way of incorporating these memory-level interactions into the Fields model in Section 9.1.

2.2.2 Load Criticality

A general instruction criticality analysis can be selectively applied to load instructions in order to perform memory-side optimizations.[3] However, as these more general mechanisms are not specifically targeted towards memory operations, they may not reveal enough information about these instructions. The intent of criticality is to differentiate between instructions and find those ones expected to provide the most improvement when selected. A successful criticality detection algorithm for memory should provide significant differentiation between different load instruction instances. In the general criticality case, the latency of an operation can often be used to distinguish the criticality of an

[3]Store operations are generally not considered, as stores do not hold up forward progress within the processor. Instead, their associated writes are queued within a buffer, and opportunistically drained by the system when appropriate. As such, store operations are rarely expected to fall along the critical path of execution.

instruction. Unfortunately, from the example viewpoint of DRAM, where all requests reaching DRAM incur long latencies, all of these instruction instances could potentially be marked as critical, in effect providing no differentiation amongst these loads.

Since memory-based optimizations only need to analyze memory instructions, a number of more targeted criticality detection mechanisms have been proposed. Due to the often complex nature of interactions throughout the memory system, no formal method of memory instruction classification has been proposed to date. Instead, a selection criterion is chosen, often based on expert intuition, with the expectation that it will correlate well with the identification of memory instructions that are most important to target. The community has proposed several such criteria for critical load identification, but have not achieved any consensus on whether any one metric is better suited than the others.

The wide range of load criticality criteria will be discussed further in Chapter 5. For now, we will focus on two fine-grained metrics for load criticality identification, which have wide applicability for our target optimizations.

Direct Consumer Count

Subramaniam et al. proposed a load criticality predictor based on the observation that loads with a larger number of consumer instructions are more likely to be critical to the program's execution, and thus the number of consumers can be used as an indicator of criticality [73]. They add counters to the ROB to track direct dependencies only, which can be determined when consumers enter the rename stage. The number of consumers is then stored in a PC-indexed Criti-

cal Load Prediction Table (CLPT), and if this count exceeds a certain threshold (which, as they show, is application-dependent), they mark the instruction as critical the next time it is issued.

From the perspective of the memory scheduler, we hypothesize that this measure of criticality may be informative, even if only a fraction of loads marked critical may ever be seen by the memory scheduler (the L2 misses). Thus, we include this criterion for criticality in our scheduler study (Section 4). We explore two configurations: One which simply marks L2 misses as critical or not according to the predictor (*CLPT-Binary*), and another one where the dependency count used by the predictor is actually supplied to the memory scheduler (*CLPT-Consumers*), so that the scheduler can prioritize among the L2 misses marked critical.

Long-Latency Loads Blocking at Commit Time

Recall that in out-of-order processors, once load instructions are issued to memory and their entries are saved in the load queue, these instructions exit the execution stage but remain in the ROB until the requested operation is complete and the appropriate value is supplied. This means that while other resources of the back end have been freed up, long-latency load instructions may reach the ROB head before they complete, where they will block the commit stage, possibly for many cycles. A long-latency block at the ROB head can lead to the ROB filling up, and eventually prevent the processor from continuing to fetch/dispatch new instructions. In the worst case, this may lead to a complete stall of the processor.

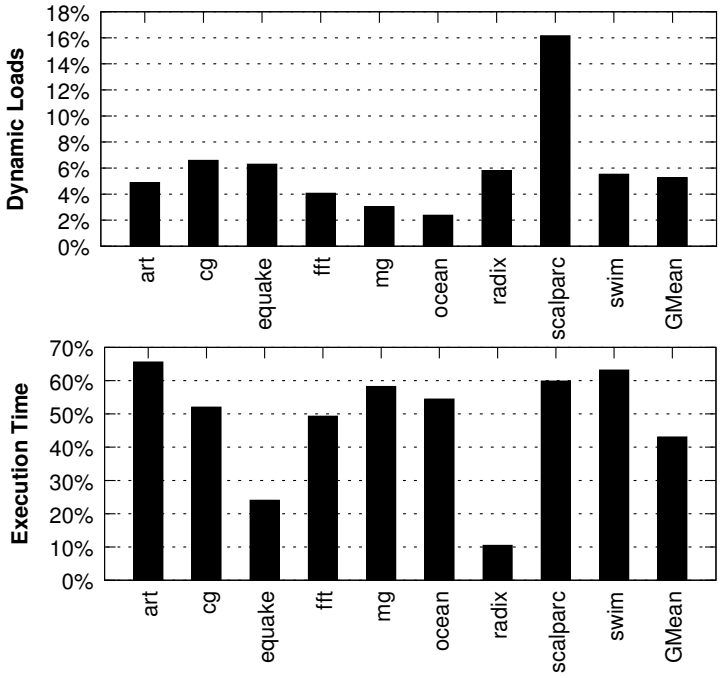

Figure 2.6: Percentage of dynamic long-latency loads that block at the ROB head (top), and percentage of processor cycles that these loads block the ROB head (bottom) when scheduled using FR-FCFS, averaged across all threads of each parallel application.

Runahead and CLEAR try to attack this problem by extending the architecture with special execution modes. In Runahead, when a long-latency load blocks the ROB head, the architectural state is checkpointed, allowing execution to continue, albeit "skipping" instructions that are in the load's dependency chain (easily detectable by "poisoning" the destination register) [10, 54]. After the load completes, the processor systematically rolls back to that point in the program order. The goal is to use Runahead mode to warm up processor and caches in the shadow of the long-latency load. In CLEAR, a value prediction is provided to the destination register instead of poisoning it, which allows the hardware to leverage the (correct) execution in the shadow of that load when the prediction is correct, or to still warm up processor and cache structures otherwise [39]. Checkpoint support is still needed.

Targeting these loads to "unclog" the ROB could significantly reduce the processor critical path. Figure 2.6 shows that while these loads only account for 6.1% of all dynamic loads, on average, they end up blocking the ROB head for 48.6% of the total execution time. (See Section 4.2 for experimental setup.) This is consistent with the findings of Runahead and CLEAR.

We hypothesize that this criterion for criticality may be useful to a criticality-aware memory scheduler. Consequently, our study of the memory scheduler (Chapter 4) includes it as well. Note that we only use Runahead/CLEAR's concept of load criticality: Our mechanism is devoid of the checkpoint/rollback execution modes characteristic of those proposals.

CHAPTER 3

THE COMMIT BLOCK PREDICTOR

Loads that block at the head of the reorder buffer (ROB) do so for a large amount of program execution time, as shown in Figure 2.6. Since these stalls are only triggered by a small handful of dynamic load instances, we believe that targeting these loads for prioritization within the memory subsystem could boost overall program performance. We expect that such loads contribute greatly to the critical path of program execution.

Unlike the consumer count mechanism of Subramaniam et al. [73], neither Runahead [10, 54] nor CLEAR [39] *directly* use a criticality predictor, since their mechanisms do not activate until the loads actually block the ROB head (i.e., they implicitly predict that such loads are critical). As a result, we must design a predictor that can inform the scheduler as soon as the load issues. As we shall see in Section 4.3.1, simply detecting the stall once it occurs will not be of much help, as several opportunities for prioritizing the request will be missed.

We will evaluate the utility and overhead of this predictor in the context of two mechanisms—memory scheduling (Chapter 4) and hardware prefetching (Chapter 6).

3.1 Design

We propose a new hardware table, the *Commit Block Predictor* (CBP), that tracks loads which have previously blocked the ROB. Figure 3.1 shows how the CBP interacts within the processor. When a load instruction blocks the ROB head, the predictor is accessed and annotated accordingly. In our evaluation, we explore

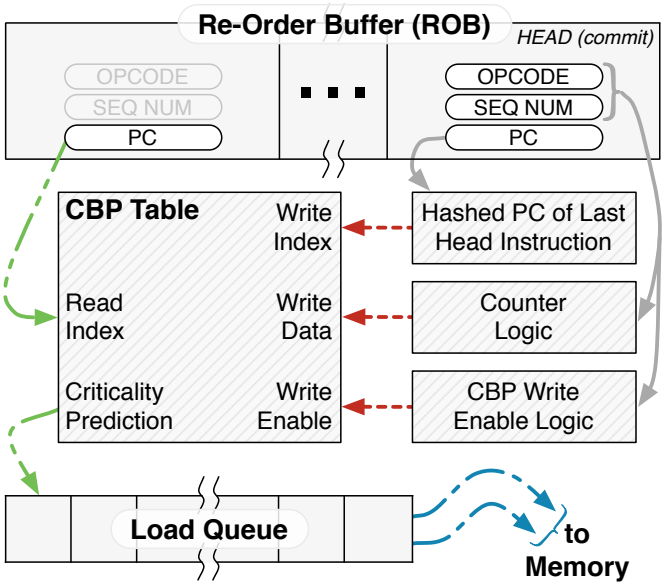

Figure 3.1: Overview of Commit Block Predictor (CBP) operation. Solid gray lines represent the per-cycle register counter and updates, dashed red lines illustrate a write to the CBP table (when a stalled load commits), green dash-dot-dot lines show the table lookup for loads to be issued, and blue dash-dot lines depict load issue to memory.

different annotations: *(a)* a simple saturating bit, *(b)* a count of the number of times the load has blocked the ROB head, or *(c)* the load's stall time. (The stall time can only be written to the CBP once the load commits. Section 3.2 goes into these options in more detail.) When a load instruction with the same (PC-based) CBP index is issued to the memory subsystem in the future (i.e., another dynamic instance of the same static load is being issued), it is flagged as critical. On a last level cache miss, this flag is sent along with the address request to memory, where it will be caught by the scheduler.

Detecting that a memory instruction remains at the head of the ROB requires two pieces of logic: hardware to detect that the instruction at the head is a load, and hardware to recognize that the instruction currently at the head is the same

one that was there the previous cycle. A load can be recognized by reading opcode-related status bits. In order to recognize that an instruction has been waiting at the ROB head, a register stores the ROB sequence number for the instruction that was at the ROB head the previous cycle. If this matches the sequence number of the current head instruction, then a stall is successfully detected.

The CBP table is simply an SRAM indexed by the PC, assuming that the criticality behavior is recurrent for all dynamic instances of the same static instruction. Whenever a load blocks at the head of the ROB, we make a copy of the PC bit substring, which will be used to index the CBP table. (As in branch prediction tables, we index the finite table with an appropriate bit substring of the PC, resulting in some degree of aliasing.) In its simplest form, the table simply records *if* a load stalls—we call this *Binary* criticality prediction. In this case, the state of the table does not change when a load is no longer detected as critical—the prediction effectively saturates.

The contents of the SRAM table can optionally be reset after a fixed interval. In our studies, we implement this reset logic as a simple countdown timer that clears the entire table at once. We evaluate the impact of table aliasing and the usefulness of CBP resetting in the context of memory scheduling in Section 4.3.3, while the CBP tables used for prefetching in Chapter 6 perform a global reset every 100,000 processor cycles.

3.2 Ranking Degrees of Criticality

Previous implementations of criticality have typically only used a binary metric, as an instruction is either on the critical path or not. While useful in itself, such a binary metric fails to provide stratification amongst the marked requests. As an example, each DRAM transaction queue in the memory scheduler contains more than one critical load for 30.1% of the overall execution time on average (in comparison, the queue contains at least one critical load 49.2% of the time). We can potentially improve the performance of our criticality-aware memory systems by distinguishing amongst these critical loads. As a result, we choose to extend our idea of criticality further: The more time a particular load contributes to the critical path, the more important it may be to address that load. For example, if we have to choose between speeding up a load that stalls the ROB for 5 cycles and one that stalls the ROB for 250 cycles, we may choose to speed up the 250-cycle one. (This, however, may not always be the best choice, because a long-blocking stall might simply be masking other delays within the processor pipeline.) Based on this hypothesis, it seems intuitive that we may benefit from ranking critical loads in some order of importance.

As mentioned before, we choose to evaluate several different metrics for our CBP that have the potential to capture this importance. The first, *BlockCount*, counts the number of times a load blocks the ROB, regardless of the amount of time it stalls for; this is based on the belief that if a load continues to block the ROB, servicing that particular load will be more effective than tracking and optimizing a load that only blocks the ROB a handful of times. We also look at *LastStallTime*, a short-term memory to see the most recent effect a load had on the ROB, while allowing us to be agnostic of older, potentially outdated behav-

ior. Another metric evaluated is *MaxStallTime*, which effectively captures the long-term behavior of a critical load through the largest single observed ROB stall duration. This assumes that if a load is found to be more critical than others, it is likely to remain relatively more critical, regardless of how subsequent scheduling may reduce the magnitude of stalling. Finally, we measure *TotalStallTime*, which accumulates the total number of cycles across the entire execution for which a load has stalled at the head of the ROB. We use this to capture a combination of block count and dynamic instruction stall time, although we are aware of its potential to skew towards frequently-executed loads even if they may no longer be critical.

Figure 3.1 shows a high-level overview of how the CBP interacts with the ROB. Consider the criterion where criticality is measured based on a blocking load's maximum ROB stall time (the other proposed criteria are equally simple or simpler). We then use a counter to keep track of the duration of the stall as follows: Every cycle, the sequence number of the current ROB head is compared to the saved sequence number. (For a 128-entry ROB, this represents a 7-bit equivalence comparator.) If it is equal, then the stall counter is incremented. (We shall discuss the size of this counter in Section 4.3.6.) If it is not equal, the saved PC is used to index the CBP table, which is a small, tagless direct-mapped array. We then read the entry currently stored for that PC substring, and pass it into a comparator, which will check to see if the value in the stall counter is greater than the value in already in the table. If the stall counter value is greater, it will then written to the table; otherwise, the entry will remain unaltered. For either outcome, the stall counter itself is reset, so that it can prepare to track the next instruction.

Since these ranked criticality metrics require the CBP to update data already stored within the table (with the exception of *LastStallTime*, which simply overwrites the old entry), the CBP tables will require an extra read port. They will also need minimal additional logic (such as a comparator in the case of *MaxStallTime*) in order to perform the value update itself. The storage overhead due to the additional ranking data is discussed in Section 4.3.6.

CHAPTER 4

CRITICALITY-BASED MEMORY SCHEDULING

While we tend to decouple memory schedulers from the processor, using processor-side information to assist the controller can be beneficial for two reasons. First, such information can greatly improve the quality of scheduling decisions, providing a form of load instruction analysis that the memory cannot perform for lack of data. Second, as successive generations of memory increase in frequency, the amount of complexity that we can add to the memory controller (which must make scheduling decisions within a clock cycle) decreases greatly.

On the processor side, however, it is common to have sophisticated predictors that measure program behavior over time and eventually influence execution. Instruction criticality is one such processor-side metric. Whereas the notion of load criticality used in earlier memory scheduling proposals [28, 36] is typically from the memory's perspective and tends to be solely age-based, proper instruction criticality can be used to determine which instructions (in our case, which loads) contribute the most to the overall execution time of the program. Intuitively, if we target the loads that stall the processor for the longest amount of time, we can significantly reduce run time. By informing the controller about which loads are most urgent from the processor's perspective, a simple scheduling mechanism can afford them priority in the memory system.

Specifically, we propose to pair a priority-based memory scheduler with our Commit Block Predictor (CBP) from Chapter 3. The CBP is a simple processor-side mechanism to predict load instructions that may block a core's reorder buffer (ROB) in a CMP, and potentially stall the processor pipeline. Using these

very small, simple per-core predictors, we can track such blocking loads, and bring them to the attention of the scheduler, where they are afforded priority.

Using a sophisticated multicore simulator that includes a detailed DDR3 DRAM model, we show that pairing this mechanism up with a lean FR-FCFS scheduler [65] can improve performance by 9.3%, on average, for parallel workloads on an 8-core CMP, with essentially no changes in the processor core itself. We show that the hardware overhead of the prediction logic is very small, and that the simple design is well-suited for high-speed memory technologies. We compare the performance and design features of our proposed scheduler against several state-of-the-art schedulers, using both parallel applications and multiprogrammed workloads.

4.1 Using Criticality Information in the Memory Scheduler

We add our concept of load criticality into the FR-FCFS memory scheduler [65]. The basic FR-FCFS algorithm calls for CAS commands to be prioritized over RAS commands, and in the case of a tie, the oldest command is issued. We choose two arrangements in which we add criticality to the scheduler. The first, *Crit-CASRAS*, prioritizes commands in the following order: *(1)* critical loads to an open row (CAS), *(2)* critical loads to a closed row (RAS), *(3)* non-critical commands to an open row, and *(4)* non-critical commands to a closed row, with the tiebreaker within priority groups selecting the oldest command. The second, *CASRAS-Crit*, uses the following priority: *(1)* critical CAS, *(2)* non-critical CAS, *(3)* critical RAS, and then *(4)* non-critical RAS, again breaking ties by choosing older over younger commands.

Note that *Crit-CASRAS* requires an extra level of arbitration not currently in FR-FCFS, whereas *CASRAS-Crit* may leverage the tie-breaking age comparator to incorporate criticality. As we shall see later, the performance of both schemes is identical, even if we assume no extra delay for *Crit-CASRAS*, and thus we advocate for the more compact *CASRAS-Crit* implementation.

The PC of each new dynamic load is used to index the CBP table and read the predicted criticality. There are several alternatives for implementing this lookup. The prediction can be retrieved at load issue, either by adding the PC bit substring to each load queue entry, or by using the ROB sequence number (already in the entry) to look up the PC inside the ROB. Retrieving at load issue requires a CBP table with two read ports and one write port, as our architecture assumes that up to two loads can be issued each cycle. Alternatively, we can perform the lookup at decode, and store the actual prediction in each load queue entry. As the PCs will be consecutive in this case, we can use a quad-banked CBP table to perform the lookup. Our evaluation assumes that we retrieve at load issue and add the PC substring to the load queue, but our storage overhead estimations (Section 4.3.6) consider the cost of all three possibilities.

The criticality information read from the CBP is piggybacked onto the memory request (the bus is expanded to accommodate these few extra bits—see Table 4.5). In the case of an L2 miss, the information is sent along with the requested address to the memory controller, where it is saved inside the transaction queue. In the FR-FCFS scheduler, the arbiter already contains comparators that are used to determine which of the pending DRAM commands are the oldest (in case of a tie after selecting CAS instructions over RAS instructions). We can simply prepend our criticality information to the sequence number (i.e., as

Figure 4.1: Combined criticality and request age for use by the *CASRAS-Crit* memory scheduler. While this only illustrates a *Binary* criticality prediction, the concept trivially extends to any of our ranked metrics from Section 3.2.

upper bits). As a result, the arbiter's operations do not change at all, and we only need to widen the comparators by the number of additional bits. By placing the criticality magnitude as the upper bits, as shown in Figure 4.1, we prioritize by criticality magnitude first, using the relative age of the request only in the event of a tie. To avoid starvation, we conservatively cap non-critical memory operations to 6,000 DRAM cycles, after which they will be prioritized as well. We observe in our experiments that this threshold is never reached.

4.2 Experimental Methodology

4.2.1 Architectural Model

We assume an architecture that integrates eight cores with a quad-channel, quad-ranked DDR3-2133 memory subsystem. Our memory simulation faithfully models all DRAM timing latencies for a Micron MT41J128M8 DDR3 DRAM device [49]. The microarchitectural features of the baseline processor are shown in Table 4.1; the parameters of the DDR3 memory subsystem are shown in Table 4.2. We implement our model using a modified version of the SESC simulator [63] to reflect this level of detail in the memory.

We explore the sensitivity of our proposal to the number of ranks, the memory speed, and the size of the load queue in Section 4.3.5. The trends reported in this chapter were also observed using a slower DDR3-1066 model.

4.2.2 Applications

We evaluate our proposal on a variety of parallel and multiprogrammed workloads from the server and desktop computing domains. We simulate nine memory-intensive parallel applications, running eight threads each, to completion. Our parallel workloads represent a mix of scalable scientific programs from different domains, as shown in Table 4.3.

For our multiprogrammed workloads, we use eight four-application bundles from the SPEC 2000 and NAS benchmark suites, which constitute a healthy mix of CPU-, cache-, and memory-sensitive applications (see Table 4.4). In each case, we fast-forward each application for one billion instructions, and then execute the bundle concurrently until *all* applications in the bundle have executed at least 500 million instructions each. For each application, results are compared using only the first 500 million instructions. Reference input sets are used.

4.3 Evaluation

In this section, we first examine performance of adding binary criticality to FR-FCFS, observing the behavior under our two proposed approaches (see Section 4.1). We then explore the impact of ranked criticality. Afterwards, we try to gain insight as to why the CBP-based predictors outperform the CLPT pre-

Table 4.1: Parameters of the simulated architecture.

Frequency	4.27 GHz
Number of Cores	8
Fetch/Issue/Commit Width	4 / 4 / 4
Int/FP/Ld/St/Br Units	2 / 2 / 2 / 2 / 2
Int/FP Multipliers	1 / 1
Int/FP Issue Queue Size	32 / 32 entries
ROB (Reorder Buffer) Entries	128
Int/FP Registers	160 / 160
Ld/St Queue Entries	32 / 32
Max. Unresolved Branches	24
Branch Misprediction Penalty	9 cycles min.
Branch Predictor	Alpha 21264 (tournament)
RAS Entries	32
BTB Size	512 entries, direct-mapped
iL1/dL1 Size	32 kB
iL1/dL1 Block Size	32 B / 32 B
iL1/dL1 Round-Trip Latency	2 / 3 cycles (uncontended)
iL1/dL1 Ports	1 / 2
iL1/dL1 MSHR Entries	16 / 16
iL1/dL1 Associativity	Direct-mapped / 4-way
Memory Disambiguation	Perfect
Coherence Protocol	MESI
Consistency Model	Release consistency
Shared L2 Cache	4 MB, 64 B block, 8-way
L2 MSHR Entries	64
L2 Round-Trip Latency	32 cycles (uncontended)

Table 4.2: Micron DDR3-2133 DRAM model [49].

Transaction Queue	64 entries
Peak Data Rate	17.067 GB/s
DRAM Bus Frequency	1,066 MHz (DDR)
Number of Channels	4 (2 for quad-core)
DIMM Configuration	Quad rank per channel
Number of Banks	8 per rank
Row Buffer Size	1 KB
Address Mapping	Page interleaving
Row Policy	Open page
Burst Length	8
$\mathbf{t_{RCD}}$	14 DRAM cycles
$\mathbf{t_{CL}}$	14 DRAM cycles
$\mathbf{t_{WL}}$	7 DRAM cycles
$\mathbf{t_{CCD}}$	4 DRAM cycles
$\mathbf{t_{WTR}}$	8 DRAM cycles
$\mathbf{t_{WR}}$	16 DRAM cycles
$\mathbf{t_{RTP}}$	8 DRAM cycles
$\mathbf{t_{RP}}$	14 DRAM cycles
$\mathbf{t_{RRD}}$	6 DRAM cycles
$\mathbf{t_{RTRS}}$	2 DRAM cycles
$\mathbf{t_{RAS}}$	36 DRAM cycles
$\mathbf{t_{RC}}$	50 DRAM cycles
Refresh Cycle	8,192 refresh commands every 64 ms
$\mathbf{t_{RFC}}$	118 DRAM cycles

Table 4.3: List of simulated parallel applications and their input sets.

	Data Mining	[59]
scalparc	Decision tree	125k pts., 32 attributes
	NAS OpenMP	[5]
cg	Conjugate gradient	Class A
mg	Multigrid solver	Class A
	SPEC OpenMP	[4]
art-omp	Self-organizing map	MinneSPEC-Large
equake-omp	Earthquake model	MinneSPEC-Large
swim-omp	Shallow water model	MinneSPEC-Large
	SPLASH-2	[78]
fft	Fast Fourier transform	1M points
ocean	Ocean movements	514×514 ocean
radix	Integer radix sort	2M integers

Table 4.4: List of multiprogrammed workloads. P, C, and M are processor-, cache-, and memory-sensitive, respectively [5, 22].

AELV	ammp - ep - lu - vpr	C P C C
CMLI	crafty - mesa - lu - is	P P C M
GAMV	mg - ammp - mesa - vpr	M C P C
GDPC	mg - mgrid - parser - crafty	M C C P
GSMV	mg - sp - mesa - vpr	M C P C
RFEV	art - mcf - ep - vpr	C M P C
RFGI	art - mcf - mg - is	C M M M
RGTM	art - mg - twolf - mesa	C M M P

dictor proposed by Subramaniam et al. We then quantify our hardware overhead. Finally, we compare our scheduler against AHB [23], MORSE-P [28, 52], PAR-BS [53], and TCM [38].

4.3.1 Naive Predictor-Less Implementation

We first examine the usefulness of sending ROB stall information only at the moment a load starts blocking the ROB. Without any predictors, we simply detect in the ROB when a load is blocking at the head, and then forward this information to the memory controller, which in all likelihood already has the memory request in its queue. *For this naive experiment only*, we optimistically assume that extra side channel bandwidth is added to the processor, allowing us to transmit the data (the load ID and criticality flag) from the ROB to the DRAM transaction queue. (We do, however, assume realistic communication latencies.)

Using this forwarding mechanism, we achieve an average speedup of 3.5%, low enough that one could consider it to be within simulation noise. This poor performance may be attributed to the lack of a predictor: As we do not have any state that remembers the behavior of these loads, subsequent instances of the static load will again only inform the memory controller when the new dynamic instance blocks the ROB head once more. We therefore use a predictor in our implementation (without forwarding at block time) to prioritize these blocking loads earlier in their lifetime, with the hope of further reducing their ROB stall time.

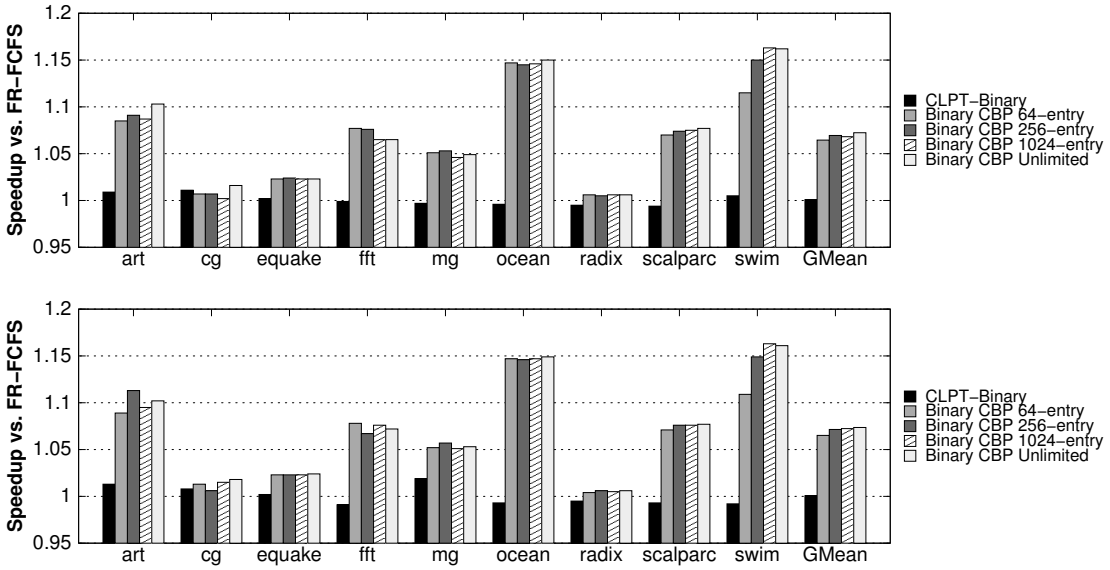

Figure 4.2: Speedups from *Binary* criticality prediction (sweeping over CBP table size) within the memory scheduler, using the *Crit-CASRAS* (top) and *CASRAS-Crit* (bottom) algorithms.

4.3.2 First Take: Binary Criticality

We study the effects of adding criticality to the FR-FCFS scheduler, as proposed in Section 4.1. We evaluate our CBP tables, as well as the Critical Load Prediction Table (CLPT) mechanism proposed by Subramaniam et al. [73]. As discussed in Section 2.2.2, we believe that their method of determining criticality also has the potential to inform the memory scheduler. We reproduce their predictor as described, and from an analysis of our benchmarks, we choose a threshold of at least three consumers to mark an instruction as critical.

Figure 4.2 shows the performance of these two predictors. For a 64-entry *Binary* CBP table, both the *Crit-CASRAS* and *CASRAS-Crit* algorithms achieve 6.5% speedup over baseline FR-FCFS. As expected, prioritizing loads that block the head of the ROB allows execution to resume more quickly, resulting in a tangible improvement in execution time. Furthermore, loads that sit behind the

instructions blocking the ROB head can mask part of their miss latency, reducing their impact (and importance) on the critical path. In Section 4.3.3, we will see that ranking the degree of criticality allows us to achieve greater performance benefits.

Figure 4.2 also shows that increasing the size of the table has little effect on the performance of the scheduler. In fact, the 64-entry *Binary* table gets within one percentage point of the unlimited, fully-associative table (7.4%). We will investigate the impact of table size in more depth in Section 4.3.3. We also note that the *CLPT-Binary* predictor shows no appreciable speedup over FR-FCFS; we discuss this further in Section 4.3.3.

From the results presented so far, the *Crit-CASRAS* and *CASRAS-Crit* algorithms perform on par with each other, displaying the same trends across all of our evaluations. This means that we see roughly equal benefits from picking a critical RAS instruction or a non-critical CAS instruction, and that overall, the cost paid for additional precharge and activate commands is made up for by criticality-based performance benefits. As a result, *we present the remainder of our results with only the* CASRAS-Crit *algorithm*, because as we discussed in Section 4.1, it is simpler to implement in hardware.

4.3.3 Scheduling with Ranked Criticality

As we motivated in Section 3.2, we expect significant benefits from being able to determine how much more critical an instruction is with respect to others. We observe the impact of our four ranking metrics on speedup in Figure 4.3, this time only using a 64-entry table. We also evaluate *CLPT-Consumers*, a ranked

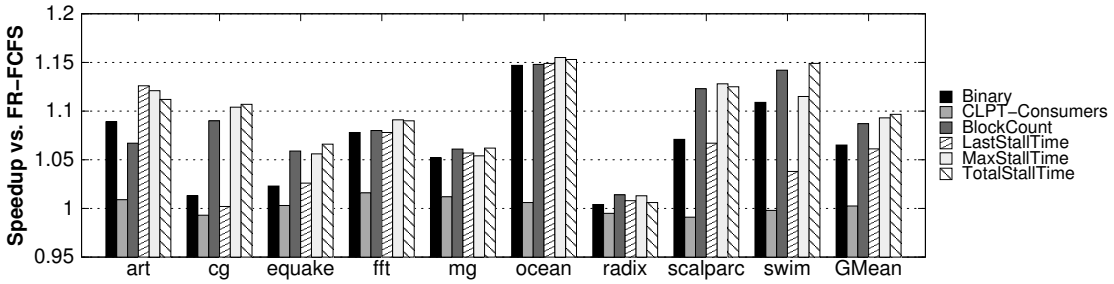

Figure 4.3: Speedups from ranking criticality within the memory scheduler, using the *CASRAS-Crit* algorithm. The *CLPT-Consumers* predictor uses the total consumer count for each load as the criticality magnitude. The CBP tables are 64 entries.

implementation of the CLPT predictor that uses the number of direct consumers to rank the criticality of a load.

For most of the CBP-based ranked predictors, we see improvements across the board over the *Binary* CBP. Using the *BlockCount* CBP improves performance by 8.7% over FR-FCFS. A critical load within a oft-recurring execution loop will stand to reap more benefits over a critical load that is only executed a handful of times, since servicing the more common load every time results in a greater accumulation of time savings in the long run. Using *LastStallTime* does not provide any tangible benefit over binary criticality. One reason could be a ping-ponging effect: if an unmarked load blocks at the head of the ROB for a long time and is subsequently flagged as quite critical, prioritizing the load could significantly reduce its block time, reducing the perceived degree of criticality. When the load reappears, its lower priority means that it is serviced behind other, more critical loads, and again blocks for a long time at the ROB head.

We can avoid this issue by measuring the maximum stall time of a load. At the risk of being oblivious to outlier behavior, we use the maximum stall time as a more stable gauge of how critical a load might be, under the assumption

that if it stalled for this long at some point in time, it is likely to stall for a similar amount in a subsequent iteration. *MaxStallTime* does quite well, with an average speedup of 9.3%. While *TotalStallTime* does perform the best of all of our metrics, the meager increase over *MaxStallTime* does not provide the large synergistic boost hoped for from combining block count and stall time. Ultimately, *TotalStallTime* falls short because it relies too much on history, and tilts in favor of recurrent loads (as their latencies will accumulate more rapidly). Finally, even with ranking, the *CLPT-Consumers* predictor fails to produce speedups.

Using these benchmarks, we can take the worst-case values for each of our predictors and determine an upper bound for the number of bits needed to store each. These are quantified in Table 4.5. Note that the width of the total stall time predictor will depend on two factors: *(a)* the length of program execution, and *(b)* whether the counters are reset at set intervals to account for program phase behavior (which becomes important on hashing collisions). For the purposes of our study, we take the maximum observed value to give an idea of how large the counter can be. One could also implement saturation for values that exceed the bit width, or probabilistic counters for value accumulation [64], but we do not explore these.

Prediction Table Size

We test three CBP table sizes (64 entries, 256 entries, and 1,024 entries) and compare them against a fully-associative table with an unlimited number of entries, which provides unaliased prediction, to see how table size restriction affects performance. Figure 4.2 shows the effect on performance for our binary criticality predictor, and Figure 4.4 shows performance for our *MaxStallTime* predictor.

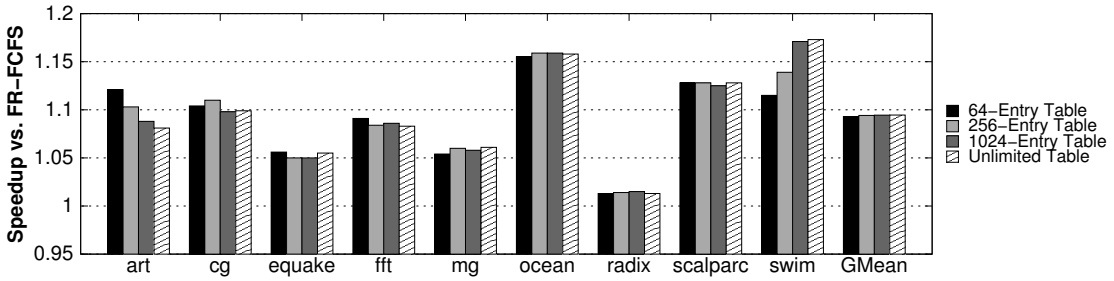

Figure 4.4: Speedups observed while sweeping table size for *MaxStallTime* criticality prediction. The *Unlimited Table* allows for an unrestricted number of entries into a fully-associative table.

We omit results for the other prediction metrics for brevity, but we see near-identical trends in relative performance.

We effectively see no performance drop when we go from an unlimited number of entries down to a 64-entry predictor. Despite there being anywhere from 10^5 to 10^7 critical dynamic loads per thread, these only stem from a few hundred static instructions, for the most part. Since we index our table by the PC of the instruction, we have a much smaller number of loads to track due to program loops. The one exception is *ocean*, which has approximately 1,700 critical static instructions per core. Interestingly, we do not see a notable drop in its performance, possibly because critical loads only make up 2.4% of the total number of dynamic loads. Since our predictor can also effectively pick which loads to defer (we discuss this duality more in Section 4.3.4), we can still in essence prioritize critical loads, despite the fact that 32.4% of the non-critical loads in *ocean* are incorrectly predicted as critical.

There are a couple of applications, *fft* and *art*, where the smaller tables actually outperform the unlimited entry table. The behavior of *art* is a particular anomaly, as it outperforms the unlimited table by a large margin. Upon further inspection, we find that this is due to its large memory footprint, by far

the largest of the SPEC-OMP applications. This is exacerbated by the program structure, which implements large neural nets using two levels of dynamically-allocated pointers. With the large footprint, these double pointers often generate back-to-back load misses with a serial dependency, which are highly sensitive to any sort of memory reordering.

Due to the different ordering, for example, going from an unlimited table to a 64-entry table for our *MaxStallTime* predictor *increases* the L2 hit rate by 3.3%, whereas no other benchmark shows a tangible change in L2 hit rate. This effect is compounded by the fact that our small predictor is quite accurate for *art*, with only 4.8% of non-critical loads incorrectly predicted as critical. This is because *art* has one of the smallest number of static critical loads out of our benchmarks, averaging 156 static critical loads per thread.

Table Saturation

The small table sizes that we use leave our predictor vulnerable to aliasing. We study these effects by comparing the restricted tables versus the content of the unlimited entry table. Of concern is the fact that, on average, 25.4% of our dynamic non-critical loads are being incorrectly marked as critical by the scheduler for our finite table size configurations. Much of this effect is due to table saturation—over time, a larger portion of the table will be marked as critical, eliminating any distinction between the loads. One way to avoid this is to perform a periodic reset on the table contents. Ideally, this not only limits the age of the predictor entries, but it also allows us to adapt better to phase behavior in the applications.

We explore several interval lengths for the table reset (5K, 10K, 50K, 100K, 500K, and 1M cycles). We use our three fastest-executing applications (*fft*, *mg*, and *radix*) as a training set, to determine which of these periods is best suited for our predictor without overfitting to our benchmark suite. For our 64-entry table, the training set performs best on the 100K-cycle interval, for both *Binary* CBP and *MaxStallTime* CBP. We use the remaining six applications as our test set. Without reset, a 64-entry *Binary* table obtained a speedup of 7.5% on the test set (data not shown). Using the 100K-cycle reset, we can improve this to 9.0%, equaling the performance of the unlimited-entry table. The performance differences for *MaxStallTime* are negligible (as we saw previously in Figure 4.4, the 64-entry table already performs almost identically to the unlimited-size configuration).

We also test table reset intervals on the unlimited-entry table. This allows us to determine whether the effects of resetting are due to a reduction in aliasing alone, or if the staleness of the data also contributes to lesser performance. In all cases, though, resetting the unlimited-entry table does not affect performance, suggesting that criticality information is useful in the long term.

Understanding CLPT Performance

Subramaniam et al.'s CLPT predictor has not shown any notable speedups in either binary or ranked magnitude capacity. Recall that CLPT uses the number of direct consumers to determine load criticality. Our simulations show that roughly 85% of all dynamic load instructions only have a single direct consumer, indicating that we do not have enough diversity amongst loads to exploit speedups in the memory scheduler. To see what happens if we increase

the number of critical loads, we re-execute the *CLPT-Binary* predictor using a criticality threshold of 2 (i.e., any load that has more than one direct consumer will be marked critical). Again, the speedups are quite minimal. We believe that the types of loads the CLPT predictor targets are largely complementary to the ones that the CBP chooses to optimize, and that CLPT is likely better suited for the cache-oriented optimizations proposed by Subramaniam et al. [73].

4.3.4 Effect on Load Latency

To gain some additional insight on where the speedups of the criticality scheduler come from, we examine the difference in L2 cache miss latency between critical and non-critical loads, as shown in Figure 4.5. As expected, for all of our benchmarks, we see a drop in the latency for critical loads. A number of these benchmarks show significant decreases, such as *ocean* and *fft*, which correspond to high speedups using our predictors. It is, however, important to note that several benchmarks only show more moderate drops. These moderate drops still translate into speedups because load instructions do not spend their entire lifetime blocking the head of the ROB. In fact, it will take many cycles after these instructions have been issued until they even reach the ROB head, so a significant part of the L2 miss latency is masked by the latency of other preceding instructions. Of the portion of the latency that does contribute to ROB commit stalls, the decrease becomes a much larger proportional drop, hence providing non-trivial speedups.

Interestingly, looking at the non-critical load latencies, we see that for a few applications, these latencies are actually increasing. What this tells us is that our scheduler is exploiting the slack in these non-critical loads, delaying their com-

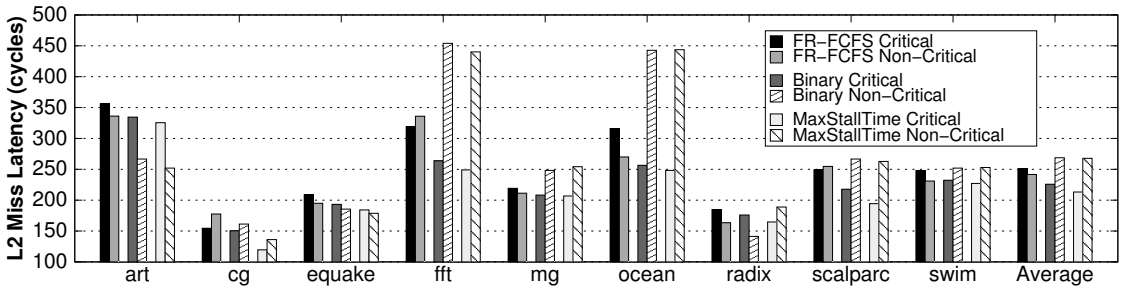

Figure 4.5: Average L2 miss latency for critical and non-critical loads within the memory scheduler, using the *CASRAS-Crit* algorithm and a 64-entry CBP table.

pletion significantly (in deference to critical loads) without affecting execution time, as they do not fall on the critical path.

Again, *art* proves to be an interesting outlier, experiencing a large drop in both latencies. As discussed in Section 4.3.3, the program structure of *art* renders it extremely sensitive to memory reordering. In the case of the *Binary* CBP, we see that, like other benchmarks, *art* sees a drop in the percentage of execution time spent stalling on the ROB. However, unique to *art*, the amount of execution time for which the load queue is full decreases by 17.8%, freeing up queue space to exploit greater memory-level parallelism.

4.3.5 Sensitivity to Architectural Model

To explore how our predictors work over several types of devices available on the market today, we sweep over the number of ranks for both a DDR3-1600 and a DDR3-2133 memory subsystem. Figure 4.6 shows these results, relative to an FR-FCFS scheduler with a single rank for each respective subsystem. With fewer ranks, there is greater contention in the memory controller, as the memory

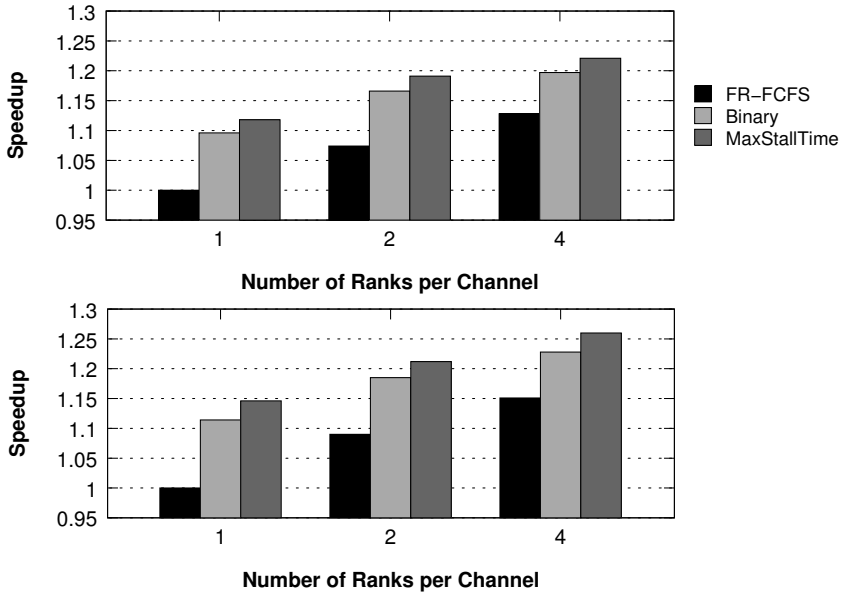

Figure 4.6: Sweep over number of ranks per channel, for DDR3-1600 (top) and DDR3-2133 (bottom) memory. Speedups are relative to a single-rank memory subsystem with an FR-FCFS scheduler.

provides fewer opportunities for parallelization. In these scenarios, we observe that our predictor-based prioritization sees greater performance benefits. For example, a single-rank DDR-2133 memory can see speedups of 14.6% with our 64-entry *MaxStallTime* predictor.

We also explore the impact of the load queue size on our results. With our existing 32-entry load queue, the queue is full for 19.3% of the execution time. Our predictors lower this somewhat, but capacity stalls still remain. Figure 4.7 shows the effects of increasing the load queue size. With 48 entries, we see most of load queue capacity stalls go away. Even then, we still experience speedups of 6.4% for our *Binary* CBP and 8.3% for *MaxStallTime*. Increasing the queue further to 64 entries has a minimal change from the 48-entry results, since we had already eliminated most of the capacity stalls at 48 entries.

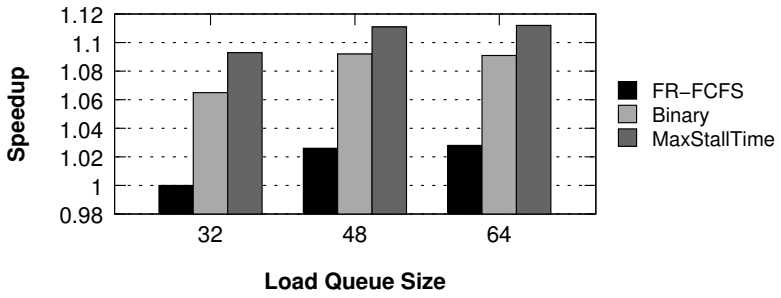

Figure 4.7: Sweep over load queue sizes. Speedups are relative to using an FR-FCFS scheduler along with processors that have a 32-entry load queue.

4.3.6 Storage Overhead

We now quantify the storage overhead required for the *CASRAS-Crit* algorithm described in Section 4.1. We start with the *Binary* predictor. Inside each processor, we need a 7-bit register for the sequence number and a 6-bit register for the PC-based table index, as well as a 7-bit equivalence comparator. For the CBP table, we require a 64 x 1 b table (recall that the CBP table is tagless). As discussed earlier, we may need to expand the load queue depending on the table lookup implementation: For lookup-at-decode, each load queue entry must be expanded by 1 bit, while storing the PC bit substring in the load queue requires 6 bits. The total storage overhead within each core therefore ranges between 77 and 269 bits.

We assume that this data can be sent to the memory controller by adding a bit to the address bus on-chip (between the processors, caches, and the memory controller), in conjunction with the initial load request. Inside the controller, each transaction queue entry requires an extra bit, resulting in another 64 bits of overhead per channel. The comparators of the arbiter must also grow by one bit each. In terms of SRAM overhead (ignoring the enlarged comparators), for our

Table 4.5: Criticality counter widths.

Criticality Metric	Max Obs. Value	Width
Binary	1	1 b
BlockCount	1,975,691	21 b
LastStallTime	13,475	14 b
MaxStallTime	13,475	14 b
TotalStallTime	112,753,587	27 b

8-core quad-channel system, the binary criticality implementation yields 6.5% speedup at a cost of between 109 and 301 bytes. Adding hardware to reset the tables at 100K-cycle intervals can boost this speedup to 7.3%.

The *MaxStallTime* predictor requires 14 bits per entry (Table 4.5). While the sequence number and PC registers remain unchanged, the CBP table must now be 64 x 14 b, and the load queue entries must also be expanded for lookup-at-dispatch, resulting in a total overhead ranging from 909 to 1,357 bits per core. We also need an additional read port on the CBP table, and a 14-bit comparator, to see if the new stall time is greater than the currently-stored maximum. Additionally, the storage overhead within the DRAM transaction queue is now 896 bits, and the arbiter comparators must expand by 14 bits each. For our 8-core processor, this costs between 1,357 and 1,805 bytes of SRAM to obtain a 9.3% speedup.

Using the same methodology, we find that the largest of the candidate predictors, *TotalStallTime*, adds from 2,605 to 3,469 bytes of SRAM, widening comparators by 27 bits.

Table 4.6: Comparison of various state-of-the-art memory schedulers with our proposed CBP-based schedulers.

Scheduler	AHB [23]	TCM [38]	MORSE-P [28, 52]	*Binary* CBP	*MaxStallTime* CBP
Avg. Parallel Application Speedups (rel. to FR-FCFS)	1.6%	0.6%	11.2%	6.5%	9.3%
Avg. Multiprogrammed Workload Weighted Speedups (rel. to PAR-BS)	3.1%	1.9%	11.3%	5.2%	6.0%
Storage Overhead (8 cores, 4 mem controllers)	31 B	4816 B	≤ 512 kB†	109–301 B	1,357–1,805 B
Uses Processor-Side Info	No	No	Yes	Yes	Yes
Scales to High-Speed Mem	Yes	Yes	No	Yes	Yes
Works for Low Contention	Yes	No	Yes	Yes	Yes

†For DDR3-2133. 320 kB to match *MaxStallTime* CBP performance. 128 kB for DDR3-1066.

4.3.7 Comparison to Other Schedulers

We compare our criticality-based scheduler to three state-of-the-art memory schedulers: the adaptive history-based (AHB) scheduler proposed by Hur and Lin [23], the fairness-oriented thread cluster memory (TCM) scheduler [38], and MORSE-P, a self-optimizing scheduler that targets parallel application performance [28, 52]. Table 4.6 summarizes the main differences between these schedulers. They are described in more detail in Section 7.3.

AHB and TCM have simple hardware designs, but unfortunately our results will show that they do not perform as well, as their simplicity does not adapt well to different memory patterns and environments. MORSE-P, on the other hand, is very sophisticated, using processor-side information to adapt for optimal performance. However, as we will show, the original controller design is complex enough that it likely cannot make competitive decisions within a DRAM cycle for faster memory technologies. Our CBP-based predictors combine the best of both worlds, using processor-side information to provide speedups in several scenarios while maintaining a lean controller design.

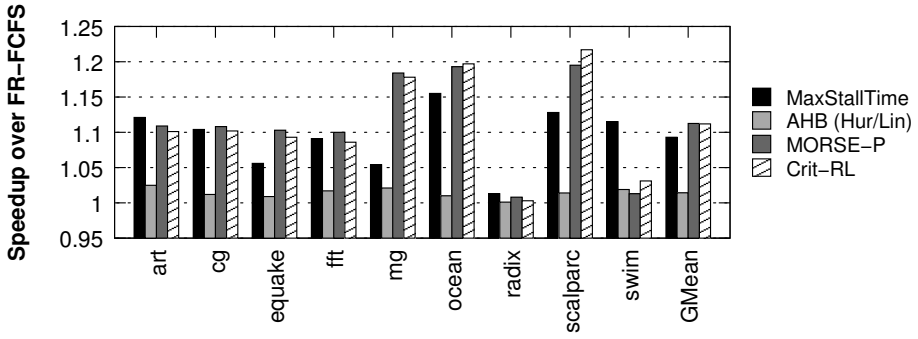

Figure 4.8: Performance of two state-of-the-art schedulers compared to our proposal. *Crit-RL* adds criticality to a self-optimizing memory controller, using the features listed in Table 4.7.

Parallel Applications

Figure 4.8 shows the performance of our schedulers against both AHB and MORSE-P. We see that AHB, which was designed to target a much slower DDR2 system, does not show high speedups in a more modern high-speed DRAM environment. On the other hand, MORSE still does quite well, achieving an 11.2% speedup. (For now, we optimistically assume that MORSE can evaluate 24 commands within a single DRAM cycle–the same as the original paper. As we will see, with high-speed DRAM interfaces, this is unlikely unless significant additional silicon is allocated to the scheduler.)

To study its potential impact, we added our binary and ranked criticality predictions to the MORSE reinforcement learning algorithm as features, using 64-entry prediction tables. We ran multi-factor feature selection [52] from a total of 35 features, including all of the original MORSE-P features, on our training set (*fft*, *mg*, and *radix*). Table 4.7 shows a list of the selected features, in the order they were chosen. One property of feature selection is that for a given branch of features, the feature picked first tends to have the most impact on

Table 4.7: State attributes found using feature selection for *Crit-RL* self-optimizing memory scheduler.

1	Binary Criticality *(prediction sent by processor core from 64-entry table)*
2	Number of Reads to the Same Rank
3	Number of Reads in Queue
4	Number of Writes to the Same Row
5	ROB Position Relative to Other Commands from Same Core
6	Number of Writes in Queue that Reference Open Rows

improved performance. Promisingly, feature selection chose binary criticality first. However, the resulting controller, *Crit-RL*, only matches the performance of MORSE (see Figure 4.8). The lack of improvement implies that MORSE has features which implicitly capture the notion of criticality.

One major disadvantage of MORSE is the long latency required to evaluate which command should be issued. While the original design worked in the context of a DDR3-1066 memory system [52], faster memory clock speeds make this design infeasible. For DDR3-1066, the controller, running at the processor frequency, could be clocked eight times every DRAM cycle; we can now only clock it four times in a DDR3-2133 system (937 ps). As the original design incorporated a five-stage pipeline (1.17 ns latency), we can no longer compute even a single command. Even assuming zero wire delay and removing all pipelining, the CMAC array access latency (180.1 ps, modeled using CACTI [75] at a 32 nm technology) and the latency of the 32-adder tree and the 6-way comparator (approximately 700 ps [52]) leave less than 60 ps to perform the command selection logic. As a result, we believe it is difficult to implement MORSE for high-speed memory.

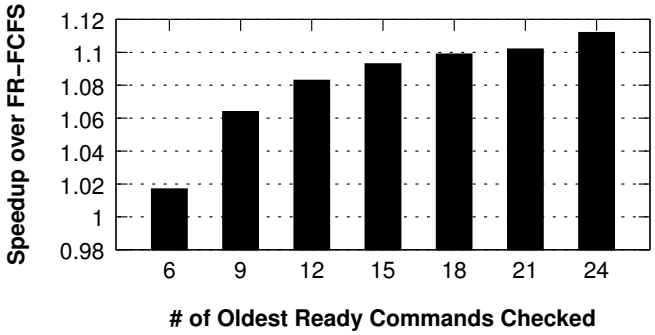

Figure 4.9: Performance of MORSE-P when restricting the number of ready commands that can be considered by the scheduler in a single DRAM cycle.

Let us assume, optimistically, that the latency of evaluating one command in MORSE does indeed fit within one cycle. Without modifying the hardware design, we can now only examine six ready commands per cycle using the original two-way design, with tri-ported CMAC arrays in each way. Additional commands can only be examined by adding more ways, but this comes at the cost of replicating the CMAC arrays (as adding read ports would further increase access latency, which is already too long) and increasing the depth of the comparator tree. Figure 4.9 shows the performance obtained sweeping over different numbers of commands. In each case, when more ready commands exist than can be evaluated, we examine commands by age, oldest first. Achieving the full 24-command potential of MORSE now requires eight ways, resulting in an SRAM overhead of 128 kB per controller. In order to match the performance of our *MaxStallTime* predictor (with 9.3% speedup using at most 1,805 B), MORSE must process 15 commands, requiring 80 kB of overhead (with five ways) per controller.

Multiprogrammed Workloads

We now study the impact of our criticality-based scheduler on multiprogrammed workloads. In this section, we provide our results relative to PAR-BS [53]. We also show results for the more recent TCM proposal [38]. Our multiprogrammed workloads are four-application bundles (see Section 4.2.2). Consequently, in our architecture, we reduce the number of DRAM channels from four to two, to maintain the 2:1 ratio of processor cores to channels used so far. We also cut the number of L2 MSHR entries in half.

We use weighted speedup [68] to quantify the schedulers' effects on throughput. To calculate weighted speedup, the IPC for each application is normalized to the IPC of the same application executing alone in the baseline PAR-BS configuration, as has been done in prior work [45], and then the normalized IPCs are summed together. Compared to PAR-BS, our criticality-based scheduler has a weighted speedup of 5.2% for a 64-entry *Binary* CBP. The best-performing criticality ranking, *MaxStallTime*, yields a weighted speedup of 6.0% (Figure 4.10). We see similar speedups for our other ranking criticality predictors (not plotted here).

As a comparison, we have also implemented TCM [38], which attempts to balance system throughput (*weighted speedup*) with fairness (*maximum slowdown*). Figure 4.10 shows that TCM obtains only a 1.9% weighted speedup over PAR-BS for multiprogrammed workloads.[1] Not only does our predictor outperform TCM in terms of throughput, but it also improves on maximum slowdown, decreasing it by 11.6%.

[1]Unsurprisingly, as both PAR-BS and TCM were designed to target thread heterogeneity, they do not show improvements when applied to our parallel workloads—in fact, PAR-BS experiences an average parallel workload slowdown of 6.4% when compared to FR-FCFS.

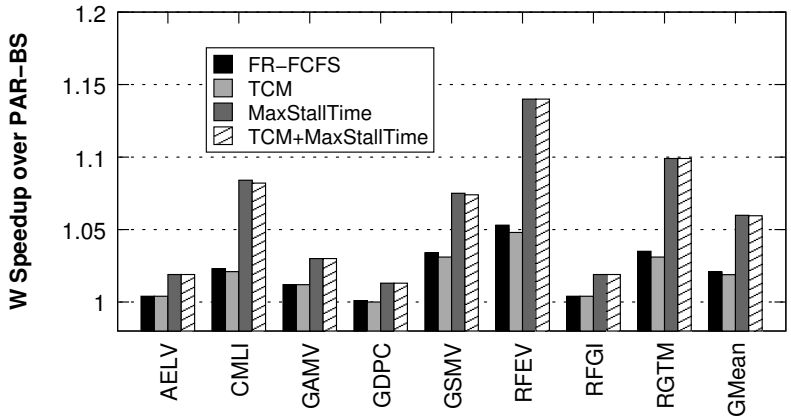

Figure 4.10: Weighted speedups for multiprogrammed workloads from using ranked criticality. Note that weighted speedup is relative to PAR-BS [53]. The CBP table is 64 entries, and the PAR-BS marking cap is set to 5 [53].

The apparent discrepancy from previous TCM results [38] arises from differing workloads and target memory architectures. While the workloads reported for TCM tend to have several memory-intensive programs, our workloads contain a mix of CPU-, cache-, and memory-sensitive applications. Through experimental verification, we observe that our interference amongst programs is much lower than the TCM workloads. We also use a more aggressively parallel memory system, which allows for more concurrent requests and relieves significant memory pressure (see Table 4.2).[2] We show results for our CBP-based predictors under less aggressive systems in Section 4.3.5.

TCM does not perform well for our simulated memory architecture, mainly because the clustering is largely ineffective without large amounts of contention. Since it is clear that clustering could be beneficial in a contentious environment, and that our criticality-based predictor performs well in low contention, we pro-

[2]Though the results shown does not include the XOR-based address-to-bank mapping [82] used in the original PAR-BS work [53], we performed a series of simulations using the mapping. We found that it provided a 2% performance boost for *all* of our mechanisms, indicating that the alternative mapping offers no advantage unique to any one scheduler.

pose combining the two to achieve synergy. This combined scheduler, which we call *TCM+MaxStallTime*, still uses the thread rank from TCM as the main request priority. In case of a tie, whereas TCM would perform FR-FCFS, we instead replace this with criticality-aware FR-FCFS.

Figure 4.10 shows that, even with thread prioritization, we do not exceed the performance of our criticality-based scheduler. Part of this is the result of the latency-sensitive cluster. For our four-application workloads, that cluster will likely only consist of a single thread, which will be the most CPU-bound of the bundle. By definition, latency-sensitive threads are threads that stall waiting on infrequent loads to be serviced, which is very similar to our notion of load criticality. The majority of their memory requests will likely be treated as more critical than those of other threads, which is exactly what the maximum stall time ranking is trying to differentiate. As a result, this redundancy removes much of the expected additional speedup. For the remaining threads, since our environment is less contentious, fairness does not matter much, and as a result, TCM is effectively performing *CASRAS-Crit* scheduling, which is why the TCM and *TCM+MaxStallTime* results look quite similar.

In a high-contention memory design, we expect that *TCM+MaxStallTime* will perform at least as well as TCM (since that is the first-level prioritization). As a result, we believe that combining the two schedulers can provide us with best-of-both-worlds performance.

4.4 Summary

We have shown that processor-side load criticality information may be used profitably by memory schedulers to deliver higher performance. With very small and simple predictors per core in a CMP, we can track loads that block the ROB head, and flag them for the benefit of the memory scheduler, which affords them priority. We quantitatively show that pairing this mechanism with a novel criticality-aware scheduler, based on FR-FCFS [65], can improve performance by 9.3%, on average, for parallel workloads on an 8-core CMP, with minimal hardware overhead, and essentially no changes in the processor core itself. In the face of increasing DRAM frequencies, we believe that such lean memory controllers, which integrate predigested processor-side information, provide an essential balance between improved scheduling decisions and implementability.

CHAPTER 5

THE RELEVANCE OF COMMIT STALLS TO LOAD CRITICALITY

The general concept of load criticality is to identify those load instructions or load requests which should be prioritized by the memory subsystem. While a similar notion has been formally defined for instruction criticality (Section 2.2.1), no such formalization exists specifically for load criticality. Instead, load criticality mechanisms have to date been restricted to ad hoc metric selection. As a result, there is little consistency across different works on which metrics should be used to determine load criticality, making comparison cumbersome.

The main goal of this chapter is to better understand why the selection of loads stalling at commit correlates well as a memory-level prioritization mechanism, and to investigate how effective the Commit Block Predictor is at identifying these loads. This approach aims to provide several insights:

- How well the Commit Block Predictor (CBP) from Chapter 3 works against other ad hoc metrics;
- Performing a load-by-load comparison using traces to identify where the performance gains of the CBP come from; and
- Identifying shortcomings of the CBP that could be used to drive future criticality predictor improvements.

The analysis performed here uses the CBP and other load criticality metrics specifically in the context of memory scheduling. The memory scheduler used throughout this chapter is the *CASRAS-Crit* scheduler, which was defined in Section 4.1.

5.1 A Lack of Consensus

The concept of load criticality has been around for approximately 15 years [18, 72]. While the idea of identifying a select group of loads as more important is a well-accepted idea, no one single criterion for critical load identification has emerged. In fact, a survey of papers over the 15-year period reveals that each load criticality proposal has used a unique metric for identifying these loads. Given this, it stands to reason that our selection of commit stalls may not be the best such indicator of criticality.

In order to get a better sense of these other metrics, we split them into two groups. The first group contains *coarse-grained* criteria for load criticality, where criticality is not assigned based on the individual properties of each load. Instead, an identical classification is assigned to a series of loads, with this classification typically updated in a periodic manner. Many such works do not explicitly refer to load criticality, but in effect are identifying which loads should be prioritized. The second group consists of *fine-grained* load criticality criteria, where each individual load receives its own prediction. Our commit stall based mechanisms falls into this second group, as we make criticality predictions based only on the static instruction itself.

Coarse-grained criteria typically make their predictions based on the properties of phases, threads, or the processor itself. Fisk and Bahar used the instruction issue rate of the processor to determine criticality—loads issued to the memory system during a low issue rate period were classified as critical [18]. A number of priority-based memory scheduling algorithms use coarse-grained criticality. The TCM scheduler prioritizes loads that originate from latency-

sensitive threads over those loads from bandwidth-sensitive threads [38]. The Minimalist Open-page scheduler identifies loads from threads with lower memory-level parallelism (MLP) as more critical [36], somewhat similar to the TCM approach. The memory request prioritization buffer (MRPB) was proposed to perform request reordering and cache bypassing for GPU memory requests; the requests are reorganized in a warp-aware manner, and prioritization is performed such that requests from certain warps are prioritized over those from other warps [32].

Since our work belongs to the category of fine-grained criteria, we are more interested in evaluating these metrics. Unlike many of the coarse-grained criteria, which express criticality *across* different threads, the fine-grained criteria typically express the *intra-thread* differences in criticality. Often, we find that the choice of criticality metric can be driven by the desired optimization, though the choice of criterion is still primarily ad hoc.

Early work by Srinivasan and Lebeck used dependence chain analysis to determine if a load was critical (e.g., a load that has a dependent mispredicted branch instruction, or a load with a dependent load that will miss in the L1 cache) [72, 71]. They used a predicted criticality both to alter parts of the cache into a critical-load-only victim cache, and to guide prefetching.

Jaleel et al. proposed the Re-Reference Interval Prediction (RRIP) policy for cache replacement [30]. Typically, cache line replacement candidates are chosen based on temporal locality—if a line was reused, it moves to the top of an ordered list, where the bottom list member is the one evicted if an upcoming cache request requires space for a new line. RRIP instead classifies the likelihood that a memory location will be reused, and lines that are more likely to be reused are

considered more critical to preserve, and are therefore moved to the top of the replacement list.

Several cache-oriented optimizations were proposed by Subramaniam et al. using their concept of load criticality [73]. For each load, they maintain a history of how many direct consumer instructions were dependent on a particular dynamic instance of a load instruction. The number of consumers is saved into a PC-indexed table, and is used in conjunction with a confidence counter to predict future criticality. They target several efficiency-related optimizations for the x86 architecture, such as "faking" the performance of a second cache read port by deferring non-critical loads, only performing forwarding from the store queue for critical loads, and having a cache insertion policy that is dependent on load criticality.

Prieto et al. considered several load criticality metrics to manage off-chip bandwidth via memory scheduling [60]. The majority of their studies use the distance of a load from the ROB head as the load criticality metric, where loads closer to the head are considered more critical. This metric was also used as load criticality by İpek et al. when they were comparing features for a reinforcement-learning-based memory scheduler [28]. Prieto et al. also consider, but dismiss, criteria such as instruction cache misses and the outstanding branch count.

While the criteria may differ significantly, one common thread through the fine-grained criticality work is an assumption that criticality exhibits some form of loop recurrent behavior. As a result, all of these criticality predictors (including our CBP predictor) assume that the criticality behavior of a static instruction will be similar across its dynamic instances, and therefore use the PC to index their prediction mechanisms.

5.2 Performing a Systematic Search for Criticality

We first attempt to quantify the performance of using commit blocks for predicting scheduling prioritization as opposed to other ad hoc metrics. A total of 96 different metrics were added to the SESC simulator [63], including our commit block metric (a complete list of these metrics can be found in Appendix A). These metrics were each predicted using a fully-associative table with an unlimited number of entries, to eliminate any effects of aliasing on the evaluation. The values of each entry were not capped, though several of the metrics were naturally bounded as a result of the properties of the metric itself (e.g., the fixed size of various queues within the processor). The tables are indexed by the program counter address of each instruction, and therefore expect some form of recurrent behavior to occur based on the static instruction.

For each metric, twelve different methods of prediction were investigated:[1]

- The value observed for the last dynamic instruction instance, where a greater value indicates higher priority;
- The value observed for the last dynamic instruction instance, where a greater value indicates lower priority;
- The maximum value observed for any prior dynamic instruction instance, where a greater value indicates higher priority;
- The maximum value observed for any prior dynamic instruction instance, where a greater value indicates lower priority;

[1] Inverse priorities (i.e., where a greater value indicates lower priority) were not investigated for the 16-entry moving average and moving sum metrics, as the 128-entry moving average and the total sum metrics did not exhibit a strong correlation to inverse priorities in prior experiments.

- The total sum of all observed values for every prior dynamic instruction instance, where a greater value indicates higher priority;
- The total sum of all observed values for every prior dynamic instruction instance, where a greater value indicates lower priority;
- The number of times a condition was observed, where a greater value indicates higher priority;
- The number of times a condition was observed, where a greater value indicates lower priority;
- The moving average of the value observed for the last 128 dynamic instruction instances, where a greater value indicates higher priority;
- The moving average of the value observed for the last 128 dynamic instruction instances, where a greater value indicates lower priority;
- The moving average of the value observed for the last 16 dynamic instruction instances, where a greater value indicates higher priority; and
- The moving sum of the value observed for the last 16 dynamic instruction instances, where a greater value indicates higher priority.

It is important to note that for some metrics, several of the above prediction methods were similar (e.g., the binary metric of whether a load stalled at commit or not would have the same value for the total sum of all observed values and the number of times the condition were observed); this was almost always restricted to binary versions of the metrics. Including such duplication, a total of 1152 different prediction mechanisms were evaluated. As was done in Section 4.1, the prediction is bundled with the load as it progresses to the memory scheduler upon load issue, and is used as the second-level prioritization after row locality (*CASRAS-Crit*) to maintain hardware simplicity.

In order to capture the first-order effects of metric interaction, we also experiment with using weighted sums to determine a combined prediction. Each criterion was normalized using the average overall value observed in the independent training set runs. For each criterion, the best of the twelve predictor types was used. In order to cull the search space, our experiments used the fifty top criteria found during the independent runs.

5.3 A Trace-Based Analysis of the Commit Block Predictor

One of the difficulties in understanding the specific improvements delivered by memory scheduling optimizations is the dynamic behavior. Recall that the transaction queue inside each memory controller contains a series of pending DRAM requests. If different requests are selected in the queue by two separate scheduling algorithms, future scheduler behavior will diverge, as the remaining entries in the queue will be different. Unfortunately, this makes side-by-side simulation of multiple scheduling algorithms infeasible. To make matters worse, synchronization behavior within parallel applications can even create divergence in the program execution behavior, making it difficult to perform a post-simulation analysis.

We work around these difficulties by performing a robust trace-based analysis of the memory scheduler behavior on a subset of our applications. For each load instruction, we annotate information about the scheduling decisions that took place for the requests related to that instruction only if it generated a primary miss to DRAM. Load instructions generating secondary misses include a pointer to the instruction generating a primary miss. We also annotate general

information about instruction behavior to all of the instructions. The annotated information is written to a trace file *at instruction commit*, in order to guarantee that the instructions being compared are identical. (The PC is also recorded, and checked by the trace reader to detect when synchronization-related trace divergence occurs across different simulation runs.)

Due to the poor scalability of trace files, the traces currently only record a portion of program execution. For each of the parallel applications analyzed, each thread is fast forwarded by 200 million instructions to skip initialization. This fast forwarding process does not perform DRAM scheduling, so all simulations will be advanced to the same exact starting point. From here, traces will be recorded until each of the eight threads has executed at least 10 million instructions. Within our trace reader, we perform our analysis on a per-thread basis, and perform a side-by-side comparison of each individual instruction across two different simulations (one for a baseline FR-FCFS scheduler, and another for our *Binary* CBP-based scheduler). The side-by-side comparison continues until the trace reader detects control flow divergence, or when one of the trace files runs out of instructions. While several threads were able to complete their analysis on the complete trace, many others experienced divergence. We note, however, that even in the worst case, at least 3 million instructions were evaluated for each thread.

5.4 Evaluation

We first perform the systematic search described in Section 5.2, in order to identify where our CBP-based criticality metric falls with respect to other proposed

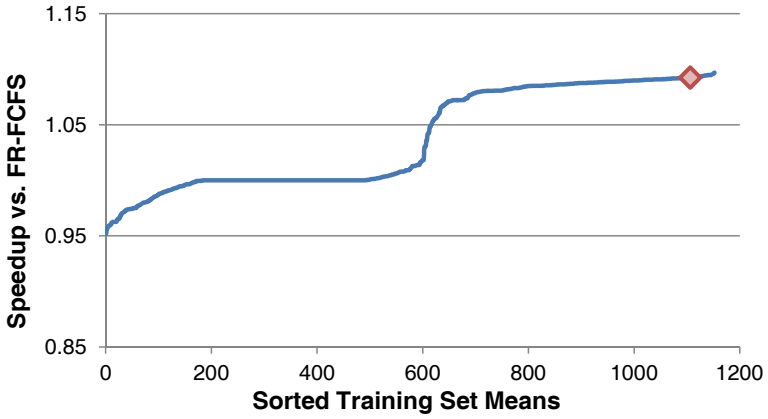

Figure 5.1: Mean speedup for a sweep of different criticality metrics, with means sorted on the x-axis in increasing order of speedup. The diamond indicates the *MaxStallTime* predictor.

criteria in the context of memory scheduling. Afterwards, we examine the results of our trace-based analysis for a subset of parallel applications, in order to get more direct confirmation of the effectiveness of the CBP. In both of these studies, we simulate fully-associative criticality predictors that can hold an unlimited number of entries, in order to understand the effectiveness of the predictor concepts.

5.4.1 Systematic Metric Search

To avoid overfitting, only a portion of our benchmarks were used as a training set for our systematic search. For these experiments, three of our parallel benchmarks (*fft*, *mg*, and *swim*) were chosen at random.

These training set results are shown in Figure 5.1, sorted by the speedup obtained on the training set. As can be seen, the results generally split into three different groups: Metrics that performed worse than the baseline FR-

FCFS scheduler, metrics that had no material effect on scheduling performance, and metrics that performed significantly better than FR-FCFS. The *MaxStallTime* CBP metric, shown as a diamond on the graph, performs amongst the best of our metrics, confirming our prior intuition that tracking loads that block at commit time provides a good metric-based indicator of load criticality to the memory scheduler. None of the metrics exceeded a speedup of 10% (with *MaxStallTime* achieving a training set speedup of 9.3%).

We then performed a search using the weighted sum of two normalized criteria, in order to see if we could exploit any synergistic effects. As mentioned in Section 5.2, we perform an elitist selection of the 50 best criteria in an attempt to control the search space. For each pair of criteria, a combined prediction was obtained by first weighting each normalized prediction equally. Subsequently, uneven weightings were tested, by weighting each of the metrics as 75% of the overall metric in two independent tests (for a total of three combinations per pair). Despite testing this range of weights, we found that even using such combined interactions, the performance of our scheduler never exceeded a speedup of 10%. If multi-factor synergy can be exploited for a combined criticality criterion, we expect that a more sophisticated interaction must be captured between the two criteria in order for the combination to be successful.

5.4.2 The Effectiveness of Targeting Commit Blocks

We select three applications to study using our trace-based analysis: *art*, due to its relatively high mean memory latency and its unusual behavior as noted in Section 4.3.3; *cg*, since it does not benefit at all from *Binary* criticality but does benefit significantly from *MaxStallTime* ranking; and *swim*, as it exhibits aver-

Table 5.1: Changes in LLC load miss behavior due to *Binary* CBP-based scheduling, as observed by trace-based analysis. The loads summarized in the table were actually observed to stall, as opposed to only being predicted to stall.

Observed Statistic	***art***	***cg***	***swim***
Pct. Misses Stalling Commit with FR-FCFS	21.6%	38.4%	69.0%
Pct. Misses Stalling Commit That No Longer Stall	60.2%	22.6%	17.7%
Pct. Misses That Were Not Stalling Commit but Now Do	17.0%	14.1%	44.0%
Decrease in Avg. Commit Stall Time (cycles)	189.0	15.9	42.8
Avg. Memory Latency for Misses Stalling Commit (cycles)	457.0	240.6	292.7
Avg. Mem. Latency Decrease for Misses Stalling Commit	186.3	11.4	36.6
Avg. # of Scheduling Decisions Stalling Misses Don't Win	51.0	6.8	31.0
Decrease in Decisions That Stalling Misses Don't Win	30.8	1.0	10.1

age memory latency behavior and benefits from both *Binary* and *MaxStallTime* criticality.

Table 5.1 summarizes how the Commit Block Predictor impacts the behavior of last level cache (LLC) misses (i.e., DRAM requests). Notably, CBP-based scheduling seems to adapt well to both *art* and *swim*, which exhibit significantly different characteristics. For example, the majority of DRAM requests from *swim* stall at commit, whereas only one in every five requests does so for *art*. In contrast, the longer memory latencies of *art* are cut down much more significantly, with a 41% drop in memory latency blocking loads, whereas *swim* experiences only a 15% reduction, albeit over a much larger percentage of its DRAM requests. (As we can see for all applications, the decrease in memory latency for blocking loads translates almost directly to a decrease in the average commit stall duration.) Unlike *art* and *swim*, *cg* effectively shows no benefit with

Binary CBP-based scheduling. For the phase analyzed, there is a slightly higher speedup (approximately 3%), which is a result of the minor stall time decrease noted in Table 5.1.

For further insight, we look inside the memory scheduler itself. During every DRAM clock cycle that a memory request in the transaction does not win the scheduling decision (i.e., another request has higher priority), we increment a counter. Ideally, for loads blocking at the head of the ROB, we want our scheduler to bump their requests higher up in the prioritization. Therefore, we measure the change in the number of scheduling decisions lost. (Note that this statistic also includes the dynamic effects that result from processing DRAM requests in a different order.) For both *art* and *swim*, critical DRAM requests are losing a large number of scheduling decisions, and our prioritization improves upon that significantly (approximately commensurate with the percent decrease in commit stall time). However, the critical requests for *ch* are not losing often, suggesting that there is only limited contention within the memory system.

Using the trace-based analysis, we also compare the performance of the Commit Block Predictor to observed blocking behavior. For all three benchmarks, virtually all of the blocking loads are predicted by the CBP. Unfortunately, we find that there are an extremely large number of false positives (i.e., loads that do not block at the head of the ROB but are predicted to do so)—*art* incorrectly marks 62% of non-blocking loads, *swim* marks 92%, and *cg* marks almost 100% of them as critical. The extremely high rate of false positives for *cg* offer some insight into why *MaxStallTime* can deliver speedups when the *Binary* predictor cannot. With nearly every DRAM request flagged as critical, there is effectively no net gain from a binary classification approach, since only a triv-

ial amount of loads will be marked as non-critical. In contrast, *MaxStallTime* provides a workaround by ranking amongst the critical requests. This differentiation allows the criticality-aware scheduler to have some effect.

Recall that these results used CBP tables with an unlimited number of entries, so there is no aliasing occurring. What the high false positive rate suggests is that our prior assumption of load criticality is incorrect. Criticality behavior is in fact *not the same* for all dynamic instances of a static instruction, and we must find a better method of tracking these histories for load criticality predictions. We propose one future approach to improved prediction indexing in Section 9.2.

CHAPTER 6

SELECTIVE HARDWARE PREFETCHING USING CRITICALITY

Prefetching has allowed programs to realize significant performance benefits by identifying inter-block spatial locality and memory access pattern regularity. Hardware prefetchers are now commonplace in commodity processors, as discussed in Section 2.1.3. Most of these are variations of prefetcher designs from the 1990s, when single-threaded applications and single-core machines were prevalent [19, 34, 35, 58, 61]. For a while now, the consensus has been that stream prefetchers perform the best of all the general designs. In fact, several contemporary commercial processors include some form of stream prefetcher [25, 33, 74]. As such, recent work has predominantly studied how to further improve the performance of the stream prefetcher [12, 33, 42, 70, 79].

One area of study for improving hardware prefetcher performance has been prefetcher control [12, 33, 46, 70]. Prefetching represents a balance between potentially significant gains when the prefetches are correct, and harmful cache pollution and unnecessary inter-thread interference when the prefetches are incorrect. When unchecked, this interference can have a significant negative impact on overall performance [12, 61]. Unfortunately, for entry-based designs such as the stream prefetcher, it is often not feasible to perform this control at anything other than a global granularity, as we shall see. For typical prefetch control mechanisms, this results in a rather indiscriminate approach to deciding when to ease off on prefetching.

The concept of load criticality offers a more elegant solution. Indeed, the decision on whether a prefetcher should issue prefetch requests is in many senses a risk analysis. For loads requested by the processor (*demand* requests) that miss

in the L1 cache, yet still arrive early enough to not fall along the critical path of program execution, prefetching those loads earlier will not have any material impact on the program performance, but will still expose all of the potential risks from pollution and contention. In contrast, using ranking to identify the degree of criticality, loads that are more critical stand to benefit further from prefetching, as the potential savings to program execution time are that much greater.

Focused prefetching using load criticality has been studied in the past, in the context of single-thread workloads and a global history based prefetcher [46]. However, today's multicore-based computers are very different. Unlike single-core microprocessors, multicore chips can run multiple programs in parallel (which compete for cache space and memory bandwidth), as well as parallel applications (whose threads share memory regions). The memory landscape itself has changed as well, with DRAM providing greater bandwidth, though at longer latencies than before. As we shall see, simply applying focused prefetching to traditional prefetchers has little effect.

The work in this chapter revisits several assumptions on prefetcher design that have to date been taken for granted. We show that *selective prefetch filtering* based on the notion of load criticality can indeed be effective once these assumptions have been correct, and that these improved prefetchers can outperform the best stream prefetcher, considered to be state-of-the-art, by a mean speedup of 7.8% for parallel applications, and a mean weighted speedup of 13.8%. Furthermore, we analyze the memory behavior of parallel applications, and find that highly-aggressive sequential prefetchers, previously dismissed as unhelpful [61], can in fact deliver significant performance improvements for these

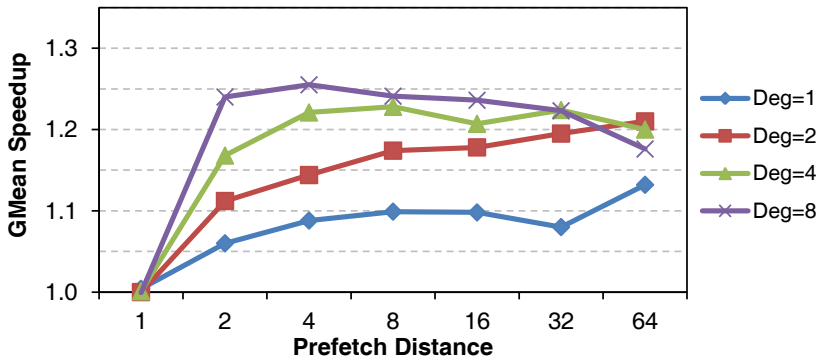

Figure 6.1: Parameter sweep over distance and degree for 64-entry stream prefetchers, relative to no prefetcher.

workloads over the best stream prefetcher, achieving a 10.9% mean speedup when used in tandem with our criticality-based prefetch filtering.

6.1 Establishing a Baseline

In order to establish a baseline prefetcher for our work, we initially implemented four widely known prefetchers: The next line prefetcher, the stride prefetcher, the Markov chain prefetcher, and the stream prefetcher. For these studies, the prefetcher was attached to the last level cache (LLC) as shown in Figure 2.2(a), with the exception of the stream prefetcher, which was attached as shown in Figure 2.2(b) for improved performance [70, 74]. Of these four prefetchers, we found the stream prefetcher to significantly outperform the other prefetcher variants, and as a result use that as our baseline. Details about our experimental methodology can be found in Section 6.6.

To find the best configuration for the stream prefetcher, we sweep over the three tunable parameters—the number of prefetch entries, the prefetch distance, and the prefetch degree—as shown in Figure 2.3. Figure 6.1 shows the mean

performance across our parallel benchmarks when sweeping over distance and degree for a 64-entry stream prefetcher. Sweeping over the number of prefetcher entries revealed that up to 64 entries, the performance did increase with the number of available entries, but that additional entries beyond that (up to a 256-entry prefetcher) showed no effective impact, suggesting that the extra entries were of little benefit.

We observe from Figure 6.1 that, for our architecture configuration and applications, overly aggressive prefetchers offer diminishing returns as the distance grows. With large monitoring windows, the prefetcher will have a greater propensity to fetch data further away from the current request, reducing the likelihood that the data will be used. The best of the prefetcher configurations, with a distance of 4 and a degree of 8, achieves a mean speedup of 25.1% over not having a prefetcher.

Experiments were also conducted to determine the effect of prefetcher entries shared amongst all cores versus assigning the stream entries privately by core (akin to having per-core prefetchers). In all of our studies, we found that parallel applications did not benefit from assigning entries to an individual core. As we shall see in Section 6.4.1, this is because such partitioning myopically ignores the synergy that can be had across multiple threads of a parallel application, and reduces the opportunities for successful prefetching. In addition, maintaining a shared prefetcher in the last level cache avoids unnecessarily polluting the much smaller private caches (where per-core prefetchers often reside) with memory that is not yet, and may never be, needed. Without per-core prefetchers, we can also avoid the cache coherence issues discussed by Enright Jerger et al. [13].

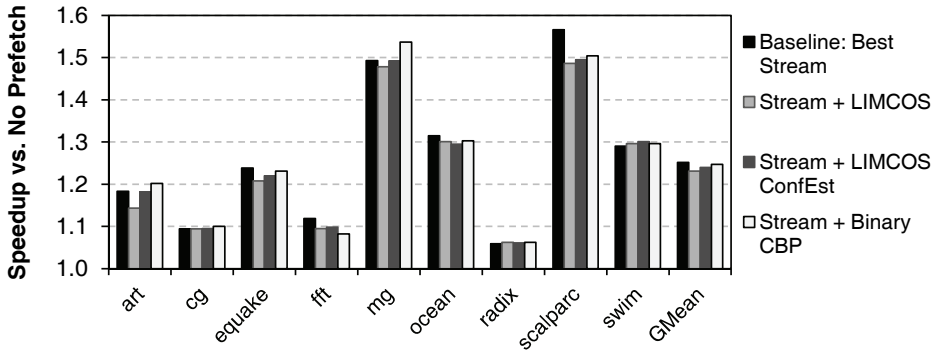

Figure 6.2: Speedup *over not having a prefetcher* from applying for both LIMCOS filtering and our Commit Block Predictor based filtering to the best existing stream prefetcher.

6.2 Prior Work in Focused Prefetcing

In the context of prefetching for single-thread workloads, Manikantan and Govindarajan proposed a processor-based predictor that measured loads which stall at the head of the reorder buffer [46], similar to our Commit Block Predictor (CBP) from Chapter 3. Unlike our predictor, they propose hardware to classify only those loads incurring the majority of commit stalls (LIMCOS), as they believe that a very strict subset of loads will result in improved performance for the prefetchers that they examined. As their studies focus primarily on single-threaded workloads, their insights ignore any cache interactions resulting from parallel applications.

The LIMCOS classifier attempts to be more judicious in the loads that it permits to be prefetched. Instead of classifying all load instructions based on criticality, it attempts to identify only those static instructions that stall commit for the most amount of time. This requires greater predictor sophistication at the circuit level, as they require 32-entry tagged fully-associative tables inside each

core with a replacement policy. A separate counter is also needed to track the total number of cycles that load instructions within the table have stalled for. They also propose an alternative mechanism, which we call *LIMCOS ConfEst*, that replaces the counter with a confidence estimator using saturating counters, and uses 8-way associativity for their 32-entry tables. The tagging and associativity allow the LIMCOS classifier to maintain better precision when determining which loads are the largest stall contributors [46].

In an attempt to observe the benefits of LIMCOS in the parallel application domain, we adapt the LIMCOS classifiers for our prefetchers. Previously, the classifier filtered prefetch requests generated by the prefetcher, meaning that training was performed on both critical and non-critical requests. In order to better focus the stream prefetcher, we instead have the LIMCOS classifier filter out addresses being sent to the prefetcher by non-critical loads, reducing unnecessary training and entry allocation. As we see in Figure 6.2, simply adding the LIMCOS classifiers to the stream prefetcher have no material effect on the overall prefetcher performance. We also use our *Binary* CBP from Chapter 3 for comparison, and also see that it has no material effect.

As we will see, implicit and ineffective filtering is hampering any effect that criticality-based filters can have. We propose modifications that, when used in conjunction with criticality-based filters, offer us much greater performance. As there is no material difference, our results will initially present criticality filtering using our CBP mechanism, as it is a more general-purpose design that can also be used for memory scheduling. We will revisit the effectiveness of LIMCOS classifiers in the context of our changes in Section 6.7.6.

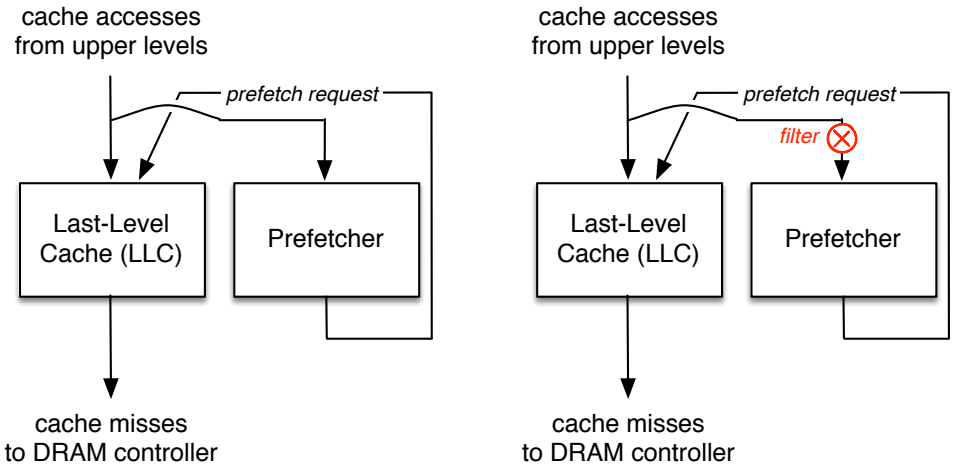

Figure 6.3: Proposed cache attachment approaches for more aggressive prefetching: (a) Prefetcher reads sequence of all cache accesses (*Access* attachment); (b) *Access*-attached prefetcher only reads sequence of all *critical* cache accesses, where non-critical accesses are screened out by a filter.

6.3 It's All About Location: Prefetcher Attachment

The traditional attachment of a prefetcher has in effect used the cache as an implicit filter. Ideally, the prefetcher should observe a complete sequence of memory accesses. Most prefetchers, however, observe the sequence of misses leaving the cache that they are attached to, as shown in Figure 2.2(a). For capacity and conflict misses, the miss sequence does not longer represents a fully-intact stream of requests, as typically only some of the desired lines have been evicted from the cache (depending on the program's memory access pattern).

While this can help control the bandwidth utilization of the prefetcher, this type of filtering is not at all designed to cater to the training behavior that a prefetcher ideally desires. As discussed in Section 2.1.3, advanced prefetchers are dependent on regularity in order to identify patterns within the request access

stream—the more intact this observed request stream is, the easier it will be for the prefetcher to correctly learn the expected pattern. From the perspective of the prefetcher, the filtering taking place by the cache is rather haphazard, and in effect undermines the principle assumption of prefetcher designs.

In order to address this problem, stream prefetchers are instead attached as shown in Figure 2.2(b), which we refer to as *split* attachment [70, 74]. In this layout, the stream prefetcher will only allocate entries and train the direction of prefetch using cache misses. However, once an entry has been trained, stream prefetchers try to increase the frequency of prefetching by tracking hits to the monitoring window using *all cache accesses* (hits and misses). As can be expected, the access sequence will represent a much more complete request stream than the miss sequence, allowing the stream prefetcher to more quickly anticipate the need for subsequent data. However, this does not completely solve the problem, as allocation and training are still muddled by the implicit filtering. A more intact training stream will increase the chances of finding a series of requests that fall within the same monitoring window, which can significantly reduce training time and exploit prefetching opportunities earlier.

We propose to solve this by also moving allocation and training observation up to all cache accesses as well, which is shown in Figure 6.3(a) (we refer to this as *Access* attachment). As expected, simply increasing the number of requests monitored by the prefetcher will greatly increase its activity, and if left unchecked will still fail to deliver performance improvements, due to the much greater potential for cache pollution and contention—Section 6.7.1 shows that an *Access*-attached prefetcher shows no real gain on its own over traditionally-attached prefetchers.

This presents an opportunity for us to replace the haphazard implicit filtering of the cache with our criticality-based filtering (Figure 6.3(b)), which will act as a risk analysis for individual addresses being passed to the prefetcher. Now that it can operate unhindered (which was the main drawback when these filters were used in Section 6.2), our "smarter" selectiveness aims to hide from the prefetcher loads whose acceleration would have a minimal impact on the program. Another advantage is that the CBP-based filters perform its predictions based on the static instruction behavior, and once a static instruction is predicted as critical, it will stay critical until the CBP entry is reset. Since address request streams are typically generated by the same static instruction in a loop, the filter will implicitly classify almost all of the requests within a stream with the same criticality ranking, thereby ensuring that if the prefetcher sees one of the requests, it will receive all of the intact sequence. We will take a more detailed look at filtering in Section 6.5.2.

6.4 Exploiting Parallel Application Behavior for Prefetching

One subtle lesson from eliminating the implicit filtering mechanism is that our criticality-based filters provide a more sensible way of selectively controlling the aggressiveness of a prefetcher. This means that while the judiciousness of the stream prefetcher can often help to disambiguate troublesome and incomplete patterns, our selective filtering may obviate the need for such cautiousness. In this light, we examine the common memory allocation patterns of parallel applications, and revisit the potential of the highly-aggressive sequential prefetcher, which was previously dismissed as too aggressive to be of use [61].

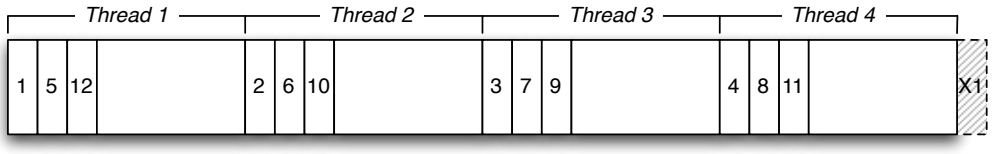

Figure 6.4: Example of prefetching for a contiguous data segment statically partitioned amongst four threads of a parallel program. The numbers labeling each cache block represent one possible ordering of the accesses, as observed by the shared cache. Access X1 refers to the block that would be incorrectly prefetched based on the observed strides of accesses 1–4.

6.4.1 Memory Layout for Parallel Applications

A common approach for parallelizing a program is to distribute the available work amongst several threads. To do this, large segments of regular (and usually contiguous) data are assigned to each thread, which must then perform some computation and perhaps synchronize with other cores. These approaches cover the loop constructs available for the `parallel` pragma in OpenMP [57].

Take, for example, a large, contiguous segment of data, as shown in Figure 6.4. For t threads, the simple way to partition this data is to split it into t equal pieces, and to assign each thread to each piece. (In OpenMP, this is referred to as `static` partitioning, and has the lowest overhead as well as the greatest probability for load imbalance.) Assuming the threads start in order, each thread will read the first cache block in its assigned piece. Figure 6.4 shows an example of this with four threads, as seen in accesses 1–4. At this point, a simple stride-based prefetcher would observe that a predictable stride has emerged and start prefetching. Unfortunately, this stride is incorrect, as the prefetcher will attempt to retrieve the first block past the end of our shared data segment (access X1 in the figure). In reality, we wanted to prefetch the second

block in each thread's data piece (accesses 5–8). Unfortunately, even though this behavior may seem to exhibit a regular pattern, this too becomes problematic, as without fine-grained synchronization, threads will exhibit nondeterministic interleaving of instructions, as shown for accesses 9–12.

The "smarter" stream prefetcher was designed to avoid such fixed-stride behavior. However, if the monitoring window is too large, the stream prefetcher too will prefetch addresses such as X1, which are outside of the shared data segment. Unfortunately, restrictive windows present an opposing problem—small windows require longer training times before initiating prefetch. For example, in Figure 6.4, if the window is restricted to the size of the data piece assigned to Thread 1, the stream prefetcher will allocate an entry upon observing access 1, but will not start issuing prefetches until after access 12 is made (since it requires two subsequent cache misses to train on). In general, regardless of monitoring window size, training can often unnecessarily work against the contiguous memory layout offered by the parallel programming model.

Striping the data can also lead to problems with regularity. Both static striping (assigning stripes to threads at the beginning of the parallel section; `static` partitioning with a non-unit chunk size in OpenMP) and dynamic striping (having threads pull the next stripe from a centralized work queue when they finish their current work; `dynamic` partitioning in OpenMP) exhibit significant pattern irregularity due to thread nondeterminism. What this means is that while per-core prefetchers may work well for multiprogrammed workloads, they result in missed opportunities for parallel applications. For striped data, the accesses may be spread out far enough that they do not fall within the same monitoring window, and will fail to trigger a prefetch.

As an aside, parallel applications stand to benefit greatly from prefetch request synergy across threads, especially in the case of striped data. If we were to rely solely on per-core prefetchers, these opportunities for synergy would be missed entirely, resulting in unnecessarily prolonged delays before successful prefetching takes place. In a pathological case of striped data, the prefetcher may not be of any effect at all. Using shared prefetchers in the shared last level cache allows us to benefit the most from the contiguous memory allocation. We expect that this is the reason for the lackluster performance observed when conducting our per-core experiments.

6.4.2 Simpler Is Better: Sequential Prefetching

Since we see that our parallel applications will likely have large contiguous blocks of memory, we revisit previously-held assumptions on sequential prefetcher performance. For every observed request, a sequential prefetcher simply issues requests for the next N cache lines (this is discussed in more detail in Section 2.1.3). Unlike stream or stride prefetchers, there is no training, and therefore no need for a tracking table. Traditionally, processors have only implemented prefetchers where $N = 1$, as Przybylski showed that, for sequential workloads, the negative impact of pollution outweighed the performance benefits of useful prefetching for $N > 1$ [61]. While this still holds true for multiprogrammed workloads, we find this assumption to be incorrect for parallel applications, due to their frequent contiguous memory segments. Even then, a very large value of N can still prefetch many more lines than necessary, and the subsequent pollution can undermine potential speedups.

Table 6.1: FDP configurations tested.

Aggressiveness	**FDP–Original**	**FDP–*Access***	**FDP–Sequential**
Very Conservative	dist = 4, degr = 1	dist = 1, degr = 1	$N = 1$
Conservative	dist = 8, degr = 1	dist = 2, degr = 1	$N = 2$
Middle-of-the-Road	dist = 16, degr = 2	dist = 4, degr = 1	$N = 4$
Aggressive	dist = 32, degr = 4	dist = 8, degr = 1	$N = 8$
Very Aggressive	dist = 64, degr = 4	dist = 16, degr = 1	$N = 16$

6.5 You Are What You Eat: Throttling vs. Filtering

As we increase the aggressiveness of our prefetchers, we must be careful about the potential negative impacts of this aggression. While we expect that our prefetchers will retrieve a greater number of useful requests, we may also issue prefetch requests for unnecessary cache lines. Ideally, we would like a mechanism that avoids issuing as many unnecessary requests as possible without negatively impacting the useful prefetches. Typically, prior work has used accuracy as the metric of choice for prefetcher throttling, which we examine in Section 6.5.1. However, as we will discuss in Section 6.5.2, the use of accuracy can be misleading, hence we turn to load criticality in an attempt to provide a more sophisticated throttling decision.

6.5.1 Accuracy-Based Prefetch Throttling

Accuracy has often been a metric of choice for determining the success of a prefetcher, and has been used to relax the aggressiveness of the prefetching mechanism during phases when it does not issue a high number of useful re-

Table 6.2: Best threshold parameters found for FDP.

A_{high}	A_{low}	$T_{lateness}$	$T_{pollution}$	P_{high}	P_{low}
0.7	0.6	0.005	0.005	0.009	0.005

quests [12, 42, 70]. In order to measure accuracy at runtime, each cache line is extended by a single bit, which is set when a prefetch request that has yet to be demanded is inserted into the cache. On a successful subsequent read of a cache line, if this prefetch bit is already set, the cache will clear the bit and notify the prefetcher. Internally, the prefetcher will increment a counter that tracks the number of prefetches that were used. This can be compared to a counter that tracks the total number of prefetches issued to determine the accuracy. A static threshold is typically used to determine if the observed accuracy is adequate, and highly-inaccurate prefetchers are throttled or ignored completely.

As we are using a single prefetcher in the last level cache, we focus on feedback-directed prefetching (FDP), which extends upon accuracy to provide a sophisticated throttling mechanism [70]. FDP uses a stream prefetcher to move between one of five aggression states—very conservative, conservative, middle-of-the-road, aggressive, and very aggressive. Every interval, a decision tree is used to determine if the prefetcher should move to a more conservative state (e.g., if the accuracy falls below a certain threshold and the pollution rate is above a certain level) or to a more aggressive state (e.g., if accuracy is high and if the average lateness is above a certain threshold). The cache is augmented with additional hardware to track lateness and expected pollution. In the original work, the authors show that for select single-thread applications, they can outperform their best-performing static stream prefetcher.

When implementing FDP, we swept across all of the thresholds to find the best values for our architecture, which are shown in Table 6.2.[1] In addition to running the prefetchers described in the original work, we also experiment with using two different sets of prefetchers, selecting parameters that we found performed best for our setup. These prefetcher configurations are listed in Table 6.1. We did not implement the multicore scheme proposed by Ebrahimi et al. [12] because it simply extends FDP to coordinate private per-core prefetchers—a mismatch to parallel applications (Section 6.4.1).

6.5.2 Criticality-Based Prefetch Filtering

While accuracy is typically used as a prefetcher evaluation metric, it fails to completely represent the success of the prefetcher. For example, inside the processor, there may be several loads that have a lot of *slack* (i.e., the load returns to the processor well before it is consumed). If these loads have a lot of slack, and this slack is loop recurrent, then prefetching for that load will have little material impact on processor performance. If we prefetch for such a high-slack load, we run the risk of incorrectly prefetching lines that may never be used or may pollute the cache by evicting useful lines, all without any potential for improvement in return.

Accuracy is unable to provide us with such control. Accuracy is typically considered for an entire prefetcher, and not per request. Tracking accuracy per request would require exorbitant overhead, as we would need to annotate each cache line with the source of the prefetch (e.g., a stream entry ID). Furthermore,

[1] This may result in an FDP configuration that is overfitted to our setup, and thus potentially optimistic performance-wise. Because FDP is a competing mechanism to our solution, this works for FDP and against us in our evaluation.

even if per-request accuracy were feasible, it would be slow to react, as the accuracy would only be computed every interval, thus reducing the impact that filtering can have.

We instead turn to the notion of load criticality for fine-grained prefetch filtering, focusing on our CBP mechanisms. We have shown in Section 2.2.2 that long-latency loads stalling at commit, at the head of the reorder buffer (ROB), contribute significantly to the runtime of an application, as they prevent resources from being freed up inside the processor. It follows that these loads blocking commit are likely the ones that, when prefetched for, have the greatest potential to increase program performance. Typically, prefetch control mechanisms require the cache to be annotated with some data (e.g., accuracy bit, timestamps for lateness). In contrast, load criticality leaves the cache structure untouched, simply relying on minimally invasive CBP tables to store all their information.

We focus specifically on two of the CBP ranking mechanisms: *Binary* ranking, which just marks if a load previously blocked at commit, and *MaxStallTime*, which records the maximum observed stall time for a load. With *Binary* ranking, we simply restrict prefetching to only those loads that are predicted to stall at commit, ignoring the others. For *MaxStallTime*, we employ the ranking in one of two ways. One configuration, *MvAvg*, uses a global 128-entry moving average buffer within the prefetcher; we prefetch for a load only if its criticality exceeds the average. The second approach, *Thres*, defines a static threshold, where prefetching only takes place for loads predicted to have a criticality greater than the threshold value. In a sense, these two options allow us to explore the importance to the prefetcher of relative criticality in comparison to global criticality.

We also experimented with the other ranking metrics proposed in Section 3.2, but found no benefit over the selected rankings. We also studied the load criticality ranking proposed by Subramaniam et al. [73], but found that despite tuning the thresholds, it also yielded no benefits. Srinivasan et al. previously explored the use of criticality to screen prefetch requests, but their criticality detection mechanism is very complex, and their results show meager performance gains over no prefetcher [71].

6.6 Experimental Methodology

We have modified the SESC simulator [63] to simulate an eight-core CMP (Table 6.3) with a DDR-2133 memory subsystem, which faithfully models all DRAM timing latencies and power [48] for a Micron MT41J128M8 device [49], as shown in Tables 4.2 and 6.4.

Table 4.3 shows the parallel applications we used, which cover a mix of application domains. All of the parallel sections of these applications were simulated. Our multiprogrammed workloads consist of bundles of SPEC 2000 and NAS benchmarks using reference input sets, as shown in Table 4.4. These workloads were fast-forwarded for one billion instructions, and the bundle was executed until the workloads executed at least 500 million instructions each. As these workloads are quad core, the number of cache banks and stream prefetcher entries were cut in half, and the memory was reduced to two channels.

For our simulated prefetchers, we assume that all prefetch requests must access the L2 cache before being sent to DRAM, and must contend for the bank controller with requests coming from the L1 caches. Before accessing the cache,

Table 6.3: Parameters for simulated architecture.

Frequency	4.3 GHz
Number of Cores	8
Fetch/Issue/Commit Width	4 / 4 / 4
Int/FP/Ld/St/Br Units	2 / 2 / 2 / 2 / 2
Int/FP Multipliers	1 / 1
Int/FP Issue Queue Size	32 / 32 entries
ROB (Reorder Buffer) Entries	128
Int/FP Registers	160 / 160
Ld/St Queue Entries	32 / 32
Max. Unresolved Branches	24
Branch Misprediction Penalty	9 cycles min.
Branch Predictor	Alpha 21264 (tournament)
RAS Entries	32
BTB Size	512 entries, direct-mapped
iL1/dL1 Cache	32 kB each, 32 B block, direct-mapped / 4-way
iL1/dL1 Round-Trip Latency	2 / 3 cycles (uncontended)
iL1/dL1 Ports	1 / 2
iL1/dL1 MSHR Entries	16 / 16
Memory Disambiguation	Perfect
Coherence Protocol	MESI
Consistency Model	Release consistency
Shared L2 Cache	8 banks, 512 kB per bank, 64 B block, 8-way
L2 MSHR Entries	16 per bank
L2 Round-Trip Latency	32 cycles (uncontended)

Table 6.4: Micron DDR3-2133 DRAM energy model [49].

$\mathbf{V_{DD}}$	1.5 V
$\mathbf{I_{DD}0}$	75 mA
$\mathbf{I_{DD}2P0}$	12 mA
$\mathbf{I_{DD}2P1}$	40 mA
$\mathbf{I_{DD}2N}$	50 mA
$\mathbf{I_{DD}3P}$	40 mA
$\mathbf{I_{DD}3N}$	55 mA
$\mathbf{I_{DD}4R}$	175 mA
$\mathbf{I_{DD}4W}$	180 mA

these requests are placed in a prefetch request queue, which can drain one request per cycle. For the sake of fairness, the number of MSHR entries remains unchanged in the presence of a prefetcher, despite this increase in contention. The tagless Commit Block Predictor tables are 64 entries, and are reset every 100,000 cycles.

6.7 Evaluation

6.7.1 Stream Prefetcher

Moving the stream prefetcher to monitor all accesses does not affect performance on its own, but it decreases the uncertainty that the prefetcher has about predicting which loads to retrieve next. By simply modifying the stream prefetcher to allocate and train on all accesses (Figure 6.3(a)), we see that the speedup over not having a prefetcher does not change significantly, with the

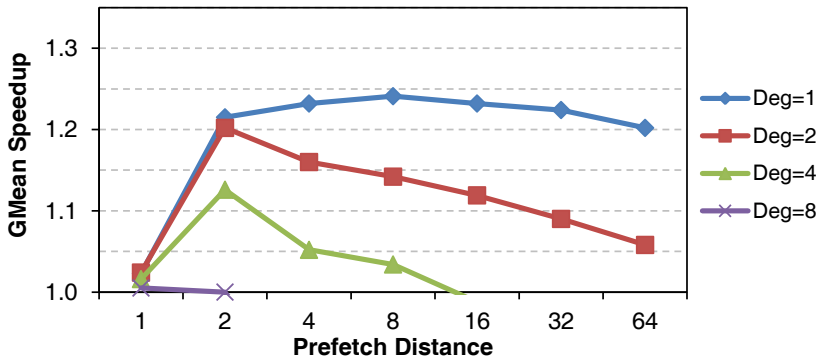

Figure 6.5: Parameter sweep over distance and degree for *Access*-attached 64-entry stream prefetchers, which are allocating and training on all accesses, relative to no prefetcher.

best *Access*-attached prefetcher obtaining a 24.1% mean speedup. Figure 6.5 does show that the trend in degree has changed from Figure 6.1. This is a result of *Access*-attached prefetchers being able to observe a more intact request stream. Traditional stream prefetchers that only monitor misses are not confident about which part of the memory access stream will miss next, as some of the loads within the current data segment may already be in the cache. This can be detrimental with a lower degree—for example, if the next access in sequence is a hit, the prefetcher would never observe the memory operation, thus not triggering a prefetch, even if subsequent loads are not in the cache. A high degree allows the prefetcher to make up for this patchy observation by broadly anticipating several upcoming requests at a time, though if the requests aren't used, this increases the potential for unnecessary pollution. Since access-attached prefetchers see all of the accesses, they do not need to compensate for anything, as each request in the sequence will be observed and can potentially issue a prefetch. In fact, we see that for high degrees, the access-attached prefetchers become so aggressive that the memory contention eventually starts hurting performance.

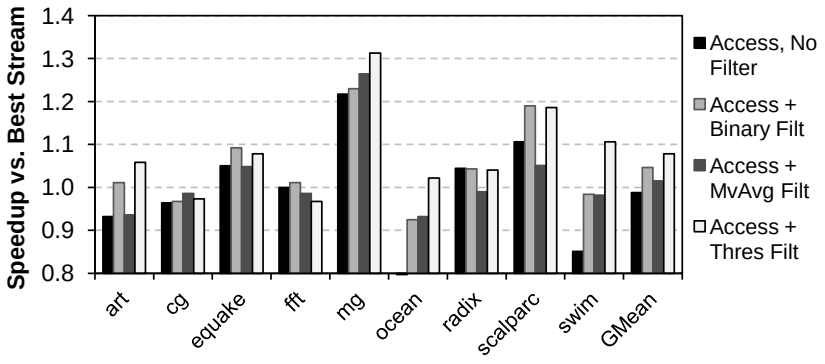

Figure 6.6: Performance of *Access* stream prefetcher with criticality filtering, relative to the best baseline stream prefetcher.

Figure 6.6 shows the effect of adding our criticality-based prefetch filters (Figure 6.3(b), see Section 6.5.2), where non-critical incoming requests do not get passed to the prefetcher. As we mentioned already, without filtering, the access prefetcher performs slightly worse than the baseline stream prefetcher. In particular, *ocean* and *swim* are hurt significantly, undermining other performance improvements. In both cases, the benchmarks are reliant on temporal locality (e.g., *swim* has loops that alternate between writing to a large temporary array and writing to a large permanent array [3]), and the increased pollution chips away at performance. (Note, though, that the performance is never worse than if there were no prefetcher, as shown in Figure 6.7.)

Criticality-based filtering allows an access-attached prefetcher to improve its performance significantly by selectively ignoring memory requests that do not greatly impact program execution time, thus reducing the risk of unnecessary memory contention due to unused prefetches. *Binary* filtering, which simply uses the *Binary* CBP mechanism, shows that even with a very simple notion, we can increase the effectiveness of the *Access* prefetcher such that it outperforms the best baseline stream prefetcher by a mean of 4.6%. The performance of both

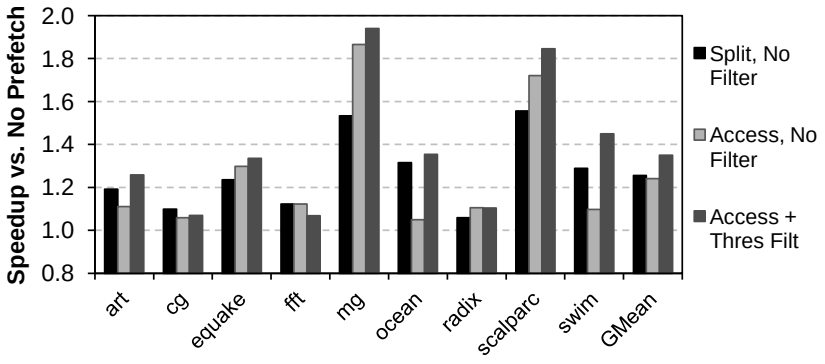

Figure 6.7: Performance of best stream prefetcher configurations (see Table 6.5), *relative to no prefetcher*.

Table 6.5: Best prefetcher settings for parallel applications.

Prefetcher Type	**Parameters**
baseline stream	distance = 4, degree = 8
Access stream	distance = 8, degree = 1
unfiltered sequential	$N = 8$
criticality-filtered sequential	$N = 16$

ocean and *swim* improve significantly, with the *Binary* filter able to restrict much of the previously-harmful pollution.

We turn to more sophisticated mechanisms of filtering, this time using the *MaxStallTime* ranking. As mentioned in Section 6.5.2, one such approach is to use the *MvAvg* filter to prefetch only loads that are perceived as being highly critical. We see that the performance of the moving average buffer is underwhelming, with a mean of 1.5%. Due to the contiguous nature of the memory layout, the stream prefetchers typically have a high accuracy (e.g., the misses generated by the baseline stream prefetcher have a 98% accuracy rate). Since the risk of harmful prefetching is very low, and by definition, the critical loads are the ones hindering program progress, the payoff for prefetching low-criticality

loads is very high, and in effect all the moving average buffer does is unfairly restrict them.

We also explored the use of the *Thres* static threshold filter based on not just criticality, but *slack* as well. In addition to the *MaxStallTime* predictor, we added a predictor that keeps track of how early a load arrives (i.e., how many cycles the load returns before its dynamic instruction instance commits), creating a continuum of how important we believe the load to be. We then used a subset of our parallel applications (*fft*, *mg*, and *swim*) as a training set, picking the fastest-executing programs to avoid bias. We then swept across the threshold (which dictates the minimum *MaxStallTime* a load must have to be forwarded to the prefetcher, or the maximum allowable slack). For our configuration, a minimum criticality threshold of 140 cycles was found to be best. With this threshold, we were able to provide a mean speedup of 7.8% over the best baseline prefetcher (Figure 6.6). We discuss bandwidth and energy implications in Section 6.7.3.

6.7.2 Sequential Prefetcher

Due to the tendency of parallel applications to use contiguous blocks of shared memory, aggressive sequential prefetchers, which were found to be harmful for sequential applications, actually outperform even the best stream prefetchers. As we hypothesized in Section 6.4.2, sequential prefetchers seem pretty well suited to the data layout of parallel applications. We first sweep over the number of subsequent lines prefetched, N (Figure 6.8). Without filtering, we see that the optimal value of N for a miss-attached prefetcher running parallel applications is 8, with a mean speedup of 34.6% over no prefetcher, in contrast to the earlier findings of Przybylski [61] for sequential applications, where perfor-

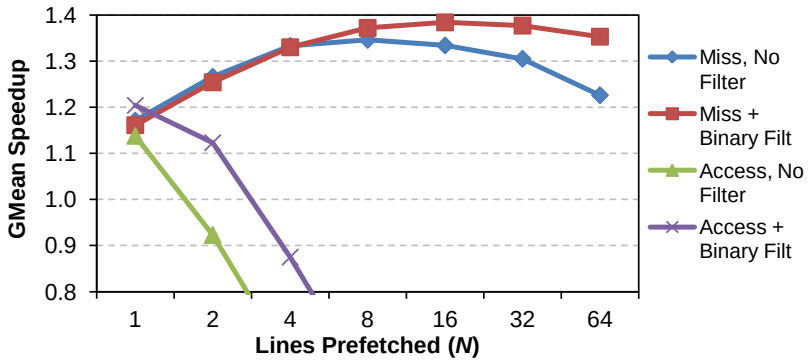

Figure 6.8: Sweep over N for sequential prefetchers. *Speedup relative to no prefetcher*.

mance benefits do not exist for $N > 1$. As we see, though, the curve saturates as early as $N = 4$, which indicates that despite trying to prefetch significantly more lines, the effects of memory contention eventually overrule the additional spatial locality (and that beyond a certain value of N, there is little additional locality to exploit). Unsurprisingly, access-attached sequential prefetchers perform poorly, as they will issue prefetches regardless of the likelihood of spatial locality (unlike stream prefetchers, which by design are somewhat judicious due to training).

Figure 6.9 shows how binary filtering improves upon an aggressive sequential prefetcher. With filtering, the best configuration is for $N = 16$, which when unfiltered provides a geometric mean speedup of 6.2% over the best baseline stream prefetcher. Adding *Binary* criticality filtering boosts this speedup to 10.2%. Interestingly enough, *swim* shows significant speedups here, mainly because our prefetcher is always fetching the very next block, which works much better with the application's array accesses. Again, we try both the *MvAvg* and *Thres* filters, with *Thres* (again at a 140-cycle stall time) performing the best, at a mean speedup of 10.9% over the best baseline stream prefetcher.

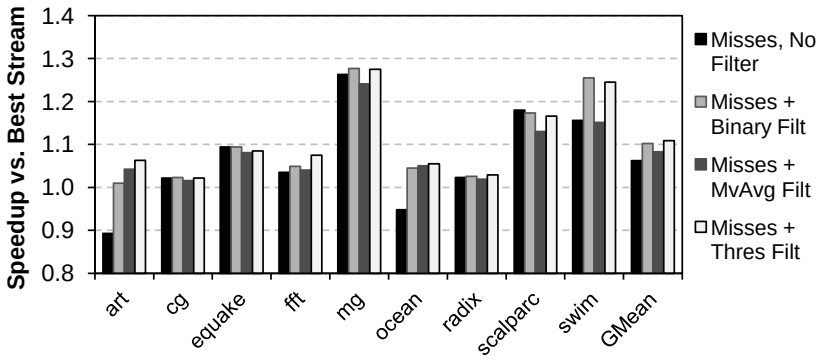

Figure 6.9: Performance of sequential prefetchers with criticality filtering, relative to the best baseline stream prefetcher.

The benefits of filtering can clearly be seen in Figure 6.8. For a prefetcher monitoring misses, as N grows, the binary filter is able to buy back significant performance with greater aggression, effectively pushing the saturation point while reducing the taper-off. Even for the access-attached prefetcher, filtering noticeably reduces the harm otherwise caused by excessive prefetching (while not shown, the same was true for harmful stream prefetcher configurations).

6.7.3 DRAM Bandwidth

Criticality filtering allows aggressive prefetchers to obtain significantly better performance with minimal change in bandwidth. Since aggressive prefetchers are expected to significantly increase memory traffic, we break down their impact across the system. We first focus on DRAM bandwidth, which we expect would take the greatest impact from more prefetching. Figure 6.10 shows the bandwidth normalized to the bandwidth issued by a prefetcher-less program (which represents the minimum required bandwidth). We break down the prefetches into five categories:

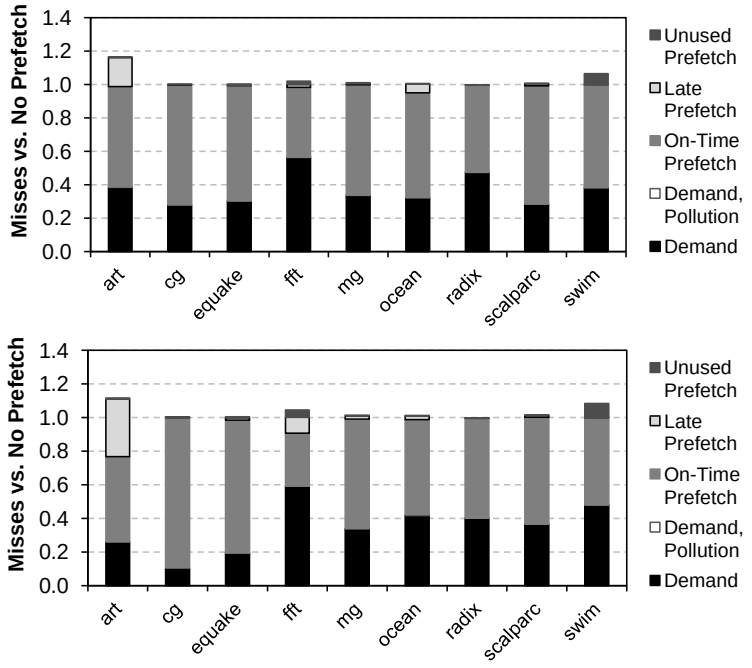

Figure 6.10: Breakdown of DRAM bandwidth utilization (L2 misses) for baseline stream prefetcher (top), *Access* stream with static *Thres* filtering (bottom).

- **Demand**—Requests by the processor for cache lines that would be misses regardless of the presence of a prefetcher;
- **Demand, Pollution**—Requests by the processor for cache lines that would have hit in the cache if it had not been for prefetcher-induced pollution;
- **On-Time Prefetch**—Requests by the prefetcher which were subsequently used by the processor, and arrived before the processor accessed the cache;
- **Late Prefetch**—Requests by the prefetcher which were subsequently used, but were requested by the processor before arriving in the cache, causing a partial miss; and
- **Unused Prefetch**—Requests by the prefetcher that were not used by the processor.

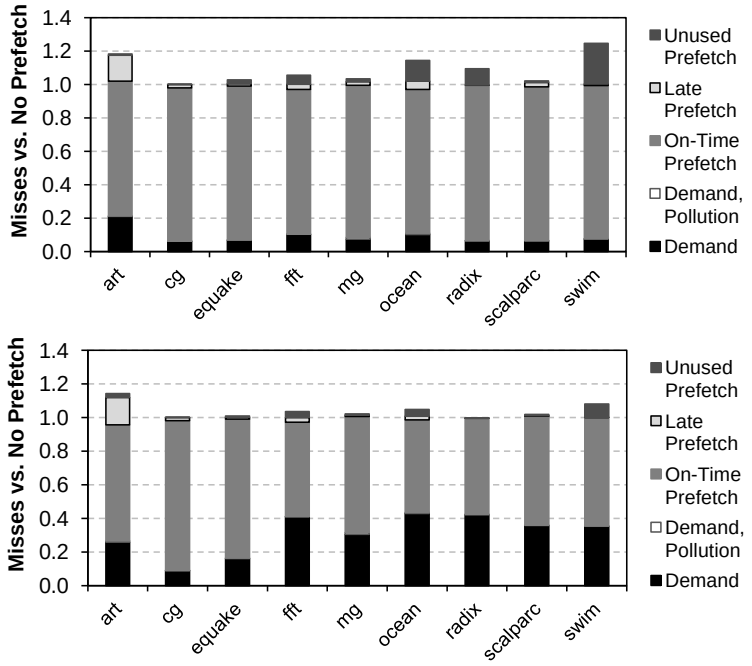

Figure 6.11: Breakdown of DRAM bandwidth utilization for sequential prefetcher, without filtering (top) and with static *Thres* filtering (bottom).

Figure 6.10 shows two effects, both of which contribute to performance. First, the *Access* prefetcher is able to successfully convert some of the demand requests into prefetches, such as for *scalparc*, which translates into large performance improvements (Figure 6.6). Second, the criticality filter is able to minimize the number of extraneous prefetch requests from loads that don't impact program performance, which provides further performance improvements in general due to reduced contention. Note that the sum of all useful requests is sometimes greater than 1. This is not an anomaly, but instead indicates that there were requests that were evicted from the cache due to prefetcher aggressiveness, but that the prefetcher itself successfully predicted the need to retrieve these lines later during execution. Aside from these requests, we see that, unlike for sequential workloads, the number of lines prematurely evicted due to

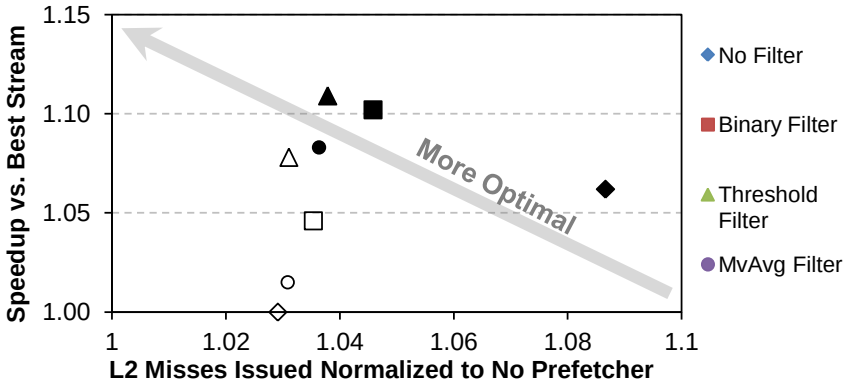

Figure 6.12: Prefetcher performance with various criticality filters. Solid markers are sequential prefetchers, and hollow markers are stream prefetchers. Aside from the *No Filter* marker for the stream prefetcher, which shows our *best stream* baseline, the other stream prefetchers are *Access* attached.

pollution is very minimal, a possible effect of the greater regularity of shared memory accesses.

Figure 6.11 shows the same data for the sequential prefetchers. Here, we see that without filtering, the sequential prefetcher can successfully predict the vast majority of memory requests, much of which is because the sequential prefetcher does not require a training period like the stream prefetcher does. However, this comes at a cost, as the prefetcher is also issuing many more unused prefetch requests. In all, this increases mean DRAM bandwidth usage by 8.7% over not having a prefetcher. Criticality filtering reduces the number of useless prefetches back down to the same level as stream prefetchers, but does so at the cost of not prefetching several useful requests. Fortunately, since these useful requests are non-critical, the lack of prefetching does not impact performance, as we saw in Section 6.7.2.

Figure 6.12 summarizes the mean bandwidth consumption and performance for our proposed filtered prefetchers. As we see, the prefetchers that use *Thres*

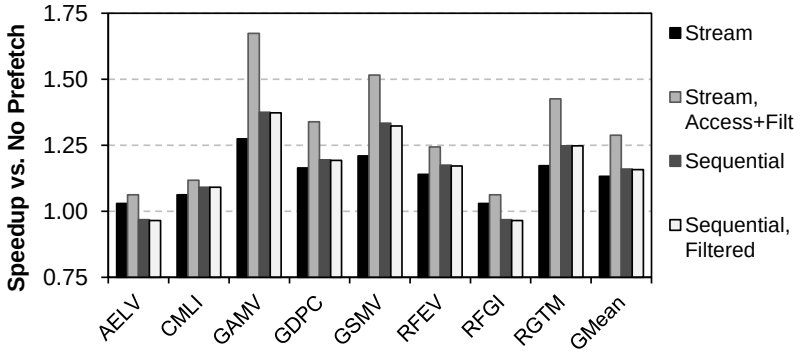

Figure 6.13: Performance of best parallel application prefetchers on multiprogrammed workloads.

filtering fall to the top left of the graph, indicating that they offer the best trade-off between performance and increased bandwidth utilization.

Because we improve performance while minimizing the impact on bandwidth, there should be a reduction in DRAM energy consumption. We modeled the energy consumption of our DRAM devices (see Section 6.6 for details). Compared to the best baseline stream prefetcher, our *Access* stream prefetcher with *Thres* filtering reduces total DRAM energy by an average of 6.4% across our applications. As we see in Figure 6.12, this is because the filtered stream prefetcher provides a significant performance improvement with very little increase in bandwidth. The sequential prefetcher with filtering saves even more energy, with a mean savings of 9.2%.

6.7.4 Multiprogrammed Workloads

Since a multicore processor can run both parallel applications and sequential workloads, it is important for us to evaluate how our prefetchers work in the context of multiprogrammed workloads. Figure 6.13 shows the performance of

Table 6.6: Best prefetcher settings for multiprogrammed workloads.

Prefetcher Type	**Parameters**
baseline stream	dist = 4, degr = 8
Access stream	dist = 16, degr = 1
sequential	$N = 4$

the best stream prefetcher and the best sequential prefetcher, with and without filtering. (As mentioned in Section 6.6, since we use quad-application workloads, we cut the number of stream entries in half to maintain the balance with respect to the earlier eight-core experiments for parallel applications.) We see that, without filtering, the mean weighted speedup [68] of the best sequential prefetcher (16.0%) is slightly higher than the best stream prefetcher (13.2%). We do see slight performance degradations for two of the workloads (*AELV* and *RFGI*) under the sequential prefetcher, whereas all workloads perform at least as well as not having a prefetcher for stream.

When we add criticality filtering to the prefetchers, we see that, unlike for our parallel applications, the stream prefetcher actually outperforms the sequential prefetcher, with the criticality-filtered stream prefetcher getting a mean weighted speedup of 28.8%. Unlike parallel applications, these workloads do not share memory, and are often contending for the same shared memory resources. Whereas the sequential prefetcher performed better due to predictable data layout and inter-thread synergy, our multiprogrammed applications do not display either characteristic. Here, the "smartness" of the stream prefetcher allows it to be more efficient at identifying which loads to prefetch, though clearly it cannot do so well without the same optimizations that we made for parallel applications.

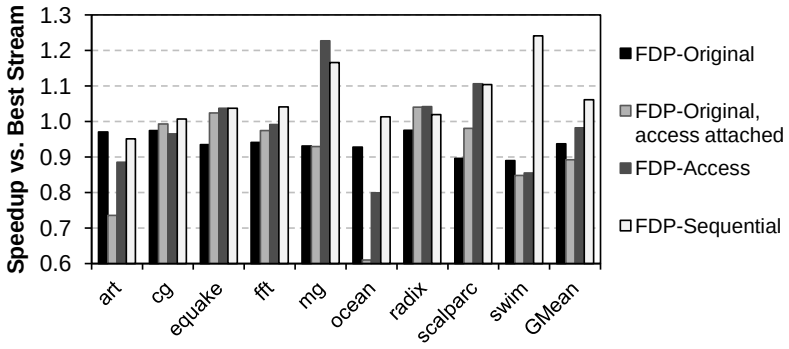

Figure 6.14: Performance of various modified FDP mechanisms (see Table 6.1) for parallel applications.

It is clear that a single static prefetcher design will not exploit the best performance for both types of applications. In order to avoid biasing the processor towards a particular workload type, a multicore processor should be able to provide a mechanism that can switch between prefetchers at run-time. One possible solution is to provide both prefetchers (as the sequential prefetcher adds a trivial hardware overhead), and to use a programmable register to switch between them. This register could even be assigned per-core, and could be set by the operating system (which will know a parallel application is running when it spawns a new thread). Alternatively, a sampling mechanism (e.g., similar to the one proposed by Jiménez et al. for the POWER7 to alter its stream prefetcher configuration [33]) could be used to determine which of the prefetchers works best for a given core in an interval. If desired, this could even be extended to support multiple aggression settings.

6.7.5 Accuracy-Based Throttling

We implement the feedback-directed prefetching (FDP) mechanism (see Section 6.5.1) to evaluate the utility of accuracy in throttling the prefetcher. We test

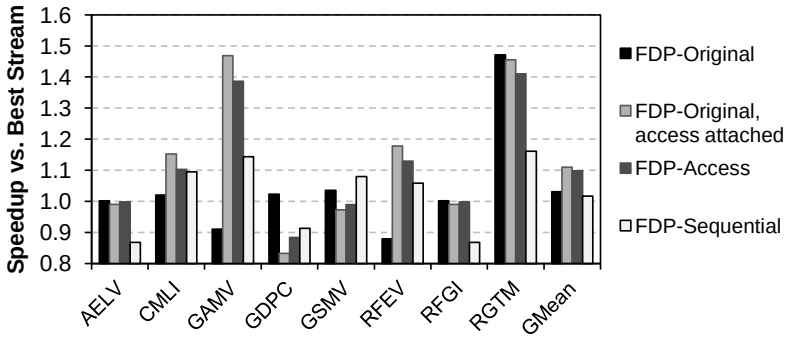

Figure 6.15: Performance of various FDP mechanisms (see Table 6.1) for multiprogrammed workloads.

three sets of prefetchers for FDP, and also try a configuration where we move the original FDP mechanism to be attached at access. As we see in Figure 6.14, most of these FDP mechanisms do not perform as well for our parallel applications. The only FDP configuration that outperforms the best baseline stream prefetcher, *FDP–Sequential*, only achieves a mean speedup of 6.1%, which is no better than the best static sequential prefetcher. FDP depends on accuracy, pollution, and lateness to guide its aggression, and as we have seen, our parallel applications do not exhibit low accuracy, and pollution is a non-issue, resulting in FDP's underperformance for parallel applications. Importantly, since all these observations are global, FDP misses out on opportunities to selectively filter requests, and thus can unnecessarily restrict prefetch performance.

As we turn to multiprogrammed workloads (Figure 6.15), we see that FDP offers better benefits, with all of the configurations on average performing better than the best baseline stream prefetcher. While FDP performs best when the original configuration is moved up to observe all accesses, with a mean weighted speedup of 11.0% over the best baseline, the overall results confirm the expected behavior of FDP in the multiprogramming context.

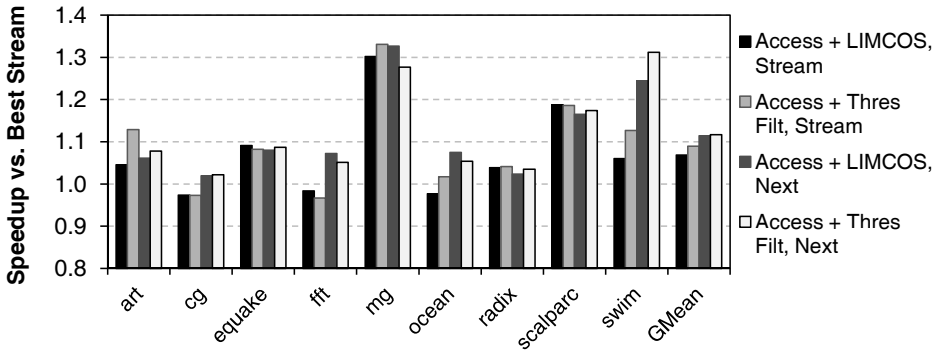

Figure 6.16: Speedup over the best stream prefetcher baseline for both LIMCOS filtering and our Commit Block Predictor based filtering when applied to aggressive prefetchers. The filtered stream and next configurations are those listed in Table 6.5.

6.7.6 Filtering Using the LIMCOS Classifiers

Now that we have refined our criticality-assisted prefetching mechanism, we revisit the LIMCOS classifiers [46] that we initially studied in Section 6.2. Figure 6.16 illustrates how the LIMCOS classifier compares with static *Thres* filtering using our simpler Commit Block Predictors (CBPs). (We do not show results for the *ConfEst* mechanism, as it performs slightly worse than their stall counting based classifier.) For the stream prefetcher, our filtering mechanism, despite its simpler hardware, actually exhibits a slight edge over the LIMCOS classifier, where we maintain a modest improvement in performance over the LIMCOS classifier for three of our parallel applications. With the sequential prefetcher, our filters effectively perform comparably, though again our criticality predictor does so using simpler hardware mechanisms. As such, we believe that the additional sophistication employed by the LIMCOS classifiers is unwarranted for our applications.

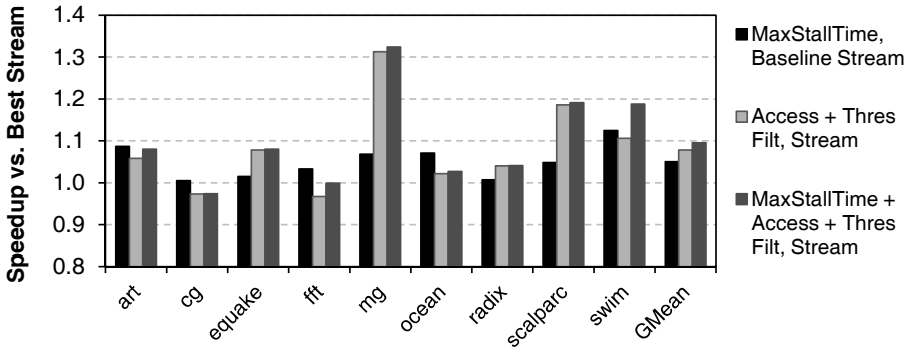

Figure 6.17: Speedup due to combining criticality-aware scheduling with CBP-filtered stream prefetchers.

6.7.7 Synergy with Criticality-Based Memory Scheduling

So far, the effects of adding criticality within the memory system have been studied for memory scheduling and prefetching in isolation. Since these tasks are somewhat independent of each other, we explore the feasibility of combining the two solutions together. In these combined mechanisms, a single *MaxStallTime* CBP is used, which performs a global reset every 100,000 cycles. Furthermore, prefetch requests issued by the prefetcher to DRAM are *not* marked as critical, regardless of the criticality of the triggering load instruction. This allows the memory scheduler to focus strictly on *demand misses*, which will now be those requests where the prefetcher was unable to successfully issue timely requests. CBP operation remains unchanged from Chapter 3.

Figure 6.17 shows the performance after combining the two mechanisms using a stream prefetcher. We see that initially, combining a CBP-based scheduler with the best traditional *non-filtered* stream prefetcher yields a mean speedup of 5.0% for our parallel applications, over simply using FR-FCFS scheduling with the prefetcher. In comparison, using FR-FCFS with the *Thres*-filtered, *Access*-

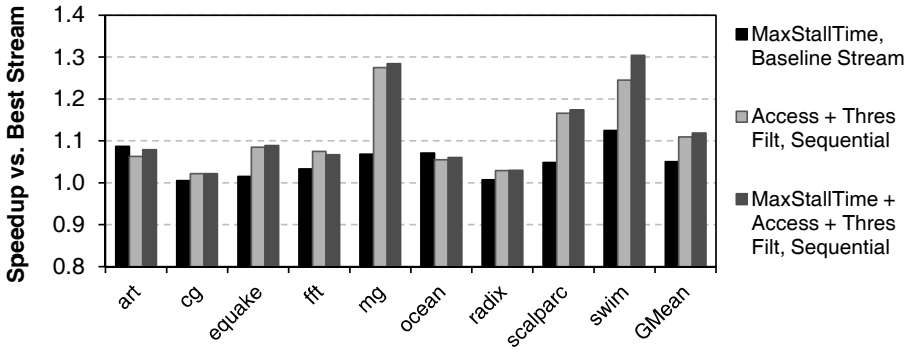

Figure 6.18: Speedup due to combining criticality-aware scheduling with CBP-filtered sequential prefetchers.

attached stream prefetcher was previously found to yield a 7.8% mean speedup. With a CBP-based scheduler, our improved stream prefetcher shows meager improvements, delivering an average 9.6% speedup over the best stream prefetcher baseline despite Figure 6.10 showing that 34% of DRAM requests are still demand misses (a third of which originate from load instructions).

The lack of synergy is more apparent in Figure 6.18, which shows the results after combining CBP-based scheduling with a *Thres*-filtered sequential prefetcher. Previously, the filtered prefetcher on its own was achieving a mean speedup of 10.9%. With criticality-aware scheduling added, only one benchmark, *swim*, sees any appreciable improvement (the mean speedup recorded for all parallel applications is 11.9%, within noise of the FR-FCFS results). From Figure 6.11, we see that 30% of DRAM requests are demand misses not predicted by the prefetcher, again with a third of these originating from load instructions.

The poor combined performance may be attributable in part to using the same criticality metric for both mechanisms. Since the current CBP design does not factor any feedback on whether a load instruction has been successfully prefetched, demand misses from load instructions may be subject to stale pre-

dictions, which in turn hurts the ability of the scheduler to differentiate amongst the candidate requests successfully. The remaining demand requests may also benefit from a different notion of criticality, one better suited to the orthogonality of these two mechanisms. We leave this exploration to future work.

6.7.8 Hardware Overhead

The overhead of load criticality prediction using the Commit Block Predictor remains quite minimal, and for an eight core system is within 1.8 kB of storage overhead (301 B for only the *Binary* predictor). Both of our threshold-based filters (*MvAvg* and *Thres*) require a 14-bit comparator sitting just outside the prefetcher to screen out non-critical or low-criticality memory requests. In order to implement the *MvAvg* filter detection, we also require an additional 224 B for our 128-entry FIFO queue, as well as a 22-bit counter, adder, and subtractor.

While other prefetch control mechanisms expend significant overhead in monitoring useful prefetches, late prefetches, and maintaining hardware to predict misses due to pollution, our filtering mechanisms require no such monitoring within the cache. The binary filtering logic is especially simple, and is expected to have a minimal energy impact within the processor.

6.8 Summary

We have revisited prefetching in the context of modern multicore-based systems, and have found that aggressive sequential prefetchers, previously dismissed as harmful, surprisingly outperform the "smarter" stream prefetcher

when running parallel applications. Our data also shows that accuracy-based throttling, which is typically used to modulate prefetching aggressiveness, is not productive in this context. Instead, we have proposed to implement selective prefetch filtering, based on the notion of load criticality. Our results show that it is a better use of the available bandwidth, by filtering prefetch loads whose latency can be tolerated well by the processor.

Criticality-filtered sequential prefetchers provide a mean speedup of 10.9% for parallel applications over the best stream prefetcher, with all applications performing at least as well as they did under the stream prefetcher. Conversely, for multiprogrammed workloads, we show that stream prefetchers may be a better choice, especially when combined with our filtering mechanism. Consequently, we have proposed dynamic mechanisms that can allow the system to automatically adopt the best prefetcher in each case.

CHAPTER 7

RELATED WORK

7.1 Instruction Criticality

Fields et al. proposed a method for statically determining the critical path of an application using directed graphs, and proposed a token-based hardware mechanism to approximate this in hardware [15]. The offline graph mechanism is described in greater detail in Section 2.2.1. Using this model, Fields proposed a tracking predictor to determine criticality. This mechanism uses a series of forward-flow tokens that propagate during sampling phases. A token is assigned to one of the instruction's nodes, and based on the dependency edges that extend from that node, the token is propagated amongst the last-arriving of these edges. When there is no last-arriving edge from the node, the token dies. The predictor checks for the token after a fixed time has passed; if the token is still alive, the instruction is assumed to be critical, and is saved in a PC-indexed table. Subsequent occurrences of that PC will be marked as critical. While the token-based method of criticality tracking is most faithful to the original definition of instruction criticality, the hardware implementation is quite complex. Fields et al. use their predictors in the context of value prediction and clustered architectures.

Li et al. extend the original Fields model for shared memory processors [44]. In addition to the program edges originally modeled, read-after-write (RAW) dependencies are added across the graphs of different threads. However, many system-level interactions due to thread interference are not factored into this model.

One of the first works to study criticality was by Srinivasan and Lebeck [72]. They examined the amount of time that each load could be delayed, as well as the loads that must be serviced within a single cycle, to maintain ideal issue/commit rates. In doing so, they identify loads which are on the critical path, and show that guaranteeing that these loads hit in the L1 cache increases performance by an average of 40% [71]. For an online implementation, they mark loads as critical if their value is consumed by a mispredicted branch or by a subsequent load that misses in the L1 cache, or if the issue rate after load issue falls below a fixed threshold. There is significant overhead for their table structure, as it maintains an array of bits, the size of the LSQ, for each ROB entry.

Tune et al. use a number of statistics to determine whether an instruction is critical [76], based on a series of profiling observations. They flag an instruction as a candidate for being critical if: *(a)* it is the oldest instruction in the issue queue, *(b)* the instruction produces a value that is consumed by the oldest instruction in the issue queue, *(c)* it is the oldest instruction in the ROB, *(d)* the instruction has the most number of consumers in the issue queue, or *(e)* its execution allows at least three instructions in the issue queue to be marked as ready. If the number of times an instruction has been marked as a candidate exceeds a fixed threshold, the instruction is considered to be critical. They evaluate each of these five criticality metrics for value prediction, finding that criterion *(a)* is the most effective. While criterion *(c)* is similar to our idea of tracking ROB blocks by long-latency loads, Tune's implementation also tracks non-load instructions, as well as instructions at the head of the ROB that do not stall. As with other dependency-based predictors, the ability to capture criteria *(b)*, *(d)*, and *(e)* can be costly in hardware. Like Fields, they use their predictors in the context of value prediction and clustered architectures.

Salverda and Zilles provide some level of stratification for criticality by ranking instructions on their likelihood of criticality, based on their prior critical frequency [66]. They hypothesize that a larger number of critical occurrences correlates to a higher need for optimizing a particular instruction. This work is extended upon by using probabilistic criticality likelihood counters to both greatly reduce the storage overhead of the Fields et al. token-based mechanism and increase its effectiveness [64].

7.2 Load Criticality

The idea of load criticality remains ill defined in literature, with different proposals using different ad hoc metrics for criticality classification. On a high level, these approaches can be partitioned into two areas: Coarse granularity metrics, which prioritize loads based on some phase-dependent behavior in the application, and fine granularity metrics, which use characteristics about each individual instruction to determine its prioritization.

A number of coarse granularity metrics exist. Fisk and Bahar consider loads issued when the overall processor issue rate is low to be critical [18]. They use this concept to filter out non-critical loads into a small buffer akin to the victim cache. Criticality is tracked similarly to Srinivasan and Lebeck, where a low issue rate is used to determine criticality. Thread cluster memory (TCM) scheduling [38] classifies threads into either a latency-sensitive or bandwidth-sensitive cluster. Requests from latency-sensitive threads are prioritized over those from bandwidth-sensitive threads, while inside the bandwidth-sensitive cluster, threads are prioritized to maximize fairness. The Minimalist Open-

page scheduler also ranks threads based on the importance of the request [36]. Threads with low memory-level parallelism (MLP) are ranked higher than those with high MLP, which are themselves ranked higher than prefetch requests.

Several works also exist for fine granularity criticality classification. Srinivasan and Lebeck used the types of instructions in the dependence chain of a load (e.g., a load produces a value used by a branch) to determine the criticality of the load [71]. Fisk and Bahar use the number of dependencies as a second measure of criticality [18]. Unlike other predictor-based models of criticality, they determine the status of a load based on what occurs as the load is being serviced, since the criticality need only be determined at cache line allocation time. While this eliminates prediction table overhead, they must send this dependence information to the MSHRs every cycle, which, for a large number of outstanding loads, can be costly in terms of bandwidth.

İpek et al. [28], as well as Prieto et al. [60], propose using the distance of a load from the head of the reorder buffer as an indicator of load criticality. Prieto et al. propose alternative load criticality metrics as well, including instruction cache misses and the number of outstanding branches [60]. Subramaniam et al. use the number of direct consumers to gauge the criticality of a load instruction [73].

Runahead [10, 54] and CLEAR [39] both propose mechanisms to alleviate the effect of blocking the head of the ROB at commit, which often include long-latency loads. These works, along with the consumer count proposed by Subramaniam et al., are described in greater detail in Section 2.2.2.

7.3 Memory Scheduling

MORSE is a state-of-the-art self-optimizing memory scheduler extending upon the original proposal by İpek et al. [28, 52]. The reinforcement learning algorithm uses a small set of features to learn the long-term behavior of the memory requests. At each point in time, the DRAM command with the best expected long-term impact on performance is selected. The authors use feature selection to derive the best attributes to capture the state of the memory system. It is important to note that criticality, as defined by İpek et al., only considers the age of the load request, and does not take into account that the oldest outstanding load may not fall on the critical path of processor execution, as we have shown.

The adaptive history-based (AHB) memory scheduler by Hur and Lin uses previous memory request history to predict the amount of delay a new request will incur, using this to prioritize scheduling decisions that are expected to have the shortest latency [23].

The most relevant schedulers, those that focus on prioritizing requests from critical threads, are described in Section 7.2.

Other work in the area of memory scheduling has targeted a variety of applications. Ebrahimi et al. demonstrate a memory scheduler for parallel applications [11]. Using a combination of hardware and software, the thread holding the critical section lock is inferred by the processor, and its memory requests are prioritized. In the event of a barrier, thread priorities are shuffled to decrease the time needed for all threads to reach the barrier. Fairness-oriented schedulers (e.g., PAR-BS [53], ATLAS [37]) target reducing memory latency strictly in the context of multiprogrammed workloads.

7.4 Prefetching

There is a large body of work on hardware and software prefetcher design, much of which is surveyed by Vanderwiel and Lilja [77].

Software prefetching involves the use of explicit instructions within the ISA to inform the processor that it should prefetch data. Intel [25], AMD [1], IBM [27], and ARM [2] all offer some form of prefetch instructions. The programmer or compiler must explicitly specify the desired memory address, and often, the destination cache level for retrieved data or instructions can be controlled. Compiler support for automatically inserting prefetch instructions dates back to the work of Mowry et al., which performed a locality analysis and loop splitting to determine when the instructions should be scheduled [50].

Three of the basic hardware prefetchers—sequential, stride, and stream—are discussed at length in Section 2.1.3. Markov prefetchers try to exploit pattern irregularity by building chains of requests in the hopes of identifying address recurrence [34]. While promising, Markov prefetchers are unable to extrapolate observed misses to newer patterns, and require significant hardware area in order to track a useful history length. Global history buffer (GHB) based prefetching uses a linked list of the most recent data requests to identify strides in the access pattern [56]. The GHB prefetcher can either improve prefetchers such as the Markov and stride prefetchers, or it can be used to implement new types of prefetch mechanisms, such as delta correlation for identifying recurrent access patterns.

Sandbox prefetching works to mitigate the effects of aggressive prefetchers by using a Bloom filter to store the addresses of potential prefetch request ad-

dresses, and to confirm over time whether a prefetch request would have been accurate [62]. If the accuracy exceeds a certain threshold, then the prefetch requests will begin to be issued to the DRAM. While sandbox prefetching offers a mechanism to combine the requests of the stream and next line prefetchers, it still performs accuracy-based filtering, which as we saw can be overly conservative, and still requires training time.

Lee et al. analyze the performance of both hardware and software prefetchers for single-thread applications on modern systems [43]. They find that software prefetching can actually be invariant to the hardware configuration if the instruction issues its request to cover at least the minimum prefetch delay for the system. On the other hand, traditional hardware prefetchers can often be overly conservative, and miss opportunities to exploit short streams. They advocate for a combined approach where software prefetching instructions augment the hardware prefetcher coverage.

Some proposals have looked into updating prefetcher designs for CMPs. Ebrahimi et al. examined the coordination of multiple private prefetchers to throttle performance and aggression [12]. Enright Jerger et al. examine the impact of cache coherence on private prefetchers for multiprocessors, and propose a taxonomy for classifying such prefetcher requests [13]. This work was extended to analyze the impact of data prefetching on CMPs [20]. Son et al. propose a scalable compiler-directed technique to perform software prefetching in a CMP [69].

Others have proposed adaptive methods for data fetching in processors. The Amoeba-Cache combines tags and data into a single space within a cache to provide dynamically-adjusted line sizes [40]. The ZCache takes a somewhat differ-

ent approach, using hashing and linking to create sets with variable numbers of ways [67]. While not dynamic, Przybylski studied the effects of changing block size and fetch size in caches [61].

Aside from multiple prefetcher coordination, a number of projects look at adapting based on prefetching feedback. Feedback-directed prefetching provides a framework that can tune the aggressiveness of a prefetcher and the cache insertion policy at run time [70]. Jimenez et al. describe a modification to the POWER7 prefetcher that allows it to adaptively change its aggression [33]. Lee et al. studied a memory scheduler that can adapt the priority of prefetch requests based on current bandwidth utilization [42]. PACMan proposes a dynamic cache insertion policy that caters to the longevity of a prefetched cache line [79]. The LIMCOS classifier identifies those loads generating the greatest number of commit stalls for the purpose of filtering prefetches [46]. We evaluate our filtering mechanism against both feedback-directed prefetching and LIMCOS classification in Section 6.7.

CHAPTER 8

CONCLUSION

This thesis explores a novel approach to making decisions within the memory system. By symbiotically leveraging assistance from the processor, we can greatly improve the sophistication of several decisions made within the memory system. While this work specifically explores one instance of this concept, through the use of load criticality based on instructions that stall commit within the processor, we envision that the concept could be extended significantly to encompass a wide variety of information from the processor core. Even with our simple instance, we find several applications where we can improve performance by an average of about 10%.

Our contributions in memory scheduling identify and address two requirements simultaneously: (1) the need to simplify DRAM controllers, and (2) the need for a more program-oriented scheduling approach. *We propose to shift a substantial fraction of the scheduling mechanism to the processor side.* In our design, the processor-side scheduler can tap onto the much richer processor-side state, where the program actually executes, to infer the relative importance of each memory request. This relative importance is packaged into a numerical ranking that is sent along with the request to the memory controller, where a quick priority-based scheduling decision can be made. Because the processor-side scheduler is dissociated from the controller, it is no longer bound by the DRAM clock cycle; furthermore, it can begin analyzing memory requests as early as the decode stage in the CPU (or earlier if PC-based prediction is used). The net result is highly sophisticated memory scheduling *and* simple, fast memory controller designs.

With respect to prefetchers, we show that archaic assumptions about hardware prefetcher design are restricting their potential. By leveraging the processor to inform the prefetcher of the loads that are impacting program performance the greatest, we can again make a smarter decision. Ultimately, the choice to prefetch involves a risk—while there are significant performance savings to be had when the prefetch is correct and early enough, a mistake by the prefetcher can be costly, using unnecessary bandwidth and energy while generating additional cache pollution. In many cases, if unchecked, this pollution can undermine any benefits from prefetching. By using the degree of criticality as a form of risk analysis, we can selectively elect to take the risk of pollution and contention if the potential rewards are high (in our case, if the program stands to benefit greatly from a successful prefetch). The effectiveness of such fine-grained control also allows us to significantly increase the aggression of the prefetchers, providing further performance benefits while keeping the potential harmful impacts low.

We believe the insights we have discussed here are fundamental to the future of memory controller design. More importantly, our processor-assisted approach encourages architects to challenge traditional assumptions of modular design. By slightly blurring the boundaries of the cores and memory, our simple idea delivered significant performance boosts. In an era where Systems-on-Chip continue to integrate functionality on die, there will likely be many more such opportunities where simple inter-module cooperation can lead to significant synergistic benefits. We hope that our work encourages designers to revisit previous design assumptions, which as we saw may no longer hold, and deal with discrete components in a holistic, system-aware context.

CHAPTER 9

FUTURE WORK

9.1 Revisiting the Instruction Criticality Graph Model

While the results from Section 5.2 show that commit stalls were amongst the best single metric observed, they still do not offer insight as to whether we achieved the maximum attainable performance improvement from using the concept of load criticality. In practice, a large number of interactions within the processor and the memory hierarchy can have a combined impact on scheduling (as was exploited by reinforcement-learning-based memory schedulers [28, 52]). We believe that turning to the instruction criticality graphs proposed by Fields et al. [15] can help identify opportunities for improvement.

As discussed in Section 2.2.1, a directed acyclic graph (DAG) of program execution can be constructed by combining information about timing and instruction dependencies. Figure 2.5 shows the graph as originally proposed. In the original work, the delays due to a memory operation were largely considered implicitly; most notably, the time recorded for the E node during a load instruction is when the instruction exits the ALU after effective address calculation [15, 16]. This assumption worked well for Fields et al. because they assumed a fixed latency lookup for DRAM, providing miss penalties that had regular predictability due to the lack of request reordering.

Unfortunately, though we would like to use a DAG to identify load criticality, the graph as previously proposed makes several mistakes for long-latency memory operations. For example, we observed that for a miss to DRAM, even

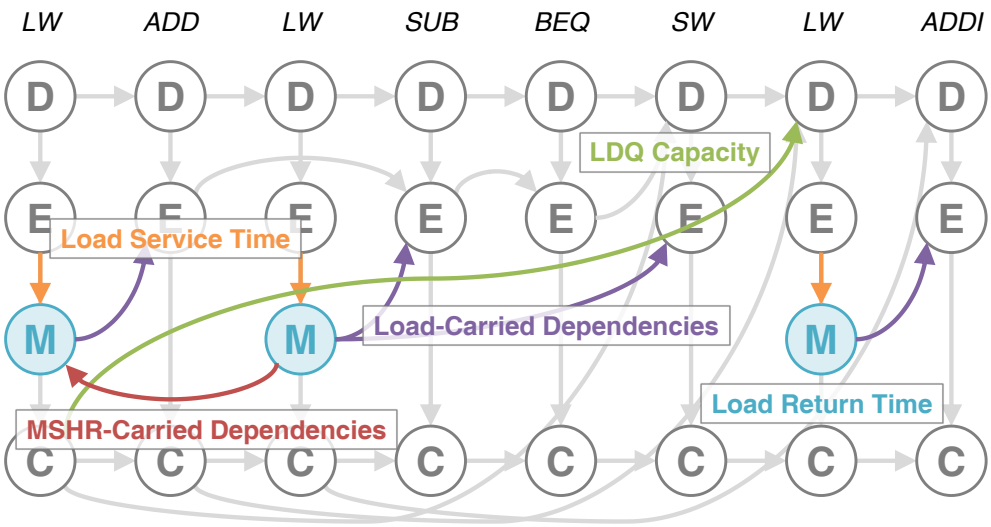

Figure 9.1: Modified directed acyclic graph for instruction criticality, containing new nodes and edges to encapsulate load behavior.

though the consumer instruction was waiting on the producing load to return from memory, the critical path was incorrectly routed through the D node of the consumer instruction, since the data dependency was only captured by an $E \rightarrow E$ edge. While altering the E node to represent when an operation completes may have corrected this particular mistake, we instead choose to extend the graph to capture more robust information about the behavior of a load.

Figure 9.1 shows a potential modified graph. Here, we introduce the M node, which exists only for load instructions and captures the time at which a load returns from memory. We also introduce a number of new edges to the graph:

- additional $C \rightarrow D$ edges, representing when the load queue is at capacity;
- $E \rightarrow M$, which captures the time of the load operation itself;
- $M \rightarrow E$, to represent data dependencies where the producer is a load instruction (replacing the $E \rightarrow E$ edges for load instructions);
- $M \rightarrow M$, capturing secondary miss behavior due to the MSHR; and

- $M \rightarrow C$, representing the need to wait for a load to return before an instruction can commit.

These edges allow us to track the behavior of load-related operations. Previously, Fields et al. marked instructions as critical if the critical path of execution timing passed through the E node, as their optimizations targeted execution in the processor back end. For our work, we focus not on a node, but in particular on the $E \rightarrow M$ edge, which represents the latency of the memory operation (which is what we directly control in the memory scheduler). Those $E \rightarrow M$ edges falling along the critical path have at least part of the memory operation contributing to the overall execution time of the program, and shortening the load return delay should in theory reduce the execution time of the program.

9.2 History-Based Criticality Prediction

As mentioned in Section 5.2, load criticality and instruction criticality are largely thought of as program counter dependent (i.e., dynamic instances of the same static instruction will exhibit similar criticality behavior), due to loop recurrent behavior. However, our offline analysis reveals that this assumption is untrue in several cases. During the execution of a loop that contains a load instruction, if the loop iterations have some degree of data independence, the hardware will effectively perform some level of loop unrolling, where multiple iterations of the loop will be in flight concurrently. Effectively, this allows the processor to exploit memory-level parallelism by issuing several memory requests simultaneously. Often, even if those subsequent operations must go to DRAM, if the oldest such load incurs a long load latency by going to DRAM, it can mask most

if not all of the latencies of these parallel load operations, hence making them non-critical. (This is really an extension of our motivation from Section 2.2.2 to encompass loads from the same static instruction.)

This behavior suggests that some form of history-based tracking, such as those from the field of branch prediction, would allow our criticality predictors to be more judicious. For example, the two-level adaptive branch predictor [81] can be modified for use in criticality. Instead of using the observed history to record whether a branch was taken or not taken, the predictor can record whether a load instruction was critical or not critical. As was done in the two-level predictor, a saturating counter can be used to predict whether the next load will be critical. A request predicted to be critical can then look up a second table (such as our *MaxStallTime* predictor) to determine the rank of criticality that should be assigned, akin to how a branch target buffer is looked up to determine the target address if a branch is predicted to be taken.

One modification to the branch predictor mechanism involves the way in which the histories are updated. For criticality, we expect to maintain two history tables—a speculative table that tracks the expected history based on our predictions at instruction dispatch time, and a non-speculative table that records the observed criticality behavior at commit time and uses this to update the saturating counter. Traditionally, such an approach for branch prediction requires an expensive fix-up mechanism in the case of mispredicted branches, as the wrong-path instructions are squashed, and their corresponding histories must be reverted. Since our use of the branch predictor mechanism exclusively takes place in the processor back end, we are guaranteed to only observe correct-path instructions, and thus do not need to worry about any such fix-up.

APPENDIX A

LIST OF CRITERIA TESTED DURING THE SYSTEMATIC SEARCH

The following is a list of criticality criteria that were investigated while performing a systematic search for the best known load criticality metric for memory scheduling (see Section 5.2 for more details).

1. Number of cycles for which the instruction stalled commit
2. If the instruction stalled commit
3. If the instruction did not stall commit
4. Number of cycles for which the instruction stalled commit and the reorder buffer was full
5. If the instruction stalling at commit caused the reorder buffer to fill up
6. If the instruction stalling at commit did not cause the reorder buffer to fill up
7. Number of cycles for which the instruction stalled commit and no free physical registers were available
8. If the instruction stalling at commit caused the processor to run out of physical registers
9. If the instruction stalling at commit did not cause the processor to run out of physical registers
10. Number of cycles for which the issue queue was full while the instruction was ready
11. If the issue queue was full when the instruction was ready
12. If the issue queue was not full when the instruction was ready

13. Number of cycles for which the instruction stalled commit and the load queue was full
14. If the instruction stalling at commit caused the load queue to fill up
15. If the instruction stalling at commit did not cause the load queue to fill up
16. Number of cycles for which the instruction stalled commit and the store queue was full
17. If the instruction stalling at commit caused the store queue to fill up
18. If the instruction stalling at commit did not cause the store queue to fill up
19. Number of cycles a branch was unable to be issued as too many other branches were outstanding
20. If a branch was unable to issue as too many other branches were outstanding
21. If a branch was able to issue as too many other branches were not outstanding
22. Number of instructions in the issue queue at dispatch
23. Number of free issue queue entries at dispatch
24. If the issue queue was free when the instruction was ready to dispatch
25. If the issue queue was full when the instruction was ready to dispatch
26. Distance from the head of the reorder buffer the instruction was inserted at when dispatched
27. Number of free reorder buffer entries at dispatch
28. If the reorder buffer has a free entry when the instruction was ready to dispatch
29. If the reorder buffer was full when the instruction was ready to dispatch

30. Number of cycles the instruction spent between instruction fetch and instruction dispatch
31. Number of cycles the instruction spent between instruction fetch and instruction execution
32. Number of cycles the instruction spent between instruction fetch and load issue
33. Number of cycles the instruction spent between instruction fetch and load return
34. Number of cycles the instruction spent between instruction fetch and instruction commit
35. Number of cycles the instruction spent in the processor back end
36. Number of cycles between load issue and load return
37. Number of cycles between the start of effective address calculation and load return
38. Number of valid load queue entries at dispatch
39. Number of free load queue entries at dispatch
40. If the load queue has a free entry when the instruction was ready to dispatch
41. If the load queue was full when the instruction was ready to dispatch
42. Number of valid store queue entries at dispatch
43. Number of free store queue entries at dispatch
44. If the store queue has a free entry when the instruction was ready to dispatch
45. If the store queue was full when the instruction was ready to dispatch

46. Difference between the number of load queue entries and the number of store queue entries when the instruction was dispatched
47. Difference between the number of store queue entries and the number of load queue entries when the instruction was dispatched
48. If the load was forwarded from the store queue
49. If the load was not forwarded from the store queue
50. Distance of the instruction from the head of the reorder buffer when the load is issued
51. Distance of the instruction from the last available entry of the reorder buffer when the load is issued
52. Distance of the instruction from the head of the load queue when the load is issued
53. Distance of the instruction from the last available entry of the load queue when the load is issued
54. Number of valid reorder buffer entries when the load is issued
55. Number of free reorder buffer entries when the load is issued
56. If the reorder buffer has a free entry when the load is issued
57. If the reorder buffer was full when the load is issued
58. Number of valid load queue entries when the load is issued
59. Number of free load queue entries when the load is issued
60. If the load queue has a free entry when the load is issued
61. If the load queue was full when the load is issued
62. Number of valid store queue entries when the load is issued

63. Number of free store queue entries when the load is issued
64. If the store queue has a free entry when the load is issued
65. If the store queue was full when the load is issued
66. Distance of the instruction from the head of the reorder buffer when the load returns
67. Distance of the instruction from the last available entry of the reorder buffer when the load returns
68. Distance of the instruction from the head of the load queue when the load returns
69. Distance of the instruction from the last available entry of the load queue when the load returns
70. Number of valid reorder buffer entries when the load returns
71. Number of free reorder buffer entries when the load returns
72. If the reorder buffer has a free entry when the load returns
73. If the reorder buffer was full when the load returns
74. Number of valid load queue entries when the load returns
75. Number of free load queue entries when the load returns
76. If the load queue has a free entry when the load returns
77. If the load queue was full when the load returns
78. Number of valid store queue entries when the load returns
79. Number of free store queue entries when the load returns
80. If the store queue has a free entry when the load returns
81. If the store queue was full when the load returns

82. Number of cycles since the last instruction was fetched
83. Number of cycles since the last instruction was dispatched
84. Number of cycles since the last load was issued
85. Number of cycles since the last load was dispatched
86. Number of cycles since the last instruction was committed
87. If the last branch was mispredicted
88. If the last branch was correctly predicted
89. If the last branch was taken
90. If the last branch was not taken
91. If the last branch was predicted as taken
92. If the last branch was predicted as not taken
93. Number of cycles since the last mispredicted branch
94. Number of cycles since the last branch
95. Misprediction penalty for the last branch
96. Number of cycles since the last instruction cache miss

BIBLIOGRAPHY

[1] Advanced Micro Devices, Inc. *AMD64 Architecture Programmer's Manual, Volume 3: General-Purpose and System Instructions*, October 2013. `http://support.amd.com/TechDocs/24594.pdf`.

[2] ARM Limited. *ARM Architecture Reference Manual: ARMv7-A and ARMv7-R Edition*, July 2012.

[3] V. Aslot. Performance characterization of the SPEC OMP benchmarks, May 2002.

[4] V. Aslot and R. Eigenmann. Quantitative performance analysis of the SPEC OMPM2001 benchmarks. *Scientific Programming*, 11(2):105–124, 2003.

[5] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. NAS parallel benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, March 1994.

[6] I. Bhati, Z. Chishti, and B. Jacob. Coordinated refresh: Energy efficient techniques for DRAM refresh scheduling. In *International Symposium on Low Power Electronics and Design*, 2013.

[7] K. K.-W. Chang, D. Lee, Z. Chishti, A. R. Alameldeen, C. Wilkerson, Y. Kim, and O. Mutlu. Improving DRAM performance by parallelizing refreshes with accesses. In *International Symposium on High-Performance Computer Architecture*, 2014.

[8] M.-T. Chang, P. Rosenfeld, S.-L. Lu, and B. Jacob. Technology comparison for large last-level caches (L^3Cs): Low-leakage SRAM, low write-energy STT-RAM, and refresh-optimized eDRAM. In *International Symposium on High-Performance Computer Architecture*, 2013.

[9] T. Chen, R. Raghavan, J. Dale, and E. Iwata. Cell Broadband Engine Architecture and its first implementation, 2005. `http://www.ibm.com/developerworks/power/library/pa-cellperf/`.

[10] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *International Conference on Supercomputing*, 1997.

[11] E. Ebrahimi, R. Miftakhutdinov, C. Fallin, C. J. Lee, O. Mutlu, and Y. N. Patt. Parallel application memory scheduling. In *International Symposium on Microarchitecture*, 2011.

[12] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt. Coordinated control of multiple prefetchers in multi-core systems. In *International Symposium on Microarchitecture*, 2009.

[13] N. D. Enright Jerger, E. L. Hill, and M. H. Lipasti. Friendly fire: Understanding the effects of multiprocessor prefetches. In *International Symposium on Performance Analysis of Systems and Software*, 2006.

[14] W. Felter and T. Keller. Power measurement on the Apple Power Mac G5. Technical Report RC23276, International Business Machines Corporation, 2004.

[15] B. Fields, S. Rubin, and R. Bodík. Focusing processor policies via critical-path prediction. In *International Symposium on Computer Architecture*, 2001.

[16] B. A. Fields. *Using Criticality to Attack Performance Bottlenecks*. Ph.D. dissertation, Dept. of Electrical Engineering and Computer Sciences, University of California at Berkeley, December 2006.

[17] B. A. Fields, R. Bodík, and M. D. Hill. Slack: Maximizing performance under technological constraints. In *International Symposium on Computer Architecture*, 2002.

[18] B. R. Fisk and R. I. Bahar. The non-critical buffer: Using load latency tolerance to improve data cache efficiency. In *International Conference on Computer Design*, 1999.

[19] J. W. C. Fu, J. H. Patel, and B. L. Janssens. Stride directed prefetching in scalar processors. In *International Symposium on Microarchitecture*, 1992.

[20] N. Fukumoto, T. Mihara, K. Inoue, and K. Murakami. Analyzing the impact of data prefetching on chip multiprocessors. In *Asia-Pacific Computer Systems Architecture Conference*, 2008.

[21] S. Ghose, H. Lee, and J. F. Martínez. Improving memory scheduling via processor-side load criticality information. In *International Symposium on Computer Architecture*, 2013. DOI: 10.1145/2485922.2485930.

[22] J. L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, 33(7):28–35, July 2000.

[23] I. Hur and C. Lin. Adaptive history-based memory schedulers. In *International Symposium on Microarchitecture*, 2004.

[24] Intel Corporation. *ARK: Product Information*. `http://ark.intel.com/`.

[25] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, July 2013. `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf`.

[26] International Business Machines Corporation. *Power servers*. `http://www-03.ibm.com/systems/power/hardware/`.

[27] International Business Machines Corporation. *Power ISA Version 2.07*, May 2013. `https://www.power.org/wp-content/uploads/2013/05/PowerISA_V2.07_PUBLIC.pdf`.

[28] E. İpek, O. Mutlu, J. F. Martínez, and R. Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *International Symposium on Computer Architecture*, 2008.

[29] B. Jacob, S. W. Ng, and D. T. Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, San Francisco, CA, 2007.

[30] A. Jaleel, K. B. Theobald, S. C. Steely Jr., and J. Emer. High performance cache replacement using re-reference interval prediction (RRIP). In *International Symposium on Computer Architecture*, 2010.

[31] JEDEC Solid State Technology Association. *JESD79-4: DDR4 SDRAM*, September 2012. `http://www.jedec.org/sites/default/files/docs/JESD79-4.pdf`.

[32] W. Jia, K. A. Shaw, and M. Martonosi. MRPB: Memory request prioritization for massively parallel processors. In *International Symposium on High-Performance Computer Architecture*, 2014.

[33] V. Jiménez, R. Gioiosa, F. J. Cazorla, A. Buyuktosunoglu, P. Bose, and F. P. O'Connell. Making data prefetch smarter: adaptive prefetching on

POWER7. In *International Conference on Parallel Architectures and Compilation Techniques*, 2012.

[34] D. Joseph and D. Grunwald. Prefetching using Markov predictors. In *International Symposium on Computer Architecture*, 1997.

[35] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *International Symposium on Computer Architecture*, 1990.

[36] D. Kaseridis, J. Stuecheli, and L. K. John. Minimalist open-page: A DRAM page-mode scheduling policy for the many-core era. In *International Symposium on Microarchitecture*, 2011.

[37] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *International Symposium on High-Performance Computer Architecture*, 2010.

[38] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *International Symposium on Microarchitecture*, 2010.

[39] N. Kırman, M. Kırman, M. Chaudhuri, and J. F. Martínez. Checkpointed early load retirement. In *International Symposium on High-Performance Computer Architecture*, 2005.

[40] S. Kumar, H. Zhao, A. Shriraman, E. Matthews, S. Dwarkadas, and L. Shannon. Amoeba-Cache: Adaptive blocks for eliminating waste in the memory hierarchy. In *International Symposium on Microarchitecture*, 2012.

[41] B. C. Lee, E. İpek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable DRAM alternative. In *International Symposium on Computer Architecture*, 2009.

[42] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt. Prefetch-aware memory controllers. In *International Symposium on Microarchitecture*, 2008.

[43] J. Lee, H. Kim, and R. Vuduc. When prefetching works, when it doesn't, and why. *IEEE Transactions on Architecture and Code Optimization*, 9(1):2:1–2:29, March 2012.

[44] T. Li, A. R. Lebeck, and D. J. Sorin. Quantifying instruction criticality for

shared memory multiprocessors. In *Symposium on Parallel Algorithms and Architectures*, 2003.

[45] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu. RAIDR: Retention-aware intelligent DRAM refresh. In *International Symposium on Computer Architecture*, 2012.

[46] R. Manikantan and R. Govindarajan. Performance oriented prefetching enhancements using commit stalls. *Journal of Instruction-Level Parallelism*, 13(1–28), March 2011.

[47] Micron Technology, Inc. *512Mb DDR2 SDRAM Component Data Sheet: MT47H128M4*, March 2006. `http://www.micron.com/-/media/documents/products/data%20sheet/dram/ddr2/512mbddr2.pdf`.

[48] Micron Technology, Inc. *Technical Note TN-41-01: Calculating Memory System Power for DDR3*, June 2006. `http://www.micron.com/-/media/documents/products/technical%20note/dram/tn41_01ddr3_power.pdf`.

[49] Micron Technology, Inc. *1Gb DDR3 SDRAM Component Data Sheet: MT41J128M8*, September 2012. `http://www.micron.com/~/media/Documents/Products/DataSheet/DRAM/1Gb_DDR3_SDRAM.pdf`.

[50] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992.

[51] J. Mukundan, H. Hunter, K.-H. Kim, J. Stuecheli, and J. F. Martínez. Understanding and mitigating refresh overheads in high-density DDR4 DRAM systems. In *International Symposium on Computer Architecture*, 2013.

[52] J. Mukundan and J. F. Martínez. MORSE: Multi-objective reconfigurable self-optimizing memory scheduler. In *International Symposium on High-Performance Computer Architecture*, 2012.

[53] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *International Symposium on Computer Architecture*, 2008.

[54] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt. Runahead execution: An

alternative to very large instruction windows for out-of-order processors. In *International Symposium on High-Performance Computer Architecture*, 2003.

[55] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair queueing memory systems. In *International Symposium on Microarchitecture*, 2006.

[56] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. In *International Symposium on High-Performance Computer Architecture*, 2004.

[57] OpenMP Architecture Review Board. *OpenMP Specifications*, `http://www.openmp.org`.

[58] S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *International Symposium on Computer Architecture*, 1994.

[59] J. Pisharath, Y. Liu, W. Liao, A. Choudhary, G. Memik, and J. Parhi. NU-MineBench 2.0. Technical Report CUCIS-2005-08-01, Northwestern University, August 2005.

[60] P. Prieto, V. Puente, and J. A. Gregorio. CMP off-chip bandwidth scheduling guided by instruction criticality. In *International Conference on Supercomputing*, 2013.

[61] S. Przybylski. The performance impact of block sizes and fetch strategies. In *International Symposium on Computer Architecture*, 1990.

[62] S. H. Pugsley, Z. Chishti, C. Wilkerson, P.-F. Chuang, R. L. Scott, A. Jaleel, S.-L. Lu, K. Chow, and R. Balasubramonian. Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers. In *International Symposium on High-Performance Computer Architecture*, 2014.

[63] Jose Renau, Basilio Fraguela, James Tuck, Wei Liu, Milos Prvulovic, Luis Ceze, Smruti Sarangi, Paul Sack, Karin Strauss, and Pablo Montesinos. SESC simulator, January 2005. `http://sesc.sourceforge.net/`.

[64] N. Riley and C. Zilles. Probabilistic counter updates for predictor hysteresis and stratification. In *International Symposium on High-Performance Computer Architecture*, 2006.

[65] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *International Symposium on Computer Architecture*, 2000.

[66] P. Salverda and C. Zilles. A criticality analysis of clustering in superscalar processors. In *International Symposium on Microarchitecture*, 2005.

[67] D. Sanchez and C. Kozyrakis. The ZCache: Decoupling ways and associativity. In *International Symposium on Microarchitecture*, 2010.

[68] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[69] S. W. Son, M. Kandemir, M. Karakoy, and D. Chakrabarti. A compiler-directed data prefetching scheme for chip multiprocessors. In *Symposium on Principles and Practice of Parallel Programming*, 2009.

[70] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *International Symposium on High-Performance Computer Architecture*, 2007.

[71] S. T. Srinivasan, R. D. Ju, A. R. Lebeck, and C. Wilkerson. Locality vs. criticality. In *International Symposium on Computer Architecture*, 2001.

[72] S. T. Srinivasan and A. R. Lebeck. Load latency tolerance in dynamically scheduled processors. In *International Symposium on Microarchitecture*, 1998.

[73] S. Subramaniam, A. Bracy, H. Wang, and G. H. Loh. Criticality-based optimizations for efficient load processing. In *International Symposium on High-Performance Computer Architecture*, 2009.

[74] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–25, January 2002.

[75] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. Jouppi. CACTI 5.3. `http://quid.hpl.hp.com:9081/cacti/`.

[76] E. Tune, D. Liang, D. M. Tullsen, and B. Calder. Dynamic prediction of

critical path instructions. In *International Symposium on High-Performance Computer Architecture*, 2001.

[77] S. P. Vanderwiel and D. J. Lilja. Data prefetch mechanisms. *ACM Computing Surveys*, 32(2):174–199, June 2000.

[78] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *International Symposium on Computer Architecture*, 1995.

[79] C.-J. Wu, A. Jaleel, M. Martonosi, S. C. Steely, and J. Emer. PACMan: Prefetch-aware cache management for high performance caching. In *International Symposium on Microarchitecture*, 2011.

[80] W. A. Wulf and S. A. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23(1):20–24, March 1995.

[81] T.-Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive branch prediction. In *International Symposium on Computer Architecture*, 1992.

[82] Z. Zhang, Z. Zhu, and X. Zhang. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *International Symposium on Microarchitecture*, 2000.