Bounding Memory Interference Delay in COTS-based Multi-Core Systems

Hyoseung Kim*, Dionisio de Niz†, Björn Andersson†, Mark Klein†, Onur Mutlu*, Ragunathan (Raj) Rajkumar*
*Electrical and Computer Engineering, Carnegie Mellon University
†Software Engineering Institute, Carnegie Mellon University
hyoseung@cmu.edu, {dionisio, baandersson, mk}@sei.cmu.edu, onur@cmu.edu, raj@ece.cmu.edu

Abstract—**In commercial-off-the-shelf (COTS) multi-core systems, a task running on one core can be delayed by other tasks running simultaneously on other cores due to interference in the shared DRAM main memory. Such memory interference delay can be large and highly variable, thereby posing a significant challenge for the design of predictable real-time systems. In this paper, we present techniques to provide a tight upper bound on the worst-case memory interference in a COTS-based multi-core system. We explicitly model the major resources in the DRAM system, including banks, buses and the memory controller. By considering their timing characteristics, we analyze the worst-case memory interference delay imposed on a task by other tasks running in parallel. To the best of our knowledge, this is the first work bounding the request re-ordering effect of COTS memory controllers. Our work also enables the quantification of the extent by which memory interference can be reduced by partitioning DRAM banks. We evaluate our approach on a commodity multi-core platform running Linux/RK. Experimental results show that our approach provides an upper bound very close to our measured worst-case interference.**

I. Introduction

In multi-core systems, main memory is a major shared resource among processor cores. Tasks running concurrently on different cores contend with each other to access main memory, thereby increasing their execution times. As memory-intensive applications are becoming more prevalent in real-time embedded systems, an upper bound on the memory interference delay is needed to evaluate their schedulability. Moreover, the reduction of this interference is critical to make effective use of multicore platforms.

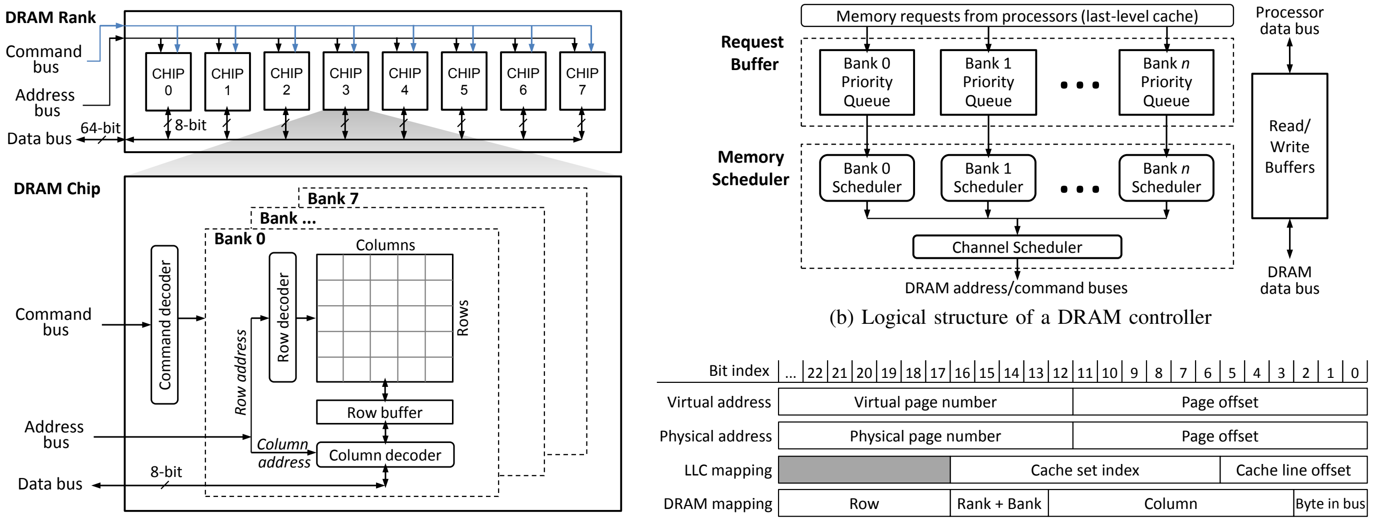
Previous studies on bounding memory interference delay [9, 43, 32, 37, 5] model main memory as a *black-box* system, where each memory request takes a constant service time and memory requests from different cores are serviced in either Round-Robin (RR) or First-Come First-Serve (FCFS) order. This memory model, however, is not safe for commercial-off-the-shelf (COTS) multi-core systems because it hides critical details necessary to place an upper bound on its timing. Specifically, in modern COTS-based systems, the main memory typically consists of DRAM to cope with high performance and capacity demands. The DRAM system contains multiple resources such as ranks, banks and buses, and the access time varies considerably depending on the requested address and the rank/bank states. In addition, memory requests are scheduled by an on-chip, out-of-order memory controller based on the First-Ready First-Come First-Serve (FR-FCFS) policy [35, 29, 25, 45], where memory requests arriving early may be serviced later than ones arriving later if the memory system is not ready to service the former. Therefore, the over-simplified memory model used by previous studies may produce pessimistic or optimistic estimates on the memory interference delay in a COTS multicore system.

In this paper, we propose a *white-box* approach for bounding memory interference. By explicitly considering the timing characteristics of major resources in the DRAM system, including the re-ordering effect of FR-FCFS and the rank/bank/bus timing constraints, we obtain a tight upper bound on the worst-case memory interference delay for a task when it executes in parallel with other tasks. Our technique combines two approaches: a request-driven and a job-driven approach. The *request-driven* approach focuses on the task's own memory requests, and the *job-driven* approach focuses on interfering memory requests during the task's execution. Combining them, our analysis yields a tight upper bound on the worst-case response time of a task in the presence of memory interference. To reduce the negative impact of memory interference, we propose to use software DRAM bank partitioning [22, 39]. We consider both dedicated and shared bank partitions due to the limited availability of DRAM banks, and our analysis results in an upper bound on the interference delay in both cases.

Our approach does *not* require any modifications to hardware components or application software. Therefore, it is readily applicable to COTS-based multicore real-time systems. In the evaluation section, we show the effect of our approach on a well-known COTS multicore platform.

The rest of this paper is organized as follows. Section II explains how modern DRAM systems work. Section III describes the system and task model used in this paper. Section IV presents how we bound memory interference. A detailed evaluation is provided in Section V. Section VI reviews related work, and Section VII concludes the paper.

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute. This material has been approved for public release and unlimited distribution. Carnegie Mellon is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University. DM-0000668



(a) DRAM device organization

(b) Logical structure of a DRAM controller

(c) Task address to cache and DRAM mapping (Intel i7-2600)

Fig. 1: Modern DDR SDRAM systems

II. Background on DRAM Systems

The memory interference delay in a DRAM system is largely affected by two major components: (i) the DRAM chips where the actual data are stored, and (ii) the memory controller that schedules memory read/write requests to the DRAM chips. In this section, we provide a brief description of these two components. Our description is based on DDR3 SDRAM systems, but it generally applies to other types of COTS DRAM systems. For more information, interested readers may refer to [35, 29, 25, 27].

A. DRAM Organization

A DRAM system as shown in Figure 1(a) is organized as a set of *ranks*, each of which consists of multiple DRAM chips. Each DRAM chip has a narrow data interface (e.g. 8 bits), so the DRAM chips in the same rank are combined to widen the width of the data interface (e.g. 8 bits x 8 chips = 64 bits data bus). A DRAM chip consists of multiple DRAM *banks* and memory requests to different banks can be serviced in parallel. Each DRAM bank has a two-dimensional array of rows and columns of memory locations. To access a column in the array, the entire row containing the column first needs to be transfered to a *row-buffer*. This action is known as *opening* a row. Each bank has one row-buffer that contains at most one row at a time. The size of the row-buffer is therefore equal to the size of one row, which is 1024 or 2048 columns in a DDR3 SDRAM chip [12].

The DRAM access latency varies depending on which row is currently stored in the row-buffer of a requested bank. If a memory request accesses a row already in the row-buffer, the request is directly serviced from the row-buffer, thereby resulting in a short latency. This case is called a *row hit*. If the request is to a row that is different from the one in the row-buffer, the currently open row should be closed by a *precharge* command and the requested row should be delivered to the row-buffer by an *activate* command. Then the request can be serviced from the row-buffer. This case is called a *row conflict* and results in a much longer latency. In both cases, transferring data through the data bus incurs additional latency. The data is transferred in a burst mode and a *burst length (BL)* determines the number of columns transferred per read/write access.

B. Memory Controller

Figure 1(b) shows the structure of a memory controller in a modern DRAM system. The memory controller is a mediator between the last-level cache of a processor and the DRAM chips. It translates read/write memory requests into corresponding DRAM commands and schedules the commands while satisfying the timing constraints of DRAM banks and buses. To do so, a memory controller consists of a request buffer, read/write buffers, and a memory scheduler. The request buffer holds the state information of each memory request, such as an address, a read/write type, a timestamp and its readiness status. The read/write buffers hold the data read from or to be written to the DRAM. The memory scheduler determines the service order of the pending memory requests.

The memory scheduler has a two-level hierarchical structure.[1] As shown in Figure 1(b), the first level consists of per-bank *priority queues* and *bank schedulers*. When a memory request is generated, the request is enqueued into the priority queue that corresponds to the request's bank index. The bank scheduler determines priorities of pending requests and generates a sequence of DRAM commands to service each request. The bank scheduler also tracks the state of the bank. If the highest-priority command does not violate any timing constraints of the bank, the command is said to be *ready* for the bank and is sent to the next level. The second level consists of

[1] The physical structure of priority queues, bank schedulers, and the channel scheduler depends on the implementation. They can be implemented as a single hardware structure [29].

a *channel scheduler*. It keeps track of DRAM commands from all bank schedulers, and monitors the timing constraints of ranks and address/command/data buses. Among the commands that are *ready* with respect to such channel timing constraints, the channel scheduler issues the highest-priority command. Once the command is issued, the channel scheduler signals ACK to the corresponding bank scheduler, and then the bank scheduler selects the next command to be sent.

Memory Scheduling Policy: Scheduling algorithms for COTS memory controllers have been developed to maximize the data throughput and minimize the average-case latency of DRAM systems. Specifically, modern memory controllers employ First-Ready First-Come First-Serve (FR-FCFS) [35, 29] as their base scheduling policy. FR-FCFS first prioritizes ready DRAM commands over others, just as the two-level scheduling structure does. At the bank scheduler level, FR-FCFS re-orders memory requests as follows:

1) Row-hit memory requests have higher priorities than row-conflict requests.
2) In case of a tie, older requests have higher priorities.

At the channel scheduler level, FR-FCFS issues DRAM commands in the order of their arrival time. Therefore, under FR-FCFS, the oldest row-hit request has the highest priority and the youngest row-miss request has the lowest priority.

C. Bank Address Mapping and Bank Partitioning

In modern DRAM systems, physical addresses are interleaved among multiple banks (and ranks) to exploit bank-level parallelism for average-case performance improvement. The granularity of address interleaving is typically equal to the size of one row, because mapping adjacent addresses to the same row may provide better row-buffer locality. This strategy is called a *row-interleaved* address mapping policy and it is widely used in many COTS systems. As an example, Figure 1(c) shows the address mapping of the system equipped with the Intel i7-2600 processor which follows the row-interleaved policy.[2] In this system, bits 13 to 16 of the physical address are used for the rank and bank indices.

The row-interleaved policy, however, can significantly increase the memory access latency in a multi-core system [26, 22, 13]. For instance, multiple tasks running simultaneously on different cores may be mapped to the same DRAM banks. This mapping can unexpectedly decrease the row-buffer hit ratio of each task and introduce re-ordering of the memory requests, causing significant delays in memory access.

Software bank partitioning [22, 39] is a technique used to avoid the delays due to shared banks. By dedicating a specific DRAM bank to each task (or a set of tasks running on the same core), bank partitioning can prevent the unexpected eviction of the currently open row and the negative effect of request re-ordering. The key to this technique is in the mapping between physical addresses and rank-bank indices. If a task is assigned only physical pages with a specific rank-bank index b, all the memory accesses of that task are performed on the rank-bank b. By controlling the physical page allocation in the OS, the physical memory space can be divided into *bank partitions* and a specific bank partition can be assigned to a task. However, since the number of DRAM banks available in a system is growing much slower than the number of processor cores, it may not be feasible to assign a dedicated DRAM bank to each core's taskset. In our work, we therefore consider not only dedicated DRAM banks to reduce memory interference delay but also shared banks to cope with their limited availability.

TABLE I: DRAM timing parameters [1]

Parameters	Symbols	DDR3-1333	Units
DRAM clock cycle time	t_{CK}	1.5	nsec
Precharge latency	t_{RP}	9	cycles
Activate latency	t_{RCD}	9	cycles
CAS read latency	*CL*	9	cycles
CAS write latency	*WL*	7	cycles
Burst Length	*BL*	8	columns
Write to read delay	t_{WTR}	5	cycles
Write recovery time	t_{WR}	10	cycles
Activate to activate delay	t_{RRD}	4	cycles
Four activate windows	t_{FAW}	20	cycles
Refresh to activate delay	t_{RFC}	160	nsec
Average refresh interval	t_{REFI}	7.8	μsec

III. System Model

Our system model assumes a multi-core system with the DDR SDRAM sub-system presented in Section II. Specifically, the memory controller uses the FR-FCFS policy, and the arrival times of memory requests are assumed to be recorded when they arrive at the memory controller. For simplicity, we assume that DRAM consists of a single rank, but systems with multiple ranks can also be analyzed by our proposed method. The memory controller uses an *open-row* policy which keeps the row-buffer open. We assume that the DRAM is not put into a low-power state at any time.

Four DRAM commands are considered in this work: precharge (PRE), activate (ACT), read (RD) and write (WR). Depending on the current state of the bank, the memory controller generates a sequence of DRAM commands for a single read/write memory request as follows:

- *Row-hit* request: RD/WR
- *Row-conflict* request: PRE, ACT and RD/WR

Note that the auto-precharge commands (RDAP/WRAP) are not generated under the open-row policy. We do not consider the refresh (REF) command because the effect of REF in memory interference delay is rather negligible compared to that of other commands.[3] The DRAM timing parameters used in this work are summarized in Table I and are taken from Micron's datasheet [1].

[2] The DRAM mapping of Figure 1(c) is for the single-channel configuration in this system. More details on this system will be given in Section V.

[3] The effect of REF (E_R) in memory interference delay can be roughly estimated as $E_R^{k+1} = \lceil \{(\text{total delay from analysis}) + E_R^k\}/t_{REFI} \rceil \cdot t_{RFC}$, where $E_R^0 = 0$. For the DDR3-1333 with 2Gb density below 85°C, t_{RFC}/t_{REFI} is 160ns/7.8μs $= 0.02$, so the effect of REF results in only about 2% increase in the total memory interference delay. A more detailed analysis on REF can be found in [7].

The system is equipped with a single-chip multi-core processor that has N_P identical cores running at a fixed clock speed. The processor has a last-level cache (LLC), and the LLC and the DRAM are connected by a single memory channel. We assume that all memory requests sent to the DRAM system are misses in the LLC, which is valid in cache-enabled systems. A missed cache-line can be fetched from the DRAM by a single memory request because of the burst-mode data transfer. Therefore, the number of memory requests is equal to the number of LLC misses, and the addresses of memory requests to each DRAM bank are aligned to the size of *BL* (burst length). In this paper, we assume that each core has a fully timing-compositional architecture as described in [40]. This means that each core is in-order with one outstanding cache miss and any delays from shared resources are additive to the task execution times.

We focus on *partitioned fixed-priority preemptive task scheduling* because it is widely used in many commercial real-time embedded OSes such as OSEK [2] and VxWorks [41]. For the task model, we assume sporadic tasks with constrained deadlines. We do not make any assumptions on the priority assignment schemes, so any fixed-priority assignment can be used, such as Rate Monotonic [21]. Tasks are ordered in decreasing order of priorities, i.e. $i < j$ implies that task τ_i has higher priority than task τ_j. Each task has a unique priority and n is the lowest priority. We assume that tasks fit in the memory capacity. It is also assumed that tasks do not suspend themselves during execution and do not share data.[4] Task τ_i is thus represented as follows:

$$\tau_i = (C_i, T_i, D_i, H_i)$$

- C_i: the worst-case execution time (WCET) of any job of task τ_i, when τ_i executes in isolation.
- T_i: the minimum inter-arrival time of each job of τ_i
- D_i: the relative deadline of each job of τ_i ($D_i \leq T_i$)
- H_i: the maximum number of DRAM requests generated by any job of τ_i

Note that no assumptions are made on the memory access pattern of a task (e.g. access rate). Parameters C_i and H_i can be obtained by either measurement-based or static-analysis tools. It is assumed that task preemption does not incur cache-related preemption delay (CRPD), so H_i does not change due to preemption. This assumption is easily satisfied in COTS systems by using cache coloring [16]. However, it is worth noting that our analysis can be easily combined with CRPD analyses such as in [4]. As we only use the number of memory accesses rather than access patterns, the memory interference of additional cache reloads due to preemption can be bounded by the maximum number of cache reloads that CRPD analyses provide.

Bank partitioning is considered to divide DRAM banks into N_{BP} partitions. Each bank partition is represented as a unique integer in the range from 1 to N_{BP}. It is assumed that the number of bank partitions assigned to a task does not affect the task's WCET. Bank partitions are assigned to cores and tasks running on the same core use the same set of bank partitions. Depending on assignment, bank partitions may be dedicated to a specific core or shared among multiple cores.

Lastly, each task is assumed to have sufficiently cache space of its own to store one row of each DRAM bank assigned to it.[5] This is a reasonable assumption in a modern multi-core system which typically has a large LLC. For instance, Figure 1(c) shows a physical address mapping to the LLC and the DRAM in the Intel Core-i7 system. For the LLC mapping, the last 6 bits of a physical address are used as a cache line offset, and the next 11 bits are used as a cache set index. For the DRAM mapping, the last 13 bits are used as a column index and the next 4 bits are used as a bank index. In order for a task to store one row in its cache, consecutive $2^{13-6} = 128$ cache sets need to be allocated to the task. If cache coloring is used, this is equal to 2 out of 32 cache partitions in the example system.

We use the following notation for convenience:

- $hp(\tau_i)$: the set of tasks with higher priorities than i
- $proc(\tau_i)$: the processor core index where τ_i is assigned
- $task(p)$: the set of tasks assigned to a processor core p
- $bank(p)$: the set of bank partitions assigned to a core p
- $shared(p, q)$: the intersection of $bank(p)$ and $bank(q)$

IV. Bounding Memory Interference Delay

The memory interference delay that a task can suffer from other tasks can be estimated by using either of two factors: (i) the number of memory requests generated by the task itself, and (ii) the number of interfering requests generated by other tasks that run in parallel. For instance, if a task τ_i does not generate any memory requests during its execution, this task will not suffer from any delays regardless of the number of interfering memory requests from other tasks. In this case, the use of factor (i) will give a tight estimate. Conversely, assume that other tasks simultaneously running on different cores do not generate any memory requests. Task τ_i will not experience any delays because there is no extra contention on the memory system from τ_i's perspective, so the use of factor (ii) will give a tight estimate in this case.

In this section, we present our approach for bounding memory interference based on the aforementioned observation. We first analyze the memory interference delay using two different approaches: *request-driven* (Sec. IV-A) and *job-driven* (Sec. IV-B). Then by combining them, we present a response-time-based schedulability analysis that tightly bounds the worst-case memory interference delay of a task.

A. Request-Driven Bounding Approach

The request-driven approach focuses on the number of memory requests generated by a task τ_i (H_i) and the amount of additional delay imposed on each request of τ_i. In other words, it estimates the total interference delay by $H_i \times$(per-request

[4] These assumptions will be relaxed in future work.

[5] This assumption is required to bound the re-ordering effect of the memory controller, which will be described in Section IV-A.

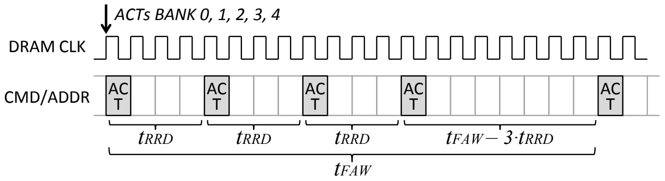


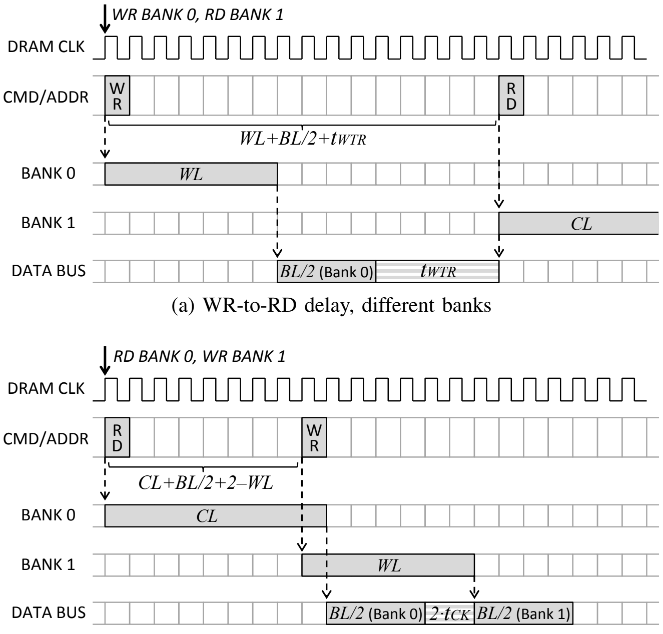
Fig. 2: Inter-bank row-activate timing constraints

interference delay), where the per-request delay is bounded by using DRAM and processor parameters, not by using task parameters of other tasks.

The interference delay for a memory request generated by a processor core p can be categorized into two types: *inter-bank* and *intra-bank*. If there is one core q that does not share any bank partitions with p, the core q only incurs inter-bank memory interference delay to p. If there is another core q' that shares bank partitions with p, the core q' incurs intra-bank memory interference. We present analyses on the two types of interference delay and calculate the total interference delay based on them.

Inter-bank interference delay: Suppose that a core p is assigned dedicated bank partitions. When a memory request is generated by one task on p, the request is enqueued into the request queue of the appropriate DRAM bank. Then, a sequence of DRAM commands is generated based on the type of the request, i.e., one command (RD/WR) for a row-hit request, and three commands (PRE, ACT, RD/WR) for a row-conflict request. At the bank scheduler, there is no interference delay from other cores because p does not share its banks. In contrast, once a command of the request is sent to the channel scheduler, it can be delayed by the commands from other banks, because the FR-FCFS policy at the channel scheduler issues *ready* commands (with respect to the channel timing constraints) in the order of arrival time. The amount of delay imposed on each DRAM command is determined by the following factors:

- *Address/command bus scheduling time*: Each DRAM command takes one DRAM clock cycle on the address/command buses. For a PRE command, as it is not affected by other timing constraints, the delay caused by each of the commands that have arrived earlier is:
$$L_{inter}^{PRE} = t_{CK}$$
- *Inter-bank row-activate timing constraints*: The JEDEC standard [12] specifies that there be a minimum separation time of t_{RRD} between two ACTs to different banks, and no more than four ACTs can be issued during t_{FAW} (Figure 2). Thus, in case of an ACT command, the maximum delay from each of the commands that have arrived earlier is:
$$L_{inter}^{ACT} = \max(t_{RRD}, t_{FAW} - 3 \cdot t_{RRD}) \cdot t_{CK}$$
- *Data bus contention and bus turn-around delay*: When a RD/WR command is issued, data is transfered in burst mode on both the rising and falling edges of the DRAM clock signal, resulting in $BL/2$ of delay due to data bus contention.



(a) WR-to-RD delay, different banks

(b) RD-to-WR delay, different banks

Fig. 3: Data bus contention and bus turn-around delay

In addition, if a WR/RD command comes after an RD/WR command, the data flow direction of the data bus needs to be reversed, resulting in *data bus turn-around delay*. Figure 3 depicts the data bus contention and bus turn-around delay in two cases. In case of WR-to-RD, RD needs to wait $WL + BL/2 + t_{WTR}$ cycles. In case of RD-to-WR, WR needs to wait $CL + BL/2 + 2 - WL$ cycles.[6] Therefore, for a WR/RD command, the maximum delay from each of the commands that have arrived earlier is:
$$L_{inter}^{RW} = \max(WL + BL/2 + t_{WTR},\\ CL + BL/2 + 2 - WL) \cdot t_{CK}$$

Using these parameters, we derive the inter-bank interference delay imposed on each memory request of a core p. Recall that each memory request may consist of up to three DRAM commands: PRE, ACT and RD/WR. Each command of a request can be delayed by all commands that have arrived earlier at other banks. The worst-case delay for p's request occurs when (i) a request of p arrives after the arrival of the requests of all other cores that do not share banks with p, and (ii) each previous request causes PRE, ACT and RD/WR commands. Therefore, the worst-case per-request inter-bank interference delay for a core p, RD_p^{inter}, is given by:
$$RD_p^{inter} = \sum_{\substack{\forall q:\, q \neq p \,\wedge \\ shared(q,p)=\emptyset}} \left(L_{inter}^{PRE} + L_{inter}^{ACT} + L_{inter}^{RW} \right) \quad (1)$$

Intra-bank interference delay: Memory requests to the same

[6] The bound can be made tighter if we know the exact write-handling policy. Many controllers handle the write requests in batches when the write buffer is close to full so that the bus turn-around delay can be amortized across many requests [20].

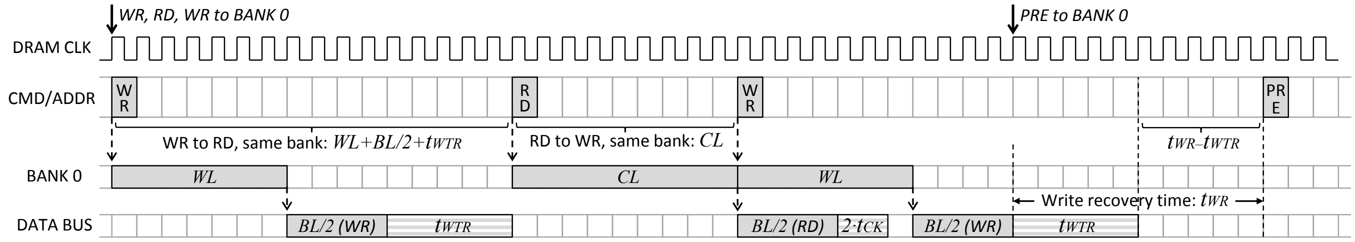


Fig. 4: Timing diagram of consecutive row-hit requests

bank are queued into the bank request buffer and their service order is determined by the bank scheduler. A lower-priority request should wait until all higher priority requests are completely serviced by the bank. The delay caused by each higher-priority request includes (i) the inter-bank interference delay for the higher priority request, and (ii) the service time of the request within the DRAM bank. The inter-bank interference delay can be calculated by Eq. (1). The service time within the DRAM bank depends on the type of the request:

- *Row-hit service time*: The row-hit request is for a requested column already in the row-buffer. Hence, it can simply read/write its column. In case of read, RD takes $CL+BL/2$ for data transfer and may cause 2 cycles of delay to the next request for data bus turn-around time [12]. In case of write, WR takes $WL+BL/2$ for data transfer and may cause $\max(t_{WTR}, t_{WR})$ of delay to the next request for bus turn-around or write recovery time, depending on the type of the next request. Thus, in the worst case, the service time for one row-hit request is:

$$\begin{aligned} L_{hit} = &\max\{CL + BL/2 + 2, \\ &WL + BL/2 + \max(t_{WTR}, t_{WR})\} \cdot t_{CK} \end{aligned}$$

- *Row-conflict service time*: The row-conflict request must issue PRE and ACT commands to open a row before accessing a column, and it takes $t_{RP} + t_{RCD}$. Thus, the worst-case service time for one row-conflict request is:

$$L_{conf} = (t_{RP} + t_{RCD}) \cdot t_{CK} + L_{hit}$$

- *Consecutive row-hit requests*: If m row-hit requests are present in the memory request buffer, their service time is much smaller than $m \cdot L_{hit}$. Due to the data bus turn-around time, the worst-case service time happens when the requests alternate between read and write, as depicted in Figure 4. WR followed by RD causes $WL + BL/2 + t_{WTR}$ of delay to RD, and RD followed by WR causes *CL* of delay to WR. As WR-to-RD causes larger delay than RD-to-WR in DDR3 SDRAM [12, 20], m row-hits takes $\lceil \frac{m}{2} \rceil \cdot (WL+BL/2+t_{WTR}) + \lfloor \frac{m}{2} \rfloor \cdot CL$ cycles. In addition, if a PRE command is the next command to be issued after the m row-hits, it needs to wait an extra $t_{WR} - t_{WTR}$ cycles due to the write recovery time. Therefore, the worst-case service time for m consecutive row-hit requests is:

$$\begin{aligned} L_{conhit}(m) = \{\, &\lceil m/2 \rceil \cdot (WL + BL/2 + t_{WTR}) + \\ &\lfloor m/2 \rfloor \cdot CL + (t_{WR} - t_{WTR})\} \cdot t_{CK} \end{aligned}$$

Under the FR-FCFS policy, the bank scheduler serves row-conflict requests in the order of their arrival times. When row-hit requests arrive at the queue, the bank scheduler *re-orders* memory requests such that row-hits are served earlier than older row-conflicts. For each open row, the maximum row-hit requests that can be generated in a system is represented as N_{cols}/BL, where N_{cols} is the number of columns in one row. This is due to the fact that, as described in the system model, (i) each task is assumed to have enough cache space to store one row of each bank assigned to it, (ii) the memory request addresses are aligned to the size of *BL*, and (iii) tasks do not share memory. Once the tasks that have their data in the currently open row fetch all columns in the open row into their caches, all the subsequent memory accesses to the row will be served at the cache level and no DRAM requests will be generated for those accesses. If one of the tasks accesses a row different from the currently open one, this memory access causes a row-conflict request so that the re-ordering effect no longer occurs. In many systems, as described in [27, 25, 6], the re-ordering effect can also be bounded by a hardware threshold N_{cap}, which caps the number of re-ordering between requests. Therefore, the maximum number of row-hits that can be prioritized over older row-conflicts is:

$$N_{reorder} = \min\left(N_{cols}/BL, N_{cap}\right) \tag{2}$$

We now analyze the intra-bank interference delay for each memory request generated by a processor core p. Within a bank request buffer, each request of p can be delayed by both the re-ordering effect and the previous memory requests in the queue. Therefore, the worst-case per-request interference delay for a core p (RD_p^{intra}) is calculated as follows:

$$RD_p^{intra} = reorder(p) + \sum_{\substack{\forall q:\, q\neq p\, \wedge \\ shared(q,p)\neq\emptyset}} \left(L_{conf} + RD_q^{inter}\right) \tag{3}$$

$$reorder(p) = \begin{cases} 0 & \text{if } \nexists q : q \neq p \wedge shared(q,p) \neq \emptyset \\ L_{conhit}(N_{reorder}) + \sum\limits_{\substack{\forall q:\, q\neq p\, \wedge \\ shared(q,p)=\emptyset}} L_{inter}^{RW} \cdot N_{reorder} & \text{otherwise} \end{cases} \tag{4}$$

In (3), the summation part calculates the delay from memory requests that can be queued before the arrival of p's request. It considers processor cores that share bank partitions with p. Since row-conflict causes a longer delay, the worst-case delay from each of the older requests is the sum of the row-conflict service time (L_{conf}) and the per-request inter-bank interference delay (RD_q^{inter}). The function $reorder(p)$ calculates the delay from the re-ordering effect. As shown in (4), it gives zero if there is no core sharing bank partitions with p. Otherwise, it calculates the re-ordering effect as the sum of the consecutive row-hits' service time ($L_{conhit}(N_{reorder})$) and the inter-bank delay for the row-hits ($\sum L_{inter}^{RW} \cdot N_{reorder}$).

Total interference delay: A memory request from a core p experiences both inter-bank and intra-bank interference delay. Hence, the worst-case per-request interference delay for p, RD_p, is represented as follows:

$$RD_p = RD_p^{inter} + RD_p^{intra} \tag{5}$$

Since RD_p is the worst-case delay for each request, the total memory interference delay of τ_i is upper bounded by $H_i \cdot RD_p$.

B. Job-Driven Bounding Approach

The job-driven approach focuses on how many interfering memory requests are generated during a task's job execution time. In the worst case, every memory request from other cores can delay the execution of a task running on a specific core. Therefore, by capturing the maximum number of requests generated by the other cores during a time interval t, the job-driven approach bounds the memory interference delay that can be imposed on tasks running on a specific core in any time interval t.

We define $A_p(t)$, which is the maximum number of memory requests generated by the core p during a time interval t as:

$$A_p(t) = \sum_{\forall \tau_i \in task(p)} \left\lceil \frac{t}{T_i} \right\rceil \cdot H_i \tag{6}$$

Note that this calculation is not overly pessimistic, because we do not make assumptions on memory access patterns (e.g. access rate or distribution). It is possible to add this type of assumption, such as the specific memory access pattern of the tasks [9, 5] or using memory request throttling mechanisms [44, 10, 43]. This will help us to calculate a tighter $A_p(t)$, while other equations in our work can be used independent of such additional assumptions.

Inter-bank interference delay: The worst-case inter-bank interference delay imposed on a core p during a time interval t is represented as follows:

$$JD_p^{inter}(t) = \sum_{\substack{\forall q:\, q \neq p \,\wedge \\ shared(q,p)=\emptyset}} A_q(t) \cdot \left(L_{inter}^{ACT} + L_{inter}^{RW} + L_{inter}^{PRE} \right) \tag{7}$$

In this equation, the summation considers processor cores that do not share bank partitions with p. The other cores sharing banks with p will be taken into account in Eq. (8). The number of memory requests generated by other cores ($A_q(t)$) is multiplied by the maximum inter-bank interference delay from each of these requests ($L_{inter}^{ACT} + L_{inter}^{RW} + L_{inter}^{PRE}$).

Intra-bank interference delay: The worst-case intra-bank interference delay imposed on a core p during t is as follows:

$$JD_p^{intra}(t) = \sum_{\substack{\forall q:\, q \neq p \,\wedge \\ shared(q,p)\neq\emptyset}} \left(A_q(t) \cdot L_{conf} + JD_q^{inter}(t) \right) \tag{8}$$

Eq. (8) considers other cores that share bank partitions with p. The number of requests generated by each of these cores during t is calculated as $A_q(t)$. Since a row-conflict request causes larger delay than a row-hit one, $A_q(t)$ is multiplied by the row-conflict service time L_{conf}. In addition, JD_q^{inter} is added because each interfering core q itself may be delayed by inter-bank interference depending on its bank partitions. Note that the re-ordering effect of the bank scheduler does not need to be considered here because Eq. (8) captures the worst case where all the possible memory requests generated by other cores arrived ahead of any request from p.

Total interference delay: The worst-case memory interference delay is the sum of the worst-case inter-bank and intra-bank delays. Therefore, the memory interference delay for a core p during a time interval t, $JD_p(t)$, is upper bounded by:

$$JD_p(t) = JD_p^{inter}(t) + JD_p^{intra}(t) \tag{9}$$

C. Response-Time Based Schedulability Analysis

We have presented the request-driven and the job-driven approaches to analyze the worst-case memory interference delay. Since each of the two approaches bounds the interference delay by itself, a tighter upper bound can be obtained by taking the smaller result from the two approaches. Based on the analyses of the two approaches, the iterative response time test [14] is extended as follows to incorporate the memory interference delay:

$$\begin{aligned} R_i^{k+1} = C_i &+ \sum_{\tau_j \in hp(\tau_i)} \left\lceil \frac{R_i^k}{T_j} \right\rceil \cdot C_j \\ &+ \min \left\{ H_i \cdot RD_p + \sum_{\tau_j \in hp(\tau_i)} \left\lceil \frac{R_i^k}{T_j} \right\rceil \cdot H_j \cdot RD_p,\ JD_p(R_i^k) \right\} \end{aligned} \tag{10}$$

where R_i^k is the worst-case response time of τ_i at the k^{th} iteration, and p is $proc(\tau_i)$. The test terminates when $R_i^{k+1} = R_i^k$. The task τ_i is schedulable if its response time does not exceed its deadline: $R_i^k \leq D_i$. The first and the second terms are the same as the classical response time test. In the third term, the memory interference delay for τ_i is bounded by using the two approaches. The request-driven approach bounds the delay with the addition of of $H_i \cdot RD_p$ and $\sum \lceil \frac{R_i^k}{T_j} \rceil \cdot H_j \cdot RD_p$, which is the total delay imposed on τ_i and its higher priority tasks. The job-driven approach bounds the delay by $JD_p(R_i^k)$, that captures the total delay incurred during τ_i's response time.

We can make the following observations from our analysis: (i) memory interference increases with the number of cores, (ii) tasks running on the same core do not interfere with each other, and (iii) the use of bank partitioning reduces the interference delay. These observations lead to an efficient task allocation under partitioned scheduling by co-locating memory-intensive tasks on the same core with dedicated DRAM banks. For a description of how a task allocation can reduce memory interference, please refer to [15].

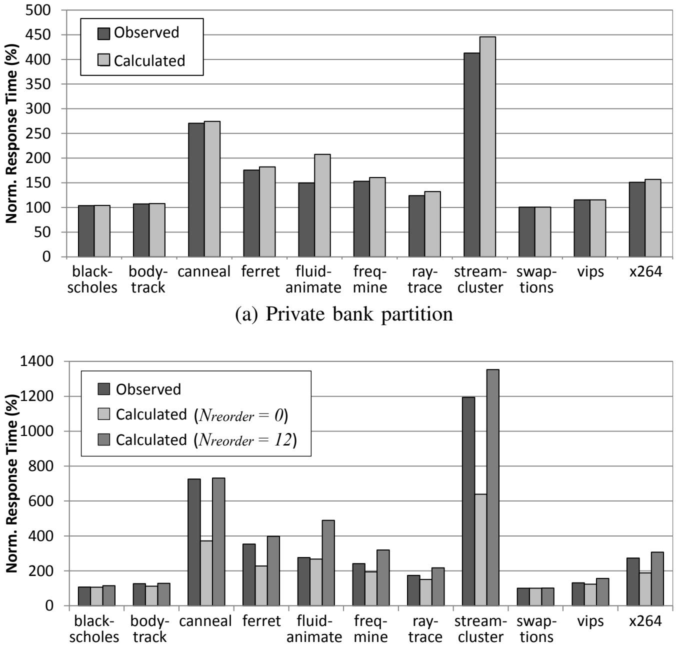
V. Evaluation

In this section, we show how our analysis effectively bounds memory interference delay in a real system. We first describe our experimental setup and then present the results.

A. Experimental Setup

The target system is equipped with the Intel Core i7-2600 quad-core processor.[7] The on-chip memory controller of the processor supports dual memory channels, but by installing a single DIMM, only one channel is activated in accordance with our system model.[8] The system uses a single DDR3-1333 DIMM that consists of 2 ranks and 8 banks per each rank. The timing parameters of the DIMM are shown in Table I. We used the latest version of Linux/RK [30, 33] for software cache and bank partitioning [16, 39].[9] Cache partitioning divides the shared L3 cache of the processor into 32 partitions, and bank partitioning divides the DRAM banks into 16 partitions (1 DRAM bank per partition). For the measurement tool, we used the Linux/RK profiler [17] that records execution times and memory accesses (LLC misses) using hardware performance counters. To reduce measurement inaccuracies, we disabled the hardware prefetcher, simultaneous multithreading, and dynamic clock frequency scaling of the processor. All unrelated system services such as GUI and networking were also disabled. In addition, we used the memory reservation mechanism of Linux/RK [11, 18] to protect each application against unexpected page swap-outs.

We use the eleven PARSEC benchmarks [8] and the two types of synthetic tasks (*memory-intensive* and *memory-non-intensive*) for our experiment.[10] Our focus is on analyzing the memory interference delays on the benchmarks. Each benchmark is assigned to Core 1, and the three instances of synthetic tasks are assigned to the other cores (Core 2, 3, 4) to generate interfering memory requests. To meet the memory size requirement of the benchmarks[11], each benchmark is assigned 20 private cache partitions. The synthetic tasks are each assigned 4 private cache partitions. Each of the benchmarks and the synthetic tasks is assigned 1 bank partition, and we evaluate two cases where tasks share or do not share bank partitions. When running in isolation, the synthetic memory-intensive task generates up to 40K DRAM requests per msec[12] (combination of read and write) and the memory-non-intensive task generates up to 1K DRAM requests per msec.



(a) Private bank partition

(b) Shared bank partition

Fig. 5: Response times with three memory-intensive tasks

B. Results

We first evaluate the response times of benchmarks with the three memory-intensive tasks. Figure 5 compares the maximum observed response times with the calculated response times from our analysis, when three memory-intensive tasks are running in parallel. The x-axis of each subgraph denotes the benchmark names, and the y-axis shows the normalized response time of each benchmark. As each benchmark is solely assigned to Core 1, the response time increase is equal to the amount of memory interference suffered from other cores. The difference between the observed and calculated values represents the pessimism embedded in our analysis. Figure 5(a) shows the response times with a private bank partition per core. We observed up to 4.1x of response time increase in the target system (*canneal*). The results from our analysis are only 8% more than the observed values on average. The worst over-estimation is found in *fluidanimate*. We suspect that this over-estimation comes from the varying memory access patten of the benchmark, because our analysis considers the worst-case memory access scenario. Recall that our analysis bounds memory interference based on two approaches: request-driven and job-driven. In this experiment, as the memory-intensive tasks generate an enormous number of memory requests, the

[7] Although the cores of this processor are not fully timing compositional, in practice, the experimental results show that our analysis is effective in bounding memory interference. Furthermore, we have not observed any timing anomalies in our experiments.

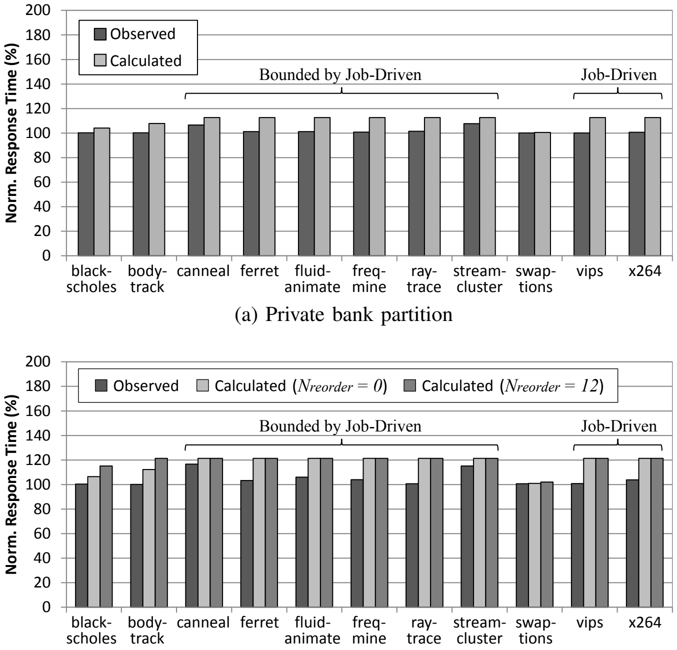
[8] This is why the DRAM address mapping in Figure 1(c) does not have a bit for channel selection.

[9] Linux/RK is available at https://rtml.ece.cmu.edu/redmine/projects/rk.

[10] Two PARSEC benchmarks, *dedup* and *facesim*, are excluded from the experiment due to their frequent disk accesses for data files.

[11] Software cache partitioning simultaneously partitions the entire physical memory space into the number of cache partitions. Therefore the spatial memory requirement of a task determines the minimum number of cache partitions for that task [16].

[12] The memory-intensive task is a modified version of the *stream* benchmark [24]. Since it has very high row-buffer locality with little computations, "40K requests per msec" is likely close to the maximum possible value that a single core can generate with a single bank partition in the target system.



(a) Private bank partition

(b) Shared bank partition

Fig. 6: Response times with three memory-non-intensive tasks

response times of all benchmarks are bounded by the request-driven approach. When only the job-driven approach is used, the results are unrealistically pessimistic (>10000x; not shown in the figure for simplicity). Thus, these experimental results show the advantage of the request-driven approach.

Figure 5(b) illustrates the response times when all cores share the same bank partition. With bank sharing, we observed up to 12x of response time increase in the target platform. To calculate the response time when a bank partition is shared, the re-ordering window size $N_{reorder}$ is required. If we disregard the re-ordering effect of FR-FCFS in this platform ($N_{reorder} = 0$), the analysis generates overly optimistic values. In case of *canneal*, the analysis that does not account for the re-ordering effect results in a 555% lower value than the observed one. As the precise number of $N_{reorder}$ is not publicly available,[13] we inferred this value based on our analysis. When $N_{reorder} = 12$, our analysis bounds all cases. This implies that the re-ordering window size of the target hardware is *at least* 12. The results in the bank sharing case shows the importance of considering the request re-ordering effect of the DRAM system in designing a predictable multi-core system.

We next evaluate the response times with the memory-non-intensive tasks. Figure 6(a) and Figure 6(b) depict the response times with a private and a shared bank partition, respectively. In contrast to the memory-intensive case, the smallest upper-bounds on the response times are mostly obtained by the job-driven approach due to the low number of interfering memory requests. The average over-estimates are 8% and 13% for a private and a shared bank, respectively. The experimental results show that our analysis bounds memory interference delay with low pessimism in a real hardware platform, under both high and low memory contentions.

VI. Related Work

With multi-core processors being the norm today, the real-time systems research community has been increasingly interested in the impact of contention on resources in the memory system on the timing of software.

Researchers have developed special (non-COTS) components of memory systems for real-time systems. The Predator memory controller [3] uses credit-based arbitration and closes an open row after each access. The AMC memory controller [31] spreads the data of a single cache block across all DRAM banks so as to reduce the impact of interference by serializing all memory requests. The PRET DRAM controller [34] hardware partitions banks among cores for predictability. Researchers have also proposed techniques that modify a program and carefully set up time-triggered schedules so that there is no instant where two processor cores have outstanding memory operations [36].

We have heard, however, a strong interest from software practitioners in techniques that can use COTS multi-core processors and existing applications without requiring modifications and, therefore, this has been the focus of this paper. In this context, some previous work considers the entire memory system as a single resource, such that a processor core requests this resource when it generates a cache miss and it must hold this resource exclusively until the data of the cache miss are delivered to the processor core that requested it [32, 5, 9, 37, 23]. They commonly assumed that each memory request takes a constant service time and memory requests from multiple cores are serviced in the order of their arrival time. However, these assumptions may lead to overly pessimistic or optimistic estimates in COTS systems, where the service time of each memory request varies and the memory controller re-orders the memory requests [25].

Instead of considering the memory system as a single resource, recent work [42] makes a more realistic assumption about the memory system, where the memory controller has one request queue per DRAM bank and one system-wide queue connected to the per-bank queues. That analysis, however, only considers the case where each processor core is assigned a private DRAM bank. Unfortunately, the number of DRAM banks is growing more slowly than the number of cores, and the memory space requirement of a workload in a core may exceed the size of a single bank. Due to this limited availability of DRAM banks, it is necessary to consider sharing of DRAM banks among multiple cores. With bank sharing, memory requests can be re-ordered in the per-bank queues, thereby increasing memory request service times. The work in [42] unfortunately does not model this request re-ordering effect. In this paper, we have eliminated this limitation.

Finally, there has been recent work in the architecture community on the design of memory controllers that can dynamically estimate application slowdowns [38]. These designs

[13] As given in (2), $N_{reorder}$ can be bounded without the knowledge of the N_{cap} value. The DRAM used in this platform has N_{cols} of 1024 and BL of 8, so the $N_{reorder}$ value does not exceed 128.

do not aim to provide worst-case bounds and can underestimate memory interference. Future memory controllers might incorporate ideas like batching and thread prioritization (e.g., [28, 19]). This will lead to a different analysis, which could be interesting future work that builds on ours.

VII. Conclusions

In this paper, we have presented an analysis for bounding memory interference in a multi-core systems. Our analysis is based on a realistic memory model, which considers the JEDEC DDR3 SDRAM standard, the FR-FCFS policy of the memory controller, and shared/private DRAM banks. To provide a tight upper-bound on the memory interference delay, our analysis uses the combination of the request-driven and job-driven approaches. Experimental results from a real hardware platform show that our analysis can estimate the memory interference delay (with only 8% of over-estimation on average under severe memory contention).

As multi-core processors are already ubiquitous, the contention on shared main memory should be seriously considered. We believe that our analysis based on a realistic memory model can be effectively used for designing predictable multicore real-time systems. For future work, we plan to explore the effect of hardware prefetchers on memory interference delay. Considering a non-timing-compositional architecture that allows out-of-order execution and multiple outstanding cache misses is also an important future research issue.

References

[1] Micron 2Gb DDR3 Component: MT41J256M8-15E. http://download.micron.com/pdf/datasheets/dram/ddr3/2Gb_DDR3_SDRAM.pdf.

[2] OSEK/VDX OS. http://portal.osek-vdx.org/files/pdf/specs/os223.pdf.

[3] B. Åkesson, K. Goossens, and M. Ringhofer. Predator: a predictable SDRAM memory controller. In *CODES+ISSS*, 2007.

[4] S. Altmeyer, R. Davis, and C. Maiza. Cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. In *RTSS*, 2011.

[5] B. Andersson, A. Easwaran, and J. Lee. Finding an upper bound on the increase in execution time due to contention on the memory bus in COTS-based multicore systems. *SIGBED Review*, 7(1):4, 2010.

[6] R. Ausavarungnirun et al. Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems. In *ISCA*, 2012.

[7] B. Bhat and F. Mueller. Making DRAM refresh predictable. In *ECRTS*, 2010.

[8] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*, 2008.

[9] D. Dasari, B. Andersson, V. Nélis, S. M. Petters, A. Easwaran, and J. Lee. Response time analysis of COTS-based multicores considering the contention on the shared memory bus. In *IEEE TrustCom*, 2011.

[10] E. Ebrahimi et al. Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. In *ASPLOS*, 2010.

[11] A. Eswaran and R. Rajkumar. Energy-aware memory firewalling for QoS-sensitive applications. In *ECRTS*, 2005.

[12] JEDEC. DDR3 SDRAM Standard. http://www.jedec.org.

[13] M. K. Jeong et al. Balancing DRAM locality and parallelism in shared memory CMP systems. In *HPCA*, 2012.

[14] M. Joseph and P. K. Pandya. Finding response times in a real-time system. *Comput. J.*, 29(5):390–395, 1986.

[15] H. Kim et al. Bounding memory interference delay in COTS-based multi-core systems. Technical Report CMU/SEI-2014-TR-003, Software Engineering Institute, Carnegie Mellon University, 2014.

[16] H. Kim, A. Kandhalu, and R. Rajkumar. A coordinated approach for practical OS-level cache management in multi-core real-time systems. In *ECRTS*, 2013.

[17] H. Kim, J. Kim, and R. R. Rajkumar. A profiling framework in Linux/RK and its application. In *RTSS@Work*, 2012.

[18] H. Kim and R. Rajkumar. Shared-page management for improving the temporal isolation of memory reservations in resource kernels. In *RTCSA*, 2012.

[19] Y. Kim et al. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *MICRO*, 2010.

[20] C. J. Lee et al. DRAM-aware last-level cache writeback: Reducing write-caused interference in memory systems. Technical Report TR-HPS-2010-002, UT Austin, 2010.

[21] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[22] L. Liu et al. A software memory partition approach for eliminating bank-level interference in multicore systems. In *PACT*, 2012.

[23] M. Lv, G. Nan, W. Yi, and G. Yu. Combining abstract interpretation with model checking for timing analysis of multicore software. In *RTSS*, 2010.

[24] J. D. McCalpin. STREAM: Sustainable memory bandwidth in high performance computers. http://www.cs.virginia.edu/stream.

[25] T. Moscibroda and O. Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *USENIX Security Symposium*, 2007.

[26] S. P. Muralidhara et al. Reducing memory interference in multicore systems via application-aware memory channel partitioning. In *MICRO*, 2011.

[27] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *MICRO*, 2007.

[28] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *ISCA*, 2008.

[29] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair queuing memory systems. In *MICRO*, 2006.

[30] S. Oikawa and R. Rajkumar. Linux/RK: A portable resource kernel in Linux. In *IEEE Real-Time Systems Symposium (RTSS) Work-In-Progress*, 1998.

[31] M. Paolieri, E. Quiñones, F. Cazorla, and M. Valero. An analyzable memory controller for hard read-time CMPs. *IEEE Embedded Systems Letters*, 1(4):86–90, 2010.

[32] R. Pellizzoni, A. Schranzhofer, J. Chen, M. Caccamo, and L. Thiele. Worst case delay analysis for memory interference in multicore systems. In *DATE*, 2010.

[33] R. Rajkumar et al. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *SPIE/ACM Conference on Multimedia Computing and Networking*, 1998.

[34] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee. PRET DRAM controller: Bank privatization for predictability and temporal isolation. In *CODES+ISSS*, 2011.

[35] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *ISCA*, 2000.

[36] J. Rosén, A. Andrei, P. Eles, and Z. Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *RTSS*, 2007.

[37] S. Schliecker, M. Negrean, and R. Ernst. Bounding the shared resource load for the performance analysis of multiprocessor systems. In *DATE*, 2010.

[38] L. Subramanian et al. MISE: Providing performance predictability and improving fairness in shared main memory systems. In *HPCA*, 2013.

[39] N. Suzuki et al. Coordinated bank and cache coloring for temporal protection of memory accesses. In *ICESS*, 2013.

[40] R. Wilhelm et al. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):966–978, 2009.

[41] Windriver VxWorks. http://www.windriver.com.

[42] Z. P. Wu, Y. Krish, and R. Pellizzoni. Worst case analysis of DRAM latency in multi-requestor systems. In *RTSS*, 2013.

[43] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In *ECRTS*, 2012.

[44] X. Zhang, S. Dwarkadas, and K. Shen. Hardware execution throttling for multi-core resource management. In *USENIX ATC*, 2009.

[45] W. K. Zuravleff and T. Robinson. Controller for a synchronous DRAM that maximizes throughput by allowing memory requests and commands to be issued out of order. *U.S. Patent Number 5,630,096*, 1997.