Improving Memory Scheduling via Processor-Side Load Criticality Information

Saugata Ghose Hyodong Lee José F. Martínez

Computer Systems Laboratory
Cornell University
Ithaca, NY 14853 USA

http://m3.csl.cornell.edu/

ABSTRACT

We hypothesize that performing processor-side analysis of load instructions, and providing this pre-digested information to memory schedulers judiciously, can increase the sophistication of memory decisions while maintaining a lean memory controller that can take scheduling actions quickly. This is increasingly important as DRAM frequencies continue to increase relative to processor speed. In this paper we propose one such mechanism, pairing up a processor-side load criticality predictor with a lean memory controller that prioritizes load requests based on ranking information supplied from the processor side. Using a sophisticated multicore simulator that includes a detailed quad-channel DDR3 DRAM model, we demonstrate that this mechanism can improve performance significantly on a CMP, with minimal overhead and virtually no changes to the processor itself. We show that our design compares favorably to several state-of-the-art schedulers.

1. INTRODUCTION

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 [9, 10] 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 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 pipeline. Using 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 [22] 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.

2. LOAD CRITICALITY: A PROCESSOR-SIDE PERSPECTIVE

Prior work in the field of instruction criticality has primarily focused on detecting criticality for all instructions within the processor. The generally accepted method of determining the critical path of program execution was proposed by Fields et al. [5]. A graph of dynamic instructions is constructed, modeling each instruction as three nodes: dispatch time, execution time, and commit time. Since an instruction cannot execute until dispatched, and cannot commit until executed, these nodes are connected together with directed edges to model the timing constraint between stages. The nodes of an instruction are also connected with those of other instructions using directed edges, to account for the dependencies between them within a processor. In this way, the critical-path model can capture in-order dispatch and commit, a finite ROB size, branch computations, and data dependencies. From the node times, the lengths of the dependent edges can be computed. We can then run a longest-path algorithm between the first dispatch and the final commit to determine the total application execution time. All edges that do not fall on this path are expected to have their delays masked by those that are on the path.

From the definition of criticality by Fields et al., traditional criticality prediction approaches tend to favor the selection of long-latency instructions as critical. While this can be quite useful for optimizing instructions in general, it does not differentiate amongst memory accesses. After some preliminary evaluation, we opted to exclude this predictor from our study.

ISCA '13 Tel Aviv, Israel

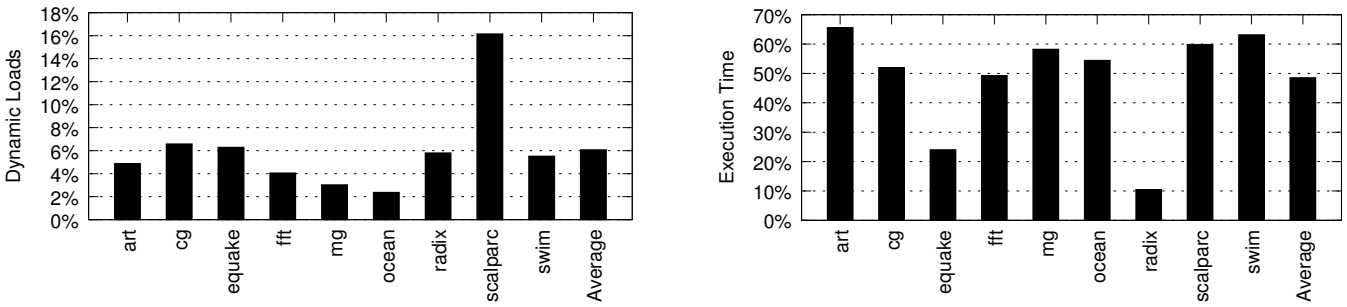

Figure 1: Percentage of dynamic long-latency loads that block at the ROB head (left), and percentage of processor cycles that these loads block the ROB head (right) when scheduled using FR-FCFS, averaged across all threads of each parallel application.

Much more relevant to our proposal is to confine the study of criticality to load misses, which are most directly relevant to the memory scheduler. We explore this criticality from two different broad perspectives argued in past work.

Based on Subramaniam et al. [29]—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 [29]. 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 Critical 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 study. 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.

Based on Runahead and CLEAR [3, 13, 18]—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.

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) [3, 18]. 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 [13]. Checkpoint support is still needed.

Targeting these loads to "unclog" the ROB could significantly reduce the processor critical path. Figure 1 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 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 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.

3. SCHEDULER IMPLEMENTATION

Unlike Subramaniam et al., neither Runahead nor CLEAR *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). For criticality based on this criterion, we must design a predictor that can inform the scheduler as soon as the load issues.

We propose a new hardware table, the *Commit Block Predictor* (CBP), that tracks loads which have previously blocked the ROB. Figure 2 shows how the CBP interacts within the processor. The CBP is simply an SRAM indexed by the PC. (As in branch prediction tables, we index the finite table with an appropriate bit substring of the PC, resulting in some degree of aliasing.) When a load instruction blocks the ROB head, the predictor is accessed and annotated accordingly. In our evaluation, we explore 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.) The next section goes into these options in more detail.

When a load instruction with the same (PC-based) CBP index is issued in the future, 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.

3.1 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. On average, each DRAM transaction queue contains more than one critical load for 30.1% of the overall execution time (in comparison, the queue contains at least one critical load 49.2% of the time). We can potentially im-

prove the performance of our scheduler by distinguishing amongst them. 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 behavior. 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.

3.2 Incorporating Criticality into FR-FCFS

We add our concept of load criticality into the FR-FCFS memory scheduler [22]. 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.

Figure 2 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 ROB stall time (the other proposed criteria are equally simple or simpler). When a load blocks at the head of the ROB, we make a copy of the PC bit substring, used to index the CBP table, and ROB sequence number. 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

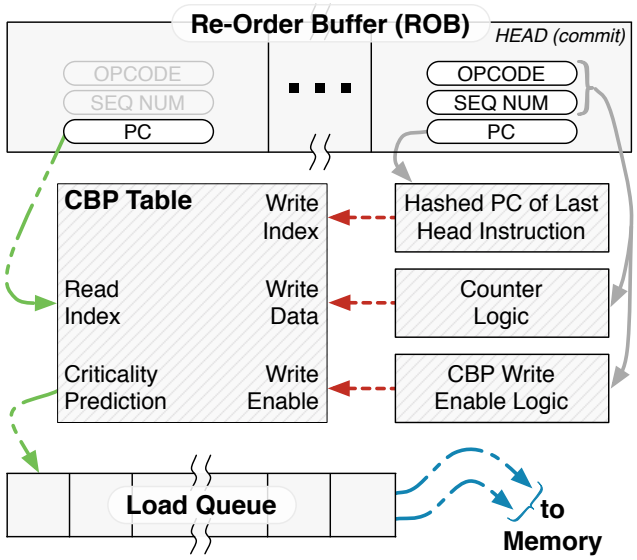

Figure 2: Overview of Commit Block Predictor (CBP) operation. Solid gray lines represent the per-cycle register counter and updates, dashed red lines illustrate a write to the CBP table (when a stalled load commits), green dash-dot-dot lines show the table lookup for loads to be issued, and blue dash-dot lines depict load issue to memory.

is incremented. (We shall discuss the size of this counter in Section 5.7.) If it is not equal, the saved PC is used to index the CBP table, which is a small, tagless direct-mapped array. The counter value is then written to the table, and the counter itself is reset.

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 5.7) 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 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 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, we prioritize by criticality magnitude first, using the sequence number 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.

Table 1: Core parameters.

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

Table 2: Simulated parallel applications and their input sets.

Data Mining		[19]
scalparc	Decision tree	125k pts., 32 attributes
NAS OpenMP		[2]
cg	Conjugate gradient	Class A
mg	Multigrid solver	Class A
SPEC OpenMP		[1]
art-omp	Self-organizing map	MinneSPEC-Large
equake-omp	Earthquake model	MinneSPEC-Large
swim-omp	Shallow water model	MinneSPEC-Large
SPLASH-2		[32]
fft	Fast Fourier transform	1M points
ocean	Ocean movements	514×514 ocean
radix	Integer radix sort	2M integers

Table 3: Parameters of the shared L2 cache and memory.

Shared L2 Cache Subsystem	
Shared L2 Cache	4 MB, 64 B block, 8-way
L2 MSHR Entries	64
L2 Round-Trip Latency	32 cycles (uncontended)
Micron DDR3-2133 DRAM	[15]
Transaction Queue	64 entries
Peak Data Rate	8.528 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: Multiprogrammed workloads. P, C, and M are processor-, cache-, and memory-sensitive, respectively [2, 7].

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

4. EXPERIMENTAL METHODOLOGY

Architectural Model—We assume an architecture that integrates eight cores with a quad-channel, quad-ranked DDR3-2133 memory subsystem. Our memory model is based on the Micron DDR3 DRAM specification [15]. The microarchitectural features of the baseline processor are shown in Table 1; the parameters of the L2 cache, and the DDR3 memory subsystem, are shown in Table 3. We implement our model using a modified version of the SESC simulator [20] 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 5.6. While not shown for brevity, the trends in the results reported in this paper were also observed using a slower DDR3-1066 model.

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 2.

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). 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.

5. 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 3.2). We then explore the impact of ranked criticality. Afterwards, we try to gain insight as to why the CBP-based predictors outperform the CLPT predictor proposed by Subramaniam et al. We also examine the impact on our scheduler's performance of adding a state-of-the-art prefetcher. We then quantify our hardware overhead. Finally, we compare our scheduler against AHB [8], MORSE-P [9, 16], PAR-BS [17], and TCM [12].

5.1 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

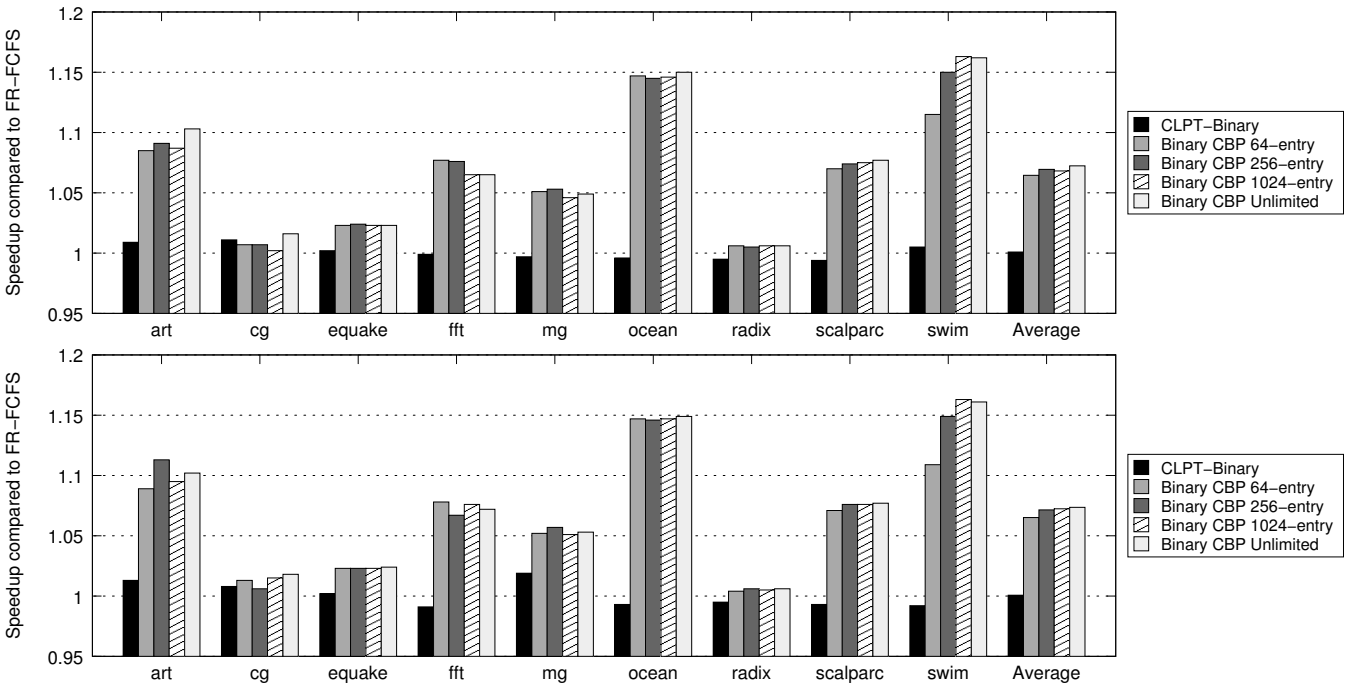

Figure 3: Speedups from *Binary* criticality prediction (sweeping over CBP table size) within the memory scheduler, using the *Crit-CASRAS* (top) and *CASRAS-Crit* (bottom) algorithms.

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.

5.2 First Take: Binary Criticality

We study the effects of adding criticality to the FR-FCFS scheduler, as proposed in Section 3.2. We evaluate our CBP tables, as well as the Critical Load Prediction Table (CLPT) mechanism proposed by Subramaniam et al. [29]. As discussed in Section 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 3 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 5.3, we will see that ranking the degree of criticality allows us to achieve greater performance benefits.

Figure 3 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 5.3.1. We also note that the *CLPT-Binary* predictor shows no appreciable speedup over FR-FCFS; we discuss this further in Section 5.3.3.

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 3.2, it is simpler to implement in hardware.

5.3 Ranking Degrees of Criticality

As we motivated in Section 3.1, 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, this time only using a 64-entry table. We also evaluate *CLPT-Consumers*, a ranked 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 *Block-Count* 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 *Last-StallTime* 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

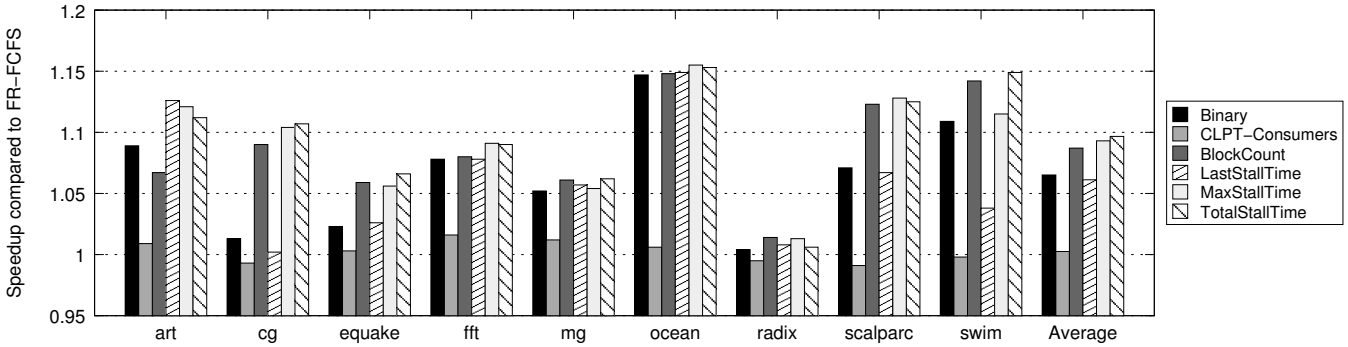

Figure 4: Speedups from ranking criticality within the memory scheduler, using the *CASRAS-Crit* algorithm. For the *CLPT-Consumers* predictor, the total consumer count for each load is used as the criticality magnitude. The CBP tables are 64 entries.

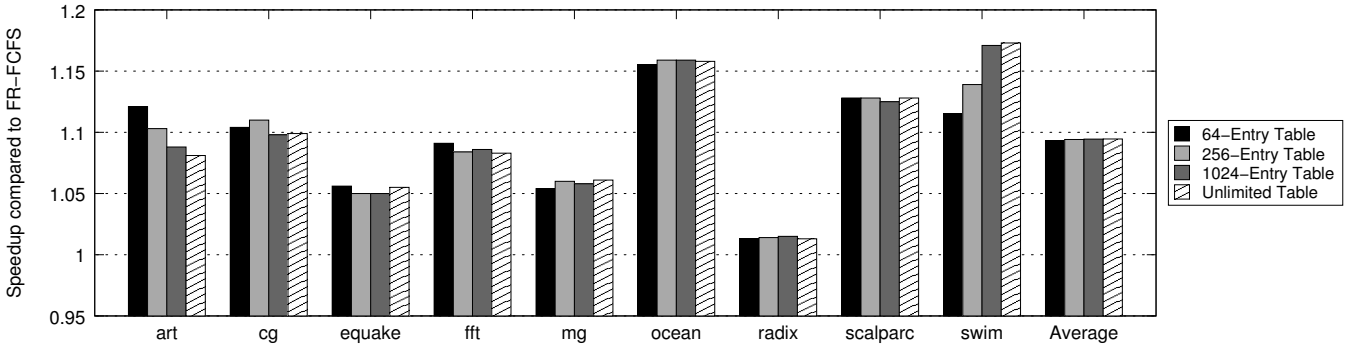

Figure 5: Speedups observed while sweeping table size for *MaxStallTime* criticality prediction. The *Unlimited Table* allows for an unrestricted number of entries into a fully-associative table.

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 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 [21], but we do not explore these.

5.3.1 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 3 shows the effect on performance for our binary criticality predictor, and Figure 5 shows performance for our *MaxStallTime* predictor. 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 5.4), 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 be-

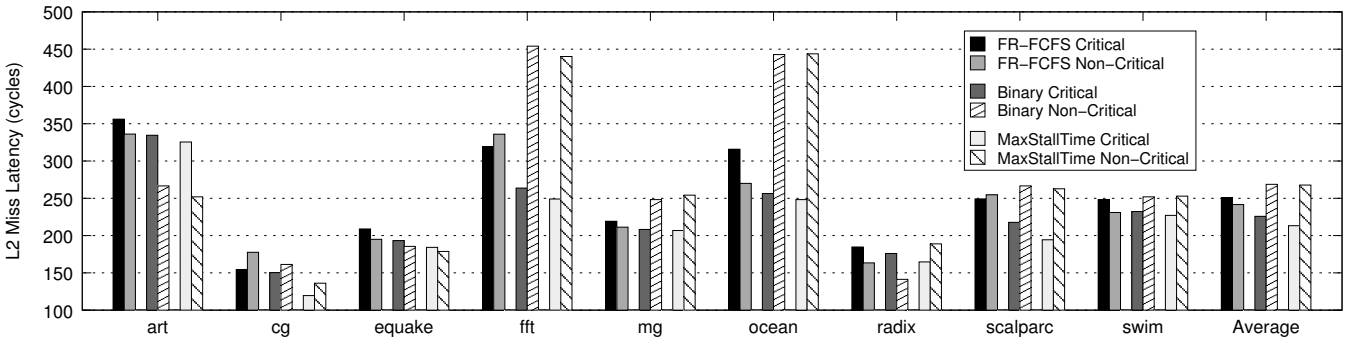

Figure 6: Average L2 miss latency for critical and non-critical loads within the memory scheduler, using the *CASRAS-Crit* algorithm and a 64-entry CBP table.

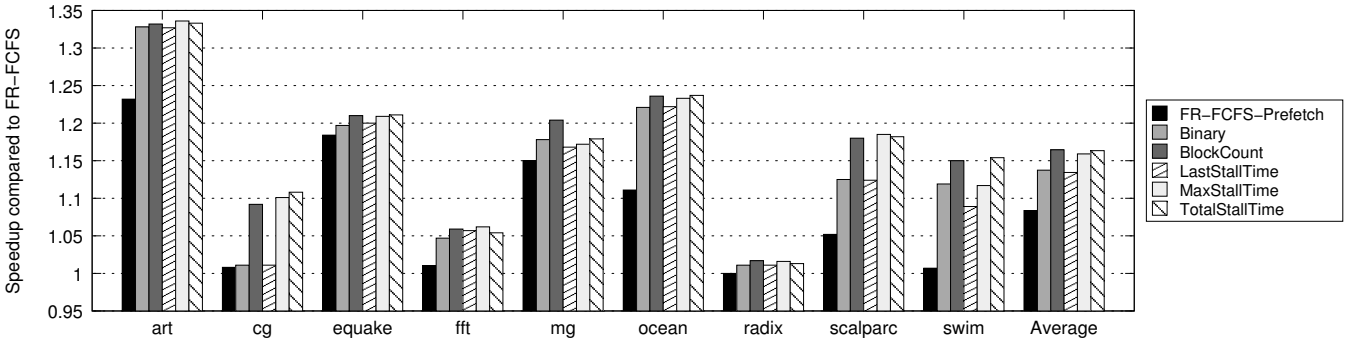

Figure 7: Speedups for FR-FCFS and our proposed criticality predictors with an L2 stream prefetcher, normalized to FR-FCFS without prefetching.

cause *art* has one of the smallest number of static critical loads out of our benchmarks, averaging 156 static critical loads per thread.

5.3.2 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 5, 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.

5.3.3 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. [29].

5.4 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 6. 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

Table 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

Table 6: State attributes used by *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

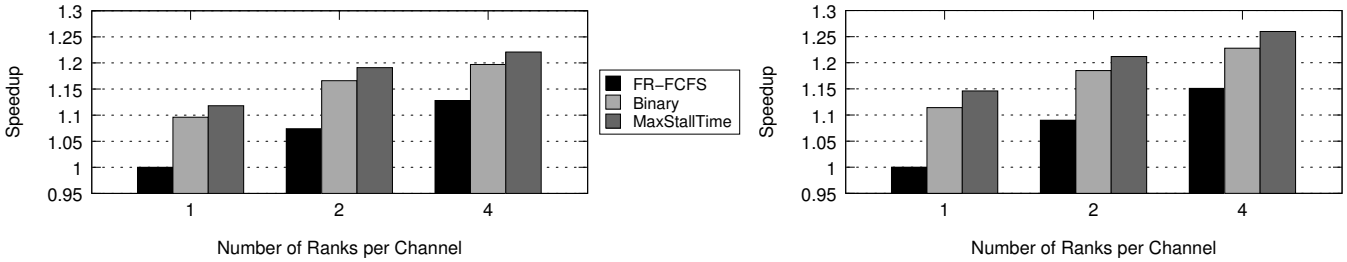

Figure 8: Sweep over number of ranks per channel, for DDR3-1600 (left) and DDR3-2133 (right) memory. Speedups are relative to a single-rank memory subsystem with an FR-FCFS scheduler.

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 completion 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 5.3.1, 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.

5.5 Effect of Prefetching

Memory prefetchers attempt to alleviate load miss serialization by exploiting some form of locality and predicting that, on a miss, a nearby cache line will also be read by an application in the near future. This has the potential to overlap with some of the latency-reducing benefits of criticality-aware memory scheduling, as it may reduce the number of loads that block the head of the ROB. To investigate this behavior, we implement a stream prefetcher inside the L2 cache [26]. We use an aggressive configuration of 64 streams, a prefetch distance of 64, and a prefetch degree of 4. Just adding the prefetch mechanism to baseline FR-FCFS yields a 8.4% speedup. Though this seems low compared to published results for sequential applications, prior work has shown that prefetching does not perform well as the number of threads of a parallel application increases [25]. The prefetcher fails as similar address streams generated by each parallel thread confuse the training agent.

Figure 7 shows the speedups observed if a 64-entry CBP is added to this *FR-FCFS-Prefetch* memory system. While the speedups are somewhat diminished, we still obtain notable improvements for our criticality prediction over FR-FCFS *with* prefetching (from 4.9% for *Binary* CBP up to 7.4% for *TotalStallTime*). We also increased the number of streams to 128 and 256 to make sure that this was not a limiting factor in performance. For the two sizes, we found the performance, both with and without the CBP, to be similar to that of the 64-stream prefetcher.

Notice that prefetchers only observe a global context, which may not assist all of the threads, depending on the application design. Criticality-aware FR-FCFS only adds prioritization metrics to data loads, and flags requests *for each thread*, providing us with greater potential to speed up all of the application threads.

5.6 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 8 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 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 9 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.

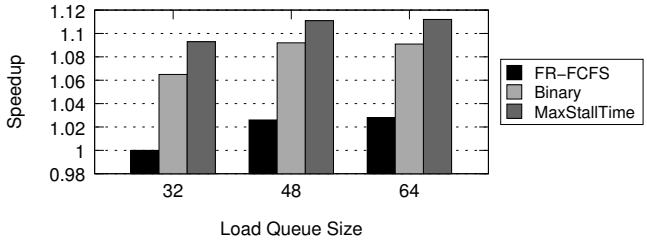

Figure 9: Sweep over load queue sizes. Speedups are relative to a 32-entry load queue with an FR-FCFS scheduler.

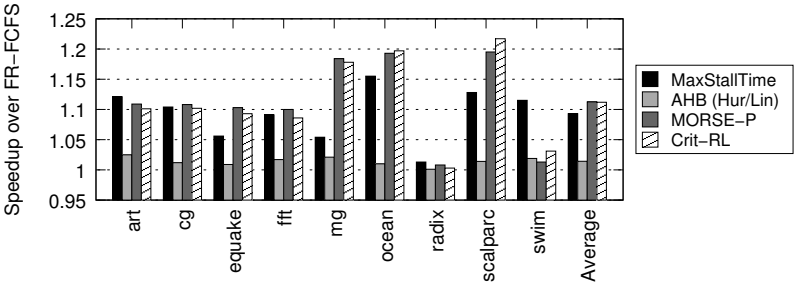

Figure 10: Performance of two state-of-the-art schedulers compared to our proposal. *Crit-RL* adds criticality to a self-optimizing memory controller, using the features listed in Table 6.

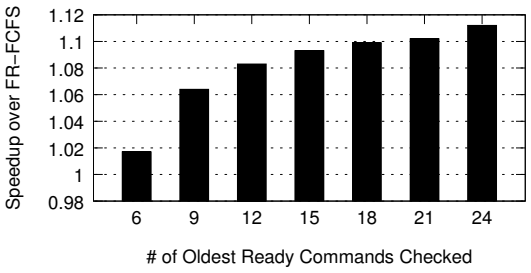

Figure 11: Performance of MORSE-P when restricting the number of ready commands that can be considered in a single DRAM cycle.

5.7 Storage Overhead

We now quantify the storage overhead required for the *CASRAS-Crit* algorithm described in Section 3.2. 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 equivalency 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 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 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.

5.8 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 [8], the fairness-oriented thread cluster memory (TCM) scheduler [12], and MORSE-P, a self-optimizing scheduler that targets parallel application performance [9, 16]. Table 7 summarizes the main differences between these schedulers. They are described in more detail in Section 6.2.

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.

5.8.1 Parallel Applications

Figure 10 shows the performance of our schedulers against 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 [16] from a total of 35 features, including all of the original MORSE-P features, on our training set (*fft*, *mg*, and *radix*). Table 6 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 improved performance. Promisingly, feature selection chose binary criticality first. However, the resulting controller, *Crit-RL*, only matches the performance of MORSE (see Figure 10). 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 [16], 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 [30] at a 32 nm technology) and the latency of the 32-adder tree and the 6-way comparator (approximately 700 ps [16]) 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.

Table 7: Comparison of various state-of-the-art memory schedulers with our proposed CBP-based schedulers.

Scheduler	AHB (Hur/Lin) [8]	TCM [12]	MORSE-P [9, 16]	*Binary* CBP	*MaxStallTime* CBP
Avg. Parallel Application Speedups (relative to FR-FCFS)	1.6%	0.6%	11.2%	6.5%	9.3%
Avg. Multiprogrammed Workload Weighted Speedups (relative to PAR-BS)	3.1%	1.9%	11.3%	5.2%	6.0%
Storage Overhead (for 8 cores, 4 memory controllers)	31 B	4816 B	DDR3-1066: 128 kB DDR3-2133: $\leq$ 512 kB†	109–301 B	1,357–1,805 B
Uses Processor-Side Information	No	No	Yes	Yes	Yes
Scales to High-Speed Memory	Yes	Yes	No	Yes	Yes
Works for Low Contention	Yes	No	Yes	Yes	Yes

†320 kB to match *MaxStallTime* CBP performance.

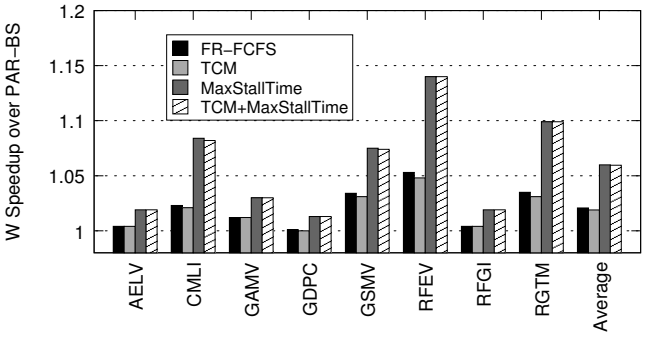

Figure 12: Weighted speedups for multiprogrammed workloads from adding ranked criticality. CBP table is 64 entries, and PAR-BS's marking cap is set to 5 [17].

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 triported 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 11 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.

5.8.2 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 [17]. We also show results for the more recent TCM proposal [12]. Our multiprogrammed workloads are four-application bundles (see Section 4). 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 [24] 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 [14], 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 12). We see similar speedups for our other ranking criticality predictors (not plotted here).

As a comparison, we have also implemented TCM [12], which attempts to balance system throughput (*weighted speedup*) with fairness (*maximum slowdown*). Figure 12 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%.

The apparent discrepancy from previous TCM results [12] 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 3). We show results for our CBP-based predictors under less aggressive systems in Section 5.6.

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 propose 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 12 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.

We expect that, in a high-contention memory design, we would see *TCM+MaxStallTime* performing at least as well as TCM (since

[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.

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.

6. RELATED WORK

6.1 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 [5]. Runahead [3, 18] and CLEAR [13] both propose mechanisms to alleviate the effect of blocking the head of the ROB at commit. Subramaniam et al. use the number of direct consumers to gauge the criticality of a load instruction [29]. These works are described in greater detail in Section 2.

One of the first works that examined criticality was by Srinivasan and Lebeck [28]. They studied 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% [27]. 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 the load is issued falls below a fixed threshold. However, there is significant overhead for their table structure, as it maintains an array of bits, the size of the LSQ, for each ROB entry.

Fisk and Bahar use criticality to filter out non-critical loads into a small buffer akin to the victim cache [6]. Criticality is tracked similarly to Srinivasan and Lebeck, where a low issue rate is used to determine criticality. Fisk and Bahar also use the number of dependencies as a second measure of criticality. 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.

Tune et al. use a number of statistics to determine whether an instruction is critical [31], 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 [23]. 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 [21].

6.2 Memory Scheduling

MORSE is a state-of-the-art self-optimizing memory scheduler extending on the original proposal by İpek et al. [9, 16]. 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 in fact fall on the critical path of processor execution, as we have shown happens.

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 [8].

The most relevant to our work are those that focus on scheduling critical threads. Thread cluster memory (TCM) scheduling [12] classifies threads into either a latency-sensitive or bandwidth-sensitive cluster. Latency-sensitive threads are prioritized over 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 [10]. Threads with low memory-level parallelism (MLP) are ranked higher than those with high MLP, which are themselves ranked higher than prefetch requests. Though the authors refer to this memory-based ranking as criticality, it is different than our concept of criticality, which is based on instruction behavior inside the processor.

Other work in the area of memory scheduling has targeted a variety of applications. Ebrahimi et al. demonstrate a memory scheduler for parallel applications [4]. 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 [17], ATLAS [11]) target reducing memory latency strictly in the context of multiprogrammed workloads.

7. CONCLUSION

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 [22], 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 pre-digested processor-side information, provide an essential balance between improved scheduling decisions and implementability.

8. ACKNOWLEDGEMENTS

We thank Gurindar Sohi for feedback on our group's MORSE work, which inspired us to carry out this study. This research was

supported in part by NSF CAREER Award CCF-0545995, NSF Award CNS-0720773, and gifts from IBM and Intel.

9. REFERENCES

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

[2] 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.

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

[4] 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.

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

[6] 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.

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

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

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

[15] 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`.

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

[17] 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.

[18] 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.

[19] 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.

[20] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC simulator, January 2005. `http://sesc.sourceforge.net/`.

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

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

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

[24] 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.

[25] 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.

[26] 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.

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

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

[29] 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.

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

[31] 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.

[32] 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.