Parallelism-Aware Memory Interference Delay Analysis for COTS Multicore Systems

Heechul Yun†, Rodolfo Pellizzoni⋆, Prathap Kumar Valsan†
† University of Kansas, USA. {heechul.yun, prathap.kumarvalsan}@ku.edu
⋆ University of Waterloo, Canada. rpellizz@uwaterloo.ca

***Abstract*—In modern Commercial Off-The-Shelf (COTS) multicore systems, each core can generate many parallel memory requests at a time. The processing of these parallel requests in the DRAM controller greatly affects the memory interference delay experienced by running tasks on the platform.**

In this paper, we present a new parallelism-aware worst-case memory interference delay analysis for COTS multicore systems. The analysis considers a COTS processor that can generate multiple outstanding requests and a COTS DRAM controller that has a separate read and write request buffer, prioritizes reads over writes, and supports out-of-order request processing. Focusing on LLC and DRAM bank partitioned systems, our analysis computes worst-case upper bounds on memory-interference delays, caused by competing memory requests.

We validate our analysis on a Gem5 full-system simulator modeling a realistic COTS multicore platform, with a set of carefully designed synthetic benchmarks as well as SPEC2006 benchmarks. The evaluation results show that our analysis produces safe upper bounds in all tested benchmarks, while the current state-of-the-art analysis significantly under-estimates the delays.

I. Introduction

In modern Commercial Off-The-Shelf (COTS) multicore systems, many parallel memory requests can be sent to the main memory system at any given time for the following two reasons. First, each core employs a variety of techniques—such as non-blocking cache, out-of-order issues, and speculative execution—to hide memory access latency. These techniques allow the core to continue to execute new instructions while it is still waiting for memory requests of previous instructions to be completed. Second, multiple cores can run multiple threads, each of which generates memory requests.

These parallel memory requests from the processor put high pressure on the main memory system. To deliver high performance, modern DRAM consists of multiple resources called banks that can be accessed in parallel. For example, a typical DRAM chip has 8 banks, supporting up to 8 parallel accesses [20]. To efficiently utilize the available bank level parallelism, modern COTS DRAM controllers employ sophisticated techniques such as out-of-order request processing, overlapped request dispatches, and interleaved bank mapping [26], [21], [5].

While parallel processing of multiple memory requests generally improves overall memory performance, it is very difficult to understand precise memory performance especially when multiple applications run concurrently, because each memory request is more likely to be interfered by other requests. In analyzing memory latency on COTS systems, many early works modeled DRAM as a single resource, having a constant access time, which is arbitrated by a simple round-robin policy [27], [36]. Recently, Kim et al. proposed more realistic analysis model which considers DRAM banks and the FR-FCFS [26] scheduling policy, which is commonly used policy in COTS systems. The analysis, however, assumes that each core can only generate one outstanding memory request at a time, while in many modern COTS multicore processors, especially high-performance ones such as Freescale P4080 and ARM Cortex-A15, a core can generates multiple outstanding requests [8], [2]. For example, each Cortex-A15 core can generate up to six outstanding cache-line fill (memory read) requests, which, in turn, can generate additional write-back (memory write) requests [2]. Furthermore, the analysis does not consider the fact that COTS DRAM controllers prioritize reads over writes and process writes opportunistically [5].

In this work, we present a parallelism-aware memory interference delay analysis. We model a COTS DRAM controller that has a separate read and write request buffer. Multiple outstanding memory requests can be queued in the buffers and processed out-of-order to maximize memory performance. Also, reads are prioritized over writes in our model. These features are commonly found in modern COTS multicore systems and crucially important in understanding memory interference. As such, we claim our system model well represents real COTS multicore platforms. To minimize interference, we focus on a system in which the LLC and DRAM banks are partitioned. This is easily achievable on COTS multicore systems via software [18], [29]. Our analysis, then, provides an analytic upper bound on the worst-case memory interference delay for each read memory request of the task under analysis. Note that the derived bound does not require any assumption on the interfering tasks' memory access patterns. However, we also show that if the number of read and write requests generated by each core is known, then the analytical bounds can be significantly improved.

We evaluate the proposed analysis on the Gem5 full-system simulator [4], modeling a realistic COTS multicore platform based on ARM architecture, with a set of synthetic benchmarks as well as SPEC2006 benchmarks. The synthetic benchmarks are specially designed to simulate high memory-interference delay. The results show that our analysis provides safe upper bounds in all tested benchmarks while [13] significantly under-estimates the delays, by up to 62%, in the tested benchmarks.

The remaining sections are organized as follows: Section II provides background on COTS multicore systems and LLC and DRAM bank partitioning techniques. Section III discusses the state-of-art memory interference delay analysis. We present

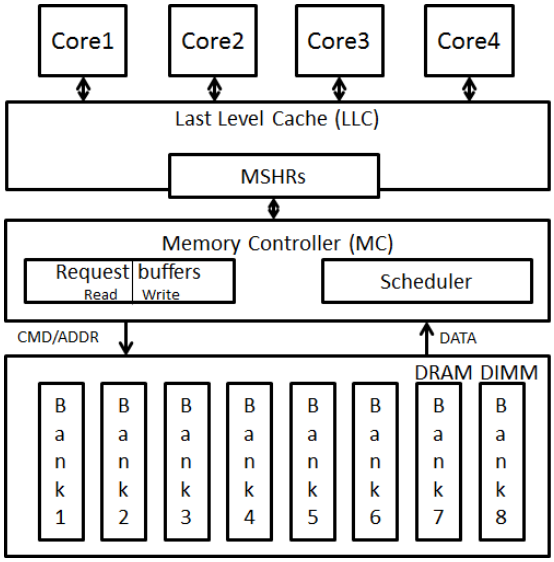


Fig. 1: Modern COTS multicore architecture

our analysis in Section IV and provide evaluation results in Section V. Section VI discusses related work. Finally, we conclude in Section VII.

II. Background: Modern COTS Multicore Systems

A modern COTS multicore system, shown in Figure 1, supports a high degree of memory level parallelism through a variety of architectural features. In this section, we provide some background on important architectural features of modern COTS multicore systems, and review existing software based resource partitioning techniques.

A. Non-blocking Cache and MSHR

At the cache level, non-blocking caches are used to handle multiple simultaneous cache-misses. This is especially crucial for the shared last level cache (LLC), as it is shared by all cores. The state of the outstanding memory requests are maintained by a set of miss status holding registers (MSHRs). On a cache-miss, the LLC allocates a MSHR entry to track the status of the ongoing request and the entry is cleared when the corresponding memory request is serviced from the main memory. As such, the number of MSHRs effectively determines the maximum number of outstanding memory requests directed to the DRAM controller.

B. DRAM Controller

The DRAM controller receives requests from the LLC (or other DMA devices) and generates DRAM specific commands to access data in the DRAM. Modern DRAM controllers often include separate read and write *request buffers* and *prioritize reads* over writes because writes are not on the critical path for program execution. Write requests are buffered on the write buffer of the DRAM controller and serviced when there are no pending read requests or the write queue is near full [21], [5]. The DRAM controller and the DRAM module are connected through a command/address bus and a data bus. Modern DRAM modules are organized into ranks and each rank is divided into multiple *banks*, which can be accessed in *parallel* provided that no collisions occur on either buses. Each bank comprises a row-buffer and an array of storage cells organized as *rows* and *columns*. In order to access the data stored in a DRAM row, an activate command (*ACT*) must be issued to load the data into the row buffer first (*open the row*) before it can be read or written. Once the data is in the row buffer, any numbers of subsequent read or write commands (*RD, WR*) can be issued to access data in the row. If, however, a request wishes to access a different row from the same bank, the row buffer must be written back to the array (*close the row*) with a pre-charge command (*PRE*) first before the second row can be activated.

C. Memory Scheduling Algorithm

Due to hardware limitations, the memory device takes time to perform different operations and therefore timing constraints between various commands must be satisfied by the controller. The operation and timing constraints of memory devices are defined by the JEDEC standard [12]. The key facts concerning timing constraints are: 1) the latency for accessing a closed row is much longer than accessing a row that is already open; 2) different banks can be operated in parallel since there are no long timing constraints between banks. To maximize memory performance, modern DRAM controllers typically use a First-Ready First-Come-First-Serve (FR-FCFS) [26] scheduling algorithm that prioritizes: (1) Ready column access commands over row access commands; (2) Older commands over younger commands. This means that the algorithm can process memory requests in out-of-order of their arrival times. Note that a DRAM command is said to be *ready* when it can be scheduled immediately as it satisfies all timing constraints imposed by previously scheduled commands and the current DRAM status.

D. DRAM Bank and Cache Partitioning

In COTS systems, all banks are shared by all cores in the system. This can cause unpredictable delays due to bank conflicts. For example, if two applications running in parallel on different cores access two different rows in the same bank, they can force the memory controller to continuously pre-charge the row buffer and open a new row every time an access is performed. This loss of row locality can result in a much degraded row hit ratio and thus corresponding latency increases for both applications.

Software bank partitioning [35], [18], [29] can be used to avoid the problems of shared banks. The technique leverages the page-based virtual memory system of modern operating systems and allows us to allocate memory to specific DRAM banks. Each core, then, can be assigned to use its private DRAM banks, effectively eliminates bank sharing among cores without requiring any hardware modification. Similar techniques can also be applied to partition the shared LLC as explored in [38], [19], [6], [31], [17]. It is shown that partitioning DRAM banks and LLC substantially reduce memory interference among the cores [35].

However, the LLC cache space and DRAM banks are not the only shared resources contributing to memory interference. Most notably, at the DRAM chip level, all DRAM banks fundamentally share the common command and data bus.

Hence, contention in the buses can become a bottleneck. Furthermore, as many memory requests can be buffered inside the DRAM controller's request buffers, its scheduling policy can greatly affect memory interference delay.

Goal: The goal of this paper is to analyze the worst-case memory interference delay in a cache and DRAM bank partitioned system, focusing mainly on delay in the DRAM controller and command and data bus between the controller and the DRAM module.

III. The State of the Art Delay Analysis and the Problem

In this section, we first review a state of art memory interference delay analysis for COTS memory systems [13], which was proposed by Kim et al., and investigate some of its assumptions that are not generally applicable in modern COTS multicore systems.

The analysis models a modern COTS memory system in great detail. While there has been a similar effort in the past [32], this is the first work that considers the DRAM bank level request reordering effect (i.e., out-of-order execution of young row-hit column requests over older row-miss requests). Here, we briefly summarize the assumed system model and part of the memory interference delay analysis, relevant for the purpose of this paper.

The system model assumes a single channel DRAM controller and a DDR memory module. The DRAM controller uses FR-FCFS scheduling algorithm. At the high level, the analysis computes the worst-case memory interference delay of the task under analysis [1] τ_i either (1) as a function of number of memory requests H_i of the task (referred as request driven approach) or (2) as a function of the number of memory requests generated by the other tasks on different cores (referred as job driven approach)—it takes the minimum of the two—similar to prior work [36]. The unique characteristics of the analysis is that it considers both inter-bank and intra-bank (including request reordering) memory interference delay. For the purpose of this paper, however, we focus on their inter-bank delay analysis that assumes each core is assigned *dedicated DRAM bank partitions*.

The analysis assumes that each memory request of τ_i is composed of PRE, ACT, and RD/WR DRAM commands (i.e., a row-miss) and each of the command can be delayed by DRAM commands generated by other tasks on different cores, due to inter-bank timing constraints imposed by the JEDEC standard [12]. These timing constraint-imposed inter-bank delay for PRE, ACT, and RD/WR commands are denoted as L_{inter}^{PRE}, L_{inter}^{ACT}, and L_{inter}^{RW}, respectively.

One major assumption of the analysis is that each core can generate only *one outstanding memory request* to the DRAM controller. Based on this assumption, the worst-case per-request inter-bank memory interference delay on a core p, RD_p^{inter}, is simply expressed by $RD_p^{inter} = \sum_{\forall q:q\neq p}(L_{inter}^{PRE} + L_{inter}^{ACT} + L_{inter}^{RW})$. Finally, the total memory interference delay of a task is calculated by multiplying RD_p^{inter} to the number of total LLC misses H_i of τ_i.

[1]The analysis can also be applied to a preemptive fixed-priority system by considering the overall interference delay of the busy interval for the task under analysis.

The analysis, however, has two main problems when it is applied to modern COTS multicore systems. On the one hand, it is overly *optimistic* as it assumes that each interfering core can only generate one outstanding memory request at a time. Hence, it essentially limits the maximum number of competing requests to the number of cores in the system. However, this is not true for many modern COTS multicore processors as each core can generate many parallel memory requests at a time. Because DRAM performance is much slower than CPU performance, these requests are queued inside the DRAM controller and can aggravate the overall delay.

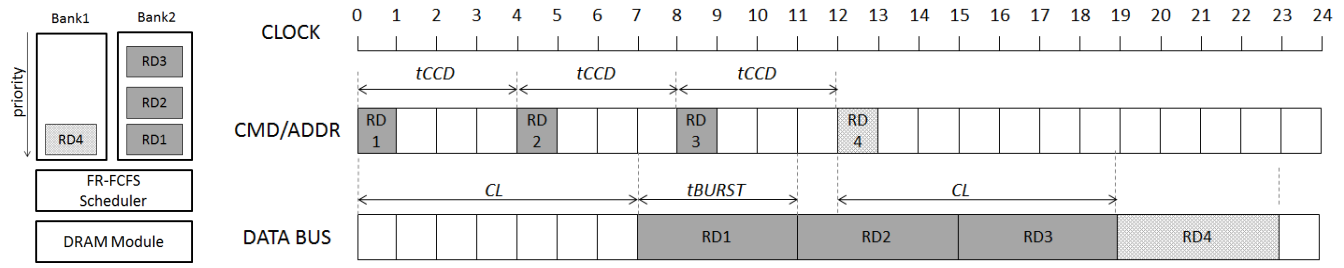
Figure 2 illustrates this problem that can occur in modern COTS systems. In the figure, three parallel requests RD1, RD2, and RD3 are already in the command queue for Bank2, when the request RD4 has arrived at Bank1. Note that the DRAM commands are numbered in the order of their arrival times in the DRAM controller. At memory clock 0, both RD1 and RD4 are ready, but RD1 is scheduled as FR-FCFS policy prioritizes older requests over younger ones. Similarly, RD2 and RD3 are prioritized over RD4 at time 4 and 8, respectively. At other times such as at clock 1, RD4 cannot be scheduled due a channel timing constraint ($tCCD$), even though it is ready w.r.t. the Bank1.

On the other hand, it is also overly *pessimistic* as a memory request—composed of PRE, ACT, and RD/WR DRAM sub-commands—is assumed to suffer inter-bank interference for each sub-command, while in reality the delays of executing sub-commands of a memory request are not additive on efficient modern COTS memory controllers. Figure 3 shows such a case. In the figure, each bank has one row miss DRAM request. Hence, each has to open a new row with an ACT command followed by a RD command. Following the FR-FCFS policy, ACT1 on Bank2 is executed first at clock 0. Even though ACT2 is targeting a different bank, it is not scheduled immediately due to the required minimum separation time $tRRD$ between two inter-bank ACT commands; i.e., for the case in the figure, we have $L_{inter}^{ACT} = tRRD$. At clock 4, however, ACT2 can be issued even though ACT1 on Bank2 is still in progress. In other words, the two memory requests are *overlapped*. Similarly, RD2 cannot be scheduled immediately after RD1 due to data bus conflict, i.e., $L_{inter}^{ACT} = tBURST$. But since the requests are overlapped and ACT2 was scheduled 4 clock cycles after ACT1, when RD2 is finally issued at time 11, there is no extra inter-bank delay other than the initial delay of $tRRD$. In summary, the overall delay suffered by the request of Bank1 is equal to $\max\left(L_{inter}^{ACT}, L_{inter}^{RD}\right)$ rather than $L_{inter}^{ACT} + L_{inter}^{RD}$.

From the point of view of WCET analysis, the former problem is more serious as it undermines the safety of the computed WCET. We experimentally validated the former problem on our test platform with carefully engineered synthetic tasks, as we will detail in Section V-B.

IV. Parallelism-Aware Memory Interference Delay Analysis

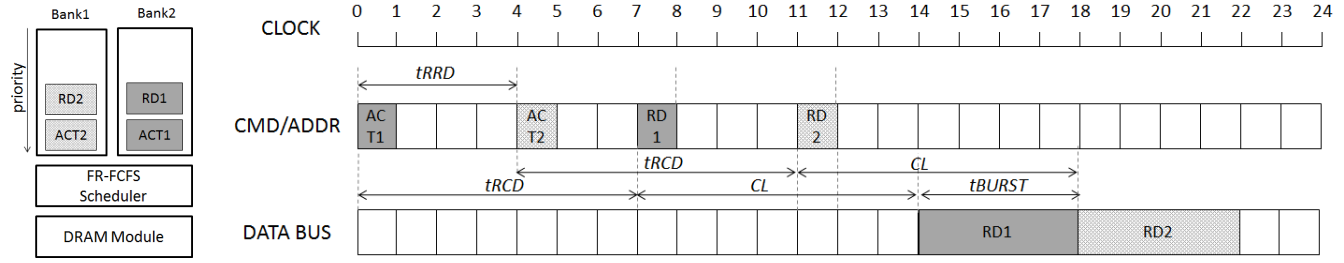
We start by formalizing the assumptions of our analysis. We consider a modern multicore architecture described in Section II. Specifically, there are N_{proc} identical cores in a single processor chip. The processor contains per-core private



(a) Initial bank queue status

(b) Timing diagram under FR-FCFS schedule

Fig. 2: Inter-bank delay caused by three previously arrived outstanding requests. (DRAM commands are numbered according to their arrival time to the DRAM controller.)



(a) Initial bank queue status

(b) Timing diagram under FR-FCFS schedule

Fig. 3: Inter-bank delay caused by a previously arrived row-miss request.

caches and a shared LLC. Both caches are non-blocking and the numbers of MSHRs in the caches determine the local and global limit of outstanding memory read requests. We assume the caches employ a write-back write-allocate policy. Hence a write to DRAM only occurs when there is a cache miss (either read or write) in the LLC that evicts a modified cache-line in the cache, and program execution can proceed without waiting for the write request to be processed in the DRAM. Therefore, for the analysis purpose, we only consider memory interference delay imposed to each read request of the task under analysis. Note that the number of DRAM read requests is equal to the number of LLC misses because, in a write-back write-allocate cache, a write miss also generates a DRAM read request to allocate the line in the cache and then write to it. Finally, similarly to [13] and other related work, we assume that memory delay is additive to a task's computation time. We discuss this assumption in more details in Appendix.

On the DRAM controller side, we assume a modern DRAM controller that supports the FR-FCFS scheduling policy [26], [30] and is connected to a DRAM module. At each memory clock tick, we assume a highly efficient FR-FCFS scheduler that picks the highest priority ready command among all requests and can overlap multiple requests simultaneously as long as DRAM bank and channel level timing constraints and the FR-FCFS priority rules are satisfied [21]. We model FCFS priorities by assigning timestamps to commands, with the earliest command having highest priority; the timestamp of a command corresponds to the arrival time of the generating request, hence, if a request for a close row generates a PRE, ACT and RD or WR command, all three commands have the same timestamp. We assume open-page policy is used for bank management to maximize data locality. For simplicity, our analysis assumes a single rank DRAM module, but it can be extended to consider a multi-rank DRAM module.

All previously mentioned assumptions closely follow common behavior of COTS DRAM controllers [21]. Hence, based on such assumptions, in Section IV-A we first provide a bound on the delay suffered by a read request of the task under analysis based on the interference caused by other read requests only. Incorporating the effects of write requests, which are not on the critical path of the program, requires us to specify additional assumptions. In general, COTS DRAM controllers have both a read request buffer and a write request buffer, and prioritize reads over writes; both read and write requests are processed in *batches* of consecutive requests to amortize the cost of the data bus turnaround delay [5]. The exact batch processing policy of each controller may vary; for the purpose of constructing a sound worst-case delay bounds, we consider a typical *watermark* approach [5], [21]. The policy defines a high watermark and a low watermark value for the number of write requests in the write buffer, as well as a write batch length N_{wd}. If the read buffer is empty, the controller uses the low watermark; otherwise, it uses the high watermark [2]. The controller starts processing a write batch if the number of queued writes is higher than or equal to the

[2]The memory controller of the Gem5 simulator [10], which we use in Section V, also follows the same watermark policy in processing write requests.

current watermark, and it keeps servicing write requests until it has issued at least N_{wd} writes. Finally, queued writes are also serviced according to FR-FCFS arbitration. The delay analysis extended with write batch processing is presented in Section IV-B.

TABLE I: System model parameters.

CPU	N_{proc}	Number of cores	4
	N_{rq}	Maximum no. of prior read requests	18
DRAM controller	N_{wd}	Minimum writes per batch	18
	Q_{read}	Read-buffer size	64
	Q_{write}	Write-buffer size	64
	W_{high}	High watermark value	54
	W_{low}	Low watermark value	32
DRAM	$tRCD$	Row activation time	8
	$tBURST$	Data burst duration	4
	$tRRD$	Activate to activate delay	6
	$tFAW$	Four activate windows	27
	tRC	Row cycle time	30

Table I shows the parameters used in the analysis, together with their values used in the simulator in Section V [3] [4]. N_{rq} denotes the maximum number of prior read requests queued in the read request buffer, which is determined by the number of entries in the read MSHR, size of read request buffer and number of outstanding requests per core. Consistently with the described watermarking approach for write handling, we assume $W_{high} > W_{low} \geq N_{wd}$ (i.e., when we reach a watermark there are enough queued writes to complete a batch of length N_{wd}), and $Q_{write} - W_{high} < N_{wd}$ (i.e., even if the buffer is full, performing N_{wd} writes reduces the number of queued writes below the high watermark). We also assume that $tBURST = 4, tRRD \geq 4$ and $tFAW \geq 4 \cdot tRRD$, which is true for all modern DDR devices.

Finally, we assume DRAM banks and the LLC space are partitioned on a per-core basis. In other words, each core is assigned its own private DRAM banks and LLC space. This can be easily achieved by using software partitioning techniques on COTS systems [35], [18].

In summary, our system model significantly differs from [13] in that (1) it models multiple parallel memory requests buffered in the DRAM controller, and (2) it maintains separate read and write request queues in the DRAM controller and reads are prioritized over writes.

A. Read Batch Delay Analysis

We now present our analysis that considers parallel memory requests. As mentioned in the previous section, write memory requests are not in the critical path of program execution in modern COTS systems. Hence, our primary concern is memory interference delay to read requests of the task under analysis. More in detail, in this section we compute an upper bound on the delay that a newly arrived read request (*request under analysis*) can suffer due to a batch of other read requests only. We discuss the case of write interference in Section IV-B.

[3]The DRAM parameters in Table I are based on LPDDR2 memory commonly used with ARM processors. For ease of explanation, figures throughout the paper are drawn using the timing parameters for DDR3 1066 memory, which are shorter and result in more compact drawings. The derived analytical bounds are applicable to any modern DDR device.

[4]We do not consider the auto-refresh operation in our analysis because it periodically occurs at a relatively long fixed interval and its impact to the overall memory interference is small (<2%), as discussed in [3], [13].

Given that the maximum number of prior queued read request is N_{rq}, the worst case delay $L(N_{rq})$ is produced when the request under analysis has the largest timestamp of all read requests in the queue. Furthermore, since a read request might target a close row, in the worst case the request under analysis is composed of a PRE, an ACT and a RD command. Therefore, we need to compute the delays L^{PRE}, L^{ACT} and L^{RD} suffered by the PRE, ACT and RD commands, respectively. As noted in Section III, the challenge is, then, to compose the three delays by taking into account the overlapping of memory requests, such that the overall delay is obtained as the maximum of the per-command delays rather than the sum.

We can formalize this key idea by using the same *delay composition* strategy as in [11]. In details, we model each request as a *job* executed in sequence on three *pipeline stages*; the stages model the interference of PRE commands on other PRE commands, ACT on ACT, and RD on RD. Note that while a request can be composed of only one (RD), two (ACT and RD), or three commands (PRE, ACT and RD), the commands are always executed in the same sequence according to the same timing constraints for all requestors. Furthermore, the priority of a job remains the same for all three stages, since we assume that all commands of a given request have the same timestamp. Finally, jobs are executed non-preemptively, since commands and their related timing constraints cannot be revoked once issued. Hence, the following main result applies:

Theorem 1: The delay caused by an interfering request to the request under analysis is upper bounded by the maximum delay on a single stage, i.e., either the delay caused by the interfering PRE command to the PRE command under analysis, or ACT to ACT, or RD to RD.

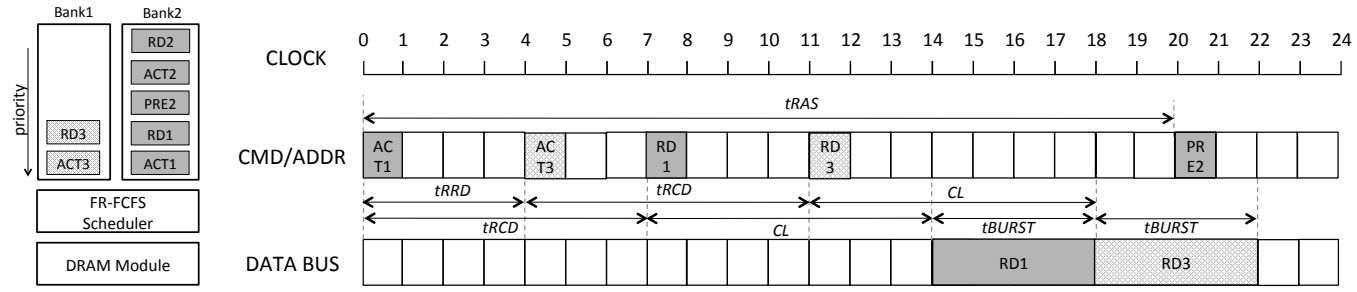
Proof: The theorem follows from the Non-Preemptive Pipeline Delay Composition Theorem in [11], where memory read requests are modeled as jobs executing on three sequential stages (one each for PRE, ACT and RD), and the per-job priority among ready jobs, which is fixed over all stages, is equivalent to the timestamp for the commands of the request [5]. ■

By virtue of Theorem 1, we can compute the overall delay $L(N_{rq})$ to the request under analysis in the following way: we assume that each interfering request causes delay on a single stage, with the numbers of interfering PRE, ACT and RD commands being N_{PRE}, N_{ACT} and N_{RD}, respectively; due to Theorem 1, it must hold $N_{PRE}+N_{ACT}+N_{RD} = N_{rq}$. We then compute upper bounds to the delays $L^{PRE}(N_{PRE})$, $L^{ACT}(N_{ACT})$, $L^{RD}(N_{RD})$ caused on the PRE, ACT and RD command under analysis, respectively, and we compute the overall delay $L(N_{rq})$ as follows:

$$L(N_{rq}) = \max_{N_{PRE}+N_{ACT}+N_{RD}=N_{rq}} \big(L^{PRE}(N_{PRE}) + + L^{ACT}(N_{ACT}) + L^{RD}(N_{RD})\big). \quad (1)$$

Before we compute the upper bounds, we need to make some fundamental observations. First of all, each set of

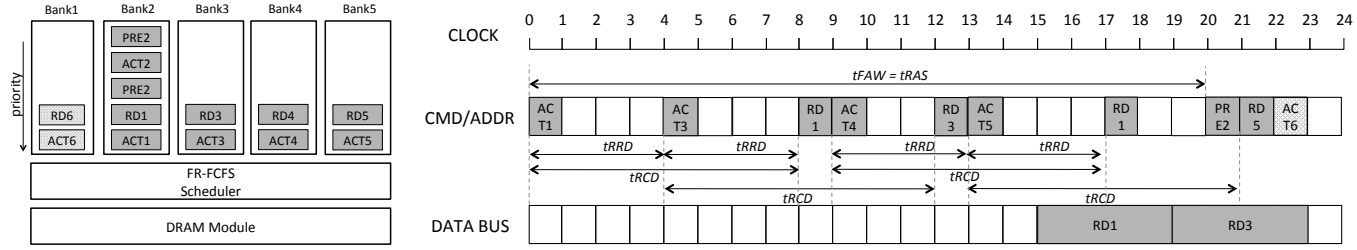
[5]Note that we do not need to consider the per-stage delay included in the theorem, since our goal is to compute the delay suffered by the request under analysis rather than its latency. The blocking term due to lower priority, meaning higher timestamp, requests is included in the constant delays in Equations 3, 4.



(a) Initial bank queue status

(b) Timing diagram under FR-FCFS schedule

Fig. 4: Out-of-order ACT processing due to bank timing constraints.



(a) Initial bank queue status

(b) Timing diagram under FR-FCFS schedule

Fig. 5: Command Bus Conflicts for ACT Commands.

interfering PRE, ACT or RD commands must be executed continuously such that the corresponding command under analysis cannot be scheduled until all interfering commands have completed. As an example, consider Figure 4 for the delay caused by ACT commands. While Bank2 has two ACT commands with lower timestamp than the ACT under analysis (ACT3), only ACT1 can interfere with ACT3 because ACT3 can be issued at time 4, while ACT2 must wait for PRE2 to be issued. Furthermore, after ACT3 is issued, the request under analysis can immediately start the $tRCD$ timer and then attempt to issue a RD command; hence, further ACT commands cannot delay the request under analysis by triggering ACT-related timing constraints ($tRRD$ and $tFAW$). However, these later-scheduled commands could still cause command bus contention, since the command bus is shared among the three stages. To better understand this situation, consider Figure 5, which shows a pattern for 4 interfering ACT commands of Banks 2 to 5. Since the $tFAW$ constraint is activated by four consecutive ACT commands and $tFAW \geq 4 \cdot tRRD$ for all memory devices, one could assume that the worst case delay is equal to $tFAW$. In reality, as Figure 5 shows, the ACT command under analysis (ACT6) can be further delayed due to command bus contention. First at time 20, since Bank 2 has a PRE2 command with smaller timestamp that ACT commands of other banks and PRE2 must be issued rather than ACT6. Then, since the $tRCD$ constraint for RD5 elapses at time 21, RD5 is also issued before ACT6, resulting in a total delay of 22 time units rather than 20. Since in general considering the exact effect of command bus conflicts is extremely difficult, in the rest of the section we pessimistically assume that a command always suffers worst case bus conflicts independently of the number of other interfering commands.

Based on the intuition above, we can show the following delay upper bounds:

$$L^{PRE}(N_{PRE}) = 2 \cdot N_{PRE}, \quad (2)$$

$$\begin{aligned} L^{ACT}(N_{ACT}) &= tFAW - 3 \cdot tRRD - 1 + \\ &+ \max\big(N_{ACT} \cdot (tRRD + 2), \\ &\quad \lfloor N_{ACT}/4 \rfloor \cdot (tFAW + 2) + \\ &+ (N_{ACT}\%4) \cdot (tRRD + 2)\big), \end{aligned} \quad (3)$$

$$\begin{aligned} L^{RD}(N_{RD}) &= tBURST - 1 + \\ &+ N_{RD} \cdot (tBURST + 2). \end{aligned} \quad (4)$$

where $\%$ represents the module operator. The following Lemmas 2, 4, 5 formally prove the bounds on $L^{PRE}(N_{PRE})$, $L^{ACT}(N_{ACT})$, $L^{RD}(N_{RD})$. Note that intuitively, the two clock cycles of delay added to most timing parameters in Equations 3, 4 represent the effect of bus conflicts; Lemma 3 is used to bound the effect of command bus conflicts for ACT and RD commands.

Lemma 2: The maximum delay caused by N_{PRE} interfering PRE commands on the PRE under analysis is upper bounded by Equation 2.

Proof: Since the request under analysis does not share its bank with any other request in the queue, each interfering PRE command can only cause interference for one clock cycle on the command bus. However, we also need to add command bus contention from later-scheduled ACT and RD commands.

Since $tRRD \geq 4$ and $tBURST = 4$, at most one ACT command and one RD command can be issued every 4 cycles; the remaining 2 cycles must thus be available for

PRE commands. Now assume that the PRE under analysis is ready at time 0. In the worst case there cannot be an ACT or RD command scheduled at time 0; otherwise, such command would be overlapped with the PRE under analysis, thus reducing the overall delay. In summary, an interfering PRE command must be scheduled at time 0, and then we can have 2 cycles of command bus contention from ACT, RD commands for every 2 issued PRE commands; therefore, the bound in Equation 2 holds. ■

Lemma 3: The maximum command bus contention caused by PRE and RD command on an ACT command that interferes with the request under analysis is 2 cycles. Similarly, the maximum command bus contention caused by PRE and ACT command on a RD command that interferes with the request under analysis is 2 cycles.

Proof: Any two RD or two ACT commands must be separated by at least 4 cycles, thus command bus contention caused by RD or ACT commands is limited to one cycle.

Since we are interested in command bus contention on ACT or RD commands that interfere with the request under analysis, it follows that the request under analysis must have already issued its PRE command. Since the request under analysis has the largest timestamp, all other banks in use must have been precharged at least once; hence, the time at which any further PRE command can be issued must be dependent on the issue time of the previous ACT ($tRAS$ constraint) or RD command ($tRTP$ command) on the same bank. Again due to the separation of ACT and RD commands, no two PRE commands can be issued back-to-back. Hence, command bus contention caused by PRE commands is also limited to one cycle, completing the proof. ■

Lemma 4: The maximum delay caused by N_{ACT} interfering ACT commands on the ACT under analysis is upper bounded by Equation 3.

Proof: Assume that the ACT under analysis is ready at time 0. In the worst case, either the $tFAW$ or $tRRD$ constraint could be activated by an ACT issued at time -1; this ACT does not need to be included in the set of N_{ACT} interfering commands, since it has already been issued before the ACT under analysis is ready. If the $tRRD$ constraint is triggered, this causes a delay of $tRRD - 1$ on the first ACT in the interfering set. Since 4 ACTs must be issued to trigger $tFAW$ and each ACT can be issued $tRRD$ after the previous one (i.e., a total of $3{\cdot}tRRD$ cycles between the first and fourth ACT), if the $tFAW$ constraint is triggered it causes an initial delay of $tFAW - 3 \cdot tRRD - 1$. Since $tFAW \geq 4 \cdot tRRD$, the latter case maximizes the delay.

According to Lemma 3, each ACT command can suffer command bus contention for at most two cycles. Since the continuous sequence of interfering ACTs must respect the $tRRD$ constraint, the interfering delay after the initial $tFAW - 3 \cdot tRRD$ cycles must thus be upper bounded by $N_{ACT} \cdot (tRRD + 2)$. Since furthermore the ACT commands must also satisfy the $tFAW$ constraint, the delay is also bounded by $\lfloor N_{ACT}/4 \rfloor \cdot (tFAW + 2) + (N_{ACT}\%4) \cdot (tRRD + 2)$. Taking the maximum of the bounds results in Equation 3. ■

Lemma 5: The maximum delay caused by N_{RD} interfering RD commands on the RD under analysis is upper bounded by Equation 4.

Proof: Assume that the RD under analysis is ready at time 0. Similarly to the proof of Lemma 4, in the worst case a RD command not included in the set of N_{RD} interfering commands could be issued at time -1, resulting in an initial delay of $tBURST - 1$ cycles. Due to Lemma 3 and the constraint on $tBURST$, each interfering RD command can cause a further delay of $tBURST + 2$, yielding Equation 4. ■

Finally, the following theorem derives the bound on $L(N_{rq})$ by maximizing Equation 1 based on the derived bounds.

Theorem 6: The maximum delay $L(N_{rq})$ suffered by the request under analysis due to N_{rq} other read requests is upper bounded by:

$$\begin{aligned} L(N_{rq}) &= tFAW + tBURST - 3 \cdot tRRD - 2 + \\ &+ \max\big(N_{rq} \cdot tMAX, \lfloor N_{rq}/4 \rfloor \cdot (tFAW + 2) + \\ &+ (N_{rq}\%4) \cdot tMAX\big), \end{aligned} \tag{5}$$

where $tMAX = \max(tRRD, tBURST) + 2$.

Proof: The theorem follows immediately by evaluating Equation 1 according to Equations 2, 3, 4. Note that since $tRRD, tBURST > 2$ for all DDR devices, the values of L^{ACT} and L^{RD} increase more with each additional request compared to L^{PRE}; hence, $L(N_{rq})$ is maximal for $N_{PRE} = 0$, resulting in $L^{PRE}(0) = 0$. Then, in Equation 5 the constant term $tFAW + tBURST - 3 \cdot tRRD - 2$ is obtained by summing the two constant delay terms in Equations 3, 4, while the $\max$ term is obtained by maximizing the sum of Equations 3, 4 with $N_{ACT} + N_{RD} = N_{rq}$. ■

B. Write Batch Delay Analysis

We next consider the delay caused by write handling, according to the watermark policy described in Section IV, to the delay of the read request under analysis. We first compute the maximum number N_B of write batches that can delay the execution of the batch of read requests considered in Section IV-A. Then, we determine the maximum delay L^W caused by a single write batch. The overall delay suffered by the request under analysis can then be computed as $RD = L(N_{rq}) + N_B \cdot L^W$.

Theorem 7: The maximum number of write batches that can delay the execution of the read batch for the request under analysis is:

$$N_B = 1 + \left\lceil \frac{N_{rq}}{N_{wd}} \right\rceil, \tag{6}$$

and each batch delays the request under analysis for at most N_{wd} writes.

Proof: First note that while the request under analysis is in the read buffer, the controller must use the high watermark by definition. Furthermore, as noted earlier in this section, a write request (write-back from cache) can only happen as a result of a read request (fetch in cache), which has been generated by the same core.

In the worst case, the write buffer can be full due to prior requests when the request under analysis arrives. Since we assume $Q_{write} - W_{high} < N_{wd}$, this forces the controller to execute a single write batch; at most N_{wd} writes can be serviced after the request under analysis arrives before

the number of queued writes becomes lower than the high watermark and the batch ends.

Now note that as each of the N_{rq} prior read request is completed, the corresponding interfering core can add a new read request to the read queue. These later read requests have higher timestamp than the read under analysis, and thus cannot delay it as discussed in Section IV-A; however, each such read request can also generate a write request, for a total of N_{rq} later write requests that can be enqueued before the request under analysis is completed. Since a write batch is triggered as soon as the high watermark is reached, in the worst case later write requests can trigger $\lceil N_{rq}/N_{wd} \rceil$ additional batches, each comprising N_{wd} writes. Considering the batches generated by prior and later writes yields Equation 6. ■

Theorem 8: A write batch of length N_{wd} delays the read under analysis by at most:

$$L^W = (N_{wd} + 1) \cdot tRC. \tag{7}$$

Proof: In the worst case, all queued write requests might belong to the same core and target different rows in the same bank. This forces a delay tRC between the ACT commands of successive writes in the bundle, which is the longest possible timing constraint. Similarly, an additional tRC delay might be required between the last read issued before the write batch and the first write in the batch, and between the last write in the batch and the first read issued after the batch ends. This leads to a total delay of $(N_{wd}+1) \cdot tRC$, concluding the proof. ■

Note that the delay computed in Equation 7 is extremely pessimistic, since it assumes a pathological case where all queued writes have been produced by the same core. In practice, if the write buffer size is large enough, at any point in time the buffer is likely to contain requests by different cores. Since the write queue also employs FR-FCFS, such requests would then overlap over pipeline stages, as shown for reads in Section IV-A. This would result in a bound:

$$L^W = 2 \cdot tRC + 2 + L(N_{wd} - 1), \tag{8}$$

where the term $2 \cdot tRC$ accounts for the delay between the last read/first write and last write/first read, $L(N_{wd} - 1)$ is the pipelined delay suffered by the last write and computed based on Equation 5, and finally the term 2 accounts for the bus conflicts suffered by the ACT command of the first write in the batch, as detailed in Lemma 3 [6]. While there is no formal guarantee that Equation 8 upper bounds the delay caused by every write batch, we nevertheless use the bounds of both Equation 7 and Equation 8 in our evaluation since the employed full-system simulator cannot produce a pathological case similar to the one in the proof of Theorem 8.

C. Request and Job Driven Analysis

Sections IV-A and IV-B compute the worst-case delay RD suffered by a read request of task τ_i based on read and write requests of other interfering cores, respectively. If τ_i produces H_i^R read requests, the worst-case delay suffered by the task can then simply be obtained as $H_i^R \cdot RD$. Note that this *request driven* analysis makes no assumption on the behavior of the interfering cores. However, if number of requests produced by interfering cores is known, then a *job driven* analysis can result in lower delay bounds. This is especially true in the case of write-induced delay, since most programs generate a significantly lower amount of write requests compared to read requests, while the worst case per-request pattern used in Theorem 7 requires a higher number of interfering write requests compared to the N_{rq} interfering read requests.

Therefore, assume that the maximum number of read requests A^R and write requests A^W generated by interfering cores during the execution of τ_i is known ([13] discusses how to derive it based on the tasks' schedule), as well as the number H_i^W of write requests generated by τ_i itself. We can then bound the delay induced by write and read batches on the H_i^R read requests of a job of τ_i as follows.

Write batches: note that at most $A^W + H_i^W$ write requests can be inserted in the write queue after the job of τ_i begins. Following the same reasoning as in the proof of Theorem 7, the maximum number of write batches that can interfere with reads of τ_i is thus $N_B^{tot} = 1 + \lceil (A^W + H_i^W)/N_{wd} \rceil$.

Read batches: since task τ_i generates H_i^R reads, we need to consider the delay for H_i^R read batches. Let x_k to denote the number of interfering prior read requests that compose the $k-th$ batch for τ_i; the read batch delay is then $\sum_{k=1}^{H_i^R} L(x_k)$. In the worst case, there can be N_{rq} queued reads before a job of τ_i starts; hence, it must hold $\sum_{k=1}^{H_i^R} x_k = N_{rq} + A^R$.

The total job driven delay JD for a job of τ_i can then be obtained by summing the delay induced by read and write batches. To simplify notation, define $\bar{A}^R = N_{rq} + A^R$; maximizing the read batch delay expression over the values of x_k yields a total delay:

$$\begin{aligned} JD &= H_i^R \cdot (tFAW + tBURST - 3 \cdot tRRD - 2) + \\ &+ \max\left(\bar{A}^R \cdot tMAX, \lfloor \bar{A}^R/4 \rfloor \cdot (tFAW + 2) + \right. \\ &+ \left. (\bar{A}^R \% 4) \cdot tMAX\right) + N_B^{tot} \cdot L^W. \end{aligned} \tag{9}$$

V. Evaluation Results

In this section, we first present details of our simulation setup. We then present our evaluation results obtained using a set of synthetic and SPEC2006 benchmarks.

A. Simulation Setup

We use Gem5 full-system simulator [4] with a realistic memory controller model [10] that closely captures important timing and structural characteristics of COTS DRAM controllers. On the Gem5 simulator, we model a quad core ARM (out-of-order) system, largely based on the ARM Cortex A15 processor [2]. Both L1 and L2 caches are non-blocking: The core-private L1 cache can generate up to 6 outstanding cache-line fill requests and 6 corresponding write-back requests to the shared L2 cache; while the L2 cache in a real Cortex-A15 supports up to 11 outstanding read requests [2], our simulated system supports up to 24 outstanding read requests (4 cores · 6 requests) to minimize additional interference in the MSHRs [34] as our focus is in the DRAM-level interference. The simulator setup is shown in Table II.

[6]Note that we do not need to add command bus contention for either the last write request in the batch or the first read request after the batch, since bus contention is already accounted for in the delay suffered by those requests.

TABLE II: Baseline processor and DRAM system configuration

Core	Quad-core, ARMv7, out-of-order, 4GHz frequency
L1-I&D caches	private 32 K-byte, 2-way set-assoc., hit latency: 1ns/2ns(I/D); MSHRs: 2/6(I/D)
L2 cache	shared 1MByte, 16-way set assoc., 12ns hit latency, 24 MSHRs
DRAM controller	64-entry read buffer, 64-entry write buffer, 85%/50% high/low watermark, 18 minimum writes-per-switch, addr. mapping: RoRaBaChCo, open-page policy
DRAM chip	LPDDR2 @ 533Mhz (1 rank and 8banks)

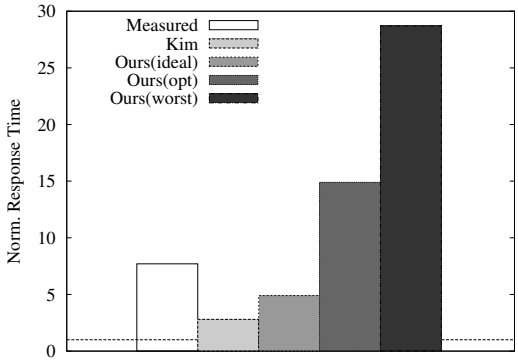


Fig. 6: Measured and analytic worst-case response times of a *Latency* benchmark with three memory-intensive co-runners

On the simulator we run a full Linux 3.14 kernel which is patched to use the PALLOC [35] memory allocator. We use PALLOC to partition DRAM banks and the shared L2 cache. For the purpose of our evaluation, we assign one private DRAM bank and 1/4 (256KiB) private L2 cache partition to each core. Therefore, there are neither cache space evictions nor DRAM bank conflicts caused by memory accesses from contending cores.

B. Results with Synthetic Benchmarks

We now investigate the validity of our analysis compared to experimental results obtained using a set of carefully engineered synthetic benchmarks.

In this experiment, our goal is to simulate and measure the worst-case memory interferences on a system in which DRAM banks and the LLC are partitioned. We use *Latency* benchmark [37] as the task under analysis. The benchmark is a pointer-chasing application over a randomly shuffled linked-list. Due to data dependency, it can only generate one outstanding memory request at a time. Furthermore, because the size of linked list is two times bigger than the size of the LLC, each memory access is likely to result in a cache miss, hence generating a DRAM request. As a result, its execution time highly depends on DRAM performance and any delay in its memory access will directly contribute to its execution time increase.

We first run the Latency benchmark alone on Core0 to collect its solo execution time and the number of LLC misses. We then co-schedule three memory intensive tasks on the other cores (Core1-3) to generate high memory traffic and measure the response time increase of the Latency benchmark. Note that the number of LLC misses of Latency does not change between solo and co-scheduled execution as the LLC space is partitioned. Furthermore, because each core also has a dedicated DRAM bank, the number of DRAM row hits and misses also would not change. Therefore, any response time increase mainly comes from contention in the DRAM controller and its shared command and data bus which we modeled in Section IV. For co-scheduled memory intensive tasks, we use the Bandwidth benchmark [37], which *writes* a big array continuously. Because the benchmark does not have data dependencies in accessing memory, modern out-of-order cores can generate as many outstanding requests as possible. Furthermore, because a write miss in the Bandwidth benchmark generates two requests, a cache-line refill (read) and a write-back (write), the read and write buffers in the DRAM controller will quickly be occupied, which in turn will delay the requests of the task under analysis.

Figure 6 shows both measured and analytically calculated response times of the Latency benchmark (normalized to its solo execution time). In the figure, *Kim* denotes the analysis in [13] while *Ours* denotes our analysis in Section IV. We present three variations of our analysis: *Ours(ideal)* represents our read delay analysis Equation 5 without considering write draining. It is equivalent as assuming that the size of the write-queue of the DRAM controller is infinite and therefore write draining never delays pending reads; *Ours(opt)* and *Ours(worst)* represent two versions of our analysis that consider write-draining as described in Section IV-B. They differ in that Ours(opt) uses Equation 8 to calculate a single batch of write draining delay while Ours(worst) uses Equation 7 for the same. From the figure, we make several important observations. First, as expected, Kim significantly under-estimates the actual measured delay—by 63%. This is because the analysis models only three competing memory requests while in this experiment there can be up to 36 competing memory requests (6 reads and writes from each competing core). Second, Ours(ideal) also under-estimate the delay because it does not take the write-draining into account. On the other hand, Ours(opt) more closely matches with the measured result. This is because, in this experiment, the write requests are processed in parallel as assumed in Equation 8. Lastly, Ours(worst) produces a safe but highly pessimistic bound as it considers the pathological case, which is difficult to produce in real experiments.

C. Results with SPEC2006 Benchmarks

In this subsection, we present results with SPEC2006 benchmarks. The basic experiment setup is the same as the previous subsection—i.e., one task under analysis and three co-scheduled Bandwidth benchmark instances—except that we now use SPEC2006 benchmarks as the tasks under analysis instead of the Latency benchmark.

Figure 7 shows the results. Overall, the results show a similar trend as observed in Section V-B. Compared to measured response times, Kim under-estimates the response times of all

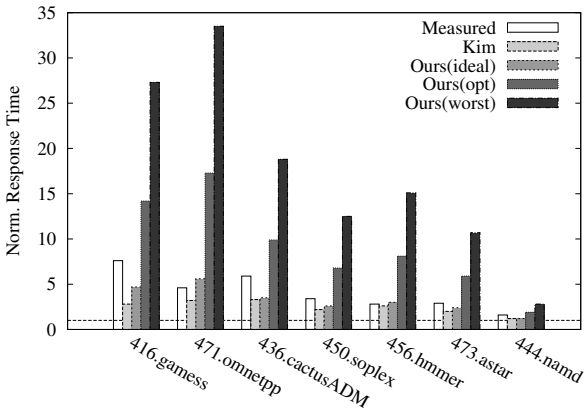


Fig. 7: Measured and analytic worst-case response times of SPEC2006 with three memory-intensive co-runners

tested benchmarks as it only consider thee concurrent requests. Among our analysis, Ours(ideal) also under-estimate all but two benchmarks (471.omnetpp and 456.hmmer) as it does not consider write-draining. Ours(opt), on the other hand, is close to the measured values and no tested benchmark is under-estimated. However, the degree of pessimism in the analysis varies considerably among the benchmarks depending on their memory access characteristics. Lastly, Ours(worst) produces highly pessimistic results, especially for memory intensive benchmarks such as 471.omnetpp. Again, this is because the analysis considers a highly pathological situation which is difficult to generate in real experiments. Still, if the benchmark under analysis is critical and not memory intensive, which are often the case in many time-critical control applications, such a bound can be useful. For example, Ours(worst) produces a reasonable delay bound for 444.namd because the number of read requests to be analyzed is relatively small.

VI. Related Work

As memory performance is becoming increasingly important in modern multicore systems, there has been great interest in the real-time research community to minimize and analyze memory related interference delay for designing more predictable real-time systems.

Initially, many researchers model the cost to access the main memory as a constant and view the main memory as a single resource shared by the cores [36], [23], [33], [28]. However, modern DRAM systems are composed of many sophisticated components and the memory access cost is far from being a constant as it varies significant depending on the states of the variety of components comprising the memory system.

Many researchers turn to hardware approaches and develop specially designed real-time DRAM controllers that are highly predictable and provide certain performance guarantees. The works in [24], [7], [32], [14] implement hardware based private banking schemes which eliminate interferences caused by sharing the banks. They differ in that close page policy is used in [24], [7] while open page policy is used in [32], [14]. The works in [22], [1], [9] utilize interleaved bank mapping, which effectively transforms multiple memory banks as a single unit of access to simplify resource management. They, however, use different arbitration policies such as TDM [9], round-robin [22], and CCSP [1]. The work in [15] proposes a DRAM command scheduling method to efficiently support variable transaction sizes while providing WCET guarantees. While these proposals are valuable, especially for hard real-time systems, they generally do not offer high average performance. For example, none of the real-time DRAM controllers implement read prioritization and write buffering—a common performance optimization technique to hide write processing delay in the critical path, which is modeled in our work. Also, the real-time DRAM controllers are not available in COTS systems.

To improve performance isolation in COTS systems, several recent papers proposed software-based bank partitioning techniques [35], [18], [29]. They exploit the virtual memory of modern operating systems to allocate memory on specific DRAM banks without requiring any other special hardware support. Similar techniques have long been applied in partitioning shared caches [16], [17], [38], [31], [19]. These resource partitioning techniques eliminate space contention of the partitioned resources, hence improve performance isolation. However, as shown in [35], [13], modern COTS systems have many other still shared components that affect memory performance. A recent attempt to analyze these effects [13], which is reviewed in Section III, considers many aspects of COTS DRAM controllers such as request re-ordering, but it does not consider read prioritization and assumes one outstanding memory request per core can be requested to the DRAM controller. In contrast, we model a modern COTS DRAM controller that handles multiple outstanding memory requests from each core and out-of-order memory request processing (i.e., prioritizing reads over writes). We believe our system model and the analysis capture commonly found architectural features in modern COTS systems, hence better applicable for analyzing memory interference on COTS multicore systems.

VII. Conclusion

We have presented a new parallelism-aware worst-case memory interference delay analysis for COTS multicore systems. We model a COTS DRAM controller that has a separate read and a write request buffer, which buffer multiple outstanding memory requests from the LLC and processes them in out-of-order: it prioritizes reads over writes and row-hits over misses. These are common characteristics of COTS memory systems but known to be difficult to analyze for worst-case performance.

In this work, we have shown that memory interference delays in such a complex COTS system can be analytically bounded, with the help of appropriate software-based resource partitioning mechanisms [18], [29], [35]. Our analysis provides an analytic upper bound on the worst-case memory interference delay for each read memory request of the task under analysis and the derived bound does not require any assumption on the interfering tasks' memory access patterns. We also have shown that if the number of read and write requests generated by each core is known, then the analytical bounds can be improved. We have validated our analysis

on the Gem5 full-system simulator using both synthetic and SPEC2006 benchmarks.

Compared to previous COTS focused effort [13], which significantly under-estimates the delays (by up to 63%), we claim our system model and the derived analysis bound are more closely matched with real COTS systems. Our evaluation results, however, also clearly have shown the inherent weaknesses of COTS architecture when it comes to worst-case performance; in particular, while architectural optimizations such as write buffering and batch processing have beneficial effects on average-case performance, they nevertheless induce pathological arrival patterns that result in highly pessimistic delay bounds. As future work, we will examine low-cost architectural supports for COTS systems that can provide better isolation and reduce pessimism in the analysis.

ACKNOWLEDGEMENTS

This research is supported in part by NSF CNS 1302563.

REFERENCES

[1] B. Akesson, K. Goossens, and M. Ringhofer. Predator: a predictable SDRAM memory controller. In *Hardware/software codesign and system synthesis (CODES+ISSS)*. ACM, 2007.

[2] ARM. *ARM Cortex-A15 MPCore Processor Technical Reference Manual (Revision: r4p0)*, 2013.

[3] B. Bhat and F. Mueller. Making dram refresh predictable. *Real-Time Systems*, 47(5):430–453, 2011.

[4] N. Binkert, B. Beckmann, G. Black, S. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[5] N. Chatterjee, N. Muralimanohar, R. Balasubramonian, A. Davis, and N. Jouppi. Staged reads: Mitigating the impact of DRAM writes on DRAM reads. In *High Performance Computer Architecture (HPCA)*, 2012.

[6] X. Ding, K. Wang, and X. Zhang. SRM-buffer: an OS buffer management technique to prevent last level cache from thrashing in multicores. In *European Conf. on Computer Systems (EuroSys)*. ACM, 2011.

[7] L. Ecco, S. Tobuschat, S. Saidi, and R. Ernst. A mixed critical memory controller using bank privatization and fixed priority scheduling. In *Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2014.

[8] Freescale. *e500mc Core Reference Manual (Rev.3)*, 2013.

[9] S. Goossens, B. Akesson, and K. Goossens. Conservative open-page policy for mixed tim-criticality memory controllers. In *Design, Automation and Test in Europe (DATE)*, 2013.

[10] A. Hansson, N. Agarwal, A. Kolli, T. Wenisch, and A. Udipi. Simulating DRAM controllers for future system architecture exploration. 2014.

[11] P. Jayachandran and T. Abdelzaher. Delay composition in preemptive and non-preemptive real-time pipelines. *Real-Time Systems*, 2008.

[12] JEDEC. DDR3 SDRAM Standard JESD79-3F, 2012.

[13] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. R. Rajkumar. Bounding Memory Interference Delay in COTS-based Multi-Core Systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.

[14] Y. Krishnapillai, Z. P. Wu, and R. Pellizzoni. A Rank-Switching, Open-Row DRAM Controller for Time-Predictable Systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 27–38. IEEE, 2014.

[15] Y. Li, B. Akesson, and K. Goossens. Dynamic command scheduling for real-time memory controllers. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 3–14. IEEE, 2014.

[16] J. Liedtke, H. Haertig, and M. Hohmuth. OS-Controlled cache predictability for real-time systems. In *Real-Time Technology and Applications Symposium (RTAS)*. IEEE, 1997.

[17] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *High Performance Computer Architecture (HPCA)*. IEEE, 2008.

[18] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu. A software memory partition approach for eliminating bank-level interference in multicore systems. In *Parallel Architecture and Compilation Techniques (PACT)*. ACM, 2012.

[19] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-Time Cache Management Framework for Multi-core Architectures. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2013.

[20] Micron Technology, Inc. *2Gb DDR3 SDRAM: MT41J256M8, Rev. Q 04/13 EN*, 2006.

[21] C. Natarajan, B. Christenson, and F. Briggs. A study of performance impact of memory controller features in multi-processor server environment. In *Proceedings of the 3rd workshop on Memory performance issues*, pages 80–87. ACM, 2004.

[22] M. Paolieri, E. Quiñones, and F. J. Cazorla. Timing effects of DDR memory systems in hard real-time multicore architectures: Issues and solutions. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(1s):64, 2013.

[23] R. Pellizzoni, A. Schranzhofery, J. Cheny, M. Caccamo, and L. Thiele. Worst case delay analysis for memory interference in multicore systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2010.

[24] J. Reineke, I. Liu, H. Patel, S. Kim, and E. Lee. PRET DRAM controller: Bank privatization for predictability and temporal isolation. In *Hardware/software codesign and system synthesis (CODES+ISSS)*. ACM, 2011.

[25] J. Reineke and R. Sen. Sound and efficient wcet analysis in the presence of timing anomalies. In *OASIcs-OpenAccess Series in Informatics*, volume 10. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2009.

[26] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. Owens. Memory access scheduling. In *ACM SIGARCH Computer Architecture News*. ACM, 2000.

[27] S. Schliecker and R. Ernst. Real-time Performance Analysis of Multiprocessor Systems with Shared Memory. *ACM Trans. Embed. Comput. Syst.*, 10(2):22:1–22:27, Jan. 2011.

[28] A. Schranzhofer, J. Chen, and L. Thiele. Timing analysis for TDMA arbitration in resource sharing systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2010.

[29] N. Suzuki, H. Kim, D. d. Niz, B. Andersson, L. Wrage, M. Klein, and R. Rajkumar. Coordinated bank and cache coloring for temporal protection of memory accesses. In *Computational Science and Engineering (CSE)*, pages 685–692. IEEE, 2013.

[30] D. Wang. *Modern DRAM Memory systems: Performance Analysis and Scheduling Algorithm*. PhD thesis, University of Maryland at College Park, 2005.

[31] B. Ward, J. Herman, C. Kenna, and J. Anderson. Making Shared Caches More Predictable on Multicore Platforms. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2013.

[32] Z. Wu, Y. Krish, and R. Pellizzoni. Worst Case Analysis of DRAM Latency in Multi-Requestor Systems. In *Real-Time Systems Symposium (RTSS)*, 2013.

[33] G. Yao, R. Pellizzoni, S. Bak, E. Betti, and M. Caccamo. Memory-centric scheduling for multicore hard real-time systems. *Real-Time Systems*, 2012.

[34] H. Yun. Parallelism-aware memory interference delay analysis for COTS multicore systems. *CoRR*, abs/1407.7448, 2014.

[35] H. Yun, R. Mancuso, Z. Wu, and R. Pellizzoni. PALLOC: DRAM Bank-Aware Memory Allocator for Performance Isolation on Multicore Platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.

[36] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory Access Control in Multiprocessor for Real-time Systems with Mixed Criticality. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2012.

[37] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. MemGuard: Memory Bandwidth Reservation System for Efficient Performance Isolation in Multi-core Platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013.

[38] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multicore cache management. In *European Conf. on Computer Systems (EuroSys)*, 2009.

APPENDIX

Note that throughout the paper, we assume *delay additivity*: the memory interference delay is additive to the execution time of the task under analysis. Such assumption is similarly used by related work on main memory delay analysis [13], [32]. However, one might ask whether the underlying processor hardware truly satisfies the assumption. The following two definitions, which use a similar formalism as in [25], formally capture this concept[7].

Definition 9 (Instructions Semantics): A program fragment is a sequence of n (possibly one) instructions $\iota_0 \dots \iota_n$. We write $s \xrightarrow[\iota_0 \dots \iota_n]{t} s'$ to mean that executing instructions $\iota_0 \dots \iota_n$ takes t time and causes the processor to transition from state s to s'. In general, a set of instructions could cause the processor to transition from s to one of multiple states s' (non-determinism). In this case, we define the maximum time to execute a set of instruction starting from state s as: $\max(s, \iota_0 \dots \iota_n) = \max\{t | s \xrightarrow[\iota_0 \dots \iota_n]{t} s'\}$.

Definition 10 (Delay Additivity): An architecture is not delay additive if there exists a program fragment and a processor state s such that: there are states s_1, s_2, with $s \xrightarrow[\iota_0]{t_1} s_1, s \xrightarrow[\iota_0]{t_2} s_2$, $t_1 > t_2$, and $\max(s_1, \iota_1 \dots \iota_n) > \max(s_2, \iota_0 \dots \iota_n)$.
Essentially, an architecture is not delay additive if suffering some delay while executing an instruction causes additional delay on the rest of the program; in this case, computing the modified execution time of the task as the execution time when running in isolation plus the delay factor $H_i^R \cdot RD$ according to request driven analysis would not be safe, since the task could suffer an increase in execution time larger than $H_i^R \cdot RD$.

We argue that determining whether a given architecture is delay additive is outside the scope of this paper. In general, one could conceive of instruction scheduling policies that are not additive; in particular, any policy that behaves similarly to a bin-packing heuristic (where items are instructions and bins are execution units) would likely violate additivity. This said, we make the following two key observations: (1) in practice, when instruction latency is large as is the case for main memory accesses, modern out-of-order architectures are likely to behave in a delay additive manner. If the delay suffered by a memory instruction increases between solo and co-scheduled execution, then typically either the same (if the pipeline stalled) or more instructions can be executed out-of-order in parallel with the memory instruction. Thus, the increase in execution time can either be equal or less than the added instruction delay. (2) The goal of the paper is to derive a bound that can be shown to be safe through measurement. In general, software systems running on COTS hardware can only be certified through measurement; formally proving that an analysis bound is correct would require complete knowledge of the underlying hardware, which is typically not available for COTS systems. Still, being able to derive safe bounds is invaluable at design time, since applications are typically first developed in isolation and then integrated together.

[7]Delay additivity is strictly related, but not equivalent, to the concept of timing anomalies [25]. Intuitively, an architecture exhibits a timing anomaly if taking *less* time to execute an instruction leads to a larger overall execution time; while an architecture if not delay additive is taking *more* time to execute an instruction leads to an even larger execution time.