Bounding and Reducing Memory Interference in COTS-based Multi-Core Systems*

Hyoseung Kim† hyoseung@cmu.edu
Dionisio de Niz‡ dionisio@sei.cmu.edu
Björn Andersson‡ baandersson@sei.cmu.edu
Mark Klein‡ mk@sei.cmu.edu
Onur Mutlu† onur@cmu.edu
Ragunathan (Raj) Rajkumar† raj@ece.cmu.edu

†Electrical and Computer Engineering, Carnegie Mellon University
‡Software Engineering Institute, Carnegie Mellon University

Abstract

In multi-core systems, main memory is a major shared resource among processor cores. A task running on one core can be delayed by other tasks running simultaneously on other cores due to interference in the shared main memory system. 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 reduce this interference and provide an upper bound on the worst-case interference on a multi-core platform that uses a commercial-off-the-shelf (COTS) DRAM 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. We find that memory interference can be significantly reduced by (i) partitioning DRAM banks, and (ii) co-locating memory-intensive tasks on the same processing core. Based on these observations, we develop a memory interference-aware task allocation algorithm for reducing memory interference. We evaluate our approach on a COTS-based multi-core platform running Linux/RK. Experimental results show that the predictions made by our approach are close to the measured worst-case interference under workloads with both high and low memory contention. In addition, our memory interference-aware task allocation algorithm provides a significant improvement in task schedulability over previous work, with as much as 96% more tasksets being schedulable.

*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. 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-0001596

1 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, 54, 40, 45, 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 model has assumptions that are not representative of commercial-off-the-shelf (COTS) memory components. Specifically, in modern systems, COTS DRAM systems have been widely used as main memory to cope with high performance and capacity demands. The DRAM system contains multiple resources such as ranks, banks, and buses; the access time of these resources depends on previous accesses. In addition, for the scheduling of memory requests, many memory controllers use schemes based on the First-Ready First-Come First-Serve (FR-FCFS) policy [43, 36, 32, 56, 46], where memory requests arriving early may be serviced later than the 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 and reducing 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 an 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* approach focused on the task's own memory requests and a *job-driven* approach focused on interfering memory requests during the task's execution. Combining them, our analysis yields a tighter 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 [29, 48, 53, 17, 52]. 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. In the evaluation section, we show the effectiveness of our analysis on a well-known COTS multicore platform.

In addition, we develop a memory interference-aware task allocation algorithm that accommodates memory interference delay during the allocation phase. The key idea of our algorithm is to

co-locate memory-intensive tasks on the same core with dedicated DRAM banks. This approach reduces the amount of memory interference among tasks, thereby improving task schedulability. Experimental results indicate that our algorithm yields a significant improvement in task schedulability over existing approaches such as in [38], with as much as 96% more tasksets being schedulable.

The rest of this paper is organized as follows. Section 2 explains how modern DRAM systems work. Section 3 describes the system and task model used in this paper. Section 4 presents how we bound memory interference. Section 5 provides our memory interference-aware allocation algorithm. Section 6 provides a detailed evaluation. Section 7 reviews related work. Section 8 concludes the paper.

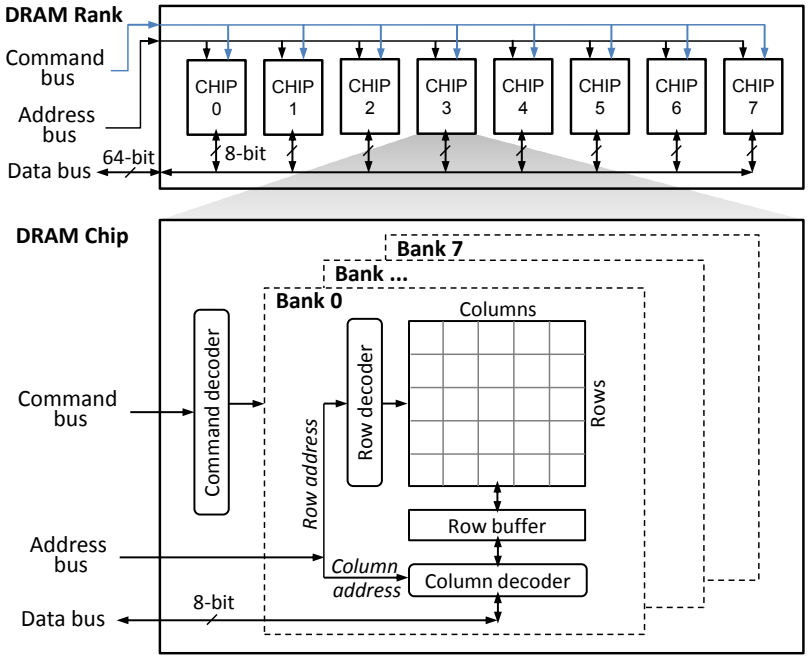
2 Background on DRAM Systems

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 [43, 36, 32, 34].

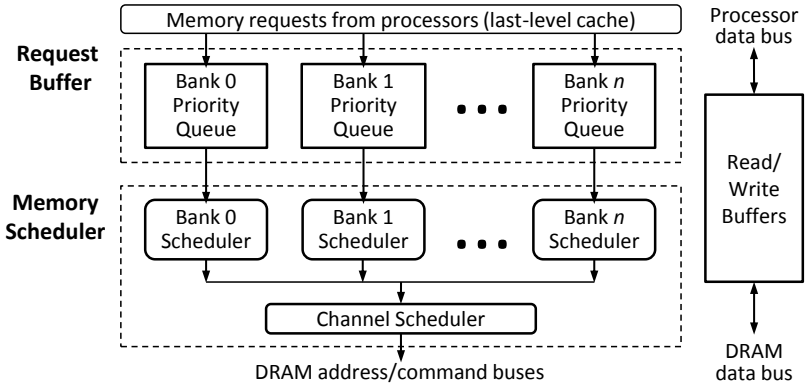
2.1 DRAM Organization

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/chip $\times$ 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 [13].

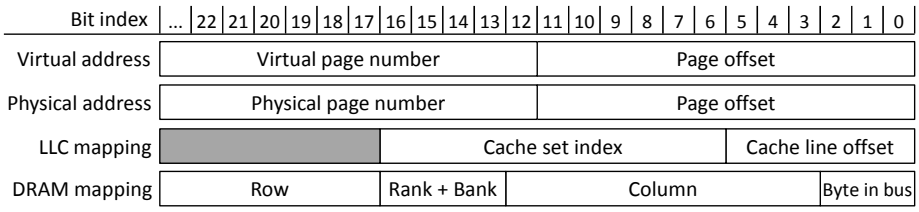
The DRAM access latency varies depending on which row is currently stored in the row-buffer of a requested bank. If a memory request accesses a row already in the row-buffer, the request is directly serviced from the row-buffer, resulting in a short latency. This case is called a *row hit*. If the request 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



(a) DRAM device organization



(b) Logical structure of a DRAM controller



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

Figure 1: Modern DDR SDRAM systems

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.

2.2 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 typically 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 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) [43, 36] 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.

[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 [36] or as multiple decoupled structures [34, 35, 6].

Note that, in order to prevent starvation, many DRAM controllers impose a limit on the number of consecutive row-hit requests that can be serviced before a row-conflict request [34, 46]. We will discuss such a limit in Section 4. 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 newest row-miss request has the lowest priority.

2.3 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 [33, 29, 14]. 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 [29, 48, 53, 17, 52] is a technique used to avoid the delays due to shared banks. By dedicating a set of specific DRAM banks to each core, bank partitioning prevents both (i) the unexpected close of the currently-open row and (ii) the negative effect of request re-ordering. Therefore, with bank partitioning, bank-level interference among tasks simultaneously executing on different cores can be effectively eliminated. The key to bank partitioning 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 core by allocating corresponding physical pages to the tasks of the core. Each bank partition may comprise one or more DRAM banks. If the memory requirement of the tasks of a core is larger than the size of one DRAM bank, each bank partition can be configured to have multiple DRAM banks to sufficiently satisfy the memory requirement with a single bank partition. However, due to the resulting smaller number of bank partitions, it may not be feasible to assign a dedicated bank partition to each core.

[2]The DRAM mapping of Figure 1(c) is for the single-channel configuration in this system. Section 6 gives more details on this system.

In our work, we therefore consider not only dedicated DRAM banks to reduce memory interference delay but also shared banks to cope with the limited number of banks.

3 System Model

We present our system model here. Our system model assumes a multi-core system with the DDR SDRAM sub-system described in Section 2. 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. DRAM consists of one or more ranks. 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

Each DRAM command is assumed to have the same priority and arrival time as the corresponding memory request. 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 1 and are taken from Micron's datasheet [1].

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 data fetched from the DRAM system by cores are stored in the LLC. A single memory request can fetch one entire cache line from the DRAM because of the burst-mode data transfer. The addresses of memory requests are aligned to the size of BL (burst length). We limit our focus on memory requests from cores and leave DMA (Direct Memory Access) as our future work. In this paper, we assume that each core has a fully timing-compositional architecture as described in [49]. This means that each core is in-order with one outstanding memory request and any delays from shared resources are additive to the task execution times.

[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℃, 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].

Table 1: 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
Activate to precharge delay	t_{RAS}	24	cycles
Row cycle time	t_{RC}	33	cycles
Read to precharge delay	t_{RTP}	5	cycles
Refresh to activate delay	t_{RFC}	160	nsec
Average refresh interval	t_{REFI}	7.8	μsec
Rank-to-rank switch delay	t_{RTRS}	2	cycles

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 [50]. We later show that the use of partitioned scheduling can reduce the amount of memory access interference by co-locating memory-access-intensive tasks.

For the task model, we assume sporadic tasks with constrained deadlines. Any fixed-priority assignment can be used, such as Rate Monotonic [28]. 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 is assumed to have a unique priority ranging from 1 to n.[4] 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.[5] Task τ_i is 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: an upper bound on the 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. When a measurement-based approach is used, C_i and H_i need to be conservatively estimated. Especially

[4] An arbitrary tie-breaking rule can be used to assign a unique priority to each task.
[5] These assumptions will be relaxed in future work.

in a system with a write-back cache, H_i should take into account dirty lines remaining in the cache. 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 [18]. 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 comprises one or more DRAM banks that are not shared with other bank partitions, and is represented as a unique integer in the range from 1 to N_{BP}. It is assumed that the number of DRAM banks in each bank partition and the number of bank partitions assigned to a task do 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 sufficient cache space of its own to store one row of each DRAM bank assigned to it.[6] 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 $bank(q)$

[6] This assumption is required to bound the re-ordering effect of the memory controller, which will be described in Section 4.1.

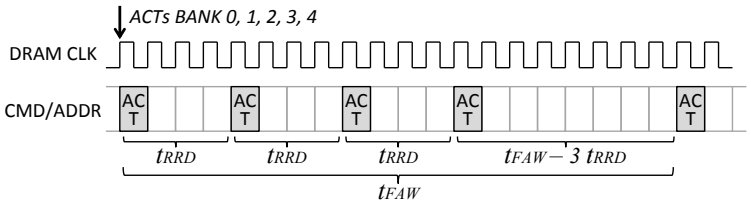


Figure 2: Inter-bank row-activate timing constraints

4 Bounding Memory Interference Delay

The memory interference delay that a task can suffer from other tasks can be bounded 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 tighter bound. 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 tighter bound in this case.

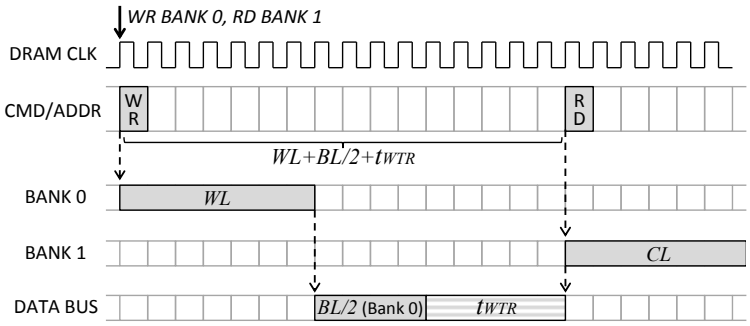
In this section, we present our approach for bounding memory interference based on the aforementioned observation. We first analyze the memory interference delay using two different approaches: *request-driven* (Section 4.1) and *job-driven* (Section 4.2). Then by combining them, we present a response-time-based schedulability analysis that tightly bounds the worst-case memory interference delay of a task.

4.1 Request-Driven Bounding Approach

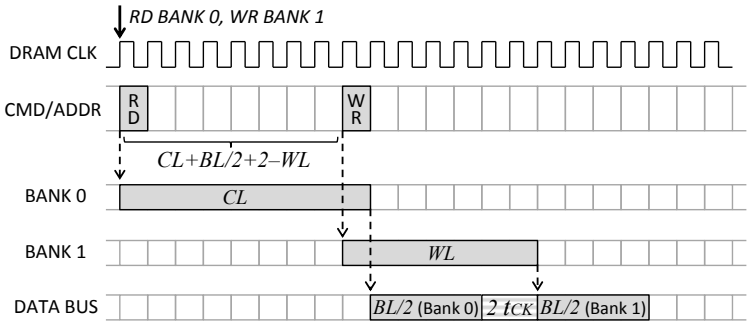
The request-driven approach focuses on the number of memory requests generated by a task τ_i (H_i) and the amount of additional delay imposed on each request of τ_i. In other words, it bounds the total interference delay by $H_i \times$(per-request interference delay), where the per-request delay is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



(a) WR followed by RD, different banks in the same rank



(b) RD followed by WR, different banks in the same rank

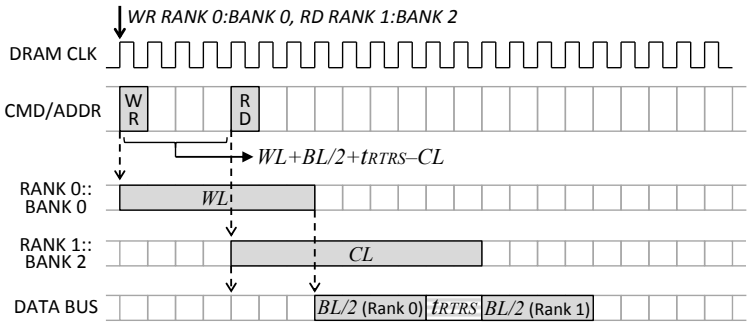
Figure 3: Data bus turn-around delay

When a memory request is generated by one task on p, the request is enqueued into the request queue of the corresponding 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:

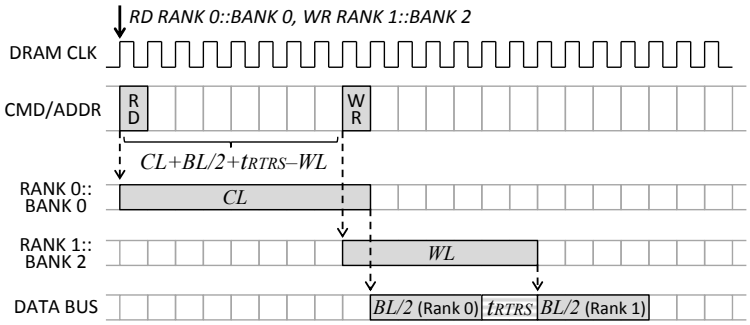
- *Address/command bus scheduling time*: Each DRAM command takes one DRAM clock cycle on the address/command buses. For a PRE command, as it is not affected by other timing constraints, the delay caused by each of the commands that have arrived earlier is:

$$L_{inter}^{PRE} = t_{CK}$$

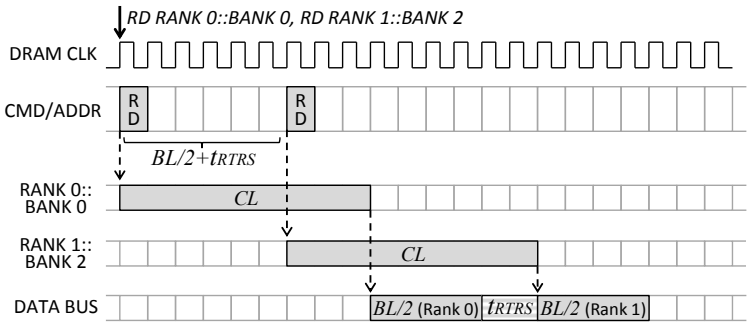
- *Inter-bank row-activate timing constraints*: The JEDEC standard [13] specifies that there be a minimum separation time of t_{RRD} between two ACTs to different banks, and no more than



(a) WR followed by RD, different ranks



(b) RD followed by WR, different ranks



(c) RD followed by RD, different ranks

Figure 4: Rank-to-rank switch delay

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 turn-around and rank-to-rank switch 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. In addition, if a WR/RD command is followed by an RD/WR command to different banks in the same rank, 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. When a WR command is followed by an RD command to different banks in the same rank, RD is delayed by $WL + BL/2 + t_{WTR}$ cycles (Figure 3(a)). When RD is followed by WR, WR is delayed by $CL + BL/2 + 2 - WL$ cycles (Figure 3(b)). If two consecutive WR/RD commands are issued to different ranks, there is *rank-to-rank switch delay*, t_{RTRS}, between the resulting two data transfers. Figure 4 shows the rank-to-rank switch delay in three cases. When WR is followed by RD to different ranks, RD is delayed by $WL + BL/2 + t_{RTRS} - CL$ cycles (Figure 4(a)). When RD is followed by WR, WR is delayed by $CL + BL/2 + t_{RTRS} - WL$ cycles (Figure 4(b)). Lastly, when the two commands are of the same type, the latter is delayed by $BL/2 + t_{RTRS}$ cycles (Figure 4(b)). Therefore, for a WR/RD command, the maximum delay from each of the commands that have arrived earlier is given as follows:

$$\begin{aligned} L_{inter}^{RW} = \max(& WL + BL/2 + t_{WTR}, \\ & CL + BL/2 + 2 - WL, \\ & WL + BL/2 + t_{RTRS} - CL, \\ & CL + BL/2 + t_{RTRS} - WL, \\ & BL/2 + t_{RTRS}) \cdot t_{CK} \end{aligned}$$

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:

$$RD_p^{inter} = \sum_{\substack{\forall q:\, q \neq p \,\wedge \\ shared(q,p)=\emptyset}} \left(L_{inter}^{PRE} + L_{inter}^{ACT} + L_{inter}^{RW} \right) \tag{1}$$

Intra-bank interference delay: Memory requests to the same 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, an RD command takes $CL + BL/2$ for data transfer and may cause 2 cycles of delay to the next request for data bus turn-around time [13]. Note that the read-to-precharge delay (t_{RTP}) does not need to be explicitly considered here because the worst-case delay of an RD command is larger than t_{RTP} in DDR3 SDRAM [13] (or Table 1 for DDR3-1333), i.e., $t_{RTP} < CL + BL/2 + 2$. In case of write, a WR command 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:

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

- *Row-conflict service time*: The row-conflict request should open a row before accessing a column by issuing PRE and ACT commands, which may take up to t_{RP} and t_{RCD} cycles, respectively. Hence, the worst-case service time for one row-conflict request is represented as follows:

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

  If the next request is also row-conflict and issues PRE and ACT commands, constraints on the active-to-precharge delay (t_{RAS}) and the row-cycle time (t_{RC}, a minimum separation between two ACTs in the same bank) should be satisfied. The row-conflict service time L_{conf} satisfies t_{RAS} because $t_{RCD} \cdot t_{CK} + L_{hit}$ is larger than $t_{RAS} \cdot t_{CK}$ in DDR3 SDRAM [13] (or Table 1 for DDR3-1333). L_{conf} also satisfies t_{RC}, because t_{RC} is equal to $t_{RAS} + t_{RP}$ where t_{RP} is time for the PRE command of the next request to be completed.

- *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 5. 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 [13, 26], 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

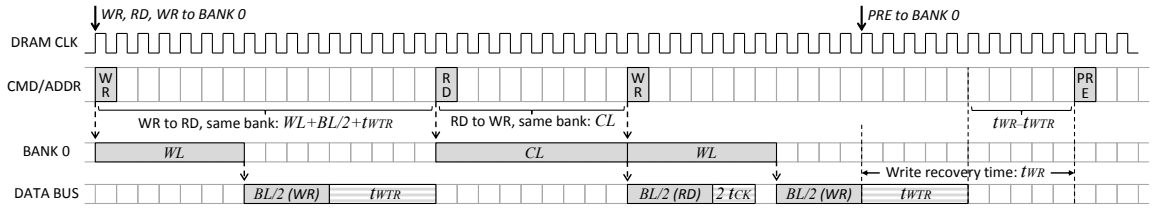


Figure 5: Timing diagram of consecutive row-hit requests

service time for m consecutive row-hit requests is:

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

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 [34, 32, 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:

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

The exact value of N_{cap} is not publicly available on many platforms. Even so, we can still obtain a theoretical bound on $N_{reorder}$ by N_{cols}/BL, the parameters of which are easily found in the JEDEC standard [13].

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:

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

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

In Eq. (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 than row-hit, 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 Eq. (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 ($N_{reorder} \cdot \sum L_{inter}^{RW}$). In addition, since the memory request of p that was originally row-hit could become row-conflict due to interfering requests from cores sharing bank partitions with p, Eq. (4) captures delays for additional PRE and ACT commands ($(t_{RP} + t_{RCD}) \cdot t_{CK}$).

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 $H_i \cdot RD_p$.

4.2 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:

$$A_p(t) = \sum_{\forall \tau_i \in task(p)} \left(\left\lceil \frac{t}{T_i} \right\rceil + 1 \right) \cdot H_i \tag{6}$$

The "+1" term is to capture the carry-in job of each task during a given time interval t. Note that this calculation is quite 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 assumptions, such as the specific memory access pattern of the tasks [9, 5] or using memory request throttling mechanisms [55, 11, 54]. This helps 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:

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

It is worth noting that the job-driven approach will give a tighter bound than the request-driven approach when the number of interfering memory requests from other cores is relatively small compared to the number of the memory requests of the task under analysis. Conversely, in the opposite case, the request-driven approach will give a tighter bound than the job-driven approach. We will compare the results of these two approaches in Section 6.

4.3 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 [16] 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.

4.4 Memory Controllers with Write Batching

Many recent memory controllers handle 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 [26, 57]. Although the modeling of memory controllers using write batching is not within the scope of our work, we believe that our analysis could still be used to bound memory interference in systems with such memory controllers. If a memory controller uses write batching, the worst-case delay of a single

memory operation can be much larger than the one computed by $L_{inter}^{PRE} + L_{inter}^{ACT} + L_{inter}^{RW}$, due to write-buffer draining.[7] However, this does not restrict the applicability of our theory on such memory controllers. We discuss this from two cases as follows.

First, consider a job of task τ_i and how it experiences interference from a job of task τ_j where τ_j is assigned to a different core than τ_i. If the job of τ_i starts its execution at a time when the write buffer is fully filled with the memory requests of the job of τ_j, then the job of τ_i suffers additional interference from at most w memory requests, where w is the size of the write buffer. However, this effect can only happen once per the job of τ_i and be bounded by a constant value. Afterwards, the total number of interfering memory requests remains the same during the execution of the job of τ_i. In addition, since the use of write batching reduces memory bus turn-around delay, it may even shorten the response time of the job of τ_i.

Second, consider a job of task τ_i and how it experiences interference from a job of task τ_j where τ_j is assigned to the same core as τ_i. If the job of τ_i starts its execution at a time when the write buffer is full with the memory requests of the job of τ_j, all the memory requests in the write buffer need to be serviced first, which can delay the execution of the job of τ_i. However, this effect can only happen once per context switch and hence it can be accounted for as a preemption cost.

5 Reducing Memory Interference via Task Allocation

In this section, we present our memory interference-aware task allocation algorithm to reduce memory interference during the allocation phase. Our algorithm is motivated by the following observations we have made from our analysis given in Section 4: (i) memory interference for a task is caused by other tasks running on other cores in parallel, (ii) tasks running on the same core do not interfere with each other, and (iii) the use of bank partitioning reduces the memory 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, the amount of memory interference among tasks can be significantly reduced, thereby providing better schedulability.

Our memory interference-aware allocation algorithm (MIAA) is given in Algorithm 1. MIAA takes three input parameters: Γ is a set of tasks to be allocated, N_P is the number of available processor cores, and N_{BP} is the number of available bank partitions. MIAA returns *schedulable*, if every task in Γ can meet its deadline, and *unschedulable*, if any task misses its deadline.

[7] Note that the write-buffer draining does not completely block read requests until all the write requests are serviced. In a memory controller with write batching, read requests are always exposed to the memory controller, but write requests are exposed to and scheduled by the memory controller only when the write buffer is close to full [26]. Hence, even when the write buffer is being drained, a read request can be scheduled if its commands are ready with respect to DRAM timing constraints (e.g., read and write requests to different banks).

Algorithm 1 MIAA(Γ, N_P, N_{BP})

Input: Γ: a taskset to be allocated, N_P: the number of processor cores, N_{BP}: the number of available bank partitions
Output: Schedulability of Γ

```
1: 𝔾 ← MemoryInterferenceGraph(Γ)
2: task(p_1) ← ∅
3: bank(p_1) ← LeastInterferingBank(N_BP, ℙ, 𝔾, Γ)
4: ℙ ← {p_1}
5: Φ ← {Γ}
6: while Φ ≠ ∅ do
7:    /* Allocates bundles */
8:    Φ' ← Φ; Φ_rest ← ∅
9:    for all φ_i ∈ Φ' in descending order of utilization do
10:      Φ ← Φ \ {φ_i}
11:      p_bestfit ← BestFit(φ_i, ℙ)
12:      if p_bestfit ≠ invalid then
13:         for all p_j ∈ ℙ : p_j ≠ p_bestfit ∧ ¬schedulable(p_j) do
14:            Φ ← Φ ∪ {RemoveExcess(p_j, 𝔾)}
15:      else
16:         Φ_rest ← Φ_rest ∪ {φ_i}
17:   if |Φ_rest| = 0 then
18:      continue
19:   /* Breaks unallocated bundles */
20:   all_singletons ← true
21:   for all φ_i ∈ Φ_rest do
22:      if |φ_i| > 1 then
23:         all_singletons ← false
24:         p_emptiest ← argmin_{p_i ∈ ℙ} (utilization(p_i))
25:         (φ_j, φ_k) ←ExtractMinCut(φ_i, 1 − utilization(p_emptiest), 𝔾)
26:         Φ ← Φ ∪ {φ_j, φ_k}
27:      else
28:         Φ ← Φ ∪ {φ_i}
29:   /* Opens a new processor core */
30:   if all_singletons = true then
31:      if |ℙ| = N_P then
32:         return unschedulable
33:      φ ← ⋃_{φ_i ∈ Φ} φ_i
34:      task(p_new) ← ∅
35:      bank(p_new) ←LeastInterferingBank(N_BP, ℙ, 𝔾, φ)
36:      ℙ ← ℙ ∪ {p_new}
37:      Φ ← {φ}
38: return schedulable
```

To understand the intensity of memory interference among tasks, MIAA first constructs a memory interference graph $\mathbb{G}$ (line 1 of Alg. 1). The graph $\mathbb{G}$ is a fully-connected, weighted, undirected graph, where each node represents a task and the weight of an edge between two nodes represents the amount of memory interference that the corresponding two tasks can generate. Algorithm 2 gives the pseudo-code for constructing $\mathbb{G}$. For each pair of two tasks, τ_i and τ_j, the edge weight between the two tasks is calculated as follows. First, the two tasks are assumed to be assigned to two empty cores that share the same bank partition. Then, the response times of the two tasks, R_i and R_j, are calculated by using Eq. (10), assuming that no other tasks are executing in the system. Since each task is the only task allocated to its core, the task response time is equal to the sum of the task WCET and the memory interference delay imposed on the task. Hence, we use $(R_i - C_i)/T_i + (R_j - C_j)/T_j$ as the edge weight between τ_i and τ_j ($weight(\mathbb{G}, \tau_i, \tau_j)$), which repre-

Algorithm 2 MemoryInterferenceGraph(Γ)

Input: Γ: a taskset ($\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$)
Output: $\mathbb{G}$: a graph with tasks as nodes and memory interference-intensity among nodes as edge weights
1: Construct a fully-connected undirected graph $\mathbb{G}$ with tasks in Γ as nodes
2: **for** $i \leftarrow 1$ **to** n **do**
3: **for** $j \leftarrow i+1$ **to** n **do**
4: Let two processors, p_1 and p_2, share the same bank partition
5: $task(p_1) \leftarrow \{\tau_i\}$
6: $task(p_2) \leftarrow \{\tau_j\}$
7: $R_i \leftarrow$ response time of τ_i
8: $R_j \leftarrow$ response time of τ_j
9: $weight(\mathbb{G}, \tau_i, \tau_j) \leftarrow (R_i - C_i)/T_i + (R_j - C_j)/T_j$
10: **return** $\mathbb{G}$

Algorithm 3 LeastInterferingBank($N_{BP}, \mathbb{P}, \mathbb{G}, \varphi$)

Input: N_{BP}: the number of bank partitions, $\mathbb{P}$: a set of processor cores, $\mathbb{G}$: a memory interference graph, φ: a set of tasks that have not been allocated to cores yet
Output: b: a bank partition index ($1 \leq b \leq N_{BP}$)
1: **if** $|\mathbb{P}| < N_{BP}$ **then**
2: **return** $indexof(unused\_bank\_partition())$
3: $p_{min} \leftarrow p_1$
4: $w_{p_{min}} \leftarrow \infty$
5: **for all** $p_i \in \mathbb{P}$ **do**
6: $w_{p_i} \leftarrow \sum_{\tau_j \in task(p_i)} \sum_{\tau_k \in \varphi} weight(\mathbb{G}, \tau_j, \tau_k)$
7: **if** $w_{p_{min}} > w_{p_i}$ **then**
8: $p_{min} \leftarrow p_i$
9: $w_{p_{min}} \leftarrow w_{p_i}$
10: **return** $bank(p_{min})$

sents the amount of CPU utilization penalty that may occur due to memory interference among τ_i and τ_j.

After constructing the graph $\mathbb{G}$, MIAA opens one core, p_1, by adding it to the core set $\mathbb{P}$. It is worth noting that every time a new core is opened (added to $\mathbb{P}$), a bank partition is assigned to it by the `LeastInterferingBank()` function given in Algorithm 3. The purpose of `LeastInterferingBank()` is to find a bank partition that likely leads to the least amount of memory interference to the tasks that have not been allocated yet (input parameter φ of Alg. 3). If the number of cores in $\mathbb{P}$ does not exceed the number of bank partitions (N_{BP}), `LeastInterferingBank()` returns the index of an unused bank partition (line 2 of Alg. 3). Otherwise, `LeastInterferingBank()` tries to find the least interfering bank by using $\mathbb{G}$ as follows. For each core p_i, it calculates w_{p_i} that is the sum of the weights of all edges between the tasks in p_i and the tasks in φ. Then, it returns the bank partition index of a core p_{min} with the smallest $w_{p_{min}}$.

The allocation strategy of MIAA is to group memory-intensive tasks into a single bundle and allocate as many tasks in each bundle as possible into the same core. To do so, MIAA first groups all tasks in Γ into a single bundle and assign that bundle as an element of the set of bundles to be allocated (line 5 of Alg. 1). Then, it allocates all bundles in Φ based on the best-fit decreasing (BFD) heuristic (from line 9 to line 16). Here, we define the utilization of a bundle φ_i as $\sum_{\tau_k \in \varphi_k} C_k/T_k$. Bundles are sorted in descending order of utilization and MIAA tries to allocate each bundle to a

Algorithm 4 BestFit($\varphi, \mathbb{P}$)

Input: φ: a task bundle to be allocated, $\mathbb{P}$: a set of available processor cores
Output: p_i: the processor core where φ is allocated ($p_i = invalid$, if the allocation fails)
1: **for all** $p_i \in \mathbb{P}$ in non-increasing order of utilization **do**
2: $task(p_i) \leftarrow task(p_i) \cup \varphi$
3: **if** $schedulable(p_i)$ **then**
4: **return** p_i
5: $task(p_i) \leftarrow task(p_i) \setminus \varphi$
6: **return** $invalid$

Algorithm 5 RemoveExcess($p_i, \mathbb{G}$)

Input: p_i: a processor core, $\mathbb{G}$: a memory interference graph
Output: φ: a set of tasks removed from p_i
1: $\varphi \leftarrow \emptyset$
2: **repeat**
3: $w_{\tau_{min}} \leftarrow \infty$
4: **for all** $\tau_j \in task(p_i)$ **do**
5: $w_{\tau_i} \leftarrow \sum_{\tau_k \in task(p_i) \wedge \tau_k \neq \tau_j} weight(\mathbb{G}, \tau_j, \tau_k)$
6: **if** $w_{\tau_{min}} > w_{\tau_i}$ **then**
7: $\tau_{min} \leftarrow \tau_j$
8: $w_{\tau_{min}} \leftarrow w_{\tau_i}$
9: $task(p_i) \leftarrow task(p_i) \setminus \{\tau_{min}\}$
10: $\varphi \leftarrow \varphi \cup \{\tau_{min}\}$
11: **until** $schedulable(p_i)$
12: **return** φ

core by using the `BestFit()` function given in Algorithm 4. This algorithm finds the best-fit core that can schedule a given bundle with the least amount of remaining utilization. The utilization of a core p_i is defined as $\sum_{\tau_k \in task(p_i)} C_k/T_k$. If a bundle is allocated (line 12 of Alg. 1), that bundle may introduce additional memory interference to all other cores. Therefore, we need to check if the other cores can still schedule their tasksets. If any core becomes unschedulable due to the just-allocated bundle, MIAA uses the `RemoveExcess()` function to remove enough tasks from the core in order to make it schedulable again, and puts the removed tasks as a new bundle into Φ (line 14). Conversely, if a bundle is not allocated to any core (line 15), it is put into Φ_{rest} and will be considered later.

We shall explain the `RemoveExcess()` function before moving onto the next phase of MIAA. The pseudo-code of `RemoveExcess()` is given in Algorithm 5. The goal of this function is to make the core p_i schedulable again while keeping as many memory-intensive tasks as possible. To do so, the function extracts one task at a time from the core with the following two steps. In step one, it calculates the weight w_{τ_i} for each task τ_i, which is the sum of all edge weights from τ_i to the other tasks on the same core. In step two, it removes a task τ_{min} with the smallest $w_{\tau_{min}}$ from the core. These two steps are repeated until the core becomes schedulable. Then, the function groups the removed tasks into a single bundle and returns it.

Once the bundle allocation phase is done, MIAA attempts to break unallocated bundles in Φ_{rest} (line 21 of Alg. 1). If an unallocated bundle φ_i contains more than one task, it is broken into two sub-bundles by the `ExtractMinCut()` function such that the utilization of the first sub-bundle does

Algorithm 6 ExtractMinCut($\varphi, max\_util, \mathbb{G}$)

Input: φ: a task bundle to be broken, max_util: the maximum utilization allowed for the first sub-bundle, $\mathbb{G}$: a memory interference graph
Output: (φ', φ''): a tuple of sub-bundles

```
1: Find a task τ_i ∈ φ with the highest utilization
2: φ' ← {τ_i}
3: φ'' ← φ \ {τ_i}
4: while |φ''| > 1 do
5:     w_τmax ← −1
6:     for all τ_i ∈ φ'' do
7:         w ← Σ_{τ_j ∈ φ'} weight(G, τ_i, τ_j)
8:         if w_τmax < w then
9:             τ_max ← τ_i
10:            w_τmax ← w
11:    if utilization(φ' ∪ {τ_max}) ≤ max_util then
12:        φ' ← φ' ∪ {τ_max}
13:        φ'' ← φ'' \ {τ_max}
14:    else
15:        break
16: return (φ', φ'')
```

not exceed the remaining utilization of the emptiest core (line 25). If φ_i has only one task in it, φ_i is put again into Φ. Algorithm 6 gives the pseudo-code of `ExtractMinCut()`. The primary goal of this function is to break a bundle into two sub-bundles while minimizing memory interference among them. To meet this goal, the function first finds a task with the highest utilization in the input bundle and puts that task into the first sub-bundle φ'. All the other tasks are put into the second sub-bundle φ''. Then, the function selects a task in φ'' with the maximum sum of edge weights to the tasks in φ' and moves that task to φ'. This operation repeats as long as φ'' has enough tasks and the utilization of φ' does not exceed the requested sub-bundle utilization (max_util). When `ExtractMinCut()` returns the two sub-bundles, MIAA puts them into Φ (line 26 of Alg. 1).

If all unallocated bundles are singletons, meaning that none of them can be broken into sub-bundles, and the number of cores used is less than N_P, MIAA adds a new core to $\mathbb{P}$ (line 36). Since the addition of a new core opens up the possibility of allocating all remaining bundles together to the same core, MIAA merges the remaining bundles into a single bundle (line 33) and puts it into Φ. MIAA then repeats the whole process again until Φ becomes empty.

MIAA is based on the BFD heuristic which has $O(n \cdot m)$ complexity, where n is the number of tasks and m is the number of processor cores used. On the one hand, the complexity of MIAA could be better than that of BFD due to the bundled allocation of tasks. On the other hand, the complexity of MIAA could be worse than that of BFD due to `RemoveExcess()` which can undo task allocation. However, MIAA is guaranteed to complete in bounded time. The worst case of `RemoveExcess()` happens when it removes all the previously-allocated tasks from cores. Then, MIAA opens a new core until there is any remaining core. If there is no remaining core, MIAA completes and returns a failure result.

It is worth noting that MIAA allocates at most one bank partition to each core, assuming that

one bank partition is sufficient to meet the memory requirements of any set of tasks that may be allocated to a single core. This assumption can be satisfied by configuring one bank partition to have multiple DRAM banks, as discussed in Section 2.3. However, we believe that explicitly modeling each task's memory requirement can help in providing better schedulability, which remains as our future work.

6 Evaluation

In this section, we first compare the memory interference delay observed in a real platform with the one predicted by our analysis. Then, we evaluate our memory interference-aware allocation algorithm.

6.1 Memory Interference in a Real Platform

6.1.1 Experimental Setup

The target platform is equipped with an Intel Core i7-2600 quad-core processor running at 3.4 GHz. 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 platform 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 1.

We used the latest version of Linux/RK [37, 41] for software cache and bank partitioning [18, 48].[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 [19] that records execution times and memory accesses (LLC misses) using hardware performance counters. In addition, we used the memory reservation mechanism of Linux/RK [12, 20] to protect each application against unexpected page swap-outs. To reduce measurement inaccuracies and improve predictability, we disabled the stride-based and adjacent cache-line prefetchers, simultaneous multithreading, and dynamic clock frequency scaling of the processor. All unrelated system services such as GUI and networking were also disabled.

It is worth noting that some of our assumptions do not hold in the target platform. First, the processor of the target platform is not fully timing-compositional, in that it can generate multiple outstanding memory requests and hide memory access delay by out-of-order execution. Second, the memory controller of the target platform uses write batching, and there may be other discordances between the memory controller and our system model because detailed information on the memory

[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`.

controller is not open to the public. However, we have chosen the target platform because (i) it is equipped with DDR3 SDRAM which is our main focus in this work, and (ii) it can run an OS that provides the software cache and DRAM bank partitioning features needed for our experiments. We will explore how the aforementioned differences between the target platform and our system model influences our experimental results.

6.1.2 Results with Synthetic Tasks

Our focus here is on analyzing the memory interference of the two types of synthetic tasks. At first, we use the synthetic *latency* task [53], which traverses a randomly ordered linked list. Due to the data dependency of pointer-chasing operations in linked-list traversals, the *latency* task generates only one outstanding memory request at a time, nullifying the effect of multiple outstanding memory requests in the target platform. We configure the working set size of the *latency* task to be four times of the L3 cache in order to minimize cache hits. In addition, we configure the *latency* task to generate only read requests in order to avoid the write-batching effect of the memory controller. We execute multiple instances of the *latency* task to generate interfering memory requests and to measure delay caused by them. Each instance is allocated to a different core, and assigned 4 cache partitions and 1 bank partition. We evaluate two cases where the instances share and do not share bank partitions.

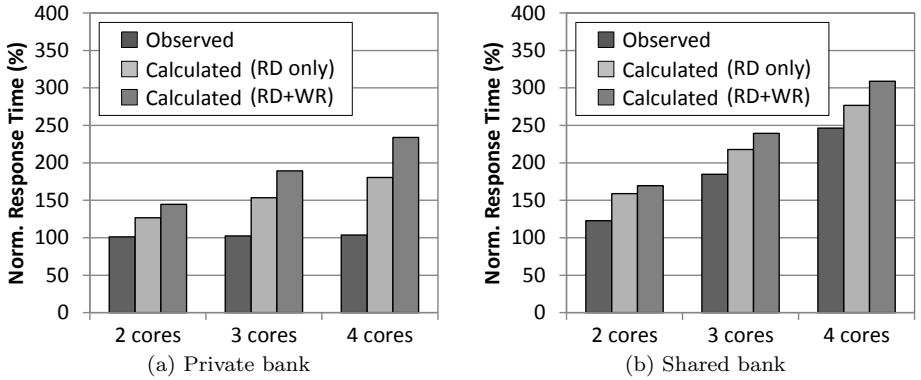
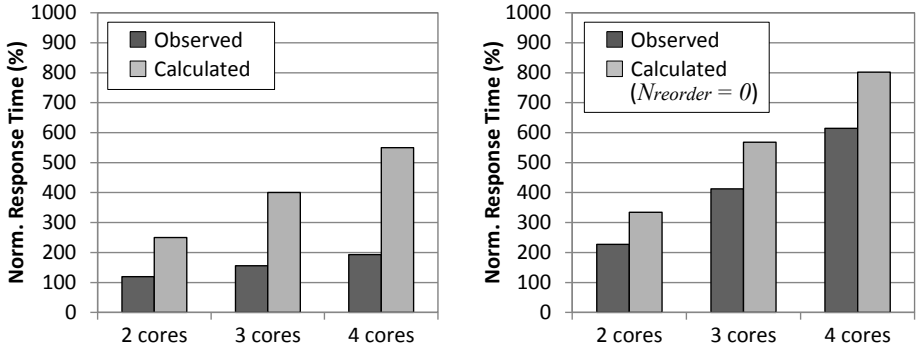


Figure 6: Response times of a synthetic task that generates one outstanding read memory request at a time

Figure 6 compares the maximum observed response times of one instance of the *latency* task with the calculated response times from our analysis, while the other instances are running in parallel. Since the *latency* task generates only read requests, we present results from a variation of our analysis, *"Calculated (RD only)"*, which considers only read requests in L_{inter}^{RW} and L_{hit}, in addition to *"Calculated (RD+WR)"*, which is our original analysis considering both read and write requests. The x-axis of each subgraph denotes the total number of cores used, e.g., "2 cores" means

that two instances run on two different cores and the other cores are left idle. The y-axis shows the response time of the instance under analysis, normalized to the case when it runs alone in the system. Since each instance is allocated alone to each core, 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 6(a) shows the response times when each instance has a private bank partition. We observed a very small increase in response times even when all four cores were used. This is because (i) each instance of the *latency* task does not experience intra-bank interference due to its private bank partition, and (ii) each instance generates a relatively small number of memory requests, so each memory request is likely serviced before the arrival of requests from other cores. However, our analysis pessimistically assumes that each memory request may always be delayed by the memory requests of all other cores. In addition, the executions of DRAM commands at different banks can be overlapped as long as DRAM timing constraints are not violated, but our analysis does not consider such an overlapping effect.

Figure 6(b) depicts the response times when all cores share the same bank partition. We set the re-ordering window size $N_{reorder}$ to zero in our analysis, because the *latency* task accesses a randomly ordered linked list and has very low row-buffer locality, thereby hardly generating row-hit requests. As can be seen in this figure, the results from both of our analyses bound the observed response times. The pessimism of our analysis in the shared bank case is not as significant as the one in the private bank case. This is due to the fact that the use of a single shared bank serializes the executions of DRAM commands from multiple cores, making their executions close to the worst-case considered by our analysis.



(a) Private bank (b) Shared bank

Figure 7: Response times of a synthetic task that generates multiple outstanding read and write memory requests at a time

Next, we use a synthetic *memory-intensive* task which has the opposite characteristics of the *latency* task. The memory-intensive task is a modified version of the *stream* benchmark [31]. The

memory-intensive task generates a combination of read and write requests with very high row-buffer locality and little computation. In addition, it can generate multiple outstanding memory requests in the target platform due to the lack of data dependency. Therefore, by using the memory-intensive task, we can identify the effects of the differences between the target platform and our analysis. Similar to the *latency* task experiments, we execute multiple instances of the memory-intensive task, with each assigned 4 cache partitions and 1 bank partition, and compare private and shared bank cases.

Figure 7 compares the response times of one instance of the memory-intensive task, while the other instances are running in parallel. Since the memory-intensive task generates both read and write requests, we do not consider our read-only analysis used in Figure 6. Figure 7(a) shows the response times with a private bank partition per core. Since the memory-intensive task generates a larger number of memory requests than the *latency* task, the observed response times of the memory-intensive task is longer than the ones of the *latency* task. Interestingly, although the memory-intensive task might generate multiple outstanding memory requests at a time, our analysis could bound memory interference delay. This is because the extra penalty caused by multiple outstanding memory requests can be compensated by various latency-hiding effects in the target platform. First, an increase in the memory access latency can be hidden by the out-of-order execution of the target processor. Second, the memory controller handles the write requests in batches, which can reduce the processor stall time. However, in order to precisely analyze memory interference in a platform like ours, both the extra penalty caused by multiple outstanding memory requests and the latency-hiding effects from out-of-order execution and write batching should be accounted for by analysis, which remains as future work.

Figure 7(b) illustrates the response times when all cores share the same bank partition. Since the memory-intensive task has very high row-buffer locality, we expected that a large re-ordering window size $N_{reorder}$ would be needed for our analysis to bound the re-ordering effect. However, as shown in this figure, our analysis could bound memory interference even when we set $N_{reorder}$ to zero. We suspect that the re-ordering effect on the memory-intensive task is canceled out by the memory latency-hiding techniques of the target platform.

6.1.3 Results with PARSEC Benchmarks

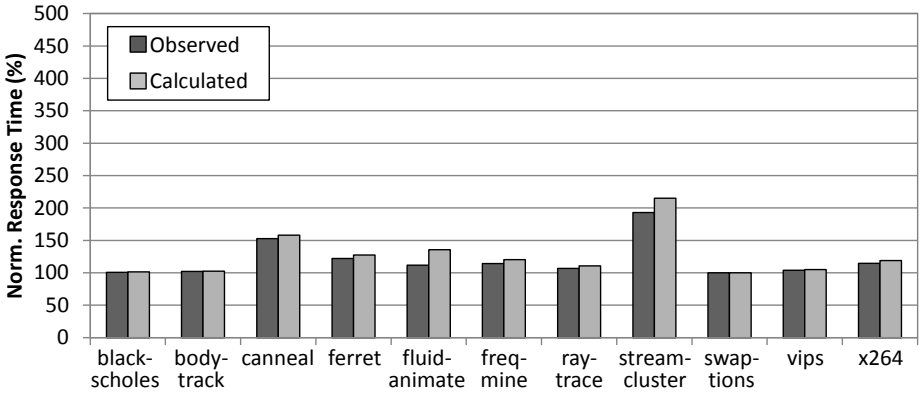
We now analyze the memory interference delay of the PARSEC benchmarks [8], which are closer to the memory access patterns of real applications compared to the synthetic tasks used in Section 6.1.2. A total of eleven PARSEC benchmarks are used in this experiment. Two PARSEC benchmarks, *dedup* and *facesim*, are excluded from the experiment due to their frequent disk accesses for data files. In order to compare the impact of different amounts of interfering memory

requests, we use the two types of synthetic tasks, *memory-intensive* and *memory-non-intensive*. Each PARSEC benchmark is assigned to Core 1 and the 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, each benchmark is assigned 20 private cache partitions.[10] 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. The memory-intensive task is the one used in Section 6.1.2. When running in isolation, the memory-intensive task generates up to 40K DRAM requests per msec (combination of read and write). 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. The memory-non-intensive task has a similar structure to the *stream* benchmark [31], but it has multiple non-memory operations between memory operations, thereby generating much fewer DRAM requests. When running alone in the system, the memory-non-intensive task generates up to 1K DRAM requests per msec.

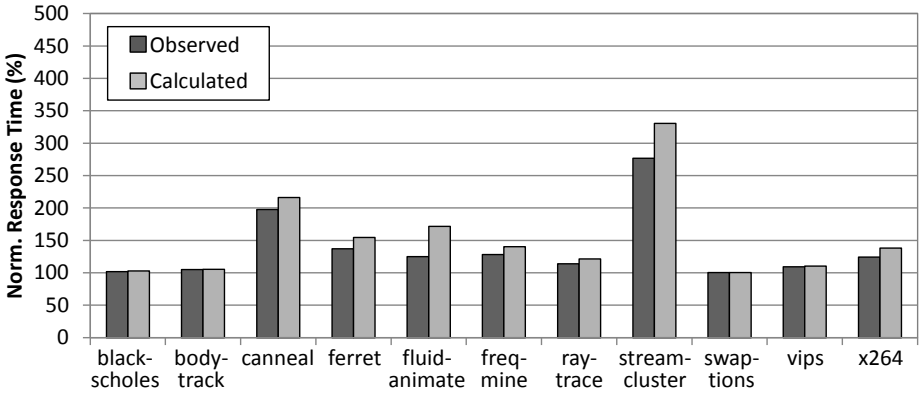
We first evaluate the response times of benchmarks with memory-intensive tasks. Figure 8 and Figure 9 compare the maximum observed response times with the calculated response times from our analysis, when memory-intensive tasks are running in parallel. The x-axis of each subgraph denotes the benchmark names, and the y-axis shows the response time of each benchmark normalized to the case when it runs alone in the system. Figure 8 shows the response times with a private bank partition per core. We observed up to 4.1x of response time increase with three memory-intensive tasks in the target system (*streamcluster* in Figure 8(c)). Our analysis could bound memory interference delay in all cases. 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 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 9 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. Our analysis requires the re-ordering window size $N_{reorder}$ to calculate the response time when a bank partition

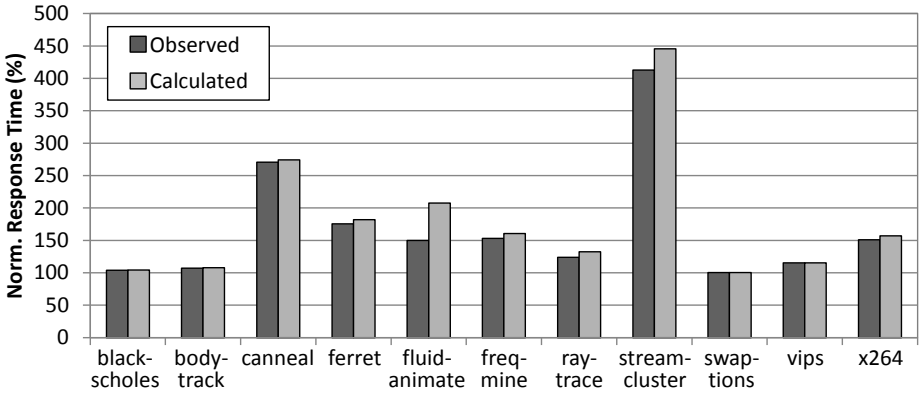
[10]Software cache partitioning simultaneously partitions the entire physical memory space into the number of cache partitions. Therefore the spatial memory requirement of a task determines the minimum number of cache partitions for that task [18].



(a) One memory-intensive task on Core 2



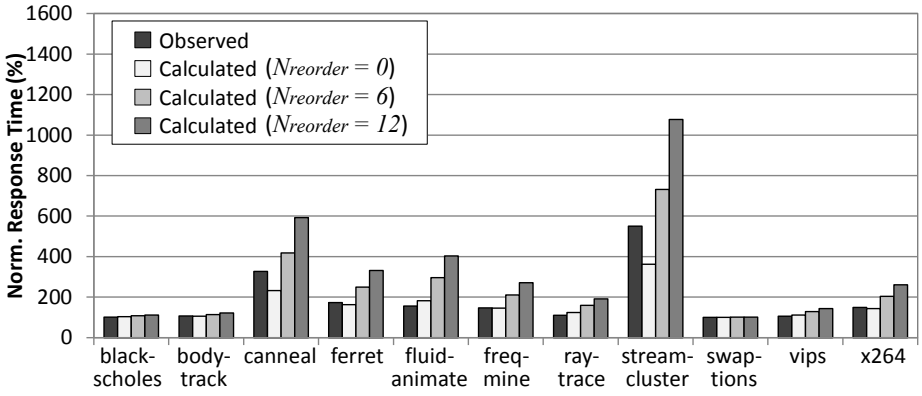
(b) Two memory-intensive tasks on Core 2 and 3



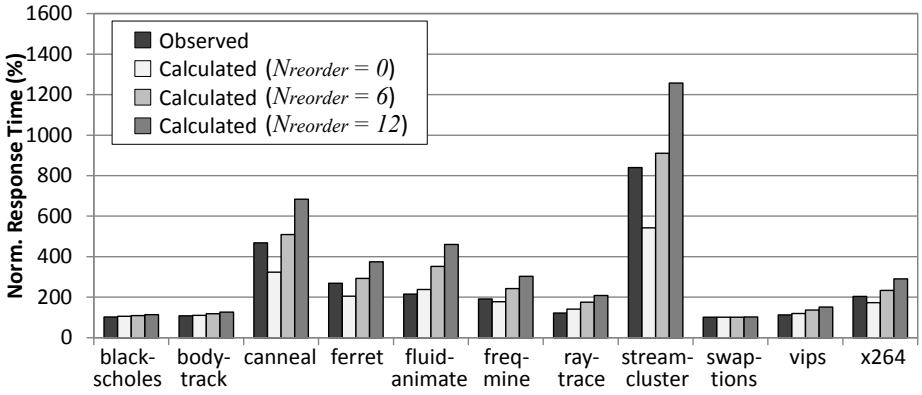
(c) Three memory-intensive tasks on Core 2, 3 and 4

Figure 8: Response times of benchmarks with a private bank partition when memory-intensive tasks run in parallel

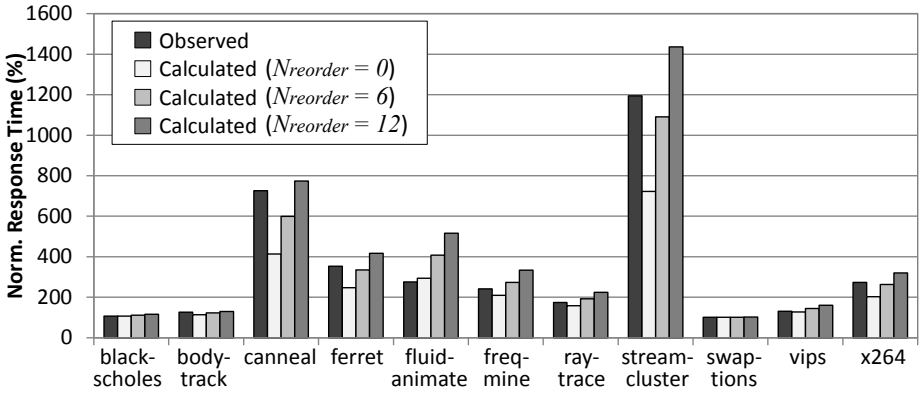
is shared. However, we cannot obtain the precise $N_{reorder}$ value because the N_{cap} value of the target platform is not publicly available. Although the N_{cap} value is crucial to reduce the pessimism in our analysis, $N_{reorder}$ can still be bounded without the knowledge of the N_{cap} value, as given in Eq. (2). The DRAM used in this platform has N_{cols} of 1024 and BL of 8, so the $N_{reorder}$ value does not exceed 128. In this figure, for purposes of comparison, we present the results from our analysis when



(a) One memory-intensive task on Core 2



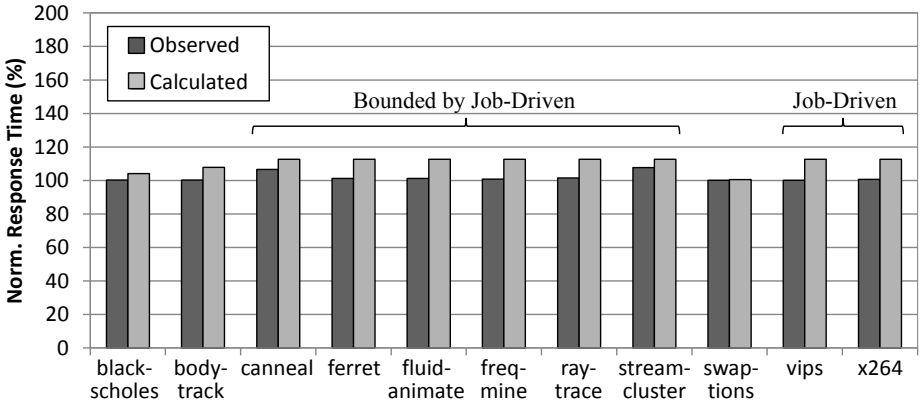
(b) Two memory-intensive tasks on Core 2 and 3



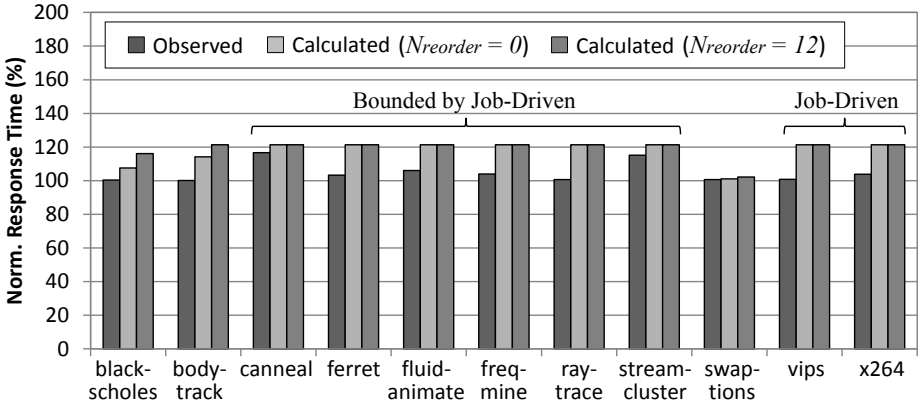
(c) Three memory-intensive tasks on Core 2, 3 and 4

Figure 9: Response times of benchmarks with a shared bank partition when memory-intensive tasks run in parallel

$N_{reorder}$ is set to 0, 6, and 12. 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 *streamcluster* with three memory-intensive tasks (Figure 9(c)), the analysis that does not account for the re-ordering effect results in only about half of the observed one. When $N_{reorder} = 12$, our analysis can find bounds in all cases. However, this does not necessarily mean that the re-ordering window size of the memory



(a) Private bank partition



(b) Shared bank partition

Figure 10: Response times of benchmarks with three memory-non-intensive tasks

controller is 12. As we have discussed in Section 6.1.2, multiple outstanding memory requests and various latency hiding techniques cancel their effects on each other in the target platform. Hence, the exact size re-ordering window size of the memory controller can be either greater or smaller than 12.

We next evaluate the response times with memory-non-intensive tasks. Figure 10(a) and Figure 10(b) depict the response times of benchmarks with a private and a shared bank partition, respectively, when three memory-non-intensive tasks run in parallel. 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 experimental results show that our analysis can closely estimate memory interference delay under scenarios with both high and low memory contention.

Table 2: Base parameters for allocation algorithm experiments

Parameters	Values
Number of processor cores (N_P)	8
Number of bank partitions (N_{BP})	8
Number of tasks to be allocated	20
Task period (T_i)	uniform from [100, 200] msec
Task utilization (U_i)	uniform from [0.1, 0.3]
Task WCET (C_i)	$U_i \cdot T_i$
Task deadline (D_i)	equal to T_i
Ratio of memory-intensive tasks to memory-non-intensive tasks	5:5
H_i for memory-intensive task	uniform from [10000, 100000]
H_i for memory-non-intensive task	uniform from [100, 1000]

6.2 Memory Interference-Aware Task Allocation

In this subsection, we evaluate the effectiveness of our memory interference-aware allocation (MIAA) algorithm. To do this, we use randomly-generated tasksets and capture the percentage of schedulable tasksets as the metric.

6.2.1 Experimental Setup

The base parameters we use for experiments are summarized in Table 2. Once a taskset is generated, the priorities of tasks are assigned by the Rate Monotonic Scheduling (RMS) policy [28]. The same DRAM parameters as in Table 1 are used, and the re-ordering window size of 12 is used ($N_{reorder} = 12$).

We consider the following six schemes for performance comparison: (i) the best-fit decreasing algorithm (BFDnB), (ii) BFD with bank partitioning (BFDwB), (iii) the first-fit decreasing algorithm (FFDnB), (iv) FFD with bank partitioning (FFDwB), (v) the IA^3 algorithm proposed in [38] (IA3nB), and (vi) IA^3 with bank partitioning (IA3wB). The BFD and FFD algorithms are traditional bin-packing heuristics, and IA^3 is a recent interference-aware task allocation algorithm based on FFD. As none of these algorithms is originally designed to consider bank partitioning, all cores share all available bank partitions under BFDnB, FFDnB and IA3nB. Conversely, under BFDwB, FFDwB and IA3wB, bank partitions are assigned to cores in round-robin order so that each core can have a dedicated bank partition. In all these algorithms, we use our response-time test given in Eq. 10 to check if a task to be allocated can fit into a candidate core.

IA^3 requires each task to have a set of WCET values to represent memory interference as part of the task's WCET. Specifically, IA^3 assumes that the worst-case memory interference is affected only by the number of cores used and is not affected by the memory access characteristics of other tasks running in parallel. Hence, under IA3nB and IA3wB, we calculate each task's WCET value as $C'_i = C_i + RD \cdot H_i$, and use C'_i instead of C_i when the FFD module of IA^3 sorts tasks in descending

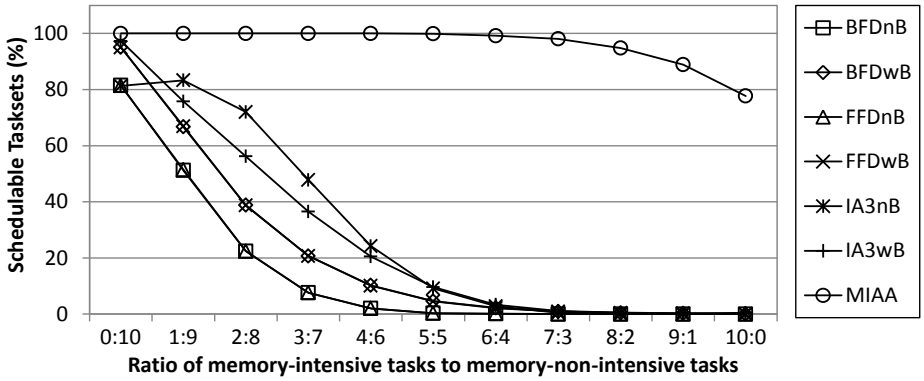


Figure 11: Taskset schedulability as the ratio of memory-intensive tasks increases

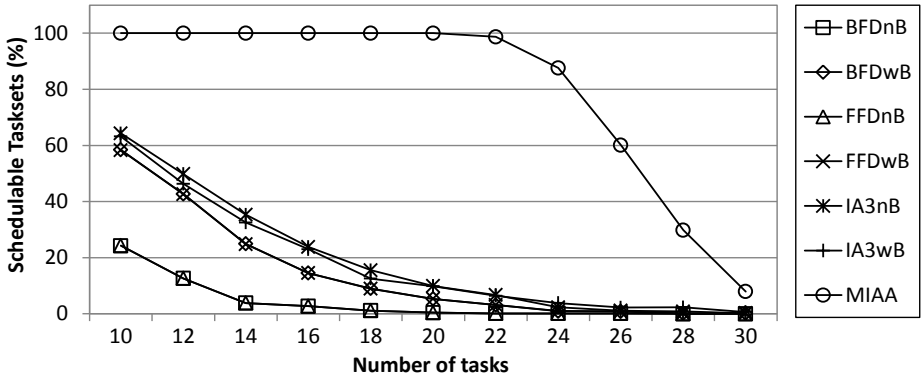


Figure 12: Taskset schedulability as the number of tasks increases

order of utilization. We have observed that the use of C_i' with a conventional response-time test [16] to check if a task to be allocated can fit into a candidate core yields worse performance than the use of C_i with our response-time test. Hence, we use only C_i' when sorting tasks in utilization. In addition, IA^3 is designed to allocate cache partitions to cores as well, but we do not consider this feature in our experiments.

6.2.2 Results

We explore four main factors that affect taskset schedulability in the presence of memory interference: (i) the ratio of memory-intensive tasks, (ii) the number of tasks, (iii) the utilization of each task, and (iv) the number of cores allowed to use. We generate 10,000 tasksets for each experimental setting, and record the percentage of tasksets where all the tasks pass the response-time test given in Eq. 10.

Figure 11 shows the percentage of schedulable tasksets as the ratio of memory-intensive tasks to memory-non-intensive tasks increases. The left-most point on the x-axis represents that all tasks are memory-non-intensive, and the right-most point represents the opposite. The percentage

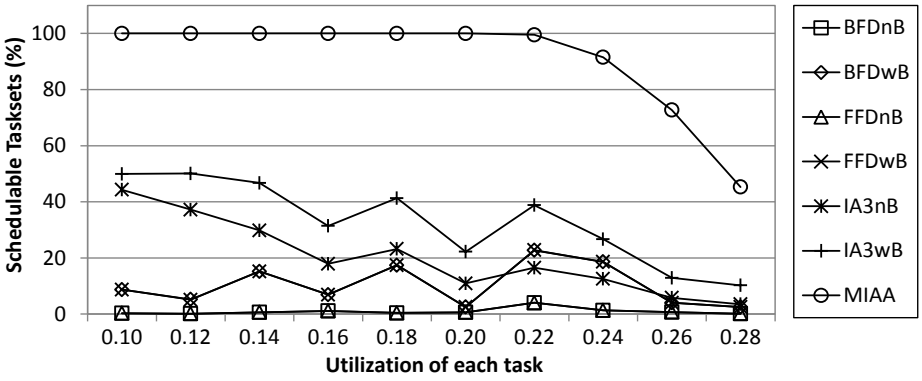


Figure 13: Taskset schedulability as the utilization of each task increases

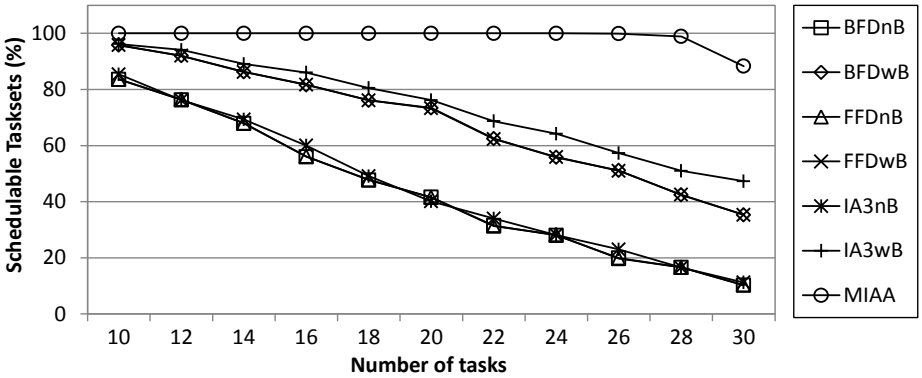


Figure 14: Taskset schedulability when tasks have medium memory intensity

difference between MIAA and the other schemes becomes larger as the ratio of memory-intensive tasks increases. For instance, when the ratio is 7:3, MIAA schedules 98% of tasksets while the other schemes schedule only less than 2% of tasksets. This big difference mainly comes from the fact that MIAA tries to co-allocate memory-intensive tasks to the same core to avoid memory interference among them. The schedulability of MIAA also decreases as the ratio of approaches to 10:0. This is a natural trend, because the amount of memory interference increases while the number of cores remains unchanged.

We now explore the trend of increasing the number of tasks and the utilization of each task. Figure 12 and Figure 13 depict the results. In Figure 13, each point k on the x-axis represents that the utilization of each task ranges $[k - 0.01, k + 0.01]$. As can be seen, MIAA performs the best, and BFDnB and FFDnB show the worst performance. Especially, all the schemes except MIAA schedule less than 70% of tasksets even if there are only 10 tasks in each taskset or the utilization of each task ranges only $[0.09, 0.11]$. This is due to the fact that, when a new task is allocated, tasks that have been allocated earlier on other cores may become unschedulable due to the memory interference from the new task. As MIAA is the only scheme accommodating such a case, it provides significantly higher schedulability than the other schemes.

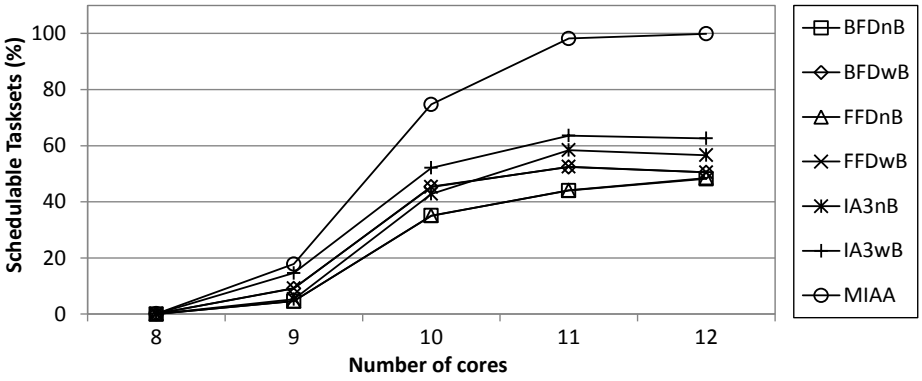


Figure 15: Taskset schedulability as the number of cores increases

In Figure 14, we consider the case where tasks have medium memory intensity. For this purpose, we randomly select the H_i value for each task in the range of $[100, 10000]$. The results in this figure show that MIAA also outperforms the other schemes when tasks do not have a bimodal distribution of memory intensity.

Lastly, we compare in Figure 15 the percentage of schedulable tasksets under different schemes when the number of cores is more than the number of bank partitions. In this experiment, the number of tasks per taskset is set to 25, and the H_i value and the utilization U_i of each task are randomly selected from $[100, 10000]$ and $[0.2, 0.4]$, respectively. The percentage under MIAA increases with the number of cores. Especially, MIAA can schedule 98% of tasksets with 11 cores. However, the other schemes cannot schedule more than 70% of tasksets, even with 12 cores. These experimental results show that a task allocation algorithm cannot scale well on multi-core platforms without explicit consideration of memory interference and MIAA yields a significant improvement in task schedulability compared to previous schemes. We expect that the performance of MIAA can be further improved by elaborating the functions used by MIAA, such as `LeastInterferingBank()` and `ExtractMinCut()`. This remains as part of future work.

7 Related Work

This paper builds upon the first work [17] that upper bounds the request re-ordering effect of COTS memory controllers. The new contributions of this paper are as follows: (i) an extended analysis for bounding memory interference delay in DRAM with multiple ranks, (ii) a memory interference-aware task allocation algorithm (MIAA) that reduces the degree of memory interference, (iii) a discussion on memory controllers with write batching, and (iv) experimental results exploring the impact of multiple outstanding requests and latency hiding techniques. Specifically, to our knowledge, MIAA is the first approach to mitigate memory interference while preserving schedulability

by co-allocating memory-intensive tasks on the same core with dedicated DRAM banks.

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 [39] 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 [42] hardware partitions banks among cores for predictability. A memory controller that allows different burst sizes for different memory requests has been proposed [27]. A memory controller that partitions the set of banks so that a single memory access can fetch data from multiple banks (bank interleaving) within a partition of banks has been proposed and it uses the open-row policy [23]. 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 [44].

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 [40, 5, 9, 45, 30]. 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 [32].

Instead of considering the memory system as a single resource, recent work [51] 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 [51] unfortunately does not model this request re-ordering effect. In this paper, we have eliminated this limitation.

In the field of task allocation, the problem of finding an optimal allocation of tasks to cores is known to be NP-complete [15]. Hence, many near-optimal algorithms based on the bin-packing heuristics have been proposed as practical solutions to the task allocation problem [10, 24, 25]. The

IA^3 algorithm [38] is the first approach to take memory interference into account when allocating tasks. IA^3 pessimistically assumes that the amount of memory interference for a task is only affected by the number of cores used, and does not consider the actual number of interfering memory requests generated by other tasks that run in parallel. Unlike IA^3, our memory interference-aware allocation algorithm (MIAA) reduces memory interference by considering the memory access intensity of each task. We have shown in Section 6 that our algorithm significantly outperforms IA^3.

Finally, there has been recent work in the computer architecture community on the design of memory controllers and memory systems that can dynamically estimate application slowdowns [47, 32, 11, 58]. These designs do not aim to provide worst-case bounds and can under-estimate memory interference. Future memory controllers might incorporate ideas like batching and thread prioritization (e.g., [35, 22, 21, 46]). This will lead to a different analysis, which could be interesting future work that builds on ours.

8 Conclusions and Future Work

In this paper, we have presented an analysis for bounding memory interference on a multi-core platform with a COTS DRAM system. 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 tighter 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, although some of our assumptions do not hold in the platform used, our analysis can closely estimate the memory interference delay under workloads with both high and low memory contention.

We have also presented a memory interference-aware task allocation algorithm that accommodates memory interference delay during the task allocation phase. Experimental results indicate that our algorithm yields significant benefits in task schedulability, with as much as 96% more tasksets being schedulable than previous schemes.

As multi-core processors are already ubiquitous, contention in shared main memory should be seriously considered and mitigated. We believe that our analysis and task allocation algorithm can be effectively used for designing predictable multi-core real-time systems. We believe plenty of future work exists in this area. We plan to (i) explore the effect of hardware prefetchers on memory interference delay, (ii) examine a non-timing-compositional architecture that allows out-of-order execution and multiple outstanding cache misses, and (iii) examine the effects of upcoming memory schedulers that serve heterogeneous agents.

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. Akesson, K. Goossens, and M. Ringhofer. Predator: a predictable SDRAM memory controller. In *IEEE/ACM international conference on Hardware/software codesign and system synthesis (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 *IEEE Real-Time Systems Symposium (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, K. K.-W. Chang, L. Subramanian, G. H. Loh, and O. Mutlu. Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems. In *International Symposium on Computer Architecture (ISCA)*, 2012.

[7] B. Bhat and F. Mueller. Making DRAM refresh predictable. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2010.

[8] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *International Conference on Parallel Architectures and Compilation Techniques (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 International Conference on Trust, Security and Privacy in Computing and Communications*, 2011.

[10] D. de Niz and R. Rajkumar. Partitioning bin-packing algorithms for distributed real-time systems. *International Journal of Embedded Systems*, 2(3):196–208, 2006.

[11] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt. Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[12] A. Eswaran and R. Rajkumar. Energy-aware memory firewalling for QoS-sensitive applications. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2005.

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

[14] M. K. Jeong, D. H. Yoon, D. Sunwoo, M. Sullivan, I. Lee, and M. Erez. Balancing DRAM locality and parallelism in shared memory CMP systems. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2012.

[15] D. S. Johnson, A. Demers, J. D. Ullman, M. R. Garey, and R. L. Graham. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM Journal on Computing*, 3(4):299–325, 1974.

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

[17] 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 *IEEE Real-Time Technology and Applications Symposium (RTAS)*, 2014.

[18] H. Kim, A. Kandhalu, and R. Rajkumar. A coordinated approach for practical OS-level cache management in multi-core real-time systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2013.

[19] H. Kim, J. Kim, and R. R. Rajkumar. A profiling framework in Linux/RK and its application. In *Open Demo Session of IEEE Real-Time Systems Symposium (RTSS@Work)*, 2012.

[20] H. Kim and R. Rajkumar. Shared-page management for improving the temporal isolation of memory reservations in resource kernels. In *IEEE Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2012.

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

[22] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2010.

[23] Y. Krishnapillai, Z. P. Wu, and R. Pellizzoni. A rank-switching, open-row DRAM controller for mixed-criticality systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2014.

[24] K. Lakshmanan, D. de Niz, R. Rajkumar, and G. Moreno. Resource allocation in distributed mixed-criticality cyber-physical systems. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2010.

[25] K. Lakshmanan, R. Rajkumar, and J. P. Lehoczky. Partitioned fixed-priority preemptive scheduling for multi-core processors. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2009.

[26] C. J. Lee, V. Narasiman, E. Ebrahimi, O. Mutlu, and Y. N. Patt. DRAM-aware last-level cache writeback: Reducing write-caused interference in memory systems. Technical Report TR-HPS-2010-002, UT Austin, 2010.

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

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

[29] 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 *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012.

[30] M. Lv, G. Nan, W. Yi, and G. Yu. Combining abstract interpretation with model checking for timing analysis of multicore software. In *IEEE Real-Time Systems Symposium (RTSS)*, 2010.

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

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

[33] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda. Reducing memory interference in multicore systems via application-aware memory channel partitioning. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011.

[34] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2007.

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

[36] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair queuing memory systems. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2006.

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

[38] M. Paolieri, E. Quiñones, F. Cazorla, R. Davis, and M. Valero. IA^3: An interference aware allocation algorithm for multicore hard real-time systems. In *IEEE Real-Time Technology and Applications Symposium (RTAS)*, 2011.

[39] 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.

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

[41] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *SPIE/ACM Conference on Multimedia Computing and Networking*, 1998.

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

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

[44] 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 *IEEE Real-Time Systems Symposium (RTSS)*, 2007.

[45] S. Schliecker, M. Negrean, and R. Ernst. Bounding the shared resource load for the performance analysis of multiprocessor systems. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2010.

[46] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu. The blacklisting memory scheduler: Achieving high performance and fairness at low cost. In *IEEE International Conference on Computer Design (ICCD)*, 2014.

[47] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu. MISE: Providing performance predictability and improving fairness in shared main memory systems. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2013.

[48] N. Suzuki, H. Kim, D. de Niz, B. Andersson, L. Wrage, M. Klein, and R. R. Rajkumar. Coordinated bank and cache coloring for temporal protection of memory accesses. In *IEEE International Conference on Embedded Software and Systems (ICESS)*, 2013.

[49] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. 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.

[50] Windriver VxWorks. `http://www.windriver.com`.

[51] Z. P. Wu, Y. Krish, and R. Pellizzoni. Worst case analysis of DRAM latency in multi-requestor systems. In *IEEE Real-Time Systems Symposium (RTSS)*, 2013.

[52] M. Xie, D. Tong, K. Huang, and X. Cheng. Improving system throughput and fairness simultaneously in CMP systems via dynamic bank partitioning. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2014.

[53] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *IEEE Real-Time Technology and Applications Symposium (RTAS)*, 2014.

[54] 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.

[55] X. Zhang, S. Dwarkadas, and K. Shen. Hardware execution throttling for multi-core resource management. In *USENIX Annual Technical Conference (USENIX ATC)*, 2009.

[56] 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.

[57] V. Seshadri, A. Bhowmick, O. Mutlu, P. B. Gibbons, M. Kozuch, T. C. Mowry, et al. The dirty-block index. In *International Symposium on Computer Architecture (ISCA)*, 2014.

[58] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu. The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015.