Analysis of Memory-Contention in Heterogeneous COTS MPSoCs

Mohamed Hassan
McMaster University, Canada
mohamed.hassan@mcmaster.ca

Rodolfo Pellizzoni
University of Waterloo, Canada
rpellizz@uwaterloo.ca

Abstract

Multiple-Processors Systems-on-Chip (MPSoCs) provide an appealing platform to execute Mixed Criticality Systems (MCS) with both time-sensitive critical tasks and performance-oriented non-critical tasks. Their heterogeneity with a variety of processing elements can address the conflicting requirements of those tasks. Nonetheless, the complex (and hence hard-to-analyze) architecture of Commercial-Off-The-Shelf (COTS) MPSoCs presents a challenge encumbering their adoption for MCS. In this paper, we propose a framework to analyze the memory contention in COTS MPSoCs and provide safe and tight bounds to the delays suffered by any critical task due to this contention. Unlike existing analyses, our solution is based on two main novel approaches. 1) It conducts a hybrid analysis that blends both request-level and task-level analyses into the same framework. 2) It leverages available knowledge about the types of memory requests of the task under analysis as well as contending tasks; specifically, we consider information that is already obtainable by applying existing static analysis tools to each task in isolation. Thanks to these novel techniques, our comparisons with the state-of-the art approaches show that the proposed analysis provides the tightest bounds across all evaluated access scenarios.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Computer systems organization → Embedded and cyber-physical systems; Computer systems organization → Architectures

Keywords and phrases DRAM, Memory, COTS, Multi-core, Real-Time, Embedded Systems, Analysis

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2020.17

1 Introduction

Unlike traditional embedded systems, Mixed Criticality Systems (MCS) such as those deployed in automotive and avionics embrace both safety-critical as well as high-performance tasks. Accordingly, low-end microcontrollers often used for traditional real-time embedded systems no longer meet the requirements of MCS. To address this challenge, researchers have explored the deployment of multi-core architectures (e.g. [8, 22, 27]). Among those architecture, Multiple-Processors Systems-on-Chip (MPSoCs) standout as a viable option to meet the various demands of MCS [12]. Their heterogeneity provides an opportunity to leverage different Processing Elements (PEs) to meet different tasks' requirements. For instance, real-time cores such as the ARM R5 in the Xilinx's Zynq Ultrascale+ [4] adopt a simpler architecture and hence are easier to analyze. Therefore, they can be used for time-sensitive safety-critical tasks. On the other hand, performance-oriented PEs such as GPUs and the ARM A-series cores can be utilized by high-performance tasks. That said, MPSoCs have their own challenges when deployed in MCS. Shared memory components such as on-chip caches and off-chip Dynamic Random Access Memories (DRAMs) create interference among PEs as they contend to access this shared memory. Memory interference can lead to a 300% increase in the total Worst-Case Execution Time (WCET) of a task in an 8-core system if a task spends only 10% of its execution time in memory accesses [28]. Similar trends were reported for the multi-core Freescale's P4080 platform [25]. In this paper, we focus on the analysis of memory contention delays in heterogeneous commercial-off-the-shelf (COTS) MPSoC platforms, where our goal is to derive a safe bound on these delays suffered by any critical task in a MCS executing on these platforms upon accessing the off-chip DRAM.

1.1 Related Work and Motivation

There exist several works whose goal is to manage interference due to contention while accessing the off-chip DRAM. Some of these works address this interference by entirely redesigning the memory controller to make DRAM accesses more predictable [7, 13, 17, 23, 34], which we refer the reader to the survey in [9] for their evaluation and comparison. Others propose operating system level solutions to alleviate the interference by partitioning DRAM banks among PEs [21, 26, 36], while controlling the maximum number of accesses issued by each PE [1, 2, 39].

Since this work focuses on analyzing DRAM interference in COTS architectures to provide safe memory delay bounds, the closest related efforts are [14, 20, 37]. The first two [20, 37] provide both job- and request-driven bounds, while the third [14] provides request-driven bounds only. *Job-driven* analysis utilizes information about total number of requests from competing cores to calculate the total Worst-Case Memory Delay (WCD) suffered by a core. *Request-driven* analysis, in contrast, derives a bound on the per-request WCD suffered by any single memory request. This bound is then multiplied by the total number of requests issued by the core to compute the total memory delay. Four observations about these efforts motivate our work. 1) Both [20] and [37] assume a specific platform with a particular architecture and OS configuration, and thus, the derived bounds are only limited to COTS platforms that follow these assumptions. 2) Although [14] addresses this limitation by exploring a wide set of COTS platforms, it only provides request-driven bounds. 3) Comparing both request- and job-driven analyses, we find that whichever one provides tighter bounds is dependent on the characteristics of running applications. In particular, it depends on the relative ratio between the number of requests of the core under analysis and the total number of interfering requests from competing cores. If the former is much smaller, then request-driven analysis will provide the tighter bound. On the other hand, if the latter is much smaller, then job-driven analysis will provide the tighter bound. Considering the minimum of both bounds as proposed in [20, 37] is certainly a viable approach. However, instead of conducting each analysis separately and then considering the smallest result, a hybrid approach that blends both analyses at a per-core basis can further tighten the bound. 4) All aforementioned works do not differentiate between different types of requests issued by cores such as reads vs writes, and DRAM row hits vs DRAM row conflict requests. Leveraging such information, as we show in this work, can significantly reduce the WCD and provide tighter bounds.

Motivated by these observations, this paper makes the following contributions.

1. It proposes an approach that blends both request- and job-driven analyses in the same framework. Both analyses are combined to form a single optimization problem. The solution to this problem provides a tighter, yet safe, bound on the cumulative memory delay suffered by requests of the core under analysis. We open-source the problem formulation that implements the analysis for the community to use and extend [1].
2. Unlike existing solutions, this framework leverages information, if available, about the requests issued by the core under analysis as well as interfering cores. Specifically, we consider the number of read and write requests, and the number of DRAM row hits and row conflicts issued by each task. This information can be obtained by analyzing all tasks in the system in isolation either statically using static analysis tools or experimentally. That said, we make no assumption about the times at which those requests are issued or their sequence patterns since this information is

[1] https://gitlab.com/FanusLab/memory-contention-analysis

run-time dependent and is affected by the behavior of competing tasks, and hence, not possible to obtain by simply analyzing the tasks in isolation.

3. Contrary to existing job-analysis [20, 37], we cover a wide set of commodity COTS platforms. Namely, we consider the same 144 platform instances covered by [14].
4. Unlike [14], which provides bounds for only 81 out of those 144 platform instances and declares the remaining 63 instances unbounded, the proposed framework is able to safely bound all 144 instances thanks to its hybrid approach using both request- and job-driven analyses.
5. We conduct a comprehensive evaluation to compare with both job-driven analyses in [20, 37] as well as request-driven analyses in [14, 20, 37] using a variety of interference scenarios. This comparison shows that the proposed approach achieves tighter bounds under all scenarios. The proposed approach provides 24%–42% tighter bounds compared to [37], 23%–$21\times$ tighter bound compared to [20], and a minimum of 4% tighter bound compared to [14], while it is able to provide bounds for scenarios that are deemed unbounded by [14] as aforementioned.

2 Background

2.1 Background on DRAM

DRAM consists of cells that are grouped in banks. Each bank resembles an array of columns and rows, and has a row buffer that holds the most recently accessed row in that bank. An on-chip Memory Controller (MC) manages accesses to the DRAM by issuing DRAM commands on the command bus. Namely, we have three main commands: ACT, CAS, and PRE. 1) If the requested row is already available in the row buffer, the request consists of only a CAS command that executes the actual read (R) or write (W) operation. We call the request in this case an *open* request. 2) If the requested bank is idle (i.e., does not have an activated row in the buffer), the MC issues an ACT command first to activate the row, followed by a CAS command. 3) If the requested row is different from the activated row in the row buffer (a *bank conflict*), the MC issues all three commands: PRE to precharge the old row, ACT to activate the requested row, and CAS to read/write. We call the request that suffers a bank conflict, a *close* request. The MC is able to issue only one command at any single cycle to the DRAM. Therefore, if there are more than one command that are ready to be sent to the DRAM at the same cycle, we say that there is a *command bus conflict*. Only one of them will be issued by the MC, while the others are delayed to subsequent cycles.

The JEDEC DRAM standard [18] defines a set of timing constraints on the three commands that must be satisfied by all MC designs; the value of each constraint depends on the specific DRAM device type and speed. Table 1 exemplifies with constraints from a single-rank DDR3 device; it also shows the value of the constraints for the particular device speed we use in the evaluation. It is important to note that the proposed analysis is not specific to this particular device and can be applied to any DRAM. For DDR4 devices, the bank-group timing constraints need to be also considered in addition to the ones in Table 1; however, a similar analysis can be applied. Each constraint represents the minimum number of clock cycles that must elapse between the transmission of a command or data and a successive command or data; with the exception of $tFAW$, which represents the minimum distance every four, rather than two, successive ACT commands. We distinguish between two types of constraints: *intra-bank constraints* are applied between data/commands issued to the same bank, while *inter-bank ACT/CAS constraints* are applied between data/commands of the same type (ACT or CAS) issued to any bank. Correspondingly, we shall say that a request causes intra-bank delay on another one if it triggers intra-bank constraints; or ACT/CAS delay if it triggers inter-bank ACT/CAS constraints. Note that there are no inter-bank constraints for PRE commands; however, due to the effect of command bus conflicts, a PRE command can still cause PRE delay on another PRE command. For ease of exposition, Figure 1 depicts an example of intra-bank constraints (Figure 1a) as well as

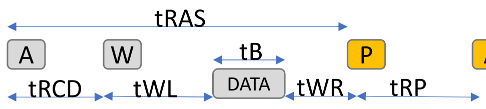
(a) Intra-bank (conflict) constraints

Parameters	Description	cycles
$tRCD$	ACT to CAS delay	9
tRL	RD to Data Start	9
tRP	PRE to ACT Delay	9
tWL	WR to Data Start	8
$tRAS$	ACT to PRE Delay	24
tRC	ACT to ACT (same bank)	33
tWR	Data End of WR to PRE	10
$tRTP$	Read to PRE Delay	5

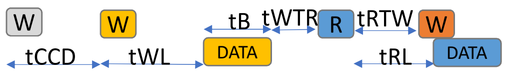
(b) Inter-bank constraints

Parameters	Description	cycles
	Inter-bank CAS constraints	
$tCCD$	CAS to CAS delay	4
$tRTW$	RD to WR Delay	6
$tWTR$	WR to RD Delay	5
	Inter-bank ACT constraints	
$tRRD$	ACT to ACT (diff bank in same rank)	4
$tFAW$	Four bank activation window	20

Table 1 JEDEC timing constraints for DDR3-1333H [18].



(a) Intra-bank constraints.



(b) Inter-bank constraints.

Figure 1 DRAM timing constraints example. tB is the data transfer time (4 cycles for a burst length of 8).

inter-bank constraints (Figure 1b). Note that when considering two consecutive requests, intra-bank constraints can affect the latency of the second request only in the case of a bank conflict: if the two requests access the same bank without conflict, then the second request must be open and only the inter-bank CAS constraints apply. Hence, we also refer to intra-bank constraints and delay as *conflict constraints/delay*. A command (or request) is denoted as *intra-* or *(inter-)ready* when it meets all its intra- (or inter-)bank constraints. A command cannot be issued before it is both intra- and inter-ready. DRAM cells have to be periodically refreshed to prevent data leakage through issuing REF (*refresh*) commands. Refresh delays can be often neglected compared to other delays [20]. It can also be added as an extra delay term to the execution time of a task using existing methods [3, 35]. Accordingly and similar to previous works [14, 20, 37], we do not account for the refresh delay.

Arbitration. Requests are first queued into per-bank queues. Then two-level arbitration is deployed as follows: 1) *Intra-bank arbitration* is implemented between requests of the same bank. This usually uses a First Ready-First Come First Serve (FR-FCFS) scheduling mechansim [20, 24, 31]. FR-FCFS prioritizes open requests, which target data already available in the row buffer over close requests. 2) *Inter-bank arbitration*: the MC deploys a Round Robin (RR) mechanism to arbitrate among intra-ready commands at the head of the bank queues [6, 14, 16, 32, 33, 36]. In case of a command bus conflict, we assume the following priority order is enforced: CAS, ACT, and then PRE such that CAS have the highest priority upon bus conflicts, while PRE commands have the least. This is known as column-first scheduling and it targets to reduce latency [24, 31].

2.2 System Model and Platform Instances

We consider a system with P PEs, where some of these PEs are *critical* (P_{cr}) and others are non-critical (P_{ncr}) such that $P = P_{cr} + P_{ncr}$. PEs share write-back write-allocate Last-Level Cache (LLC); hence, writes to DRAM occurs only because of cache eviction of dirty cache blocks. We find this to be the common policy deployed in COTS architectures and it is also adopted by previous related works [14, 37]. Requests that miss in the LLC are sent to the DRAM. We consider a single-channel single-rank DRAM subsystem with N_B banks. Similar to related work [14, 20, 37], we do not make any assumption about the computation and memory access patterns of the PE under analysis, or any of the interfering PEs. Nonetheless, as we detail in Section 3, our goal is to improve upon existing DRAM analyses, and in particular the framework in [14], by incorporating knowledge about the number of requests produced by all PEs in the system. The overall behavior of the memory subsystem depends on both the MC configuration, as well as on the characteristics of PEs that generate memory requests.

Symbol	Description	Instances	Symbol	Description	Instances
P	Number of PEs	all	N_{Bcr}	Number of banks assigned to critical PEs	$part = PartAll$
P_{cr}	Number of critical PEs	all	N_{Bncr}	Number of banks assigned to non-critical PEs	$part = PartAll$
P_{ncr}	Number of non-critical PEs	all	N_{thr}	Intra-bank reorder threshold	$thr = 1$
N_B	Number of DRAM banks	all	W_{btch}	Write batch length	$wb = 1$
N_{Bp}	Number of DRAM banks assigned to the p-th PE	all	PR	Number of outstanding requests	$pipe \neq IO$

Table 2 System model symbols.

To this end, the work in [14] defined a set of fundamental platform features that affect the delay analysis; the combination of the features, specified as a tuple $\langle wb, thr, pr, breorder, pipe, part\rangle$, characterizes one of 144 possible platform instances. Since we reuse the same features in our analysis, here we summarize their values and corresponding behavior.

Read-Write Arbitration: $wb \in \{0, 1\}$. If $wb = 0$, the MC assigns the same priority for both reads and writes. If $wb = 1$, the MC employs write batching, where it prioritizes reads while queuing writes in a dedicated write buffer. We consider the same watermarking implementation discussed in related work [14, 30, 37]: the MC services a batch of W_{btch} writes when the number of buffered writes exceeds a given threshold.

First-Ready Threshold: $thr \in \{0, 1\}$. FR-FCFS arbitration reorders intra-ready requests over non intra-ready ones targeting the same bank. If $thr = 1$, the MC deploys a thresholding mechanism [15, 20] to avoid starvation, where at most N_{thr} intra-ready requests can be re-ordered ahead of any other request targeting the same bank. If $thr = 0$, then no reordering threshold is implemented.

PE Prioritization: $pr \in \{0, 1\}$. If $pr = 1$, the MC prioritizes requests of critical PEs over non-critical ones [15, 30]. If $pr = 0$, all PEs are treated equally.

Inter-bank Reordering: $breorder \in \{0, 1\}$. As discussed, the MC employs a RR arbiter which selects among banks with intra-ready commands. If the command of the highest priority bank is not inter-ready, then the MC can reorder ahead of it the command of the next highest priority bank (based on the RR order) with a ready command. If $breorder = 1$, then the reordered commands can be of the same type; in particular, a W command can be reordered ahead of a R command of vice-versa. As shown in [14], this can lead to a situation where an unbounded number of CAS commands is reordered ahead of another CAS command. To avoid starvation, we also consider $breorder = 0$, where inter-bank reordering is allowed only for commands of different type.

PE pipeline architecture: $pipe \in \{IO, IOCr, OOO\}$. If $pipe = IO$, all PEs are in order and can generate only one pending memory request at a time. If $pipe = OOO$, all PEs are out-of-order, and we let PR to denote the maximum number of outstanding requests in the MC queue for each PE. If $pipe = IOCr$, then critical PEs are in order, while non-critical ones are out-of-order [4].

Bank Partitioning: $part \in \{PartAll, PartCr, NoPart\}$. Several previous works (e.g. [7, 10, 17, 35]) have proposed DRAM bank partitioning, where banks are partitioned among PEs, to reduce bank conflicts between PEs. Partitioning is typically implemented by manipulating the page table in the OS [21, 26, 36]. If $part = PartAll$, then partitioning is applied to all PEs. If $part = PartCr$, then partitioning is applied only to critical PEs, while non-critical PEs can use all banks. If $part = NoPart$, no partitioning is used.

Table 2 further summarizes the parameters associated with each platform instance. In Table 2, N_{B_p} depends on the applies bank partitioning scheme. For instance, if we have $NB = 8$ and $P_{cr} = P_{ncr} = 2$, under $NoPart$: $N_{B_p} = NB = 8$ for all PEs, for $PartAll$: $N_{B_p} = 8/4 = 2$, while for $PartCr$: $N_{B_p} = 8/2 = 4$ for critical PEs and $N_{B_p} = 8$ for non-critical PEs. $N_{B_{cr}}$ and $N_{B_{ncr}}$ apply only under $PartAll$ since it is the only partitioning scheme, where critical and non-critical PEs do not share banks; hence, each bank can be indicated as either critical or non-critical.

3 Preliminaries

We are interested in computing a bound on the cumulative delay $\Delta(t)$ suffered by requests generated by one or more tasks running on a critical PE under analysis PE_i in an interval of time t. Specifically, we bound the *processing* delay of requests of PE_i, that is, the extra delay suffered after the request arrives at the head of the request queue for PE_i. For an out-of-order architecture, we do not consider *queueing* delay due to a request being queued after other requests of PE_i itself; such delay depends on the exact time at which requests are issued and should be handled while statically analyzing PE_i. Let e be the WCET of the task(s) in isolation, that is, while the other PEs in the system are inactive and do not cause any delay. Further assume that delay is composable, that is, $e + \Delta(t)$ is an upper bound to the execution time of the task(s) when suffering a cumulative delay $\Delta(t)$ (note that even if the PE is not timing compositional, the delay can still be composed by computing an appropriate upper-bound to e as described in [11]). Then the execution time $\bar{e}$ of the task(s) can be bounded by the recurrence: $\bar{e} = e + \Delta(\bar{e})$.

We assume that either through static analysis or measurements, it is possible to formulate bounds on the number of requests that the task(s) produces in isolation (the *original schedule* of memory requests). The number and type of such constraints depends on the capability of the analysis or measurement framework. A coarse method might be only capable of deriving the maximum number of requests $H(i)$, while an improved method might be able to bound the maximum number $HR(i)$ and $HW(i)$ of read and write requests, respectively. There also exist analyses [5] that are able to differentiate between open and close requests, hence deriving bounds $HR^o(i), HR^c(i)$ on the number of open and close read requests, and similarly $HW^o(i), HW^c(i)$ for write requests. Note that in this case it might hold $HR^o(i) + HR^c(i) > HR(i)$, as the analysis might not be able to classify as open or close some of the requests. Hence, to provide a general analysis, we will consider all presented terms, with the assumption that coarse estimation methods might result in a value of $+\infty$ for some of the terms (i.e., they cannot provide a useful bound).

We are now interested in determining the number of requests of each type produced by the task(s) when running together with the other $P - 1$ interfering PEs (the *interfered schedule*). For simplicity, we will assume that the behavior of PE_i, in terms of produced memory requests, is not affected by interference; note that if the PE uses a cache, this implies that the cache must be private or partitioned. Hence, the bounds on the number of reads and write requests still hold. However, the type of each request (open or close) depends on the state of the device, which can be affected by other PEs. Therefore, with no loss of generality, let $R^o(i), R^c(i), W^o(i), W^c(i)$ to denote the number of open/close read and write requests for PE_i in the interfered schedule. We then have:

$$\text{if } wb = 0 : R^o(i) \le HR^o(i), W^o(i) \le HW^o(i) \tag{1}$$
$$\text{if } \textit{PartAll} \text{ and } wb = 0 \ : R^c(i) \le HR^c(i) \tag{2}$$
$$\text{if } \textit{PartAll} \text{ and } wb = 0 \ : W^c(i) \le HW^c(i) \tag{3}$$
$$\text{if } \textit{PartAll} \text{ and } wb = 0 \ : R^c(i) + W^c(i) \le HR^c(i) + HW^c(i) \tag{4}$$
$$R^c(i) + R^o(i) \le HR(i) \tag{5}$$
$$W^c(i) + W^o(i) \le HW(i) \tag{6}$$
$$R^c(i) + R^o(i) + W^c(i) + W^o(i) \le H(i) \tag{7}$$

Equations 5-7 bound the number of reads, writes and all requests, respectively; based on our assumptions, they are always valid. Equations 1-4 bound the number of open and close requests, and instead depend on the platform features. If the platform employs write batching, then we make no assumption on the number of open or close requests: write requests produced by other PEs can change the time and order in which batches are issued, which in turn can change the type of any request. If $part = PartCr$ or $NoPart$, then PE_i shares banks with some other PE. In this case, bank conflicts can turn requests that were open in isolation into close requests. Hence, in this case we cannot consider the bounds on close requests (Equations 2-4), while the bound on open requests (Equation 1) still holds. Finally, we discuss how to bound the number of requests for an interfering PE_p with $p \neq i$. If PE_p is a core executing a known task set, then the same approach in Equation 1-7 can be employed, where $H(p)$ and related terms express the maximum number of requests produced by the task set in any interval of length t. In particular, related work [20] shows how to compute $H(p)$ assuming a partitioned, fixed priority scheduling scheme. Other work assumes memory regulation [38], where PE_p is assigned a memory budget Q_p, and cannot issue more than Q_p requests in a regulation interval of length P. In this case, assuming that the window of time t starts synchronously with the regulation interval, we simply compute the value in Equation 8. Note that for an out-of-order PE, term PR is added to account for requests that might be queued at the memory controller before the beginning of the first regulation period.

$$H(p) = \lceil t/P \rceil \cdot Q_p + \begin{cases} 0 \text{ if } IO \text{ or } (p \text{ is } cr \text{ and } IOCr) \\ PR \text{ otherwise} \end{cases} \tag{8}$$

4 Memory Delay Analysis

In this section, we show how to compute a cumulative WCD bound Δ for the requests of critical core under analysis PE_i. In details, we consider the delay due to additional timing constraints, as well as bus conflicts, caused by either interfering requests of other PEs, or previous requests of PE_i itself. For $wb = 0$, the WCD bound includes the delay suffered by both reads and writes requests of PE_i, which we call the *critical requests*. For $wb = 1$, we only consider delay suffered by read requests, as under write batching write requests of PE_i itself are queued so that they do not delay program execution; however, in this case we consider the delay caused by writes of PE_i on the critical read requests of PE_i. To facilitate accounting for the various timing constraints, we will obtain Δ by determining which delay is caused by each request (either conflict, PRE, ACT or CAS), and then adding together three corresponding *delay terms*: L^{Conf} represents the cumulative delay due to conflict constraints; while L^{ACT} and L^{CAS} represent the cumulative ACT and CAS delays. Note that we do not define a delay term for PRE because, as we prove in Section 4.4, in the worst-case interference pattern such delay is zero. We first categorize the effect of the interfering requests of other PE_i in Section 4.1, and then discuss the effect of self-interference caused by previous requests of PE_i in Section 4.2. Finally, Sections 4.3 and Sections 4.4 detail how to compute the delay terms.

4.1 Interfering Requests

We start with a set of observations, based on the timing constraints in Section 2.1, to help classifying interfering requests based on which type of delay they cause.

▷ Observation 1. Consider two requests targeting different bank. If both requests are close, then the first one can cause PRE, ACT and CAS delay to the second one; otherwise, it can only cause CAS delay.

Note that Observation 1 holds because in order to suffer PRE or ACT delay, both the delaying and the delayed request must issue a PRE/ACT command.

▷ Observation 2. Consider two requests targeting the same bank. If the second request is close, then the first one can cause conflict delay to it; otherwise, it can only cause CAS delay. The conflict delay is larger than the CAS delay.

▷ Observation 3. Conflict constraints are larger than PRE, ACT and CAS constraints. Hence, when two consecutive requests can target either the same or different banks, the delay suffered by the second request is larger or equal if they target the same bank compared to different banks (specifically, equal if the request is open, and larger if close).

We next discuss how to determine the number of interfering requests for each delay term. Based on Observation 3, we can maximize Δ by assuming that all interfering requests that can target the same bank as a request of PE_i do so. Therefore, when counting interfering requests, we classify them between *intra-bank requests*, which can delay each other and critical requests of PE_i on the same bank based on Observation 2, and *inter-bank requests*, which can delay intra-bank requests based on Observation 1; specifically, we next discuss how to systematically divide the interfering requests into several *interference components*.
(1) **Intra-bank conflict requests**: $R^{Conf,c}$, $W^{Conf,c}$ are the number of read and write interfering requests targeting the same bank as any one request of PE_i, and which are serviced ahead of that request because they arrived before it. As noted in Section 3, in this case we can make no assumption on the type of the requests. Hence, we assume the worst case where all such requests, as well as the request of PE_i, are close [2].
(2) **Intra-bank reorder requests**: $R^{Reorder,o}$, $W^{Reorder,o}$ are the numbers of interfering requests of other PEs targeting the same bank as any one request of PE_i, and which arrived after that request but are reordered ahead of it due to first-ready arbitration. Since the interfering requests are ready, they must be open requests, while the request of PE_i must be close.
(3) **Inter-bank-close requests**: $R_c^{InterB,c}$, $R_c^{InterB,o}$, $W_c^{InterB,c}$, $W_c^{InterB,o}$ are interfering requests (read/write and open/close, based on the superscript) that target a different bank than any one request of PE_i, and delay close requests targeting the same bank as PE_i: the $R^c(i) + W^c(i)$ close requests of PE_i itself, and the $R^{Conf,c}$/$W^{Conf,c}$ conflict requests. By Observation 1, the open requests $R_c^{InterB,o}$ and $W_c^{InterB,o}$ contribute CAS delay, while the close requests $R_c^{InterB,c}$ and $W_c^{InterB,c}$ contribute to PRE, ACT and CAS delay.
(4) **Inter-bank-open requests**: R_o^{InterB}, W_o^{InterB} are interfering requests (R and W) that target a different bank than any one request of PE_i, and delay open requests targeting the same bank as PE_i: the $R^o(i) + W^o(i)$ open requests of PE_i itself, and the $R^{Reorder,o}$, $W^{Reorder,o}$ reorder requests. By Observation 1, these $R_o^{InterB} + W_o^{InterB}$ requests can only contribute CAS delay.

Note that for instances with $wb = 1$, the intra- and inter-bank components only include read requests, while write requests are considered in the write batching component. Hence we impose:

$$\text{if } wb : W^{Conf,c} = W^{Reorder,o} = W_c^{InterB,c} = W_c^{InterB,o} = W_o^{InterB} = 0 \tag{9}$$

(5) **Write batching requests**: For instances with $wb = 1$, W^{WB} represents the total number of write requests; contrarily to the previous components, W^{WB} includes both interfering write requests, as well as write requests of PE_i itself, since write requests of all PEs are reordered and issued in write batches. As again noted in Section 3, when $wb = 1$ we can make no assumption on the type (open or close) of write requests executed in write batches, hence we consider a worst case situation where all requests are close and target the same bank, thus contributing to L^{Conf}.

The described interference components depend on the total number of requests produced by each interfering PE, as well as on the platform instance. We detail how to bound the interference components in Section 5; while in the rest of this section we focus on computing the latency terms assuming that the values of the interference components are known. Finally, Section 5.6 shows that

[2] Note that if PE_i shares a bank with another interfering PE, then Equations 2, 3 do not apply; hence the optimization problem can set $R^o(i) = W^o(i) = 0$ and maximize the number of close request $R^c(i), W^c(i)$ based on Equations 5, 6.

Symbol	Interfering Direction	Interfering Request Type	Interfered Request Type (or PE ID)	Description	Delay
$R^o(i)$ $(W^o(i))$	$R\,(W)$	Open	PE_i	Total number of open reads from PE_i	
$R^c(i)$ $(W^c(i))$	$R\,(W)$	Close	PE_i	Total number of close reads from PE_i	
Self Interference Component					
$R^{OtC}(i)$ $(W^{OtC}(i))$	$R\,(W)$	-	PE_i	Requests that were open and became close due to interference.	L^{Self}
$R^{CAS}(i)$ $(W^{CAS}(i))$	$R\,(W)$	-	PE_i	Requests that cause self CAS delay	
$R^{Conf}(i)$ $(W^{Conf}(i))$	$R\,(W)$	-	PE_i	Requests that cause extra self conflict delay.	
$N^{None}(i)$	R or W	-	PE_i	Requests that cause no extra self conflict delay.	
$N^{ACT}(i)$	R or W	-	PE_i	Close requests targeting different banks.	
$N^{ACT,a}(i)$	R or W	-	PE_i	Requests from $N^{ACT}(i)$ that were originally close.	
$N^{ACT,b}(i)$	R or W	-	PE_i	Requests from $N^{ACT}(i)$ that were originally open.	
Intra-Bank Conflict Requests					
$R^{Conf,c}[(p)]$ $(W^{Conf,c}[(p)])$	$R\,(W)$	Close	Close	Requests causing conflict interference.	L^{Conf}
x^{Conf}	R or W	Close	Close	Total number of triggered conflict delays.	
Intra-Bank Reorder Requests					
$R^{Reorder,o}[(p)]$ $(W^{Reorder,o}[(p)])$	$R\,(W)$	Open	Close	Number of requests causing intra-bank reorder interference.	L^{CAS}
x^{CAS}	R or W	Open or Close	Open or Close	Total number of triggered CAS delays.	
Inter-Bank-Close Requests					
$N_{reqs,c}$	R or W	–	Close	Number of interfered requests for inter-bank-close component.	L^{ACT}, L^{CAS}
$R_c^{InterB,c}[(p)]$ $(W_c^{InterB,c}[(p)])$	$R\,(W)$	Close	Close	Number of requests that cause inter-bank interference on PE_i's close requests or any of the $N_{reqs,c}$ requests. They interfere either on ACT (N_{ACT}^{InterB}), CAS read ($R_{CAS,c}^{InterB}$), or CAS write ($W_{CAS,c}^{InterB}$) commands.	
$R_c^{InterB,o}[(p)]$ $(W_c^{InterB,o}[(p)])$	$R\,(W)$	Open	Close		
$N_{ACT,c}^{InterB,c}$	R or W	Close	Close		
$R_{CAS,c}^{InterB}$ $(W_{CAS,c}^{InterB})$	$R\,(W)$	Close or Open	Close		
Inter-Bank-Open Requests					
$N_{reqs,o}$	R or W	–	Open	Number of interfered requests for inter-bank-open component.	L^{CAS}
$R_o^{InterB}[(p)]$ $(W_o^{InterB}[(p)]\,)$	$R\,(W)$	Close or Open	Open	Number of interfering requests that cause inter-bank interference on PE_i's open requests or any of the $N_{reqs,o}$ requests .	
Auxiliary CAS Delay Variables					
R_{CAS}^{InterB} (W_{CAS}^{InterB})	$R\,(W)$	Open or Close	Open or Close	Requests from other PEs targeting other banks and causing inter-bank CAS delays.	L^{CAS}
$x^{CAS,RW}$ $(x^{CAS,WR})$	$R\,(W)$	Open or Close	Open or Close	Total number of requests causing a R-to-W (W-to-R) CAS delays.	
Write Batching Requests					
$W^{btch}[(p)]$	W	Close or Open	–	Number of interfering write requests that arrive while no critical request is active.	L^{WB}
$W^{before}[(p)]$	W	Close or Open	–	Number of interfering write requests that arrive while a critical request is active, and are executed before (after) the critical request	
$W^{after}[(p)]$	W	Close or Open	–		

Table 3 Optimization problem variables. We use L to denote a delay term; N to denote number of requests; R/W to denote number of read/write requests; and x to denote number of constraints. If a variable has the (p) index, then it refers to a specific PE_p. Otherwise, the variable indicates values over all PEs. For superscripts, c/o denotes the type (open or close) of the requests; for inter-bank requests, the subscript c/o denotes the type of the following intra-bank request.

we can compute a bound on Δ by solving a Linear Programming (LP) problem. To facilitate the reader, Table 3 summarizes all variables used in the optimization problem [3].

It remains to summarize the impact of the intra- and inter-bank interfering requests on the delay terms. For intra-bank interfering requests, based on Observation 2 let x^{Conf} to denote the number of conflict delays triggered by the interfering requests, and x^{CAS} to denote the number of triggered CAS delays. We can then bound the total number of delays $x^{Conf} + x^{CAS}$ based on the total number of intra-bank interfering requests, and we can bound x^{Conf} based on the number of close requests targeting a same bank as PE_i: the conflict requests, and the close critical requests of PE_i:

$$x^{Conf} + x^{CAS} \leq R^{Conf,c} + W^{Conf,c} + R^{Reorder,o} + W^{Reorder,o} \tag{10}$$

$$x^{Conf} \leq R^{Conf,c} + W^{Conf,c} + R^c(i) + (1 - wb) \cdot W^c(i). \tag{11}$$

Note that we multiply the write requests of PE_i by $(1 - wb)$ since they are not critical if $wb = 1$. Instead, in the $wb = 1$ case, the conflict delays caused by the W^{WB} requests in write batches will be directly accounted for in the L^{Conf} term in Section 4.3.

For inter-bank interfering requests, $R_c^{InterB,o}$, $W_c^{InterB,o}$, R_o^{InterB} and W_o^{InterB} only induce CAS delays, as previously explained. To bound the cumulative delay induced by the $R_c^{InterB,c}$ and $W_c^{InterB,c}$ requests, we employ the following pipeline theorem from [37]:

▶ **Theorem 1** (Theorem 1 in [37]). *The delay caused by an interfering request to a request under analysis, where the two requests target different banks, is upper bounded by the delay caused by one interfering command on the same command of the request under analysis, i.e., either the PRE delay, or the ACT delay, or the CAS delay.*

[3] Note that $H(i), HR(i), HW(i), HR^c(i), HR^o(i), HW^c(i), HR^o(i)$ for all cores, as introduced in Section 3, do not appear in the table because they are inputs to the analysis, hence constants in the LP problem.

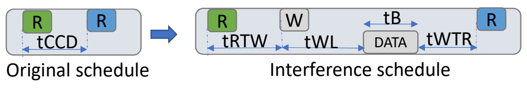


Figure 2 Self interference example.

Note that in Section 4.4 we will prove that the PRE delay is always less than the ACT delay. Hence, to maximize the bound on Δ, it suffices to assume that based on Theorem 1, each request in $R_c^{InterB,c}$ and $W_c^{InterB,c}$ can cause either ACT or CAS delay. We thus introduce terms $N_{ACT,c}^{InterB,c}, R_{CAS,c}^{InterB,c}, W_{CAS,c}^{InterB,c}$ to denote the number of requests (possibly distinguishing between R and W direction) that cause ACT and CAS delay, respectively, obtaining the following constraints:

$$N_{ACT,c}^{InterB,c} + R_{CAS,c}^{InterB,c} + W_{CAS,c}^{InterB,c} \leq R_c^{InterB,c} + W_c^{InterB,c}, \tag{12}$$

$$N_{ACT,c}^{InterB,c} + R_{CAS,c}^{InterB,c} \leq R_c^{InterB,c}, \tag{13}$$

$$N_{ACT,c}^{InterB,c} + W_{CAS,c}^{InterB,c} \leq W_c^{InterB,c}. \tag{14}$$

Finally, we use R_{CAS}^{InterB} (W_{CAS}^{InterB}) to denote the total number of reads (writes) from other PEs targeting other banks and causing inter-bank CAS delays. Hence, we get:

$$R_{CAS}^{InterB} = R_{CAS,c}^{InterB,c} + R_o^{InterB} + R_c^{InterB,o}, \tag{15}$$

$$W_{CAS}^{InterB} = W_{CAS,c}^{InterB,c} + W_o^{InterB} + W_c^{InterB,o}. \tag{16}$$

4.2 Self-Interference

Section 4.1 summarized the delay caused by interfering requests in terms of the timing constraints and bus conflicts induced by such requests. However, when two critical requests of PE_i are executed back-to-back in the original schedule, the first request of PE_i can induce further delays on any request that interferes with the second request of PE_i itself; ignoring such *self-interference* effects leads to an unsafe bound. Consider the example in Figure 2. Originally, PE_i issued two consecutive open R requests to the same bank; the minimum distance between the two requests, based on their CAS commands, is equal to the CAS-to-CAS constraint $tCCD$. In the interfered schedule, one W request of another core is interleaved between the two requests of PE_i; as a consequence, the distance between the two requests becomes equal to $tRTW + tWL + tB + tWTR$. Hence, the added delay is $tRTW + tWL + tB + tWTR - tCCD$, which is larger than the maximum delay $tWL + tB + tWTR$ of a single CAS.

To produce a safe delay bound, we thus proceed as follows: we carefully analyze each scenario (Cases (1a)-(3) below) involving two consecutive critical requests of PE_i, and whenever we found that the effect of self-interference is non-zero, we handle it by adding an additional delay term to the analysis (in the case of the example in Figure 2, to L^{CAS}), and subtracting the minimum distance between the requests in the original schedule ($tCCD$ in the example). When analyzing the scenarios, it is important to keep in mind, as discussed in Section 3, that requests of PE_i that were open in the original schedule can become close in the interfered schedule if write batching is enabled or PE_i shares banks with some other PE. Let $R^{OtC}(i)$ and $W^{OtC}(i)$ be upper bounds on the number of such R and W open-to-close requests; since $HR^o(i), HW^o(i)$ represent open requests in the original schedule, and $R^o(i), W^o(i)$ in the interfered schedule, it must hold:

$$R^{OtC}(i) \leq HR^o(i) - R^o(i), \tag{17}$$

$$W^{OtC}(i) \leq HW^o(i) - W^o(i), \tag{18}$$

$$\text{if } PartAll \text{ and } wb = 0 : R^{OtC}(i) = W^{OtC}(i) = 0. \tag{19}$$

- Case (1a) and (1b): the two requests of PE_i target the same bank, and the second one is close in the interfered schedule. In this case, the second request could be delayed by other close conflict requests in the interfered schedule, which could be in turn delayed by a conflict delay due to the first request of PE_i. However, if the second request of PE_i was also close in the original schedule (Case (1a)), then the minimum distance between the two requests in the original schedule is equal to the same conflict delay; hence, in this case self-interference does not add any extra delay. If instead the second request was open in the original schedule (Case (1b), meaning it is an open-to-close request), then the minimum distance in the original schedule could be $tCCD$; hence, to produce a safe bound, in this case we add one conflict delay to L^{conf}, and subtract $tCCD$ from the WCD Δ. We let $R^{Conf}(i), W^{Conf}(i)$ to denote the number of R and W requests for Case (1b), and $N^{None}(i)$ to denote requests that do not add any extra delay as per Case (1a).
- Case (2a) and (2b): assume that Case (1a), (1b) do not apply (that is, the requests target different banks and/or the second request is open in the interfered schedule). Then, it can still be possible for the first request of PE_i to cause either PRE, CAS or ACT delay on one or more interfering requests, which in turn cause the same type of delay to the second request of PE_i. Case (3), which we represented in Figure 2, covers the CAS delay; Case (2a) and (2b) cover the PRE and ACT delay. Since only close requests can cause or suffer PRE/ACT delay, it follows that for Case (2a), (2b) the two requests of PE_i must be close in the interfered schedule. Therefore, by assumption they must target different banks. The minimum distance between them is either the ACT-to-ACT constraint $tRRD$ if both were close in the original schedule (Case (2a)), or $tCCD$ if at least one was open (Case (2b), the request is open-to-close). As Section 4.1 mentions and Section 4.4 proves, the ACT delay is larger than the PRE delay; hence, for these cases we add an ACT term to L^{ACT} and subtract either $tRRD$ (2a) or $tCCD$ (2b). We use $N^{ACT,a}(i)$ and $N^{ACT,b}(i)$ for the number of requests added to L^{ACT} in Case (2a) and (2b), respectively, and $N^{ACT}(i)$ for their sum.

We can then bound the self-interference terms for Cases (1a), (1b), (2a), (2b) based on the number and type of requests of PE_i as follows:

$$R^{Conf}(i) + W^{Conf}(i) \leq R^{OtC}(i) + (1 - wb) \cdot W^{OtC}(i) \tag{20}$$
$$if NB_i = 1 : N^{None}(i) = R^c(i) - R^{OtC}(i) + (1 - wb) \cdot (W^c(i) - W^{OtC}(i)) \tag{21}$$
$$N^{ACT}(i) = N^{ACT,a}(i) + N^{ACT,b}(i) \tag{22}$$
$$N^{ACT,b}(i) \leq R^{OtC}(i) + (1 - wb) \cdot W^{OtC}(i) \tag{23}$$
$$N^{ACT,a}(i) + N^{ACT,b}(i) \leq R^c(i) + (1 - wb) \cdot W^c(i) \tag{24}$$
$$if NB_i = 1 : N^{ACT}(i) = 0 \tag{25}$$

Note that again we multiply write requests of PE_i by $(1 - wb)$ since if $wb = 1$ such requests are not critical, and thus do not contribute to self-interference. If $NB_i = 1$, all requests of PE_i target the same bank; hence, Case (2a) and (2b) cannot hold (Equation 25), and instead all critical requests that were close in the original schedule ($R^c(i) - R^{OtC}(i)$ for reads and $W^c(i) - W^{OtC}(i)$ for writes) must be included in Case (1a) (Equation 21).

- Case (3): finally, we cover the CAS delay case. For each of the $R^{CAS}(i), W^{CAS}(i)$ R and W requests of PE_i that add extra CAS delay, we add a CAS term to L^{CAS} and substract $tCCD$ from the WCD bound Δ. Next, consider again the example in Figure 2. Note that to cause extra delay, the interfering request must have the opposite direction compared to the first request of PE_i: otherwise, the delay would be equal to the minimum CAS separation of $tCCD$, and no extra delay would be added. Hence, we can bound $R^{CAS}(i), W^{CAS}(i)$ based on the total number of W and R interfering requests that can cause CAS delay, respectively:

$$R^{CAS}(i) \leq W^{Conf,c} + W^{Reorder,o} + W^{InterB}_{CAS} \tag{26}$$
$$W^{CAS}(i) \leq R^{Conf,c} + R^{Reorder,o} + R^{InterB}_{CAS} \tag{27}$$

The cumulative number of self-interfering requests (either R only, W only, or either R or W) can then be bounded based on the number of critical requests of PE_i:

$$R^{Conf}(i) + W^{Conf}(i) + N^{ACT}(i) + R^{CAS}(i) + W^{CAS}(i) + N^{None}(i)$$
$$\leq R^c(i) + R^o(i) + (1 - wb) \cdot (W^c(i) + W^o(i)) - 1 \tag{28}$$
$$R^{Conf}(i) + R^{CAS}(i) \leq R^c(i) + R^o(i) \tag{29}$$
$$W^{Conf}(i) + W^{CAS}(i) \leq (1 - wb) \cdot W^c(i) + (1 - wb) \cdot W^o(i) \tag{30}$$

Note that we subtract 1 in Equation 28 because the last request of PE_i cannot cause self-interference to another request of PE_i.

Finally, we shall use L^{self} to denote the sum of the self-delay in the original schedule that must be subtracted from the WCD Δ. We obtain:

$$L^{self} = (R^{Conf}(i) + W^{Conf}(i) + N^{ACT,b}(i) + R^{CAS}(i) + W^{CAS}(i)) \cdot tCCD + N^{ACT,a}(i) \cdot tRRD. \tag{31}$$

4.3 Conflict delay L^{Conf}

Based on Sections 4.1, 4.2, the total number of requests causing conflict delay is bounded by x^{Conf} intra-bank requests; plus $R^{Conf}(i) + W^{Conf}(i)$ self-interference requests; plus N^{WB} write requests if $wb = 1$. Based on Figure 1a, the conflict delay for a pair of successive requests can either be the larger $tRCD + tWL + tB + tWR + tRP$ or the smaller $tRAS + tRP$. Hence, we use variable $x^{Conf,W}$ to denote the number of conflicts of the first type, which require the first request in the pair to be a (open or close) write. We can then bound $x^{Conf,W}$ based on both the number of conflict delays $x^{Conf} + R^{Conf}(i) + W^{Conf}(i) + wb \cdot N^{WB}$; and the number of write requests that can trigger a conflict delay, which includes all write intra-bank requests $W^{Conf,c} + W^{Reorder,o}$, the write self-interference requests $W^{Conf}(i)$, and the write batching requests N^{WB}. This yields the following expressions:

$$L^{Conf} \leq x^{Conf,W} \cdot (tRCD + tWL + tB + tWR + tRP)$$
$$+ (x^{Conf} + R^{Conf}(i) + W^{Conf}(i) + wb \cdot N^{WB} - x^{Conf,W}) \cdot (tRAS + tRP) \tag{32}$$

$$x^{Conf,W} \leq x^{Conf} + R^{Conf}(i) + W^{Conf}(i) + wb \cdot N^{WB} \tag{33}$$
$$x^{Conf,W} \leq W^{Conf,c} + W^{Reorder,o} + W^{Conf}(i) + wb \cdot N^{WB} \tag{34}$$

4.4 L^{ACT} and L^{CAS} delays

To compute the maximum PRE and ACT delays, we make use of the following observation:

$\triangleright$ Observation 4. Since for modern memory devices (e.g. DDR3/4), the value of inter-bank constraints $tRRD$ and $tCCD$ is at least 4, no more than one ACT and one CAS command can be issued every 4 cycles. Since furthermore the command priority is CAS $>$ ACT $>$ PRE, it follows that every PRE command can suffer at most 2 cycles of command bus conflict, and every ACT command at most 1 cycle. CAS commands do not suffer bus conflicts.

Based on Observation 4, a PRE command can delay another PRE command by at most 3 cycles: one for the PRE command itself, and two more due to bus conflicts. For the case of ACT delay, we need to consider the $tRRD$ and $tFAW$ inter-bank ACT constraints. Note that $tRRD > 3$; hence, ACT delay is always greater than PRE delay as previously noticed. Since the $tFAW$ constraints is applied every 4 consecutive ACT, a valid upper bound to L^{ACT} can be constructed by multiplying the number of ACT delay terms, with is equal to $N_{ACT,c}^{InterB,c} + N^{ACT}(i)$, by the maximum of $tRRD$ and $tFAW/4$, then adding 1 to account for bus conflicts:

$$L^{ACT} \leq (N_{ACT,c}^{InterB,c} + N^{ACT}(i)) \cdot \big(\max(tRRD, tFAW/4) + 1\big) \tag{35}$$

Next, we discuss the CAS delay L^{CAS}. Based on Sections 4.1, 4.2, the total number of requests that cause CAS delay is $x^{CAS} + R^{CAS}(i) + W^{CAS}(i) + R_{CAS}^{InterB} + W_{CAS}^{InterB}$. The inter-bank CAS constraint between a pair of requests depends on the direction of the requests themselves: for a W followed by a R, $tWL + tB + tWTR$; for a R followed by a W, $tRTW$; and for two requests of the same direction, $tCCD$. Therefore, let variables $x^{CAS,WR}$ and $x^{CAS,RW}$ to indicate the number of W-to-R and R-to-W pairs. We then have:

$$\begin{aligned} L^{CAS} &\leq x^{CAS,WR} \cdot (tWL + tB + tWTR) + x^{CAS,RW} \cdot tRTW \\ &+ (x^{CAS} + R^{CAS}(i) + W^{CAS}(i) + R_{CAS}^{InterB} + W_{CAS}^{InterB} - x^{CAS,WR} - x^{CAS,RW}) \cdot tCCD \end{aligned} \tag{36}$$

To bound $x^{CAS,WR}$ and $x^{CAS,RW}$, we determine the maximum number of R and W requests for the first and second request in each pair. We note that interfering requests can be either; interfered critical requests of PE_i can only be the latter; and self-interfering requests of PE_i can only be the former. This yields:

$$R^{CAS,first} = R^{CAS}(i) + R^{Conf,c} + R^{Reorder,o} + R_{CAS}^{InterB} \tag{37}$$
$$R^{CAS,second} = R^{c}(i) + R^{o}(i) + R^{Conf,c} + R^{Reorder,o} + R_{CAS}^{InterB} \tag{38}$$
$$W^{CAS,first} = W^{CAS}(i) + W^{Conf,c} + W^{Reorder,o} + W_{CAS}^{InterB} \tag{39}$$
$$W^{CAS,second} = (1 - wb) \cdot (W^{c}(i) + W^{o}(i)) + W^{Conf,c} + W^{Reorder,o} + W_{CAS}^{InterB} \tag{40}$$
$$x^{CAS,WR} \leq W^{CAS,first} \wedge x^{CAS,WR} \leq R^{CAS,second} \tag{41}$$
$$x^{CAS,RW} \leq R^{CAS,first} \wedge x^{CAS,RW} \leq W^{CAS,second} \tag{42}$$
$$x^{CAS,WR} + x^{CAS,RW} \leq x^{CAS} + R^{CAS}(i) + W^{CAS}(i) + R_{CAS}^{InterB} + W_{CAS}^{InterB} \tag{43}$$

Total Cumulative Delay Bound. Finally, Δ is simply computed based on the sum of all terms computed so far:

$$\Delta = L^{Conf} + L^{ACT} + L^{CAS} - L^{self} \tag{44}$$

5 Interference Computation

Based on Equation 44, in Section 4 we have determined a bound Δ on the cumulative WCD suffered by requests of PE_i, assuming that the total number of requests per interfering component is known. We now seek to determine how many requests of each interfering PE contribute to each component. To this end, as shown in Table 3, for each interfering component we define a new set of variables with index (p) to represent the number of requests for that component that belong to PE_p. The total number of requests for each intra- and inter-component is then equal to the sum over all cores: $\mathcal{V} = \sum_{\forall p \neq i} \mathcal{V}(p)$, where $\mathcal{V}$ is either $R^{Conf,c}$, $W^{Conf,c}$, $R^{Reorder,o}$, $W^{Reorder,o}$, $R_c^{InterB,c}$, $R_c^{InterB,o}$, $W_c^{InterB,c}$, $W_c^{InterB,o}$, R_o^{InterB}, or W_o^{InterB}.

We now proceed as follows. In Section 5.1, we first bound the number of per-PE interfering requests based on the total number of requests generated by each PE_p (job-driven bound). In Sections 5.2–5.5, we then bound the per-PE interfering requests based on the platform instance (request-driven bound). Note that for the write batching component, in Section 5.5 we will need to distinguish among three sets of requests for each PE, based on when the requests are generated: $W^{btch}(p)$, $W^{before}(p)$ and $W^{after}(p)$. Furthermore, the write batching component includes requests of PE_i itself. Hence, the write batching component is bounded as follows:

$$W^{WB} = \sum_{\forall p} \big(W^{btch}(p) + W^{before}(p) + W^{after}(p)\big) \tag{45}$$

Finally, we show how to compute the WCD bound by solving a LP problem in Section 5.6.

5.1 Job-Driven Bounds

Recall from Section 3 that $R^o(p), R^c(p), W^o(p), W^c(p)$ denote the total number of R/W open/close requests for PE_p. We thus have:

$$R^{Conf,c}(p) + R_c^{InterB,c}(p) \leq R^c(p) \tag{46}$$

$$W^{Conf,c}(p) + W_c^{InterB,c}(p) \leq W^c(p) \tag{47}$$

$$R_c^{InterB,o}(p) + R^{Reorder,o}(p) \leq R^o(p) \tag{48}$$

$$W_c^{InterB,o}(p) + W^{Reorder,o}(p) \leq W^o(p) \tag{49}$$

$$R^{Conf,c}(p) + R_c^{InterB,c}(p) + R_c^{InterB,o}(p) + R^{Reorder,o}(p) + R_o^{InterB}(p) \leq R^c(p) + R^o(p) \tag{50}$$

$$W^{Conf,c}(p) + W_c^{InterB,c}(p) + W_c^{InterB,o}(p) + W^{Reorder,o}(p) + W_o^{InterB}(p) \leq W^c(p) + W^o(p) \tag{51}$$

$$W^{btch}(p) + W^{before}(p) + W^{after}(p) \leq W^c(p) \tag{52}$$

Equations 46-49 bound the number of requests of each type (open/close) and direction (R/W). Equations 50 bounds the number of read requests over all components; note that read inter-bank-open requests R_o^{InterB} can be either open or close. Similarly, Equation 51 bounds the write requests used in delay components when $wb = 0$; while Equation 52 bounds the write requests used in the write-batching delay for $wb = 1$.

5.2 Request-Driven Bounds: Conflict Requests

We introduce a constant n_p^{Conf} to denote the maximum number of conflict requests of PE_p that can interfere with one critical request of PE_i. Since conflict requests target the same bank as the critical request, n_p^{Conf} is zero if PE_p does not share any bank with PE_i. Otherwise, since conflict requests arrive before a critical request, we can set n_p^{Conf} equal to the maximum number of outstanding requests of PE_p, which is 1 if PE_p is in-order, and PR if out-of-order. Hence:

$$n_p^{Conf} = \begin{cases} \text{if } cr\text{:} \begin{cases} 0 & \text{if } PartAll \text{ or } PartCr \\ 1 & \text{if } NoPart \text{ and } (\textit{IOCr} \text{ or } \textit{IO}) \\ PR & \text{if } NoPart \text{ and } \textit{OOO} \end{cases} \\ \text{if } ncr\text{:} \begin{cases} 0 & \text{if } PartAll \\ 1 & \text{if pr or } ((NoPart \text{ or } PartCr) \text{ and } \textit{IO}) \\ PR & \text{if } (NoPart \text{ or } PartCr) \text{ and } (\textit{OOO} \text{ or } \textit{IOCr}) \end{cases} \end{cases} \tag{53}$$

Since conflict interfered requests must be close, the number of requests from core under analysis is bounded by $R^c(i) + (1 - wb) \cdot W^c(i)$, which yields the following constraint:

$$\forall p \neq i : R^{Conf,c}(p) + W^{Conf,c}(p) \leq n_p^{conf} \cdot (R^c(i) + (1 - wb) \cdot W^c(i)) \tag{54}$$

Finally, if $pr = 1$, at most one outstanding request of non-critical PEs can interfere with a critical request; hence we also obtain:

$$\text{if } pr\text{:} \sum_{\forall p \neq i, p \text{ ncr}} \left(R^{Conf,c}(p) + W^{Conf,c}(p) \right) \leq (R^c(i) + (1 - wb) \cdot W^c(i)) \tag{55}$$

5.3 Request-Driven Bounds: Reorder Requests

Reorder requests target the same bank as requests of PE_i. Hence, if PE_p and PE_i do not share any bank, the number of reorder requests of PE_p is zero. Similarly, if $pr = 1$, requests of non-critical PEs cannot be reordered ahead of critical requests, since PE_p has higher priority. Accordingly:

$$\text{if } PartAll \text{ or } PartCr, \forall p \neq i, p \text{ cr} : R^{Reorder,o}(p) = W^{Reorder,o}(p) = 0 \tag{56}$$

$$\text{if } PartAll \text{ or } pr, \forall p \neq i, p \text{ ncr} : R^{Reorder,o}(p) = W^{Reorder,o}(p) = 0 \tag{57}$$

Furthermore, if $thr = 1$, by definition no more than N_{thr} requests can be reordered ahead of each close request of PE_i. Hence it also holds:

$$\text{if } thr: R^{Reorder,o} + W^{Reorder,o} \le N_{thr} \cdot \big(R^c(i) + (1 - wb) \cdot W^c(i)\big) \tag{58}$$

5.4 Request-driven bounds: inter-bank requests

Let $N_{reqs,c}$, $N_{reqs,o}$ be the number of requests that can be delayed by inter-bank-close and inter-bank-open requests, as introduced in Section 4.1:

$$N_{reqs,c} = R^c(i) + (1 - wb) \cdot W^c(i) + R^{Conf,c} + W^{Conf,c} \tag{59}$$

$$N_{reqs,o} = R^o(i) + (1 - wb) \cdot W^o(i) + R^{Reorder,o} + W^{Reorder,o} \tag{60}$$

We first bound the number of requests generated by PE_p that can interfere on each of the $N_{reqs,c}$ close requests. Due to the assumption of RR arbitration among banks, the number of interfering inter-bank requests is limited by the number of banks in case where inter-bank reordering is not allowed ($breorder = 0$) or write batching is deployed ($wb = 1$), since it cancels the effects of inter-bank reordering [14]. Since PE_p can only access NB_p banks by definition, it must hold:

if ($wb = 1$ or $breorder = 0$):

$$\forall p \neq i: R_c^{InterB,o}(p) + R_c^{InterB,c}(p) + W_c^{InterB,o}(p) + W_c^{InterB,c}(p) \le NB_p \cdot N_{reqs,c} \tag{61}$$

Similarly, we can bound the interference caused by all critical (other than PE_i) and non-critical PEs based on the total number of banks they can access. Note that by definition, inter-bank interfering requests target a different bank than the request they interfere upon. Hence, inter-bank requests of critical PEs can target $N_{cr} - 1$ banks, while inter-bank requests of any PEs can target $N_B - 1$ banks:

if ($wb = 1$ or $breorder = 0$):

$$\sum_{\forall p \neq i, p \text{ cr}} \Big(R_c^{InterB,o}(p) + R_c^{InterB,c}(p) + W_c^{InterB,o}(p) + W_c^{InterB,c}(p)\Big) \le (N_{cr} - 1) \cdot N_{reqs,c} \tag{62}$$

$$\text{if } (wb = 1 \text{ or } breorder = 0)\text{:} \Big(R_c^{InterB,o} + R_c^{InterB,c} + W_c^{InterB,o} + W_c^{InterB,c}\Big) \le (N_B - 1) \cdot N_{reqs,c} \tag{63}$$

Finally, similarly to the constraint for conflict requests in Equation 55, if the MC uses a priority scheme, then the number of interfering requests from all non-critical PEs is in worst-case one for each of the N_{reqs} interfered requests:

if pr and ($wb = 1$ or $breorder = 0$):

$$\sum_{\forall p \neq i, p \text{ ncr}} \Big(R_c^{InterB,o}(p) + R_c^{InterB,c}(p) + W_c^{InterB,o}(p) + W_c^{InterB,c}(p)\Big) \le N_{reqs,c} \tag{64}$$

We now consider inter-bank-open requests. All derived constraints depend on the RR arbitration and bank assignment; hence, Equations 61-64 also apply to the number of interfering inter-bank-open requests $R_o^{InterB}(p) + W_o^{InterB}(p)$, except that we consider $N_{reqs,o}$ in place of $N_{reqs,c}$.

5.5 Request-driven bounds: write-batching requests

If $wb = 1$, the $R^o(i) + R^c(i)$ critical read requests of PE_i can suffer interference from write batches created by writes of either PE_i or other PEs. Since the MC gives priority to read requests over write batches, in the worst case a critical R request can be delayed by a single batch of W_{btch} write requests started before the R arrives. Hence, if we let $W^{btch}(p)$ to denote the number of interfering write requests of PE_p that arrive while no critical request is active (arrived but not completed), we have:

$$\sum_{\forall p} W^{btch}(p) \le W_{btch} \cdot (R^o(i) + R^c(i)) \tag{65}$$

However, after a critical request arrives but before it completes, further writes that arrive in the system may fill the write buffer, forcing additional batches to be processed. Therefore, we next consider the number of interfering write requests that arrive while a critical request is active. Recall that the system has a write-back write-allocate last-level cache. Accordingly, a write request can only be generated in conjunction with a read request; hence, we will reason about the maximum number of read requests that can be generated while a critical request is active. In particular, we use W^{after} to denote the number of write requests corresponding to reads that arrive while a critical request is active, and are executed after the critical request (but their corresponding writes can be executed before that critical request due to the batching scheme); and W^{before} to denote the number of write requests corresponding to read requests that arrive while a critical request is active, and are executed before it. We start by bounding W^{after}. Similar to the conflict interference in Section 5.2, we introduce a constant n_p^{after} to denote the maximum number of requests that can arrive while the critical request is active and are executed after it. Hence, it can be computed by Equation 66, and $W^{after}(p)$ can be accordingly computed by Equation 67.

$$n_p^{after} = \begin{cases} 1 & \text{if } IO \text{ or } (IOCr \text{ and } p\ cr) \\ PR & \text{if } OOO \text{ or } (IOCr \text{ and } p\ ncr) \end{cases} \tag{66}$$

$$\forall p \neq i : W^{after}(p) \leq n_p^{after} \cdot (R^o(i) + R^c(i)) \tag{67}$$

We now consider the W^{before} requests. First, if $pr = 1$, each critical request can suffer interference from a maximum of one W^{before} request from all the non-critical PEs, therefore:

$$\text{if pr: } \sum_{\forall p, p \text{ ncr}} W^{before}(p) \leq R^o(i) + R^c(i) \tag{68}$$

Second, if interfering PE_p does not share banks with PE_i, then $W^{before}(p)$ can be only due to the inter-bank RR arbitration among banks. Recall that PE_p is assigned NB_p banks, the total number of banks assigned to critical cores other than the bank targeted by the request under analysis is $N_{cr} - 1$, and the total number of banks assigned to all cores other than the bank targeted by the request under analysis is $NB - 1$. As a result, the following three conditions hold from the RR arbitration:

$$\forall p \neq i : \text{if } (PartAll) \text{ or } (PartCr \text{ and } p \text{ is cr}): W^{before}(p) \leq NB_p \cdot (R^o(i) + R^c(i)) \tag{69}$$

$$\text{if } PartAll \text{ or } PartCr\text{: } \sum_{\forall p \neq i \wedge p \text{ cr}} W^{before}(p) \leq (N_{cr} - 1) \cdot (R^o(i) + R^c(i)) \tag{70}$$

$$\text{if } PartAll\text{: } \sum_{\forall p \neq i} W^{before}(p) \leq (NB - 1) \cdot (R^o(i) + R^c(i)) \tag{71}$$

Finally, if no partitioning is deployed, we also have the FR-FCFS reordering. Thus, each request from the core under analysis can be interfered by N_{thr} (if threshold is deployed) due to intra-bank FR-FCFS reordering, while each of these requests can also be delayed by $NB - 1$ requests from RR inter-bank arbitration. Additional $NB - 1$ requests can interfere with the request under analysis itself. This gives a total of $(N_{thr} + 1) \cdot (NB - 1)$:

$$\text{if } thr = 1\text{: } \sum_{\forall p \neq i} W^{before}(p) \leq (N_{thr} + 1) \cdot (NB - 1) \cdot (R^o(i) + R^c(i)) \tag{72}$$

5.6 Optimization Problem

Consider the variables in Table 3; by definition, numbers of requests and constraints are positive integers, and the same holds for delay terms since we measure them in clock cycles. Furthermore,

(a) Benchmarks.

High				Low			
BM	#Reads	#Writes	total	BM	#Reads	#Writes	total
matrix	280000	38428	318428	rspeed	2000	482	2479
a2time	166000	21751	187751	pntrch	2000	479	2478
aifftr	101000	77234	178234	basefp	2000	478	2478

(b) Configuration parameters.

P	4	Pcr	2	$Pncr$	2
N_{thr}	8	W_{btch}	16	PR	4
N_B	8				
N_{Bp}	8 ($noPart$), 2 ($PartAll$), or 4/8 ($PartCr$ and p is cr/ncr)				
N_{Bcr}	8 ($noPart$ or $PartCr$), 4 ($PartAll$)				
N_{Bncr}	8 ($noPart$ or $PartCr$), 4 ($PartAll$)				

Table 4 Evaluation Setup.

all constraints introduced in Sections 3-5 are linear in such variables. Hence, we could compute an upper bound on Δ by solving an integer LP problem, with the optimization objective of maximizing Equation 44. In practice, we consider a linear relaxation of the same problem, where all variables are treated as reals; by construction, the resulting LP problem still yields a valid bound on Δ. The number of variables and constraints is proportional to P; hence the complexity of solving the linear programming problem is polynomial in the number of PEs.

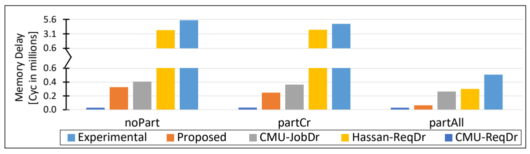
6 Evaluation

Simulation Environment. We use MacSim [19], a heterogeneous multi-processor simulator integrated with DRAMSim2 [32]. MacSim models x86 architecture and supports IO and OOO PEs. It also allows the configuration of the maximum number of pending requests through managing the number of entries in MSHR registers. MacSim has a frontend that includes the virtual-to-physical mapping. This enables us to implement partitioning without running a complete OS. We implement the three partitioning schemes discussed in Section 3. DRAMSim2 [28] is a cycle-accurate DRAM simulator, which we extend to also support priority assignment amongst PEs as well as write batching. We implement the optimization framework in Matlab and it finishes within few seconds for all experiments using a machine with a quad-core i7 processor and 8GB DRAM and is running Linux.

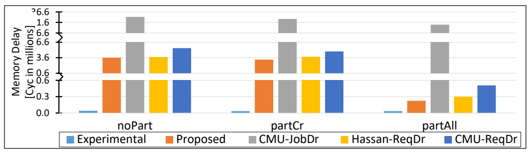
Benchmarks. We use benchmarks from the EEMBC-auto suite [29], which include representative applications from the embedded automotive domain. Recall from Section 5 that the maximum number of interfering requests, and thus the memory delay incurred by the PE under analysis, depend both on the number of memory requests initiated by this PE as well as the number of requests issued by the competing PEs. Therefore, we construct experiments that capture different scenarios. Towards doing so, we classify the used benchmarks into two categories: *High* and *Low* as shown in Table 5a. The *High* (*Low*) benchmarks are those that issue a large (a small) number of memory requests.

Experiments Setup. We compare the proposed analysis with five state-of-the-art approaches; two of them are job-driven analyses: CMU-JobDr [20] and YUN-JobDr [37]; and three are request-driven analyses: CMU-ReqDr [20], YUN-ReqDr [37], and Hassan-ReqDr [14]. We also compare against the experimental WCD observed from the simulator, denoted as Experimental. We use a system composed of four cores: two in-order critical and the other are OOO non-critical ones. Table 5b lists all values of used parameters. Since both CMU and YUN do not support mixed criticalities, for their analysis all tasks are considered to have the same criticality. In addition, since they also cover only certain system configurations, we compare against these solutions under all configurations they support. To evaluate each analysis under different interference scenarios, we run different experiments using different mix of the benchmarks in Table 5a. Namely, we evaluate with four different scenarios: Low-Low, Low-High, High-Low, and High-High, where the first term refers to the task under analysis and the second term refers to interfering tasks. For instance, in a Low-High scenario, the interfered task is chosen to be the *rspeed* benchmark, which is in the Low category in Table 5a, while the interfering tasks are *matrix, a2time*, and *aifftr* from the High category.

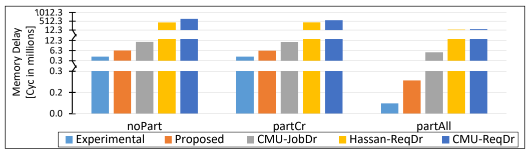
1) System Configurations Supported By CMU. Both job- and request-driven CMU's analyses [20] can be applied to platform instances with in-order pipelines, all cores have same priority, and the memory controller deploys a FR-FCFS threshold but does not deploy write batching. However,



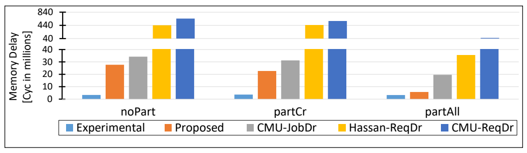
(a) Low-Low.



(b) Low-High.

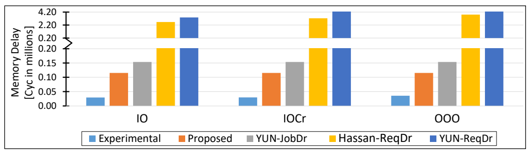


(c) High-Low.

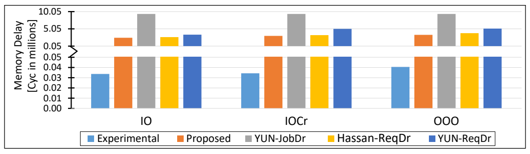


(d) High-High.

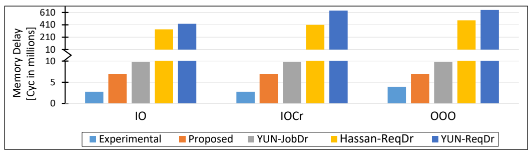
Figure 3 Results for configurations that are considered by CMU [20].



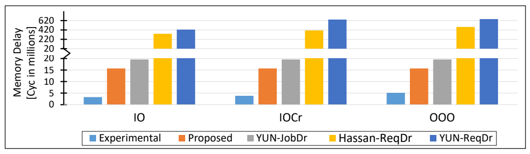
(a) Low-Low.



(b) Low-High.



(c) High-Low.

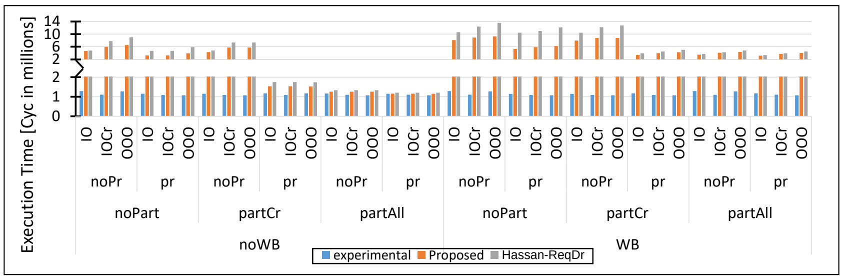


(d) High-High.

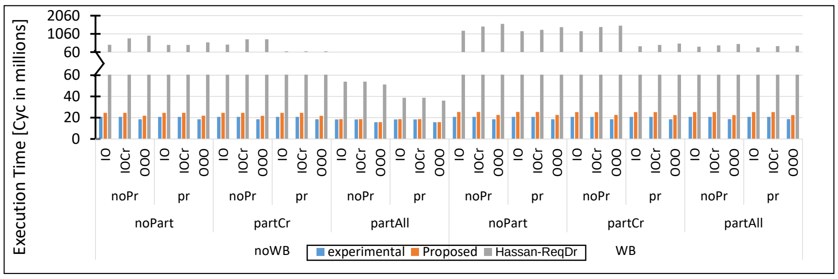
Figure 4 Results for configurations that are considered by YUN [37].

the analysis can cover different bank partitioning scenarios. Accordingly, these are the instances we used in our experiments when comparing against those approaches. Figure 3 delineates the total memory delay suffered for different considered approaches under different interference scenarios. We make the following observations. 1) As aforestated, the request- and job-driven approaches are incomparable: neither approach is better than the other under all scenarios. Although CMU-JobDr provides tighter delay bounds than request-driven ones (both CMU-ReqDr and Hassan-ReqDr) in Figures 3a, 3c, and 3d, the request driven approaches have better bounds for the Low-High scenario in Figure 3b. Since request-driven analysis considers only the number of requests from the core under analysis, when this number is relatively small compared to the total number of competing requests, this analysis provides tighter bounds. This is the case for the Low-High scenario. For other scenarios, the number of requests of the core under analysis is relatively large and leads to the larger delay bounds of the request-driven analyses. 2) The proposed analysis provides the tightest bounds across all scenarios. For the Low-High scenario, Proposed provides up to a 34% tighter bound than the second best approach, which is Hassan-ReqDr ($PartAll$ in Figure 3b). For all other scenarios, Proposed provides at least 24% ($noPart$ in Figure 3a) and up to 22.6× ($PartAll$ in Figure 3c) tighter bound than CMU-JobDr, which is the second best approach in all these scenarios.

2) System Configurations Supported By YUN. The platform instances covered in [37] (for both YUN-JobDr and YUN-ReqDr) are partitioning banks across cores ($PartAll$), all cores have same priority, and the memory controller deploys both FR-FCFS threshold and write batching. Although [37] only evaluates OOO cores, we find that the analysis is extensible to any core pipeline (by managing maximum number of pending requests from each core). Therefore, we experiment



(a) Low-High.



(b) High-Low.

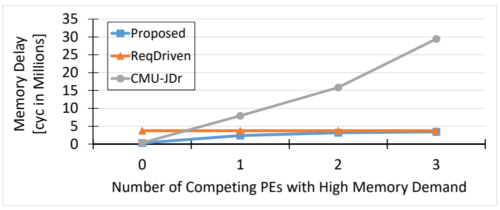
Figure 5 Comparison of the total response time with Hassan-ReqDr [14] for all configurations.

with different pipelining configurations and show the results in Figure 4. 1) YUN-JobDr provides tighter bounds than request driven analyses (YUN-ReqDr and Hassan-ReqDr) for all interference scenarios except for Low-High. 2) The Proposed approach still provides the tightest bounds across all scenarios. Proposed provides at least 25% (IO in Figure 4d) and up to 42% (OOO in Figure 4c) better bounds compared to YUN-JobDr (next best approach) in the Low-Low, High-Low, and High-High scenarios. In the Low-High scenario in Figure 4b, it provides up to 15% better bounds than the second best option of Hassan-ReqDr.

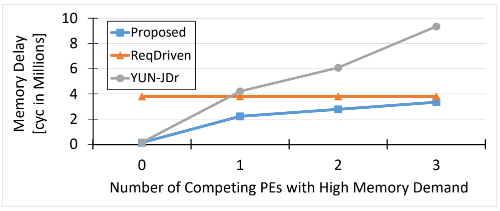
3) System Configurations Supported By Hassan-ReqDr. We now compare the proposed analysis with the Hassan-ReqDr analysis for all the supported platform instances discussed in Section 3. We compared both approaches for all interference scenarios; however, for space considerations, in Figure 5, we only show results for the Low-High and High-Low scenarios, which best illustrate the main lessons we want to highlight. 1) Proposed provides tighter bound than Hassan-ReqDr for all platform instances. 2) For the Low-High scenario (Figure 5a), bounds of both solutions are very close. This is because for his scenario, the number of requests from the core under analysis is relatively small compared to the total number of competing requests. Therefore, request-driven analysis (Sections 5.2–5.5) provides tighter bounds than job-driven analysis (Section 5.1). However, Proposed still outperforms Hassan-ReqDr thanks to the leveraged knowledge about the running tasks. As a result, in Figure 5a, it provides up to 98% (instance WB-$noPart$-pr-IO) and 24% on average tighter bounds across all platform instances. 3) For the High-Low scenario (Figure 5b), we observe a large gap between Proposed and Hassan-ReqDr. In Figure 5b, Proposed provides up to 71× and 18× on average tighter bound across all configurations. Two main reasons are behind such significant gap: no partitioning ($noPart$), and write batching (WB). Both features, if considered, forces Hassan-ReqDr to consider a pathological worst-case scenario that is overly pessimistic. For $noPart$, Hassan-ReqDr considers every request of the core under analysis to have the worst-case intra-bank (conflict and reorder) interference from competing cores. Similarly for WB, Hassan-ReqDr assumes that every read request will suffer a worst-case write batching delay even if there are not enough number of competing requests to cause this much interference for every single request

Partitioning	noWB	WB	Partitioning	noWB	WB	Partitioning	noWB	WB
$PartAll$	15706330	24540906	$PartCr$	24560480	24540906	$noPart$	24560480	24540906

Table 6 Proposed's WCD (in cycles) for the 63 platform instances that are declared unbounded under Hassan-ReqDr (satisfying condition in Equation 73). We found WCD to depend only on WB and paritioning values. Values are for the Low-High interference scenario.



(a) A comparison for a configuration with no WB, no partitioning, IO pipeline, and no priority.



(b) A comparison for a configuration with WB, $PartAll$, OOO pipeline, and no priority.

Figure 6 Varying number of "High" competing cores.

from the core under analysis. On the other hand, by leveraging the job-driven analysis and considering the number of competing requests, Proposed provides tighter bounds.

4) Configurations Unbounded by Hassan-ReqDr. In [14], the authors considered 144 platform instances. The Hassan-ReqDr analysis bounded 81 of them, while 63 instances were proven to be unbounded under this analysis. We identify those 63 instances by the following condition:

$$\left(breorder{=}1 \text{ and } wb{=}0\right) \text{ or } \left(thr{=}0 \text{ and } \big(part{=}noPart \text{ or } (part{=}PartCr \text{ and } pr{=}0)\big)\right) \tag{73}$$

Leveraging the job-driven analysis, the Proposed approach is able to bound all these cases using information about memory requests of competing tasks (Section 5.1). Table 6 shows the obtained bounds for the Low-High scenario.

5) Hybrid Analysis under different Interference Severity. To further show the benefit of the proposed hybrid analysis compared to the state-of-the-art request- and job-driven analyses, we investigate with different number of competing tasks with high memory demand (High from Table 5a). In this set of experiments, we use $rspeed$ (Low) benchmark as the one under analysis and vary the number of high competing cores. The total number of cores in the experiment is four. For instance, 2 in the x-axis of Figure 6 indicates that in addition to the core under analysis, two cores are running benchmarks from the High category, while one core is running a benchmark from the Low category. From Figure 6, it is clear that the effectiveness of the request- vs job-driven analysis is dependent on the relative ratio between the number of requests of the core under analysis and the number of requests from competing cores. On the other hand, the proposed hybrid analysis is able to achieve better bound compared to both approaches for all cases.

7 Conclusions

We propose a novel approach to bound interference delays due to contention upon accessing off-chip DRAMs in heterogeneous COTS MPSoCs. The proposed hybrid framework blends both request- and job-driven analyses to provide tighter bounds than those determined by each analysis separately and then taking the minimum of both. The framework also leverages information about the memory behavior of running task such as number of read and write requests, which are usually available from statically analyzing each task in isolation. We evaluate the proposed approach across a wide set of COTS platform instances, where it outperforms existing state-of-the-art analyses (both request- and job-driven).

References

1 Ankit Agrawal, Gerhard Fohler, Johannes Freitag, Jan Nowotsch, Sascha Uhrig, and Michael Paulitsch. Contention-aware dynamic memory bandwidth isolation with predictability in COTS multicores: An avionics case study. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2017.

2 Ankit Agrawal, Renato Mancuso, Rodolfo Pellizzoni, and Gerhard Fohler. Analysis of dynamic memory bandwidth regulation in multi-core real-time systems. In *IEEE Real-Time Systems Symposium (RTSS)*, 2018.

3 Balasubramanya Bhat and Frank Mueller. Making DRAM refresh predictable. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2010.

4 Vamsi Boppana, Sagheer Ahmad, Ilya Ganusov, Vinod Kathail, Vidya Rajagopalan, and Ralph Wittig. UltraScale+ MPSoC and FPGA families. In *IEEE Hot Chips Symposium (HCS)*, 2015.

5 Roman Bourgade, Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. Accurate analysis of memory latencies for WCET estimation. In *International Conference on Real-Time and Network Systems (RTNS)*, 2008.

6 Mauricio Calle and Ravi Ramaswami. Multi-bank scheduling to improve performance on tree accesses in a DRAM based random access memory subsystem, January 4 2005. US Patent 6,839,797.

7 Leonardo Ecco, Sebastian Tobuschat, Selma Saidi, and Rolf Ernst. A Mixed Critical Memory Controller Using Bank Privatization and Fixed Priority Scheduling. In *Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2014.

8 Georgia Giannopoulou, Nikolay Stoimenov, Pengcheng Huang, and Lothar Thiele. Scheduling of mixed-criticality applications on resource-sharing multicore systems. In *ACM International Conference on Embedded Software (EMSOFT)*, 2013.

9 Danlu Guo, Mohamed Hassan, Rodolfo Pellizzoni, and Hiren Patel. A Comparative Study of Predictable DRAM Controllers. *ACM Transaction on Embedded Computer Systems (TECS)*, 2018.

10 Danlu Guo and Rodolfo Pellizzoni. A request bundling dram controller for mixed-criticality systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017.

11 Sebastian Hahn, Michael Jacobs, and Jan Reineke. Enabling compositionality for multicore timing analysis. In *International conference on real-time networks and systems (RTNS)*, 2016.

12 Mohamed Hassan. Heterogeneous MPSoCs for Mixed Criticality Systems: Challenges and Opportunities. *IEEE Design & Test*, 2017.

13 Mohamed Hassan, Hiren Patel, and Rodolfo Pellizzoni. A Framework for Scheduling DRAM Memory Accesses for Multi-Core Mixed-time Critical Systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2015.

14 Mohamed Hassan and Rodolfo Pellizzoni. Bounding DRAM interference in COTS heterogeneous MPSoCs for mixed criticality systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2018.

15 Intel. External memory interface handbook volume 2: Design guidelines, 2017.

16 Bruce Jacob, Spencer Ng, and David Wang. *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.

17 Javier Jalle, Eduardo Quinones, Jaume Abella, Luca Fossati, Marco Zulianello, and Francisco J Cazorla. A dual-criticality memory controller (DCmc): Proposal and evaluation of a space case study. In *IEEE Real-Time Systems Symposium (RTSS)*, 2014.

18 DDR3 SDRAM JEDEC. JEDEC jesd79-3b, 2008.

19 H Kim, J Lee, N Lakshminarayana, J Lim, and T Pho. Macsim: Simulator for heterogeneous architecture, 2012.

20 Hyoseung Kim, Dionisio de Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Raj Rajkumar. Bounding memory interference delay in COTS-based multi-core systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.

21 N. Kim, B. Ward, M. Chisholm, J. Anderson, and F.D. Smith. Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning. *Real-Time Systems*, 2017.

22 Haohan Li and Sanjoy Baruah. Global mixed-criticality scheduling on multiprocessors. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2012.

23 Yonghui Li, Benny Akesson, and Kees Goossens. Dynamic Command Scheduling for Real-Time Memory Controllers. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2014.

24 Onur Mutlu and Thomas Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 146–160. IEEE, 2007.

25 Jan Nowotsch, Michael Paulitsch, Daniel Bühler, Henrik Theiling, Simon Wegener, and Michael Schmidt. Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2014.

26 Xing Pan, Yasaswini Gownivaripalli, and Frank Mueller. Tintmalloc: Reducing memory access divergence via controller-aware coloring. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2016.

27 Risat Mahmud Pathan. Schedulability analysis of mixed-criticality systems on multiprocessors. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2012.

28 Rodolfo Pellizzoni, Andreas Schranzhofer, Jian-Jia Chen, Marco Caccamo, and Lothar Thiele. Worst case delay analysis for memory interference in multicore systems. In *IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2010.

29 Jason Poovey. Characterization of the EEMBC benchmark suite. *North Carolina State University*, 2007.

30 Qualcomm. Qualcomm snapdragon 600e processor apq8064e recommended memory controller and device settings application note, 2016.

31 Scott Rixner, William J Dally, Ujval J Kapasi, Peter Mattson, and John D Owens. Memory access scheduling. In *ACM SIGARCH Computer Architecture News*, volume 28, pages 128–138. ACM, 2000.

32 Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. DRAMSim2: A cycle accurate memory system simulator. *IEEE Computer Architecture Letters (CAL)*, 2011.

33 Jeffrey Stuecheli, Dimitris Kaseridis, Hillery C Hunter, and Lizy K John. Elastic refresh: Techniques to mitigate refresh penalties in high density memory. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2010.

34 Prathap Kumar Valsan and Heechul Yun. MEDUSA: a predictable and high-performance DRAM controller for multicore based embedded systems. In *IEEE International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*, 2015.

35 Zheng Pei Wu, Rodolfo Pellizzoni, and Danlu Guo. A Composable Worst Case Latency Analysis for Multi-Rank DRAM Devices under Open Row Policy. *Real-Time Systems*, 2016.

36 Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.

37 Heechul Yun, Rodolfo Pellizzon, and Prathap Kumar Valsan. Parallelism-aware memory interference delay analysis for COTS multicore systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2015.

38 Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013.

39 Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memory bandwidth management for efficient performance isolation in multicore platforms. *IEEE Transactions on Computers (TC)*, 2016.